CS 189 Introduction to Machine Learning
Spring 2019 Jonathan Shewchuk

HW5

Due: Wednesday, April 3 at 11:59 pm

Deliverables:

1. Submit your predictions for the test sets to Kaggle as early as possible. Include your Kaggle scores in your write-up (see below). The Kaggle competition for this assignment can be found at:
   - Spam dataset: `https://www.kaggle.com/t/a9049f15799147cf843ae0b040b1754b`
   - Titanic dataset: `https://www.kaggle.com/t/e6248db4571544b8bad7d6681a472b35`

2. Submit a PDF of your homework, **with an appendix listing all your code**, to the Gradescope assignment entitled "Homework 5 Write-Up". In addition, please include, as your solutions to each coding problem, the specific subset of code relevant to that part of the problem. You may typeset your homework in LaTeX or Word (submit PDF format, **not** .doc/.docx format) or submit neatly handwritten and scanned solutions. **Please start each question on a new page.** If there are graphs, include those graphs in the correct sections. **Do not** put them in an appendix. We need each solution to be self-contained on pages of its own.
   - In your write-up, please state with whom you worked on the homework.
   - In your write-up, please copy the following statement and sign your signature next to it. (Mac Preview and FoxIt PDF Reader, among others, have tools to let you sign a PDF file.) We want to make it *extra* clear so that no one inadverdently cheats.

     "I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted."

3. Submit all the code needed to reproduce your results to the Gradescope assignment entitled "Homework 5 Code". Yes, you must submit your code twice: in your PDF write-up following the directions as described above so the readers can easily read it, and once in compilable/interpretable form so the readers can easily run it. Do **NOT** include any data files we provided. Please include a short file named README listing your name, student ID, and instructions on how to reproduce your results. Please take care that your code doesn't take up inordinate amounts of time or memory. If your code cannot be executed, your solution cannot be verified.

1 Movement Dataset

Important: Peruse the entire problem at least once before attempting any part. Do not be intimidated by the length. We just want to be very precise with the instructions. Once you understand them, the problem itself should not take very long to do. It should be pretty fun! And the result will be rewarding.

High-quality data is imperative to apply machine learning algorithms to real-world problems. Understanding the data collection process will aid you enormously in future ML endeavors. In preparation for the neural network homework (Homework 6), we will collectively build an eight-movement dataset, MDS189. For your part, you are to: 1) record videos of yourself (or a friend) doing each movement, 2) upload your videos (can be unlisted, but not private) to YouTube, 3) extract the frames from each video, 4) use the extracted frames to label each video with one `json` file, 5) label the body joints in 'key frames', and 6) submit your data using **this** Google Form.

Very Important:

1. You are not required to be the subject of your video recordings. You are welcome to ask a friend.
2. You are not required to have your data be included in the MDS189 dataset that we release to the class for the neural network homework. When you submit your data, you will have the option to opt out of including your recordings in MDS189.
3. If you prefer, you are welcome to wear a ski mask, head covering, sunglasses, or other mechanism of hiding your face.
4. If you are not comfortable with any of the above options, please contact your friendly GSI Panna (panna@berkeley), and we will find a suitable alternative.

Data Use Disclosure: MDS189 will *only* be used for Homework 6 in *this* class *this* semester. It will not be publicly released, and it will not be used in any other or subsequent class. If you opt out of including your data in MDS189, your data will be used solely for grading your Homework 5. If you opt in to including your data in MDS189 (thank you!), please note that while your CalNet ID is used in your label files (Section 1.4), it will not be included in MDS189. We are only asking for CalNet IDs in your label files for grading purposes in this assignment.

The experience of building a dataset can be very rewarding (especially when you get to start applying learning algorithms to that data), so we want everyone to have the opportunity to contribute. Hopefully the MDS189 analysis we get to do in the next homework will be exciting and potentially novel, as no other dataset like this exists!

1.1 The Movements in MDS189

A brief overview, the corresponding label (bold), and a sample YouTube link (boxed) for each movement:

1. **reach** An overhead reach for the sky.
2. **squat** An overhead bodyweight squat, i.e., an air squat with your arms reaching for the sky.
3. **inline** An inline step and touch, alternating legs.
4. **lunge** A forward lunge, alternating legs.

5. $\boxed{\textbf{hamstrings}}$ A leg raise, alternating legs.
6. $\boxed{\textbf{stretch}}$ A windshield wiper stretch, alternating legs.
7. $\boxed{\textbf{deadbug}}$ A fun core stability movement.
8. $\boxed{\textbf{pushup}}$ A chest and core movement, in which you push yourself up off the ground.

