Machine Learning

Sara Pohland

Created: January 29, 2021
Last Modified: December 14, 2023

Contents

Part I

Overview of Machine Learning Methods

Chapter 1

Introduction to Machine Learning

1.1 Machine Learning Abstractions

There are four abstractions of machine learning: data/application, model, optimization problem, and optimization algorithm. In these notes, we focus primarily on models and optimization problems. However, optimization problems are covered much more thoroughly in the convex optimization notes. The convex optimization notes also contain information on optimization algorithms.

1.1.1 Data & Application

The first component of machine learning is the data set, which is composed of data samples with various features. Data is considered **labeled** if there is some value or class associated with each data sample, and it is considered **unlabeled** if we are not given any label associated with each data sample. A set of data may be used for various different applications. The type of machine learning problem we perform depends heavily on the type of data we are given. Below are the general classes of machine learning problems:

1. **Supervised learning** – Learning problems involving labeled data.
   (a) **Classification** – If the labels corresponding to each data sample are categorical, then we are interested in performing classification to predict the class of unseen data.
   (b) **Regression** – If the labels corresponding to each data sample are quantitative, then we are interested in performing regression to estimate the associated value of unseen data.
2. **Unsupervised learning** – Learning problems involving unlabeled data.

(a) **Clustering** – If we are want to determine the similarity among unlabeled data samples, then we are in interested clustering data.

(b) **Dimensionality reduction** – If we are want to determine the relative positioning of unlabeled data samples, then we are interested in dimensionality reduction.

1.1.2 Model

Given a data set for a desired application, we want to generate a model of the data. In supervised learning problems, we often use some kind of decision function to predict the class or value associated with the data samples. In unsupervised learning problems and some supervised learning problems, we use a model that does not have a decision function. When training a model, we generally break the data into the following three groups:

1. **Training set** – The training set is the subset of the given data (usually around 80%) that is used to train a model. The **training error** is the fraction of the training data classified incorrectly by the trained model, and **training accuracy** is the fraction of the training data classified correctly.

2. **Validation set** – The validation set is the subset of the given data (usually around 20%) that is not used to train the model and is instead used to tune hyperparameters of the model. The **validation error** is the fraction of the validation data classified incorrectly by the trained model, and the **validation accuracy** is the fraction of the data classified correctly.

3. **Test set** – The test set is the set of unseen data, which is not seen during training and is used to test the classifier. The **test error** is the fraction of the test data classified incorrectly by the trained model, and the **test accuracy** is the fraction of the test data classified correctly.

Often, we train the model multiple times on the training set with different hyperparameters and choose the parameter values that give us the lowest validation error. Machine learning models generally run into two major issues:

1. **Underfitting** – Underfitting occurs when the trained model is not able to accurately reflect the underlying distribution of the data, resulting in high training error, as well high validation/test error.

2. **Overfitting** – Overfitting occurs when training error is low, but the validation/test error deteriorates because the model does not generalize well to new examples. This occurs when our model is too sensitive to outliers or other spurious patterns in the training data.

Models are generally considered to be either low capacity or high capacity. **Low capacity** models are less complicated and tend to underfit easily, while **high capacity** models are more complicated and tend to overfit easily.

1.1.3 Optimization Problem

Given a model of the data, we generate an optimization problem, which is composed of variables, constraints, and an objective function. Optimization problems may be constrained or unconstrained, and they can be convex or non-convex. For information on these classifications of optimization problems, please reference the convex optimization notes. Some popular categories of convex optimization problems are linear programs, quadratic programs, second-order cone programs, and semidefinite programs. These convex programs, along with others, are all discussed in detail in the convex optimization notes.

1.1.4 Optimization Algorithm

Finally, we solve our optimization problem using some type of optimization algorithm. In some cases, we are able to solve an optimization problem numerically using optimality conditions. In other cases, we must use iterative techniques to solve our optimization problem. Often, we use gradient descent or Newton's method to find approximates solutions to optimization problems, but there are several other algorithms that may be employed. Optimality conditions and iterative techniques are discussed in the convex optimization notes.

Part II

Supervised Learning Techniques

Chapter 2

Introduction to Supervised Learning Techniques

2.1 Overview of Supervised Learning

In supervised learning, we are given a set of labeled data for which we want to learn a model representing the relationship among the data. Consider a data set composed of a sample of n observations with d features each. Each observation is represented as a point in d-dimensional space (i.e. $\boldsymbol{x_i} \in \mathbb{R}^d,\ i = 1, \ldots, n$) and is called a **sample point**. Each sample point has a corresponding label y_i.

2.1.1 Classification & Regression

In **classification** problems, the labels are categorical and represent the class to which a sample point belongs. For classification problems, we aim to generate a model to determine the class of unseen data. For *binary* classification problems, there are only two classes and we want to determine whether unseen data belongs to the given class. In **regression** problems, the labels are quantitative. For regression problems, we use labeled training data to generate a model that predicts the numerical value associated with unseen data.

2.2 Bias & Variance

Recall that for supervised learning problems, we seek to learn a model representing the relationship among the labeled data. Suppose there is some unknown function g, which represents the underlying distribution that relates a sample point, $\boldsymbol{x_i}$, to its label, y_i. We are given n sample points, $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$, with corresponding labels, $y_1, \ldots, y_n$. The goal of supervised learning is to find a hypothesis function, h, that estimates the unknown function, g. There are two main sources of error present when searching for a hypothesis function:

1. **Bias** – Bias is error that results from the inability of the hypothesis function, h, to perfectly fit the unknown function, g. For example, this may occur if we try to fit a quadratic function g with a linear function h.

2. **Variance** – Variance is error that results from fitting random noise in the data. Even if we choose a quadratic function h to fit the quadratic function g, the functions may not be exactly the same due to noise.

2.2.1 Implications of Bias & Variance

The following are some important notes about bias and variance:

1. Models with high levels of bias and low levels of variance are more likely to underfit the data because their assumptions are too restrictive and inexpressive to fit the underlying data distribution well.

2. Models with high levels of variance and low levels of bias are more likely to overfit the data because they are prone to fitting the noise of the data, rather than the underlying distribution, due to their weak assumptions.

3. The training error reflects the bias but not the variance, while the validation and test error reflect both the bias and the variance.

4. In general, increasing the amount of training data reduces the training accuracy but increases the validation/test accuracy. For many distributions, the variance goes to zero as the number of sample points goes to infinity. If the hypothesis, h, can exactly fit the underlying distribution, g, then for many distributions, bias also goes to zero as the number of sample points goes to infinity. However, if the model cannot fit the underlying distribution well enough, then bias remains high.

5. Adding a feature with good predictive power generally reduces bias, but adding any feature increases variance. Therefore, it is generally not beneficial to add new features if they are uncorrelated with the sample labels or if they are linear combinations of already existing features. It may also not be good to add new features that are particularly noisy.

6. Noise in the training set affects bias and variance, while noise in the test set only affects an error that cannot be reduced by changing our model.

7. We cannot precisely measure bias and variance of real-world data because we do not know the underlying probability distributions. However, if we generate synthetic data from known probability distributions and generate noise from a known probability distribution, then we can compute the bias and variance of the synthetic data. This allows us to gain additional insight into our learning algorithms.

2.2.2 Feature Selection

As we stated in the previous section, adding a feature with strong predictive power reduces bias, but all features increase variance. We want to identify and remove poorly predictive features to reduce variance, which should result in less overfitting and smaller test errors. This also leads to simpler models that are more interpretable and can increase the speed of prediction on test points.

Recall that we assume each sample point has d features. To choose the true best subset of features to use, we would need to try all $2^d - 1$ nonempty subsets of features and use validation to determine which subset results in the best classifier. This would give us the best subset of features, but this method is infeasible if d is large. There are two computationally feasible heuristics we often use to determine a good subset of features that is not necessarily optimal:

1. **Forward stepwise selection** – In forward stepwise selection, we start with 0 features and repeatedly add the "best" feature until validation errors start to increase. To determine the best feature at each iteration, we train a model for each feature that is not already being used and choose the best feature via validation. In forward stepwise selection, we train $O(d^2)$ models instead of $O(2^d)$ models, which is much more feasible.

2. **Backward stepwise selection** – In backward stepwise selection, we start with all d features and repeatedly remove the "worst" feature until validation error stops decreasing. To determine the worst feature at each iteration, we train a model with each remaining feature removed and choose the feature whose removal gives the largest reduction in validation error. In backward stepwise selection, we again train $O(d^2)$ models.

Generally, forward stepwise selection is better to use if we believe that only a small portion of the entire set of features provides strong predictive power. Conversely, backward stepwise selection is better to use if we believe that a large portion of the set of features provides strong predictive power.

2.3 Receiver Operating Characteristics

2.3.1 Outcomes of Binary Classifier

In binary classification problems, data samples are either in the class of interest or not in the class. Data samples that are within a given class are referred to as positive samples, and samples that are not in the given class are referred to as negative samples. For each data sample, the binary classifier predicts whether that sample is in the class (positive) or not in the class (negative). This then leads to four possible outcomes, which are shown in figure 2.1. From these four outcomes, we define the **true positive rate**, **true negative rate**, **false positive rate**, and **false negative rate**, as shown on the following page.

		True Condition	
		In Class (Positive)	Not in Class (Negative)
Prediction	In Class (Positive)	True Positive	False Positive
	Not in Class (Negative)	False Negative	True Negative

Figure 2.1: A binary classifier has four possible outcomes: true positive, true negative, false positive, and false negative.

$$\text{True Positive Rate} = \frac{\text{Number of true positives}}{\text{Total number of positive samples}}$$

$$\text{True Negative Rate} = \frac{\text{Number of true negatives}}{\text{Total number of negative samples}}$$

$$\text{False Positive Rate} = \frac{\text{Number of false positives}}{\text{Total number of negative samples}}$$

$$\text{False Negative Rate} = \frac{\text{Number of false negatives}}{\text{Total number of positive samples}}$$

Note that the true positive rate and false negative rate sum to one, and the true negative rate and false positive rate sum to one. We also refer to the true negative rate as the **specificity** and the true positive rate as the **sensitivity**.

2.3.2 ROC Curve

Receiver operating characteristic (ROC) curves are used to evaluate a trained classifier. A ROC curve shows the true positive rate versus the false positive rate for a range of settings. Figure 2.2 displays a sample ROC curve for an arbitrary classifier. The area under the ROC curve provides a rough measure of the classifier's effectiveness. For a perfect classifier, the area under the ROC curve is one, while for a random classifier, the area is one half.

The ROC curve can be used to choose model parameters based on the relative importance of the false positive rate versus the false negative rate. If we deem false negatives worse than false positives, we would choose parameters that give us a point on the curve with higher sensitivity that is further from the x-axis. Conversely, if we deem false positives worse, we would choose parameters that give us a point on the curve with higher specificity that is closer to the y-axis.

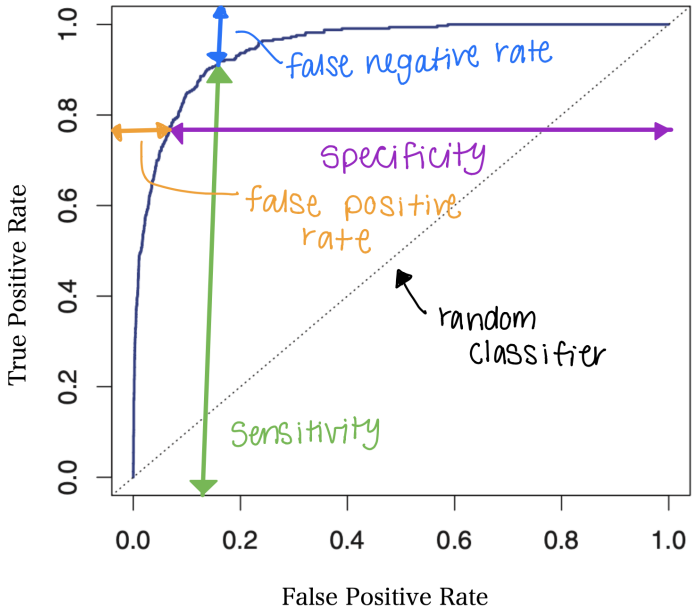

Figure 2.2: The ROC curve is a plot of the true positive rate versus the false positive rate for test sets. From the ROC curve, you can read off the true positive rate (sensitivity), the true negative rate (specificity), the false positive rate, and the false negative rate. Note that while this example is concave, the ROC curve can be concave, convex, or neither convex nor concave.

Chapter 3

Linear Classifiers

3.1 Decision Boundaries

3.1.1 General Decision Boundaries

Consider a data set of n d-dimensional sample points: $\{\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}\}$. Each sample point, $\boldsymbol{x_i} \in \mathbb{R}^d$, belongs to a class, C_i. We will assume that there are only two classes and that every point is either in class C or not in class C.

A **decision boundary** is a boundary chosen by a classifier to separate items in different classes. Many classifiers compute a **decision function**, f, which is also known as a **predictor function** or **discriminant function**. The goal of this function is to map a point in the feature space to a scalar value such that

$$\begin{cases} f(\boldsymbol{x}) > 0 & \text{if } \boldsymbol{x} \in C \\ f(\boldsymbol{x}) \leq 0 & \text{if } \boldsymbol{x} \notin C \end{cases}.$$

If the classifier uses a decision function, the decision boundary is defined as

$$\{\boldsymbol{x} \in \mathbb{R}^d : f(\boldsymbol{x}) = 0\}.$$

3.1.2 Linear Decision Boundaries

For a d-dimensional feature space, linear decision functions have the form

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + \alpha,$$

where $\boldsymbol{w} \in \mathbb{R}^d$ is a **weight vector** and $\alpha \in \mathbb{R}$ is a **bias term**. For this linear decision function, the decision boundary is the following hyperplane:

$$\mathcal{H} = \{\boldsymbol{x} \in \mathbb{R}^d : \boldsymbol{w}^T \boldsymbol{x} + \alpha = 0\}.$$

If $\boldsymbol{x}$ and $\boldsymbol{y}$ are two points on the hyperplane $\mathcal{H}$, then it is easy to show that $\boldsymbol{w}^T(\boldsymbol{y} - \boldsymbol{x}) = 0$. This tells us that $\boldsymbol{w}$ is the **normal vector** of the hyperplane

$\mathcal{H}$. The signed distance from any point $\boldsymbol{z}$ to the hyperplane $\mathcal{H}$ is

$$d = \frac{\boldsymbol{w}^T\boldsymbol{z} + \alpha}{||\boldsymbol{w}||_2}.$$

The signed distance is positive if $\boldsymbol{z}$ is on the side of the hyperplane that the normal vector, $\boldsymbol{w}$, points in. It is negative if it is on the opposite side.

3.1.3 Linearly Separable

Input data is considered **linearly separable** if there exists a hyperplane that separates all of the sample points in class C from points not in class C. Figure 3.1 depicts two sets of data: one that is linearly separable and one that is not.

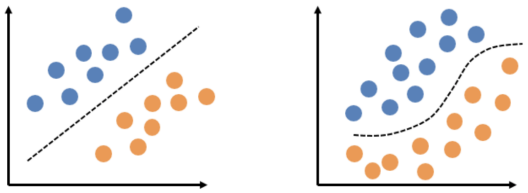

Figure 3.1: In the image on the left, there is a hyperplane that separates the data in the blue class from the data in the orange class, so the data is linearly separable. In the image on the right, the data is separable, but it cannot be separated by a hyperplane, so it is not linearly separable.

3.2 Centroid Method

There are several different linear classification methods. One linear classification method is the **centroid method**, which is performed as follows:

1. Compute the mean, $\boldsymbol{\mu_C}$, of all of the points in class C and the mean, $\boldsymbol{\mu_X}$, of all of the points not in class C (i.e. points in $\bar{C}$).

2. Define the decision function as the hyperplane that bisects the line segment connecting $\boldsymbol{\mu_C}$ and $\boldsymbol{\mu_X}$. More concretely, it is defined such that

$$f(\boldsymbol{x}) = (\boldsymbol{\mu_C} - \boldsymbol{\mu_X})^T\boldsymbol{x} - \frac{1}{2}||\boldsymbol{\mu_C} - \boldsymbol{\mu_X}||_2^2.$$

3. Classify a point $\boldsymbol{x}$ by assigning a label y such that

$$y = \begin{cases} C & \text{if } f(\boldsymbol{x}) > 0 \\ \bar{C} & \text{if } f(\boldsymbol{x}) \leq 0 \end{cases}.$$

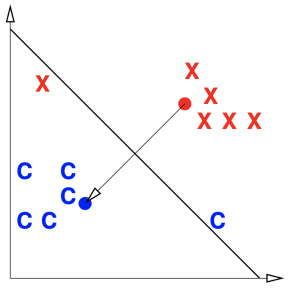

Figure 3.2: The data is composed of sample points in class C, which are marked with blue C's, and points not in class C, which are marked with red X's. The centroid method computes the mean of the data points in class C, which is indicated with a blue dot, and the mean of the data points not in class C, which is indicated with a red dot. The decision function is the hyperplane that bisects the line segment connecting the means.

Figure 3.2 provides an example of the centroid method. Notice that, in this example, the centroid method does not perfectly classify the sample points, despite the fact that the given data is linearly separable. This error is due to outliers in the data. In general, the centroid method is not guaranteed to find a decision boundary that completely separates linearly separable data.

3.3 Perceptron Algorithm

Another linear classification method is the **perceptron algorithm**. In contrast to the centroid algorithm, the perceptron algorithm always correctly classifies linearly separable data. Consider a data set composed of n d-dimensional sample points, $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$. Assume $\boldsymbol{x_i}$ has corresponding label y_i defined such that

$$y_i = \begin{cases} 1 & \text{if } \boldsymbol{x_i} \in C \\ -1 & \text{if } \boldsymbol{x_i} \notin C \end{cases}.$$

3.3.1 Perceptron Algorithm Without Bias

We will first discuss the perceptron algorithm with no bias term, assuming that the decision boundary passes through the origin. The goal of the perecptron algorithm with no bias is to find the weight vector $\boldsymbol{w}$ such that

$$\begin{cases} \boldsymbol{w}^T \boldsymbol{x_i} > 0 & \text{if } y_i = 1 \\ \boldsymbol{w}^T \boldsymbol{x_i} \leq 0 & \text{if } y_i = -1 \end{cases}.$$

We can express these two constraints on the weight vector as a single constraint:

$$y_i(\boldsymbol{w}^T \boldsymbol{x_i}) \geq 0.$$

Let z be the prediction for some point $\boldsymbol{x}$ in the feature space, which has the true label y. For the perceptron algorithm, we define the following loss function:

$$L(z, y) := \begin{cases} 0 & \text{if } yz \geq 0 \\ -yz & \text{otherwise} \end{cases}.$$

If $\boldsymbol{x}$ is correctly labeled by the classifier, then z and y would have the same signs, so yz would be positive and we would not incur any loss. If $\boldsymbol{x}$ is incorrectly labeled by the classifier, then z and y would have opposite signs, so yz would be negative and we would incur a loss of $-yz$, which is a positive value.

For a sample point $\boldsymbol{x_i}$ with true label y_i, the perceptron algorithm with no bias predicts the label $z_i = \boldsymbol{w}^T \boldsymbol{x_i}$. For this algorithm, we then define the risk function as the sum of the losses for the predictions of all the sample points:

$$R(\boldsymbol{w}) := \sum_{i=1}^{n} L(\boldsymbol{w}^T \boldsymbol{x_i}, y_i).$$

Given a weight vector $\boldsymbol{w}$, let $\mathcal{V}$ be the set of misclassified points, which is defined as $\mathcal{V} := \{i \in \{1, \ldots, n\} : y_i(\boldsymbol{w}^T \boldsymbol{x_i}) < 0\}$. We can now express the risk as

$$R(\boldsymbol{w}) = \sum_{i \in \mathcal{V}} -y_i(\boldsymbol{w}^T \boldsymbol{x_i}).$$

With this risk function, the optimal weight vector $\hat{\boldsymbol{w}}$ is defined as

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w} \in \mathbb{R}^d} R(\boldsymbol{w}).$$

Note that the perecptron risk function is not smooth, but it is convex. To solve this optimization problem, we can use gradient descent with the update rule

$$\boldsymbol{w_{k+1}} = \boldsymbol{w_k} - \eta \nabla_w R(\boldsymbol{w})\big|_{\boldsymbol{w}=\boldsymbol{w_k}},$$

where η is a positive step size that determines the rate of convergence. To use this algorithm, we first need to compute the gradient of the risk function:

$$\nabla_w R(\boldsymbol{w}) = \nabla_w \left(\sum_{i \in \mathcal{V}} -y_i(\boldsymbol{w}^T \boldsymbol{x_i}) \right) = \sum_{i \in \mathcal{V}} \nabla_w \Big(-y_i(\boldsymbol{w}^T \boldsymbol{x_i}) \Big) = \sum_{i \in \mathcal{V}} -y_i \boldsymbol{x_i}.$$

This now allows us to write the perceptron algorithm shown in algorithm 1. The issue with this algorithm is that each step takes $O(nd)$ time. We can speed up this algorithm by using stochastic gradient descent, instead of batch gradient descent. With stochastic gradient descent, we pick only one misclassified sample at each step, rather than performing gradient descent using all misclassified samples. With this modification, each step takes $O(d)$ time instead of $O(nd)$. This modified algorithm is provided in algorithm 2.

Algorithm 1: Perceptron Algorithm with Batch Gradient Descent

$\boldsymbol{w} \leftarrow$ arbitrary non-zero starting point
$\eta \leftarrow$ desired step size (learning rate)
while $R(\boldsymbol{w}) > 0$ **do**
  $\mathcal{V} \leftarrow$ set of indices for which $y_i(\boldsymbol{w}^T \boldsymbol{x_i}) < 0$
  $\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta \sum_{i \in \mathcal{V}} y_i \boldsymbol{x_i}$
end
return $\boldsymbol{w}$

Algorithm 2: Perceptron Algorithm with Stochastic Gradient Descent

$\boldsymbol{w} \leftarrow$ arbitrary non-zero starting point
$\eta \leftarrow$ desired step size (learning rate)
while $y_i(\boldsymbol{w}^T \boldsymbol{x_i}) < 0$ *for some* i **do**
  $\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta y_i \boldsymbol{x_i}$
end
return $\boldsymbol{w}$

3.3.2 Perceptron Algorithm With Bias

Previously, we assumed that the separating hyperplane passed through the origin, which allowed us to assume $\alpha = 0$. Now we will no longer assume that the separting hyperplane passes through the origin. To deal with this difference, we define a new weight vector $\tilde{\boldsymbol{w}} \in \mathbb{R}^{d+1}$ and sample point $\tilde{\boldsymbol{x}} \in \mathbb{R}^{d+1}$ such that

$$\tilde{\boldsymbol{w}} = \begin{bmatrix} \boldsymbol{w} \\ \alpha \end{bmatrix} \quad \text{and} \quad \tilde{\boldsymbol{x}} = \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}.$$

Now we can express our decision function as $f(\tilde{\boldsymbol{x}}) = \tilde{\boldsymbol{w}}^T \tilde{\boldsymbol{x}}$. We can use this decision function with our previous perceptron algorithm to find a separating hyperplane that does not pass through the origin.

3.3.3 Convergence of Perceptron Algorithm

If the training data is linearly separable, then the perceptron algorithm will perfectly classify the training data, resulting in zero training error. Furthermore, the perceptron algorithm will find a separating hyperplane that correctly classifies the data in at most $O(r^2/\gamma^2)$ iterations, where $r = \max_i ||\boldsymbol{x_i}||_2$ is the radius of the data and γ is the maximum margin (i.e. the distance from the decision boundary to the nearest sample point). If the data is not linearly separable, then the algorithm never terminates.

Chapter 4

Support Vector Machines (SVMs)

4.1 Hard-Margin SVM

4.1.1 Maximum Margin Classifier

Consider a data set of n d-dimensional sample points: $\{\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}\}$. Each sample point, $\boldsymbol{x_i} \in \mathbb{R}^d$, has a corresponding label $y_i \in \{1, -1\}$. Recall that the decision boundary for any linear classifier can be expressed as the hyperplane $\mathcal{H} = \{\boldsymbol{x} \in \mathbb{R}^d : \boldsymbol{w}^T\boldsymbol{x} + \alpha = 0\}$. The **maximum margin classifier** maximizes the distance from the linear decision boundary to the closest training point on either side of the decision boundary. The gap between the decision boundary and the closest training point on each side is called the **margin**. Figure 4.1 displays a maximum margin classifier, which demonstrates these concepts.

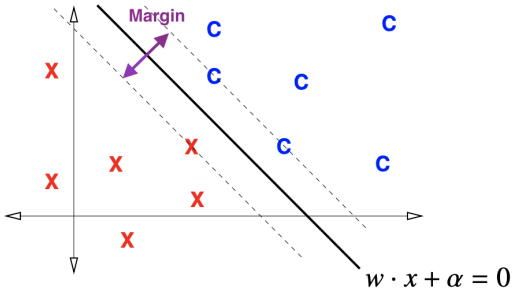

Figure 4.1: The hyperplane $\mathcal{H} = \{\boldsymbol{x} \in \mathbb{R}^d : \boldsymbol{w}^T\boldsymbol{x} + \alpha = 0\}$ for a maximum margin classifier provides the maximum distance between the boundary and the nearest sample point. This distance is called the margin and is equal on either side of the boundary.

4.1.2 Hard-Margin SVM Problem

The goal of the **hard-margin SVM** problem is to find the weight vector, $\boldsymbol{w}$, and bias, α, such that the linear decision boundary, $\mathcal{H} = \{\boldsymbol{x} \in \mathbb{R}^d : \boldsymbol{w}^T\boldsymbol{x} + \alpha = 0\}$, provides the maximum margin classification. In order to ensure that the points are correctly classified, we enforce the following set of constraints:

$$\begin{cases} \boldsymbol{w}^T\boldsymbol{x_i} + \alpha \geq 1 & \text{if } y_i = 1 \\ \boldsymbol{w}^T\boldsymbol{x_i} + \alpha \leq -1 & \text{if } y_i = -1 \end{cases} \text{ for } \ i = 1, \ldots, n.$$

We can combine these two sets of constraints into a single constraint:

$$y_i(\boldsymbol{w}^T\boldsymbol{x_i} + \alpha) \geq 1 \text{ for } i = 1, \ldots, n.$$

Recall that the signed distance from any point $\boldsymbol{x_i}$ to the hyperplane $\mathcal{H}$ is

$$d_i = \frac{\boldsymbol{w}^T\boldsymbol{x_i} + \alpha}{||\boldsymbol{w}||_2}.$$

Therefore, the unsigned distance from any point $\boldsymbol{x_i}$ to the hyperplane $\mathcal{H}$ is

$$|d_i| = \frac{|\boldsymbol{w}^T\boldsymbol{x_i} + \alpha|}{||\boldsymbol{w}||_2}.$$

Recall that the margin is the distance from the decision boundary to the nearest sample point. Therefore, the margin is the smallest unsigned distance across all n sample points. Based on our constraints for the sample points, we know

$$|\boldsymbol{w}^T\boldsymbol{x_i} + \alpha| \geq 1, \ \forall i \in \{1, \ldots, n\}.$$

Therefore, we have the following constraint on the unsigned distance:

$$|d_i| = \frac{|\boldsymbol{w}^T\boldsymbol{x_i} + \alpha|}{||\boldsymbol{w}||_2} \geq \frac{1}{||\boldsymbol{w}||_2}, \ \forall i \in \{1, \ldots, n\}.$$

This implies that the margin can never be less than $\frac{1}{||\boldsymbol{w}||_2}$. Therefore, to maximize the margin, we want to maximize $\frac{1}{||\boldsymbol{w}||_2}$, which is equivalent to minimizing $||\boldsymbol{w}||_2$. This function is not smooth, so we will instead minimize the smooth function $||\boldsymbol{w}||_2^2$. Now we can express the hard-margin SVM problem as

$$\begin{aligned} &\min_{\boldsymbol{w},\alpha} \ ||\boldsymbol{w}||_2^2 \\ &\text{s.t.} \quad y_i(\boldsymbol{w}^T\boldsymbol{x_i} + \alpha) \geq 1, \ i = 1, \ldots, n \end{aligned}$$

After finding the optimal weight vector $\hat{\boldsymbol{w}} \in \mathbb{R}^d$ and optimal bias term $\hat{\alpha} \in \mathbb{R}$ that solve this problem, we can express the hard-margin SVM decision rule as

$$f(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \hat{\boldsymbol{w}}^T\boldsymbol{x} + \hat{\alpha} \geq 0 \\ -1 & \text{otherwise} \end{cases}.$$

As with the perceptron algorithm, the hard-margin SVM problem can achieve zero training error on any linearly separable data set, but it will never find a solution for a data set that is not linearly separable. As another note, for a classifier with weight vector $\hat{\boldsymbol{w}}$, the margin is exactly $\frac{1}{||\hat{\boldsymbol{w}}||_2}$ and there is a space of width $\frac{2}{||\hat{\boldsymbol{w}}||_2}$ around the linear decision boundary that contains no sample points. An example of a hard-margin SVM classifier is shown in figure 4.2.

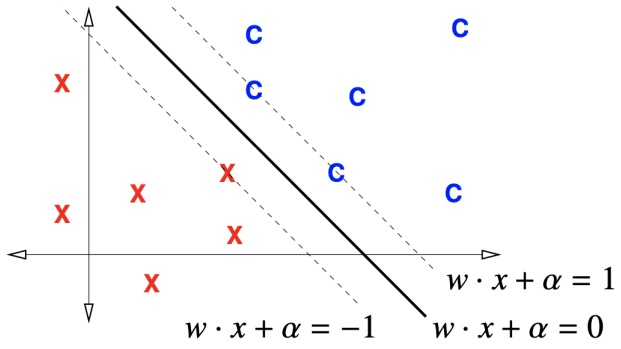

Figure 4.2: This is an example of a hyperplane obtained through a hard-margin SVM classification problem. The decision boundary is given by $\mathcal{H} = \{\boldsymbol{x} \in \mathbb{R}^d : \boldsymbol{w}^T\boldsymbol{x} + \alpha = 0\}$. If $\boldsymbol{x}$ is in class C, then $\boldsymbol{w}^T\boldsymbol{x} + \alpha \geq 1$. If $\boldsymbol{x}$ is not in class C, then $\boldsymbol{w}^T\boldsymbol{x} + \alpha \leq -1$.

4.1.3 Support Vectors

A **support vector** is defined as a training sample that lies on the margin. A support vector, $\boldsymbol{x}_+$, in the given class satisfies $\hat{\boldsymbol{w}}^T\boldsymbol{x}_+ + \alpha = 1$, and a support vector, $\boldsymbol{x}_-$, not in the given class satisfies $\hat{\boldsymbol{w}}^T\boldsymbol{x}_- + \alpha = -1$. The example shown in figure 4.2 has two support vectors in class C and one not in this class.

Theorem: There must be at least one support vector for each class.

Proof: Let's assume that there is no support vector for class C. We could either define C as $C := \{i \in \{1, \ldots, n\} : y_i = 1\}$ or $C := \{i \in \{1, \ldots, n\} : y_i = -1\}$. The proof holds for either definition of C. This assumption implies that

$$|\hat{\boldsymbol{w}}^T\boldsymbol{x_i} + \hat{\alpha}| > 1, \ \forall i \in C.$$

There is some constant $\epsilon > 1$ for which we can write this strict inequality as

$$|\hat{\boldsymbol{w}}^T\boldsymbol{x_i} + \hat{\alpha}| \geq \epsilon, \ \forall i \in C.$$

Now let's suppose we have new parameters $\bar{\boldsymbol{w}}$ and $\bar{\alpha}$, which we will define as

$$\bar{\boldsymbol{w}} := \frac{2\hat{\boldsymbol{w}}}{1+\epsilon} \quad \text{and} \quad \bar{\alpha} := \frac{2\hat{\alpha}}{1+\epsilon}.$$

For these new parameters, we can notice that

$$|\bar{\boldsymbol{w}}^T \boldsymbol{x_i} + \bar{\alpha}| = \left| \left(\frac{2\hat{\boldsymbol{w}}}{1+\epsilon} \right)^T \boldsymbol{x_i} + \frac{2\hat{\alpha}}{1+\epsilon} \right| = \frac{2}{1+\epsilon} \left| \hat{\boldsymbol{w}}^T \boldsymbol{x_i} + \hat{\alpha} \right| \geq \frac{2\epsilon}{1+\epsilon}, \ \forall i \in C.$$

Note that $\frac{2\epsilon}{1+\epsilon} > 1$ for all $\epsilon > 1$. Therefore, the parameters $\bar{\boldsymbol{w}}$ and $\bar{\alpha}$ satisfy

$$|\bar{\boldsymbol{w}}^T \boldsymbol{x_i} + \bar{\alpha}| > 1, \ \forall i \in C.$$

This implies that $\bar{\boldsymbol{w}}$ and $\bar{\alpha}$ satisfy the constraints of the hard-margin SVM problem. Now notice that we can also express the squared norm of $\bar{\boldsymbol{w}}$ as

$$||\bar{\boldsymbol{w}}||_2^2 = \left|\left| \frac{2\hat{\boldsymbol{w}}}{1+\epsilon} \right|\right|_2^2 = \left(\frac{2}{1+\epsilon} \right)^2 ||\hat{\boldsymbol{w}}||_2^2.$$

Note that $\frac{2}{1+\epsilon} < 1$ for all $\epsilon > 1$. Therefore, $||\bar{\boldsymbol{w}}||_2^2 < ||\hat{\boldsymbol{w}}||_2^2$. This implies that $\bar{\boldsymbol{w}}$ and $\bar{\alpha}$ result in a linear classifier with a larger margin than the one defined by the parameters $\hat{\boldsymbol{w}}$ and $\hat{\alpha}$. This contradicts our assumption that $\hat{\boldsymbol{w}}$ and $\hat{\alpha}$ are the optimal solutions to the hard-margin SVM problem. Now we have proven by contradiction that there is at least one support vector for each class.

4.2 Soft-Margin SVM

Hard-margin SVMs are sensitive to outliers and fail if the data is not linearly separable. **Soft-margin SVMs** address this issue by allowing some points to violate the margin. For hard-margin SVMs, we enforced the constraints:

$$y_i(\boldsymbol{w}^T \boldsymbol{x_i} + \alpha) \geq 1 \text{ for } i = 1, \ldots, n.$$

For soft-margin SVMs, we will instead use the constraints:

$$y_i(\boldsymbol{w}^T \boldsymbol{x_i} + \alpha) \geq 1 - \xi_i, \ i = 1 \ldots, n$$

$$\xi_i \geq 0, \ i = 1 \ldots, n$$

In these constraints, ξ_i is a slack variable that is zero if the sample point $\boldsymbol{x_i}$ is correctly classified and positive if $\boldsymbol{x_i}$ violates the margin. The magnitude of ξ_i indicates how much $\boldsymbol{x_i}$ violates the margin. This is demonstrated in figure 4.3.

For the soft-margin SVM, we still want to minimize $||\boldsymbol{w}||_2^2$, but now we also want to add a loss term to the objective function to limit how much points violate the margin. The soft-margin SVM problem can then be expressed as

$$\begin{aligned} \min_{\boldsymbol{w},\alpha,\xi} \quad & ||\boldsymbol{w}||_2^2 + c \sum_{i=1}^{n} \xi_i \\ \text{s.t.} \quad & y_i(\boldsymbol{w}^T \boldsymbol{x_i} + \alpha) \geq 1 - \xi_i, \ i = 1 \ldots, n \\ & \xi_i \geq 0, \ i = 1 \ldots, n \end{aligned}$$

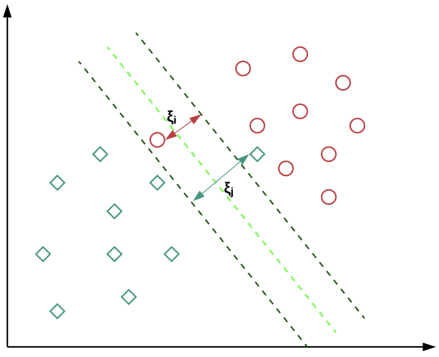

Figure 4.3: This data is not linearly separable, so hard-margin SVM would fail to classify this data. By introducing slack variables, soft-margin SVM allows two of the points to violate the margin. Notice that we no longer require support vectors.

In this optimization problem, the constant c is a scalar regularization hyperparameter, which is chosen using validation. If c is small, then we care more about maximizing the margin, $\frac{1}{||\boldsymbol{w}||_2}$, while allowing more points in the training data to violate the margin. We generally get decision boundaries that are less sensitive to outliers, which gives us the risk of underfitting. If c is large, then we care more about limiting the extent to which sample points violate the margin, while shrinking the size of the margin. We generally get decision boundaries that are more sensitive to outliers, which gives us the risk of overfitting. As c becomes larger, the soft-margin SVM problem becomes closer to the hard-margin SVM problem. In the limit $c \to \infty$, we recover the hard-margin SVM.

4.3 Adding Features

To improve the results for hard-margin or soft-margin SVMs, we can add additional features. If we augment the data with additional features, the optimal value of the objective function will either decrease or stay the same, but it cannot increase. This is because we can always use the optimal weight vector $\hat{\boldsymbol{w}}$ for the original data and set $\hat{w}_i = 0$ for the weights corresponding to added features. Therefore, the new optimum must be at least as good as the original.

Again, assume we are working with a data set of n d-dimensional sample points: $\{\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}\}$. So far, we have considered linear decision boundaries to classify this data. We said these points are linearly separable if there exists a hyperplane that separates all of the sample points in a given class from points not in this class. While SVMs generally generate linear decision boundaries, we can also use SVMs to find nonlinear decision boundaries by generating nonlinear features.

4.3.1 Parabolic Lifting Map

One way to find nonlinear decision boundaries by generating nonlinear features is by using a **parabolic lifting map** $\phi : \mathbb{R}^d \to \mathbb{R}^{d+1}$, which is defined such that

$$\phi(\boldsymbol{x}) = \begin{bmatrix} \boldsymbol{x} \\ ||\boldsymbol{x}||_2^2 \end{bmatrix}.$$

With the parabolic lifting map, we can find a linear classifier in the ϕ-space, which induces a sphere classifier in the x-space. Note that $\phi(\boldsymbol{x_1}), \ldots, \phi(\boldsymbol{x_n})$ are linearly separable if and only if $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ are separable by a hypersphere. The role of the parabolic lifting map in SVMs is demonstrated in figure 4.4.

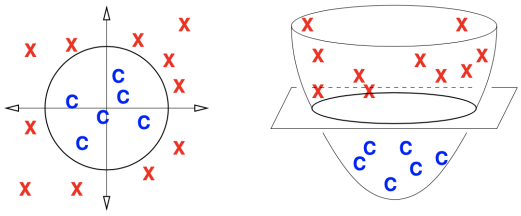

Figure 4.4: On the left, the data is shown in x-space, and on the right, the data is shown in ϕ-space. In x-space, the data is not linearly separable, but it is separable by a hypersphere. This implies that in ϕ-space, the data is linearly separable.

4.3.2 Ellipsoid & Hyperboloid Decision Boundaries

We just showed that we can use SVMs to separate data using a hypersphere. We can also use SVMs to separate data with an axis-aligned ellipsoid or hyperboloid. To do so, we can use the map $\phi : \mathbb{R}^d \to \mathbb{R}^{2d}$, which is defined such that

$$\phi(\boldsymbol{x}) = \begin{bmatrix} x_1^2 & \ldots & x_d^2 & x_1 & \ldots & x_d \end{bmatrix}^T.$$

The linear classifier in ϕ-space can then be expressed as

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \phi(\boldsymbol{x}) + \alpha,$$

where $\boldsymbol{w} \in \mathbb{R}^{2d}$ is the weight vector and $\alpha \in \mathbb{R}$ is the bias. We can generalize this to ellipsoids and hyperboloids that are not necessarily axis-aligned. To do so, we can use the map $\phi : \mathbb{R}^d \to \mathbb{R}^{(d^2+3d)/2}$, which is defined such that

$$\phi(\boldsymbol{x}) = \begin{bmatrix} x_1^2 & \ldots & x_d^2 & x_1x_2 & \ldots & x_1x_d & \ldots & x_1 & \ldots & x_d \end{bmatrix}^T.$$

Again, the linear classifier in ϕ-space can then be expressed as

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \phi(\boldsymbol{x}) + \alpha,$$

where $\boldsymbol{w} \in \mathbb{R}^{(d^2+3d)/2}$ is the weight vector and $\alpha \in \mathbb{R}$ is the bias term.

4.3.3 Polynomial Decision Boundaries

We can also use SVMs to separate data using a polynomial decision boundary. A cubic polynomial function in $\mathbb{R}^2$ uses the map $\phi : \mathbb{R}^2 \to \mathbb{R}^9$ defined such that

$$\phi(\boldsymbol{x}) = \begin{bmatrix} x_1^3 & x_2^3 & x_1^2 x_2 & x_1 x_2^2 & x_1^2 & x_2^2 & x_1 x_2 & x_1 & x_2 \end{bmatrix}^T.$$

This is useful because a hyperplane can only separate a linear function into at most two regions, while a hyperplane can separate a quadratic function into at most three regions and a cubic function into at most four regions. Figure 4.5 gives an example of hard-margin SVM using a cubic polynomial function.

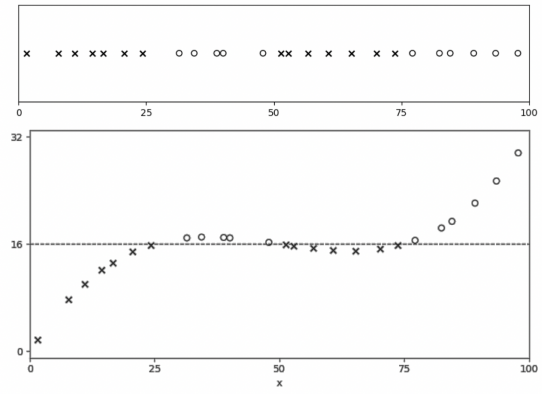

Figure 4.5: We are not able to separate the one-dimensional data $\{\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}\}$ with a single line. However, if we add the features x_i^2 and x_i^3, a line can separate the cubic function into four segments that perfectly classify the training data.

Note that if we have sample points with d features and want to use a polynomial decision boundary of degree p, then we need to use a map $\phi : \mathbb{R}^d \to \mathbb{R}^{O(d^p)}$. This increases the number of feature significantly and can make the SVM problem computationally infeasible to solve as the degree or number of features increases.

Chapter 5

Bayes Decision Rule

5.1 Two Classes

Consider a system composed of d-dimensional feature vectors and corresponding labels. Let $\boldsymbol{X}$ be a random feature vector with a random label Y. For now, we will assume there are only two classes and that the range of Y is the discrete set $\{1, -1\}$. The range of $\boldsymbol{X}$, which we'll denote $\mathcal{X}$, is some subset of $\mathbb{R}^d$. We define the **decision rule** as the map $r : \mathbb{R}^d \to \{1, -1\}$, which maps a feature vector to the label 1 or -1, corresponding to the class of the sample point.

