# 5 Machine Learning Abstractions and Numerical Optimization

## ML ABSTRACTIONS [some meta comments on machine learning]

[When you write a large computer program, you break it down into subroutines and modules. Many of you know from experience that you need to have the discipline to impose strong abstraction barriers between different modules, or your program will become so complex you can no longer manage nor maintain it.]

[When you learn a new subject, it helps to have mental abstraction barriers, too, so you know when you can replace one approach with a different approach. I want to give you four levels of abstraction that can help you think about machine learning. It's important to make mental distinctions between these four things, and the code you write should have modules that reflect these distinctions as well.]

APPLICATION/DATA

data labeled or not?
yes: labels categorical (classification) or quantitative (regression)?
no: similarity (clustering) or positioning (dimensionality reduction)?

---

MODEL [what kinds of hypotheses are permitted?]

e.g.:
– decision fns: linear, polynomial, logistic, neural net, ...
– nearest neighbors, decision trees
– features
– low vs. high capacity (affects overfitting, underfitting, inference)

---

OPTIMIZATION PROBLEM

– variables, objective fn, constraints
e.g., unconstrained, convex program, least squares, PCA

---

OPTIMIZATION ALGORITHM

e.g., gradient descent, simplex, SVD

[In this course, we focus primarily on the middle two levels. As a data scientist, you might be given an application, and your challenge is to turn it into an optimization problem that we know how to solve. We will talk about optimization algorithms, but usually data analysts use optimization codes that are faster and more robust than what they would write themselves.]

[The second level, the model, has a huge effect on the success of your learning algorithm. Sometimes you get a big improvement by tailoring the model or its features to fit the structure of your specific data. The model also has a big effect on whether you overfit or underfit. And if you want a model that you can interpret so you can do *inference*, the model has to have a simple structure. Lastly, you have to pick a model that leads to an optimization problem that can be solved. Some optimization problems are just too hard.]

[It's important to understand that when you change something in one level of this diagram, you probably have to change all the levels underneath it. If you switch your model from a linear classifier to a neural net, your optimization problem changes, and your optimization algorithm changes too.]

[Not all machine learning methods fit this four-level decomposition. Nevertheless, for everything you learn in this class, think about where it fits in this hierarchy. If you don't distinguish which math is part of the model and which math is part of the optimization algorithm, this course will be very confusing for you.]

## OPTIMIZATION PROBLEMS

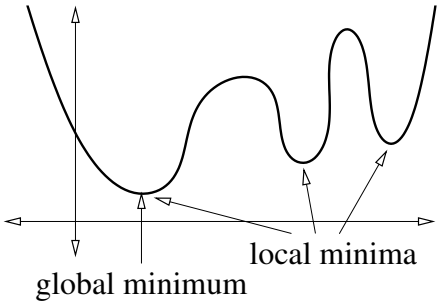
[I want to familiarize you with some types of optimization problems that can be solved reliably and efficiently, and the names of some of the optimization algorithms used to solve them. An important skill for you to develop is to be able to go from an application to a well-defined optimization problem. That skill depends on your ability to *recognize* well-studied types of optimization problems.]

### Unconstrained

Goal: Find $w$ that minimizes (or maximizes) a continuous objective fn $f(w)$.

$f$ is smooth if its gradient is continuous too.

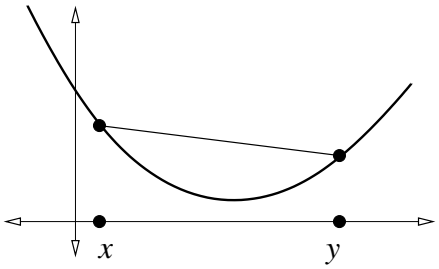
A global minimum of $f$ is a value $w$ such that $f(w) \leq f(v)$ for every $v$.
A local minimum " " " " " " " " " "
for every $v$ in a tiny ball centered at $w$.
[In other words, you cannot walk downhill from $w$.]