Setup and record instructions, and 'key frame' definitions are in the YouTube video descriptions. For each movement video, we have defined 1-2 key frames (images) that are the snapshots that are most representative of the movement. To produce your label files (Section 1.4), you will need to specify the frame number(s) of the key frame(s) in each of your movement videos. Note: please don't assume you already know how to do a movement (e.g., pushup) from your own experience. It is imperative that you watch the sample videos and read their corresponding descriptions before recording your own.

Very Important: These movements should not cause any pain. **STOP** the movement immediately if you (or your friend) experience any pain. If you have a disability that prevents you from doing a (set of) movement(s), and you want to be the subject of your video recordings for the movements you are able to do, proceed as follows.

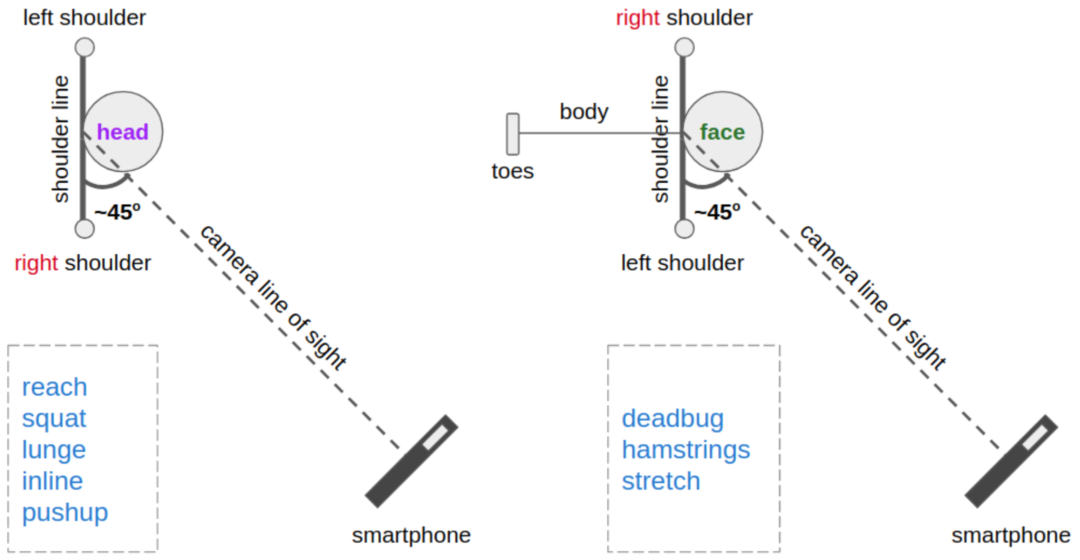
1. Please state in your write-up that you are physically unable to do XYZ movement(s).
2. Please use my video recording(s) for the XYZ movement(s) you cannot do. You can download my (Panna's) videos from $\boxed{\textbf{here}}$. Only for the videos of me, please use the subject id `panna` in your annotation files. Note: The script I provide (details in Section 1.4) to help automate annotation file production assumes the same subject ID across all videos, so please don't forget to update the relevant subject IDs!

If there's any specific concern, please reach out to your friendly GSI Panna (panna@berkeley).

1.2 Video Collection Instructions

We want to make MDS189 as consistent as possible. To that end, please follow these instructions.

- There must only be **one** person, the subject, visible in each video. The subject should be fully visible. If someone else is visible at all, you need to re-record. There should be no mirrors in your videos.
- For the safety of the subject, we suggest that the subject should ideally be barefoot and wearing clothes that don't hinder movement. This is **not** a requirement; the subject is free to wear socks. However, the TA staff found that performing these motions while wearing socks on slippery surfaces can be dangerous, so please take care.
- The orientation between the subject and camera must match the sample videos. The camera should be at a 45° angle to the subject's shoulder line. In standing videos, the subject's face and both shoulders should be visible. In ground videos, the head should be closer to the camera than the feet. See Fig. 1.
- Record all videos in portrait mode on a smartphone, ideally at **30 frames per second**; otherwise please re-code to 30 fps.
- Record all videos with the subject maintaining a forward, neutral gaze throughout.



Do not stress about getting exactly a 45° angle. The main point is that it's not a frontal or side view. A good rule of thumb is that both shoulders should be fairly visible. Look at the provided example videos to help guide you.