5.1.1 Bayes Decision Rule: Asymmetric Loss

The **risk** of a decision function is defined as the expected loss. We will initially assume the range of X is a countable subset of $\mathbb{R}^d$ and that X can be modeled by a discrete distribution. Let $\boldsymbol{x}$ be a realization of $\boldsymbol{X}$ and y be a realization of Y. The loss of the decision function for feature vector $\boldsymbol{x}$ with label y is $L(r(\boldsymbol{x}), y)$. Assuming we are working with discrete distributions, the risk is given by

$$R(r) = \mathbb{E}_{\boldsymbol{X},Y}[L(r(\boldsymbol{x}), y)] = \sum_{\boldsymbol{x}\in\mathcal{X}} \sum_{y\in\{1,-1\}} L(r(\boldsymbol{x}), y) p_{\boldsymbol{X},Y}(\boldsymbol{x}, y).$$

Using Baye's rule, we can also express the risk for a decision function as

$$\begin{aligned} R(r) &= \sum_{\boldsymbol{x}\in\mathcal{X}} \sum_{y\in\{1,-1\}} L(r(\boldsymbol{x}), y) p_{Y|\boldsymbol{X}}(y|\boldsymbol{x}) p_{\boldsymbol{X}}(\boldsymbol{x}) \\ &= \sum_{\boldsymbol{x}\in\mathcal{X}} \Big(L(r(\boldsymbol{x}), 1) p_{Y|\boldsymbol{X}}(1|\boldsymbol{x}) p_{\boldsymbol{X}}(\boldsymbol{x}) + L(r(\boldsymbol{x}), -1) p_{Y|\boldsymbol{X}}(-1|\boldsymbol{x}) p_{\boldsymbol{X}}(\boldsymbol{x}) \Big) \\ &= \sum_{\boldsymbol{x}\in\mathcal{X}} \Big(L(r(\boldsymbol{x}), 1) p_{Y|\boldsymbol{X}}(1|\boldsymbol{x}) + L(r(\boldsymbol{x}), -1) p_{Y|\boldsymbol{X}}(-1|\boldsymbol{x}) \Big) p_{\boldsymbol{X}}(\boldsymbol{x}). \end{aligned}$$

We want to find the decision rule that results in the lowest risk. We'll denote the optimal decision rule r^*. From our definition of risk, r^* can be expressed as

$$\begin{aligned} r^* &= \underset{r:\mathcal{X}\to\{1,-1\}}{\arg\min} \; R(r) \\ &= \underset{r:\mathcal{X}\to\{1,-1\}}{\arg\min} \sum_{\boldsymbol{x}\in\mathcal{X}} \Big(L(r(\boldsymbol{x}),1) p_{Y|\boldsymbol{X}}(1|\boldsymbol{x}) + L(r(\boldsymbol{x}),-1) p_{Y|\boldsymbol{X}}(-1|\boldsymbol{x}) \Big) p_{\boldsymbol{X}}(\boldsymbol{x}). \end{aligned}$$

For a single sample point $\boldsymbol{x}$, the optimal decision rule $r^*(\boldsymbol{x})$ satisfies

$$\begin{aligned} r^*(\boldsymbol{x}) &= \underset{r(\boldsymbol{x})\in\{1,-1\}}{\arg\min} \Big(L(r(\boldsymbol{x}),1) p_{Y|\boldsymbol{X}}(1|\boldsymbol{x}) + L(r(\boldsymbol{x}),-1) p_{Y|\boldsymbol{X}}(-1|\boldsymbol{x}) \Big) p_{\boldsymbol{X}}(\boldsymbol{x}) \\ &= \underset{r(\boldsymbol{x})\in\{1,-1\}}{\arg\min} \Big(L(r(\boldsymbol{x}),1) p_{Y|\boldsymbol{X}}(1|\boldsymbol{x}) + L(r(\boldsymbol{x}),-1) p_{Y|\boldsymbol{X}}(-1|\boldsymbol{x}) \Big). \end{aligned}$$

If we assume that we do not incur loss for correctly classified samples, then $L(z,y) = 0$ when $z = y$. Under this condition, the optimal decision rule satisfies

$$r^*(\boldsymbol{x}) = \begin{cases} 1 & \text{if } L(-1,1) p_{Y|\boldsymbol{X}}(1|\boldsymbol{x}) > L(1,-1) p_{Y|\boldsymbol{X}}(-1|\boldsymbol{x}) \\ -1 & \text{otherwise} \end{cases}.$$

This says that we should choose 1 if the expected loss for choosing -1 is higher and -1 if the expected loss for choosing 1 is higher. This decision rule is known as **Bayes decision rule**. We can equivalently express this decision rule as

$$r^*(\boldsymbol{x}) = \underset{y\in\{1,-1\}}{\arg\min} \; L(y,-y) p_{Y|\boldsymbol{X}}(-y|\boldsymbol{x}).$$

The **Bayes decision boundary** corresponding to this decision rule is given by

$$\{\boldsymbol{x} \in \mathcal{X} \; : \; L(-1,1) p_{Y|\boldsymbol{X}}(1|\boldsymbol{x}) = L(1,-1) p_{Y|\boldsymbol{X}}(-1|\boldsymbol{x})\}.$$

Extension to Continuous Distributions

If we are working with continuous distributions, instead of discrete distributions, then we define the risk function using the probability density function (PDF), $f_{\boldsymbol{X}}(\boldsymbol{x})$, instead of the probability mass function (PMF), $p_{\boldsymbol{X}}(\boldsymbol{x})$. Instead of summing over the range $\mathcal{X}$, we instead need to integrate over $\mathcal{X}$. The optimal decision rule and decision boundary are nearly the same as in the discrete case. In the continuous case, we can express Bayes decision rule as

$$r^*(\boldsymbol{x}) = \underset{y\in\{1,-1\}}{\arg\min} \; L(y,-y) f_{Y|\boldsymbol{X}}(-y|\boldsymbol{x}).$$

Similarly, Bayes decision boundary in the continuous case is given by

$$\{\boldsymbol{x} \in \mathcal{X} \; : \; L(-1,1) f_{Y|\boldsymbol{X}}(1|\boldsymbol{x}) = L(1,-1) f_{Y|\boldsymbol{X}}(-1|\boldsymbol{x})\}.$$

5.1.2 Bayes Decision Rule: Symmetric Loss

Previously, we did not make any assumptions about the loss function, L. We will now assume that the loss function is symmetric. When L is not symmetric (i.e. $L(-1,1) \neq L(1,-1)$), we need to weigh the posterior probabilities with the losses. When the loss function is symmetric (i.e. $L(-1,1) = L(1,-1)$), we simply pick the class with the largest posterior probability. One common symmetric loss function is the **zero-one loss function**, which is defined as

$$L(z,y) = \begin{cases} 1 & \text{if } z \neq y \\ 0 & \text{if } z = y \end{cases}.$$

This says that we incur a loss of 1 for incorrectly classified samples and 0 for correctly classified samples. Because we do not incur loss for correctly classified samples with this loss function, we can express the Bayes decision rule as

$$r^*(\boldsymbol{x}) = \underset{y\in\{1,-1\}}{\arg\min}\, p_{Y|\boldsymbol{X}}(-y|\boldsymbol{x}) = \underset{y\in\{1,-1\}}{\arg\max}\, p_{Y|\boldsymbol{X}}(y|\boldsymbol{x}).$$

Using Bayes rule, this decision rule can equivalently be expressed as

$$r^*(\boldsymbol{x}) = \underset{y\in\{1,-1\}}{\arg\max}\, \frac{p_{\boldsymbol{X}|Y}(\boldsymbol{x}|y)p_Y(y)}{p_{\boldsymbol{X}}(\boldsymbol{x})}.$$

Removing the constant term, we can also express the decision rule as

$$r^*(\boldsymbol{x}) = \underset{y\in\{1,-1\}}{\arg\max}\, p_{\boldsymbol{X}|Y}(\boldsymbol{x}|y)p_Y(y).$$

The Bayes decision boundary corresponding to this decision rule is then

$$\{\boldsymbol{x} \in \mathcal{X} \ : \ p_{Y|\boldsymbol{X}}(1|\boldsymbol{x}) = p_{Y|\boldsymbol{X}}(-1|\boldsymbol{x})\} \ \text{ or}$$

$$\{\boldsymbol{x} \in \mathcal{X} \ : \ p_{\boldsymbol{X}|Y}(\boldsymbol{x}|1)p_Y(1) = p_{\boldsymbol{X}|Y}(\boldsymbol{x}|-1)p_Y(-1)\}.$$

Extension to Continuous Distributions

If we are working with continuous distributions, instead of discrete distributions, the optimal decision rule and decision boundary are nearly the same. In the continuous case, we can express Bayes decision rule as

$$r^*(\boldsymbol{x}) = \underset{y\in\{1,-1\}}{\arg\max}\, f_{Y|\boldsymbol{X}}(y|\boldsymbol{x}).$$

Using Bayes rule, we can also express this decision rule as

$$r^*(\boldsymbol{x}) = \underset{y\in\{1,-1\}}{\arg\max}\, f_{\boldsymbol{X}|Y}(\boldsymbol{x}|y)p_Y(y).$$

Similarly, Bayes decision boundary in the continuous case is given by

$$\{\boldsymbol{x} \in \mathcal{X} \ : \ f_{Y|\boldsymbol{X}}(1|\boldsymbol{x}) = f_{Y|\boldsymbol{X}}(-1|\boldsymbol{x})\} \ \text{ or}$$

$$\{\boldsymbol{x} \in \mathcal{X} \ : \ f_{\boldsymbol{X}|Y}(\boldsymbol{x}|1)p_Y(1) = f_{\boldsymbol{X}|Y}(\boldsymbol{x}|-1)p_Y(-1)\}.$$

5.1.3 Bayes Risk

The **Bayes risk**, which is also known as the optimal risk, is the risk associated with the Bayes classifier. Recall that we expressed the risk of a decision rule as

$$R(r) = \sum_{\boldsymbol{x} \in \mathcal{X}} \Big(L(r(\boldsymbol{x}), 1) p_{Y|\boldsymbol{X}}(1|\boldsymbol{x}) + L(r(\boldsymbol{x}), -1) p_{Y|\boldsymbol{X}}(-1|\boldsymbol{x}) \Big) p_{\boldsymbol{X}}(\boldsymbol{x}).$$

Assuming that we do not incur loss for correctly classified samples, we found that the Bayes decision rule must satisfy

$$r^*(\boldsymbol{x}) = \underset{y \in \{1,-1\}}{\arg\min}\, L(y, -y) p_{Y|\boldsymbol{X}}(-y|\boldsymbol{x}).$$

Because we assume that we do not incur loss for correctly classified samples, $L(r(\boldsymbol{x}), y) = 0$ if $r(\boldsymbol{x}) = y$. Therefore, the Bayes (optimal) risk is given by

$$\begin{aligned} R(r^*) &= \sum_{\boldsymbol{x} \in \mathcal{X}} \left(\min_{y \in \{1,-1\}} L(y, -y) p_{Y|\boldsymbol{X}}(-y|\boldsymbol{x}) \right) p_{\boldsymbol{X}}(\boldsymbol{x}) \\ &= \min_{y \in \{1,-1\}} \sum_{\boldsymbol{x} \in \mathcal{X}} L(y, -y) p_{Y|\boldsymbol{X}}(-y|\boldsymbol{x}) p_{\boldsymbol{X}}(\boldsymbol{x}). \end{aligned}$$

Using Bayes rule, we can equivalently express the Bayes risk as

$$R(r^*) = \min_{y=\{1,-1\}} \sum_{\boldsymbol{x} \in \mathcal{X}} L(y, -y) p_{\boldsymbol{X}|Y}(\boldsymbol{x}| - y) p_Y(-y).$$

Note that Bayes risk is zero when the conditional class distributions $p_{\boldsymbol{X}|Y}(\boldsymbol{x}|1)$ and $p_{\boldsymbol{X}|Y}(\boldsymbol{x}| - 1)$ do not overlap or if one of the prior probabilities $p_Y(1)$ or $p_Y(-1)$ is equal to one. The Bayes risk does not depend on the specific training data; even if the data is linearly separable or the Bayes decision rule perfectly classifies the training data, we generally still have non-zero risk.

Extension to Continuous Distributions

If we are working with continuous distributions, instead of discrete distributions, then we define the risk function using the probability density function (PDF), $f_{\boldsymbol{X}}(\boldsymbol{x})$, and conditional PDF, $f_{Y|\boldsymbol{X}}(y|\boldsymbol{x})$. Instead of summing over the range $\mathcal{X}$, we need to integrate over $\mathcal{X}$. The Bayes (optimal) risk is nearly the same as in the discrete case. In the continuous case, the risk can be expressed as

$$\begin{aligned} R(r^*) &= \min_{y \in \{1,-1\}} \int_{\mathcal{X}} L(y, -y) f_{Y|\boldsymbol{X}}(-y|\boldsymbol{x}) f_{\boldsymbol{X}}(\boldsymbol{x})\, d\boldsymbol{x} \\ &= \min_{y=\{1,-1\}} \int_{\mathcal{X}} L(y, -y) f_{\boldsymbol{X}|Y}(\boldsymbol{x}| - y) p_Y(-y)\, d\boldsymbol{x}. \end{aligned}$$

5.2 Multiple Classes

Again, consider a system composed of d-dimensional feature vectors and corresponding labels. Let $\boldsymbol{X}$ be a random feature vector with a random label Y. We will now assume that there are an arbitrary number of classes and the range of Y is C, where C is the set of possible classes. We define the **decision rule** as the map $r : \mathbb{R}^d \to C$, which maps a feature vector to a label in the set of classes.

5.2.1 Bayes Decision Rule: Asymmetric Loss

We can extend the definition of the Bayes decision rule with asymmetric loss to handle an arbitrary number of classes. Again, the loss of the decision function for feature vector $\boldsymbol{x}$ with label y is denoted $L(r(\boldsymbol{x}), y)$. For the two class case with discrete distributions, we said the optimal decision rule is given by

$$r^*(\boldsymbol{x}) = \underset{y \in \{1,-1\}}{\arg\min}\, L(y, -y) p_{Y|\boldsymbol{X}}(-y|\boldsymbol{x}).$$

We can extend this definition for systems with more than two classes. For a discrete feature space, the Bayes optimal decision rule is given by

$$r^*(\boldsymbol{x}) = \underset{z \in C}{\arg\min} \sum_{y \in C} L(z, y) p_{Y|\boldsymbol{X}}(y|\boldsymbol{x}).$$

Similarly, for a continuous feature space, the Bayes optimal decision rule is

$$r^*(\boldsymbol{x}) = \underset{z \in C}{\arg\min} \sum_{y \in C} L(z, y) f_{Y|\boldsymbol{X}}(y|\boldsymbol{x}).$$

5.2.2 Bayes Decision Rule: Symmetric Loss

We can also extend the definition of the Bayes decision rule with symmetric loss for the two class case to handle more than two classes. For the two class case with discrete distributions, we said that the optimal decision rule is given by

$$r^*(\boldsymbol{x}) = \underset{y \in \{1,-1\}}{\arg\max}\; p_{\boldsymbol{X}|Y}(\boldsymbol{x}|y) p_Y(y).$$

We can extend this definition for systems with more than two classes. For a discrete feature space, the Bayes optimal decision rule is given by

$$r^*(\boldsymbol{x}) = \underset{y \in C}{\arg\max}\; p_{\boldsymbol{X}|Y}(\boldsymbol{x}|y) p_Y(y).$$

Similarly, for a continuous feature space, the Bayes optimal decision rule is

$$r^*(\boldsymbol{x}) = \underset{y \in C}{\arg\max}\; f_{\boldsymbol{X}|Y}(\boldsymbol{x}|y) p_Y(y).$$

5.2.3 Bayes Risk

Discrete Distributions

Recall that the Bayes risk is the risk associated with the Bayes classifier. For an arbitrary number of classes, we can express the risk of a decision function as

$$R(r) = \sum_{\boldsymbol{x} \in \mathcal{X}} \sum_{y \in C} L(r(\boldsymbol{x}), y) p_{Y|\boldsymbol{X}}(y, \boldsymbol{x}) p_{\boldsymbol{X}}(\boldsymbol{x}).$$

For the discrete multi-class case, the Bayes optimal decision rule is given by

$$r^*(\boldsymbol{x}) = \arg\min_{z \in C} \sum_{y \in C} L(z, y) p_{Y|\boldsymbol{X}}(y|\boldsymbol{x}).$$

Therefore, the Bayes (optimal) risk can be expressed as

$$R(r^*) = \min_{z \in C} \sum_{\boldsymbol{x} \in \mathcal{X}} \sum_{y \in C} L(z, y) p_{Y|\boldsymbol{X}}(y|\boldsymbol{x}) p_{\boldsymbol{X}}(\boldsymbol{x}).$$

Using Bayes rule, we can also express the Bayes risk as

$$R(r^*) = \min_{z \in C} \sum_{\boldsymbol{x} \in \mathcal{X}} \sum_{y \in C} L(z, y) p_{\boldsymbol{X}|Y}(\boldsymbol{x}|y) p_Y(y).$$

Continuous Distributions

Similarly, for the continuous multi-class case, we can express the risk as

$$R(r) = \int_{\mathcal{X}} \left(\sum_{y \in C} L(r(\boldsymbol{x}), y) f_{Y|\boldsymbol{X}}(y, \boldsymbol{x}) f_{\boldsymbol{X}}(\boldsymbol{x}) \right) d\boldsymbol{x}.$$

For the continuous multi-class case, the Bayes optimal decision rule is

$$r^*(\boldsymbol{x}) = \arg\min_{z \in C} \sum_{y \in C} L(z, y) f_{Y|\boldsymbol{X}}(y|\boldsymbol{x}).$$

Therefore, the Bayes (optimal) risk can be expressed as

$$R(r^*) = \min_{z \in C} \int_{\mathcal{X}} \left(\sum_{y \in C} L(z, y) f_{Y|\boldsymbol{X}}(y|\boldsymbol{x}) f_{\boldsymbol{X}}(\boldsymbol{x}) \right) d\boldsymbol{x}.$$

Using Bayes rule, we can express the Bayes risk as

$$R(r^*) = \min_{z \in C} \int_{\mathcal{X}} \left(\sum_{y \in C} L(z, y) f_{\boldsymbol{X}|Y}(\boldsymbol{x}|y) p_Y(y) \right) d\boldsymbol{x}.$$

5.3 Generative & Discriminative Models

Bayes decision rule is the optimal decision rule that provides the lowest possible risk for a given system. If we know the underlying probability distributions of our data, then we can construct the ideal probabilistic classifier. However, in reality, we rarely know the probability distributions underlying real-world datasets. Some classifiers try to approximate Bayes decision rule by generating probability models from data. There are two main types of these models:

1. **Generative models** – Generative models assume a form of the likelihood distributions, $P(\{\boldsymbol{X} = \boldsymbol{x}|Y = c\})$, and prior probabilities, $P(\{Y = c\})$, for each class $c \in C$ and use data to fit the parameters of these probability models. These probability distributions can then be used to approximate Bayes decision rule. If desired, an estimate of the posterior probability, $P(\{Y = c|\boldsymbol{X} = \boldsymbol{x}\})$, for each class can be obtained through Bayes theorem.

   Example: Gaussian discriminant analysis (Section 8.1)

2. **Discriminative models** – Discriminative models assume a form of the posterior probability $P(Y = c|\boldsymbol{X} = \boldsymbol{x})$ for each class $c \in C$ directly and use data to fit the parameters of this model. This probability distribution can then be used to approximate Bayes decision rule.

   Example: Logistic regression (Section 9.5)

An alternative to approximating Bayes decision rule by estimating posterior probabilities is to find a decision boundary directly without considering underlying probability distributions, as we did with support vector machines.

Chapter 6

Multivariate Gaussians

6.1 Overview of Multivariate Gaussians

The multivariate normal distribution, multivariate Gaussian distribution, or joint normal distribution is a generalization of the one-dimensional (univariate) normal distribution to higher dimensions. For more information on the normal distribution, please see my notes on Probability & Random Processes. Multivariate Gaussians are useful in probability theory and statistics because they have nice properties. Furthermore, if we have a lot of data, the underlying distribution of datasets resembles a multivariate Gaussian in many cases. Multivariate Gaussians are relevant to our discussion of maximum likelihood estimation (MLE) in section 7 and our discussion of Gaussian discriminant analysis (GDA) in section 8.1, so we will first cover some background on these distributions.

6.2 Quadratic Forms

Before discussing multivariate Gaussians, we will first discuss quadratic forms. If $\boldsymbol{x}$ is an n-dimensional vector, then $||\boldsymbol{x}||_2^2 = \boldsymbol{x}^T\boldsymbol{x}$ is a quadratic function whose isosurfaces are spheres in n-dimensional space. If $\boldsymbol{A}$ is a symmetric positive definite matrix, then $||\boldsymbol{A}^{-1}\boldsymbol{x}||_2^2 = \boldsymbol{x}^T\boldsymbol{A}^{-2}\boldsymbol{x}$ is the quadratic form of the matrix $\boldsymbol{A}^{-2}$ and its isosurfaces are ellipsoids in n-dimensional space. The isosurface $||\boldsymbol{A}^{-1}\boldsymbol{x}||_2^2 = 1$ is an ellipsoid whose axes are the eigenvectors of the matrix $\boldsymbol{A}$ and whose radii are the corresponding eigenvalues of $\boldsymbol{A}$. Figure 6.1 depicts the isosurfaces for a quadratuc function and for the quadratic form of a matrix $\boldsymbol{A}^{-2}$.

If $\lambda_1, \ldots, \lambda_n$ are the eigenvalues of A with corresponding eigenvectors $\boldsymbol{v_1}, \ldots, \boldsymbol{v_n}$, then $||\boldsymbol{A}^{-1}\boldsymbol{x}||_2^2 = 1$ is an ellipsoid whose axes are given by $\boldsymbol{v_1}, \ldots, \boldsymbol{v_n}$ and whose radii are given by $\lambda_1, \ldots, \lambda_n$. If we are instead interested in the isosurface defined by $||\boldsymbol{A}^{-1}\boldsymbol{x}||_2^2 = c$, where c is some constant, then the axes of this ellipsoid are still given by $\boldsymbol{v_1}, \ldots, \boldsymbol{v_n}$ but the radii are now $\sqrt{c}\lambda_1, \ldots, \sqrt{c}\lambda_n$.

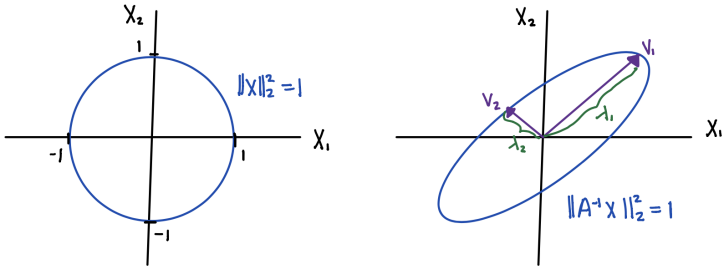

Figure 6.1: This figure depicts the isosurfaces of two quadratic functions. $\boldsymbol{x}$ is a 2-dimensional vector and $\boldsymbol{A}$ is a symmetric 2×2 matrix whose eigenvalues are λ_1, λ_2 and whose corresponding eigenvectors are $\boldsymbol{v_1}, \boldsymbol{v_2}$. The isocontours of $||\boldsymbol{x}||_2^2$ are circles, and $||\boldsymbol{x}||_2^2 = 1$ is a circle with radius one, as shown on the left. The isocontours of $||\boldsymbol{A}^{-1}\boldsymbol{x}||_2^2$ are ellipses, and $||\boldsymbol{A}^{-1}\boldsymbol{x}||_2^2 = 1$ is an ellipse whose major axis is $\boldsymbol{v_1}$ with radius λ_1 and whose minor axis is $\boldsymbol{v_2}$ with radius λ_2, as shown on the right.

6.3 Anisotropic Gaussians

An **anisotropic multivariate Gaussian** random variable $\boldsymbol{X} \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ with dimension d has the probability density function (PDF) given by

$$f_{\boldsymbol{X}}(\boldsymbol{x}) = \frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right).$$

The vector $\boldsymbol{\mu}$ is the mean, the matrix $\boldsymbol{\Sigma}$ is referred to as the **covariance matrix**, and the matrix $\boldsymbol{\Sigma}^{-1}$ is referred to as the **precision matrix**. The matrices $\boldsymbol{\Sigma} \in \mathbb{R}^{d \times d}$ and $\boldsymbol{\Sigma}^{-1} \in \mathbb{R}^{d \times d}$ are both symmetric and positive definite. To see why this is the case, please see my notes on Probability & Random Processes.

6.3.1 Quadratic Form & Isosurfaces

Notice that the PDF of the anisotropic Gaussian random variable can be expressed as $f_{\boldsymbol{X}}(\boldsymbol{x}) = n(q(\boldsymbol{x}))$, where $n : \mathbb{R} \to \mathbb{R}$ is an exponential function and $q : \mathbb{R}^d \to \mathbb{R}$ is a quadratic function defined such that

$$n(y) = \frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}y\right) \quad \text{and}$$

$$q(\boldsymbol{x}) = (\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}).$$

Given a monotonic function $n : \mathbb{R} \to \mathbb{R}$, the isosurfaces of $n(q(\boldsymbol{x}))$ are the same as those of $q(\boldsymbol{x})$, except they have different isovalues. Therefore, we will focus on the isosurfaces of the quadratic function $q(\boldsymbol{x})$.

Notice that $q(\boldsymbol{x})$ is the quadratic form of the matrix $\boldsymbol{\Sigma}^{-1}$ with respect to the vector $\boldsymbol{x} - \boldsymbol{\mu}$. Because $\boldsymbol{\Sigma}$ is a symmetric matrix, it can be decomposed as $\boldsymbol{\Sigma} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^T$, where $\boldsymbol{V}$ is an orthogonal matrix composed of the eigenvectors of $\boldsymbol{\Sigma}$ and $\boldsymbol{\Lambda}$ is a diagonal matrix whose diagonal elements are the eigenvalues of $\boldsymbol{\Sigma}$. Because $\boldsymbol{\Sigma}$ can be decomposed in this way, $\boldsymbol{\Sigma}^{1/2} = \boldsymbol{V}\boldsymbol{\Lambda}^{1/2}\boldsymbol{V}^T$, where $\boldsymbol{\Lambda}^{1/2}$ is a diagonal matrix whose diagonal elements are the square root of the eigenvalues of $\boldsymbol{\Sigma}$. Now we can see that the isosurfaces of $q(\boldsymbol{x})$ are ellipsoids whose axes are the eigenvectors of $\boldsymbol{\Sigma}$ and whose radii are the square root of the eigenvalues of $\boldsymbol{\Sigma}$. Figure 7.3 provides an example of the isosurfaces for an anisotropic Gaussian PDF. In this example, the axes are shown with respect to $\boldsymbol{x} - \boldsymbol{\mu}$. With respect to the vector $\boldsymbol{x}$, the isocontours would be centered at the mean, $\boldsymbol{\mu}$.

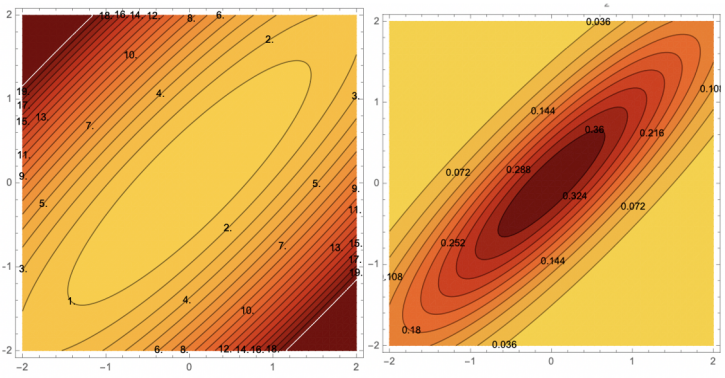

Figure 6.2: The image on the left shows an example of the isocontours of the quadratic function $q(\boldsymbol{x})$ describing an anisotropic Gaussian. The image on the right shows the isocontours of the PDF $f_{\boldsymbol{X}}(\boldsymbol{x}) = n(q(\boldsymbol{x}))$ for the same anisotropic Gaussian. Notice that the isocontours of $f_{\boldsymbol{X}}(\boldsymbol{x})$ are the same as those of $q(\boldsymbol{x})$, except they have different isovalues.

6.4 Isotropic Gaussians

An **isotropic multivariate Gaussian** random variable is a multivariate Gaussian with a covariance matrix that can be expressed as a scalar multiple of the identity matrix (i.e $\boldsymbol{\Sigma} = \sigma^2 \boldsymbol{I_d}$). An isotropic multivariate Gaussian random variable $X \sim N(\boldsymbol{\mu}, \sigma^2 \boldsymbol{I_d})$ with dimension d has the PDF given by

$$f_{\boldsymbol{X}}(\boldsymbol{x}) = \frac{1}{\left(\sqrt{2\pi}\sigma\right)^d} \exp\left(-\frac{||\boldsymbol{x} - \boldsymbol{\mu}||_2^2}{2\sigma^2}\right).$$

This PDF can also be expressed as the product of univariate PDFs:

$$f_{\boldsymbol{X}}(\boldsymbol{x}) = \frac{1}{\left(\sqrt{2\pi}\sigma\right)^d} \exp\left(-\frac{1}{2\sigma^2}\sum_{i=1}^{d}(x_i-\mu_i)^2\right) = \prod_{i=1}^{d}\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(x_i-\mu_i)^2}{2\sigma^2}\right).$$

6.4.1 Quadratic Form & Isosurfaces

Notice that the PDF of the isotropic Gaussian random variable can be expressed as $f_{\boldsymbol{X}}(\boldsymbol{x}) = n(q(\boldsymbol{x}))$, where $n : \mathbb{R} \to \mathbb{R}$ is an exponential function and $q : \mathbb{R}^d \to \mathbb{R}$ is a quadratic function defined such that

$$n(y) = \frac{1}{\left(\sqrt{2\pi}\sigma\right)^d}\exp\left(-\frac{1}{2}y\right) \quad \text{and} \quad q(\boldsymbol{x}) = \frac{||\boldsymbol{x}-\boldsymbol{\mu}||_2^2}{\sigma^2}.$$

Again, we will focus on the isosurfaces of the quadratic function $q(\boldsymbol{x})$. Notice that $q(\boldsymbol{x})$ is the quadratic form of the matrix $\boldsymbol{\Sigma}^{-1} = \frac{1}{\sigma^2}\boldsymbol{I_d}$ with respect to the vector $\boldsymbol{x}-\boldsymbol{\mu}$. In the anisotropic case, the isosurfaces of $q(\boldsymbol{x})$ were ellipsoids whose axes were the eigenvectors of $\boldsymbol{\Sigma}$ and whose radii were the square root of the eigenvalues of $\boldsymbol{\Sigma}$. In the isotropic case, every eigenvalue of $\boldsymbol{\Sigma}^{1/2}$ is equal to σ, so the radii of the isosurfaces will all be equal. Therefore, the isosurfaces of an isotropic Gaussian are spheres with radius σ. Figure 7.2 gives an example of these isosurfaces for an isotropic Gaussian. The axes are shown with respect to $\boldsymbol{x}-\boldsymbol{\mu}$. With respect to $\boldsymbol{x}$, the isocontours would be centered at the mean, $\boldsymbol{\mu}$.

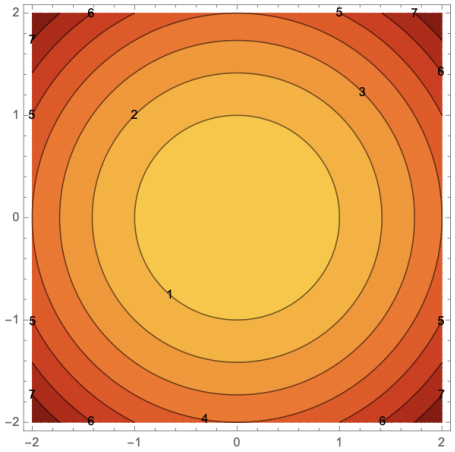

Figure 6.3: This is an example of the isocontours of the quadratic function $q(\boldsymbol{x})$ describing an isotropic Gaussian. The isocontours of the isotropic Gaussian PDF $f_{\boldsymbol{X}}(\boldsymbol{x}) = n(q(\boldsymbol{x}))$ are the same as those of $q(\boldsymbol{x})$ with different isovalues.

Chapter 7

Maximum Likelihood Estimation (MLE)

7.1 Overview of Maximum Likelihood Estimation

Recall that the Bayes decision rule gives us the optimal probabilistic classifier, but it requires that we know the underlying probability distributions of our data. In reality, we rarely know these distributions, but we can use the provided data to estimate them. We will assume a model of the distribution and use maximum-likelihood estimation to estimate the parameters of this model.

7.1.1 Likelihood Estimators

Suppose we have a data set containing n sample points, $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$. We will assume these sample points are realizations of the random variables, $\boldsymbol{X_1}, \ldots, \boldsymbol{X_n}$, which are independent and identically distributed according to the PDF $f_{\boldsymbol{X_i}}(\boldsymbol{x_i}|\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ is the parameter vector we want to estimate. Because the random variables are independent, the **likelihood** of $\boldsymbol{\theta}$ can be expressed as

$$\mathcal{L}(\boldsymbol{\theta}; \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}) := f_{\boldsymbol{X_1 \ldots X_n}}(\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}|\boldsymbol{\theta}) = \prod_{i=1}^{n} f_{\boldsymbol{X_i}}(\boldsymbol{x_i}|\boldsymbol{\theta}).$$

In **maximum likelihood estimation**, we want to find the parameter that maximizes the likelihood by solving the following optimization problem:

$$\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} \ \mathcal{L}(\boldsymbol{\theta}; \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}).$$

Note that the way we defined the likelihood function in terms of probability density functions, it cannot take on negative values. Because the natural log is a continuous, monotonically increasing function defined over positive values, we

can equivalently maximize over the natural log of the likelihood function. We call this objective function the **log likelihood** and express it as

$$l(\boldsymbol{\theta}) := \ln \mathcal{L}(\boldsymbol{\theta}; \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}) = \ln \left(\prod_{i=1}^{n} f_{\boldsymbol{X_i}}(\boldsymbol{x_i}|\boldsymbol{\theta}) \right) = \sum_{i=1}^{n} \ln f_{\boldsymbol{X_i}}(\boldsymbol{x_i}|\boldsymbol{\theta}).$$

Now we can find the optimal parameter that solves the following problem:

$$\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} \ l(\boldsymbol{\theta}).$$

The log likelihood is generally convex, so we can often use the following optimality condition to solve for the maximum likelihood estimate:

$$\nabla_\theta l(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\hat{\boldsymbol{\theta}}} = 0.$$

7.1.2 Bias of Estimators

Let $\boldsymbol{\theta}$ be the true value of the parameter characterizing the underlying distribution of our data. The maximum likelihood estimate is defined in terms of the sample points, $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$. Assume this estimate can be expressed as

$$\hat{\boldsymbol{\theta}} := g(\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}).$$

Let $\hat{\boldsymbol{\Theta}}$ be the random variable defined in terms of random variables as

$$\hat{\boldsymbol{\Theta}} := g(\boldsymbol{X_1}, \ldots, \boldsymbol{X_n}).$$

The **bias** of the estimator is the expected difference of $\hat{\boldsymbol{\Theta}}$ from the true value:

$$\text{bias}(\hat{\boldsymbol{\theta}}) = \mathbb{E}[\hat{\boldsymbol{\Theta}}] - \boldsymbol{\theta}.$$

We say an estimator is **unbiased** if its bias is zero and **biased** if it is non-zero.

7.2 Isotropic Multivariate Gaussians

Often, we assume that our data comes from an isotropic Gaussian distribution such that $\boldsymbol{X_i} \sim N(\boldsymbol{\mu}, \sigma^2 \boldsymbol{I_d})$, and we want to estimate the parameters $\boldsymbol{\mu}$ and σ^2. Recall that an isotropic multivariate Gaussian random variable with d dimensions and the parameters $\boldsymbol{\mu}$ and σ^2 has a PDF of the form

$$f_{\boldsymbol{X_i}}(\boldsymbol{x_i}) = \frac{1}{\left(\sqrt{2\pi}\sigma\right)^d} \exp\left(-\frac{||\boldsymbol{x_i} - \boldsymbol{\mu}||_2^2}{2\sigma^2} \right).$$

For a distribution of this form, we can express the log likelihood as

$$\begin{aligned} l(\boldsymbol{\mu}, \sigma^2) &= \sum_{i=1}^{n} \ln f_{\boldsymbol{X_i}}(\boldsymbol{x_i}) \\ &= \sum_{i=1}^{n} \ln \left(\frac{1}{\left(\sqrt{2\pi}\sigma\right)^d} \exp\left(-\frac{||\boldsymbol{x_i} - \boldsymbol{\mu}||_2^2}{2\sigma^2} \right) \right) \\ &= \sum_{i=1}^{n} \left(-\frac{||\boldsymbol{x_i} - \boldsymbol{\mu}||_2^2}{2\sigma^2} - d \ln \sqrt{2\pi} - d \ln \sigma \right) \end{aligned}$$

7.2.1 Sample Mean

To derive the maximum likelihood estimate for the mean $\boldsymbol{\mu}$, we can first take the gradient of the log likelihood with respect to $\boldsymbol{\mu}$, which gives us

$$\begin{aligned} \nabla_\mu l(\boldsymbol{\mu}, \sigma^2) &= \nabla_\mu \left(\sum_{i=1}^{n} \left(-\frac{||\boldsymbol{x_i} - \boldsymbol{\mu}||_2^2}{2\sigma^2} - d \ln \sqrt{2\pi} - d \ln \sigma \right) \right) \\ &= \sum_{i=1}^{n} \nabla_\mu \left(-\frac{\boldsymbol{x_i}^T \boldsymbol{x_i}}{2\sigma^2} + \frac{\boldsymbol{x_i}^T \boldsymbol{\mu}}{\sigma^2} - \frac{\boldsymbol{\mu}^T \boldsymbol{\mu}}{2\sigma^2} - d \ln \sqrt{2\pi} - d \ln \sigma \right) \\ &= \sum_{i=1}^{n} \left(\frac{\boldsymbol{x_i}}{\sigma^2} - \frac{\boldsymbol{\mu}}{\sigma^2} \right) \end{aligned}$$

Evaluating this expression at $\boldsymbol{\mu} = \hat{\boldsymbol{\mu}}$ and setting it equal to zero, we get

$$\sum_{i=1}^{n} \left(\frac{\boldsymbol{x_i}}{\sigma^2} - \frac{\hat{\boldsymbol{\mu}}}{\sigma^2} \right) = 0 \implies \sum_{i=1}^{n} (\boldsymbol{x_i} - \hat{\boldsymbol{\mu}}) = 0 \implies \sum_{i=1}^{n} \boldsymbol{x_i} - n\hat{\boldsymbol{\mu}} = 0$$

Solving for $\hat{\boldsymbol{\mu}}$ in the equation above, we get the maximum likelihood estimate for the mean, $\boldsymbol{\mu}$, which we also refer to as the **sample mean**:

$$\hat{\boldsymbol{\mu}} = \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{x_i}.$$

7.2.2 Sample Variance

To derive the maximum likelihood estimate for the variance, σ^2, we can first take the derivative of the log likelihood with respect to σ^2, which gives us

$$\begin{aligned} \frac{\partial}{\partial \sigma^2} l(\boldsymbol{\mu}, \sigma^2) &= \frac{\partial}{\partial \sigma^2} \left(\sum_{i=1}^{n} \left(-\frac{||\boldsymbol{x_i} - \boldsymbol{\mu}||_2^2}{2\sigma^2} - d \ln \sqrt{2\pi} - d \ln \sigma \right) \right) \\ &= \sum_{i=1}^{n} \frac{\partial}{\partial \sigma^2} \left(-\frac{||\boldsymbol{x_i} - \boldsymbol{\mu}||_2^2}{2\sigma^2} - d \ln \sqrt{2\pi} - \frac{d}{2} \ln \sigma^2 \right) \\ &= \sum_{i=1}^{n} \left(\frac{||\boldsymbol{x_i} - \boldsymbol{\mu}||_2^2}{2(\sigma^2)^2} - \frac{d}{2\sigma^2} \right) \end{aligned}$$

Evaluating this expression at $\sigma^2 = \hat{\sigma}^2$ and setting it equal to zero, we get

$$\sum_{i=1}^{n} \left(\frac{||\boldsymbol{x_i} - \boldsymbol{\mu}||_2^2}{2(\hat{\sigma}^2)^2} - \frac{d}{2\hat{\sigma}^2} \right) = 0 \implies \sum_{i=1}^{n} \left(\frac{||\boldsymbol{x_i} - \boldsymbol{\mu}||_2^2}{\hat{\sigma}^2} - d \right) = 0 \implies$$

$$\frac{1}{\hat{\sigma}^2} \sum_{i=1}^{n} ||\boldsymbol{x_i} - \boldsymbol{\mu}||_2^2 - nd = 0$$

Solving for $\hat{\sigma}$ in the equation above, we get the maximum likelihood estimate for the variance, σ^2, which we also refer to as the **sample variance**:

$$\hat{\sigma}^2 = \frac{1}{dn} \sum_{i=1}^{n} ||\boldsymbol{x_i} - \boldsymbol{\mu}||^2.$$

Note that we do not actually know the true mean, $\boldsymbol{\mu}$, so we need to use the sample mean $\hat{\boldsymbol{\mu}}$ when computing the sample variance $\hat{\sigma}^2$, which we express as

$$\hat{\sigma}^2 = \frac{1}{dn} \sum_{i=1}^{n} ||\boldsymbol{x_i} - \hat{\boldsymbol{\mu}}||^2.$$

7.2.3 Bias of Estimators

Sample Mean

To compute the bias of the sample mean, we first must compute its expectation.

$$\mathbb{E}[\hat{\boldsymbol{\mu}}] = \mathbb{E}\left[\frac{1}{n} \sum_{i=1}^{n} \boldsymbol{X_i} \right] = \frac{1}{n} \mathbb{E}\left[\sum_{i=1}^{n} \boldsymbol{X_i} \right] = \frac{1}{n} \sum_{i=1}^{n} \mathbb{E}\left[\boldsymbol{X_i} \right] = \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{\mu} = \frac{1}{n}(n\boldsymbol{\mu}) = \boldsymbol{\mu}$$

Now we can see that the bias of the sample mean is given by

$$\text{bias}(\hat{\boldsymbol{\mu}}) = \mathbb{E}[\hat{\boldsymbol{\mu}}] - \boldsymbol{\mu} = \boldsymbol{\mu} - \boldsymbol{\mu} = 0.$$

The bias is zero, so the sample mean is an unbiased estimator.