[Draw this by hand. minima.pdf ]

Usually, finding a local minimum is easy;
finding the global minimum is hard. [or impossible]

Exception: A function is convex if for every $x, y \in \mathbb{R}^d$,
the line segment connecting $(x, f(x))$ to $(y, f(y))$ does not go below $f(\cdot)$.




[Draw this by hand. convex.pdf ]

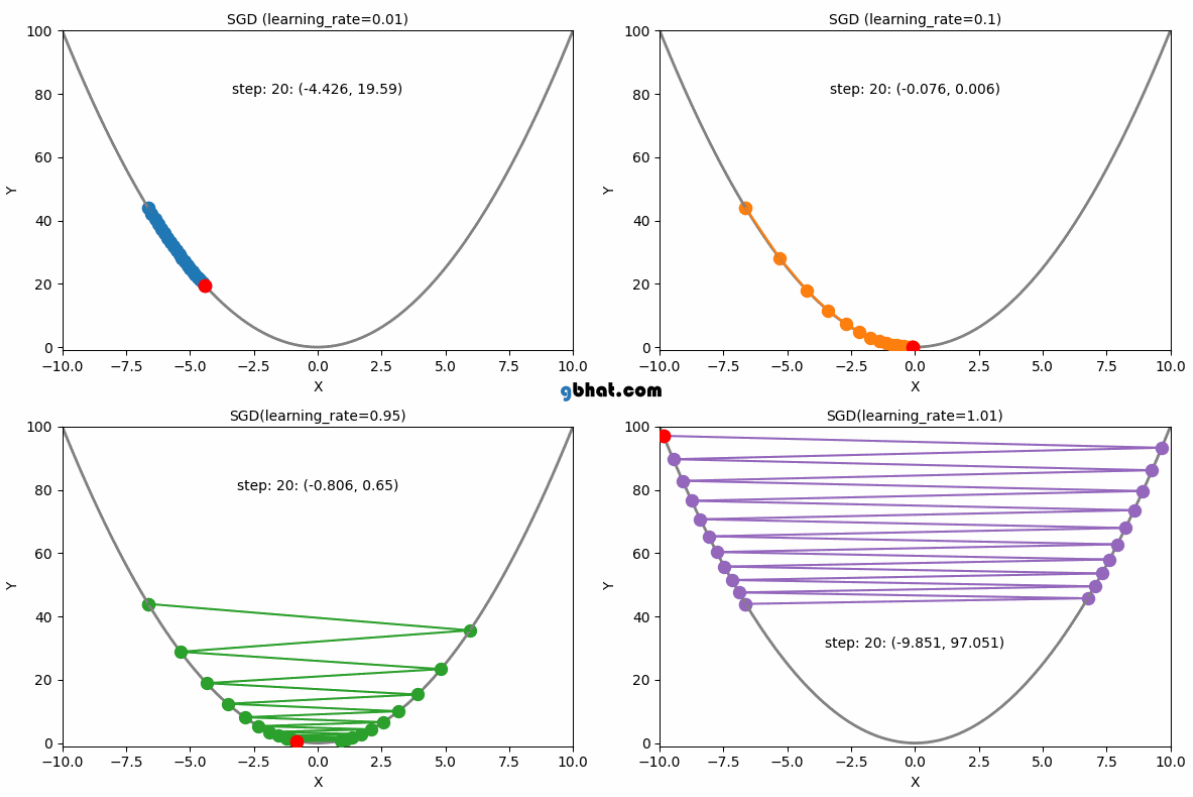
Formally: for every $x, y \in \mathbb{R}^d$ and $\beta \in [0, 1]$, $f(x + \beta(y - x)) \leq f(x) + \beta(f(y) - f(x))$.
E.g., perceptron risk fn is convex and nonsmooth.

[When you sum together convex functions, you always get a convex function. The perceptron risk function is a sum of convex loss functions, so it is convex.]

A [continuous] convex function [on a closed, convex domain] has either
- no minimum (goes to $-\infty$), or
- just one local minimum, or
- a connected set of local minima that are all global minima with equal $f$.