Figure 1: Top-down view of camera-subject orientation for the various movements.

- Be consistent with the same, unique subject for all your videos.
- If you're the subject, it's probably easiest to just ask a friend to help record you. Recording shouldn't take more than 10–15 minutes. No video should be longer than 20 seconds. The camera should be stationary.
- Perform unilateral movements (3–7) with your **left leg first**.

1.3 Video Frame Extraction

A video is a sequence of images (frames). For this part, you will extract the frames from each video using any method you like. Some possible methods for this are: 1) VLC media player has the option to extract frames, which you can access through the GUI or the terminal (option 2/3 **here**), 2) `ffmpeg` through the terminal, and 3) in Python with OpenCV's `VideoCapture` (see **here** for some example code). Make sure the image quality is not diminished. Something is wrong if your video resolution looks nice, but your extracted frames look pixelated.

It may be a good for you to experience `ffmpeg`, as it's ubiquitous in ML with video data. For that reason, we include a shell script, `extractFrames.sh`, where you can see an example of how to use `ffmpeg` to extract frames for all videos in a directory. Please see the comments in the script for details on how to run it. You are more than welcome to use the provided script to extract your frames. There are many good resources available on the web describing how to install `ffmpeg` on your machine, if it's not already there.

In your label files, you will need to list the frame number of the key frame(s) for each video (check YouTube descriptions for which frame is the relevant key frame for each movement), so it is best if you extract frames whose filenames are four digits followed by "`.jpg`"; for example, `0001.jpg`. Use leading zeros where needed to have four digits. These leading zeros are critical for your filenames to be in consecutive order when you list them in a directory. In particular, it avoids the situation where you have files like `1.jpg` next to `10.jpg`. (The print code for four-digit integers with leading zeros is `0%4d`). Frame labels should be indexed from 1 (i.e., the first extracted frame is `0001.jpg`, **not** `0000.jpg`), which is the default for `ffmpeg` and `VLC`. Frames must be `.jpg` files.

Bilateral movement videos have 1 key frame each, and unilateral ones have 2 key frames each. You'll have 13 unique, labeled key frames in total.

1.4 Label Instructions

Upload your videos to YouTube. We recommend leaving your videos unlisted so that nobody can find them without knowing the URL. *For each video*, you are to produce one label `.json` file, with the structure specified in `label_format.json`. **This** has a lot of details about JSON files—you can think of them as Python dictionaries printed in a text file. When you interact with the contents of a JSON file in Python, it should feel like a dictionary with key-value pairs. This is one reason why JSON files make for a convenient way of keeping track of the labels associated with data.

You will include the video's **shortened** YouTube link in its corresponding label file, which is why you need to upload your videos to YouTube before generating the label files. You will also include the specified frame annotations (see `label_format.json` for details) in your label files, so you must extract the frames before generating the label files.

We provide `createLabelFile.py` that may help you create label files. In the script, you will see many `TODO` comments, which tell you how to modify the script to fit your data. You will also see instructions for how to run the script at the top of the file. Alternatively, you can copy the template `label_format.json` and manually enter your label data. Or, you can copy the sample label files we provide (see Section 1.7), and modify the contents to match your label data.

1.5 Keypoint Annotations in Key Frames

You will label the 2D body joints (keypoints) in the 13 key frames, following OpenPose **BODY_25**. Copy your 13 key frames, which you identified when generating your label files, into a directory called `key_frames`. One way to do this is manually copying the images. Another way is through Python, where you loop through your label files and copy the key frames you labeled. We provide an example of this second approach in `copyKeyFrames.py`. In the script, you will see `TODO` comments, which tell you how to modify the script to fit your data. You will also see instructions for how to run the script at the top of the file. **Note**: Looking at these key frames is a very good sanity check on your key frame labels.

We will use **Labelbox** to mark the keypoints. After logging into Labelbox, upload your 13 key frames to a new project. Select the existing Image Labeling interface by clicking the blue Select button. In the configuration interface window on the left, you should see a JSON tab, which should show you the JSON associated with the label interface. Replace the contents with the data in `labelbox_template.json`. Press the blue Confirm button, and then Start Labeling! Roughly center keypoints on the body joints. Do your best. If a joint is occluded, and too hard to tell where it is, don't mark it. We have included sample images with keypoint annotations in the Google Drive **folder**. When you are all done labeling, export your labels to a JSON using the default settings. Leave the default filename, and include it in your labels folder.

