CS 189 Introduction to Machine Learning
Spring 2021 Jonathan Shewchuk

HW5

Due: Wednesday, March 31 at 11:59 pm

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: `https://www.kaggle.com/c/spring21-cs189-hw5-spam/overview`
- Titanic: `https://www.kaggle.com/c/spring21-cs189-hw5-titanic/overview`

Write-up: Submit your solution in **PDF** format to "Homework 5 Write-Up" on Gradescope.

- State your name, and if you have discussed this homework with anyone (other than GSIs), list the names *of them all*.
- Begin the solution for each question in a new page. Do not put content for different questions in the same page. You may use multiple pages for a question if required.
- If you include figures, graphs or tables for a question, any explanations should accompany them in *the same page*. Do NOT put these in an appendix!
- **Only PDF uploads to Gradescope will be accepted**. You may use LaTeX or Word to typeset your solution or scan a neatly handwritten solution to produce the PDF.
- **Replicate all your code in an appendix**. Begin code for each coding question in a fresh page. Do not put code from multiple questions in the same page. When you upload this PDF on Gradescope, *make sure* that you assign the relevant pages of your code from appendix to correct questions.

Code: Additionally, submit all your code as a ZIP to "Homework 5 Code" on Gradescope.

- **Set a seed for all pseudo-random numbers generated in your code.** This ensures your results are replicated when readers run your code.
- Include a README with your name, student ID, the values of the random seed (above) you used, and any instructions for compilation.
- Do NOT provide any data files, but supply instructions on how to add data to your code.
- Code requiring exorbitant memory or execution time won't be considered.
- Code submitted here must match that in the PDF Write-up, and produce the *exact* output submitted to Kaggle. Inconsistent or incomplete code won't be accepted.

1 Honor Code

Declare and sign the following statement:

"I certify that all solutions in this document are entirely my own and that I have not looked at anyone else's solution. I have given credit to all external sources I consulted."

While discussions are encouraged, *everything* in your solution must be your (and only your) creation. Furthermore, all external material (i.e., *anything* outside lectures and assigned readings, including figures and pictures) should be cited properly. We wish to remind you that consequences of academic misconduct are *particularly severe*!

Solution: [**RUBRIC**: (+1 point) if declared and signed.]

2 Random Forest Motivation

Ensemble learning is a general technique to combat overfitting, by combining the predictions of many varied models into a single prediction based on their average or majority vote.

(a) **The motivation of averaging.** Consider a set of uncorrelated random variables $\{Y_i\}_{i=1}^n$ with mean μ and variance σ^2. Calculate the expectation and variance of their average. (In the context of ensemble methods, these Y_i are analogous to the prediction made by classifier i.)

Solution: The average of the Y_i's has the same expectation as each individual Y_i:

$$\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n} Y_i\right] = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}[Y_i] = \frac{1}{n}\cdot n\cdot\mu = \mu,$$

but less variance than each of the individual Y_i's:

$$\mathrm{Var}\left(\frac{1}{n}\sum_{i=1}^{n} Y_i\right) = \left(\frac{1}{n}\right)^2\sum_{i=1}^{n}\mathrm{Var}(Y_i) = \frac{1}{n^2}\cdot n\sigma^2 = \frac{\sigma^2}{n}.$$

(b) **Ensemble Learning – Bagging.** In lecture, we covered bagging (Bootstrap AGGregatING). Bagging is a randomized method for creating many different learners from the same data set.

Given a training set of size n, generate T random subsamples, each of size n', by sampling with replacement. Some points may be chosen multiple times, while some may not be chosen at all. If $n' = n$, around 63% are chosen, and the remaining 37% are called out-of-bag (OOB) samples.

(a) Why 63%?

Solution: Each sample has probability $(1 - 1/n)^n$ of not being selected. For large n, $(1 - 1/n)^n \approx \lim_{n\to\infty}(1 - 1/n)^n = 1/e \doteq 0.368$

(b) If we use bagging to train our model, How should we choose the hyperparameter T? Recall, T is the number of subsamples, and typically, a few dozen to several thousand trees are used, depending on the size and nature of the training set.