[The perceptron risk function is in the last category.]
[In the last two cases, if you walk downhill, you eventually reach a global minimum.]

Gradient descent: repeat $w \leftarrow w - \epsilon \nabla f(w)$




learningrates20.gif (Gajanan Bhat, gbhat.com) [Gradient descent with different learning rates $\epsilon$. Top left: painfully small. Top right: reasonable, but still smaller than ideal. Bottom left: reasonable, but larger than ideal. Bottom right: too large; diverges. This is an animated GIF; see https://gbhat.com/machine_learning/gradient_descent_learning_rates.html .]

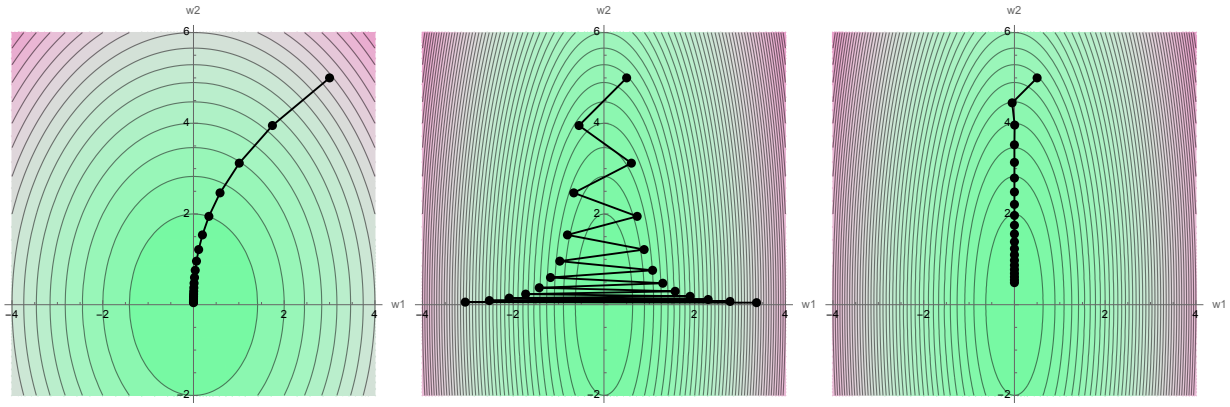
- Fails/diverges if $\epsilon$ too large.
- Slow if $\epsilon$ too small.
- $\epsilon$ often optimized by trial & error [for slow learners like neural networks].

[The best value of $\epsilon$ is hard to guess. One common technique for dealing with divergence is to check whether a step of gradient descent increases the function value rather than decreasing it; if so, reduce the step size.]

[That's a simple example of what's called an *adaptive learning rate* or a *learning rate schedule*. These adaptations become even more important when you do stochastic gradient descent or when you optimize non-convex, very twisty objective functions. We'll revisit the idea when we learn neural networks.]

[One interesting aspect of gradient descent that these figures illustrate is that it usually never reaches the exact local minimum. Instead, it gets closer and closer forever, but never exactly reaches the true minimum. We call this behavior "convergence." The last question of Homework 2 will give you some understanding of why convergence happens under the right conditions.]

[When we have a feature space with more than one dimension, another problem arises, which is that the learning rate that's good for one direction might be terrible in another direction. Consider the three examples of gradient descent below.]




goodcondition.pdf, illcondition105.pdf, illcondition055.pdf [Left: 20 iterations of gradient descent on a well-conditioned quadratic function, $f(w) = 2w_1^2 + w_2^2$, with a modest step size $\epsilon = 0.105$. Center: 20 iterations on an ill-conditioned function, $f(w) = 10w_1^2 + w_2^2$; the same step size is now too large. Right: after reducing the step size to $\epsilon = 0.055$, we have convergence again but we aren't approaching the minimum nearly as quickly.]

[The step size that works for the left example is too large for the center example; it diverges in the $w_1$-direction. At right, we reduce the step size and obtain convergence. But now convergence is slow in the $w_2$-direction.]

