CS189: Introduction to Machine Learning

Homework 5

Due: 11:59 p.m. April 9, Thursday, 2015

Decision Trees for Classification

In this homework, you will implement decision trees (and its extensions) for classification on the spam data we used in HW1.

In lectures, you were given a basic introduction to decision trees and how such trees are trained. You were also introduced to random forests and boosting algorithms (AdaBoost).

This homework is meant to test your ability to implement decision trees and averaging (random forests) as explained in class as well as to research different decision tree techniques from resources on the internet/books/blogs. The readings attached with this homework (found in the readings/ folder) might be a useful point to start off! The rules for the homework are as follows:

1. To get full credit, you must implement **Decision Trees** (without pruning) and **Random Forests**.
2. You are **NOT** allowed to use any off the shelf decision tree/random forest implementation for the homework. You can use external libraries for data preprocessing, faster computation etc. as long as you cite them in your report. You can use any programming language you wish to as long as we can run it with minimal effort.
3. This is an individual assignment. You may collaborate with other students, but everyone must submit their own code.
4. As deliverables for this homework, you need to submit your code, a README and a writeup detailed in the following section. Submissions are through **bCourses** and **Kaggle**.
5. Your report must contain:
   (a) an explanation of the decision tree techniques you implemented (stopping criteria, pruning, dealing with missing attributes, splitting criteria other than

entropy, heuristics for faster training, complex decisions at nodes, cross-validation, Adaboost, bagging etc.).

(b) an explanation of the features (extracted from the data) you may have added or removed. We recommend using bag-of-words features instead of manually choosing words to extract from the data.

(c) a report of the results you obtained. This includes the performance of your decision tree and random forest on your test set and your best kaggle score.

(d) for a data point/data points of your choosing, show what splits (ie which feature and which value of that feature to split on) that your decision tree made to classify it. An example of what this might look like:

   i. feature 5 ("viagra") ≥ 0.5

   ii. feature 2 ("thanks") < 0.5

   iii. feature 3 ("nigeria") ≥ 0.5

(e) for random forests, find the most common splits made at the top level of the trees. For example:

   i. feature 5 ("viagra") ≥ 0.5 (20 trees)

   ii. feature 2 ("thanks") < 0.5 (15 trees)

   iii. feature 3 ("nigeria") ≥ 0.5 (5 trees)

(f) (optional) a visualisation of your decision tree's structure and its split rules. One cool feature of decision trees is that it's much easier to understand what the classifier is doing compared to other algorithms that we use, especially with respect to spam data. Your visualisation will probably look something like this: `http://scikit-learn.org/stable/_images/iris.svg`. You may use external libraries like graphviz to help you do this.

(g) references to all external sources you used (code, articles, papers, books etc.) and anything else you might want to include (analysis of results, performance curves etc.)

Hopefully by the end of this homework everyone will have their own version of decision trees running and can try it on different datasets! Good luck!

1 Appendix

(Approximate) Expected Performance

Using the base feature extractor we wrote and a regular decision tree, we got 11% training error and 20% testing error. You can get much better performance by engineering better features. This is a general ballpark range of what performance you should expect your tree to get, and not a hard requirement.

Spec

This is a moderately complicated coding project, and you will have to build everything from scratch. Unlike 162, we won't make you submit a design doc, but you will have to put in some thought about how to structure your program so that your decision trees don't end up as horrific burning forests of technical debt. As a result, we have provided a rough spec that suggests how you might implement these decision trees. Although your TAs are cool and smart and walk on water, you don't need to follow this spec if you don't want to.

I DO NOT RECOMMEND IMPLEMENTING THIS IN MATLAB. AFAIK, it's quite difficult to code tree data structures in Matlab. Python + NumPy is your best bet here (in large part because you can pass functions as parameters), though you can certainly use other languages as long as you give the instructors a README on how to run your code.

Your decision trees should have an interface similar to that of the SVM implementation you used in HW1. That is, your classifiers' logic should be encapsulated in classes so that the classification flow looks something like this:

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

wbere `train_data` and `test_data` are 2D matrices with each row representing a different data point and each column representing a different feature, and where `train_labels` and `test_labels` are 1D vectors with the true classification labels of each row in `train_data` and `test_data`, respectively.

A regular decision tree (or **DTree**) is a binary tree composed of **Nodes**. You first initialise your tree with parameters. When you train your tree, your tree should basically figure out what configuration of **Nodes** to use for classification and then 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. Here, we'll detail the latter case and define a **LeafNode** class.

Node params:

- `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 at. 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 left child of the current node.

LeafNode params:

- `label` The label 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 data points "at" this node. More on that later.

Here's what the **DTree** should look like. Your DTree should store a **Node** configuration that makes up your decision tree. This is created when you call `train`, so you shouldn't have a **Node** configuration upon initialisation. As in reality, the secret sauce to your tree is how you grow it. Most of your time will be spent writing the `train` method.

DTree params:

- `depth`: The maximum depth that your decision tree can be.
- `impurity(left_label_hist, right_label_hist)`: A function passed as a parameter. It takes in the result of a split: two histograms that count the number/type of labels on the "left" and "right" side of that split, then calculates and outputs a scalar value representing the impurity (ie the "badness") of the specified split on the input data.
- `segmentor(data, labels, impurity)`: A function passed as a parameter. It takes in data, labels, and an impurity function. When called, it finds the best split rule for a **Node** using its impurity measure and input data. Each type of segmentor has a different method of choosing a threshold. This can be setting the threshold to the mean of means or median of a feature's values in the data, for example, or you can exhaustively try lots of different threshold values (either arbitrarily or from the data) and choose the combination of split feature and threshold with the lowest impurity value. The final split rule uses the split feature with the lowest impurity value and the threshold (ie the split value) chosen by the segmentor. Be careful how you implement this method! Your classifier might train very slowly if you implement this badly.
- Anything else you'd like to add!

DTree methods:

- `train(data, labels)`: Grows a decision tree by constructing nodes. Using the impurity and segmentor provided upon initialisation, 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's a lot of ways to implement this, but eventually, your DTree should store the root node of the resulting tree you constructed so you can use the tree for classification later on. Since the height of your DTrees shouldn't be astronomically large (probably 150 at most), this method is best implemented recursively. You might need to use helper functions too.
- `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.

This DTree spec allows you to easily write different segmentor and impurity functions and swap them into your tree implementation. However, the train function is general enough that you can try implementing your own training techniques (pruning, stopping criteria, etc.) to find the combination that achieves the best performance.

Note that random forests can be implemented without too much code duplication by storing groups of regular decision trees instead of tree nodes. Additionally, the training logic of random forests is quite different from regular decision trees. You will have to train each tree on different subsets of the data (bagging) and train each tree on different subsets of features (attribute bagging), but differences in functionality should ideally be encapsulated in the `train` and `predict` functions. Hopefully, the spec above gives you a good jumping-off point as you start to implement your decision trees.

Happy hacking!