Sample Variance

To compute the bias of the sample variance, we must compute its expectation.

$$\begin{aligned}
\mathbb{E}[\hat{\sigma}^2] &= \mathbb{E}\left[\frac{1}{dn} \sum_{i=1}^{n} ||\boldsymbol{X_i} - \hat{\boldsymbol{\mu}}||^2 \right] = \frac{1}{dn} \mathbb{E}\left[\sum_{i=1}^{n} ||\boldsymbol{X_i} - \hat{\boldsymbol{\mu}}||^2 \right] \\
&= \frac{1}{dn} \mathbb{E}\left[\sum_{i=1}^{n} \left(\boldsymbol{X_i}^T \boldsymbol{X_i} - 2\boldsymbol{X_i}^T \hat{\boldsymbol{\mu}} + \hat{\boldsymbol{\mu}}^T \hat{\boldsymbol{\mu}} \right) \right] \\
&= \frac{1}{dn} \mathbb{E}\left[\sum_{i=1}^{n} \boldsymbol{X_i}^T \boldsymbol{X_i} - 2 \sum_{i=1}^{n} \boldsymbol{X_i}^T \hat{\boldsymbol{\mu}} + \sum_{i=1}^{n} \hat{\boldsymbol{\mu}}^T \hat{\boldsymbol{\mu}} \right] \\
&= \frac{1}{dn} \mathbb{E}\left[\sum_{i=1}^{n} \boldsymbol{X_i}^T \boldsymbol{X_i} - 2 \sum_{i=1}^{n} \boldsymbol{X_i}^T \hat{\boldsymbol{\mu}} + n\hat{\boldsymbol{\mu}}^T \hat{\boldsymbol{\mu}} \right]
\end{aligned}$$

$$
\begin{aligned}
\mathbb{E}[\hat{\sigma}^2] &= \frac{1}{dn}\mathbb{E}\left[\sum_{i=1}^{n} \boldsymbol{X_i}^T\boldsymbol{X_i} - 2\sum_{i=1}^{n}\boldsymbol{X_i}^T\hat{\boldsymbol{\mu}} + n\hat{\boldsymbol{\mu}}^T\hat{\boldsymbol{\mu}}\right] \\
&= \frac{1}{dn}\mathbb{E}\left[\sum_{i=1}^{n} \boldsymbol{X_i}^T\boldsymbol{X_i} - 2\sum_{i=1}^{n}\boldsymbol{X_i}^T\left(\frac{1}{n}\sum_{j=1}^{n}\boldsymbol{X_j}\right) + n\left(\frac{1}{n}\sum_{i=1}^{n}\boldsymbol{X_i}\right)^T\left(\frac{1}{n}\sum_{j=1}^{n}\boldsymbol{X_j}\right)\right] \\
&= \frac{1}{dn}\mathbb{E}\left[\sum_{i=1}^{n} \boldsymbol{X_i}^T\boldsymbol{X_i} - \frac{2}{n}\sum_{i=1}^{n}\sum_{j=1}^{n}\boldsymbol{X_i}^T\boldsymbol{X_j} + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{n}\boldsymbol{X_i}^T\boldsymbol{X_j}\right] \\
&= \frac{1}{dn}\mathbb{E}\left[\sum_{i=1}^{n} \boldsymbol{X_i}^T\boldsymbol{X_i} - \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{n}\boldsymbol{X_i}^T\boldsymbol{X_j}\right] \\
&= \frac{1}{dn}\sum_{i=1}^{n}\mathbb{E}\left[\boldsymbol{X_i}^T\boldsymbol{X_i}\right] - \frac{1}{dn^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\mathbb{E}\left[\boldsymbol{X_i}^T\boldsymbol{X_j}\right] \\
&= \frac{1}{dn}\sum_{i=1}^{n}\mathbb{E}\left[\boldsymbol{X_i}^T\boldsymbol{X_i}\right] - \frac{1}{dn^2}\sum_{i=1}^{n}\left(\mathbb{E}\left[\boldsymbol{X_i}^T\boldsymbol{X_i}\right] + \sum_{j\neq i}\mathbb{E}\left[\boldsymbol{X_i}^T\boldsymbol{X_j}\right]\right)
\end{aligned}
$$

For a random variable $\boldsymbol{X}$ with covariance matrix $\boldsymbol{\Sigma}$ and mean vector $\boldsymbol{\mu}$, we have $\mathbb{E}\left[\boldsymbol{X}^T\boldsymbol{X}\right] = \mathrm{tr}(\boldsymbol{\Sigma}) + \boldsymbol{\mu}^T\boldsymbol{\mu}$. For two independent random variables $\boldsymbol{X}$ and $\boldsymbol{Y}$ with corresponding means $\boldsymbol{\mu}_X$ and $\boldsymbol{\mu}_Y$, we have $\mathbb{E}\left[\boldsymbol{X}^T\boldsymbol{Y}\right] = \boldsymbol{\mu}_{\boldsymbol{X}}^T\boldsymbol{\mu}_{\boldsymbol{Y}}$. Using these property, the expectation of the sample variance is

$$
\begin{aligned}
\mathbb{E}[\hat{\sigma}^2] &= \frac{1}{dn}\sum_{i=1}^{n}\big(d\sigma^2 + \boldsymbol{\mu}^T\boldsymbol{\mu}\big) - \frac{1}{dn^2}\sum_{i=1}^{n}\left(\big(d\sigma^2 + \boldsymbol{\mu}^T\boldsymbol{\mu}\big) + \sum_{j\neq i}\boldsymbol{\mu}^T\boldsymbol{\mu}\right) \\
&= \frac{1}{dn}n\big(d\sigma^2 + \boldsymbol{\mu}^T\boldsymbol{\mu}\big) - \frac{1}{dn^2}n\Big(\big(d\sigma^2 + \boldsymbol{\mu}^T\boldsymbol{\mu}\big) + (n-1)\boldsymbol{\mu}^T\boldsymbol{\mu}\Big) \\
&= \sigma^2 + \frac{\boldsymbol{\mu}^T\boldsymbol{\mu}}{d} - \frac{\sigma^2}{n} - \frac{\boldsymbol{\mu}^T\boldsymbol{\mu}}{dn} - \frac{\boldsymbol{\mu}^T\boldsymbol{\mu}}{d} + \frac{\boldsymbol{\mu}^T\boldsymbol{\mu}}{dn} \\
&= \sigma^2 - \frac{\sigma^2}{n} = \frac{n-1}{n}\sigma^2
\end{aligned}
$$

Now we can see that the bias of the sample variance is given by

$$
\mathrm{bias}(\hat{\sigma}^2) = \mathbb{E}[\hat{\sigma}^2] - \sigma^2 = \frac{n-1}{n}\sigma^2 - \sigma^2 = -\frac{\sigma^2}{n}.
$$

The bias of the sample variance is not equal to zero, so the sample variance of an isotropic multivariate Gaussian is a biased estimator. If we want to use an unbiased estimator for the variance, then we could use

$$
\hat{\sigma}^2_{unbiased} = \frac{n}{n-1}\hat{\sigma}^2 = \frac{n}{n-1}\cdot\frac{1}{dn}\sum_{i=1}^{n}||\boldsymbol{x_i} - \hat{\boldsymbol{\mu}}||^2 = \frac{1}{d(n-1)}\sum_{i=1}^{n}||\boldsymbol{x_i} - \hat{\boldsymbol{\mu}}||^2
$$

7.3 Anisotropic Multivariate Gaussians

We will also assume that our data comes from an anisotropic Gaussian distribution such that $\boldsymbol{X_i} \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, and we want to estimate the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. Recall that an anisotropic multivariate Gaussian random variable $\boldsymbol{X_i} \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ with dimension d has a PDF of the form

$$f_{\boldsymbol{X_i}}(\boldsymbol{x_i}) = \frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x_i} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i} - \boldsymbol{\mu})\right).$$

For a distribution of this form, we can express the log likelihood as

$$\begin{aligned} l(\boldsymbol{\mu}, \boldsymbol{\Sigma}) &= \sum_{i=1}^{n} \ln f_{\boldsymbol{X_i}}(\boldsymbol{x_i}) \\ &= \sum_{i=1}^{n} \ln\left(\frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x_i} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i} - \boldsymbol{\mu})\right)\right) \\ &= \sum_{i=1}^{n} \left(-\frac{1}{2}(\boldsymbol{x_i} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i} - \boldsymbol{\mu}) - \frac{d}{2}\ln(2\pi) - \frac{1}{2}\ln|\boldsymbol{\Sigma}|\right) \end{aligned}$$

7.3.1 Sample Mean

To derive the maximum likelihood estimate for the mean, $\boldsymbol{\mu}$, we can first take the gradient of the log likelihood with respect to $\boldsymbol{\mu}$, which gives us

$$\begin{aligned} \nabla_\mu l(\boldsymbol{\mu}, \boldsymbol{\Sigma}) &= \nabla_\mu \left(\sum_{i=1}^{n} \left(-\frac{1}{2}(\boldsymbol{x_i} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i} - \boldsymbol{\mu}) - \frac{d}{2}\ln(2\pi) - \frac{1}{2}\ln|\boldsymbol{\Sigma}|\right)\right) \\ &= \sum_{i=1}^{n} \nabla_\mu \left(-\frac{1}{2}\boldsymbol{x_i}^T \boldsymbol{\Sigma}^{-1}\boldsymbol{x_i} + \boldsymbol{x_i}^T \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu} - \frac{1}{2}\boldsymbol{\mu}^T \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu} - \frac{d}{2}\ln(2\pi) - \frac{1}{2}\ln|\boldsymbol{\Sigma}|\right) \\ &= \sum_{i=1}^{n} \left(\boldsymbol{\Sigma}^{-1}\boldsymbol{x_i} - \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}\right) = \sum_{i=1}^{n} \boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i} - \boldsymbol{\mu}) \end{aligned}$$

Evaluating this expression at $\boldsymbol{\mu} = \hat{\boldsymbol{\mu}}$ and setting it equal to zero, we get

$$\sum_{i=1}^{n} \boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i} - \hat{\boldsymbol{\mu}}) = 0 \implies \sum_{i=1}^{n} (\boldsymbol{x_i} - \hat{\boldsymbol{\mu}}) = 0 \implies \sum_{i=1}^{n} \boldsymbol{x_i} - n\hat{\boldsymbol{\mu}} = 0$$

Solving for $\hat{\boldsymbol{\mu}}$ in the equation above, we get the maximum likelihood estimate for the mean, $\boldsymbol{\mu}$, which we also refer to as the **sample mean**:

$$\hat{\boldsymbol{\mu}} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x_i}.$$

7.3.2 Sample Covariance Matrix

To derive the maximum likelihood estimate for the covariance matrix, $\boldsymbol{\Sigma}$, we can first rewrite the the log likelihood with as

$$\begin{aligned} l(\boldsymbol{\mu}, \boldsymbol{\Sigma}) &= \sum_{i=1}^{n} \left(-\frac{1}{2}(\boldsymbol{x_i} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i} - \boldsymbol{\mu}) - \frac{d}{2}\ln(2\pi) - \frac{1}{2}\ln|\boldsymbol{\Sigma}| \right) \\ &= -\frac{1}{2}\sum_{i=1}^{n} \operatorname{tr}\left((\boldsymbol{x_i} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x_i} - \boldsymbol{\mu}) \right) - \frac{dn}{2}\ln(2\pi) + \frac{n}{2}\ln|\boldsymbol{\Sigma}^{-1}| \\ &= -\frac{1}{2}\sum_{i=1}^{n} \operatorname{tr}\left((\boldsymbol{x_i} - \boldsymbol{\mu})^T (\boldsymbol{x_i} - \boldsymbol{\mu})\boldsymbol{\Sigma}^{-1} \right) - \frac{dn}{2}\ln(2\pi) + \frac{n}{2}\ln|\boldsymbol{\Sigma}^{-1}| \end{aligned}$$

Now computing the derivative of this expression with respect to $\boldsymbol{\Sigma}^{-1}$, we get

$$\begin{aligned} \frac{\partial}{\partial \boldsymbol{\Sigma}^{-1}} l(\boldsymbol{\mu}, \boldsymbol{\Sigma}) &= \frac{\partial}{\partial \boldsymbol{\Sigma}^{-1}} \left[-\frac{1}{2}\sum_{i=1}^{n} \operatorname{tr}\left((\boldsymbol{x_i} - \boldsymbol{\mu})^T (\boldsymbol{x_i} - \boldsymbol{\mu})\boldsymbol{\Sigma}^{-1} \right) - \frac{dn}{2}\ln(2\pi) + \frac{n}{2}\ln|\boldsymbol{\Sigma}^{-1}| \right] \\ &= -\frac{1}{2}\sum_{i=1}^{n} \left((\boldsymbol{x_i} - \boldsymbol{\mu})(\boldsymbol{x_i} - \boldsymbol{\mu})^T \right)^T + \frac{n}{2}\boldsymbol{\Sigma}^T \\ &= -\frac{1}{2}\sum_{i=1}^{n} (\boldsymbol{x_i} - \boldsymbol{\mu})(\boldsymbol{x_i} - \boldsymbol{\mu})^T + \frac{n}{2}\boldsymbol{\Sigma} \end{aligned}$$

Evaluating this expression at $\boldsymbol{\Sigma} = \hat{\boldsymbol{\Sigma}}$ and setting it equal to zero, we get

$$-\frac{1}{2}\sum_{i=1}^{n} (\boldsymbol{x_i} - \boldsymbol{\mu})(\boldsymbol{x_i} - \boldsymbol{\mu})^T + \frac{n}{2}\hat{\boldsymbol{\Sigma}} = 0$$

This gives us the maximum likelihood estimate for the covariance matrix, $\boldsymbol{\Sigma}$, which we also refer to as the **sample covariance matrix**:

$$\hat{\boldsymbol{\Sigma}} = \frac{1}{n}\sum_{i=1}^{n} (\boldsymbol{x_i} - \boldsymbol{\mu})(\boldsymbol{x_i} - \boldsymbol{\mu})^T.$$

Note that we do not actually know the true value of $\boldsymbol{\mu}$, so we need to use the sample mean, $\hat{\boldsymbol{\mu}}$, when computing the sample covariance matrix:

$$\hat{\boldsymbol{\Sigma}} = \frac{1}{n}\sum_{i=1}^{n} (\boldsymbol{x_i} - \hat{\boldsymbol{\mu}})(\boldsymbol{x_i} - \hat{\boldsymbol{\mu}})^T.$$

7.3.3 Bias of Estimators

The sample mean for the anisotropic case is that same as for the isotropic case. In the same way that we showed that the sample mean of an isotropic multivariate Gaussian is an unbiased estimator, we can show that the sample mean of an anisotropic multivariate Gaussian is an unbiased estimator.

In the isotropic case, we saw that the expected value of the sample variance is

$$\mathbb{E}[\hat{\sigma}^2] = \frac{n-1}{n}\sigma^2.$$

We also noted that the sample variance of an isotropic multivariate Gaussian is a biased estimator and an unbiased estimator for the variance is

$$\hat{\sigma}^2_{unbiased} = \frac{n}{n-1}\hat{\sigma}^2.$$

In the anisotropic case, the expected value of the sample covariance matrix is

$$\mathbb{E}[\hat{\boldsymbol{\Sigma}}] = \frac{n-1}{n}\boldsymbol{\Sigma}.$$

The sample covariance of an anisotropic multivariate Gaussian is a biased estimator. If we want an unbiased estimator for the covariance, we could use

$$\hat{\boldsymbol{\Sigma}}_{unbiased} = \frac{n}{n-1}\hat{\boldsymbol{\Sigma}} = \frac{n}{n-1}\cdot\frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{x_i}-\hat{\boldsymbol{\mu}})(\boldsymbol{x_i}-\hat{\boldsymbol{\mu}})^T = \frac{1}{n-1}\sum_{i=1}^{n}(\boldsymbol{x_i}-\hat{\boldsymbol{\mu}})(\boldsymbol{x_i}-\hat{\boldsymbol{\mu}})^T.$$

7.3.4 Invertibility of Sample Covariance

Note that because the sample covariance matrix, $\hat{\boldsymbol{\Sigma}}$, is the sum of square dyads $(\boldsymbol{x_i}-\hat{\boldsymbol{\mu}})(\boldsymbol{x_i}-\hat{\boldsymbol{\mu}})^T$ divided by a positive constant, this estimator is guaranteed to be symmetric and positive semidefinite. If we do not have d linearly independent data samples, then the sample covariance matrix will be singular and not positive definite. Note that this occurs whenever $d > n$, but it may occur when $d \leq n$ as well. If we want a non-singular estimator for the covariance matrix, we can instead use the matrix $(\hat{\boldsymbol{\Sigma}} + \epsilon\boldsymbol{I_d})$ as our estimate, where ϵ is positive scalar value which can be made arbitrarily small. To determine an appropriate value for ϵ, we should use hyperparameter tuning. If we do not want to use this modified version of the sample covariance matrix, we can also work with a singular covariance matrix by using its pseudoinverse instead of its inverse.

7.4 Discrete Random Variables

For continuous random variables, such as those described by multivariate Gaussian distributions, we could derive maximum likelihood estimates by taking the gradient of the log likelihood function and setting it equal to zero. For discrete random variables, we cannot use calculus to find the maximum likelihood estimates. Alternatively, we look at the likelihood ratio, which is defined as

$$R(\boldsymbol{\theta}|\boldsymbol{x}) := \frac{\mathcal{L}(\boldsymbol{\theta};\boldsymbol{x})}{\mathcal{L}(\boldsymbol{\theta}-\mathbf{1};\boldsymbol{x})}.$$

If $R(\boldsymbol{\theta}|\boldsymbol{x}) > 1$, the likelihood, $\mathcal{L}(\boldsymbol{\theta};\boldsymbol{x})$, increases as $\boldsymbol{\theta}$ increases. Conversely, if $R(\boldsymbol{\theta}|\boldsymbol{x}) < 1$, the likelihood, $\mathcal{L}(\boldsymbol{\theta};\boldsymbol{x})$, decreases as $\boldsymbol{\theta}$ increases. Therefore, the maximum likelihood estimator, $\hat{\boldsymbol{\theta}}$, must satisfy $R(\hat{\boldsymbol{\theta}}|\boldsymbol{x}) = 1$. When solving this equation, if we find that $\hat{\boldsymbol{\theta}}$ is not an integer, then the greatest integer less than the given solution is the true maximum likelihood estimate.

Chapter 8

Gaussian Discriminant Analysis (GDA)

8.1 Overview of Gaussian Discriminant Analysis

As we did previously, consider a system of d-dimensional feature vectors and corresponding labels. Let $\boldsymbol{X}$ be a random feature vector with a random label Y. Assume that the range of $\boldsymbol{X}$ is $\mathcal{X}$, which is some subset of $\mathbb{R}^d$. In addition, assume that the range of Y is C, which is a discrete set of possible classes.

Suppose that we want to determine to which class a given test point, $\boldsymbol{x}$, belongs using Bayes decision rule. Recall that, for a continuous feature space, if we assume a symmetric loss function, the Bayes optimal decision rule is

$$r^*(\boldsymbol{x}) = \arg\max_{c \in C} \; f_{\boldsymbol{X}|Y}(\boldsymbol{x}|c) p_Y(c).$$

Recall that generative models assume a form of the likelihood distributions, $f_{\boldsymbol{X}|Y}(\boldsymbol{x}|c)$, and prior probabilities, $p_Y(c)$, for each class $c \in C$ and use data to fit the parameters of these probability models. They then use these estimates to approximate Bayes decision rule. **Gaussian discriminant analysis (GDA)** is a generative model, in which the likelihood distributions are assumed to be Gaussian and the prior probabilities are assumed to be categorical.

8.2 Quadratic Discriminant Analysis (QDA)

We consider two classes of Gaussian discriminant analysis (GDA): **quadratic discriminant analysis (QDA)** and **linear discriminant analysis (LDA)**. In QDA, the objective of the decision function is quadratic, while in LDA, the objective is linear. LDA is simply a variant of QDA, so we first discuss QDA.

8.2.1 Isotropic Multivariate Gaussians

Recall that GDA is a generative model, in which the likelihood distributions are assumed to be Gaussian and the prior probabilities are assumed to be categorical. For now, assume that the likelihood distribution, $f_{\boldsymbol{X}|Y}(\boldsymbol{x}|c)$, is an isotropic multivariate Gaussian distribution and is unique for each class c.

Known Distributions

Suppose we know the actual likelihood distribution $f_c(\boldsymbol{x}) := f_{\boldsymbol{X}|Y}(\boldsymbol{x}|c)$ and prior probability $\pi_c := p_Y(c)$ for each class c. We should classify $\boldsymbol{x}$ such that

$$r(\boldsymbol{x}) = \underset{c \in C}{\arg\max}\ f_c(\boldsymbol{x})\pi_c.$$

If the sample points in each class c are drawn from an isotropic multivariate Gaussian distribution with mean $\boldsymbol{\mu_c}$ and covariance matrix $\sigma_c^2 \boldsymbol{I_d}$, then

$$f_c(\boldsymbol{x}) = \frac{1}{\left(\sqrt{2\pi}\sigma_c\right)^d} \exp\left(-\frac{||\boldsymbol{x} - \boldsymbol{\mu_c}||_2^2}{2\sigma_c^2}\right).$$

For this likelihood distribution, our classifier can be expressed as

$$r(\boldsymbol{x}) = \underset{c \in C}{\arg\max}\ \frac{1}{\left(\sqrt{2\pi}\sigma_c\right)^d} \exp\left(-\frac{||\boldsymbol{x} - \boldsymbol{\mu_c}||_2^2}{2\sigma_c^2}\right)\pi_c.$$

Removing the constant term that does not depend on c, this problem becomes

$$r(\boldsymbol{x}) = \underset{c \in C}{\arg\max}\ \frac{1}{\sigma_c^d} \exp\left(-\frac{||\boldsymbol{x} - \boldsymbol{\mu_c}||_2^2}{2\sigma_c^2}\right)\pi_c.$$

Because the natural log is monotonically increasing, we can equivalently choose

$$\begin{aligned} r(\boldsymbol{x}) &= \underset{c \in C}{\arg\max}\ \ln\left(\frac{1}{\sigma_c^d} \exp\left(-\frac{||\boldsymbol{x} - \boldsymbol{\mu_c}||_2^2}{2\sigma_c^2}\right)\pi_c\right) \\ &= \underset{c \in C}{\arg\max}\left(-\frac{||\boldsymbol{x} - \boldsymbol{\mu_c}||_2^2}{2\sigma_c^2} - d\ln\sigma_c + \ln\pi_c\right). \end{aligned}$$

Notice that the objective function of this problem is quadratic in $\boldsymbol{x}$ as desired.

Unknown Distributions

If we do not know the actual likelihood distribution, $f_c(\boldsymbol{x})$, and the actual prior probability distribution, π_c, for each class c, then we can use the maximum likelihood estimations (see section 7.2). Suppose we have a data set containing n sample points, $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$, with corresponding labels, $y_1, \ldots, y_n$. If n_c is the number of sample points in class c, the estimated conditional mean, $\hat{\boldsymbol{\mu}}_{\boldsymbol{c}}$, conditional variance, $\hat{\sigma}_c^2$, and prior probability, $\hat{\pi}_c$, for class c are

$$\hat{\boldsymbol{\mu}}_{\boldsymbol{c}} = \frac{1}{n_c}\sum_{i:y_i=c} \boldsymbol{x_i}, \qquad \hat{\sigma}_c^2 = \frac{1}{dn_c}\sum_{i:y_i=c} ||\boldsymbol{x_i} - \hat{\boldsymbol{\mu}}_{\boldsymbol{c}}||^2, \qquad \text{and} \qquad \hat{\pi}_c = \frac{n_c}{n}.$$

The decision function for QDA with isotropic multivariate Gaussians is then

$$r(\boldsymbol{x}) = \arg\max_{c \in C} \left(-\frac{||\boldsymbol{x} - \hat{\boldsymbol{\mu}}_{\boldsymbol{c}}||_2^2}{2\hat{\sigma}_c^2} - d \ln \hat{\sigma}_c + \ln \hat{\pi}_c \right).$$

8.2.2 Anisotropic Multivariate Gaussians

Once again, we assume Gaussian likelihood distributions and categorical prior probabilities. We now assume that the likelihood distribution, $f_{\boldsymbol{X}|Y}(\boldsymbol{x}|c)$, is an anisotropic multivariate Gaussian distribution and is unique for each class c.

Known Distributions

Assume we know the actual likelihood distribution $f_c(\boldsymbol{x}) := f_{\boldsymbol{X}|Y}(\boldsymbol{x}|c)$ and prior probability $\pi_c := p_Y(c)$ for each class c. We should classify $\boldsymbol{x}$ such that

$$r(\boldsymbol{x}) = \arg\max_{c \in C} \ f_c(\boldsymbol{x})\pi_c.$$

If the sample points in each class c are drawn from an anisotropic multivariate Gaussian distribution with mean $\boldsymbol{\mu_c}$ and covariance matrix $\boldsymbol{\Sigma_c}$, then

$$f_c(\boldsymbol{x}) = \frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma_c}|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu_c})^T \boldsymbol{\Sigma_c}^{-1}(\boldsymbol{x} - \boldsymbol{\mu_c}) \right).$$

For this likelihood distribution, the Bayes decision rule can be expressed as

$$r(\boldsymbol{x}) = \arg\max_{c \in C} \ \frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma_c}|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu_c})^T \boldsymbol{\Sigma_c}^{-1}(\boldsymbol{x} - \boldsymbol{\mu_c}) \right) \pi_c.$$

Removing the constant term that does not depend on c, this problem becomes

$$r(\boldsymbol{x}) = \arg\max_{c \in C} \ \frac{1}{|\boldsymbol{\Sigma_c}|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu_c})^T \boldsymbol{\Sigma_c}^{-1}(\boldsymbol{x} - \boldsymbol{\mu_c}) \right) \pi_c.$$

Because the natural log is monotonically increasing, we can equivalently choose

$$\begin{aligned} r(\boldsymbol{x}) &= \arg\max_{c \in C} \ \ln\left(\frac{1}{|\boldsymbol{\Sigma}_c|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu_c})^T \boldsymbol{\Sigma_c}^{-1}(\boldsymbol{x} - \boldsymbol{\mu_c}) \right) \pi_c \right) \\ &= \arg\max_{c \in C} \ \left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu_c})^T \boldsymbol{\Sigma_c}^{-1}(\boldsymbol{x} - \boldsymbol{\mu_c}) - \frac{1}{2} \ln |\boldsymbol{\Sigma_c}| + \ln \pi_c \right). \end{aligned}$$

Notice that the objective function of this problem is quadratic in $\boldsymbol{x}$ as desired.

Unknown Distributions

If we do not know the actual likelihood distribution, $f_c(\boldsymbol{x})$, and the actual prior probability distribution, π_c, for each class c, then we can use the maximum likelihood estimations (see section 7.3). Suppose we have a data set containing

n sample points, $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$, with corresponding labels, $y_1, \ldots, y_n$. If n_c is the number of sample points in class c, the estimated conditional mean, $\hat{\boldsymbol{\mu}}_{\boldsymbol{c}}$, conditional covariance, $\hat{\boldsymbol{\Sigma}}_{\boldsymbol{c}}$, and prior probability, $\hat{\pi}_c$, for class c are

$$\hat{\boldsymbol{\mu}}_{\boldsymbol{c}} = \frac{1}{n_c} \sum_{i:y_i=c} \boldsymbol{x_i}, \qquad \hat{\boldsymbol{\Sigma}}_{\boldsymbol{c}} = \frac{1}{n_c} \sum_{i:y_i=c} (\boldsymbol{x_i} - \hat{\boldsymbol{\mu}}_{\boldsymbol{c}})(\boldsymbol{x_i} - \hat{\boldsymbol{\mu}}_{\boldsymbol{c}})^T, \quad \text{and} \quad \hat{\pi}_c = \frac{n_c}{n}.$$

The decision function for QDA with anisotropic multivariate Gaussians is then

$$r(\boldsymbol{x}) = \underset{c \in C}{\arg\max} \left(-\frac{1}{2}(\boldsymbol{x} - \hat{\boldsymbol{\mu}}_{\boldsymbol{c}})^T \hat{\boldsymbol{\Sigma}}_{\boldsymbol{c}}^{-1} (\boldsymbol{x} - \hat{\boldsymbol{\mu}}_{\boldsymbol{c}}) - \frac{1}{2} \ln |\hat{\boldsymbol{\Sigma}}_{\boldsymbol{c}}| + \ln \hat{\pi}_c \right).$$

As discussed in section 7.3.4, the sample covariance, $\hat{\boldsymbol{\Sigma}}_{\boldsymbol{c}}$, is always positive semidefinite, but it is not always positive definite and thus may not be invertible. There are multiple ways to deal with this issue, as discussed previously.

8.3 Linear Discriminant Analysis

Linear discriminant analysis (LDA) is the second category of Gaussian discriminant analysis (GDA) and is a variant of quadratic discriminant analysis (QDA). Previously, we assumed that the likelihood distribution, $f_{\boldsymbol{X}|Y}(\boldsymbol{x}|c)$, is a multivariate Gaussian distribution and is unique for each class c. The difference between LDA and QDA is that in LDA, we have an additional assumption that all of the Gaussian likelihood distributions have the same covariance matrix.

8.3.1 Isotropic Multivariate Gaussians

As in QDA, we assume that the likelihood distributions are Gaussian and that the prior probabilities are categorical. Let us first assume that the likelihood distribution, $f_{\boldsymbol{X}|Y}(\boldsymbol{x}|c)$, is an isotropic multivariate Gaussian distribution.

Known Distributions

In QDA, we found that if the sample points in each class c are drawn from an isotropic multivariate Gaussian distribution with mean $\boldsymbol{\mu_c}$ and covariance matrix $\sigma_c^2 \boldsymbol{I_d}$, then we can express the decision function as

$$r(\boldsymbol{x}) = \underset{c \in C}{\arg\max} \left(-\frac{||\boldsymbol{x} - \boldsymbol{\mu_c}||_2^2}{2\sigma_c^2} - d \ln \sigma_c + \ln \pi_c \right).$$

In LDA, we assume all the likelihood distributions have the same covariance, so $\sigma_c^2 = \sigma^2$ for all c. This condition allows us to express the decision function as

$$r(\boldsymbol{x}) = \underset{c \in C}{\arg\max} \left(-\frac{||\boldsymbol{x} - \boldsymbol{\mu_c}||_2^2}{2\sigma^2} - d \ln \sigma + \ln \pi_c \right).$$

By removing terms that do not depend on c, we equivalently express this as

$$\begin{aligned} r(\boldsymbol{x}) &= \arg\max_{c \in C} \left(-\frac{||\boldsymbol{x} - \boldsymbol{\mu_c}||_2^2}{2\sigma^2} + \ln \pi_c \right) \\ &= \arg\max_{c \in C} \left(-\frac{\boldsymbol{x}^T\boldsymbol{x} - 2\boldsymbol{\mu_c}^T\boldsymbol{x} + \boldsymbol{\mu_c}^T\boldsymbol{\mu_c}}{2\sigma^2} + \ln \pi_c \right) \\ &= \arg\max_{c \in C} \left(\frac{\boldsymbol{\mu_c}^T\boldsymbol{x}}{\sigma^2} - \frac{||\boldsymbol{\mu_c}||_2^2}{2\sigma^2} + \ln \pi_c \right). \end{aligned}$$

Notice that the objective function of this problem is linear in $\boldsymbol{x}$ as desired.

Unknown Distributions

If we do not know the actual likelihood distribution, $f_c(\boldsymbol{x})$, and the actual prior probability distribution, π_c, for each class c, then we can use the maximum likelihood estimations (see section 7.2). Suppose we have a data set containing n sample points, $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$, with corresponding labels, $y_1, \ldots, y_n$. If n_c is the number of sample points in class c, the estimated conditional mean, $\hat{\boldsymbol{\mu}}_{\boldsymbol{c}}$, conditional variance, $\hat{\sigma}^2$, and prior probability, $\hat{\pi}_c$, for class c are

$$\hat{\boldsymbol{\mu}}_{\boldsymbol{c}} = \frac{1}{n_c} \sum_{i:y_i=c} \boldsymbol{x_i}, \qquad \hat{\sigma}^2 = \frac{1}{dn} \sum_c \sum_{i:y_i=c} ||\boldsymbol{x_i} - \hat{\boldsymbol{\mu}}_{\boldsymbol{c}}||^2, \qquad \text{and} \qquad \hat{\pi}_c = \frac{n_c}{n}.$$

The decision function for LDA with isotropic multivariate Gaussians is then

$$r(\boldsymbol{x}) = \arg\max_{c \in C} \left(\frac{\hat{\boldsymbol{\mu}}_{\boldsymbol{c}}^T\boldsymbol{x}}{\hat{\sigma}^2} - \frac{||\hat{\boldsymbol{\mu}}_{\boldsymbol{c}}||_2^2}{2\hat{\sigma}^2} + \ln \hat{\pi}_c \right).$$

8.3.2 Anisotropic Multivariate Gaussians

Once again, we assume Gaussian likelihood distributions and categorical prior probabilities. We will now also assume that the likelihood distribution, $f_{\boldsymbol{X}|Y}(\boldsymbol{x}|c)$, is an anisotropic multivariate Gaussian distribution for each class c.

Known Distributions

In QDA, we found that if the sample points in each class c are drawn from an anisotropic multivariate Gaussian distribution with mean $\boldsymbol{\mu_c}$ and covariance matrix $\boldsymbol{\Sigma_c}$, then we can express the decision function as

$$r(\boldsymbol{x}) = \arg\max_{c \in C} \left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu_c})^T \boldsymbol{\Sigma_c}^{-1} (\boldsymbol{x} - \boldsymbol{\mu_c}) - \frac{1}{2} \ln |\boldsymbol{\Sigma_c}| + \ln \pi_c \right).$$

In LDA, we assume all the likelihood distributions have the same covariance, so $\boldsymbol{\Sigma_c} = \boldsymbol{\Sigma}$ for all c. This condition allows us to express the decision function as

$$r(\boldsymbol{x}) = \arg\max_{c \in C} \left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu_c})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x} - \boldsymbol{\mu_c}) - \frac{1}{2} \ln |\boldsymbol{\Sigma}| + \ln \pi_c \right).$$

By removing terms that do not depend on c, we equivalently express this as

$$\begin{aligned} r(\boldsymbol{x}) &= \arg\max_{c \in C} \left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu_c})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x} - \boldsymbol{\mu_c}) + \ln \pi_c \right) \\ &= \arg\max_{c \in C} \left(-\frac{1}{2}\boldsymbol{x}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{x} + \boldsymbol{\mu_c}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{x} - \frac{1}{2}\boldsymbol{\mu_c}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu_c} + \ln \pi_c \right) \\ &= \arg\max_{c \in C} \left(\boldsymbol{\mu_c}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{x} - \frac{1}{2}\boldsymbol{\mu_c}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu_c} + \ln \pi_c \right). \end{aligned}$$

Notice that the objective function of this problem is linear in $\boldsymbol{x}$ as desired.

Unknown Distributions

If we do not know the actual likelihood distribution, $f_c(\boldsymbol{x})$, and the actual prior probability distribution, π_c, for each class c, then we can use the maximum likelihood estimations (see section 7.3). Suppose we have a data set containing n sample points, $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$, with corresponding labels, $y_1, \ldots, y_n$. If n_c is the number of sample points in class c, the estimated conditional mean, $\hat{\boldsymbol{\mu}}_{\boldsymbol{c}}$, pooled within-class covariance matrix $\hat{\boldsymbol{\Sigma}}$, and prior probability, $\hat{\pi}_c$, for class c are

$$\hat{\boldsymbol{\mu}}_{\boldsymbol{c}} = \frac{1}{n_c} \sum_{i:y_i=c} \boldsymbol{x_i}, \qquad \hat{\boldsymbol{\Sigma}} = \frac{1}{n} \sum_{c} \sum_{i:y_i=c} (\boldsymbol{x_i} - \hat{\boldsymbol{\mu}}_{\boldsymbol{c}})(\boldsymbol{x_i} - \hat{\boldsymbol{\mu}}_{\boldsymbol{c}})^T, \qquad \text{and} \qquad \hat{\pi}_c = \frac{n_c}{n}.$$

The decision function for LDA with anisotropic multivariate Gaussians is then

$$r(\boldsymbol{x}) = \arg\max_{c \in C} \left(\hat{\boldsymbol{\mu}}_{\boldsymbol{c}}^T \hat{\boldsymbol{\Sigma}}^{-1} \boldsymbol{x} - \frac{1}{2}\hat{\boldsymbol{\mu}}_{\boldsymbol{c}}^T \hat{\boldsymbol{\Sigma}}^{-1} \hat{\boldsymbol{\mu}}_{\boldsymbol{c}} + \ln \hat{\pi}_c \right).$$

As discussed in section 7.3.4, the sample covariance, $\hat{\boldsymbol{\Sigma}}$, is always positive semidefinite, but it is not always positive definite and thus may not be invertible. There are multiple ways to deal with this issue, as discussed previously.

8.4 Comparison of LDA & QDA

LDA and QDA are similar classification methods with important distinctions:

1. Both LDA and QDA perform well when data can only support simple decision boundaries because Gaussian models provide stable estimates. For some data sets, QDA works better, and for some, LDA works better. We can use validation to choose which type of classifier to use.

2. For a data set with only two classes, LDA has $d + 1$ parameters, while QDA has $\frac{1}{2}d(d + 3) + 1$ parameters. Because QDA has more parameters, QDA is more likely to overfit and LDA is more likely to underfit.

3. In general, LDA provides linear decision boundaries, while QDA provides quadratic decision boundaries. However, with added features, LDA can give nonlinear boundaries and QDA can give non-quadratic boundaries. Examples of the types of decision boundaries that can be obtained by LDA and QDA with different features are shown in figure 8.1.

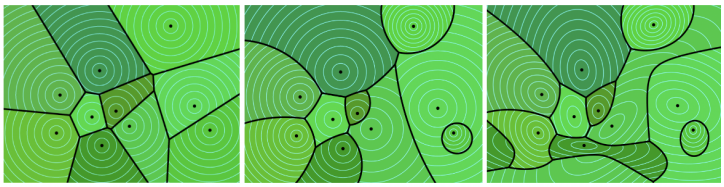

Figure 8.1: On the left, LDA was used to produce linear decision boundaries. In the center, QDA was used to produce quadratic decision boundaries. On the right, QDA with additional features was used to produce non-quadratic decision boundaries.

Chapter 9

Regression

9.1 Overview of Regression

Recall from section 2.1.1 that, within supervised learning, there are two types of machine learning problems: classification and regression. Up until now, we assumed that our data had categorical labels, and we performed **classification** to predict the class of unseen data. Now we will assume that our data has quantitative labels and will use **regression** to predict the numerical value associated with unseen data. We will generally assume that we are working with n samples points $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ with corresponding labels $y_1, \ldots, y_n$, where $\boldsymbol{x_i} \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$. There are three main components of any regression problem:

1. **Regression function**

   For a data point $\boldsymbol{x}$ and parameters $\boldsymbol{p}$, the regression function has the form $h(\boldsymbol{x}; \boldsymbol{p})$. The function h is also commonly referred to as the **hypothesis**. The parameter vector $\boldsymbol{p} \in \mathbb{R}^{d+1}$ is often expressed as a weight vector, $\boldsymbol{w} \in \mathbb{R}^d$, plus a bias term, $\alpha \in \mathbb{R}$. Some common regression functions are:

   (a) **Linear** – $h(\boldsymbol{x}; \boldsymbol{w}, \alpha) = \boldsymbol{w}^T\boldsymbol{x} + \alpha$

   (b) **Polynomial** – $h(\boldsymbol{x}; \boldsymbol{w}, \alpha) = p(\boldsymbol{x})$, where $p : \mathbb{R}^d \to \mathbb{R}$ is a polynomial

   (c) **Logistic** – $h(\boldsymbol{x}; \boldsymbol{w}, \alpha) = s(\boldsymbol{w}^T\boldsymbol{x} + \alpha)$, where $s : \mathbb{R} \to [0, 1]$ is the logistic function defined such that $s(\gamma) = 1/(1 + e^{-\gamma})$

2. **Loss function**

   For a data point $\boldsymbol{x}$ with the true label y, we use the regression function to make a prediction $z = h(\boldsymbol{x}; \boldsymbol{p})$. We express the loss function for a prediction z and true label y as $\mathcal{L}(z, y)$. Some common loss functions are:

   (a) **Squared error** – $\mathcal{L}(z, y) = (z - y)^2$

   (b) **Absolute error** – $\mathcal{L}(z, y) = |z - y|$

(c) **Logistic loss/cross-entropy** – $\mathcal{L}(z, y) = -y\ln(z) - (1-y)\ln(1-z)$, where the labels and predictions must satisfy $y \in [0, 1]$ and $z \in (0, 1)$

3. **Cost function**

   The loss function, $\mathcal{L}(z, y)$, is defined for a single data point $\boldsymbol{x}$ with true label y. We define the cost function, $J(h)$, for a set of data points $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$, with corresponding labels $y_1, \ldots, y_n$. Some common cost functions are:

   (a) **Mean loss** – $J(h) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\big(h(\boldsymbol{x_i}), y_i\big)$

   (b) **Max loss** – $J(h) = \max_{i\in\{1,\ldots,n\}} \mathcal{L}\big(h(\boldsymbol{x_i}), y_i\big)$

   (c) **Weighted sum** – $J(h) = \sum_{i=1}^{n} \omega_i \mathcal{L}\big(h(\boldsymbol{x_i}), y_i\big)$, where $\omega_1, \ldots, \omega_n$ are weight terms that we select based on how much we want to penalize the loss for different data points

9.2 Linear Least Squares Regression

Linear least squares regression uses the linear regression function, squared error loss function, and mean loss cost function. Combining all these components, linear least squares regression is defined by the following problem:

$$\min_{\boldsymbol{w}\in\mathbb{R}^d, \alpha\in\mathbb{R}} \frac{1}{n}\sum_{i=1}^{n}\big(\boldsymbol{w}^T\boldsymbol{x_i} + \alpha - y_i\big)^2.$$

We can remove the constant factor and equivalently express this problem as

$$\min_{\boldsymbol{w}\in\mathbb{R}^d, \alpha\in\mathbb{R}} \sum_{i=1}^{n}\big(\boldsymbol{w}^T\boldsymbol{x_i} + \alpha - y_i\big)^2.$$

Recall from our discussion of the perceptron algorithm with bias in section 3.3.2 that we can express the linear function $h(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + \alpha$ as $h(\tilde{\boldsymbol{x}}) = \tilde{\boldsymbol{w}}^T\tilde{\boldsymbol{x}}$, where $\tilde{\boldsymbol{x}}$ is the vector $\boldsymbol{x}$ augmented with one and $\tilde{\boldsymbol{w}}$ is the vector $\boldsymbol{w}$ augmented with α. Therefore, we can equivalently express the linear least squares problem as

$$\min_{\tilde{\boldsymbol{w}}\in\mathbb{R}^{d+1}} \sum_{i=1}^{n}\big(\tilde{\boldsymbol{w}}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}} - y_i\big)^2.$$

Let $\boldsymbol{X}$ be the $n \times (d+1)$ **design matrix** whose ith row is the transpose of the ith sample point, $\boldsymbol{x_i} \in \mathbb{R}^d$, augmented with one. Let $\boldsymbol{y}$ be an n-dimensional vector whose ith element, y_i, is the label corresponding to sample point $\boldsymbol{x_i}$. Finally, let $\boldsymbol{w} \in \mathbb{R}^{d+1}$ now be the previous weight vector augmented with the bias term, α. This allows us to express the linear least squares problem as

$$\min_{\boldsymbol{w}\in\mathbb{R}^{d+1}} ||\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}||_2^2.$$

9.2.1 Optimal Solution

Because the objective is a convex quadratic function, we can find the optimal solution, $\hat{\boldsymbol{w}}$, by computing the gradient of the objective function with respect to $\boldsymbol{w}$ and setting it equal to zero. The gradient of the objective function is

$$\begin{aligned}\nabla_w J(\boldsymbol{w}) &= \nabla_w ||\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}||_2^2 \\ &= \nabla_w \Big(\big(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\big)^T \big(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\big) \Big) \\ &= \nabla_w \big(\boldsymbol{w}^T \boldsymbol{X}^T \boldsymbol{X}\boldsymbol{w} - 2\boldsymbol{y}^T \boldsymbol{X}\boldsymbol{w} + \boldsymbol{y}^T\boldsymbol{y} \big) \\ &= 2\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} - 2\boldsymbol{X}^T\boldsymbol{y}.\end{aligned}$$

Evaluating the gradient at $\boldsymbol{w} = \hat{\boldsymbol{w}}$ and setting it equal to zero, we have

$$2\boldsymbol{X}^T\boldsymbol{X}\hat{\boldsymbol{w}} - 2\boldsymbol{X}^T\boldsymbol{y} = 0$$
$$\boldsymbol{X}^T\boldsymbol{X}\hat{\boldsymbol{w}} = \boldsymbol{X}^T\boldsymbol{y}$$

By definition of the range space, the vector $\boldsymbol{X}^T\boldsymbol{y}$ is in the range of $\boldsymbol{X}^T$. By the fundamental theorem of linear algebra, the range of $\boldsymbol{X}^T$ is the same as the range of $\boldsymbol{X}^T\boldsymbol{X}$. Therefore, the vector $\boldsymbol{X}^T\boldsymbol{y}$ is in the range of $\boldsymbol{X}^T\boldsymbol{X}$. This tells us that the equation $\boldsymbol{X}^T\boldsymbol{X}\hat{\boldsymbol{w}} = \boldsymbol{X}^T\boldsymbol{y}$ always has at least one solution. If the data samples span the feature space, $\mathbb{R}^d$, then the rank of $\boldsymbol{X}$ is $d+1$. Under this condition, the $(d+1) \times (d+1)$ matrix $\boldsymbol{X}^T\boldsymbol{X}$ has rank $d+1$, so it is invertible and the unique solution of the linear least squares equation is given by

$$\hat{\boldsymbol{w}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}.$$

If the data samples do not span all of the feature space, then the rank of $\boldsymbol{X}$ is less than $d+1$. Under this condition, $\boldsymbol{X}^T\boldsymbol{X}$ also has rank less than $d+1$, so it is singular. The problem is underconstrained and the linear equation $\boldsymbol{X}^T\boldsymbol{X}\hat{\boldsymbol{w}} = \boldsymbol{X}^T\boldsymbol{y}$ has infinitely many solutions. The minimum norm solution is

$$\hat{\boldsymbol{w}} = \boldsymbol{X}^\dagger\boldsymbol{y},$$

where $\boldsymbol{X}^\dagger$ is the pseudoinverse of the design matrix, $\boldsymbol{X}$. For more information on the solution to the linear equation, please see my linear algebra notes. Given the optimal weight vector, $\hat{\boldsymbol{w}}$, the predicted label of the augmented sample point $\tilde{\boldsymbol{x}}_{\boldsymbol{i}}$ is given by $\hat{y}_i = \hat{\boldsymbol{w}}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}}$. We can combine all these predictions into the vector

$$\hat{\boldsymbol{y}} = \boldsymbol{X}\hat{\boldsymbol{w}} = \boldsymbol{X}\boldsymbol{X}^\dagger\boldsymbol{y} =: \boldsymbol{H}\boldsymbol{y}.$$

We often refer to the $n \times n$ matrix $\boldsymbol{H} := \boldsymbol{X}\boldsymbol{X}^\dagger$ as the **hat matrix**.

9.2.2 Advantages & Disadvantages

Linear least squares regression is useful because the decision function is easily computed by solving a linear equation. Another benefit of linear least squares regression is that when $\boldsymbol{X}^T\boldsymbol{X}$ is invertible, we find a unique stable solution. However, linear least squares regression gives multiple solutions if the matrix $\boldsymbol{X}^T\boldsymbol{X}$ is singular. It also does not perform well when the data is very noisy because errors are squared, making the decision function very sensitive to outliers.