High ellipticity of the contours, a.k.a. ill-conditioning of the Hessian, means no learning rate is good in all directions.

[The Hessian matrix is said to be ill-conditioned if its largest eigenvalue is much larger than its smallest eigenvalue. Ill-conditioning can be a problem even for simple methods like linear regression, making it harder to solve the problem. In response to these observations, there are adaptive learning rate algorithms that explicitly choose different learning rates for different weights. Famous examples are Adam and RMSprop.]

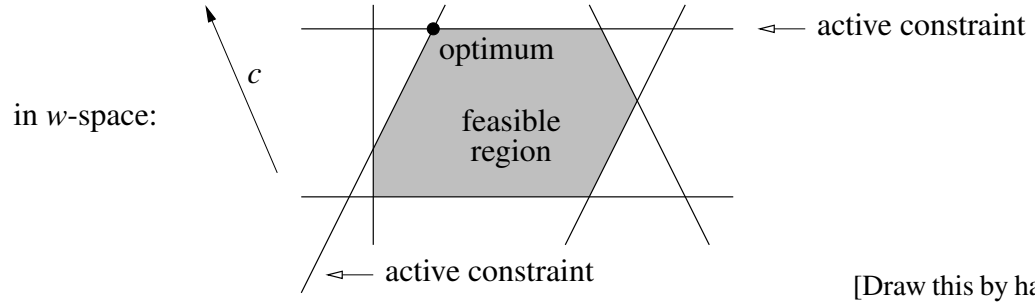
[There are many applications where you don't have a convex objective function. Then gradient descent usually can find a local minimum, but not necessarily a global minimum. And often there is no guarantee that the local minimum you find will be nearly as good as the global minimum. Nevertheless, gradient descent is used for a lot of nonconvex machine learning problems too. For example, neural networks try to optimize an objective function that has *lots* of local minima. But stochastic gradient descent is still the algorithm of choice for training neural nets. We'll talk more later in the semester about why.]

**Linear Program**

Linear objective fn + linear **inequality** constraints.

Goal: Find $w$ that maximizes (or minimizes) $c \cdot w$
subject to $Aw \leq b$

where $A$ is $n \times d$ matrix, $b \in \mathbb{R}^n$, expressing $n$ linear constraints:
$A_i \cdot w \leq b_i, \qquad i \in [1, n]$




[Draw this by hand. linprog.pdf ]

The set of points $w$ that satisfy all constraints is a convex polytope called the feasible region $F$ [shaded].
The optimum is the point in $F$ that is furthest in the direction $c$. [What does convex mean?]
A point set $P$ is convex if for every $p, q \in P$, the line segment with endpoints $p$, $q$ lies entirely in $P$.

[What is a polytope? Just a polyhedron, generalized to higher dimensions.]

The optimum achieves equality for some constraints (but not most), called the active constraints of the optimum. [In the figure above, there are two active constraints. In an SVM, active constraints correspond to the training points that touch or violate the slab, and these points are also known as support vectors.]

[Sometimes, there is more than one optimal point. For example, in the figure above, if $c$ pointed straight up, every point on the top horizontal edge would be optimal. The set of optimal points is always convex.]

Example: EVERY feasible point $(w, \alpha)$ gives a linear classifier:

| Find $w$, $\alpha$ that satisfies $y_i(w \cdot X_i + \alpha) \geq 1 \quad$ for all $i \in [1, n]$ |
|---|

[This is the problem of finding a feasible point. This problem can be cast as a slightly different linear program that uses an objective function to make all the inequalities be satisfied *strictly* if that's possible.]

IMPORTANT: The data are linearly separable iff the feasible region is not the empty set.
→ Also true for maximum margin classifier (quadratic program)

[The most famous algorithm for linear programming is the simplex algorithm, invented by George Dantzig in 1947. The simplex algorithm is indisputably one of the most important and useful algorithms of the 20th century. It walks along edges of the feasible region, traveling from vertex to vertex until it finds an optimum.]