1.6 Upload Instructions

When you are done recording and labeling, your data should have the following file and folder structure:

```
your calnet ID
```

```
|__key_frames
|  |__13 .jpg images that you uploaded to labelbox
|__labels
   |__1 .json export file from labelbox with your keypoint annotations
   |__8 .json files with your annotations for each of your videos
```

Save to a `zip` file. Run the test script `checkData.py` to verify your data structure before **uploading**.

1.7 Resources and Examples

All scripts are located in `resources` in the Google drive **folder**. We also include the intermediate and final results for one example in the `heatherlckwd` subfolder, and a couple more examples of recorded movements in `more_example_videos`, including my friend's 75 year old grandma doing the movements.

1.8 To Include in Your Write-up

1. A brief description of how you extracted the frames. If you used a shell command, just list the command used. If you used some other software, just list the software used.
2. Paste (or screenshot) the output from running `checkData.py` on your `zip` file.
3. A screenshot of the message you see displayed after submitting your data through the **Google Form**.
4. Screenshots from Labelbox of your key frames with keypoint annotations. Crop these around the subject. Caption them with the movement label and 'left', 'right', or 'both', telling us which leg is moving.

1.9 Grading

Data is an integral part of machine learning. A large part of research in this field relies heavily on the integrity of data in order to run algorithms and have faith in the results. Thus, it is vital that your data submission follows our guidelines. To help, we have provided many scripts and examples. One part of your grade will come from the write-up, where we will visually inspect your keypoint annotations based on images in your write-up. The second part will come from my run of my version of `checkData.py` on your `zip` file, where I also use `youtube-dl` to download your videos, check they're all 30 fps, and verify no two videos are identical. I will also run OpenPose on the downloaded videos and check that there's only one person in the field of view. The last part of your grade will come from a quick visual inspection of the videos, where we will check that the movement in the video corresponds to the label, and that you filmed from the correct angle. Within each of these three parts, we won't award partial credit on a per-video basis.

2 Decision Trees for Classification

In this problem, you will implement decision trees and random forests for classification on three datasets: 1) the spam dataset, and 2) a Titanic dataset to predict Titanic survivors. The data is with the assignment.

In lectures, you were given a basic introduction to decision trees and how such trees are trained. You were also introduced to random forests. Feel free to research different decision tree techniques online. You do not have to implement boosting, though it might help with Kaggle.

2.1 Implement Decision Trees

See the Appendix for more information. You are not allowed to use any off-the-shelf decision tree implementation. Some of the datasets are not "cleaned," i.e., there are missing values, so you can use external libraries for data preprocessing and tree visualization (in fact, we recommend it). Be aware that some of the later questions might require special functionality that you need to implement (e.g., max depth stopping criterion, visualizing the tree, tracing the path of a sample through the tree). You can use any programming language you wish as long as we can read and run your code with minimal effort. In this part of your writeup, **include your decision tree code.**

2.2 Implement Random Forests

You are not allowed to use any off-the-shelf random forest implementation. If you architected your code well, this part should be a (relatively) easy encapsulation of the previous part. In this part of your writeup, **include your random forest code.**

2.3 Describe implementation details

We aren't looking for an essay; 1–2 sentences per question is enough.

1. How did you deal with categorical features and missing values?
2. What was your stopping criterion?
3. How did you implement random forests?
4. Did you do anything special to speed up training?
5. Anything else cool you implemented?

2.4 Performance Evaluation

For each of the 3 datasets, train both a decision tree and random forest and report your training and validation accuracies. You should be reporting 8 numbers (2 datasets × 2 classifiers × training/validation). In addition, for both datasets, train your best model and submit your predictions to Kaggle. Include your Kaggle display name and your public scores on each dataset. You should be reporting 2 Kaggle scores.

2.5 Writeup Requirements for the Spam Dataset

1. (Optional) If you use any other features or feature transformations, explain what you did in your report. You may choose to use something like bag-of-words. You can implement any custom feature extraction code in `featurize.py`, which will save your features to a `.mat` file.

2. For your decision tree, and for a data point of your choosing from each class (spam and ham), state the splits (i.e., which feature and which value of that feature to split on) your decision tree made to classify it. An example of what this might look like:

   (a) (“viagra”) ≥ 2
   (b) (“thanks”) < 1
   (c) (“nigeria”) ≥ 3
   (d) Therefore this email was spam.

   (a) (“budget”) ≥ 2
   (b) (“spreadsheet”) ≥ 1
   (c) Therefore this email was ham.