9.2.3 Bias-Variance Decomposition

Assume that the sample points come from an unknown probability distribution D and that the labels are the sum of an unknown deterministic function of the samples, $g(\boldsymbol{x_i})$, and random noise, ϵ_i, which comes from some other unknown probability distribution D'. We can express this assumption mathematically as

$$y_i = g(\boldsymbol{x_i}) + \epsilon_i,\ \boldsymbol{x_i} \sim D,\ \epsilon_i \sim D'.$$

Recall that the goal of regression problems is to find a hypothesis h that estimates the unknown function g. We want to analyze the risk associated with this hypothesis. Consider an arbitrary test point $\boldsymbol{z} \in \mathbb{R}^d$ in the feature space with the label $\gamma = g(\boldsymbol{z}) + \epsilon$, where $\epsilon \sim D'$. In general, for some arbitrary loss function L, the risk associated with the hypothesis function is given by

$$R(h) = \mathbb{E}\left[L(h(\boldsymbol{z}), \gamma)\right].$$

When performing linear least squares regression, we use the squared error loss function, which gives us the following risk function:

$$\begin{aligned} R(h) &= \mathbb{E}\left[\left(h(\boldsymbol{z}) - \gamma\right)^2\right] \\ &= \mathbb{E}\left[h(\boldsymbol{z})^2 + \gamma^2 - 2\gamma h(\boldsymbol{z})\right] \\ &= \mathbb{E}\left[h(\boldsymbol{z})^2\right] + \mathbb{E}\left[\gamma^2\right] - 2\mathbb{E}\left[\gamma h(\boldsymbol{z})\right]. \end{aligned}$$

Because the hypothesis, $h(\boldsymbol{z})$, depends only on the training data, $\boldsymbol{x_1} \ldots, \boldsymbol{x_n}$, and the label, γ, depends only on the test point, $\boldsymbol{z}$, these two random variables are independent. As a result, the risk can be expressed as

$$\begin{aligned} R(h) &= \mathbb{E}\left[h(\boldsymbol{z})^2\right] + \mathbb{E}\left[\gamma^2\right] - 2\mathbb{E}\left[\gamma\right]\mathbb{E}\left[h(\boldsymbol{z})\right] \\ &= \mathrm{Var}(h(\boldsymbol{z})) + \mathbb{E}\left[h(\boldsymbol{z})\right]^2 + \mathrm{Var}(\gamma) + \mathbb{E}\left[\gamma\right]^2 - 2\mathbb{E}\left[\gamma\right]\mathbb{E}\left[h(\boldsymbol{z})\right] \\ &= \Big(\mathbb{E}\left[h(\boldsymbol{z})\right] - \mathbb{E}\left[\gamma\right]\Big)^2 + \mathrm{Var}(h(\boldsymbol{z})) + \mathrm{Var}(\gamma). \end{aligned}$$

Recall that $\gamma = g(\boldsymbol{z}) + \epsilon$. Because g is a deterministic function, $\mathrm{Var}(\gamma) = \mathrm{Var}(\epsilon)$. If we assume that ϵ has zero mean, then $\mathbb{E}\left[\gamma\right] = g(\boldsymbol{z})$. Under these conditions,

$$R(h) = \Big(\mathbb{E}\left[h(\boldsymbol{z})\right] - g(\boldsymbol{z})\Big)^2 + \mathrm{Var}(h(\boldsymbol{z})) + \mathrm{Var}(\epsilon).$$

This is the **bias-variance decomposition** of the risk function for linear least squares regression. The first term is the squared **bias** of the regression method, which can be interpreted as the expected difference between the hypothesis, $h(\boldsymbol{z})$, and the true label, γ, for the given test point, $\boldsymbol{z}$. The second term is the **variance** of the regression method, which can be interpreted as the expected deviation of the hypothesis, $h(\boldsymbol{z})$, from its average, $\mathbb{E}[h(\boldsymbol{z})]$. The third term is the **irreducible error**, which only depends on random noise. This error is called irreducible because it is not impacted by the hypothesis.

9.3 Polynomial Least Squares Regression

Polynomial least squares regression uses the polynomial regression function, squared error loss function, and mean loss cost function. We can obtain a polynomial least squares regression problem by modifying the linear least squares regression problem. One way to do so is to replace each data sample, $\boldsymbol{x_i}$, with the polynomial feature vector $\phi(\boldsymbol{x_i})$ before performing linear least squares regression. For example, if we have $d = 2$ two features and want to use a polynomial of degree $p = 2$, then our feature vector $\phi(\boldsymbol{x_i})$ would be

$$\phi(\boldsymbol{x_i}) = \begin{bmatrix} x_{i1}^2 & x_{i2}^2 & x_{i1}x_{i2} & x_{i1} & x_{i2} & 1 \end{bmatrix}^T$$

By using polynomial least squares regression, rather than linear least squares regression, we are able to obtain nonlinear decision boundaries. Note that for lower degrees of p, we are more likely to have high levels of bias and may underfit our data. Conversely, for higher degrees of p, we are more likely to have high levels of variance and may overfit our data. Generally, as the degree of our polynomial increases, the validation/test error will decrease at first, due to a decrease in bias. As we continue to increase the degree, the error will start to increase, due to an increase in variance. This is an example of bias-variance trade-off. We should tune the hyperparameter p to determine the value that gives us the lowest validation/test error. The impact of different degrees on the polynomial least squares regression solution is demonstrated in figure 9.1.

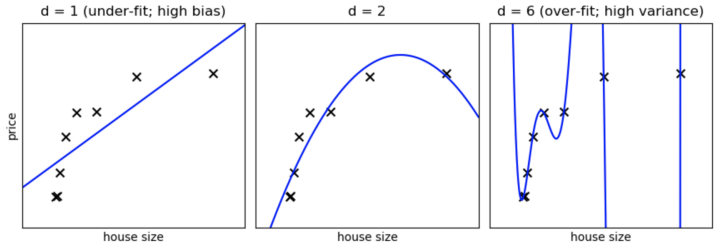

Figure 9.1: Suppose we want to use polynomial least squares regression to fit a model that relates price to house size. In this example, the degree one polynomial results in high bias and underfitting, while the degree six polynomial results in high variance and overfitting. The degree two polynomial gives us a significantly better model for the given data.

9.4 Weighted Least Squares Regression

Weighted least squares regression uses the linear regression function, squared error loss function, and weighted sum cost function. Combining all of these com-

ponents, weighted least squares regression is defined by the following problem:

$$\min_{\boldsymbol{w}\in\mathbb{R}^d,\alpha\in\mathbb{R}} \sum_{i=1}^{n} \omega_i\big(\boldsymbol{w}^T\boldsymbol{x_i} + \alpha - y_i\big)^2.$$

As with linear least squares regression, let $\tilde{\boldsymbol{x}}_{\boldsymbol{i}}$ be the sample point $\boldsymbol{x_i}$ augmented with one. Again, we will redefine $\boldsymbol{w} \in \mathbb{R}^{d+1}$ as the previous weight vector augmented with the bias term, α. Now we can express this problem as

$$\min_{\boldsymbol{w}\in\mathbb{R}^{d+1}} \sum_{i=1}^{n} \omega_i\big(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}} - y_i\big)^2.$$

Now let $\boldsymbol{X}$ be the $n \times (d+1)$ matrix whose ith row is the transpose of the ith augmented sample point, $\tilde{\boldsymbol{x}}_{\boldsymbol{i}} \in \mathbb{R}^{d+1}$. Let $\boldsymbol{y}$ be the n-dimensional vector whose ith element, y_i, is the label corresponding to sample point $\boldsymbol{x_i}$. We will also define the weight matrix $\boldsymbol{\Omega}$ as an $n \times n$ diagonal matrix whose ith diagonal entry is the weight ω_i. Now we can express this problem in as

$$\min_{\boldsymbol{w}\in\mathbb{R}^{d+1}} (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T\boldsymbol{\Omega}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}).$$

9.4.1 Optimal Solution

Because the objective is a convex quadratic function, we can find the optimal solution, $\hat{\boldsymbol{w}}$, by computing the gradient of the objective function with respect to $\boldsymbol{w}$ and setting it equal to zero. The gradient of the objective function is

$$\begin{aligned}
\nabla_w J(\boldsymbol{w}) &= \nabla_w\Big((\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T\boldsymbol{\Omega}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})\Big) \\
&= \nabla_w\big(\boldsymbol{w}^T\boldsymbol{X}^T\boldsymbol{\Omega}\boldsymbol{X}\boldsymbol{w} - 2\boldsymbol{y}^T\boldsymbol{\Omega}\boldsymbol{X}\boldsymbol{w} + \boldsymbol{y}^T\boldsymbol{\Omega}\boldsymbol{y}\big) \\
&= 2\boldsymbol{X}^T\boldsymbol{\Omega}\boldsymbol{X}\boldsymbol{w} - 2\boldsymbol{X}^T\boldsymbol{\Omega}\boldsymbol{y}.
\end{aligned}$$

Evaluating the gradient at $\boldsymbol{w} = \hat{\boldsymbol{w}}$ and setting it equal to zero, we have

$$2\boldsymbol{X}^T\boldsymbol{\Omega}\boldsymbol{X}\hat{\boldsymbol{w}} - 2\boldsymbol{X}^T\boldsymbol{\Omega}\boldsymbol{y} = 0$$

$$\boldsymbol{X}^T\boldsymbol{\Omega}\boldsymbol{X}\hat{\boldsymbol{w}} = \boldsymbol{X}^T\boldsymbol{\Omega}\boldsymbol{y}$$

If $\boldsymbol{X}^T\boldsymbol{\Omega}\boldsymbol{X}$ is invertible, then the unique solution is given by

$$\hat{\boldsymbol{w}} = (\boldsymbol{X}^T\boldsymbol{\Omega}\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{\Omega}\boldsymbol{y}.$$

If this matrix is not invertible, the minimum norm solution is

$$\hat{\boldsymbol{w}} = (\boldsymbol{X}^T\boldsymbol{\Omega}\boldsymbol{X})^{\dagger}\boldsymbol{X}^T\boldsymbol{\Omega}\boldsymbol{y}.$$

9.4.2 Advantages & Disadvantages

Weighted least squares regression is useful because it allows us to give greater weight to data we believe is more reliable and lower weight to data that may be noisy or prone to errors when estimating the numerical labels of unseen data.

9.5 Logistic Regression

Logistic regression uses the logistic regression function, logistic loss function, and mean loss cost function. Combining all of these components, logistic regression is defined by the following optimization problem:

$$\min_{\boldsymbol{w}\in\mathbb{R}^d,\alpha\in\mathbb{R}} \frac{1}{n}\sum_{i=1}^{n}\Big(-y_i\ln\big(s(\boldsymbol{w}^T\boldsymbol{x_i}+\alpha)\big)-(1-y_i)\ln\big(1-s(\boldsymbol{w}^T\boldsymbol{x_i}+\alpha)\big)\Big).$$

We can remove the constant factor and equivalently express this problem as

$$\min_{\boldsymbol{w}\in\mathbb{R}^d,\alpha\in\mathbb{R}} -\sum_{i=1}^{n}\Big(y_i\ln\big(s(\boldsymbol{w}^T\boldsymbol{x_i}+\alpha)\big)+(1-y_i)\ln\big(1-s(\boldsymbol{w}^T\boldsymbol{x_i}+\alpha)\big)\Big).$$

As with linear least squares regression, let $\tilde{\boldsymbol{x}}_{\boldsymbol{i}}$ be the sample point $\boldsymbol{x_i}$ augmented with one, and let $\boldsymbol{X}$ be the $n\times(d+1)$ matrix whose ith row is the transpose of the ith augmented sample point, $\tilde{\boldsymbol{x}}_{\boldsymbol{i}}\in\mathbb{R}^{d+1}$. Again, define $\boldsymbol{y}$ as the n-dimensional vector whose ith element, y_i, is the label corresponding to sample point $\boldsymbol{x_i}$. Finally, we will redefine $\boldsymbol{w}\in\mathbb{R}^{d+1}$ as the previous weight vector augmented with the bias term, α. Now we can express this optimization problem as

$$\min_{\boldsymbol{w}\in\mathbb{R}^{d+1}} -\sum_{i=1}^{n}\Big(y_i\ln\big(s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\big)+(1-y_i)\ln\big(1-s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\big)\Big).$$

9.5.1 Optimal Solution

The objective for the logistic regression problem is convex, but this problem does not have a closed-form solution. Therefore, we cannot find the optimal solution, $\hat{\boldsymbol{w}}$, using the optimality condition. Instead, we can find the solution using batch gradient descent, stochastic gradient descent, or Newton's method.

Batch Gradient Descent

I will first discuss how to find the solution to the logistic regression optimization problem using batch gradient descent. Recall that in batch gradient descent, we find the optimal solution by applying the iterative update rule

$$\boldsymbol{w_{k+1}} = \boldsymbol{w_k} - \eta\nabla_w J(\boldsymbol{w})\big|_{\boldsymbol{w}=\boldsymbol{w_k}},$$

where η is a positive step size that determines the rate of convergence. To find the update rule for batch gradient descent, we first need to compute the gradient

of the objective function with respect to $\boldsymbol{w}$. Using the chain rule, we get

$$\begin{aligned}\nabla_w J(\boldsymbol{w}) &= -\sum_{i=1}^{n} \nabla_w \Big(y_i \ln\big(s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\big) + (1-y_i)\ln\big(1-s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\big)\Big) \\ &= -\sum_{i=1}^{n} \Bigg(y_i \frac{1}{s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})} \frac{d}{d\gamma}\Big[s(\gamma)\Big]_{\gamma=\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}}} \tilde{\boldsymbol{x}}_{\boldsymbol{i}} \\ &\qquad + (1-y_i)\frac{-1}{1-s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})} \frac{d}{d\gamma}\Big[s(\gamma)\Big]_{\gamma=\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}}} \tilde{\boldsymbol{x}}_{\boldsymbol{i}} \Bigg).\end{aligned}$$

Notice that the derivative of the logistic function can be expressed as

$$\frac{d}{d\gamma}s(\gamma) = \frac{d}{d\gamma}\left(\frac{1}{1+e^{-\gamma}}\right) = \frac{e^{-\gamma}}{\big(1+e^{-\gamma}\big)^2} = s(\gamma)\big(1-s(\gamma)\big).$$

This allows us to express the gradient of the logistic regression objective as

$$\begin{aligned}\nabla_w J(\boldsymbol{w}) &= -\sum_{i=1}^{n} \Bigg(y_i \frac{1}{s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})} s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\big(1-s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\big)\tilde{\boldsymbol{x}}_{\boldsymbol{i}} \\ &\qquad + (1-y_i)\frac{-1}{1-s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})} s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\big(1-s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\big)\tilde{\boldsymbol{x}}_{\boldsymbol{i}} \Bigg) \\ &= -\sum_{i=1}^{n} \Big(y_i\big(1-s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\big)\tilde{\boldsymbol{x}}_{\boldsymbol{i}} - (1-y_i)s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\tilde{\boldsymbol{x}}_{\boldsymbol{i}} \Big) \\ &= -\sum_{i=1}^{n} \big(y_i\tilde{\boldsymbol{x}}_{\boldsymbol{i}} - s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\tilde{\boldsymbol{x}}_{\boldsymbol{i}} \big) = \sum_{i=1}^{n} \big(s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}}) - y_i \big)\tilde{\boldsymbol{x}}_{\boldsymbol{i}}.\end{aligned}$$

Now we can express the batch gradient descent update rule as

$$\begin{aligned}\boldsymbol{w_{k+1}} &= \boldsymbol{w_k} - \eta \sum_{i=1}^{n} \big(s(\boldsymbol{w_k}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}}) - y_i \big)\tilde{\boldsymbol{x}}_{\boldsymbol{i}} \\ &= \boldsymbol{w_k} + \eta \sum_{i=1}^{n} \big(y_i - s(\boldsymbol{w_k}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}}) \big)\tilde{\boldsymbol{x}}_{\boldsymbol{i}}.\end{aligned}$$

Note that if we define $\boldsymbol{s}(\boldsymbol{w})$ as the n-dimensional vector whose ith element is $s_i(\boldsymbol{w}) := s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})$, we can equivalently express the gradient of the objective as

$$\nabla_w J(\boldsymbol{w}) = \boldsymbol{X}^T\big(\boldsymbol{s}(\boldsymbol{w}) - \boldsymbol{y}\big).$$

This then allows us to express the batch gradient descent update rule as

$$\begin{aligned}\boldsymbol{w_{k+1}} &= \boldsymbol{w_k} - \eta\boldsymbol{X}^T\big(\boldsymbol{s}(\boldsymbol{w_k}) - \boldsymbol{y}\big) \\ &= \boldsymbol{w_k} + \eta\boldsymbol{X}^T\big(\boldsymbol{y} - \boldsymbol{s}(\boldsymbol{w_k})\big).\end{aligned}$$

Stochastic Gradient Descent

Notice that for batch gradient descent, we updated the weight vector using all of the data samples at each iteration. Sometimes it is better to update the weight vector using just one sample at each iteration. This method of gradient descent is called stochastic gradient descent. Rather than updating $\boldsymbol{w_k}$ based on the entire gradient at each iteration, we could simply use the portion of the gradient contributed to by the ith data sample. We can express the stochastic gradient descent update law for logistic regression as

$$\begin{aligned}\boldsymbol{w_{k+1}} &= \boldsymbol{w_k} - \eta\big(s(\boldsymbol{w_k}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}}) - y_i\big)\tilde{\boldsymbol{x}}_{\boldsymbol{i}} \\ &= \boldsymbol{w_k} + \eta\big(y_i - s(\boldsymbol{w_k}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\big)\tilde{\boldsymbol{x}}_{\boldsymbol{i}}.\end{aligned}$$

Newton's Method

We can also find the optimal solution, $\hat{\boldsymbol{w}}$, using Newton's method. In general, the update law for Newton's method is given by

$$\boldsymbol{w_{k+1}} = \boldsymbol{w_k} - \eta\Big(\nabla_w^2 J(\boldsymbol{w})|_{\boldsymbol{w}=\boldsymbol{w_k}}\Big)^{-1}\nabla_w J(\boldsymbol{w})|_{\boldsymbol{w}=\boldsymbol{w_k}}.$$

To determine the update rule for Newton's method, we need to compute both the gradient and Hessian of $J(\boldsymbol{w})$. We previously showed that the gradient is

$$\nabla_w J(\boldsymbol{w}) = \sum_{i=1}^{n}\big(s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}}) - y_i\big)\tilde{\boldsymbol{x}}_{\boldsymbol{i}}.$$

Using the chain rule, the Hessian of the objective is given by

$$\begin{aligned}\nabla_w^2 J(\boldsymbol{w}) &= \nabla_w \sum_{i=1}^{n}\big(s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}}) - y_i\big)\tilde{\boldsymbol{x}}_{\boldsymbol{i}} \\ &= \sum_{i=1}^{n}\nabla_w\Big(s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\tilde{\boldsymbol{x}}_{\boldsymbol{i}}\Big) \\ &= \sum_{i=1}^{n} s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\big(1 - s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\big)\tilde{\boldsymbol{x}}_{\boldsymbol{i}}\tilde{\boldsymbol{x}}_{\boldsymbol{i}}^T.\end{aligned}$$

Now we can express the Newton's method update rule as

$$\begin{aligned}\boldsymbol{w_{k+1}} &= \boldsymbol{w_k} - \eta\left(\sum_{i=1}^{n} s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\big(1 - s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\big)\tilde{\boldsymbol{x}}_{\boldsymbol{i}}\tilde{\boldsymbol{x}}_{\boldsymbol{i}}^T\right)^{-1}\left(\sum_{i=1}^{n}\big(s(\boldsymbol{w_k}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}}) - y_i\big)\tilde{\boldsymbol{x}}_{\boldsymbol{i}}\right) \\ &= \boldsymbol{w_k} + \eta\left(\sum_{i=1}^{n} s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\big(1 - s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\big)\tilde{\boldsymbol{x}}_{\boldsymbol{i}}\tilde{\boldsymbol{x}}_{\boldsymbol{i}}^T\right)^{-1}\left(\sum_{i=1}^{n}\big(y_i - s(\boldsymbol{w_k}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\big)\tilde{\boldsymbol{x}}_{\boldsymbol{i}}\right).\end{aligned}$$

Note that if $\boldsymbol{S}(\boldsymbol{w})$ is the $n \times n$ diagonal matrix whose ith diagonal element is $S_{ii}(\boldsymbol{w}) := s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})$, we can equivalently express the Hessian of the objective as

$$\nabla_w^2 J(\boldsymbol{w}) = \boldsymbol{X}^T\boldsymbol{S}(\boldsymbol{w})\big(\boldsymbol{I_n} - \boldsymbol{S}(\boldsymbol{w})\big)\boldsymbol{X}.$$

This then allows us to express the Newton's method update rule as

$$\begin{aligned}\boldsymbol{w_{k+1}} &= \boldsymbol{w_k} - \eta\Big(\boldsymbol{X}^T\boldsymbol{S}(\boldsymbol{w_k})\big(\boldsymbol{I_n} - \boldsymbol{S}(\boldsymbol{w_k})\big)\boldsymbol{X}\Big)^{-1}\boldsymbol{X}^T\big(\boldsymbol{s}(\boldsymbol{w_k}) - \boldsymbol{y}\big) \\ &= \boldsymbol{w_k} + \eta\Big(\boldsymbol{X}^T\boldsymbol{S}(\boldsymbol{w_k})\big(\boldsymbol{I_n} - \boldsymbol{S}(\boldsymbol{w_k})\big)\boldsymbol{X}\Big)^{-1}\boldsymbol{X}^T\big(\boldsymbol{y} - \boldsymbol{s}(\boldsymbol{w_k})\big).\end{aligned}$$

9.5.2 Advantages & Disadvantages

Logistic regression can be used to fit probability models because it produces predictions in the range $(0, 1)$ and is often used for binary classification with labels 0 and 1. Logistic regression is an example of a discriminative model, which directly models the posterior probability of each class as a logistic function. Recall that LDA (section 8.3) is an example of a generative model, which indirectly models the posterior probability of each class as a logistic function. Logistic regression and LDA are similar methods of finding linear decision boundaries, but there are advantages and disadvantages of both classification methods:

1. For well-separated data, LDA is a stable classification method, while logistic regression is unstable. This means that changing one sample point will not greatly change the decision boundary in LDA, but it may greatly change the decision boundary in logistic regression.

2. LDA handles multi-class problems easily, while logistic regression needs to be modified to handle problems with more than two classes.

3. LDA is slightly more accurate when class distributions are nearly normal, but logistic regression is more robust on some non-Gaussian distributions, such as distributions with large skew.

4. Logistic regression places more of an emphasis on the decision boundary, so it always separates linearly separable points.

Chapter 10

Regularization

10.1 Overview of Regularization

We can use regularization to improve the performance of a regression model. For any regression problem, there are two types of regularization: l_2 and l_1. Regularization can be used with any regression problem defined by any combination of regression, loss, and cost function, but we will focus on its application to linear least squares regression. When we use l_2 regularization with the linear least squares regression problem, we refer to the new problem as **ridge regression** or **Tikohonov regularization**. When we use l_1 regularization with the linear least squares regression problem, we refer to the new problem as **LASSO**. We will go into detail about ridge regression and LASSO in the following sections. Again, we will assume that we are working with n samples points $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ with corresponding labels $y_1, \ldots, y_n$, where $\boldsymbol{x_i} \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$.

10.2 Ridge Regression (Tikohonov Regularization)

Ridge regression, which is also referred to as **Tikohonov regularization**, adds an l_2 regularization term to the ordinary linear least squares regression problem, resulting in the following optimization problem:

$$\min_{\boldsymbol{w} \in \mathbb{R}^d, \alpha \in \mathbb{R}} \frac{1}{n} \sum_{i=1}^{n} \left(\boldsymbol{w}^T \boldsymbol{x_i} + \alpha - y_i\right)^2 + \lambda ||\boldsymbol{w}||_2^2.$$

In the optimization problem above, λ is a regularization term. For larger values of λ, we more strongly encourage weights to be closer to zero. We can remove the constant factor and equivalently express this problem as

$$\min_{\boldsymbol{w} \in \mathbb{R}^d, \alpha \in \mathbb{R}} \sum_{i=1}^{n} \left(\boldsymbol{w}^T \boldsymbol{x_i} + \alpha - y_i\right)^2 + \lambda ||\boldsymbol{w}||_2^2.$$

Let $\tilde{\boldsymbol{x}}_{\boldsymbol{i}}$ be the sample point $\boldsymbol{x}_{\boldsymbol{i}}$ augmented with one, and redefine $\boldsymbol{w} \in \mathbb{R}^{d+1}$ as the previous weight vector augmented with the bias term, α. We will now use $\boldsymbol{w'} \in \mathbb{R}^d$ to denote the original weight vector without the bias term. With these definitions, we can now express the ridge regression problem as

$$\min_{\boldsymbol{w} \in \mathbb{R}^{d+1}} \sum_{i=1}^{n} \left(\boldsymbol{w}^T \tilde{\boldsymbol{x}}_{\boldsymbol{i}} - y_i\right)^2 + \lambda ||\boldsymbol{w'}||_2^2.$$

Now let $\boldsymbol{X}$ be the $n \times (d+1)$ matrix whose ith row is the transpose of the ith augmented sample point, $\tilde{\boldsymbol{x}}_{\boldsymbol{i}} \in \mathbb{R}^{d+1}$, and let $\boldsymbol{y}$ be the n-dimensional vector whose ith element, y_i, is the label corresponding to sample point $\boldsymbol{x}_{\boldsymbol{i}}$. Now we can express the ridge regression problem as

$$\min_{\boldsymbol{w} \in \mathbb{R}^{d+1}} ||\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}||_2^2 + \lambda ||\boldsymbol{w'}||_2^2.$$

10.2.1 Optimal Solution

Because the objective is a convex quadratic function, we can find the optimal solution, $\hat{\boldsymbol{w}}$, by computing the gradient of the objective function with respect to $\boldsymbol{w}$ and setting it equal to zero. The gradient of the objective function is

$$\begin{aligned}
\nabla_w J(\boldsymbol{w}) &= \nabla_w \left(||\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}||_2^2 + \lambda ||\boldsymbol{w'}||_2^2 \right) \\
&= \nabla_w \left(\boldsymbol{w}^T \boldsymbol{X}^T \boldsymbol{X} \boldsymbol{w} - 2\boldsymbol{y}^T \boldsymbol{X} \boldsymbol{w} + \boldsymbol{y}^T \boldsymbol{y} + \lambda \boldsymbol{w'}^T \boldsymbol{w'} \right) \\
&= 2\boldsymbol{X}^T \boldsymbol{X} \boldsymbol{w} - 2\boldsymbol{X}^T \boldsymbol{y} + 2\lambda \boldsymbol{I'_{d+1}} \boldsymbol{w}.
\end{aligned}$$

Note that I am defining $\boldsymbol{I'_{d+1}}$ as the $(d+1) \times (d+1)$ identity matrix whose bottom right element has been set to zero. Evaluating the gradient at $\boldsymbol{w} = \hat{\boldsymbol{w}}$ and setting the expression equal to zero, we get

$$2\boldsymbol{X}^T \boldsymbol{X} \hat{\boldsymbol{w}} - 2\boldsymbol{X}^T \boldsymbol{y} + 2\lambda \boldsymbol{I'_{d+1}} \hat{\boldsymbol{w}} = 0$$

$$\left(\boldsymbol{X}^T \boldsymbol{X} + \lambda \boldsymbol{I'_{d+1}} \right) \hat{\boldsymbol{w}} = \boldsymbol{X}^T \boldsymbol{y}$$

Recall that the solution for linear least square regression was the solution to the linear equation $\boldsymbol{X}^T \boldsymbol{X} \hat{\boldsymbol{w}} = \boldsymbol{y}^T \boldsymbol{X}$, which only has a unique solution when $\boldsymbol{X}^T \boldsymbol{X}$ is nonsingular. The matrix $\boldsymbol{X}^T \boldsymbol{X} + \lambda \boldsymbol{I'_{d+1}}$ is always nonsingular, so ridge regression always admits the unique solution

$$\hat{\boldsymbol{w}} = \left(\boldsymbol{X}^T \boldsymbol{X} + \lambda \boldsymbol{I'_{d+1}} \right)^{-1} \boldsymbol{X}^T \boldsymbol{y}.$$

10.2.2 Variation of Ridge Regression

One variation of ridge regression is to instead minimize the cost function

$$J(\boldsymbol{w}) = ||\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}||_2^2 + \lambda \boldsymbol{w}^T \boldsymbol{D} \boldsymbol{w},$$

where $\boldsymbol{D}$ is a diagonal matrix whose last diagonal element is zero. We can use the matrix $\boldsymbol{D}$ to more heavily penalize weights associated with features that we believe are less accurate and more prone to errors. Rather than penalizing all of the weights with the value λ, we can penalize each component of the weight, w_i, with the value λD_{ii}. For higher values of D_{ii}, we are forcing the size of the ith component of $\boldsymbol{w}$ closer to zero. Note that we enforce that the last diagonal element of $\boldsymbol{D}$ is zero because we do not want to penalize the bias term α.

10.3 LASSO

LASSO, which stands for "Least Absolute Shrinkage and Selection Operator", adds an l_1 regularization term to the ordinary linear least squares regression problem, resulting in the following optimization problem:

$$\min_{\boldsymbol{w}\in\mathbb{R}^d,\alpha\in\mathbb{R}} \frac{1}{n}\sum_{i=1}^{n}\left(\boldsymbol{w}^T\boldsymbol{x_i}+\alpha-y_i\right)^2+\lambda||\boldsymbol{w}||_1.$$

We can remove the constant factor and equivalently express this problem as

$$\min_{\boldsymbol{w}\in\mathbb{R}^d,\alpha\in\mathbb{R}} \sum_{i=1}^{n}\left(\boldsymbol{w}^T\boldsymbol{x_i}+\alpha-y_i\right)^2+\lambda||\boldsymbol{w}||_1.$$

Let $\tilde{\boldsymbol{x}}_{\boldsymbol{i}}$ be the sample point $\boldsymbol{x_i}$ augmented with one, and redefine $\boldsymbol{w}\in\mathbb{R}^{d+1}$ as the previous weight vector augmented with the bias term, α. We will now use $\boldsymbol{w'}\in\mathbb{R}^d$ to denote the original weight vector without the bias term. With these definitions, we can now express the LASSO problem as

$$\min_{\boldsymbol{w}\in\mathbb{R}^{d+1}} \sum_{i=1}^{n}\left(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}}-y_i\right)^2+\lambda||\boldsymbol{w'}||_1.$$

Now let $\boldsymbol{X}$ be the $n\times(d+1)$ matrix whose ith row is the transpose of the ith augmented sample point, $\tilde{\boldsymbol{x}}_{\boldsymbol{i}}\in\mathbb{R}^{d+1}$, and let $\boldsymbol{y}$ be the n-dimensional vector whose ith element, y_i, is the label corresponding to sample point $\boldsymbol{x_i}$. Now we can express the LASSO problem as

$$\min_{\boldsymbol{w}\in\mathbb{R}^{d+1}} ||\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y}||_2^2+\lambda||\boldsymbol{w'}||_1.$$

Unlike ridge regression, LASSO does not admit a closed-form solution.

10.4 Bias-Variance Trade-Off

In practice, regularization often improves the performance of regression models. This improvement can partially be explained by the bias-variance trade-off. By adding a regularization/penalty term to the ordinary linear least squares problem, we encourage weights to be closer to zero. This changes both the bias

and the variance associated with the original regression problem. As we increase the size of the regularization parameter, λ, we penalize large weights more, which gives us smaller weights. For large weights, a small change in a sample point, $\boldsymbol{x_i}$, results in a large change in the label, y_i, which creates a lot of variance. Therefore, by increasing the size of the regularization parameter, λ, we reduce the variance. In fact, as λ approaches infinity, the variance in ridge regression and LASSO approaches zero. However, as we increase λ and encourage smaller weights, we may lose information in our model, which increases the bias. The effect of the regularization parameter is depicted in figure 10.1.

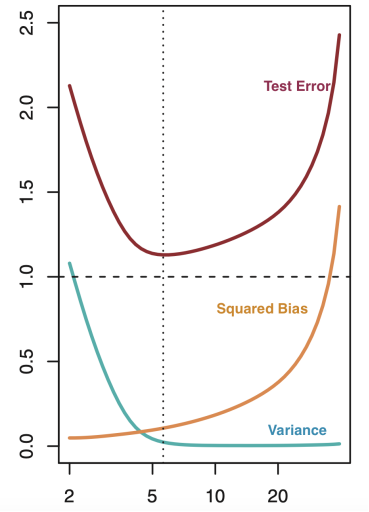

Figure 10.1: This graph shows the value of the variance (teal), squared bias (orange), and test error (red) as we increase the value of the regularization parameter, λ. Through validation, we can find the value of λ that minimizes the sum of the squared bias and variance, resulting in the best test performance.

10.5 Comparison of Regularization Methods

While both LASSO and ridge regression are able to provide improvements to the standard linear least squares regression problem, there are important differences between these two methods. We will consider the statistical implications of these methods, as well as the use of these methods for feature selection.

10.5.1 Statistical Justification

We can think of both l_2 and l_1 regularized regression problems as **maximum a posteriori (MAP)** estimation. Treat the weight vector, $\boldsymbol{W}$, as a random

vector and let $\boldsymbol{w}$ be a realization. Similarly, treat the vector of labels, $\boldsymbol{Y}$, for a given dataset as a random vector, and let $\boldsymbol{y}$ be a realization.
To find the optimal weight vector, $\hat{\boldsymbol{w}}$, we solve the following problem:

$$\hat{\boldsymbol{w}} = \arg\max_{\boldsymbol{w}} \; f_{\boldsymbol{W}|\boldsymbol{Y}}(\boldsymbol{w}|\boldsymbol{y}).$$

Because the natural log is monotonically increasing, we can equivalently solve

$$\hat{\boldsymbol{w}} = \arg\max_{\boldsymbol{w}} \; \ln f_{\boldsymbol{W}|\boldsymbol{Y}}(\boldsymbol{w}|\boldsymbol{y}).$$

Using Bayes theorem, we can express the conditional PDF in the objective as

$$f_{\boldsymbol{W}|\boldsymbol{Y}}(\boldsymbol{w}|\boldsymbol{y}) = \frac{f_{\boldsymbol{Y}|\boldsymbol{W}}(\boldsymbol{y}|\boldsymbol{w}) f_{\boldsymbol{W}}(\boldsymbol{w})}{f_Y(\boldsymbol{y})} \propto f_{\boldsymbol{Y}|\boldsymbol{W}}(\boldsymbol{y}|\boldsymbol{w}) f_{\boldsymbol{W}}(\boldsymbol{w}).$$

Therefore, the optimal weight vector can also be expressed as the solution to

$$\hat{\boldsymbol{w}} = \arg\max_{\boldsymbol{w}} \; \ln\Big(f_{\boldsymbol{Y}|\boldsymbol{W}}(\boldsymbol{y}|\boldsymbol{w}) f_{\boldsymbol{W}}(\boldsymbol{w}) \Big).$$

From our previous assumptions, the ith component of the weight vector, $\boldsymbol{w}$, is a realization of the random variable W_i, which is the ith component of $\boldsymbol{W}$. If we assume that the random variables $W_1, \ldots, W_d$ are independent and can be modeled by the PDFs $f_{W_1}(w_1), \ldots, f_{W_d}(w_d)$, then we can express $f_{\boldsymbol{W}}(\boldsymbol{w})$ as

$$f_{\boldsymbol{W}}(\boldsymbol{w}) = \prod_{i=1}^{d} f_{W_i}(w_i).$$

Similarly, the ith component of the label, $\boldsymbol{y}$, is a realization of the random variable Y_i, which is the ith component of $\boldsymbol{Y}$. If we assume that the random variables $Y_1, \ldots, Y_n$ are independent and can be modeled by the conditional PDFs $f_{Y_1|\boldsymbol{W}}(y_1, \boldsymbol{w}), \ldots, f_{Y_n|\boldsymbol{W}}(y_n|\boldsymbol{w})$, then we can express $f_{\boldsymbol{Y}|\boldsymbol{W}}(\boldsymbol{y}|\boldsymbol{w})$ as

$$f_{\boldsymbol{Y}|\boldsymbol{W}}(\boldsymbol{y}|\boldsymbol{w}) = \prod_{i=1}^{n} f_{Y_i|\boldsymbol{W}}(y_i|\boldsymbol{w}).$$

Under these assumptions, our objective can be expressed as

$$\hat{\boldsymbol{w}} = \arg\max_{\boldsymbol{w}} \; \ln\left(\prod_{i=1}^{n} f_{Y_i|\boldsymbol{W}}(y_i|\boldsymbol{w}) \prod_{i=1}^{d} f_{W_i}(w_i) \right).$$

Using properties of the natual log, we can equivalently express this problem as

$$\hat{\boldsymbol{w}} = \arg\max_{\boldsymbol{w}} \; \left(\sum_{i=1}^{n} \ln f_{Y_i|\boldsymbol{W}}(y_i|\boldsymbol{w}) + \sum_{i=1}^{d} \ln f_{W_i}(w_i) \right).$$

We can think of ridge regression as MAP estimation where we impose a Gaussian prior with zero mean on the weights. More specifically, we assume that weight w_i is sampled from the distribution $\mathcal{N}(0, \sigma)$, which has the PDF

$$f_{W_i}(w_i) = \frac{1}{\sqrt{2\pi}\sigma} e^{-w_i^2/2\sigma^2}.$$

We can think of LASSO as MAP estimation where we impose a Laplace prior with zero mean on the weights. More specifically, we assume that weight w_i is sampled from the distribution Laplace$(0, b)$, which has the PDF

$$f_{W_i}(w_i) = \frac{1}{2b} e^{-|w_i|/b}.$$

Because both of these PDFs have mean zero, they assume that weights are close to zero. Figure 10.2 depicts these two PDFs for the one-dimensional case.

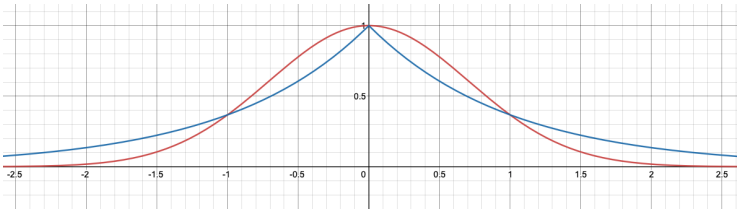

Figure 10.2: The red curve is a zero-mean Gaussian PDF and the blue curve is a zero-mean Laplace PDF for the weight $w \in \mathbb{R}$.

10.5.2 Feature Selection

It is interesting to compare the optimal solutions of the ridge regression and LASSO problems. I will denote $\hat{\boldsymbol{w}}_{\boldsymbol{1}}$ as the optimal solution to the LASSO problem and $\hat{\boldsymbol{w}}_{\boldsymbol{2}}$ as the optimal solution to the ridge regression problem.

The ith component of the solution for LASSO, $\hat{\boldsymbol{w}}_{\boldsymbol{1}}$, is equal to zero if and only if $|\boldsymbol{y}^T \boldsymbol{x}_{\boldsymbol{j}}| \leq \frac{\lambda}{2}$, where $\boldsymbol{y}$ is the vector of labels and $\boldsymbol{x}_{\boldsymbol{j}}$ is the jth column of the design matrix $\boldsymbol{X}$. The ith component of the solution for ridge regression, $\hat{\boldsymbol{w}}_{\boldsymbol{2}}$, is equal to zero if and only if $\boldsymbol{y}^T \boldsymbol{x}_{\boldsymbol{j}} = 0$. It is relatively straightforward to prove these facts, but I will not include proofs in these notes.

Assuming the above conditions are true, there is a range of values of $\boldsymbol{y}^T \boldsymbol{x}_{\boldsymbol{j}}$ for which the ith component of $\hat{\boldsymbol{w}}_{\boldsymbol{1}}$ equals zero and only a single value for which the ith component of $\hat{\boldsymbol{w}}_{\boldsymbol{2}}$ equals zero. This implies that $\hat{\boldsymbol{w}}_{\boldsymbol{1}}$ is more likely to have zero components, so we say that $\hat{\boldsymbol{w}}_{\boldsymbol{1}}$ is more likely to be sparse. This concept is demonstrated in figure 10.3. If certain features do not have strong predictive power, we want their weights to be set to zero, so this is an advantage of LASSO over ridge regression. LASSO can be used for feature selection.

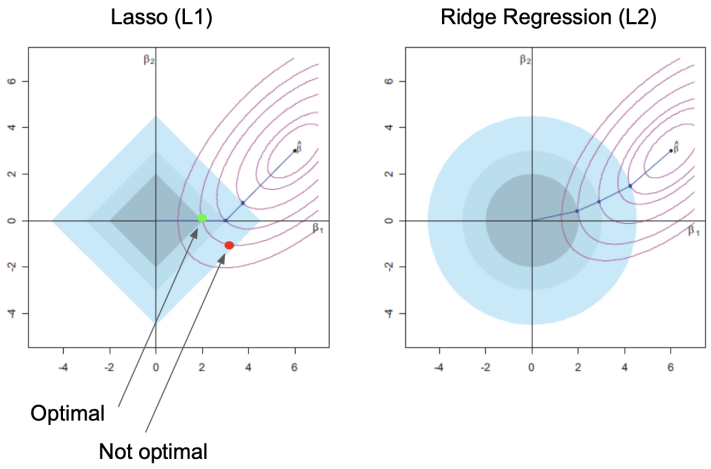

Figure 10.3: In both images, the red ellipses are the isocontours of the linear least squares objective. The image on the left also depicts the isocontours of the LASSO regularization term, $\lambda||\boldsymbol{w}||_1$, and the image on the right also depicts the isocontours of the ridge regression regularization term, $\lambda||\boldsymbol{w}||_2^2$. Feasible values of $\boldsymbol{w}$ occur where the isocontours of the least squares objective and the regularization term intersect. The value of the objective function is the sum of the isocontour values at these intersections. For LASSO, an optimal value of $\boldsymbol{w}$ often occurs when one of its components is equal to zero, so LASSO encourages sparse solutions. This is not true for ridge regression.

Chapter 11

Decision Trees

11.1 Overview of Decision Trees

Decision trees are a supervised learning method used for both classification and regression. They use the tree data structure with two types of nodes:

1. **Leaf nodes** – A leaf node is the last node in a branch of the tree, and it determines the label of the data contained at that node.

2. **Internal nodes** – An internal node is any node in the tree that is not a leaf node, and it splits on a feature value for some threshold value.

As an example, suppose we are trying to decide whether to go out for a picnic. Our set of features include the outlook, percent humidity, and wind speed. The outlook is a categorical variable, which takes on the values: sunny, rainy, and overcast. The percent humidity is a quantitative variable, which ranges from 0 to 100. The wind speed is another quantitative variable, which ranges from 0 to 50. Figure 11.1 shows an example decision tree for this problem. Note that decision trees can effectively handle both categorical and quantitative features.

11.1.1 Advantages & Disadvantages

Decision trees have a number of advantages, as well as disadvantages compared to other supervised learning techniques. A few are listed below:

Advantages	Disadvantages
1. Fast and simple	1. Tend to have high variance
2. Interpretable and easy to explain	2. Often have poor performance compared to other methods
3. Invariant under scaling/translation	
4. Robust to irrelevant features	

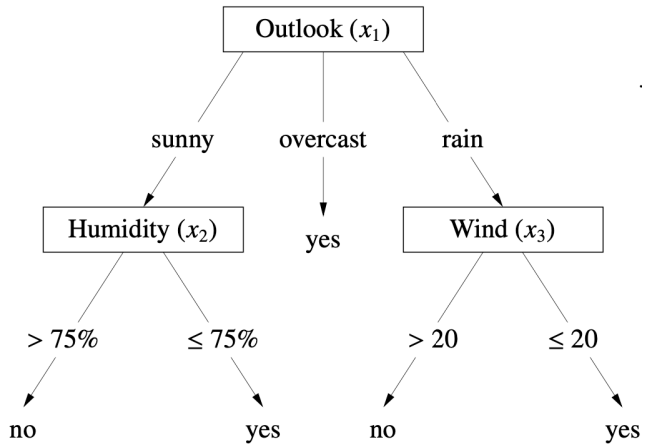

Figure 11.1: This is an example decision tree used to determine whether to go out for a picnic. There are three internal nodes: outlook (x_1), humidity (x_2), and wind (x_3). There are five leaf nodes, which specify whether to go for a picnic (yes or no). The outlook node splits on its three values: sunny, rainy, and overcast. The humidity node splits at the threshold value 75, and the wind node splits at the threshold value 20.