[Linear programming is very different from unconstrained optimization; it has a much more combinatorial flavor. If you knew which constraints would be the active constraints once you found the solution, it would be easy; the hard part is figuring out which constraints should be the active ones. There are exponentially many possibilities, so you can't afford to try them all. So linear programming algorithms tend to have a very discrete, computer science feeling to them, like graph algorithms, whereas unconstrained optimization algorithms tend to have a continuous, numerical mathematics feeling.]

[Linear programs crop up *everywhere* in engineering and science, but they're usually in disguise. An extremely useful talent you should develop is to recognize when a problem is a linear program.]

[A linear program solver can find a linear classifier, but it can't find the maximum margin classifier. We need something more powerful.]

## Quadratic Program

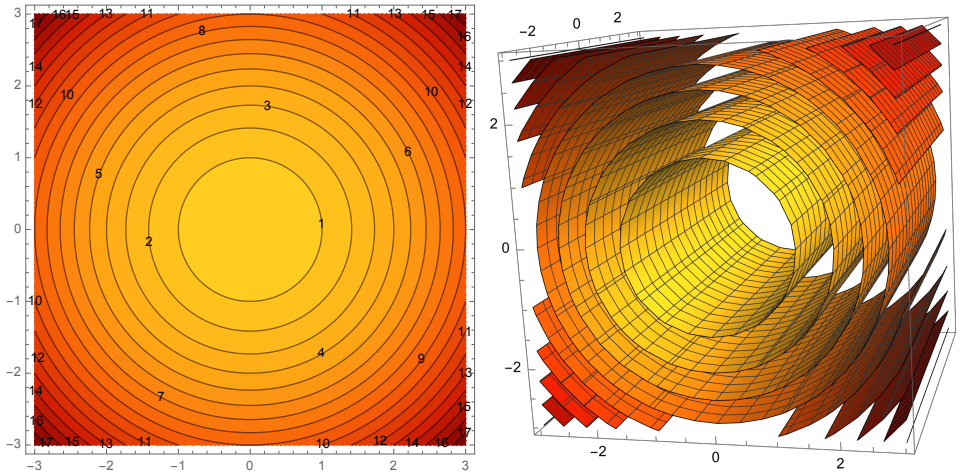
Quadratic, convex objective fn + linear inequality constraints.

Goal: Find $w$ that minimizes $f(w) = w^\top Q w + c^\top w$
subject to $Aw \leq b$

where $Q$ is a symmetric, positive semidefinite matrix.

[A matrix is positive semidefinite if $w^\top Q w \geq 0$ for all $w$.]

Example: Find maximum margin classifier.



quadratic.pdf, quadratic3D.pdf [Left: A hard-margin SVM minimizes the objective function $w_1^2 + w_2^2$. Right: There is also an $\alpha$-axis, so the isosurfaces of the objective function are really cylinders. On the left isocontours, draw two polygons—one with one active constraint, and one with two—and show the constrained minimum for each polygon. "In a hard-margin SVM, we are looking for the point in this polygon that's closest to the $\alpha$-axis."]

[If $Q$ is positive definite, a quadratic program has just one unique local minimum, which is therefore the global minimum. But in a support vector machine, $Q$ is not definite; it is only positive semidefinite, because the bias term $\alpha$ is a weight but it does not influence the objective function. Sometimes positive semidefinite quadratic programs have multiple solutions, but SVMs are a special case where there is only one unique minimum. By the way, if $Q$ is indefinite, then $f$ is not convex, the minimum is not always unique, and quadratic programming is NP-hard. But we won't need that kind of quadratic program in this class.]

Algs for quadratic programming:

- Simplex-like [commonly used for general-purpose quadratic programs, but not as good for SVMs as the following two algorithms that specifically exploit properties of SVMs]
- Sequential minimal optimization (SMO, used in LIBSVM, "SVC" in scikit)
- Coordinate descent (used in LIBLINEAR, "LinearSVC" in scikit)

Numerical optimization @ Berkeley: EECS 127/227AT/227BT/227C.