CS189: Introduction to Machine Learning

Homework 5

Due: 11:59 p.m. November 9, Monday, 2015

Homework party: November 5, 2-4pm (Wozniak Lounge).

Submission: **bCourses and Kaggle**

Submission Instructions

On bCourses, submit **2 separate files: a pdf and a zip of your code.**

1. A pdf writeup with answers to all the questions. Include at the end of the pdf a copy of all your code, including custom feature code.
2. A zip archive containing your code for each problem, and a README with instructions on how to run your code.

Do not submit a combined zip of your pdf and code. You will also need to submit test predictions to Kaggle.

Decision Trees for Classification

In this homework, you will implement decision trees and random forests for classification on the spam data we used in HW1. You will also train on a census income dataset to predict whether or not a person makes over 50k in income.

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. Feel free to research different decision tree techniques online. The optional readings attached with this homework (found in the readings/ folder) might be a useful point to start off! You do not have to implement boosting, although it might be a good idea if you want to win Kaggle.

1. You must implement **Decision Trees** and **Random Forests**, then train each on the spam data, and use each to separately classify the spam data. Repeat for the

census income classification. See the report below.

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 (in fact, we recommend it). You can use any programming language you wish to as long as we can read and run your code with minimal effort.

3. Code and report requirements:

   (a) For spam, 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 should implement any custom feature extraction code in `featurize.py`, which will save your features to a `.mat` file.

   (b) A report of the results you obtained. List the performance of your decision tree and random forest on your validation set. Also list your single best Kaggle score with any method (state which method you used).

   (c) For your decision tree, and for a data point of your choosing, state what splits (i.e. 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. ("viagra") ≥ 2

      ii. ("thanks") < 1

      iii. ("nigeria") ≥ 3

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

      i. ("viagra") ≥ 3 (20 trees)

      ii. ("thanks") < 4 (15 trees)

      iii. ("nigeria") ≥ 1 (5 trees)

4. (a), (b), (c), (d). Repeat the same parts for the census dataset. You will need to do your own feature processing. In part (a), you must report what you did to handle categorical variables and missing features. See the Appendix for detailed instructions.

5. (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 random forest techniques you used. If the decision trees you used inside your random forest were different than your standalone decision tree implementation, explain how.

1 Appendix

Data Processing for Census

You will have to process and transform the data yourself into a form suitable for consumption by your decision tree / random forest code.

Training data is in `data.csv`. Test data for Kaggle is in `test_data.csv`. A sample Kaggle submission is at `sample_submission.csv`.

Look at `adult.names` for a description of all the features in the data. Note that the label values $= \{0, 1\}$, where 1 corresponds to a person with an annual income $\geq$ \$50k. We want to predict whether or not someone has an annual income of $\geq$ \$50k.

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. Many features in this dataset represent types/categories. There are 2 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. One name for this is "one-hot encoding."

   (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 maximizes 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).

   To make coding easier, here is a Python list of the field names for the categorical variables:

   ```
   [workclass, education, marital-status,
     occupation, relationship, race, sex, native-country]
   ```

2. Missing values. Some data points are missing features. In the `csv` files, this is represented by the value '`?`'. You have three approaches:

(a) (Easiest) If a data point is missing some features, remove it from the data matrix.

(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).

(c) (Hard, but most 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 HIGHLY recommended you use the following classes to write, read, and process data:

```
csv.DictReader
sklearn.feature_extraction.DictVectorizer (binarizing categorical variables)
    (There's also sklearn.preprocessingOneHotEncoder, but it's much less clean)
sklearn.preprocessing.LabelEncoder
    (if you choose to discretize but not binarize 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.

For MATLAB:

MATLAB has identical facilities for data processing that you should use.

(Approximate) Expected Performance

For spam, using the base features and a regular decision tree, we got 20% testing error. With a random forest on the census data, we got around 15% testing error. You can get much better performance. This is a general ballpark range of what to expect; it is not a requirement.

Suggested Baseline Spec

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)
```

wbere `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 (depends 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 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.
- `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:

- `impurity(left_label_hist, right_label_hist)`: A method that takes in the result of a split: two histograms (a histogram is a mapping from labels to their frequencies) 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 (i.e. the "badness") of the specified split on the input data.
- `segmentor(data, labels)`: A method that takes in data and labels. When called, it finds the best split rule for a **Node** using the impurity measure and input data.

There are many different types of segmentors you might implement, each with 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 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 chosen by the segmentor. *Be careful how you implement this method!* Your classifier might train very slowly if you implement this badly.

- `train(data, labels)`: Grows a decision tree by constructing nodes. Using the impurity and segmentor methods, 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 (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.

Happy hacking!