11.2 Binary Decision Trees for Classification

11.2.1 Decision Tree Nodes

As mentioned in the previous section, decision trees are composed of internal nodes, which split on feature values for some thresholds, and leaf nodes, which determine the label of a data point. We will focus on binary decision trees, for which each internal node splits on only one feature based on a single threshold value. Each **internal node** contains the following information:

1. the set of data samples contained at that node,
2. the corresponding label of each sample point,
3. pointers to left and right children of that node, and
4. the split rule composed of the feature and threshold value.

Each **leaf node** contains the following information:

1. the set of data samples contained at that node,
2. the true corresponding label of each sample point, and
3. the predicted label of each sample point at that node.

For internal nodes, the left and right child nodes are defined by the split rule. Limiting ourselves to two-way splits on single features, suppose that the split rule is composed of the feature index j and threshold β. Let $\boldsymbol{X}$ be an $n \times d$ design matrix and define $S \subseteq \{1, 2, \ldots, n\}$ as a set of sample point indices corresponding to the subset of data contained at the current node. The sample points contained at the left and right child nodes are given by the following sets:

1. **Left child** – $S_l = \{i \in S : X_{ij} < \beta\}$ and $y_l = \{y_i : i \in S_l\}$
2. **Right child** – $S_r = \{i \in S : X_{ij} \geq \beta\}$ and $y_r = \{y_i : i \in S_r\}$

11.2.2 Decision Tree Training

The greedy top-down learning heuristic to train a binary decision tree for classification uses recursion with a function similar to the one in algorithm 3. The top-level call for this function is GrowTree(S, $\boldsymbol{y}$, *root*), where $S = \{1, 2, \ldots, n\}$ is the full set of indices and $\boldsymbol{y} \in \mathbb{R}^n$ is the full label vector contained at the root.

Algorithm 3: Algorithm for Decision Tree Training

```
1  Function GrowTree(S, y, node):
2      Set data to S and labels to y
3      if stopping condition met then
4          Set predicted labels of node
5      else
6          Choose best split
7          Split data and labels based on split rule
8          Initialize left and right child nodes
9          GrowTree(S_l, y_l, left child node)
10         GrowTree(S_r, y_r, right child node)
11     end
12 End Function
```

11.2.3 Choosing the Best Split

Line 6 of algorithm 3 says to choose the best split for an internal node. This entails choosing both a feature and threshold to split on. To choose the best split, we must consider different types of features and the possible thresholds for each type. We also need to define what is meant by the "best split," which depends on our choice of cost function. Often, we use the entropy to define the cost of a given split and choose the split that offers the greatest information gain. We will discuss possible thresholds, a definition of best split, entropy, and information gain in greater detail in the following sections.

Possible Thresholds

We will assume that at each internal node, we split on a single feature, giving us axis-aligned splits. To choose the best split for an internal node, we try to split each feature at each possible threshold. The possible splits for a given feature depends on the category that it falls into. There are three types of features:

1. **Binary** – A binary feature can take on one of two discrete values. Generally, we map one of the discrete values to 1 and one to 0, so the value of a binary feature is either 1 or 0. For example, one of our features may be student status, which can either be grad or undergrad. We will map grad to the binary value 1 and undergrad to the binary value 0. For a binary feature, there is only one possible split, so we choose the threshold $\beta = \frac{1}{2}$.

2. **Categorical** – A categorical features takes on one of k discrete values. For a categorical feature with more than two discrete values, we could use binary splits or multi-way splits. We will are focusing on binary decision trees, which only use binary splits. Typically for categorical features, we apply a one-hot encoding to turn one categorical feature with k possible values into k binary features. For example, one of our features may be color, which can be red, blue, or green. We could replace this feature with three new features: one that is 1 if the color is red and 0 otherwise, one that is 1 if the color is blue and 0 otherwise, and one that is 1 if the color is green and 0 otherwise. Now for each of these binary features, there is only one possible split, so we choose the threshold $\beta = \frac{1}{2}$.

3. **Quantitative** – A quantitative feature can take on a range of values. For a quantitative feature, we sort the set, S, based on the numerical value of that feature, then we try splitting halfway between each pair of unequal consecutive values to determine the best threshold value to split on.

Definition of Best Split

Now we have covered the possible threshold values for each type of feature, but we do not yet have a way to determine the best feature to split on and the best threshold for quantitative features. Let $S \subseteq \{1, 2, \ldots, n\}$ be the set of sample points considered at a given node. S_l is the set of samples in the left child node, and S_r is the set of samples in the right child node. Let $J(S)$ be the cost of the set S. To choose the best split for a given node, we try all possible splits and choose the split that minimizes either the sum or the weighted average of the cost of the child sets, S_l and S_r. More often, we use the weighted average:

$$\frac{|S_l|J(S_l) + |S_r|J(S_r)}{|S_l| + |S_r|},$$

where $|\cdot|$ is the cardinality function, indicating the number of elements in a set.

Entropy Cost Function

There are various choices for the cost function, J. One option is to label a set, S, with the class, $c \in C$, that labels the most points in S and define $J(S)$ as the number of points in S that are not in class c. This is not a good choice of cost function because there are many different splits that all have the same total cost, and we want a cost function that better distinguishes between them.

A better and more common method is to measure the entropy. Let X be some discrete random variable with PMF $p_X(x)$. The **surprise** of X taking on the value x is defined as $-\log_2 p_X(x)$. This can be interpreted as how surprised we are by this probabilistic event occurring. Suppose that X is 1 if the sun rises today and 0 otherwise. The probability that the sun rises today is effectively one, so this event gives us zero surprise. The probability that the sun does not rise today is effectively zero, so this event gives us infinite surprise. The **entropy** is the expected surprise, which can be expressed as

$$H(X) = \sum_{x \in \mathcal{X}} -p_X(x) \log_2 p_X(x).$$

Intuitively, higher entropy indicates more randomness or uncertainty. If X is a Bernoulli random variable with $p = 1$, as in the example of the sun rising, then the entropy is at a minimum of zero, and there is no randomness or uncertainty. If X is a Bernoulli random variable with $p = \frac{1}{2}$, as in the example of a fair coin flip, then the entropy is one, and there is a high amount of randomness and uncertainty. A plot of entropy for a Bernoulli variable is shown in figure 11.2.

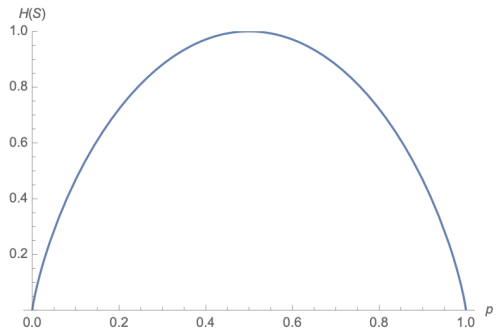

Figure 11.2: This is a plot of the entropy $H(S)$ for a set S with two classes, where p is the probability that a point is in class C and $1 - p$ is the probability that it is not in class C.

If X is a uniform random variable with more than two possible values, then we have even more randomness/uncertainty. For a uniform random variable, as we increase the number of discrete values, we increase the amount of randomness/uncertainty, as indicated by the increasing value of the entropy.

Now that we have a definition of entropy, suppose that the label y is represented by the random variable Y and define $p_c := p_Y(c)$ for all $c \in C$. Generally, we do not know the true underlying distribution of the labels, but we can estimate p_c as the proportion of points in the set S that are in class c:

$$p_c \approx \frac{\big|\{i \in S : y_i = c\}\big|}{|S|}.$$

The entropy of an index set S is then given by

$$H(S) = -\sum_{c \in C} p_c \log_2 p_c.$$

If all the points in S belong to the same class, the entropy is at a minimum of zero. The entropy is at a maximum when the class distribution is uniform. Figure 11.2 shows a plot of the entropy, $H(S)$, for a set, S, with only two classes.

Information Gain Metric

If S is the set of samples considered at a given node, S_l is the set of samples in the left child node after a split, and S_r is the set of samples in the right child node, then the weighted average of the entropy after a split is given by

$$H_{after} = \frac{|S_l|H(S_l) + |S_r|H(S_r)}{|S_l| + |S_r|}.$$

If $H(S)$ is the entropy before the split, the **information gain** is defined as

$$IG(node) = H(S) - H_{after} = H(S) - \frac{|S_l|H(S_l) + |S_r|H(S_r)}{|S_l| + |S_r|}.$$

If we use entropy as the cost for determining the best split, then we choose the split that minimizes H_{after}, which is equivalent to maximizing the information gain. Intuitively, information gain quantifies how much knowledge we gain about the labels of the sample data after a split. Note that the information gain is always non-negative because you cannot become more uncertain after gaining new knowledge. The information gain is generally positive and is only zero if the left and right child nodes have the same class distribution as the parent node. For example, suppose the label of a sample point can either be 0 or 1, and a third of the points in the parent node have label 0, which implies two thirds have label 1. The only way the information gain can be zero is if a third of the points in the left child node have label 0 and a third of the points in the right child node also have label 0. This is demonstrated in figure 11.3.

Because the entropy function (shown on the left in figure 11.3) is strictly concave, the information gain $IG(node) = H(S) - H_{after}$ is non-negative and is only zero if the two child sets both have exactly the same class distributions. Because the percentage of misclassified points (shown on the right in figure 11.3) is concave but not strictly concave, splitting the parent set may not change the weighted

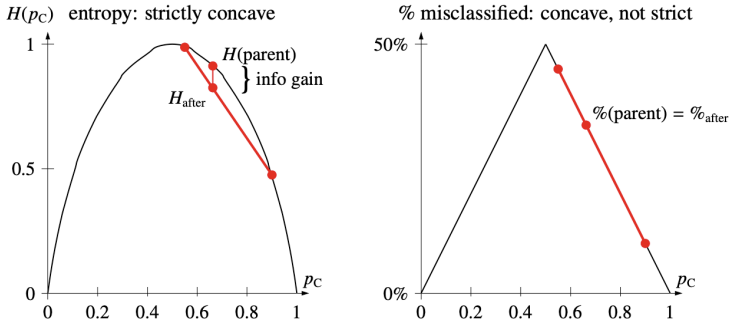

Figure 11.3: The plot on the left is the entropy for a set S that contains only two classes. The plot on the right is the percentage of points that are misclassified. For both plots, p_C is the probability that a point is in class C and $1 - p_C$ is the probability that it is not in C. In both plots, the leftmost dot is $H(S_l)$ and the rightmost dot is $H(S_r)$. The average entropy H_{after} after the split falls on the line connecting $H(S_l)$ and $H(S_r)$ directly beneath the entropy of the parent $H(S)$.

average of the percentage of misclassified points. The bigger problem is that many different splits give the same weighted average cost.

Note that while the entropy is a common and effective cost function to use for decision tree classification, it is not the only function that works well. Many concave functions work fine, but strictly concave functions are generally better.

11.2.4 Choosing the Stopping Criterion

Line 3 of algorithm 3 indicates that we stop growing the decision tree once some stopping condition is met. The most simple stopping scheme is to keep dividing tree nodes until each leaf node is pure, meaning that each leaf node contains only sample points of the same class. If we allow a decision tree to continue subdividing until leaf nodes are pure, it will achieve zero training error for any data set. However, it is often preferable to stop subdividing tree nodes before all the leaf nodes are pure. We will discuss various reasons for stopping before leaf nodes are pure, as well as common stopping criteria. We will also discuss how the stopping condition relates to the bias-variance trade-off.

Reasons to Stop Early

As mentioned, it is often preferable to stop subdividing tree nodes before the leaf nodes are pure. There are various reasons why stopping earlier is preferable:

1. Stopping earlier limits the tree size, which reduces the time required for training and classification and is desirable when we have large data sets.

2. Subdividing nodes until leaf nodes are pure may result in overfitting.
3. For some datasets, it may simply not make sense for leaf nodes to be pure due to noise or overlapping distributions.

Common Stopping Criteria

If we decide that we want to stop subdividing nodes before each leaf node is pure, there are various stopping criteria we can use. We may choose to stop if

1. the next split does not reduce entropy/error enough,
2. most of the node's points (e.g. $> 95\%$) are in the same class,
3. the node contains few sample points (e.g. < 10), or
4. the depth of the tree along that branch is too great.

The required reduction in entropy/error at each split, the maximum percentage of points in a node within a given class, the minimum number of sample points in a node, and the maximum depth of the tree can all be chosen via validation. We can also use validation to decide whether splitting a node improves the validation accuracy. This method is generally most effective, but it can be slow.

Bias-Variance Trade-Off

The stopping criteria directly affects the depth of the decision tree, which is related to the bias-variance trade-off. In general, if a decision tree is very deep, the model has higher variance and lower bias, making it more likely to overfit. Intuitively, if a decision tree is very deep, there are many conditions checked before classifying a test point, which makes the decision rule too fine-grained and sensitive to small perturbations. Consider that if only one of the many conditions is not satisfied, then this might result in a completely different prediction. On the other hand, if the tree is very shallow, the model has high bias and low variance, making it more likely to underfit. In this case, the decision rule is too coarse and the decision tree may not have enough expressive power.

11.2.5 Decision Tree Classification

After choosing a method to select the best split at each internal node and choosing the stopping criterion to stop growing the tree, we train our decision tree with the training data and corresponding labels. Once we have trained our decision tree, we can use it to predict the classes of unseen data. Given an $n' \times d$ matrix of test data, we recursively traverse the decision tree to predict the labels of the data samples, using a function similar to the one in algorithm 4. The top-level call for this function is TraverseTree(S, *root*), where the root node contains the full set of indices: $S = \{1, 2, \ldots, n'\}$.

Algorithm 4: Algorithm for Decision Tree Classification

```
Function TraverseTree(S, node):
    if at leaf node then
        Return predicted label of node
    else
        Split data based on split rule
        TraverseTree(S_l, left child node)
        TraverseTree(S_r, right child node)
    end
End Function
```

For classification problems, if we continue splitting until each leaf node is pure, then the predicted label of each leaf node is simply the class that all of the sample points at that node fall into. If we choose to stop before each leaf node is pure, then leaves that contain multiple classes can either return the majority vote of its sample points or the class posterior probabilities.

11.2.6 Algorithms & Running Times

It is important to consider the computation time required for training and making predictions with a binary decision tree for classification.

Training

Consider an $n \times d$ design matrix containing n sample points with d features each. We assume that any categorical features have been converted to binary features such that d is the number of binary and quantitative features after one-hot encoding categorical features. If all the features are binary, we try $O(d)$ splits at each internal node. If all the features at an internal node with n' sample points are quantitative, we can sort the sample points in $O(n'd)$ time. We can compute the entropy for the first split in $O(n')$, then walk through the list and update the entropy for each successive split in $O(1)$ time, summing to a total of $O(n')$ time for each of the d features. With this trick, the time spent at each internal node is $O(n'd)$, regardless of whether we are working with binary or quantitative features. Each of the n sample points participates in at most $O(h)$ nodes, where h is the depth of the decision tree. Therefore, the total running time is no greater than $O(ndh)$. This is a surprisingly reasonable running time.

Classification

To classify a test point, we move down the tree until we reach a leaf node, then we return the label of that leaf node. The worst case time is $O(h)$, where h is the depth of the decision tree. If all of the features are binary, the depth is no greater than the number of features d. If we have quantitative features, the decision trees may be deeper than the number of features. In practice, the depth is usually no greater than $O(\log n)$, but this is not always true.

11.3 Decision Tree Variations

11.3.1 Regression

Previously, we discussed how to use binary decision trees for classification. We can also use decision trees to find a piecewise constant regression function. Given a set of sample point indices, S, instead of using the entropy, $H(S)$, as the cost function, we will define the cost of S as

$$J(S) = \frac{1}{|S|} \sum_{i \in S} (y_i - \mu_S),$$

where μ_S is the mean label for the sample points in S. To find a piecewise constant regression function, at each internal node, we choose the split that minimizes the weighted average costs of the children after the split.

For classification problems, we can continue splitting until each leaf node is pure and each leaf will return the label of its sample points, or we can choose to stop early and leaves that contain multiple classes can return the majority vote of its sample points or the class posterior probabilities. For regression problems, we will generally never continue splitting until each leaf node is pure. Instead, a leaf node should return the average label of its sample points.

11.3.2 Pruning

When discussing stopping criteria, we said that in the basic stopping scheme, we keep dividing nodes until each leaf node is pure. We also provided other stopping criteria that can be used to stop before each leaf node is pure. **Pruning** is a better alternative to stopping early that also works to limit overfitting. To implement pruning, we grow the decision tree until all leaf nodes are pure. Then, we greedily remove each split whose removal improves validation performance. Pruning is often more effective than stopping early because a split that does not seem to provide much information gain may be followed by a split that does.

11.3.3 Multivariate Splits

We previously assumed that our decision tree would always split on a single feature. We can find splits that are not axis-aligned by using other classification algorithms or by generating them randomly. Multivariate splits may improve the accuracy of the decision tree, but they may result in less interpretability and/or speed. Multivariate splits could allow us to find a linear decision boundary, instead of a stair-step boundary, as shown in the example in figure 11.4.

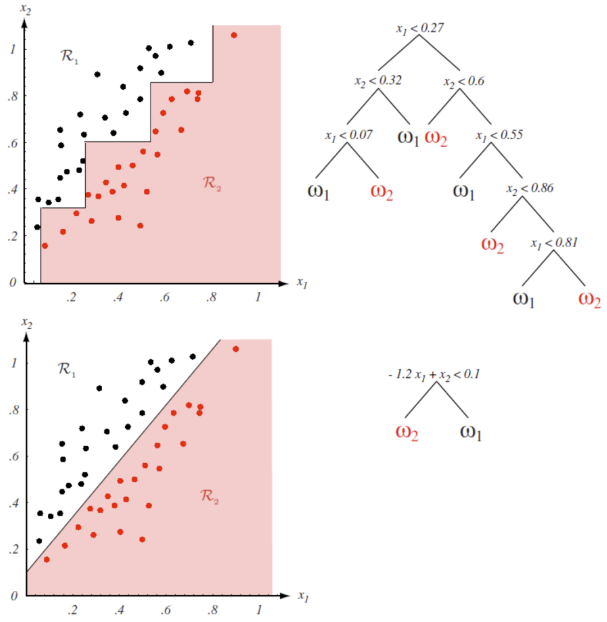

Figure 11.4: As shown in the top image, a decision tree needs many axis-aligned splits to approximate a diagonal linear decision boundary, but the boundary can be approximated with a single multivariate split, as shown in the bottom image.

Chapter 12

Nearest Neighbors Classifier

12.1 Overview of Nearest Neighbors

Suppose that we want to estimate the label of a query point, $\boldsymbol{q}$. One approach is to look at the labels of the sample points closest to this query point. The ***k* nearest neighbors** (***k*-NN**) classifier finds the k sample points closest to $\boldsymbol{q}$, using some choice of distance metric. If the labels of the sample points are quantitative, then the average label of the k nearest points is used to estimate the label of q. If the labels of the sample points are categorical, then the most popular class among the k nearest points is used to estimate the label of the query point. For classification tasks, we could also return a histogram of class probabilities, which tries to estimate the posterior probabilities of the classes from the k sample points. However, the accuracy of these estimates depends on the value of k. This method works best when you have a large amount of data.

12.1.1 Tuning the Hyperparamter

In general, the decision boundary of the k-NN classifier becomes more smooth as the value of the hyperparameter, k, increases. For smaller values of k, we have low bias and high variance, so we risk overfitting. For larger values of k, we have low variance and high bias, so we risk underfitting. Generally, the ideal value of k depends on how dense the training data is. As the data gets denser, it is best to increase the value of k. The impact of the hyperparameter, k, on the decision boundary for a k-NN classifier is demonstrated in figure 12.1.

12.1.2 Performance of Nearest Neighbors

If you have a large amount of training data, then the nearest neighbors classifier can work quite well. If the training and test points are drawn independently from the same probability distribution, then as the number of sample points, n, approaches infinity, the 1-NN error rate is strictly less than $2B - B^2$, where B is the Bayes risk. If we are only working with two classes, then as n approaches

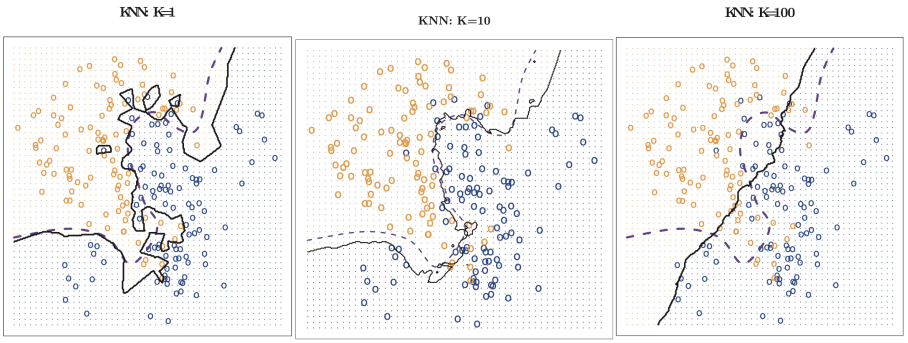

Figure 12.1: This figure depicts the decision boundary of the k nearest neighbors classifier for three choices of k: $k = 1$ (left), $k = 10$ (center), and $k = 100$ (right). In all three images, the solid line is the decision boundary of the k-NN classifier and the dotted line is the Bayes decision boundary. Notice that the 1-NN classifier results in very a non-smooth boundary with islands that gives us zero training error but does not generalize well. This classifier is badly overfitting the data. The 100-NN classifier results in a smooth boundary that is close to linear and does not represent the data well. This classifier is badly underfitting the data. The 10-NN classifier does better than the other two classifiers, finding a boundary that is reasonably close to the Bayes decision boundary.

inifinity, the 1-NN error rate is less than or equal to $2B - 2B^2$. Additionally, as the number of sample points, n, and the hyperparameter, k, approach infinity such that k/n approaches zero, the k-NN error rate converges to the Bayes risk.

12.2 Nearest Neighbor Algorithms

There are various algorithms used by the k-NN classifier. We will consider the exhaustive k-NN algorithm, an algorithm using Voronoi diagrams, and an algorithm that uses data structures referred to as k-d trees.

12.2.1 Exhaustive k-NN Algorithm

In the **exhaustive *k*-NN algorithm**, we scan through all n sample points, computing the squared distance from the current point to the query point, $\boldsymbol{q}$. As we scan, we maintain a max-heap of the k nearest neighbors seen so far, which are keyed by their corresponding squared distances. When we encounter a sample point closer to $\boldsymbol{q}$ than the point at the top of the heap, we remove the top point and insert the new point into the heap based on its squared distance.

The time to train the classifier is $O(0)$, and the time to make a prediction for a single query point is $O(nd+n \log k)$, where d is the number of features. Interestingly, if we scan through the sample points in random order, the expected time

to make a prediction for a single query point is $O(nd + k \log n \log k)$. However, this is generally not recommended because it may result in cache misses.

In some cases, we can preprocess training points to obtain sublinear prediction time. If we have a low number of features ($d \leq 5$), we can use Voronoi diagrams. If we have a medium number of features ($6 \leq d \leq 30$), we can use k-d trees. If we have a high number of dimensions ($d > 30$), the exhaustive k-NN features is fastest, but we can speed it up using PCA (dicussed in section 14).

12.2.2 Voronoi Diagrams

Let $\mathcal{X}$ be a set of n sample points with d features, which can be expressed as

$$\mathcal{X} = \{\boldsymbol{x_i} \in \mathbb{R}^d,\ i = 1, \ldots, n\}.$$

The **Voronoi cell** of some point $\boldsymbol{w} \in \mathcal{X}$ is defined as the set of points in the feature space that are closer to $\boldsymbol{w}$ than to any other point in the set $\mathcal{X}$:

$$\text{Vor}(\boldsymbol{w}) = \{\boldsymbol{p} \in \mathbb{R}^d : ||\boldsymbol{p} - \boldsymbol{w}||_2 \leq ||\boldsymbol{p} - \boldsymbol{x_i}||_2,\ \forall \boldsymbol{x_i} \in \mathcal{X}\}.$$

Note that the Voronoi cell is always a convex polyhedron or polytope. The **Voronoi diagram** of $\mathcal{X}$ is the set of all the Voronoi cells for $\mathcal{X}$. In the worst case, the size (i.e. number of vertices) of a Voronoi diagram is $O(n^{\lceil d/2 \rceil})$. In practice, the size of the Voronoi diagram is often $O(n)$. Figure 12.2 provides an example of what a Voronoi diagram might look like in two dimesnions.

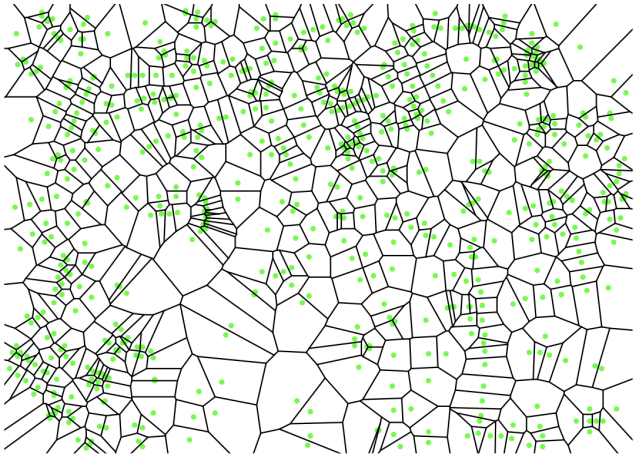

Figure 12.2: An example of a Voronoi diagram for a set containing two-dimensional sample points. Each green dot is a sample point, and the polyhedrons are the Voronoi cells for each point.

To use a Voronoi diagram for nearest neighbors classification, we also need a data structure that can perform point location. In point location, we are given

a query point, $\boldsymbol{q} \in \mathbb{R}^d$, and we want to find the point, $\boldsymbol{w} \in \mathcal{X}$, for which $\boldsymbol{q} \in \text{Vor}(\boldsymbol{w})$. If we only have two dimensions, then we should use a trapezoid map for point location. For two dimensions, the time to compute the Voronoi diagram is $O(n \log n)$ and the time to make a prediction for a query point using a trapezoidal map is $O(n \log n)$. If we are working with sample points that have more than two dimensions, then we should use a binary space partition (BSP) tree for point location. Unfortunately, it is difficult to characterize the running time for this strategy, but it is likely to be reasonably fast in 3-5 dimensions.

As an important note, the standard Voronoi diagram only supports nearest neighbor classification with $k = 1$. If you want to use $k > 1$, then you can use an order-k Vornoi diagram, which has a cell for each possible k nearest neighbors. However, order-k Vornoi diagrams are not commonly used in practice.

12.2.3 k-d Trees

While Voronoi diagrams are useful for 1-NN classification in low dimensions, ***k*-d trees** are much simpler and generally faster in 6 or more dimensions. The k-d trees are essentially decision trees used for nearest neighbor search. The major differences between k-d trees and decision trees are listed below.

1. In k-d trees, we choose the feature to split on based on the feature with the greatest width, meaning that we choose feature $\hat{i}$ such that

$$\hat{i},\ \hat{j},\ \hat{k} = \arg\max_{i,j,k} \big(X_{ji} - X_{ki}\big).$$

   Another option to avoid computing the greatest width is to rotate through the features, meaning that we split on the first feature at depth one, the second feature at depth two, and so on. This allows us to build the tree faster by a factor of $O(d)$.

2. To choose the threshold value of the feature we split on, we can either choose the median point for feature $\hat{i}$ or the box center: $\frac{1}{2}(X_{\hat{j}\hat{i}} - X_{\hat{k}\hat{i}})$. Choosing the median point as the threshold value guarantees $\lfloor \log_2 n \rfloor$ tree depth and $O(nd \log n)$ time to build the tree. If we rotate through the features instead of choosing the feature with the greatest width, then the time required to build the tree is just $O(n \log n)$. If we choose to split at the box center, instead of the median, then we improve the aspect ratio of the boxes, but this may unbalance our tree. One strategy for building k-d trees is to alternate between choosing the median and choosing the center as the threshold, which also guarantees an $O(\log n)$ tree depth.

3. Each internal node stores a single sample point that lies in the node's box. Usually we choose to store the splitting point for that node.

Figure 12.3 provides an example of a k-d tree for a two-dimensional dataset.

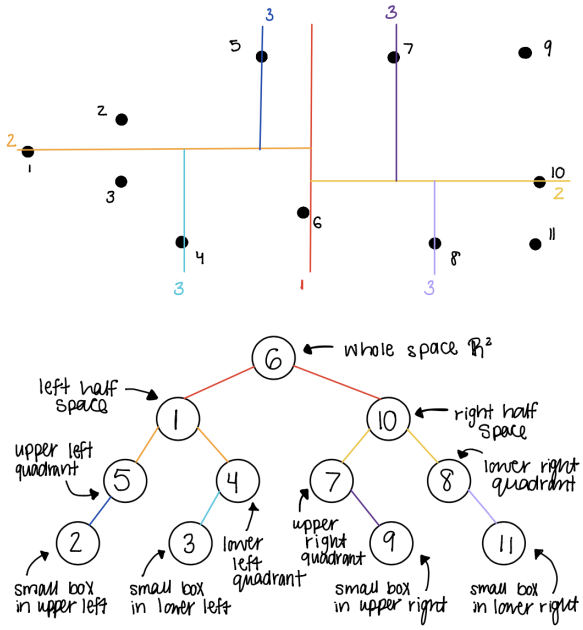

Figure 12.3: The top image depicts 11 two-dimensional sample points, and the bottom images provides a possible k-d tree for the given training data. Starting with the entire feature space, $\mathbb{R}^2$, we first split on the x feature and store the median sample point 6 in the root node. We then split the left half space on the y feature and store the median 1 and split the right half space on the y feature and store the median 10. We then split the upper left quadrant on the x feature and store 5, split the lower left quadrant on x and store 4, split the upper right quadrant on x and store 7, and split the lower right quadrant on x and store 8. Note that each subtree represents an axis-aligned box. The entire tree is the feature space $\mathbb{R}^2$, the first left subtree is the left half space, the first right subtree is the right half space, the next set of subtrees are quadrants, and the final set of subtrees (which are simply leaf nodes) are smaller boxes in $\mathbb{R}^2$.

Suppose that given a query point, $\boldsymbol{q}$, the sample point $\boldsymbol{s} \in \mathcal{X}$ is the closest point in the training set to $\boldsymbol{q}$. Rather than searching for the nearest neighbor, $\boldsymbol{s}$, we can relax the nearest neighbors problem to find a sample point $\boldsymbol{w} \in \mathcal{X}$ such that

$$||\boldsymbol{q} - \boldsymbol{w}||_2 \leq (1 + \epsilon)||\boldsymbol{q} - \boldsymbol{s}||_2,$$

where ϵ is some small positive constant. If we choose $\epsilon = 0$, then we get back the exact nearest neighbors problem. If we are working with sample points in high dimensions, then it is a good idea to solve the approximate nearest neighbors

problem because it can greatly decrease the computation time.

To solve this problem using k-d trees, we keep track of the nearest sample point (or k nearest sample points) found so far and maintain a binary min-heap of unexplored subtrees, which are keyed by the distance from $\boldsymbol{q}$. As we move through the tree, the distance to the nearest sample point found decreases and the distance to the nearest unexplored subtree increases. The query tries to avoid searching most of the boxes/subtrees by searching the boxes closest to the query point first. The search stops when the distance from $\boldsymbol{q}$ to the kth nearest neighbor found so far is less than or equal to the distance from $\boldsymbol{q}$ to the nearest unexplored box multiplied by the factor $1 + \epsilon$. At this point, we have found one point within a distance $(1 + \epsilon)r$ from the query point, where r is the distance from the query point to its true nearest neighbor. Figure 12.4 provides an example of the search process for a query point, $\boldsymbol{q}$, using a k-d tree.

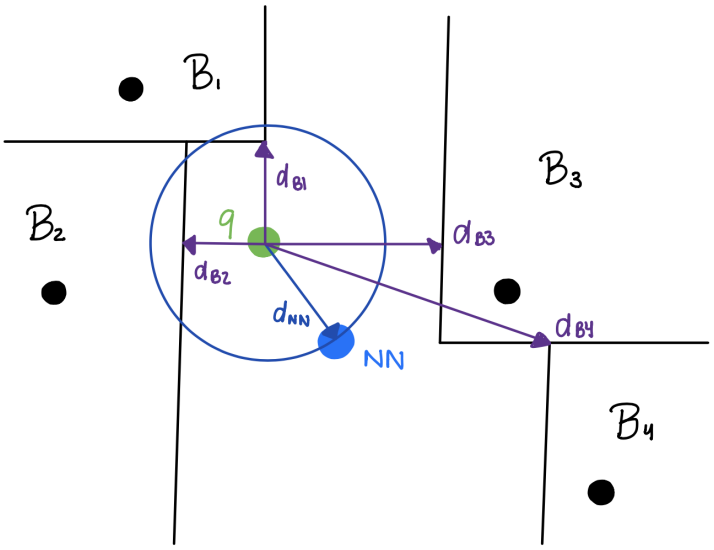

Figure 12.4: In the figure above, the query point, $\boldsymbol{q}$, is shown in green, and the closest sample point seen so far is shown in light blue. There are four unexplored boxes/subtrees: B_1, B_2, B_3, and B_4. The distance, d_{NN}, from the query point to the nearest sample point is shown in dark blue, and the distance, d_{Bi}, from the query point to the ith box/subtree is shown in purple. Because we only explore boxes/subtrees that are closer to the query point q than the nearest neighbor (NN) seen so far, we need to search boxes B_1 and B_2 for closer sample points, but we do not need to search boxes B_3 and B_4.

To use a k-d tree for nearest neighbors classification, we start by initializing a binary min-heap, Q, of unexplored trees keyed by the distance from the query

point, $\boldsymbol{q}$, the nearest neighbor (NN) found so far, and the distance, d_{NN}, to the nearest neighbor found so far. While there are still unexplored subtrees sufficiently closer to the query point than the nearest neighbor found so far, we determine the closest unexplored box/subtree, B. We then remove this subtree from the heap and look at the single sample point, $\boldsymbol{w}$, stored in the node B. If the distance from the query point to the sample point is less than than the distance to the nearest neighbor found so far, then $\boldsymbol{w}$ is the new nearest neighbor and we store its distance from $\boldsymbol{q}$. We then look at the left and right child nodes of the current subtree. If a child node is sufficiently closer to the query point than the nearest neighbor seen so far, we insert this node in the heap of unexplored subtrees with the distance from $\boldsymbol{q}$ to the given child node as its key. Once there are no more unexplored subtrees sufficiently closer than the nearest neighbor seen so far, we return the nearest neighbor (NN). Algorithm 5 describes the process to find the first nearest neighbor to a query point, $\boldsymbol{q}$, using a k-d tree.

Algorithm 5: 1-NN Query Using k-d Tree

$Q \leftarrow$ heap containing root node with key 0
$NN \leftarrow$ None
$d_{NN} \leftarrow \infty$
while Q *not empty and* $(1+\epsilon)\min_{node \in nodes(Q)} key(node) < d_{NN}$ **do**
  $B \leftarrow \arg\min_{\text{node}\in\text{nodes(Q)}} \text{key(node)}$
  $Q \leftarrow Q \setminus B$
  $w \leftarrow \text{data}(B)$
  if $dist(q, w) < d_{NN}$ **then**
    $NN \leftarrow w$
    $d_{NN} \leftarrow \text{dist}(q, w)$
  end
  $B_l \leftarrow$ left child node of B
  if $(1+\epsilon)\, dist(q, data(B_l)) < d_{NN}$ **then**
    $\text{insert}\big(Q, B_l, \text{dist}(q, \text{data}(B_l))\big)$
  end
  $B_r \leftarrow$ right child node of B
  if $(1+\epsilon)\, dist(q, data(B_r)) < d_{NN}$ **then**
    $\text{insert}\big(Q, B_r, \text{dist}(q, \text{data}(B_r))\big)$
  end
end
return NN

Note that we are not limited to using the Euclidean distance and could use any l_p norm as the distance function. If we want to find the k nearest neighbors, instead of just the first nearest neighbor, we replace NN and d_{NN} in the algorithm with a max-heap holding the k nearest neighbors seen so far keyed by their distances to the query point. As a final note, some useful software for performing nearest neighbors classification with k-d trees is ANN, FLANN, and GeRaF.

Chapter 13

Neural Networks

13.1 Overview of Neural Networks

Neural networks (NNs) are a very popular supervised learning technique that can effectively approximate nonlinear relationships between data and labels. They are more commonly used for classification problems but can easily be used for regression as well. A NN model is defined by the following components:

1. **Architecture** – The model architecture specifies the flow of information between the network layers, which defines the composition of functions that the network performs from input to output.

2. **Cost/loss function** – The model aims to minimize a cost/loss function, which depends on the true labels and predicted labels. The most commonly used loss functions are the mean squared error and the cross-entropy loss, which are discussed in section 13.1.1.

3. **Optimization algorithm** – The optimization algorithm is used to minimize the cost/loss function. Often we use stochastic or batch gradient descent, but other algorithms may be used as well.

4. **Hyperparameters** – Our neural network depends on a set of hyperparamters, including the learning rate and batch size among others.

Each layer of a neural network is defined by the following components:

1. **Parameterized function** – The parameterized function defines the layer's map from input to output. Most often, we use an affine function combined with a nonlinear **activation function** (i.e. $f(\boldsymbol{x}) = \sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$, where $\boldsymbol{x}$ is the input, $\boldsymbol{W}$ is the weight matrix, $\boldsymbol{b}$ is the bias vector, and σ is the activation function). There are various choices for the nonlinear activation function. Some common ones are the sigmoid, ReLU, softmax, and hyperbolic tangent functions, which are discussed in section 13.1.2.

2. **Parameters** – Each layer is composed of a set of parameters. If we are using the affine function combined with a nonlinear activation function, then our parameters include the weights, $\boldsymbol{W}$, and biases, $\boldsymbol{b}$.

13.1.1 Loss & Cost Functions

Suppose we have a single data sample, $\boldsymbol{x} \in \mathbb{R}^d$, and a corresponding vector of true labels, $\boldsymbol{y} \in \mathbb{R}^k$, where k is the number of classes/output features. The neural network outputs a prediction $\hat{\boldsymbol{y}} \in \mathbb{R}^k$, which predicts the values of $\boldsymbol{y}$.

Loss Function

To train a neural network, we need to pick a loss function, $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$, which maps predicted labels, $\hat{\boldsymbol{y}} \in \mathbb{R}^k$, and true labels, $\boldsymbol{y} \in \mathbb{R}^k$, to some non-negative value. Below are the two most commonly used loss functions:

1. **Squared error loss** – The squared error loss function is defined such that

$$L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = ||\hat{\boldsymbol{y}} - \boldsymbol{y}||_2^2.$$

   The gradient of the loss with respect to the predicted values is given by

$$\nabla_{\hat{y}} L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = 2(\hat{\boldsymbol{y}} - \boldsymbol{y}).$$

   Note that the squared error loss is more often used for regression problems because it gives a measure of the distance between real-valued labels.

2. **Cross-entropy loss** – The cross-entropy loss function, which is also referred to as the **logistic loss** function, is defined such that

$$L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{j=1}^{k} y_j \ln \hat{y}_j.$$

   The gradient of L with respect to $\hat{y}$ can be expressed component-wise as

$$\frac{\partial}{\partial \hat{y_j}} L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\frac{y_j}{\hat{y}_j}.$$

   Note that the cross-entropy loss function is generally used for classification problems because it assumes that inputs are probability distributions over classes. When using the cross-entropy loss for classification, it is strongly recommend that the true labels are defined such that

$$\sum_{j=1}^{k} y_j = 1.$$

   Typically, we chose the labels to be **one-hot vectors** defined such that

$$y_j = \begin{cases} 1 & \text{if } \boldsymbol{x} \in \text{class } j \\ 0 & \text{otherwise} \end{cases}.$$

Cost Function

In addition to the loss function defined for a single sample, we choose a cost function, $J(h)$, to map a hypothesis function, h, to the total cost over n samples. The predicted label is given by $\hat{\boldsymbol{y}}_{\boldsymbol{i}} = h(\boldsymbol{x}_{\boldsymbol{i}})$, where $\boldsymbol{x}_{\boldsymbol{i}}$ is the ith data sample. The most commonly used cost function is the **mean cost**, which is given by

$$J(h) = \frac{1}{n}\sum_{i=1}^{n} L(h(\boldsymbol{x}_{\boldsymbol{i}}), y_i).$$

The gradient of the mean cost with respect to the prediction $\hat{y}_i = h(\boldsymbol{x}_{\boldsymbol{i}})$ is

$$\nabla_{\hat{y}_i} J(h) = \frac{1}{n}\frac{\partial L}{\partial \hat{y}_i}.$$

13.1.2 Activation Functions

As mentioned previously, each layer of the neural network generally maps an input, $\boldsymbol{x} \in \mathbb{R}^d$, to an output defined by $f(\boldsymbol{x}) = \sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$, where $W \in \mathbb{R}^{k\times d}$ is a weight vector, $b \in \mathbb{R}^k$ is a bias vector, and $\sigma : \mathbb{R}^k \to \mathbb{R}^k$ is a nonlinear activation function. We will define $\boldsymbol{z} := \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$ as the intermediate vector and $\boldsymbol{h} := f(\boldsymbol{x})$ as the output vector. The relationship between the intermediate vector and the output is given by $\boldsymbol{h} = \sigma(\boldsymbol{z})$, where σ is applied component-wise to $\boldsymbol{z}$. Below are some of the most common nonlinear activation functions:

1. **Sigmoid/logistic function**

   If σ is the sigmoid/logistic function, the ith element of the output is

   $$h_i = s(z_i) = \frac{1}{1 + e^{-z_i}}.$$

   The Jacobian of $\boldsymbol{h}$ with respect to $\boldsymbol{z}$ can be expressed component-wise as

   $$\left[\boldsymbol{D}_{\boldsymbol{z}}\boldsymbol{h}\right]_{ij} = \frac{\partial h_i}{\partial z_j} = \begin{cases} h_i(1 - h_i) & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}.$$

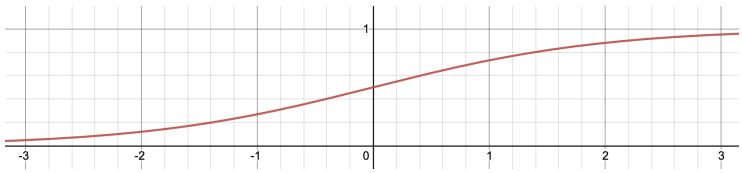

Figure 13.1: This is a graph of the output of the sigmoid/logistic function $h_i = s(z_i)$ over the values of z_i. Notice that $h_i \in (0, 1)$.

As demonstrated in figure 13.1, the output of the sigmoid/logistic activation function is always between zero and one. Therefore, it is commonly

used at the output layer of neural networks used for two-class prediction problems with $k = 1$. The output of the network can then be interpreted as the probability that a data sample is in the class of interest.

2. **Softmax function**

   If σ is the softmax function, the ith element of the output is

$$h_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}.$$

   This definition of the softmax function is numerically unstable, so more often, we define a modified version of this function such that

$$h_i = \frac{e^{z_i - m}}{\sum_{j=1}^{k} e^{z_j - m}} \quad \text{where} \quad m := \max_{j \in \{1,\dots,k\}} z_j.$$

   Whether we use the true softmax function or the stable softmax function, the Jacobian of $\boldsymbol{h}$ with respect to $\boldsymbol{z}$ can be expressed component-wise as

$$\left[\boldsymbol{D}_{\boldsymbol{z}} \boldsymbol{h}\right]_{ij} = \frac{\partial h_i}{\partial z_j} = \begin{cases} h_i(1 - h_i) & \text{if } i = j \\ -h_i h_j & \text{otherwise} \end{cases}.$$

   Like the sigmoid function, the softmax function is commonly used in the output layer of neural networks. The softmax function outputs k values in the range $(0, 1)$ that add up to one, so it can be used to represent the probability distribution over k discrete classes. For this reason, neural networks that use the softmax activation function at the output can be used for multi-class prediction. We can then interpret h_i as the probability that a data sample belongs to class i. Note that if we only have two classes, we can equivalently use the sigmoid function as discussed previously.