3. Generate a random 80/20 training/validation split. Train decision trees with varying maximum depths (try going from depth = 1 to depth = 40) with all other hyperparameters fixed. Plot your validation accuracies as a function of the depth. Which depth had the highest validation accuracy? Write 1–2 sentences explaining the behavior you observe in your plot. If you find that you need to plot more depths, feel free to do so.

2.6 Writeup Requirements for the Titanic Dataset

Train a very shallow decision tree (for example, a depth 3 tree, although you may choose any depth that looks good) and visualize your tree. Include for each non-leaf node the feature name and the split rule, and include for leaf nodes the class your decision tree would assign. You can use any visualization method you want, from simple printing to an external library; the `rcviz` library on github works well.

A Appendix

Data Processing for Titanic

Here's a brief overview of the fields in the Titanic dataset. You will need to preprocess the dataset into a form usable by your decision tree code.

1. survived: the label we want to predict. 1 indicates the person survived, whereas 0 indicates the person died.
2. pclass: Measure of socioeconomic status. 1 is upper, 2 is middle, 3 is lower.
3. age: Fractional if less than 1.
4. sex: Male/female.
5. sibsp: Number of siblings/spouses aboard the Titanic.
6. parch: Number of parents/children aboard the Titanic.
7. ticket: Ticket number.
8. fare: Fare.
9. cabin: Cabin number.
10. embarked: Port of embarkation (C = Cherbourg, Q = Queenstown, S = Southampton)

You will face two challenges you did not have to deal with in previous datasets:

1. Categorical variables. Most of the data you've dealt with so far has been continuous-valued. Some features in this dataset represent types/categories. Here are two possible ways to deal with categorical variables:
   (a) (Easy) In the feature extraction phase, map categories to binary variables. For example suppose feature 2 takes on three possible values: 'TA', 'lecturer', and 'professor'. In the data matrix, these categories would be mapped to three binary variables. These would be columns 2, 3, and 4 of the data matrix. Column 2 would be a boolean feature $\{0, 1\}$ representing the TA category, and so on. In other words, 'TA' is represented by $[1, 0, 0]$, 'lecturer' is represented by $[0, 1, 0]$, and 'professor' is represented by $[0, 0, 1]$. Note that this expands the number of columns in your data matrix. This is called "vectorizing," or "one-hot encoding" the categorical feature.
   (b) (Hard, but more generalizable) Keep the categories as strings or map the categories to indices (e.g. 'TA', 'lecturer', 'professor' get mapped to $0, 1, 2$). Then implement functionality in decision trees to determine split rules based on the subsets of categorical variables that maximize information gain. You cannot treat these as normal continuous-valued features because ordering has no meaning for these categories (the fact that $0 < 1 < 2$ has no significance when $0, 1, 2$ are discrete categories).
2. Missing values. Some data points are missing features. In the `csv` files, these are represented by the value '`?`'. You have three approaches:

(a) (Easiest) If a data point is missing some features, remove it from the data matrix (**this is useful for your first code draft, but your submission must not do this**).

(b) (Easy) Infer the value of the feature from all the other values of that feature (e.g., fill it in with the mean, median, or mode of the feature. Think about which of these is the best choice and why).

(c) (Hard, but more powerful). Use *k*-nearest neighbors to impute feature values based on the nearest neighbors of a data point. In your distance metric you will need to define the distance to a missing value.

(d) (Hardest, but more powerful) Implement within your decision tree functionality to handle missing feature values based on the current node. There are many ways this can be done. You might infer missing values based on the mean/median/mode of the feature values of data points sorted to the current node. Another possibility is assigning probabilities to each possible value of the missing feature, then sorting fractional (weighted) data points to each child (you would need to associate each data point with a weight in the tree).

For Python:

It is recommended you use the following classes to write, read, and process data:

```
csv.DictReader
sklearn.feature_extraction.DictVectorizer (vectorizing categorical variables)
    (There's also sklearn.preprocessingOneHotEncoder, but it's much less clean)
sklearn.preprocessing.LabelEncoder
    (if you choose to discretize but not vectorize categorical variables)
sklearn.preprocessing.Imputer
    (for inferring missing feature values in the preprocessing phase)
```

If you use `csv.DictReader`, it will automatically parse out the header line in the `csv` file (first line of the file) and assign values to fields in a dictionary. This can then be consumed by `DictVectorizer` to binarize categorical variables.

To speed up your work, you might want to store your cleaned features in a file, so that you don't need to preprocess every time you run your code.