Solution: An optimal number of subsamples T can be found with validation. Alternatively, we can observe the OOB error.

(c) In part (a), we see that averaging reduces variance for uncorrelated classifiers. Real-world prediction will of course not be completely uncorrelated, but reducing correlation among decision trees will generally reduce the final variance. Reconsider a set of correlated random variables $\{Z_i\}_{i=1}^n$. Suppose $\forall i \neq j$, $\text{Corr}(Z_i, Z_j) = \rho$. Calculate the variance of their average.

Solution:

$$\text{Var}\left(\frac{1}{n}\sum_{i=1}^{n} Z_i\right) = \frac{1}{n^2}\text{Var}\left(\sum_{i=1}^{n} Z_i\right) = \frac{1}{n^2}\left(\sum_{i=1}^{n}\text{Var}(Z_i) + 2\sum_{1\leq i<j\leq n}\text{Cov}(Z_i, Z_j)\right)$$

$$= \frac{\sigma^2}{n} + \frac{n(n-1)\sigma^2\rho}{n^2} = \frac{\sigma^2}{n} + \frac{n-1}{n}\rho\sigma^2.$$

We can see that for large n, the first term dominates, which limits the benefit of averaging.

(d) Is a random forest of stumps (trees with a single feature split or height 1) a good idea in general? Does the performance of a random forest of stumps depend much on the number of trees? Think about the bias of each individual tree and the bias of the average of all these random stumps.

Solution: Stumps generally have high bias; they are very simple models that cannot fit to anything with reasonable complexity. If we treat $\{Z_i\}$ as the set of possibly correlated predictions the stumps produce,

$$\mathbb{E}\left(\frac{1}{n}\sum_{i=1}^{n} Z_i\right) = \mu_z.$$

This tells us if each stump has high bias, averaging the predictions of all stumps will not reduce this bias. Thus a random forest of stumps is generally a bad idea no matter how many stumps we have.

3 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.

3.1 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.**

Solution: The sample solution codes are distributed separately.

3.2 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.**

Solution: The sample solution codes are distributed separately.

3.3 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?

Solution:

(a) Some data are missing class labels or are blank. For those, we simply remove that data. Some features are not numerical values, which is needed for thresholding, such as gender or the Port of Embarkation. For these, we hash the feature value to convert it into one hot vectors. For example, if we are dealing with a feature "make of a car", and the categories are "Toyota", "Honda", "GM", "BMW", etc. We will split it into multiple binary features "is_Toyota", "is_Honda", "is_GM", "is_BMW", etc. Some

data are missing features. Depending on the dataset, there are multiple ways to do that. One big advantage of decision trees is missing features need not be explicitly handles; we can treat missing values as a value to possibly split on. In this homework, we fill it using the mode value of that feature. We use the mode instead of the mean or median, as this makes more sense for categorical features such as gender or cabin type, which are not ordered. For such a small dataset, simply removing data with missing features is not an option.

(b) Overly deep decision tree can represent arbitrary decision boundary, but this incurs overfitting. Here are several candidates when considering stopping criteria:

- Limited depth: don't split if the node is beyond some fixed depth in the tree
- Information gain criterion: don't split if the gained information/purity is sufficiently close to zero

(c) The implementation of Random Forests can be based on that of the Decision Trees. The only things we need to modify is what data points we should use to train the model and what number of features we need to consider for each split.

(d) There are different ways to expedite efficiency of the codes. One way is to sufficiently vectorize them by using numpy functions. We can also reduce the number of columns added by imposing a minimum number of occurrences of a categorical value: if certain categorical value appears a number of times below a threshold, we just discard it without creating a new column for our data matrix.

(e) An answer of "no" is acceptable here.

3.4 Performance Evaluation

For each of the 2 datasets, train both a decision tree and random forest and report your training and validation accuracies. You should be reporting 8 numbers (2 datasets $\times$ 2 classifiers $\times$ 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.