3. **Rectified linear unit (ReLU) function**

   If σ is the ReLU function, which we also refer to as the **ramp function** or **hinge function**, then we can express the ith element of the output as

$$h_i = \max\{0, z_i\} = \begin{cases} z_i & \text{if } z_i \geq 0 \\ 0 & \text{otherwise} \end{cases}.$$

   The Jacobian of $\boldsymbol{h}$ with respect to $\boldsymbol{z}$ can be expressed component-wise as

$$\left[\boldsymbol{D}_{\boldsymbol{z}} \boldsymbol{h}\right]_{ij} = \frac{\partial h_i}{\partial z_j} = \begin{cases} 1 & \text{if } i = j,\ z_i \geq 0 \\ 0 & \text{otherwise} \end{cases}.$$

   The ReLU function is popular for neural networks with many hidden layers and large training sets because its derivative is very fast to compute.

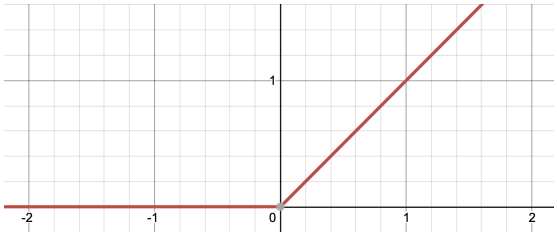

Figure 13.2: This is a graph of the output of the ReLU function $h_i = \max\{0, z_i\}$ over the values of z_i.

4. **Hyperbolic tangent function**

   If σ is the hyperbolic tangent function, the ith element of the output is

$$h_i = \tanh(z_i) = \frac{e^{z_i} - e^{-z_i}}{e^{z_i} + e^{-z_i}}.$$

   The Jacobian of $\boldsymbol{h}$ with respect to $\boldsymbol{z}$ can be expressed component-wise as

$$\left[\boldsymbol{D}_{\boldsymbol{z}}\boldsymbol{h}\right]_{ij} = \frac{\partial h_i}{\partial z_j} = \begin{cases} 2s(2z_i) - 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases},$$

   where s is the sigmoid/logistic function defined previously.

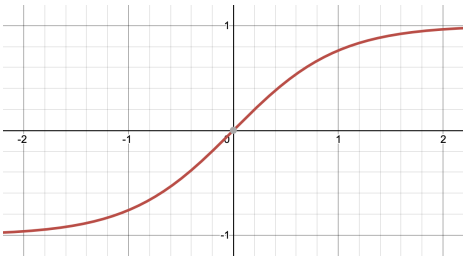

Figure 13.3: This is a graph of the output of the hyperbolic tangent function $h_i = \tanh(z_i)$. Notice that $h_i \in (-1, 1)$.

13.1.3 Backpropagation

To learn the function that relates inputs to labels, we generally use mini-batch gradient descent with backpropagation to update the parameters of the network. In mini-batch gradient descent, we operate on subsets of the data matrix. In the **backpropagation** algorithm, we first compute the forward pass of the network, during which we send a mini-batch of input data through the network. The result is a set of outputs, which we use to compute our cost/loss function. We then take the derivatives of this cost with respect to the parameters of each layer,

starting with the output of the network and using the chain rule to propagate backwards through the layers. This is called the backward pass. By starting at the output layer and propagating backwards, we can reuse computed derivatives to avoid computing the same derivatives multiple times. Backpropagation is an example of a dynamic programming algorithm that has a time complexity that is linear in the number of layers in the network, making it efficient.

13.2 Multilayer Perceptrons (MLPs)

A **multilayer perceptron (MLP)**, which we also refer to as a **feed-forward, fully-connected neural network**, is a network composed of multiple layers of perceptrons (recall section 3.3) with threshold activation functions. Each layer of an MLP performs an affine transformation of an input, followed by a nonlinear activation function. Figure 13.4 shows the basic structure of an MLP.

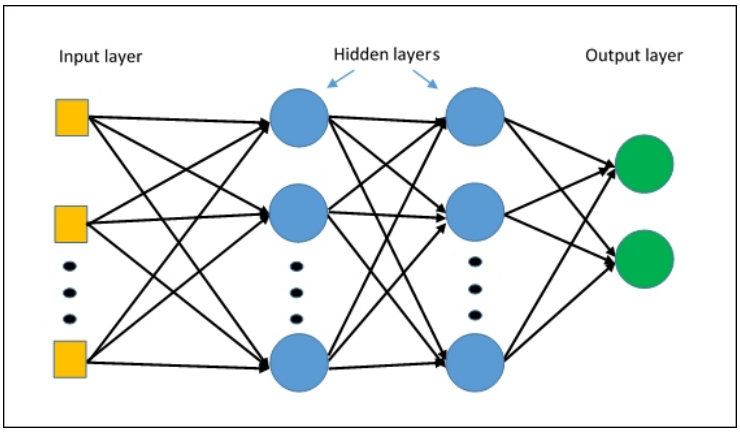

Figure 13.4: This is an example of a three-layer multilayer perceptron (MLP), which has an input layer, an output layer, and two hidden layers. Each layer may contain any number of nodes.

While each individual perceptron is a linear classifier, by combining individual perceptrons in layers and introducing nonlinear activation functions, multilayer perceptrons can learn arbitrary decision functions. Because they can capture any classification boundary, multilayer perceptrons are referred to as universal classifiers. This is a useful property, but other learning algorithms, such as decision trees, are also considered universal classifiers.

13.2.1 Fully-Connected Layer

Multilayer perceptrons (MLPs) are composed of any number of fully-connected layers. A fully-connected layer is composed of the following elements:

1. $\boldsymbol{X} \in \mathbb{R}^{b \times d}$ – A mini-batch of b data samples, each with d features, which is the input to the first layer of the neural network.
2. $\boldsymbol{Y} \in \mathbb{R}^{b \times k}$ – The true labels corresponding to each of the b data samples, which have k possible classes/output features.
3. $n^{[l]}$ – The number of nodes in layer l. Note that in the first layer, $n^{[-1]} = d$, where d is the number of input features.
4. $\boldsymbol{W^{[l]}} \in \mathbb{R}^{n^{[l-1]} \times n^{[l]}}$ – A matrix of weights connecting layer $l-1$ to layer l.
5. $\boldsymbol{b^{[l]}} \in \mathbb{R}^{n^{[l]}}$ – A vector of bias terms connecting layer $l-1$ to layer l.
6. $\sigma^{[l]}$ – The nonlinear activation function applied at layer l.
7. $\boldsymbol{Z^{[l]}} \in \mathbb{R}^{b \times n^{[l]}}$ – The intermediate output of layer l before applying the activation function, which is defined such that

   $$\boldsymbol{Z^{[l]}} = \boldsymbol{H^{[l-1]}}\boldsymbol{W^{[l]}} + \mathbf{1}_b\big(\boldsymbol{b^{[l]}}\big)^T.$$

   Note that in the first layer, $\boldsymbol{H^{[-1]}}$ is simply the input data sample, $\boldsymbol{X}$.
8. $\boldsymbol{H^{[l]}} \in \mathbb{R}^{b \times n^{[l]}}$ – The output of layer l, which is defined such that

   $$\boldsymbol{H^{[l]}} = \sigma^{[l]}(\boldsymbol{Z^{[l]}}) = \sigma^{[l]}\Big(\boldsymbol{H^{[l-1]}}\boldsymbol{W^{[l]}} + \big(\boldsymbol{b^{[l]}}\big)^T\Big).$$

9. $\hat{\boldsymbol{Y}} \in \mathbb{R}^{b \times k}$ – The predicted labels for b data samples with k possible classes/output features, which is the output of the final layer.

Figure 13.5 shows a detailed example of a three-layer MLP with the components labeled. In this example, we assume we have a single input data sample, $\boldsymbol{x}$, with $d = 4$ features and a corresponding label, $\boldsymbol{y}$, with $k = 3$ possible classes/output features. The output of the MLP is a k-dimensional prediction, $\hat{\boldsymbol{y}}$.

13.2.2 Forward Pass

In the forward pass of a feed-forward, fully-connected neural network, we compute the output, $\boldsymbol{H^{[l]}}$, of each layer sequentially, using the function defined for each layer of the network. The output of one layer is used as the input to the next, thus a neural network is a composition of functions. Within each layer, we cache the input $\boldsymbol{H^{[l-1]}}$ and the output $\boldsymbol{H^{[l]}}$ to be used in the backward pass.

13.2.3 Backward Pass

In the backward pass of a feed-forward, fully-connected neural network, we compute the derivatives of the downstream cost, J, with respect to the input, $\boldsymbol{H^{[l-1]}}$, the weights, $\boldsymbol{W^{[l]}}$, and the bias, $\boldsymbol{b^{[l]}}$, for each layer of the network. Recall that in the backward pass, we propagate gradients backwards through the network. We start by computing the gradients in the output layer and cache these values to be used when computing the gradients in earlier layers.

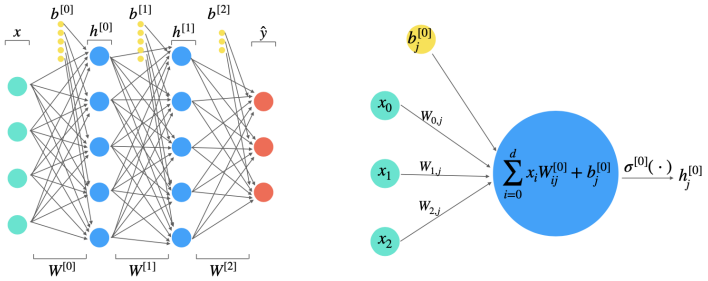

Figure 13.5: On the left is a three-layer perceptron. The data sample, $\boldsymbol{x} \in \mathbb{R}^4$, is the input to the neural network, and the predicted label, $\hat{\boldsymbol{y}} \in \mathbb{R}^3$, is the output. In the first layer, we compute $\boldsymbol{h^{[0]}} = \sigma^{[0]}\left(\boldsymbol{x}\boldsymbol{W^{[0]}} + \boldsymbol{b^{[0]}}\right)$. In the next layer, we compute $\boldsymbol{h^{[1]}} = \sigma^{[1]}\left(\boldsymbol{h^{[0]}}\boldsymbol{W^{[1]}} + \boldsymbol{b^{[1]}}\right)$. In the final layer, we compute $\hat{\boldsymbol{y}} = \sigma^{[2]}\left(\boldsymbol{h^{[1]}}\boldsymbol{W^{[2]}} + \boldsymbol{b^{[2]}}\right)$. On the right is a single fully connected neuron, demonstrating the computations that are performed component-wise.

13.3 Convolutional Neural Networks (CNNs)

Convolutional neural networks (**CNNs**), which are also referred to as **ConvNets**, are a popular variation of neural networks used for image and audio processing. Images and audio files contain very large inputs, which make it impractical to train or use an MLP for image or audio processing. To handle very large inputs, we use CNNs, which depend on the following ideas:

1. Local connectivity – A hidden unit in an early layer connects only to a small patch of units in the previous layer.

2. Shared weights – Groups of hidden units share the same set of input weights, which are called a **kernel**/**filter**/**mask**. In CNNs, we learn several kernels, where each kernel operates on every patch of an image. The number of hidden units in the first hidden layer is equal to the number of kernels multiplied by the number of patches. We can think of hidden units as learned features. CNNs learn features from multiple patches simultaneously, then apply those features everywhere. For images, filters in early layers tend to include edge detectors. If a network learns to detect edges in one patch of the input, then every patch now has an edge detector. In this way, CNNs exploit the repeated structure in images and audio.

CNNs are composed of **convolutional layers** (discussed in section 13.3.1) and **pooling layers** (discussed in section 13.3.2). CNNs generally combine convolutional layers with pooling layers to progressively shrink the spatial size of the input until is is small enough to be fed into an MLP for classification. Figure 13.6 shows an example of the typical architecture of a CNN.

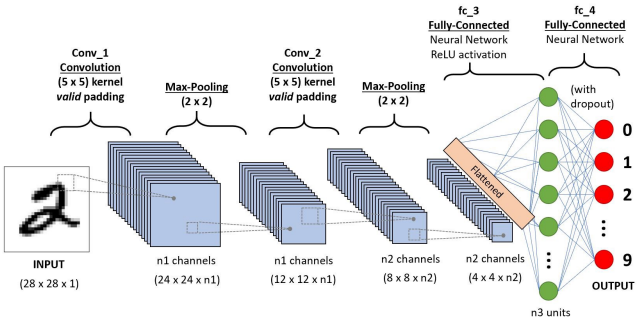

Figure 13.6: Given an input image, this CNN predicts the digit (0-9). It feeds the image through a convolutional layer, followed by a pooling layer, then another convolutional layer, followed by another pooling layer. It is then flattened and fed into a fully-connected neural network, which outputs the predicted digit.

13.3.1 Convolutional Layer

In a **convolutional layer**, each kernel is convolved with the input across every channel. A convolutional layer is composed of the following elements:

1. $\boldsymbol{X} \in \mathbb{R}^{b \times d_1 \times d_2 \times c}$ – A mini-batch of b input tensors with c channels whose heights are d_1 and whose widths are d_2.
2. $\boldsymbol{Y} \in \mathbb{R}^{b \times k}$ – The true label vector with k classes/output features corresponding to the b data samples.
3. $n^{[l]}$ – The number of channels in layer l. In the first layer, $n^{[-1]} = c$, where c is the number of input channels.
4. $\boldsymbol{W^{[l]}} \in \mathbb{R}^{k_1 \times k_2 \times n^{[l-1]} \times n^{[l]}}$ – A tensor of kernel weights at layer l, where k_1 is the height of the kernel and k_2 is the width of the kernel for the given layer. The pair (k_1, k_2) is commonly referred to as the **kernel size**.
5. $\boldsymbol{b^{[l]}} \in \mathbb{R}^{n^{[l]}}$ – A vector of bias terms for layer l.
6. $\sigma^{[l]}$ – The nonlinear activation function applied at layer l.
7. (s_1, s_2) – The size of the step taken in the convolution operation at the given layer, where s_1 is the stride length in the first dimension and s_2 is the stride length in the second dimension.
8. (p_1, p_2) – The amount of zero padding applied to the input tensor at the given layer, where the top and bottom of the input is padded with p_1 zeros, and the left and right of the input is padded with p_2 zeros. For the input tensor, the padded input is of the shape $b \times (d_1 + 2p_1) \times (d_2 + 2p_2) \times c$.

9. $\boldsymbol{Z}^{[l]} \in \mathbb{R}^{b \times r_1 \times r_2 \times n^{[l]}}$ – The feature map or intermediate output of layer l before applying the activation function, where r_1 is the height of the feature map after performing convolution at the given layer and r_2 is the width. The height and width of the feature map for that layer are

$$r_1 = \frac{d_1 - k_1 + 2p_1}{s_1} + 1 \quad \text{and} \quad r_2 = \frac{d_2 - k_2 + 2p_2}{s_2} + 1.$$

The feature map at position (x, y) for a single channel, n, is defined as

$$\begin{aligned}\boldsymbol{Z}^{[l]}[:, x, y, n] &= (\boldsymbol{H}^{[l-1]} * \boldsymbol{W}^{[l]})[:, x, y, n] \\ &= \sum_{i=0}^{k_1-1} \sum_{j=0}^{k_2-1} \sum_{k=0}^{n^{[l-1]}-1} \boldsymbol{W}^{[l]}[i, j, k, n] \boldsymbol{H}^{[l-1]}[:, s_1 x + i, s_2 y + j, k] + \boldsymbol{b}^{[l]}[n].\end{aligned}$$

Note that we assume $\boldsymbol{H}^{[l-1]}$ has already been padded with the appropriate number of zeros. At the first layer, $\boldsymbol{H}^{[-1]}$ is simply the input tensor, $\boldsymbol{X}$. An example of the convolution operation is shown in figure 13.7.

10. $\boldsymbol{H}^{[l]} \in \mathbb{R}^{b \times r_1 \times r_2 \times n^{[l]}}$ – The output of a layer l, which is defined such that

$$\boldsymbol{H}^{[l]} = \sigma^{[l]}(\boldsymbol{Z}^{[l]}).$$

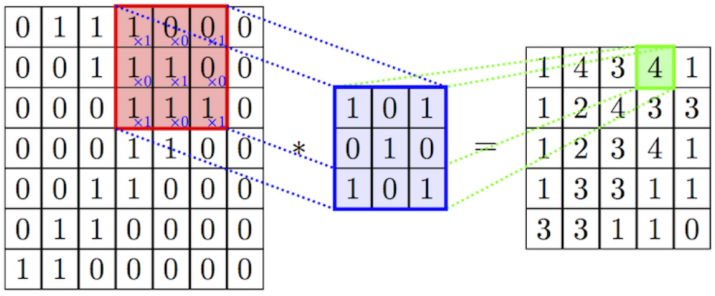

Figure 13.7: This is an example of a convolution operation for a single input tensor, $\boldsymbol{H}^{[l-1]}$, and tensor of filters, $\boldsymbol{W}^{[l]}$. The input tensor, $\boldsymbol{H}^{[l-1]}$, shown on the left, represents a 7×7 image with one channel, which has not been padded. The tensor of filters, $\boldsymbol{W}^{[l]}$, shown in the middle, represents a 3×3 kernel with one input and one output channel. We assume that we use a stride length of one. The feature map/intermediate output is the tensor, $\boldsymbol{Z}^{[l]}$, shown on the right, which has one channel and the spatial dimensions 5×5 because we are using a stride length of one. The kernel slides along the image and is multiplied by each patch of the input to produce the output.

13.3.2 Pooling Layer

Pooling layers are used in convolutional neural networks to downsample the input feature maps. Each pooling layer has a kernel of shape $k_1 \times k_2$, a stride length (s_1, s_2), and a padding size (p_1, p_2). A pooling layer takes in an input tensor, $\boldsymbol{X}$, of shape $b \times d_1 \times d_2 \times c$, where b is the batch size, d_1 is the height of the input, d_2 is the width of the input, and c is the number of channels in the input. The pooling layer then outputs an output tensor, $\boldsymbol{Y}$, of shape $b \times r_1 \times r_2 \times c$, where b is the batch size, r_1 is the height of the output, r_2 is the width of the output, and c is the number of channels in the output. Pooling layers to not change the number of channels, but they reduce the number of spatial dimensions (i.e. $r_1 < d_1$ and $r_2 < d_2$). The height and width of the output after pooling can be expressed in terms of the kernel parameters as

$$r_1 = \frac{d_1 - k_1 + 2p_1}{s_1} + 1 \quad \text{and} \quad r_2 = \frac{d_2 - k_2 + 2p_2}{s_2} + 1.$$

For each channel, we take either the maximum or average of all of the points in the window of size $k_1 \times k_2$, then we slide the window by s_1 pixels in the direction of the first dimension. When we reach the end of the input tensor in that direction, we slide the window by s_2 pixels in the direction of the second dimension. We repeat these steps until we have performed this operation over the entire padded input image. When we take the maximum value at each patch, we call this operation **max pooling**. Similarly, when we take the average value at each patch, we call this operation **average pooling**. If we use max pooling, then the output of the pooling layer can be expressed as

$$\boldsymbol{Y}[m, x, y, c] = \max\Big\{\boldsymbol{X}[m, a, b, c] : a \in [s_1 x, s_1 x + k_1],\ b \in [s_2 y, s_2 y + k_2]\Big\}.$$

If we use average pooling, then the output of the pooling layer is

$$\boldsymbol{Y}[m, x, y, c] = \frac{1}{k_1}\frac{1}{k_2}\sum_{a=s_1 x}^{s_1 x + k_1}\sum_{b=s_2 y}^{s_2 y + k_2}\boldsymbol{X}[m, a, b, c].$$

Note that for both max and average pooling operations, we assume the input, $\boldsymbol{X}$, has been padded with the appropriate number of zeros.

13.4 Neural Network Heuristics

While there is theory surrounding neural networks, neural network performance is often improved by following heuristics that appear during the practical implementation of networks. We will discuss a few of the currently existing heuristics.

13.4.1 Sigmoid Unit Saturation

Define $\boldsymbol{s}$ as the vector obtained by applying the sigmoid function componentwise to some input vector, $\boldsymbol{x}$. The derivative of $\boldsymbol{s}$ with respect to $\boldsymbol{x}$ is $\boldsymbol{s} \odot (1 - \boldsymbol{s})$.

One issue with the sigmoid activation function is that when the output of a unit using this activation function is close to zero or one (i.e. $\boldsymbol{s} \approx \mathbf{0}$ or $\mathbf{1}$) for most training points, then $\boldsymbol{s}' = \boldsymbol{s} \odot (1 - \boldsymbol{s}) \approx \mathbf{0}$. When the derivative of $\boldsymbol{s}$ is close to zero, gradient descent changes very slowly, and we say that the unit is stuck. This is an issue because it can slow down training significantly and may prevent us from finding an optimal solution. We call this issue the **sigmoid unit saturation** problem or the **vanishing gradient** problem. The graph of the sigmoid function in figure 13.8 helps demonstrate this challenge.

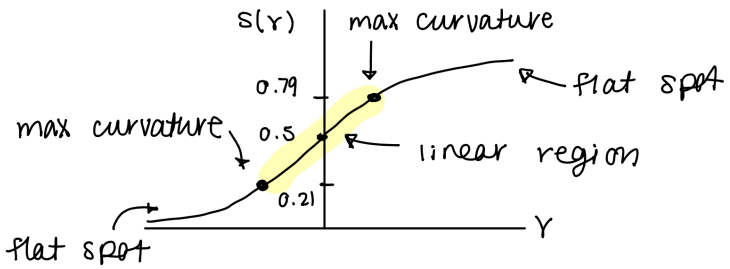

Figure 13.8: This is a graph of the sigmoid function. Notice that when s is close to 0 or 1, its slope is close to 0. We call these regions flat spots. When s is close to 0.5, the sigmoid function is approximately linear. We call this highlighted section the linear region. Ideally, we want to operate in the linear region.

There are several ways to help correct this issue:

1. For a unit with a **fan-in** of η (i.e. a unit with η incoming network connections), initialize each of the η incoming edges to have a random weight with mean 0 and standard deviation $1/\sqrt{\eta}$.
2. If the sigmoid function is used at the output layer, we could set the target values of the output to be 0.85 and 0.15, instead of 1 and 0.
3. Modify backpropagation to add a small constant (often around 0.1) to $\boldsymbol{s}'$.
4. Use the cross-entropy loss function instead of the squared error loss. Note that this only helps with stuck output units, not hidden units.
5. Replace the sigmoid activation function with the ReLU function. The ReLU activation function is less likely to contribute to vanishing gradients because it always has a constant non-zero gradient for positive inputs.

13.4.2 Heuristics for Faster Training

Neural networks can take a long time to train compared to other classifiers we have studied. Below are some heuristics to decrease training time:

1. Fix the sigmoid unit saturation (vanishing gradient) problem using the methods described in section 13.4.1.

2. When we have large, redundant data sets, use stochastic gradient descent, instead of batch gradient descent. One epoch of gradient descent presents every training sample once. Stochastic gradient descent often takes many epochs, but it makes significantly more progress per epoch compared to batch gradient descent. If the training sample is very large, stochastic gradient descent may not require many epochs to converge.

3. Normalize the data by centering each feature so its mean is zero and scaling each feature to have unit variance. Centering the data makes it easier for hidden units to get into a good operating region for the sigmoid and ReLU activation functions. Scaling the data makes the objective function better conditioned, so gradient descent converges faster.

4. "Center" the hidden units by replacing the sigmoid activation function with the hyperbolic tangent activation function. Recall that the output of the sigmoid function ranges from 0 to 1, while the output of the hyperbolic tangent function ranges from -1 to 1.

5. Use a different learning rate for each layer of weights. Earlier layers have smaller gradients, so they need a larger learning rate.

6. Use emphasizing schemes. Uncommon data samples are learned more slowly, so we may choose to emphasize them by presenting examples from rare classes more often or with a larger learning rate. We can also choose to emphasize misclassified samples. Note that emphasizing schemes may backfire if our data contains outliers.

7. Use second order approximation. Newton's method is generally impractical because the Hessian is too expensive to compute, but we can use other second order methods. The nonlinear conjugate gradient works well for smaller neural networks with small data in regression problems. Note that this should be used with batch descent only because it is too slow to use with redundant data. We can also use Stochastic Levenberg Marquardt, which approximates a diagonal Hessian.

8. Use acceleration schemes, such as RMSprop, Adam, and AMSgrad. These are relatively simple to implement and are quite popular.

13.4.3 Heuristics for Avoiding Bad Local Minima

The cost function used for neural networks generally has many local minima, and it is unlikely that we will reach the true global minimum. However, there are some heuristics to avoid "bad" local minima:

1. Fix the sigmoid unit saturation (vanishing gradient) problem using the methods described in section 13.4.1.

2. Use stochastic gradient descent, instead of batch gradient descent. The "random" motion seen in stochastic gradient descent can get you out of shallow local minima. Note that stochastic gradient descent does not always help with this issue, but in some cases it does.

3. Introduce momentum into gradient descent. Momentum can carry you through shallow local minima to deeper ones by preserving some amount of momentum between iterations. Note that this works with both batch and stochastic gradient descent. Gradient descent with momentum is implemented by first setting the momentum to be $\boldsymbol{\Delta w_0} := -\eta \nabla_w J(\boldsymbol{w})\big|_{\boldsymbol{w}=\boldsymbol{w_0}}$. The update rule for the weight vector is then given by

$$\boldsymbol{w_{k+1}} = \boldsymbol{w_k} + \boldsymbol{\Delta w_k}, \text{ where}$$

$$\boldsymbol{\Delta w_k} = -\eta \nabla_w J(\boldsymbol{w})\big|_{\boldsymbol{w}=\boldsymbol{w_k}} + \beta \boldsymbol{\Delta w_{k-1}}.$$

   The parameter $\beta < 1$ specifies how much momentum persists between iterations. Often, β is set to 0.9, but there is no one method to choose β.

13.4.4 Heuristics to Avoid Overfitting

There are various methods to reduce the variance of neural networks to avoid overfitting. Some of these heuristics are provided below:

1. Use ensembles of neural networks (discussed in section 17). We can use random initial weights and bagging (section 17.2) with neural networks, but this method is time intensive. Neural networks are already relatively slow to train, so this is not a popular method to reduce variance.

2. Introduce l_2 regularization, which is also referred to as **weight decay**. We can add the term $\lambda ||\boldsymbol{w}||_2^2$ to the cost function, where $\boldsymbol{w}$ is a vector containing all of the weights in the weight matrices for each layer. By adding this term to the cost function, the weight decays by a factor $1 - 2\epsilon\lambda$. By increasing the value of λ, we limit the size of the weights, which generally reduces overfitting at the risk of underfitting.

3. Introduce **dropout** to emulate ensembles of neural networks with only one network. During training, we can temporarily disable a random subset of the units and frequently change which subset is disabled. This gives the advantages of ensembles without the higher computation times.

4. Use fewer hidden units. The number of hidden units is a hyperparameter used to adjust the bias-variance trade-off. As we increase the number of hidden units, we increase the number of parameters, which increases the representational capacity of the network. If we have too many hidden units, we risk overfitting, but if we have too few, we risk underfitting.

5. Augment the training data with similar synthetic examples. For image processing, we can produce synthetic examples by modifying the original images by rotating, changing the contrast, editing the colors, etc. The more "volume" of the feature space that is covered by sample points, the more likely you are to get a good decision boundary that generalizes well to new points. Therefore, by generating similar synthetic data, the neural network can learn a decision function more robustly.

13.4.5 Heuristics to Avoid Underfitting

Similar to the previous section, there are various methods to reduce the bias of neural networks to avoid underfitting. Some of these heuristics are provided:

1. Add more units to hidden layers. As we increase the number of units in the hidden layers, we increase the number of parameters, which increases the representational capacity of the network. If we have too few hidden units, we risk underfitting, but if we have too many, we risk overfitting.

2. Add an additional hidden layer. By adding an additional hidden layer, we are increasing the number of parameters, making the neural network more expressive and allowing it to learn more decision functions.

3. Normalize the input data. As mentioned previously, centering the data makes it easier for hidden units to get into a good operating region for the sigmoid and ReLU activation functions. We also previously noted that scaling the data makes the objective function better conditioned. This makes it more likely to find a good local minimum.

13.4.6 Initializing Parameters

If two hidden units connected to the same inputs with the same activation function are initialized with the same parameters, then the deterministic learning algorithm will constantly update both of these units the same way. Therefore, it is important that hidden units connected to the same inputs with the same activation function are given different initial parameters. A popular scheme to initialize parameters is the **Xavier initialization** scheme, which sets each layer's weights to values chosen from a random uniform distribution.

Part III

Unsupervised Learning Techniques

Chapter 14

Principal Component Analysis (PCA)

14.1 Overview of PCA

Principal component analysis (PCA) is an unsupervised learning technique used for dimensionality reduction. Given sample points in the feature space, $\mathbb{R}^d$, we want to find $k < d$ directions that capture most of the variation in the data. Ideally, we would like to reduce high-dimensional data to a simpler, low-dimensional representation without losing too much information.

14.1.1 Purpose of PCA

There are several benefits of principal component analysis (PCA):

1. PCA reduces the number of dimensions used to represent data samples, making it faster and less expensive to perform some computations.

2. PCA removes irrelevant dimensions, which helps reduce variance in learning algorithms by limiting the effects of noisy or unreliable data. This helps to avoid overfitting when we perform classification or regression. PCA is similar to feature selection, except the "features" chosen by PCA are not axis-aligned. Rather, they are a linear combination of input features.

3. We are generally unable to visualize data in more than three dimensions. However, if we use PCA to transform our data into two or three dimensions, we can visualize a rough representation of the data.

Due to the first two benefits of principal component analysis (PCA), it is often used to preprocess data used in regression and classification problems.

14.1.2 Orthogonal Projections

Before discussing PCA in more detail, it is useful to review orthogonal projections. Suppose we have a design matrix, $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, which has been centered such that the mean of each sample point, $\boldsymbol{x_i} \in \mathbb{R}^d$, is zero. Let $\boldsymbol{w}$ be some vector in the feature space, $\mathbb{R}^d$, and $\hat{\boldsymbol{w}}$ be its corresponding unit vector of length one. The **vector projection** of a point $\boldsymbol{x_i}$ onto the vector $\boldsymbol{w}$ is

$$\vec{\boldsymbol{x}}_{\boldsymbol{i}} = (\boldsymbol{x_i}^T \hat{\boldsymbol{w}})\hat{\boldsymbol{w}} = \left(\boldsymbol{x_i}^T \frac{\boldsymbol{w}}{||\boldsymbol{w}||_2}\right) \frac{\boldsymbol{w}}{||\boldsymbol{w}||_2} = \frac{\boldsymbol{x_i}^T \boldsymbol{w}}{||\boldsymbol{w}||_2^2} \boldsymbol{w}.$$

Often, we are only interested in the **scalar projection** of $\boldsymbol{x_i}$ onto $\boldsymbol{w}$, which is

$$\tilde{x}_i = \boldsymbol{x_i}^T \hat{\boldsymbol{w}} = \boldsymbol{x_i}^T \frac{\boldsymbol{w}}{||\boldsymbol{w}||_2} = \frac{\boldsymbol{x_i}^T \boldsymbol{w}}{||\boldsymbol{w}||_2}.$$

Suppose we have k orthogonal directions $\boldsymbol{w_1}, \ldots, \boldsymbol{w_k}$, where $\boldsymbol{w_j} \in \mathbb{R}^d$. The vector projection of $\boldsymbol{x_i}$ onto the subspace spanned by these directions is

$$\vec{\boldsymbol{x}}_{\boldsymbol{i}} = \sum_{j=1}^{k} (\boldsymbol{x_i}^T \hat{\boldsymbol{w}}_{\boldsymbol{j}}) \hat{\boldsymbol{w}}_{\boldsymbol{j}} = \sum_{j=1}^{k} \frac{\boldsymbol{x_i}^T \boldsymbol{w_j}}{||\boldsymbol{w_j}||_2^2} \boldsymbol{w_j}.$$

Similarly, the scalar projection of sample point onto this subspace is

$$\tilde{x}_i = \sum_{j=1}^{k} \boldsymbol{x_i}^T \hat{\boldsymbol{w}}_{\boldsymbol{j}} = \sum_{j=1}^{k} \frac{\boldsymbol{x_i}^T \boldsymbol{w_j}}{||\boldsymbol{w_j}||_2}.$$

We refer to the k scalar values in the summation as the **principal coordinates**.

14.2 PCA Interpretations

In principal component analysis (PCA), we want to to find the best k directions to project our data onto. There are various interpretations of PCA that all allow us to find the same k directions. We will cover three PCA interpretations: fitting a Gaussian to choose directions of greatest variance, maximizing the sample variance of projected data, and minimizing the mean squared projection error. We could also interpret PCA as finding the best rank-k approximation of a matrix. However, we will not go over this interpretation in these notes.

In each interpretation, we assume we have a design matrix, $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, which has been centered such that the mean of the n sample points, $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$, is the zero vector. Notice that the matrix $\boldsymbol{X}^T \boldsymbol{X} \in \mathbb{R}^{d \times d}$ is a symmetric positive semidefinite matrix. Because it is a symmetric real matrix, it has d real eigenvalues, $\lambda_1, \ldots, \lambda_d$, and corresponding orthogonal unit vectors, $\boldsymbol{v_1}, \ldots, \boldsymbol{v_d}$, which we refer to as the **principal components**. Because $\boldsymbol{X}^T \boldsymbol{X}$ is positive semidefinite, all of its eigenvalues are non-negative. We will assume that $\lambda_1 \geq \ldots \geq \lambda_d \geq 0$.

14.2.1 Fitting a Gaussian

In the first interpretation of PCA, we fit a Gaussian distribution to the data using maximum likelihood estimation. Then we choose the k Gaussian axes with the greatest variance. Recall from section 7 that if we assume our data samples, $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$, come from an anisotropic Gaussian distribution with mean $\boldsymbol{\mu}$, then the maximum likelihood estimate of the covariance matrix is given by

$$\hat{\boldsymbol{\Sigma}} = \frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{x_i} - \boldsymbol{\mu})(\boldsymbol{x_i} - \boldsymbol{\mu})^T.$$

For the design matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ composed of sample points $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$, we can equivalently express the maximum likelihood estimate of the covariance as

$$\hat{\boldsymbol{\Sigma}} = \frac{1}{n}\big(\boldsymbol{X} - \boldsymbol{\mu}\mathbf{1}_d^T\big)^T\big(\boldsymbol{X} - \boldsymbol{\mu}\mathbf{1}_d^T\big).$$

Because we assume that $\boldsymbol{X}$ has been centered such that its mean, $\boldsymbol{\mu}$, is the zero vector, the maximum likelihood estimate of the covariance matrix becomes

$$\hat{\boldsymbol{\Sigma}} = \frac{1}{n}\boldsymbol{X}^T\boldsymbol{X}.$$

The eigenvectors corresponding to the k largest eigenvalues of the sample covariance matrix, $\hat{\boldsymbol{\Sigma}}$, represent the k directions with the greatest variance. The sample covariance matrix, $\hat{\boldsymbol{\Sigma}}$, has the same eigenvectors as $\boldsymbol{X}^T\boldsymbol{X}$, so to choose the best k-dimensional subspace, we pick the eigenvectors $\boldsymbol{v_1}, \ldots, \boldsymbol{v_k}$, corresponding to the k largest eigenvalues of $\boldsymbol{X}^T\boldsymbol{X}$. Because the eigenvectors are unit vectors, the **principal coordinates** of $\boldsymbol{x_i}$ are given by $\boldsymbol{x_i}^T\boldsymbol{v_j}$ for $j = 1, \ldots, k$.

14.2.2 Maximizing Variance

The next interpretation of PCA assumes that we want to find the directions in the feature space that maximize the sample variance of projected data. The idea is that we want to maximize the amount of variability (or information) preserved by the projected data samples. Because we are only interested in the direction, we want to find the unit vector $\boldsymbol{w}$ that solves the following optimization problem:

$$\underset{\boldsymbol{w}:||\boldsymbol{w}||_2=1}{\arg\max}\ \mathrm{Var}\big(\{\tilde{x}_1, \ldots, \tilde{x}_n\}\big).$$

To make this optimization easier to solve, we can express the variance as

$$\mathrm{Var}\big(\{\tilde{x}_1, \ldots, \tilde{x}_n\}\big) = \frac{1}{n}\sum_{i=1}^{n}\Big(\tilde{x}_i - \mathbb{E}[\tilde{x}_i]\Big)^2.$$

Because we assume that the data samples have zero mean, by the linearity of expectation, the projected points also have zero mean. Therefore,

$$\mathrm{Var}\big(\{\tilde{x}_1, \ldots, \tilde{x}_n\}\big) = \frac{1}{n}\sum_{i=1}^{n}\tilde{x}_i^2 = \frac{1}{n}\sum_{i=1}^{n}\Big(\frac{\boldsymbol{x_i}^T\boldsymbol{w}}{||\boldsymbol{w}||_2}\Big)^2.$$

We can equivalently express the variance of the projected points as

$$\begin{aligned}\text{Var}\big(\{\tilde{x}_1,\dots,\tilde{x}_n\}\big) &= \frac{1}{n}\frac{1}{||\boldsymbol{w}||_2^2}\sum_{i=1}^{n}\big(\boldsymbol{x_i}^T\boldsymbol{w}\big)^2 = \frac{1}{n}\frac{1}{||\boldsymbol{w}||_2^2}\sum_{i=1}^{n}\big(\boldsymbol{x_i}^T\boldsymbol{w}\big)^T\big(\boldsymbol{x_i}^T\boldsymbol{w}\big)\\ &= \frac{1}{n}\frac{1}{||\boldsymbol{w}||_2^2}\sum_{i=1}^{n}\boldsymbol{w}^T\boldsymbol{x_i}\boldsymbol{x_i}^T\boldsymbol{w} = \frac{1}{n}\frac{1}{||\boldsymbol{w}||_2^2}\boldsymbol{w}^T\left(\sum_{i=1}^{n}\boldsymbol{x_i}\boldsymbol{x_i}^T\right)\boldsymbol{w}\\ &= \frac{1}{n}\frac{1}{||\boldsymbol{w}||_2^2}\boldsymbol{w}^T\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} = \frac{1}{n}\frac{\boldsymbol{w}^T\boldsymbol{x}^T\boldsymbol{X}\boldsymbol{w}}{\boldsymbol{w}^T\boldsymbol{w}}.\end{aligned}$$

This leaves us with the following optimization problem:

$$\underset{\boldsymbol{w}:||\boldsymbol{w}||_2=1}{\arg\max}\ \frac{1}{n}\frac{\boldsymbol{w}^T\boldsymbol{x}^T\boldsymbol{X}\boldsymbol{w}}{\boldsymbol{w}^T\boldsymbol{w}}.$$

Because we are only interested in the optimal solution, we can remove the constant term and search for the direction that solves the following problem:

$$\underset{\boldsymbol{w}:||\boldsymbol{w}||_2=1}{\arg\max}\ \frac{\boldsymbol{w}^T\boldsymbol{x}^T\boldsymbol{X}w}{\boldsymbol{w}^T\boldsymbol{w}}.$$

The objective function in this optimization problem is the Rayleigh quotient of $\boldsymbol{X}^T\boldsymbol{X}$ and $\boldsymbol{w}$. From our knowledge of Rayleigh quotients, the optimal value of this optimization problem is the maximum eigenvalue, λ_1, and the optimal solution is the corresponding eigenvector, $\boldsymbol{v_1}$. To understand why this is true, please see my linear algebra notes. If we constrain $\boldsymbol{w}$ to be orthogonal to $\boldsymbol{v_1}$, then the optimal value is the second largest eigenvalue, λ_2, and the optimal solution is the corresponding eigenvector, $\boldsymbol{v_2}$. If we constrain $\boldsymbol{w}$ to be orthogonal to both $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$, then the optimal value is λ_3 and the optimal solution is $\boldsymbol{v_3}$. Therefore, if we want k orthogonal directions that maximize the variance of the projected points, then we should choose $\boldsymbol{v_1},\dots,\boldsymbol{v_k}$. Notice that this is the same result we found from the Gaussian interpretation of PCA.

14.2.3 Minimizing Projection Error

The next interpretation of PCA assumes that we want to find the directions that minimize the mean squared projection distance, which we refer to as the **projection error**. Because we are only interested in the direction, we want to find the unit vector $\boldsymbol{w}$ that solves the following optimization problem:

$$\underset{\boldsymbol{w}:||\boldsymbol{w}||_2=1}{\arg\min}\ \sum_{i=1}^{n}||\boldsymbol{x_i}-\vec{\boldsymbol{x}}_{\boldsymbol{i}}||_2^2.$$

Notice that we can equivalently express the mean squared projection error as

$$\sum_{i=1}^{n}||\boldsymbol{x_i}-\vec{\boldsymbol{x}}_{\boldsymbol{i}}||_2^2 = \sum_{i=1}^{n}\Big(||\boldsymbol{x_i}||_2^2 - 2\boldsymbol{x_i}^T\vec{\boldsymbol{x}}_{\boldsymbol{i}} + ||\vec{\boldsymbol{x}}_{\boldsymbol{i}}||_2^2\Big).$$

From the definitions of scalar and vector projections, we can equivalently write

$$\begin{aligned}\sum_{i=1}^{n} ||\boldsymbol{x_i} - \vec{\boldsymbol{x}}_{\boldsymbol{i}}||_2^2 &= \sum_{i=1}^{n} \left(||\boldsymbol{x_i}||_2^2 - 2\boldsymbol{x_i}^T \left(\frac{\boldsymbol{x_i}^T \boldsymbol{w}}{||\boldsymbol{w}||_2^2} \boldsymbol{w} \right) + ||\tilde{x}_i \hat{\boldsymbol{w}}||_2^2 \right) \\ &= \sum_{i=1}^{n} \left(||\boldsymbol{x_i}||_2^2 - 2 \left(\frac{\boldsymbol{x_i}^T \boldsymbol{w}}{||\boldsymbol{w}||_2} \right)^2 + \tilde{x}_i^2 \, ||\hat{\boldsymbol{w}}||_2^2 \right) \\ &= \sum_{i=1}^{n} \left(||\boldsymbol{x_i}||_2^2 - 2\tilde{x}_i^2 + \tilde{x}_i^2 \, ||\hat{\boldsymbol{w}}||_2^2 \right).\end{aligned}$$

Because $\hat{\boldsymbol{w}}$ is a unit vector with a norm of one, we can express this as

$$\begin{aligned}\sum_{i=1}^{n} ||\boldsymbol{x_i} - \vec{\boldsymbol{x}}_{\boldsymbol{i}}||_2^2 &= \sum_{i=1}^{n} \left(||\boldsymbol{x_i}||_2^2 - 2\tilde{x}_i^2 + \tilde{x}_i^2 \right) \\ &= \sum_{i=1}^{n} \left(||\boldsymbol{x_i}||_2^2 - \tilde{x}_i^2 \right) \\ &= \sum_{i=1}^{n} ||\boldsymbol{x_i}||_2^2 - \sum_{i=1}^{n} \tilde{x}_i^2.\end{aligned}$$

In the previous section, we expressed the sample variance of projected data as

$$\text{Var}\big(\{\tilde{x}_1, \ldots, \tilde{x}_n\}\big) = \frac{1}{n} \sum_{i=1}^{n} \tilde{x}_i^2.$$

Therefore, we can express the mean squared projection error as

$$\sum_{i=1}^{n} ||\boldsymbol{x_i} - \vec{\boldsymbol{x}}_{\boldsymbol{i}}||_2^2 = \sum_{i=1}^{n} ||\boldsymbol{x_i}||_2^2 - n\text{Var}\big(\{\tilde{x}_1, \ldots, \tilde{x}_n\}\big).$$

Our optimization problem can then be expressed as

$$\underset{\boldsymbol{w}:||\boldsymbol{w}||_2=1}{\arg\min} \left(\sum_{i=1}^{n} ||\boldsymbol{x_i}||_2^2 - n\text{Var}\big(\{\tilde{x}_1, \ldots, \tilde{x}_n\}\big) \right).$$

Because we are only interested in the optimal solution, we can remove the constant term and search for the direction that solves the following problem:

$$\underset{\boldsymbol{w}:||\boldsymbol{w}||_2=1}{\arg\min} \; -n\text{Var}\big(\{\tilde{x}_1, \ldots, \tilde{x}_n\}\big).$$

This optimization problem can equivalently be expressed as

$$\underset{\boldsymbol{w}:||\boldsymbol{w}||_2=1}{\arg\max} \; \text{Var}\big(\{\tilde{x}_1, \ldots, \tilde{x}_n\}\big).$$

Now we can see that minimizing the mean squared projection error is equivalent to maximizing the variance of the projected data. Therefore, if we want k orthogonal directions that minimize the mean squared projection error, then we should choose $\boldsymbol{v_1}, \ldots, \boldsymbol{v_k}$, as shown in the previous section.