Approximate Expected Performance

For spam, using the base features and a regular decision tree, we got 74.4% testing accuracy. With a random forest, we get around 75% testing accuracy on Titanic. You can get better performance. This is a general ballpark range of what to expect; we will post cutoffs on Piazza.

Suggested Architecture

This is a complicated coding project. You should put in some thought about how to structure your program so your decision trees don't end up as horrific forest fires of technical debt. Here is a rough, **optional** spec that only covers the barebones decision tree structure. This is only for your benefit—writing clean code will make your life easier, but we won't grade you on it. There are many different ways to implement this.

Your decision trees ideally should have a well-encapsulated interface like this:

```
classifier = DecisionTree(params)
classifier.train(train_data, train_labels)
predictions = classifier.predict(test_data)
```

where `train_data` and `test_data` are 2D matrices (rows are data, columns are features).

A decision tree (or **DecisionTree**) is a binary tree composed of **Nodes**. You first initialize it with the necessary parameters (which depend on what techniques you implement). As you train your tree, your tree should create and configure **Nodes** to use for classification and store these nodes internally. Your **DecisionTree** will store the root node of the resulting tree so you can use it in classification.

Each **Node** has left and right pointers to its children, which are also nodes, though some (like leaf nodes) won't have any children. Each node has a split rule that, during classification, tells you when you should continue traversing to the left or to the right child of the node. Leaf nodes, instead of containing a split rule, should simply contain a label of what class to classify a data point as. Leaf nodes can either be a special configuration of regular **Nodes** or an entirely different class.

Node fields:

- `split_rule`: A length 2 tuple that details what feature to split on at a node, as well as the threshold value at which you should split. The former can be encoded as an integer index into your data point's feature vector.
- `left`: The left child of the current node.
- `right`: The right child of the current node.
- `label`: If this field is set, the **Node** is a leaf node, and the field contains the label with which you should classify a data point as, assuming you reached this node during your classification tree traversal. Typically, the label is the mode of the labels of the training data points arriving at this node.

DecisionTree methods:

- `entropy(labels)`: A method that takes in the labels of data stored at a node and compute the entropy for the distribution of the labels.
- `information_gain(features, labels, threshold)`: A method that takes in some feature of the data, the labels and a threshold, and compute the information gain of a split using the threshold.
- `entropy(label)`: A method that takes in the labels of data stored at a node and compute the entropy (or Gini impurity).
- `purification(features, labels, threshold)`: A method that takes in some feature of the data, the labels and a threshold, and compute the drop in entropy (or Gini impurity) of a split using the threshold.
- `segmenter(data, labels)`: A method that takes in data and labels. When called, it finds the best split rule for a **Node** using the entropy measure and input data. There are many different types of segmenters you might implement, each with a different method of choosing a threshold. The usual method is exhaustively trying lots of different threshold values from the data and choosing the combination of split feature and threshold with the lowest entropy value. The final split rule uses the split feature with the lowest entropy value and the threshold chosen by the segmenter. *Be careful how you implement this method!* Your classifier might train very slowly if you implement this poorly.

- `train(data, labels)`: Grows a decision tree by constructing nodes. Using the entropy and segmenter methods, it attempts to find a configuration of nodes that best splits the input data. This function figures out the split rules that each node should have and figures out when to stop growing the tree and insert a leaf node. There are many ways to implement this, but eventually your DecisionTree should store the root node of the resulting tree so you can use the tree for classification later on. Since the height of your DecisionTree shouldn't be astronomically large (you may want to cap the height—if you do, the max height would be a hyperparameter), this method is best implemented recursively.

- `predict(data)`: Given a data point, traverse the tree to find the best label to classify the data point as. Start at the root node you stored and evaluate split rules at each node as you traverse until you reach a leaf node, then choose that leaf node's label as your output label.

Random forests can be implemented without code duplication by storing groups of decision trees. You will have to train each tree on different subsets of the data (data bagging) and train nodes in each tree on different subsets of features (attribute bagging). Most of this functionality should be handled by a random forest class, except attribute bagging, which may need to be implemented in the decision tree class. Hopefully, the spec above gives you a good jumping-off point as you start to implement your decision trees. Again, it's highly recommended to think through design before coding.

Happy hacking!

B Submission Instructions

Please submit

- a PDF write-up containing your *answers, plots, and code* to Gradescope;
- a .zip file of your *code* and a README explaining how to run your code to Gradescope; and
- your two CSV files of predictions to Kaggle.