Solution: For the spam dataset, our decision tree with a depth of 5 achieves 80.5% accuracy on the training set and 78.6% on the validation set; and our random forest achieves 80.4% accuracy on the training set and 78.6% on the validation set.

For the Titanic dataset, our decision tree with a depth of 5 achieves 81.6% accuracy on the training set and 80.0% on the validation set; and our random forest achieves 82.4% accuracy on the training set and 81.0% on the validation set.

3.5 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.

Solution: A sample path for ham/spam in our sample implementation is

(a) (“exclamation”) ≥ 1

(b) (“meter”) ≥ 1

(c) Therefore this email was ham.

(a) (“exclamation”) ≥ 0

(b) (“meter”) ≤ 0

(c) (“ampersand”) ≤ 0

(d) (“money”) ≥ 1

(e) (“exclamation”) ≥ 13

(f) Therefore this email was spam.

3. For random forests, find and state the most common splits made at the root node of the trees. For example:

   (a) (“viagra”) ≥ 3 (20 trees)

   (b) (“thanks”) < 4 (15 trees)

   (c) (“nigeria”) ≥ 1 (5 trees)

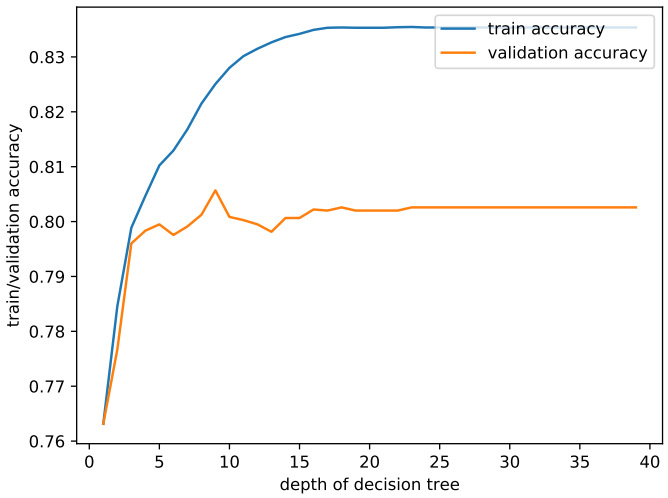
   Solution: For our sample implementation, the top three most common splits are

   (a) (“exclamation”) ≤ 0 (16 trees)

   (b) (“meter”) ≤ 0 (15 trees)

   (c) (“volumes”) ≤ 0 (14 trees)

4. 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.



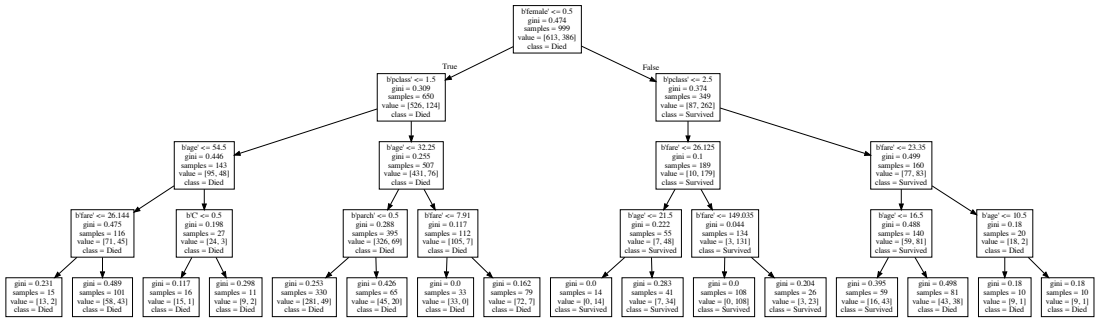
Solution:

With our sample solution, the best depth is 9. A decision tree can model any arbitrary decision boundary if no two points from different classes overlap. As we increase the depth of the tree, the model becomes more expressive and can model more complex boundary. As a result, the validation accuracy increases from more model capacity. However, as we increase the depth the variance of the model also increases. When we pass the optimal depth the accuracy decreases due to overfitting to the training data.

3.6 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.

Solution: Here is the structure for decision tree from our sample implementation



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.