14.3 More on PCA

14.3.1 Choosing Size of k

Previously, we assumed that the value of k is predetermined. However, if PCA is being used as a preprocessor for a supervised learning algorithm, k is a hyperparameter, which is generally chosen via validation. Depending on the value of k we choose, we will capture a different percentage of the variability in the training data. More specifically, the percentage of variability captured is

$$\% \text{ of variability} = \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{d} \lambda_i} * 100.$$

Note that the rank of $\boldsymbol{X}^T\boldsymbol{X}$ is equal to the number of non-zero eigenvalues. This means that if the rank of $\boldsymbol{X}^T\boldsymbol{X}$ is r, then $\lambda_1, \ldots, \lambda_r > 0$ and $\lambda_{r+1} = \ldots, \lambda_d = 0$. If we choose $k = r$ and project our data onto a k dimensional subspace using PCA, the percentage of variability captured will be

$$\frac{\sum_{i=1}^{r} \lambda_i}{\sum_{i=1}^{d} \lambda_i} * 100 = \frac{\sum_{i=1}^{r} \lambda_i}{\sum_{i=1}^{r} \lambda_i + \sum_{i=r+1}^{d} \lambda_i} * 100 = \frac{\sum_{i=1}^{r} \lambda_i}{\sum_{i=1}^{r} \lambda_i} * 100 = 100.$$

Therefore, if we project our data onto a r-dimensional subspace, where r is the rank of $\boldsymbol{X}^T\boldsymbol{X}$, we recreate a perfect representation of our data with no loss.

14.3.2 Singular Value Decomposition (SVD)

There are two main issues with principal component analysis (PCA):

1. Computing $\boldsymbol{X}^T\boldsymbol{X}$ takes $O(nd^2)$ time, which is an issue if d is large.
2. If $\boldsymbol{X}^T\boldsymbol{X}$ is poorly conditioned, its eigenvectors are numerically inaccurate.

One way to improve upon these issues is to use the singular value decomposition (SVD) of the design matrix, $\boldsymbol{X} \in \mathbb{R}^{n \times d}$. Recall that $\boldsymbol{X}^T\boldsymbol{X}$ has d eigenvalues, which are the squared singular values of $\boldsymbol{X}$. It has d corresponding eigenvectors, which are the right singular vectors of $\boldsymbol{X}$. This is useful because we can find the k greatest singular values of $\boldsymbol{X}$ and corresponding right singular vectors in $O(ndk)$ time. This is less time than it would take to compute $\boldsymbol{X}^T\boldsymbol{X}$ and perform spectral decomposition to find the eigenvalues and eigenvectors of $\boldsymbol{X}^T\boldsymbol{X}$.

Let r denote the rank of $\boldsymbol{X}$ and $\sigma_1, \ldots, \sigma_r$ denote the non-zero singular values of $\boldsymbol{X}$. The SVD of $\boldsymbol{X}$ is $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$, where $\boldsymbol{U} \in \mathbb{R}^{n \times n}$ is an orthogonal matrix whose columns are the left singular vectors, $\boldsymbol{V} \in \mathbb{R}^{d \times d}$ is an orthogonal matrix whose columns are the right singular vectors, and $\boldsymbol{\Sigma} \in \mathbb{R}^{n \times d}$ is given by

$$\boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{\Sigma_r} & \boldsymbol{0}^{r \times (d-r)} \\ \boldsymbol{0}^{(n-r) \times r} & \boldsymbol{0}^{(n-r) \times (d-r)} \end{bmatrix},$$

where $\boldsymbol{\Sigma_r} = \text{diag}(\sigma_1, \ldots, \sigma_r)$. Recall that the principal coordinates of a sample point $\boldsymbol{x_i}$ are given by $\boldsymbol{x_i}^T\boldsymbol{v_j}$, where $\boldsymbol{v_j}$ is the jth eigenvector of $\boldsymbol{X}^T\boldsymbol{X}$ or jth

right singular vector of $\boldsymbol{X}$. Because $\boldsymbol{V}$ is an orthogonal matrix, $\boldsymbol{V}^T\boldsymbol{V}$ is the identity matrix. Therefore, we can express the inner product $\boldsymbol{x_i}^T\boldsymbol{v_j}$ as

$$\boldsymbol{x_i}^T\boldsymbol{v_j} = (\boldsymbol{XV})_{ij} = (\boldsymbol{U\Sigma V}^T\boldsymbol{V})_{ij} = (\boldsymbol{U\Sigma})_{ij} = \sigma_j U_{ij}.$$

Now we can see that row i of the matrix $\boldsymbol{U\Sigma}$ gives us the principal coordinates of the sample point $\boldsymbol{x_i}$, so we do not need to explicitly compute the inner products.

14.3.3 PCA vs. LASSO

We presented PCA as a method to preprocess data by projecting data into a low-dimensional feature space. Recall from section 10.5.2 that LASSO can also be used as a method for feature subset selection. The main difference between PCA and LASSO for the purpose of feature selection is that while LASSO selects a subset of the original features, PCA produces features that are linear combinations of the original features. Also, while PCA allows you to specify how many features are chosen, LASSO does not allow you to select a number of features and may not even provide a strict subset of the original features.

Chapter 15

Clustering

15.1 Overview of Clustering

Clustering is an unsupervised learning technique used to partition data into groups of points with similar features such that points in a cluster are more similar than points across clusters. Clustering is used for the following purposes:

1. Discovery – Clustering can be used to find new points similar to those in a given cluster. For example, Netflix may use clustering to find movies similar to those you liked in the past.

2. Hierarchy – Clustering can be used to find hierarchical relationships among items. For example, biologists may use clustering to classify groups of biological organisms based on their genes.

3. Graph Partitioning – Clustering can be used to partition graphs, which may be useful for image segmentation or finding groups in social networks.

4. Quantization – Clustering can be used to compress a data set by reducing the number of choices.

15.2 k-Means Clustering

The goal of ***k*-means clustering**, which is also referred to as **Lloyd's algorithm**, is to partition a set of n d-dimensional data points, $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$, into k disjoint clusters, $S_1, \ldots, S_k$. For now, assume that we know the value of k. We want to group the samples points to minimize the variance within each of the clusters. The sample variance of the points in cluster i is

$$\frac{1}{d|S_i|} \sum_{\boldsymbol{x_j} \in S_i} ||\boldsymbol{x_j} - \boldsymbol{\mu_i}||_2^2,$$

where $\boldsymbol{\mu_i}$ is the mean of the ith cluster, S_i, which is given by

$$\boldsymbol{\mu_i} = \frac{1}{|S_i|} \sum_{\boldsymbol{x_j} \in S_i} \boldsymbol{x_j}.$$

The total within-cluster variation is the sum of the variances, which is given by

$$\sum_{i=1}^{k} \frac{1}{d|S_i|} \sum_{\boldsymbol{x_j} \in S_i} ||\boldsymbol{x_j} - \boldsymbol{\mu_i}||_2^2.$$

Because k-means clustering is an unsupervised learning technique, we assume that we do not know the true labels of the sample points. However, if $\boldsymbol{x_j}$ is assigned to cluster S_i, it is given the label $y_j = i$. To minimize the within-cluster variation, we want to find the labels that solve the following problem:

$$\underset{\boldsymbol{y} \in \mathbb{R}^n}{\arg\min} \sum_{i=1}^{k} \frac{1}{d|S_i|} \sum_{j:y_j=i} ||\boldsymbol{x_j} - \boldsymbol{\mu_i}||_2^2, \text{ where } \boldsymbol{\mu_i} = \frac{1}{|S_i|} \sum_{j:y_j=i} \boldsymbol{x_j}.$$

We can remove the constant term and equivalently express this problem as

$$\underset{\boldsymbol{y} \in \mathbb{R}^n}{\arg\min} \sum_{i=1}^{k} \frac{1}{|S_i|} \sum_{j:y_j=i} ||\boldsymbol{x_j} - \boldsymbol{\mu_i}||_2^2, \text{ where } \boldsymbol{\mu_i} = \frac{1}{|S_i|} \sum_{j:y_j=i} \boldsymbol{x_j}.$$

Note that an optimal clustering on $k+1 \leq n$ clusters has an objective value that is at least as small as that of the optimal clustering on k clusters. Furthermore, if the number of clusters is equal to the number of sample points (i.e. $k = n$), then every sample point can be placed in its own cluster. If this is the case, then the mean of cluster i would be $\boldsymbol{\mu_i} = \boldsymbol{x_i}$ for $i = 1, \ldots, n$, so the objective function would be zero, which is the minimum possible value.

15.2.1 k-Mean Heuristic

The optimization problem given previously is combinatorial, which is NP-hard and solvable in $O(nk^n)$ time by trying every partition. This is a very slow algorithm if n is large. Rather than trying every possible partition to obtain the best possible partition, we can use a heuristic to find a good partition that is likely not optimal. The following is the the ***k*-means heuristic**:

1. Initialize the algorithm using a method from section 15.2.2.
2. (a) Compute the mean, $\boldsymbol{\mu_i}$, of each cluster based on the assigned labels.
   (b) Assign each data point, $\boldsymbol{x_j}$, a label, $y_j = i$, corresponding to the nearest mean, $\boldsymbol{\mu_i}$. If there is a tie among the closest means and one option is to stay the current cluster, do not change the label of $\boldsymbol{x_j}$.
3. Repeat steps 2a and 2b until the means and labels stop changing.

Note that both steps two and three decrease the k-means objective function (unless they do not change anything, in which case the objective stays the same). Because the objective is monotonically decreasing and there are finitely many assignments, the algorithm must terminate eventually. While the algorithm is generally fast in practice, it could theoretically take an exponential number of iterations (i.e. $O(k^n)$ time). As another note, the algorithm always finds a local minimum but will generally not find a global minimum.

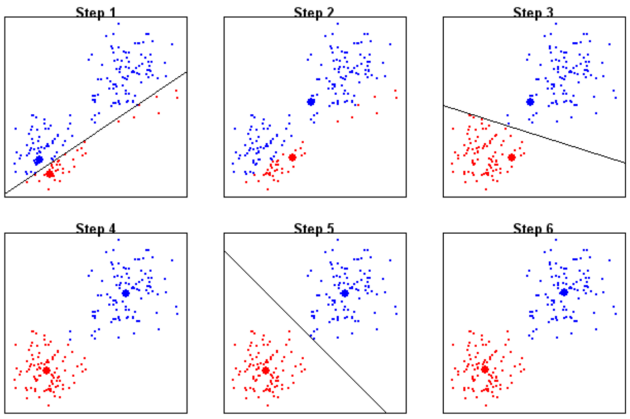

Figure 15.1: This is an example of 2-means clustering. In the first step, we randomly assign all of the data points to the red or blue class. In step two, we compute the means of the two clusters based on those labels. In step three, we label the data points as blue if they are closer to the blue mean and red if they closer to the red mean. In step four, we recompute the means. In step five, we relabel the points. Finally, in step six, we recompute the means one last time. At this point we have reached equilibrium, so the algorithm terminates.

15.2.2 Initializing the k-Means Algorithm

There are a few different methods for initializing the k-means algorithm:

1. Random partition – Randomly assign each data sample a label $i \in \{1, \ldots, k\}$, then start the k-means heuristic at step 2a.

2. Forgy method – Choose k random sample points to be the means of each of the k clusters, then start the k-means heuristic at step 2b.

3. k-means++ – As an improvement of the Forgy method, choose k random sample points to be the means of each of the k clusters with a preference for points that are far apart, then start the k-means heuristic at step 2b.

The k-means++ initialization method is a bit more involved than the other two methods, but it works well in theory and practice. For the best results, it is recommended that you run the k-means clustering algorithm multiple times with different initializations each time. Figure 15.1 is an example of the steps in the k-means clustering heuristic for $k = 2$ using random partitioning.

15.2.3 k-Medoids Clustering

The k-means clustering algorithm uses the Euclidean distance to measure the distance between sample points. In ***k*-medoids clustering**, we do not restrict ourselves to using the Euclidean distance. Instead, we specify a distance function, $d(\boldsymbol{x}, \boldsymbol{y})$, between points, $\boldsymbol{x}$ and $\boldsymbol{y}$. The distance between two points is also referred to as the **dissimilarity**. Some popular choices for the distance function are the l_1 norm and the l_∞ norm, but the best choice of distance metric very much depends on the application. While the mean is optimal for the Euclidean distance, it is not optimal for other distance metrics. In k-medoids clustering, we replace the mean with the **medoid**, which is the sample point in a given cluster that minimizes the total distance to other points in the same cluster. In some cases, k-medoids clustering is preferred to k-means clustering because k-medoids clustering is generally less sensitive to outliers.

15.3 Hierarchical Clustering

Another method to partition data is **hierarchical clustering**. In hierarchical clustering, we create a tree and treat every subtree as a cluster such that some clusters contain smaller clusters. There are two forms of hierarchical clustering:

1. **Agglomerative (bottom-up) clustering** – Start with each individual point as its own cluster and repeatedly fuse pairs of clusters with the greatest linkages. (Cluster linkage is discussed in section 15.3.1).
2. **Divisive (top-down) clustering** – Start with all of the points in one large cluster and repeatedly split clusters with low linkages.

15.3.1 Cluster Linkage

To perform either hierarchical clustering method, we need a distance function that measures the linkage between two clusters. Let A and B be two clusters of data samples. Some common linkage functions are the following:

1. **Complete linkage** – $\ell(A, B) = \max\Big\{d(\boldsymbol{x}, \boldsymbol{y}) : \boldsymbol{x} \in A,\ \boldsymbol{y} \in B\Big\}$
2. **Single linkage** – $\ell(A, B) = \min\Big\{d(\boldsymbol{x}, \boldsymbol{y}) : \boldsymbol{x} \in A,\ \boldsymbol{y} \in B\Big\}$
3. **Average linkage** – $\ell(A, B) = \frac{1}{|A||B|} \sum_{\boldsymbol{x} \in A} \sum_{\boldsymbol{y} \in B} d(\boldsymbol{x}, \boldsymbol{y})$
4. **Centroid linkage** – $\ell(A, B) = d(\boldsymbol{\mu_A}, \boldsymbol{\mu_B})$ for cluster means $\boldsymbol{\mu_A}$ and $\boldsymbol{\mu_B}$

For the first three linkage metrics, we can use any distance function, d. For the centroid linkage metric, the distance function should be the Euclidean distance. Each linkage function has different advantages and disadvantages. The single linkage measurement is the most sensitive to outliers and usually gives us the least balanced tree, so it not popular. The centroid linkage measurement can cause tree inversions, where a parent cluster is fused at a lower linkage distance than its children, so it is also unpopular. The complete linkage and average linkage measurements can both be effective in hierarchical clustering and both are popular, but complete linkage generally gives the most balanced tree.

15.3.2 Dendrogram

A **dendrogram** is an illustration of the cluster hierarchy (tree) in which the vertical axis encodes all of the linkage distances. We can cut the dendrogram into clusters by choosing a horizontal line according to the desired number of clusters or the desired inter-cluster linkage distance among clusters. Figure 15.2 provides an example of a dendrogram used to cluster data samples.

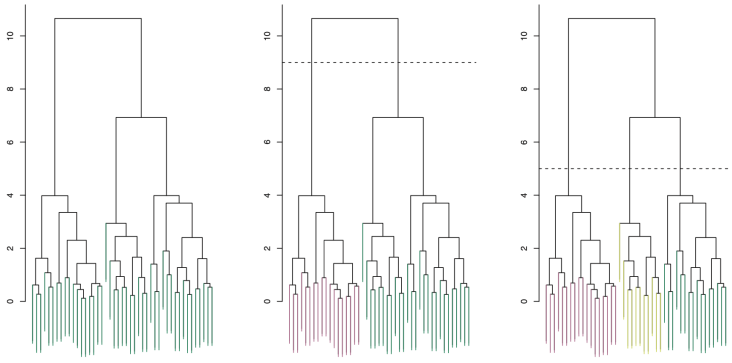

Figure 15.2: This is an example of a dendrogram, which is cut into one, two, or three clusters at various values of the inter-cluster linkage distance shown on the vertical axis.

15.4 Spectral Clustering

A final clustering method that differs from the k-means and hierarchical clustering methods is called **spectral clustering**. In spectral clustering, data samples are connected by a graph and partitions are chosen by cutting edges connecting points on that graph. To understand spectral clustering, it is valuable to have a basic understanding of graph theory. I will first give some background on graph theory, before giving greater detail on the spectral clustering method.

15.4.1 Graph Theory

A **graph**, $G = (V, E)$, is composed of a set of **nodes**/**vertices**, $V = \{1, \ldots, n\}$, and a set of **edges**, $E \subseteq V \times V$. If G is a **directed** graph, then the connections between nodes have an associated direction. For example, the edge (i, j) would indicate that there is a connection from node i to node j. If G is an **undirected** graph, then there is no direction associated with edges between nodes. For undirected graphs, E is a set of unordered pairs of nodes, meaning that

$$(i, j) \in E \;\Leftrightarrow\; (j, i) \in E \quad \forall i, j \in V.$$

If G is a **weighted** graph, then there is some weight associated with each edge of the graph. We will use w_{ij} to denote the weight of edge (i, j), connecting node i to node j. If the graph is undirected, the weights w_{ij} and w_{ji} are equal. If G is **unweighted**, there is no weight associated with each edge. For unweighted graphs, we can simply assume that w_{ij} is one for each pair (i, j) connected by an edge. In general, we can also say that w_{ij} is zero if there is no edge connecting node i to node j. We will also assume that there are no self-edges (i.e. $w_{ii} = 0$ for all $i \in V$). The **degree** of node i, which I will denote d_i, is the sum of the edge weights incident at that node, which we can express as

$$d_i = \sum_{j:(i,j)\in E} w_{ij}.$$

Note that for an unweighted graph, the degree of a node is simply the number of other nodes connected to that node by an edge in the graph. We will generally focus our discussion on weighted, undirected graphs.

Laplacian Matrix

The **Laplacian matrix**, $\boldsymbol{L}$, is the symmetric $n \times n$ matrix defined such that

$$L_{ij} = \begin{cases} d_i & \text{if } i = j \\ -w_{ij} & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}.$$

We can see that the Laplacian matrix is necessarily symmetric for an undirected graph. We can also show that the Laplacian matrix for an undirected graph is always positive semidefinite. Recall that a matrix, $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, is positive semidefinite if and only if $\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} \geq 0$ for all $\boldsymbol{x} \in \mathbb{R}^n$. Therefore, to show that the Laplacian matrix is positive semidefinite, consider the following product:

$$\boldsymbol{x}^T\boldsymbol{L}\boldsymbol{x} = \sum_{i=1}^{n}\sum_{j=1}^{n} L_{ij}x_i x_j = \sum_{i=1}^{n}\left(L_{ii}x_i^2 + \sum_{j\neq i} L_{ij}x_i x_j\right).$$

From the definition of the Laplacian matrix, we can see that

$$\begin{aligned}
\boldsymbol{x}^T\boldsymbol{L}\boldsymbol{x} &= \sum_{i=1}^{n}\left(d_i x_i^2 + \sum_{j:(i,j)\in E} -w_{ij}x_i x_j\right) \\
&= \sum_{i=1}^{n}\left(\sum_{j:(i,j)\in E} w_{ij}x_i^2 + \sum_{j:(i,j)\in E} -w_{ij}x_i x_j\right) \\
&= \sum_{i=1}^{n}\sum_{j:(i,j)\in E}\left(w_{ij}x_i^2 - w_{ij}x_i x_j\right) \\
&= \frac{1}{2}\sum_{(i,j)\in E}\left(w_{ij}x_i^2 + w_{ij}x_j^2 - 2w_{ij}x_i x_j\right) \\
&= \frac{1}{2}\sum_{(i,j)\in E} w_{ij}\left(x_i - x_j\right)^2
\end{aligned}$$

Assuming that the weights are non-negative, $\boldsymbol{x}^T\boldsymbol{L}\boldsymbol{x}$ is the sum of non-negative elements, so $\boldsymbol{x}^T\boldsymbol{L}\boldsymbol{x} \geq 0$ for all $\boldsymbol{x} \in \mathbb{R}^n$. Therefore, the Laplacian matrix for an undirected graph with no self-loops is positive semidefinite as claimed.

Eigenvalues of Laplacian

If G is a completely connected graph, then the Laplacian matrix, $\boldsymbol{L}$, has a single zero eigenvalue with corresponding eigenvector $\mathbf{1}_n$. To see this, notice that

$$(\boldsymbol{L}\mathbf{1}_n)_i = \sum_{j=1}^{n} L_{ij} = L_{ii} + \sum_{j\neq i} L_{ij} = \sum_{j:(i,j)\in E} w_{ij} + \sum_{j:(i,j)\in E} -w_{ij} = 0.$$

Now let's assume that the graph G has k connected components, $C_1, \ldots, C_k$. Assume that any two vertices in C_i can be connected by a set of edges and that there is no edge between any vertex in C_i and any vertex in C_j for $i \neq j$. The Laplacian matrix, $\boldsymbol{L}$, has exactly k zero eigenvalues and $n-k$ strictly positive eigenvalues, meaning that the dimension of its null space is k and its rank is $n-k$. The k orthogonal eigenvectors, $\boldsymbol{v^{(1)}}, \ldots, \boldsymbol{v^{(k)}}$, in the nullspace of $\boldsymbol{L}$ satisfy

$$v_j^{(i)} = \begin{cases} 1 & \text{if } j \in C_i \\ 0 & \text{otherwise} \end{cases}.$$

To see why these eigenvectors correspond to zero eigenvalues, notice that

$$\left(\boldsymbol{v^{(l)}}\right)^T\boldsymbol{L}\boldsymbol{v^{(l)}} = \frac{1}{2}\sum_{(i,j)\in E} w_{ij}\left(v_i^{(l)} - v_j^{(l)}\right)^2 = 0.$$

The equality follows because if there exists an edge $(i,j) \in E$, then i and j are in the same connected component and are assigned the same values by $\boldsymbol{v^{(l)}}$. Now suppose that we define the matrix $\boldsymbol{V}$ such that its ith column is the ith orthogonal eigenvector, $\boldsymbol{v^{(i)}}$, in the nullspace of $\boldsymbol{L}$. If the ith and jth row of $\boldsymbol{V}$ are equal, then nodes i and j belong to the same connected component.

15.4.2 Overview of Spectral Clustering

Now that we covered some background on graphs, I will discuss spectral clustering. The goal of spectral clustering is to cut a graph, G, into two smaller graphs, G_1 and G_2, of similar sizes while limiting the amount of edge weight connecting the vertices between the graphs. We generally want to minimize the **sparsity** of the cut, which is also referred to as the **cut ratio** and is defined as

$$\frac{\text{Cut}(G_1, G_2)}{\text{Mass}(G_1)\text{Mass}(G_2)},$$

where $\text{Cut}(G_1, G_2)$ is the total weight of the edges connecting vertices in G_1 to those in G_2 and $\text{Mass}(G_i)$ is the number of vertices in graph G_i. By minimizing this objective, we seek to minimize the numerator and maximize the denominator. Notice that the value of the denominator is at its maximum when $\text{Mass}(G_1) = \text{Mass}(G_2)$. Therefore, this objective penalizes unbalanced cuts. Additionally, the numerator is at its minimum when $\text{Cut}(G_1, G_2) = 0$. Therefore, this objective penalizes cuts that do not result in well-separated graphs. While we would ideally like to find the cut with the minimum sparsity, this is a combinatorial problem, which is NP-hard. Instead, we will discuss methods to find an approximate solution to the sparsest cut problem.

15.4.3 Algebraic Problem

To find an approximation for the sparsest cut problem, we will turn it into an algebraic problem. We assume that we are working with a weighted, undirected graph, $G = (V, E)$, that has no self-edges. We will also assume that the weight $w_{ij} = w_{ji}$ is zero if there is no edge connecting vertex i to vertex j.

Maximizing Graph Separation

We will first turn the goal of maximizing graph separation into an algebraic problem. Recall that to maximize graph separation, we aim to minimize $\text{Cut}(G_1, G_2)$, which is the sum of the weights of the cut edges. Let n equal the total number of vertices, $|V|$, and $\boldsymbol{y} \in \mathbb{R}^n$ be the indicator vector of labels defined such that

$$y_i = \begin{cases} 1 & \text{if vertex } i \in G_1 \\ -1 & \text{if vertex } i \in G_2 \end{cases}.$$

We want to map the vector $\boldsymbol{y}$ to the total weight of all edges cut. Notice that if the edge (i, j) is cut, then y_i and y_j have different signs. If the edge (i, j) is not cut, then y_i and y_j have the same sign. From this observation, we can write

$$w_{ij}\frac{(y_i - y_j)^2}{4} = \begin{cases} w_{ij} & \text{if edge } (i,j) \text{ is cut} \\ 0 & \text{if edge } (i,j) \text{ is not cut} \end{cases}.$$

Previously, we said that $\text{Cut}(G_1, G_2)$ is the sum of the weights of cut edges. From our observation above, we can express this as

$$\text{Cut}(G_1, G_2) = \sum_{(i,j)\in E} w_{ij} \frac{(y_i - y_j)^2}{4} = \frac{1}{4} \sum_{(i,j)\in E} w_{ij}(y_i - y_j)^2.$$

From our discussion of the Laplacian, we can recognize the expression above as

$$\text{Cut}(G_1, G_2) = \frac{1}{2} \boldsymbol{y}^T \boldsymbol{L} \boldsymbol{y}.$$

From this expression, we can see that to minimize $\text{Cut}(G_1, G_2)$, we can minimize the Laplacian quadratic form, $\boldsymbol{y}^T \boldsymbol{L} \boldsymbol{y}$. Therefore, to find the cut that results in the maximum graph separation, we aim to solve the following problem:

$$\begin{aligned} \min_{\boldsymbol{y}\in\mathbb{R}^n} \quad & \boldsymbol{y}^T \boldsymbol{L} \boldsymbol{y} \\ \text{s.t.} \quad & y_i \in \{1, -1\},\ i = 1, \ldots, n \end{aligned}$$

Bisecting Graph

Now we will incorporate the goal of obtaining a balanced cut that results in subgraphs of similar sizes into our optimization problem. Suppose we want to bisect the graph G such that there are exactly $n/2$ vertices in graph G_1 and $n/2$ vertices in graph G_2. Based on how we defined $\boldsymbol{y}$, in order to obtain a cut of this form, we require that $\mathbf{1}_n^T \boldsymbol{y} = 0$. Now to generate a bisection that results in the maximum graph separation, we aim to solve the following problem:

$$\begin{aligned} \min_{\boldsymbol{y}\in\mathbb{R}^n} \quad & \boldsymbol{y}^T \boldsymbol{L} \boldsymbol{y} \\ \text{s.t.} \quad & y_i \in \{1, -1\},\ i = 1, \ldots, n \\ & \mathbf{1}_n^T \boldsymbol{y} = 0 \end{aligned}$$

Optimization Problem Relaxation

While we have expressed the problem of minimizing sparsity as an algebraic problem, the binary constraint on the components of $\boldsymbol{y}$ still makes this problem NP-hard, which means it cannot be solved in polynomial time. We can make this problem easier to solve by relaxing the binary constraint to allow for fractional vertices. We need to replace the binary constraint with a continuous one that does not allow for y_i to be zero. We will choose to constrain $\boldsymbol{y}$ such that it lies on a hypersphere of radius $\sqrt{n}$. Now we can write this relaxed problem as

$$\begin{aligned} \min_{\boldsymbol{y}\in\mathbb{R}^n} \quad & \boldsymbol{y}^T \boldsymbol{L} \boldsymbol{y} \\ \text{s.t.} \quad & \boldsymbol{y}^T \boldsymbol{y} = n \\ & \mathbf{1}_n^T \boldsymbol{y} = 0 \end{aligned}$$

Because $\boldsymbol{y}^T\boldsymbol{y}$ is a constant equal to n, we can write the objective as

$$\frac{\boldsymbol{y}^T\boldsymbol{L}\boldsymbol{y}}{\boldsymbol{y}^T\boldsymbol{y}}.$$

To solve this problem, recall that the Laplacian matrix, $\boldsymbol{L}$, is symmetric and positive semidefinite. We have just expressed the objective function as the Rayleigh quotient of $\boldsymbol{L}$ and $\boldsymbol{y}$. In general, to minimize the Rayleigh quotient, we take $\boldsymbol{y}$ to be the eigenvector of $\boldsymbol{L}$ corresponding to the smallest eigenvalue of $\boldsymbol{L}$. Please see my linear algebra notes to understand why this is the case.

Recall from our discussion of the eigenvalues of the Laplacian matrix that $\mathbf{1}_n$ is an eigenvector of $\boldsymbol{L}$ with the corresponding eigenvalue 0. Because $\boldsymbol{L}$ is a positive semidefinite matrix, this is the smallest eigenvalue of $\boldsymbol{L}$. If we choose $\boldsymbol{y} = \mathbf{1}_n$, then graph G_1 will contain all of the vertices and G_2 will contain none, which we do not want. Therefore, instead of choosing the eigenvector corresponding to the smallest eigenvalue of $\boldsymbol{L}$, we want to choose the eigenvector corresponding to the second smallest eigenvalue, which we call the **Fielder eigenvector**.

Spectral Partitioning Algorithm

We have determined that the optimal choice of $\boldsymbol{y}$ under the relaxed constraint $\boldsymbol{y}^T\boldsymbol{y} = n$ is the Fielder vector of the Laplacian matrix, $\boldsymbol{L}$. However, we still need to restrict $\boldsymbol{y}$ such that its components are all 1 or -1. While the simplest solution may be to take the sign of the vector $\boldsymbol{y}$, it is generally better to use a **sweep cut**. This leads us to the spectral partitioning algorithm:

1. Compute the Fielder vector, $\boldsymbol{v_2}$, of the Laplacian matrix, $\boldsymbol{L}$.
2. Sort the components of $\boldsymbol{v_2}$ from low to high values.
3. Try all $n-1$ cuts between successive components and partition the graph into subgraphs using the cut that results in the minimum sparsity.

The spectral partitioning algorithm does not require that a graph G is cut into two graphs, G_1 and G_2, of the same size, but it generally results in a balanced cut, giving us a good approximation of the minimum sparsity cut. While the combinatorial problem is NP-hard, this algorithm can run in polynomial time.

15.4.4 Advantages of Spectral Clustering

Recall from section 15.2 that k-means clustering can only give us linear decision boundaries. Spectral clustering is not limited to linear decision boundaries. Figure 15.3 demonstrates a comparison of k-means and spectral clustering.

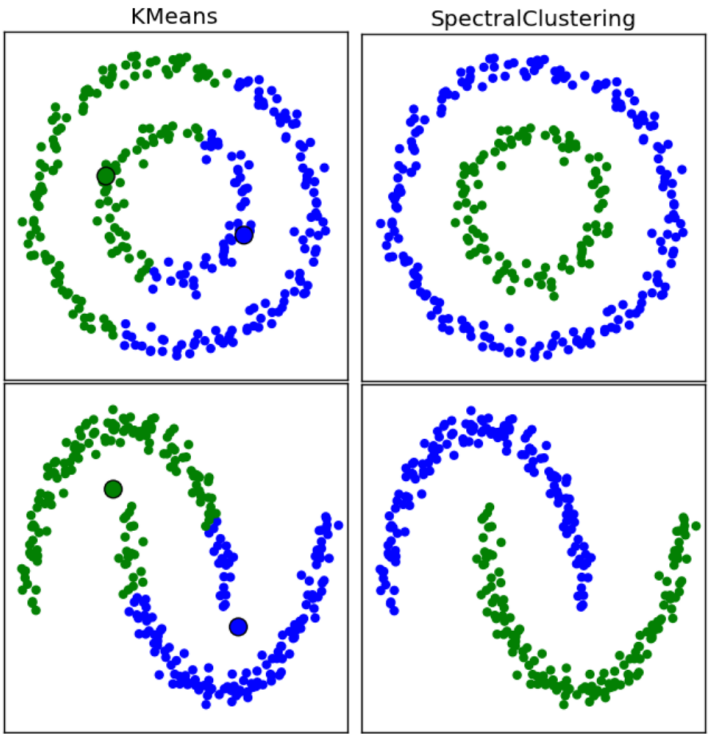

Figure 15.3: On the left are two sets of data partitioned by the k-means clustering method, and on the right are the same two data sets partitioned using the spectral clustering method. Notice that spectral clustering is able to learn nonlinear decision boundaries that cannot be obtained by k-means clustering.

15.4.5 Variations of Spectral Clustering

We discussed a basic version of the spectral clustering problem and an algorithm to solve the problem. We will also consider some variations of this problem.

Node Masses

In some cases, it may be of value to assign masses to nodes in order to give more prominence to some nodes over others. Rather than generating subgraphs with an equal number of nodes, we are now interested in generating subgraphs with the same total mass. Now instead of constraining $\boldsymbol{y}$ such that it lies on a hypersphere of radius $\sqrt{n}$, we will constrain $\boldsymbol{y}$ such that it lies on an ellipsoid.

Let $\boldsymbol{M}$ be a diagonal $n \times n$ matrix whose ith diagonal element is the mass of

node i. We can express this new optimization problem as

$$\begin{aligned} \min_{\boldsymbol{y}\in\mathbb{R}^n} \quad & \boldsymbol{y}^T\boldsymbol{L}\boldsymbol{y} \\ \text{s.t.} \quad & \boldsymbol{y}^T\boldsymbol{M}\boldsymbol{y} = \sum_{i=1}^{n} M_{ii} \\ & \mathbf{1}_n^T\boldsymbol{M}\boldsymbol{y} = 0 \end{aligned}$$

Now instead of choosing $\boldsymbol{y}$ to be the Fielder vector of $\boldsymbol{L}$, we will choose $\boldsymbol{y}$ to be the Fielder vector of the generalized eigensystem $\boldsymbol{L}\boldsymbol{v} = \lambda\boldsymbol{M}\boldsymbol{v}$, where λ is an eigenvalue and v is the corresponding eigenvector. Cheeger's inequality says that the sweep cut finds a cut with a sparsity less than or equal to

$$\sqrt{2\lambda_2 \max_{i\in\{1,\dots,n\}} \frac{L_{ii}}{M_{ii}}},$$

where λ_2 is the second smallest eigenvalue of the generalized eigensystem. The optimal cut has sparsity greater than or equal to $\lambda_2/2$.

We can use node masses to obtain a normalized cut, in which subgraphs have a similar number of edges. To generate a normalized cut, we set the mass of node i to the degree of that node (i.e. $M_{ii} = L_{ii} = d_i$). This is a popular choice for image segmentation to divide an image into different regions that possess similar properties such as intensity, texture, color etc.

Mutliple Subgraphs

Suppose we want to partition a graph, G, into more than two subgraphs. One way to do this is to use **greedy divisive clustering**. First, we cut the graph into two subgraphs using the spectral partitioning algorithm discussed previously. Then, we recursively cut the subgraphs into more subgraphs by repeatedly computing the Fielder vector and performing a sweep cut. We stop cutting the graphs once we have the desired number of clusters or until each subgraph is a single node. If we cut the subgraphs until each contains a single node, we can obtain a dendrogram, but a dendrogram formed this way may have inversions.

Another option to divide a graph, G, into k subgraphs is to use k eigenvectors, instead of just the single Fielder vector. Below is a method to form k clusters:

1. Compute the first k eigenvectors, $\boldsymbol{v_1}, \boldsymbol{v_2}, \dots, \boldsymbol{v_k}$, of the generalized eigensystem $\boldsymbol{L}\boldsymbol{v} = \lambda\boldsymbol{M}\boldsymbol{v}$. Let the ith column of $\boldsymbol{V} \in \mathbb{R}^{n\times k}$ be $\boldsymbol{v_i} \in \mathbb{R}^n$.

2. Scale the eigenvectors such that $\boldsymbol{v_i}^T\boldsymbol{M}\boldsymbol{v_i} = 1$ for all i, meaning

$$\tilde{\boldsymbol{v_i}} = \left(\sum_{i=1}^{n} M_{ii}\right)^{-1/2} \boldsymbol{v_i}.$$

3. Let $\boldsymbol{v_j^T} \in \mathbb{R}^k$ denote the jth row of $\boldsymbol{V}$, which we will refer to as the spectral vector for vertex j. Normalize each row to have a length of one, meaning

$$\tilde{\boldsymbol{v_j}}^{\boldsymbol{T}} = \frac{\boldsymbol{v_j^T}}{||\boldsymbol{v_j^T}||_2}.$$

4. Use k-means clustering to cluster the normalized spectral vectors. Because all of the spectral vectors were normalized to lie on a sphere, k-means clustering will cluster vectors that are separated by small angles.

Part IV

Improving Learning Techniques

Chapter 16

The Kernel Trick

16.1 Kernels

Sometimes it is beneficial to lift a feature set to a higher dimensional space to improve the performance of a classifier or regression model. Suppose we have a design matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ whose ith row is the d-dimensional sample point $\boldsymbol{x_i}^T$. Let $\phi : \mathbb{R}^d \to \mathbb{R}^D$ be a function, which we call a **lifting map**, that brings d-dimensional points into a D-dimensional space. Often, it is computationally intractable to compute $\phi(\boldsymbol{x_i})$ for all n data points. Instead, we use **kernels** to work with these features without actually computing them. For some lifting map, ϕ, the kernel function, $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, is defined such that

$$k(\boldsymbol{x_i}, \boldsymbol{x_j}) = \phi(\boldsymbol{x_i})^T \phi(\boldsymbol{x_j}).$$

All valid kernel functions can be expressed in this form. Note that valid kernel functions are closed under positive linear combinations. From this expression of a valid kernel, we can also see that a valid kernel must satisfy

$$k(\boldsymbol{x_i}, \boldsymbol{x_i}) = \phi(\boldsymbol{x_i})^T \phi(\boldsymbol{x_i}) = ||\phi(\boldsymbol{x_i})||_2^2 \geq 0.$$

The **kernel matrix**, $\boldsymbol{K} \in \mathbb{R}^{n \times n}$, is defined such that its ijth element is

$$K_{ij} = k(\boldsymbol{x_i}, \boldsymbol{x_j}).$$

Let $\phi(\boldsymbol{X})$ be an $n \times D$ lifted design matrix whose rows are the lifted feature vectors, $\phi(\boldsymbol{x_i})^T$. Now we can also express the kernel matrix as

$$\boldsymbol{K} = \phi(\boldsymbol{X})\phi(\boldsymbol{X})^T.$$

Because the kernel matrix is defined as a matrix product, it is always symmetric and positive semidefinite. Because $\phi(\boldsymbol{X})$ does not necessarily have full row rank, the kernel matrix is not necessarily positive definite and may not be invertible.

Any function that satisfies the constraints previously mentioned is a valid kernel function. The two most popular kernel functions by far are the polynomial kernel and the Gaussian kernel, so we will only focus on these two kernel functions.

16.1.1 Polynomial Kernel

For a data sample, $\boldsymbol{x_i}$, the polynomial feature vector, $\phi(\boldsymbol{x_i})$, contains every monomial in $\boldsymbol{x_i}$ up to some degree, p. For example, if we have $d = 2$ features and want to use a polynomial of degree $p = 2$, the polynomial feature vector is

$$\phi(\boldsymbol{x_i}) = \begin{bmatrix} x_{i1}^2 & x_{i2}^2 & x_{i1}x_{i2} & x_{i1} & x_{i2} & 1 \end{bmatrix}^T.$$

For a data sample with d features, a degree p polynomial feature vector contains $O(d^p)$ features, which is computationally intractable for typical values of d and p. Because we are generally unable to work with the polynomial feature vectors directly, we must work with the kernel function. If ϕ is a polynomial feature mapping of degree p, then the **polynomial kernel** can be expressed as

$$k(\boldsymbol{x_i}, \boldsymbol{x_j}) = (\boldsymbol{x_i}^T \boldsymbol{x_j} + 1)^p.$$

To better understand why this is true, let's again consider the case where we have $d = 2$ features and want to use a polynomial of degree $p = 2$. For this case,

$$\begin{aligned}
(\boldsymbol{x_i}^T \boldsymbol{x_j} + 1)^2 &= \left(\begin{bmatrix} x_{i1} & x_{i2} \end{bmatrix} \begin{bmatrix} x_{j1} \\ x_{j2} \end{bmatrix} + 1 \right)^2 \\
&= (x_{i1}x_{j1} + x_{i2}x_{j2} + 1)^2 \\
&= x_{i1}^2 x_{j1}^2 + x_{i2}^2 x_{j2}^2 + 2x_{i1}x_{j1}x_{i2}x_{j2} + 2x_{i1}x_{j1} + 2x_{i2}x_{j2} + 1 \\
&= \begin{bmatrix} x_{i1}^2 & x_{i2}^2 & \sqrt{2}\, x_{i1}x_{i2} & \sqrt{2}\, x_{i1} & \sqrt{2}\, x_{i2} & 1 \end{bmatrix} \cdot \\
&\qquad \begin{bmatrix} x_{j1}^2 & x_{j2}^2 & \sqrt{2}\, x_{j1}x_{j2} & \sqrt{2}\, x_{j1} & \sqrt{2}\, x_{j2} & 1 \end{bmatrix}
\end{aligned}$$

Notice that these two vectors are the same as the polynomial feature vectors, $\phi(\boldsymbol{x_i})^T$ and $\phi(\boldsymbol{x_j})^T$, except that there are constant terms included, which will not affect classification or regression problems. The polynomial kernel is beneficial because we can compute $k(\boldsymbol{x_i}, \boldsymbol{x_j})$ in $O(d)$ time, instead of $O(d^p)$ time.

16.1.2 Gaussian Kernel

The **Gaussian kernel**, or the **radial basis function (RBF) kernel**, is

$$k(\boldsymbol{x_i}, \boldsymbol{x_j}) = \exp\left(-\frac{||\boldsymbol{x_i} - \boldsymbol{x_j}||_2^2}{2\sigma^2} \right),$$

where σ is a hyperparameter that balances the bias-variance trade-off and should be chosen via validation. For larger values of σ, the Gaussian PDF is wider and the kernel function is smoother, resulting in more bias and less variance. For smaller values of σ, the Gaussian PDF is narrower and the kernel function is less smooth, resulting in less bias and more variance.

To see that the Gaussian kernel is a valid kernel function, we will consider the case when we only have $d = 1$ feature. For this case, notice that

$$\begin{aligned} k(x_i, x_j) &= \exp\left(-\frac{|x_i - x_j|^2}{2\sigma^2}\right) \\ &= \exp\left(-\frac{x_i^2}{2\sigma^2} - \frac{x_j^2}{2\sigma^2} + \frac{x_i x_j}{\sigma^2}\right) \\ &= \exp\left(-\frac{x_i^2}{2\sigma^2}\right)\exp\left(-\frac{x_j^2}{2\sigma^2}\right)\exp\left(\frac{x_i x_j}{\sigma^2}\right) \end{aligned}$$

Using the Taylor series expansion for the exponential function,

$$\begin{aligned} k(x_i, x_j) &= \exp\left(-\frac{x_i^2}{2\sigma^2}\right)\exp\left(-\frac{x_j^2}{2\sigma^2}\right)\left(1 + \frac{x_i x_j}{\sigma^2 \cdot 1!} + \frac{x_i^2 x_j^2}{\sigma^4 \cdot 2!} + \frac{x_i^3 x_j^3}{\sigma^6 \cdot 3!} + \dots\right) \\ &= \exp\left(-\frac{x_i^2}{2\sigma^2}\right)\left[1 \quad \tfrac{x_i}{\sigma\sqrt{1!}} \quad \tfrac{x_i^2}{\sigma^2\sqrt{2!}} \quad \tfrac{x_i^3}{\sigma^3\sqrt{3!}} \quad \dots\right] \cdot \\ &\qquad \exp\left(-\frac{x_j^2}{2\sigma^2}\right)\left[1 \quad \tfrac{x_j}{\sigma\sqrt{1!}} \quad \tfrac{x_j^2}{\sigma^2\sqrt{2!}} \quad \tfrac{x_j^3}{\sigma^3\sqrt{3!}} \quad \dots\right] \end{aligned}$$

Now we can see that, for the case where we have only one feature, the Gaussian lifting map lifts a sample point, x_i, to an infinite dimensional space such that

$$\phi(x_i) = \exp\left(-\frac{x_i^2}{2\sigma^2}\right)\left[1 \quad \tfrac{x_i}{\sigma\sqrt{1!}} \quad \tfrac{x_i^2}{\sigma^2\sqrt{2!}} \quad \tfrac{x_i^3}{\sigma^3\sqrt{3!}} \quad \dots\right]^T.$$

While the Gaussian kernel is very popular, the Gaussian lifting map is quite complicated and is rarely used directly. Like the polynomial kernel, the Gaussian kernel it takes $O(d)$ time to compute, which is computationally tractable.

The Gaussian kernel is very popular because it admits the following advantages:

1. The Gaussian kernel allows us to learn a decision function that is defined by a linear combination of Gaussians centered at the sample points.
2. Hypotheses computed using the Gaussian kernel are smooth. If the hypothesis is a linear combination of Gaussians, then it is C^∞ continuous.
3. The Gaussian kernel can be interpreted as a similarity measure for sample points that is maximum when two points are equal and approaches zero as the distance between the points increases.
4. Hypotheses using the Gaussian kernel behave similar to k-nearest neighbors but give smooth boundaries. Sample points "vote" for the label of a test point but points closer to the test point have greater weight.
5. Hypotheses computed with the Gaussian kernel oscillate less than those computed with the polynomial kernel, assuming the variance of the Gaussians is sufficiently large. Wide Gaussians with relatively large values of σ have small gradients, which prevents them from oscillating too much.

16.2 Kernelization

Kernelization, or the **kernel trick**, is a generalized version of featurization that greatly improves computational efficiency. The kernel trick can be used with any learning method that allows us to express learned weights as a linear combination of input samples. It cannot be used with methods like decision trees and k-nearest neighbors that do not find an optimal set of weights.

Suppose we have some learning algorithm that uses a design matrix, $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, the corresponding label vector, $\boldsymbol{y} \in \mathbb{R}^n$, and a weight vector, $\boldsymbol{w} \in \mathbb{R}^d$. For this learning algorithm, we define an optimization problem, which we refer to as the primal problem, to find the optimal primal weights, $\hat{\boldsymbol{w}}$. Once we have found the optimal weight vector, to then predict the label of a test point, $\boldsymbol{z}$, we compute

$$h(\boldsymbol{z}) = \hat{\boldsymbol{w}}^T \boldsymbol{z}.$$

To use the kernel trick for this learning method, we will first define a dual problem. Define the dual weight vector, $\boldsymbol{a} \in \mathbb{R}^n$, such that $\boldsymbol{w} = \boldsymbol{X}^T \boldsymbol{a}$. Begin by replacing all instances of $\boldsymbol{w}$ in the primal objective function with the term $\boldsymbol{X}^T \boldsymbol{a}$. Then rearrange the dual objective so that it is written only in terms of the dual weights, $\boldsymbol{a}$, the inner product between training points, $\boldsymbol{X}\boldsymbol{X}^T$ or $\boldsymbol{x_i}^T \boldsymbol{x_j}$, and the original labels, $\boldsymbol{y}$. The solution of the dual problem is the optimal dual weights, $\hat{\boldsymbol{a}}$. To then predict the label of a test point, $\boldsymbol{z}$, we compute

$$h(\boldsymbol{z}) = \hat{\boldsymbol{a}}^T \boldsymbol{X} \boldsymbol{z} = \sum_{i=1}^{n} \hat{a}_i \boldsymbol{x_i}^T \boldsymbol{z}.$$

Now suppose we want to use the lifted feature vectors. In the dual problem, we can replace instances of $\boldsymbol{X}\boldsymbol{X}^T$ with the kernel matrix, $\boldsymbol{K} = \phi(\boldsymbol{X})\phi(\boldsymbol{X})^T$, and instances of $\boldsymbol{x_i}^T \boldsymbol{x_j}$ with the kernel function, $k(\boldsymbol{x_i}, \boldsymbol{x_j}) = \phi(\boldsymbol{x_i})^T \phi(\boldsymbol{x_j})$. Now, to classify a test point, $\boldsymbol{z}$, we can compute the following weighted sum:

$$h(\boldsymbol{z}) = \sum_{i=1}^{n} \hat{a}_i \phi(\boldsymbol{x_i})^T \phi(\boldsymbol{z}) = \sum_{i=1}^{n} \hat{a}_i k(\boldsymbol{x_i}, \boldsymbol{z}).$$

Kernalization should become more clear as we apply it to specific learning methods. I will show how to use the kernel trick for ridge regression, perceptrons, logistic regression, and k-means clustering. After seeing how the kernel trick is applied to these problems, you should be able to apply it to other problems as well. The kernel trick is very commonly used with support vector machines (SVMs). However, the dual form of the SVM algorithm is complex and will not be covered in these notes. While I will not go over kernel SVMs, it is good to know that many off-the-shelf SVM packages enable the use of kernel functions.

16.3 Kernel Ridge Regression

Let $\boldsymbol{X}$ be the $n \times d$ matrix whose ith row is the transpose of the ith sample point, $\boldsymbol{x_i}$, augmented with one, and let $\boldsymbol{y}$ be the n-dimensional vector whose

ith element is the label corresponding to $\boldsymbol{x_i}$. Let $\boldsymbol{w} \in \mathbb{R}^d$ be the weight vector, whose last element is the bias term, and let $\boldsymbol{w'}$ be the weight vector without the bias term. Recall from section 10.2 that the ridge regression problem is

$$\min_{\boldsymbol{w} \in \mathbb{R}^d} \; ||\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}||_2^2 + \lambda ||\boldsymbol{w'}||_2^2,$$

where λ is the regularization parameter. In order to use the kernel trick, we have to penalize the bias term, resulting in the following optimization problem:

$$\min_{\boldsymbol{w} \in \mathbb{R}^d} \; ||\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}||_2^2 + \lambda ||\boldsymbol{w}||_2^2.$$

To make it less harmful to penalize the bias, it is recommended that we first center $\boldsymbol{X}$ and $\boldsymbol{y}$ so that their means are both zero. With this slight modification to the original ridge regression problem, the optimal solution, $\hat{\boldsymbol{w}}$, satisfies

$$(\boldsymbol{X}^T\boldsymbol{X} + \lambda \boldsymbol{I_d})\hat{\boldsymbol{w}} = \boldsymbol{X}^T\boldsymbol{y}.$$

After obtaining the optimal solution, we classify a test point, $\boldsymbol{z}$, such that

$$h(\boldsymbol{z}) = \hat{\boldsymbol{w}}^T\boldsymbol{z}.$$

16.3.1 Dual Form of Ridge Regression

Replacing all instances of $\boldsymbol{w}$ in the original ridge regression optimization problem with $\boldsymbol{X}^T\boldsymbol{a}$, we obtain the dual form of the ridge regression problem:

$$\min_{\boldsymbol{a} \in \mathbb{R}^n} ||\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{a} - \boldsymbol{y}||_2^2 + \lambda ||\boldsymbol{X}^T\boldsymbol{a}||_2^2.$$

Taking the gradient of this objective function with respect to the dual weights, we find that the optimal solution to the dual problem, $\hat{\boldsymbol{a}}$, satisfies

$$(\boldsymbol{X}\boldsymbol{X}^T + \lambda \boldsymbol{I_n})\hat{\boldsymbol{a}} = \boldsymbol{y}.$$

Note that if $\hat{\boldsymbol{a}}$ satisfies this equation, then we can write

$$\boldsymbol{X}^T\boldsymbol{y} = \boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{X}^T + \lambda \boldsymbol{I_n})\hat{\boldsymbol{a}} = (\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{X}^T + \lambda \boldsymbol{X}^T)\hat{\boldsymbol{a}} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda \boldsymbol{I_d})\boldsymbol{X}^T\hat{\boldsymbol{a}}.$$

Therefore, $\hat{\boldsymbol{w}} = \boldsymbol{X}^T\hat{\boldsymbol{a}}$ is a solution to the normal equation for ridge regression and is a linear combination of sample points. After obtaining the optimal solution, $\hat{\boldsymbol{a}}$, we can predict the label of a test point, $\boldsymbol{z}$, by computing

$$h(\boldsymbol{z}) = \hat{\boldsymbol{a}}^T\boldsymbol{X}\boldsymbol{z} = \sum_{i=1}^{n} \hat{a}_i \boldsymbol{x_i}^T\boldsymbol{z}.$$

16.3.2 Kernel Trick for Ridge Regression

If we instead want to use the feature vectors obtained by a lifting map, ϕ, then we would now need to find the optimal solution, $\hat{\boldsymbol{a}}$, that solves

$$\big(\phi(\boldsymbol{X})\phi(\boldsymbol{X})^T + \lambda \boldsymbol{I_n}\big)\hat{\boldsymbol{a}} = \boldsymbol{y}.$$

With this lifting map, our prediction would then become

$$h(\boldsymbol{z}) = \sum_{i=1}^{n} \hat{a}_i \phi(\boldsymbol{x_i})^T \phi(\boldsymbol{z}).$$

Recall that we generally do not want to evaluate the lifting map directly. We will instead use the kernel function $k(\boldsymbol{x_i}, \boldsymbol{z}) = \phi(\boldsymbol{x_i})^T\phi(\boldsymbol{z})$ and the kernel matrix $\boldsymbol{K} = \phi(\boldsymbol{X})\phi(\boldsymbol{X})^T$. Now we want to find the optimal solution, $\hat{\boldsymbol{a}}$, that solves

$$(\boldsymbol{K} + \lambda \boldsymbol{I_n})\hat{\boldsymbol{a}} = \boldsymbol{y}.$$

The prediction for a test point, $\boldsymbol{z}$, then becomes

$$h(\boldsymbol{z}) = \sum_{i=1}^{n} \hat{a}_i k(\boldsymbol{x_i}, \boldsymbol{z}).$$

Note that for training, we need to compute the kernel matrix, which takes $O(n^2 d)$ time, then solve a linear equation, which takes $O(n^3)$ time. Therefore, the total time for training is $O(n^3 + n^2 d)$. To predict the label of a test point, it takes $O(nd)$ time. If we had instead used the primal ridge regression problem for training, the total time for training would be $O(d^3 + d^2 n)$. Therefore, we prefer dual ridge regression when the number of features exceeds the number of sample points (i.e. $d > n$). If we are lifting features into a higher dimensional space (using a polynomial or Gaussian lifting map), then we generally have a lot of features and it is faster to use dual ridge regression. Note that the dual and primal ridge regression problems give us the same predictions. The difference between these two problems is the run time.

16.4 Kernel Perceptrons

Consider a data set of n d-dimensional sample points, $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$, with corresponding labels, $y_1, \ldots, y_n$. Recall from section 3.3 that the perceptron training algorithm with stochastic gradient descent is given by algorithm 6. After running this algorithm, a test point, $\boldsymbol{z}$, is then classified such that

$$h(\boldsymbol{z}) = \boldsymbol{w}^T \boldsymbol{z}.$$

Algorithm 6: Primal Form of Perceptron Algorithm

1 $\boldsymbol{w} \leftarrow$ arbitrary non-zero starting point
2 $\eta \leftarrow$ desired step size (learning rate)
3 **while** $y_i(\boldsymbol{w}^T\boldsymbol{x_i}) < 0$ *for some* i **do**
4 $\quad \boldsymbol{w} \leftarrow \boldsymbol{w} + \eta y_i \boldsymbol{x_i}$
5 **end**
6 **return** $\boldsymbol{w}$

16.4.1 Dual Form of Perceptron

To obtain the dual form of the perceptron algorithm, we will change both the update condition (line 3 in algorithm 6) and the update rule (line 4). Let $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ be the design matrix whose ith row is the ith sample point, $\boldsymbol{x_i} \in \mathbb{R}^d$, and define the vector $\boldsymbol{a} \in \mathbb{R}^n$ such that $\boldsymbol{w} = \boldsymbol{X}^T\boldsymbol{a}$. From this definition, we can write the expression on the left hand side of the update condition as

$$y_i \boldsymbol{w}^T \boldsymbol{x_i} = y_i(\boldsymbol{X}\boldsymbol{w})_i = y_i(\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{a})_i.$$

Now using the definition $\boldsymbol{a}$, we can express the update rule as

$$\boldsymbol{X}^T\boldsymbol{a} \leftarrow \boldsymbol{X}^T\boldsymbol{a} + \eta y_i \boldsymbol{x_i}.$$

Notice that we can equivalently express this update rule as

$$\sum_{j=1}^{n} a_j \boldsymbol{x_j} \leftarrow \sum_{j=1}^{n} a_j \boldsymbol{x_j} + \eta y_i \boldsymbol{x_i}.$$

Focusing on just one data sample at a time, we can perform the following update:

$$a_i \boldsymbol{x_i} \leftarrow a_i \boldsymbol{x_i} + \eta y_i \boldsymbol{x_i}.$$

This update rule can be expressed a bit more simply as

$$a_i \leftarrow a_i + \eta y_i.$$

Now we can write the dual form of the perceptron algorithm as in algorithm 7.

Algorithm 7: Dual Form of Perceptron Algorithm

1 $\boldsymbol{a} \leftarrow$ arbitrary non-zero starting point
2 $\eta \leftarrow$ desired step size (learning rate)
3 **while** $y_i(\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{a})_i < 0$ *for some* i **do**
4 $\quad a_i \leftarrow a_i + \eta y_i$
5 **end**
6 **return** $\boldsymbol{a}$

After running this algorithm, a test point, $\boldsymbol{z}$, is then classified such that

$$h(\boldsymbol{z}) = \boldsymbol{a}^T\boldsymbol{X}\boldsymbol{z} = \sum_{i=1}^{n} a_i \boldsymbol{x_i}^T \boldsymbol{z}.$$

16.4.2 Kernel Trick for Perceptrons

Now suppose that instead of using the original sample points, we want to use the feature vectors obtained by some lifting map, ϕ. Using the feature vectors would not change the update rule for the dual form of the perceptron algorithm, but we need to change the update condition. We would now check the condition

$$y_i(\phi(\boldsymbol{X})\phi(\boldsymbol{X})^T\boldsymbol{a})_i < 0.$$

Recall that it is generally more computationally efficient to work with the kernel matrix, K. Therefore, we should instead express this update condition as

$$y_i(\boldsymbol{K}\boldsymbol{a})_i < 0.$$

Now we can express the perecptron algorithm using kernels as in algorithm 8.

Algorithm 8: Kernel Perceptron Algorithm

```
a ← arbitrary non-zero starting point
η ← desired step size (learning rate)
while y_i(Ka)_i < 0  for some  i do
    a_i ← a_i + η y_i
end
return a
```

After running this algorithm, a test point, $\boldsymbol{z}$, is then classified such that

$$h(\boldsymbol{z}) = \sum_{i=1}^{n} a_i\phi(\boldsymbol{x_i})^T\phi(\boldsymbol{z}) = \sum_{i=1}^{n} a_i k(\boldsymbol{x_i}, \boldsymbol{z}).$$

Note that for training, we need to compute the kernel matrix, which takes $O(n^2d)$ time, then update each component of $\boldsymbol{a}$ and compute $\boldsymbol{K}\boldsymbol{a}$, which takes $O(n)$ time. To predict the label of a test point, it takes $O(nd)$ time. If we did not use the kernel trick, then we could have computed $\boldsymbol{w} = \phi(\boldsymbol{X})^T\boldsymbol{a}$ once in $O(nD)$ time and evaluated test points in $O(D)$ time. If the number of training points and test points exceed D/d, then using the kernel trick is faster.

16.5 Kernel Logisitic Regression

Let $\boldsymbol{X}$ be the $n \times d$ matrix whose ith row is the transpose of the ith sample point, $\boldsymbol{x_i}$, augmented with one, and let $\boldsymbol{y}$ be the n-dimensional vector whose ith element is the label corresponding to $\boldsymbol{x_i}$. Let $\boldsymbol{w} \in \mathbb{R}^d$ be the weight vector, whose last element is the bias term. Recall from section 9.5 that the batch gradient descent update rule for logistic regression can be expressed as

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta\boldsymbol{X}^T\big(\boldsymbol{y} - s(\boldsymbol{X}\boldsymbol{w})\big),$$

where the logisitc function, s, is applied component-wise. Similarly, the stochastic gradient descent update law for logistic regression is given by

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta\big(y_i - s(\boldsymbol{w}^T\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\big)\tilde{\boldsymbol{x}}_{\boldsymbol{i}},$$

where $\tilde{\boldsymbol{x}}_{\boldsymbol{i}}$ represents the ith data sample augmented with one. After the gradient descent algorithm terminates, a test point, $\boldsymbol{z}$, is then classified such that

$$h(\boldsymbol{z}) = s(\boldsymbol{w}^T\boldsymbol{z}).$$

16.5.1 Dual Form of Logistic Regression

As we did previously, define $\boldsymbol{a} \in \mathbb{R}^n$ such that $\boldsymbol{w} = \boldsymbol{X}^T\boldsymbol{a}$. With this definition, we can express the batch gradient descent update rule for logistic regression as

$$\boldsymbol{X}^T\boldsymbol{a} \leftarrow \boldsymbol{X}^T\boldsymbol{a} + \eta\boldsymbol{X}^T\big(\boldsymbol{y} - s(\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{a})\big).$$

We can equivalently express the dual form of the update rule as

$$\boldsymbol{a} \leftarrow \boldsymbol{a} + \eta\big(\boldsymbol{y} - s(\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{a})\big).$$

Similarly, we can express the stochastic gradient descent update rule as

$$a_i \leftarrow a_i + \eta(y_i - s(\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{a})_i).$$

Once these algorithms terminate, we can classify a test point, $\boldsymbol{z}$, such that

$$h(\boldsymbol{z}) = s\big(\boldsymbol{a}^T\boldsymbol{X}\boldsymbol{z}\big) = s\left(\sum_{i=1}^{n} a_1\boldsymbol{x}_{\boldsymbol{1}}^T\boldsymbol{z}\right).$$

16.5.2 Kernel Trick for Logisitc Regression

Now suppose that instead of using the original sample points, we want to use the feature vectors obtained by some lifting map, ϕ. The dual form of the batch gradient descent update rule for logistic regression now becomes

$$\boldsymbol{a} \leftarrow \boldsymbol{a} + \eta\big(\boldsymbol{y} - s(\phi(\boldsymbol{X})\phi(\boldsymbol{X})^T\boldsymbol{a})\big).$$

Using the kernel trick with the kernel matrix K, this update rule become

$$\boldsymbol{a} \leftarrow \boldsymbol{a} + \eta\big(\boldsymbol{y} - s(\boldsymbol{K}\boldsymbol{a})\big).$$

Similarly, the dual form of the stochastic gradient descent update rule is now

$$a_i \leftarrow a_i + \eta(y_i - s(\phi(\boldsymbol{X})\phi(\boldsymbol{X})^T\boldsymbol{a})_i).$$

Using the kernel trick with the kernel matrix K, this update rule become

$$a_i \leftarrow a_i + \eta(y_i - s(\boldsymbol{K}\boldsymbol{a})_i).$$

Using the lifted feature vectors, we would classify the test point z such that

$$h(\boldsymbol{z}) = s\left(\sum_{i=1}^{n} a_i\phi(\boldsymbol{x}_{\boldsymbol{i}})^T\phi(\boldsymbol{z})\right) = s\left(\sum_{i=1}^{n} a_i k(\boldsymbol{x}_{\boldsymbol{i}}, \boldsymbol{z})\right)$$

16.6 Kernel k-Means Clustering

Recall from section 15.2 that the goal of k-means clustering is to partition a set of n d-dimensional data points, $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$, into k disjoint clusters, $S_1, \ldots, S_k$. In the k-means clustering heuristic, we alternate between computing the mean of each cluster and assigning data samples to a cluster. If $\boldsymbol{x_j}$ is assigned to cluster S_i, it is given the label $y_j = i$. The mean of cluster i is then given by

$$\boldsymbol{\mu_i} = \frac{1}{|S_i|} \sum_{l:y_l=i} \boldsymbol{x_l}.$$

We assign a data sample, $\boldsymbol{x_j}$, to the cluster, S_i, whose mean is closest to the data sample. We can express this step as the following optimization problem:

$$\hat{y}_j = \underset{i \in \{1,\ldots,k\}}{\arg\min} \; ||\boldsymbol{x_j} - \boldsymbol{\mu_i}||_2^2.$$

16.6.1 Dual Form of k-Means Clustering

To determine the dual form of the the k-means clustering optimization problem, we will first expand the objective function in the original problem. Notice that

$$||\boldsymbol{x_j} - \boldsymbol{\mu_i}||_2^2 = \left(\boldsymbol{x_j} - \boldsymbol{\mu_i}\right)^T \left(\boldsymbol{x_j} - \boldsymbol{\mu_i}\right) = \boldsymbol{x_j}^T \boldsymbol{x_j} - 2\boldsymbol{\mu_i}^T \boldsymbol{x_j} + \boldsymbol{\mu_i}^T \boldsymbol{\mu_i}.$$

Plugging in our expression for the mean of the ith cluster, we have

$$\begin{aligned} ||\boldsymbol{x_j} - \boldsymbol{\mu_i}||_2^2 &= \boldsymbol{x_j}^T \boldsymbol{x_j} - 2\left(\frac{1}{|S_i|} \sum_{l:y_l=i} \boldsymbol{x_l}\right)^T \boldsymbol{x_j} + \left(\frac{1}{|S_i|} \sum_{l:y_l=i} \boldsymbol{x_l}\right)^T \left(\frac{1}{|S_i|} \sum_{l:y_l=i} \boldsymbol{x_l}\right) \\ &= \boldsymbol{x_j}^T \boldsymbol{x_j} - \frac{2}{|S_i|} \sum_{l:y_l=i} \boldsymbol{x_l}^T \boldsymbol{x_j} + \frac{1}{|S_i|^2} \sum_{l:y_l=i} \sum_{m:y_m=i} \boldsymbol{x_l}^T \boldsymbol{x_m}. \end{aligned}$$

Now we are left with the following optimization problem:

$$\hat{y}_j = \underset{i \in \{1,\ldots,k\}}{\arg\min} \left(\boldsymbol{x_j}^T \boldsymbol{x_j} - \frac{2}{|S_i|} \sum_{l:y_l=i} \boldsymbol{x_l}^T \boldsymbol{x_j} + \frac{1}{|S_i|^2} \sum_{l:y_l=i} \sum_{m:y_m=i} \boldsymbol{x_l}^T \boldsymbol{x_m} \right).$$

16.6.2 Kernel Trick for k-Means Clustering

Now suppose that instead of using the original sample points, we want to use the feature vectors obtained by some lifting map, ϕ. Now the dual form of the optimization problem defined for k-means clustering is given by

$$\hat{y}_j = \underset{i \in \{1,\ldots,k\}}{\arg\min} \left(\phi(\boldsymbol{x_j})^T \phi(\boldsymbol{x_j}) - \frac{2}{|S_i|} \sum_{l:y_l=i} \phi(\boldsymbol{x_l})^T \phi(\boldsymbol{x_j}) + \frac{1}{|S_i|^2} \sum_{l:y_l=i} \sum_{m:y_m=i} \phi(\boldsymbol{x_l})^T \phi(\boldsymbol{x_m}) \right).$$

Using the kernel function, k, we can equivalently express this problem as

$$\hat{y}_j = \underset{i \in \{1,\dots,k\}}{\arg\min} \left(k(\boldsymbol{x_j}, \boldsymbol{x_j}) - \frac{2}{|S_i|} \sum_{l:y_l=i} k(\boldsymbol{x_l}, \boldsymbol{x_j}) + \frac{1}{|S_i|^2} \sum_{l:y_l=i} \sum_{m:y_m=i} k(\boldsymbol{x_l}, \boldsymbol{x_m}) \right).$$

We can remove the first term from the objective function because it does not depend on the optimization variable, i, leaving us with the equivalent problem:

$$\hat{y}_j = \underset{i \in \{1,\dots,k\}}{\arg\min} \left(-\frac{2}{|S_i|} \sum_{l:y_l=i} k(\boldsymbol{x_l}, \boldsymbol{x_j}) + \frac{1}{|S_i|^2} \sum_{l:y_l=i} \sum_{m:y_m=i} k(\boldsymbol{x_l}, \boldsymbol{x_m}) \right).$$

Recall that we must assign all n data samples to a cluster based on the nearest mean. Therefore, we compute a solution to this optimization problem for each $j \in \{1, \dots, n\}$. Notice that the last term in the objective does not depend on j but is computed for every $j \in \{1, \dots, n\}$. Before entering the loop where we assign labels to data points, we should compute and store the following:

$$\alpha_i := \frac{1}{|S_i|^2} \sum_{l:y_l=i} \sum_{m:y_m=i} k(\boldsymbol{x_l}, \boldsymbol{x_m}), \quad i = 1, \dots, k.$$

When assigning data points to clusters, we can now solve the problem

$$\hat{y}_j = \underset{i \in \{1,\dots,k\}}{\arg\min} \left(-\frac{2}{|S_i|} \sum_{l:y_l=i} k(\boldsymbol{x_l}, \boldsymbol{x_j}) + \alpha_i \right).$$

Chapter 17

Ensembling & Adapative Boosting

17.1 Ensemble Learning

One way to improve the performance of learning algorithms is by **ensemble learning**, which we also refer to as **averaging**. Ensemble learning works by combining the predictions from various machine learning models. For regression problems, we form a prediction by taking the median or mean of the outputs for various models. For classification problems, we assign a label by taking the majority vote of the outputs or by looking at the average posterior probabilities.

We can take the average of the outputs of models obtained from different types of learning algorithms, or we can use the same learning algorithm on different training sets. Often, we do not have enough data to use many training sets. If this is the case, we can instead use random subsamples of a single training set. This is referred to as **bagging** and is more common than the first two options.

17.1.1 Bias & Variance with Ensembling

Ensemble learning, or averaging, can be used for many different learning algorithms, but it works particularly well with decision trees. The reason ensemble learning typically works well with decision trees is because decision trees can be designed to have high variance and low bias. Averaging reduces the variance, but it does not reduce the bias in general. To see why this is true, consider the set $\{\boldsymbol{Z_i}\}_{i=1}^n$ of correlated random variables with mean $\boldsymbol{\mu}$, variance σ^2, and correlation coefficient ρ for all $i \neq j$. The average of this set is given by

$$\bar{\boldsymbol{Z}} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{Z_i}.$$

Consider an arbitrary test point, $\boldsymbol{z}$, in the feature space whose true label is

given by $\gamma = g(\boldsymbol{z}) + \epsilon$, where g is a deterministic function and ϵ is zero-mean random noise. Recall from section 9.2.3 that the bias is the expected difference between the hypothesis, $h(\boldsymbol{z})$, and the true label, γ. Suppose that $\bar{\boldsymbol{Z}}$ is used as the prediction for the test point. The bias for this type of classifier is then

$$\text{bias} = \mathbb{E}[h(\boldsymbol{z})] - g(\boldsymbol{z}) = \mathbb{E}[\bar{\boldsymbol{Z}}] - g(\boldsymbol{z}) = \mu - g(\boldsymbol{z}).$$

The variance for a classifier that predicts the label of $\boldsymbol{z}$ to be $\bar{\boldsymbol{Z}}$ is

$$\text{variance} = \text{Var}(h(\boldsymbol{z})) = \text{Var}(\bar{\boldsymbol{Z}}) = \frac{1 + (n-1)\rho}{n}\sigma^2.$$

To better understand where the bias and variance equations come from, please review my notes on probability and random processes, particularly the section on the sample mean. From these equations, we can see that averaging (i.e. increasing the value of n) reduces the variance but does not reduce the bias. Because averaging reduces variance but not bias, when doing ensemble learning, we should use learners with low bias (e.g. deep decision trees) that may have high variance. Note that the hyperparamters used for ensemble learning are usually different than those used for a single learner because we should choose the hyperparameters such that the learners have higher variance and lower bias.

17.2 Bagging

Boostrap aggregating, which we refer to as **bagging**, is one form of ensemble learning. Given a training sample with n sample points, we generate T random subsamples of size n' by sampling with replacement. Since we are sampling with replacement, some points are chosen multiple times and some are not chosen at all. Duplicate points are assigned proportionally greater weight in cost function calculations. After choosing T random subsamples, we build a learner using each subsample. The metalearner takes a test point, feeds it into all T learners, then returns the average output. Recall that for regression problems, the average is either the median or mean of the output for each learner. For classification problems, the average is the majority vote or the average posterior probabilities.

17.3 Random Forests

As mentioned previously, ensemble learning is very popular for decision trees. When using ensembling for decision trees, sometimes using random subsamples of the training data does not introduce enough randomness to reduce variance significantly. It is common that we have a few really strong predictive features that tend to dominate in all random subsamples of the training data. When this is the case, almost all of the decision trees may have very similar early splits. If this happens, then all of the trees may end up looking very similar and the the learners' outputs may be too correlated, so averaging will not significantly reduce variance. To address this issue, we can use **random forests**.

17.3.1 Feature Subset Selection

The difference between random forests and bagging with decision trees is that random forests do not allow all of the decision trees to split at the same few dominant features. At each tree node, we take a random sample of m features from the set of d features. Then, we choose the best split from among those m features. Note that we choose a different random sample of features for each tree node. In practice, $m \approx \sqrt{d}$ works well for classification problems and $m \approx d/3$ works well for regression problems. However, m is a hyperparameter, which should ultimately be chosen via validation. As another note, smaller values of m generally lead to more randomness, less tree correlation, and higher bias.

17.3.2 Number of Decision Trees

The number of decision trees, T, is another hyperparameter. While increasing the number of decision trees typically improves accuracy, it also increases computation time and results in less model interpretability. Depending on the size and nature of the training set, we may use a few dozen to several thousand decision trees. We could start by determining the validation accuracy of the base model without using bagging. Then we should try generating a model using bagging with only a few decision trees. If the validation accuracy of the model is sufficiently higher than the validation accuracy of the base model, then we may only need to use a few decision trees for bagging. If there is not a sufficient improvement in validation accuracy, then we should try to increase the number of decision trees until the validation accuracy is sufficiently above that of the base model. If the computation time becomes too high while increasing the number of decision trees, we may need to find ways to speed up training or use fewer decision trees. To choose the hyperparameter T, we must find a balance between validation accuracy and computation time.

17.3.3 Algorithms & Running Times

Consider an $n \times d$ design matrix that contains n sample points with d features each. Suppose we want to construct a random forest with bagging and random subset selection. At each tree node, we take a random sample of m features from the set of d features. Consider choosing a split at a tree node that contains n' sample points. We can choose the best split for these n' sample points in $O(n'm)$ time. Therefore, the running time per sample point in that node is $O(m)$. Each of the n sample points participates in at most $O(h)$ nodes, where h is the depth of the decision tree. Therefore, the total running time for one tree is no greater than $O(nmh)$. We are working with T decision trees, so the total running time to train a random forest is $O(Tnmh)$. To classify a test point in a single tree, we move down the tree until we reach a leaf node, then we return the label of that leaf node. The worst case time is $O(h)$, where h is the depth of the decision tree. We are working with T decision trees, so the total query time is $O(Th)$.

17.4 AdaBoost

Adaptive boosting, which is commonly referred to as **AdaBoost**, is a very successful ensemble method used primarily for classification problems. The AdaBoost method takes in an $n \times d$ design matrix, $\boldsymbol{X}$, and an n-dimensional vector of labels y, where we assume $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$. The learning method trains T classifiers, $G_1, \ldots, G_T$ in sequence on weighted sample points.

Each classifier uses a different set of weights. The weight used for sample point $\boldsymbol{x_i}$ by the classifier G_t depends of the number of classifiers among $G_1, \ldots, G_{t-1}$ that misclassified it. The weight of a sample point $\boldsymbol{x_i}$ is increased before being used by classifier G_t if the previous classifier, G_{t-1}, misclassified it, and the weight is decreased if the previous classifier labeled it correctly. By assigning a greater weight to points that were previously misclassified, classifier G_t is expected to try harder to correctly classify those points.

17.4.1 Metalearner

In addition to the T classifiers, there is a metalearner that outputs a linear combination of the outputs of the T classifiers. For a test point, $\boldsymbol{z}$, I will denote the output of classifier G_t as $G_t(\boldsymbol{z})$. Based on the training accuracy of classifier G_t, it is assigned a weight β_t. More accurate classifiers are assigned higher weights so that they have a greater vote in the final prediction. The output of the metalearner, M, for the test point, $\boldsymbol{z}$, is then given by

$$M(\boldsymbol{z}) = \sum_{t=1}^{T} \beta_t G_t(\boldsymbol{z}).$$

We will assume that the output of each of the classifiers satisfies $G_t(\boldsymbol{z}) \in \{-1, 1\}$ for $t = 1, \ldots, T$. However, the output of the metalearner, $M(\boldsymbol{z})$, is continuous. To classify the test point, $\boldsymbol{z}$, we generally return the sign of $M(\boldsymbol{z})$.

17.4.2 AdaBoost Optimization Problem

Suppose we have already trained the classifiers $G_1, \ldots, G_{T-1}$ and have chosen the associated weights $\beta_1, \ldots, \beta_{T-1}$. Now we want to train the classifier G_T and choose the associated weight β_T that minimizes the total risk of the metalearner. For some loss function, $L(\rho, \ell)$, where ρ is a predicted label and ℓ is a true label, we define the total risk associated with the metalearner as the mean loss:

$$\text{Risk} = \frac{1}{n} \sum_{i=1}^{n} L\big(M(\boldsymbol{x_i}), y_i\big).$$

Therefore, we choose G_T and β_T that solve the following optimization problem:

$$\begin{aligned} \min_{G_T, \beta_T} \quad & \sum_{i=1}^{n} L\big(M(\boldsymbol{x_i}), y_i\big) \\ \text{s.t.} \quad & M(\boldsymbol{x_i}) = \sum_{t=1}^{T} \beta_t G_t(\boldsymbol{x_i}),\ i = 1, \dots, n \end{aligned}$$

Note that the constant factor $\frac{1}{n}$ was removed because it does not change the optimal solution. The choice of loss function is crucial to the design of the AdaBoost learning method. We will assume that the AdaBosst metalearner uses the exponential loss function, which is defined such that

$$L(\rho, \ell) = e^{-\rho \ell} = \begin{cases} e^{-\rho} & \text{if } l = 1 \\ e^{\rho} & \text{if } l = -1 \end{cases}.$$

Using the exponential loss function, we can express the objective function as

$$\begin{aligned} \sum_{i=1}^{n} L\big(M(\boldsymbol{x_i}), y_i\big) &= \sum_{i=1}^{n} \exp\Big(-y_i M(\boldsymbol{x_i})\Big) = \sum_{i=1}^{n} \exp\left(-y_i \sum_{t=1}^{T} \beta_t G_t(\boldsymbol{x_i})\right) \\ &= \sum_{i=1}^{n} \exp\left(\sum_{t=1}^{T} -y_i \beta_t G_t(\boldsymbol{x_i})\right) = \sum_{i=1}^{n} \prod_{t=1}^{T} \exp\Big(-y_i \beta_t G_t(\boldsymbol{x_i})\Big). \end{aligned}$$

Since $G_1(\boldsymbol{x_i}), \dots, G_{T-1}(\boldsymbol{x_i})$ and $\beta_1, \dots, \beta_{T-1}$ are known values for $i = 1, \dots, n$, we will simply the expression above by defining the following constants:

$$w_i^{(T)} := \prod_{t=1}^{T-1} \exp\Big(-y_i \beta_t G_t(\boldsymbol{x_i})\Big),\ i = 1, \dots, n.$$

This now allows us to express the objective of our optimization problem as

$$\sum_{i=1}^{n} L\big(M(\boldsymbol{x_i}), y_i\big) = \sum_{i=1}^{n} w_i^{(T)} \exp\Big(-y_i \beta_T G_T(\boldsymbol{x_i})\Big).$$

Recall that the true labels, $y_1, \dots, y_n$, as well as the predictions, $G_1(\boldsymbol{x_i}), \dots, G_{T-1}(\boldsymbol{x_i})$, either take on the value one or negative one. Therefore, the product of the true label and the prediction for a sample point, $\boldsymbol{x_i}$, is given by

$$y_i G_T(\boldsymbol{x_i}) = \begin{cases} 1 & \text{if } y_i = G_T(\boldsymbol{x_i}) \\ -1 & \text{if } y_i \neq G_T(\boldsymbol{x_i}) \end{cases}.$$

With this observation, we can separate the objective function into two sums:

$$\begin{aligned}\sum_{i=1}^{n} L\big(M(\boldsymbol{x_i}), y_i\big) &= \sum_{i:y_i=G_T(\boldsymbol{x_i})} w_i^{(T)} \exp\Big(-y_i\beta_T G_T(\boldsymbol{x_i})\Big) \\ &\qquad + \sum_{i:y_i\neq G_T(\boldsymbol{x_i})} w_i^{(T)} \exp\Big(-y_i\beta_T G_T(\boldsymbol{x_i})\Big) \\ &= \sum_{i:y_i=G_T(\boldsymbol{x_i})} w_i^{(T)} \exp\Big(-\beta_T(1)\Big) \\ &\qquad + \sum_{i:y_i\neq G_T(\boldsymbol{x_i})} w_i^{(T)} \exp\Big(-\beta_T(-1)\Big) \\ &= e^{-\beta_T} \sum_{i:y_i=G_T(\boldsymbol{x_i})} w_i^{(T)} + e^{\beta_T} \sum_{i:y_i\neq G_T(\boldsymbol{x_i})} w_i^{(T)}\end{aligned}$$

We can simplify the objective by adding an expression of zero to it. Notice that

$$\begin{aligned}\sum_{i=1}^{n} L\big(M(\boldsymbol{x_i}), y_i\big) &= e^{-\beta_T} \sum_{i:y_i=G_T(\boldsymbol{x_i})} w_i^{(T)} + e^{\beta_T} \sum_{i:y_i\neq G_T(\boldsymbol{x_i})} w_i^{(T)} \\ &\qquad + \Big(e^{-\beta_T} - e^{-\beta_T}\Big) \sum_{i:y_i\neq G_T(\boldsymbol{x_i})} w_i^{(T)} \\ &= e^{-\beta_T} \left(\sum_{i:y_i=G_T(\boldsymbol{x_i})} w_i^{(T)} + \sum_{i:y_i\neq G_T(\boldsymbol{x_i})} w_i^{(T)} \right) \\ &\qquad + \Big(e^{\beta_T} - e^{-\beta_T}\Big) \sum_{i:y_i\neq G_T(\boldsymbol{x_i})} w_i^{(T)} \\ &= e^{-\beta_T} \sum_{i=1}^{n} w_i^{(T)} + \Big(e^{\beta_T} - e^{-\beta_T}\Big) \sum_{i:y_i\neq G_T(\boldsymbol{x_i})} w_i^{(T)}.\end{aligned}$$

17.4.3 Optimal Classifier Prediction

The first term in the objective function does not depend on the choose of classifier, G_T, and the second term contains a constant factor which also does not depend on G_T. Now we can see that the best Tth classifier, which minimizes the total risk, solves the following optimization problem:

$$\min_{G_T} \sum_{i:y_i\neq G_T(\boldsymbol{x_i})} w_i^{(T)}.$$

This says that the optimal choice for classifier G_T minimizes the sum of the weights over all misclassified points. The weight function, $w_i^{(T)}$, is a bit complicated, but we can compute it recursively by noticing the following:

$$w_i^{(T+1)} = w_i^{(T)} \exp\Big(-y_i\beta_T G_T(\boldsymbol{x_i})\Big) = \begin{cases} w_i^{(T)} e^{-\beta_T} & \text{if } y_i = G_T(\boldsymbol{x_i}) \\ w_i^{(T)} e^{\beta_T} & \text{if } y_i \neq G_T(\boldsymbol{x_i}) \end{cases}.$$

This recursive formulation is a benefit of choosing the exponential loss for the metalearner. Notice that the weight for a sample point, $\boldsymbol{x_i}$, shrinks if the point was classified correctly by classifier G_T and grows if the point was misclassified.

17.4.4 Optimal Classifier Weight

Now to choose the optimal weight, β_T, we can take the derivative of the objective function with respect to β_T and set it equal to zero, which gives us

$$-\beta e^{-\beta_T} \sum_{i=1}^{n} w_i^{(T)} + \left(\beta e^{\beta_T} + \beta e^{-\beta_T}\right) \sum_{i:y_i \neq G_T(\boldsymbol{x_i})} w_i^{(T)} = 0.$$

Dividing both sides by the first term in the expression above, we get

$$1 + \left(-e^{2\beta_T} - 1\right) \frac{\sum_{i:y_i \neq G_T(\boldsymbol{x_i})} w_i^{(T)}}{\sum_{i=1}^{n} w_i^{(T)}} = 0.$$

To simply the notation, we'll define the weighted error rate for classifier G_T as

$$\text{err}_T = \frac{\sum_{i:y_i \neq G_T(\boldsymbol{x_i})} w_i^{(T)}}{\sum_{i=1}^{n} w_i^{(T)}}.$$

This now allows us to write our previous expression more simply as

$$1 - \left(e^{2\beta_T} + 1\right)\text{err}_T = 0.$$

Now we can express the optimal weight β_T in terms of the error rate as

$$\beta_T = \frac{1}{2} \ln\left(\frac{1 - \text{err}_T}{\text{err}_T}\right).$$

Note that the learner weight, β_T, is monotonically decreasing with respect to the error rate, err_T, and is anti-symmetric about $1/2$. If classifier G_T is a perfect classifier whose error rate is zero, then it receives the weight $\beta_T = \infty$. If classifier G_T is equivalent to a random classifier whose error rate is $1/2$, then it receives the weight $\beta_T = 0$. It intuitively makes sense that a more accurate classifier receives more weight in determining the output of the metalearner. Interestingly, a classifier with a training accuracy of 40% is just as useful as a classifier with a training accuracy of 60% because a classifier with training accuracy under 50% receives a negative weight.

17.4.5 Adaboost Algorithm

Now that we have an expression for the optimal classifier, G_T, and corresponding coefficient, β_T, given previous classifiers and weights, we can put together a complete AdaBoost training algorithm, which is shown in algorithm 9.
After completing training, we can classify a test point, $\boldsymbol{z}$, by computing

$$h(\boldsymbol{z}) = \text{sign}(M(\boldsymbol{z})) = \text{sign}\left(\sum_{t=1}^{T} \beta_t G_t(\boldsymbol{z})\right).$$

Algorithm 9: AdaBoost Algorithm

Initialize weights: $w_i \leftarrow \frac{1}{n},\ i = 1, \ldots, n$
for $t \leftarrow 1$ *to* T **do**
    Train classifier G_t with weights $w_1, \ldots, w_n$
    Compute weighted error rate: $\text{err} \leftarrow \frac{\sum_{i:y_i \neq G_t(\boldsymbol{x_i})} w_i}{\sum_{i=1}^{n} w_i}$
    Compute the coefficient: $\beta_t \leftarrow \frac{1}{2} \ln\left(\frac{1-\text{err}}{\text{err}}\right)$
    Update the weights: $w_i \leftarrow \begin{cases} w_i e^{-\beta_t} & \text{if } y_i = G_t(\boldsymbol{x_i}) \\ w_i e^{\beta_t} & \text{if } y_i \neq G_t(\boldsymbol{x_i}) \end{cases}$
end
return G_t, β_t for $t = 1, \ldots, T$

17.4.6 Important Notes

Below are some more important notes about AdaBoost:

1. While AdaBoost can be used with any classifier, classifiers that use linear decision boundaries do not work well with the Adaboost algorithm.

2. The exponential loss function is sensitive to outliers, so we should use another loss function if the data has been corrupted.

3. If each individual classifier achieves a training accuracy above a threshold, $\mu > 50\%$, then the training accuracy of the metalearner will eventually reach 100% with enough classifiers. More specifically, we can prove that if $\text{err}_t < 0.5$ for every learner, the number of samples misclassified by the metalearner goes to zero as the number of classifiers, T, goes to infinity.

17.4.7 Short Decision Trees

The AdaBoost method can be used with any classifier, but it is most commonly used with short decision trees for the following reasons:

1. AdaBosst trains several classifiers, and decision trees are fast to train, especially if they are short.

2. AdaBoost with decision trees can obtain a good classifier without searching for hyperparamaters.

3. It is easy to design a decision tree that consistently surpasses some minimum training accuracy threshold that is slightly above 50%.

4. AdaBoost with decision trees gives us a good amount of control over the amount of bias and variance. As you train more learners, the AdaBoost bias generally decreases and the variance often decreases at first but later increases. In some cases, boosting can lead to overfitting after many iterations, but using short decision trees helps reduce overfitting.

5. AdaBoost with short decision trees is a form of feature subset selection. When we use AdaBoost with decision trees, we assign a coefficient with greater magnitude to decision trees with more predictive power and a coefficient closer to zero to decision trees with a training accuracy closer to 50%. Short decision trees split on only a few features for some threshold values. Therefore, if a short decision tree uses features with little predictive power, then we expect its classification ability to be close to random, resulting in a coefficient close to zero. These features with little predictive power will then not be used by the metalearner for classification. This helps to reduce overfitting and running time.