CS189 Fall 2016
Introduction to Machine Learning

Homework 5

Due: 12:00 pm noon on Thursday, November 17, 2016

Submission Instructions

You will submit your PDF writeup and code to **Gradescope**. There will also be two **Kaggle** competitions.

In your submission to **Gradescope**, include separately:

1. A PDF writeup with answers to all the questions. Include at the end of the pdf a copy of all your code.
2. A zip file of all your code.

Submit to **Kaggle**:

3. A csv file with your best predictions for the census dataset.
4. A csv file with your best predictions for the spam dataset.

Note: The Kaggle invite links will be posted on Piazza.

Problem 1: Polynomial and Gaussian Kernels

1. First, we'll create an artificial two-class classification problem. Let the elements in class 1 ($y = 1$) be sampled so that

$$(x_1, x_2) = (8cos(\theta) + w_1, 8sin(\theta) + w_2)$$

where θ is uniformly distributed in $[0, 2\pi]$ and w_1 and w_2 are independent normally distributed random variables with mean zero and variance 1. For the second class ($y = -1$), let the points be sampled as

$$(x_1, x_2) = (v_1, v_2)$$

where v_1 and v_2 are independent normally distributed random variables with mean zero and variance 1. Generate 100 points for each class and construct a 200×2 dimensional data matrix X containing these points. Construct the corresponding 200×1 dimensional label vector y. Plot the points using different colors to represent the two classes.

2. Using regularized least-squares classification, build a classifier

$$\min_\alpha \|K\alpha - y\|^2 + \lambda \alpha^T K \alpha$$

using the polynomial kernel $k(x, z) = (1 + x^T z)^2$. State the accuracy on the training data. Plot the training data and the corresponding decision boundary for $\lambda = 1e - 6$.

3. Using regularized least-squares classification, build a classifier using the Gaussian kernel, $k(x, z) = \exp(-\gamma \|x - z\|^2)$. State the accuracy on the training data. Plot the points of X and the contours of the fit function

$$f(x) = \sum_{i=1}^{n} \alpha_i k(x_i, x)$$

for $\gamma = 10, 0.1, 0.001$ and $\lambda = 1e - 4$.

Problem 2: Decision Trees and Random Forests

You will implement decision trees and random forests for classification on 2 datasets: 1) spam data, and 2) a census income dataset to predict whether or not a person makes over 50k in income.

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 Appendix for more information.

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).

3. Code and report requirements for **spam** dataset:

   (a) 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.

   (b) Provide a report of the results you obtained. List the accuracy of your decision tree and random forest on your training and validation sets. 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. Code and report requirements for **census** dataset:

   Repeat all parts of 3). 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) Give 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) Give 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.

Appendix

Data Processing for Spam

There are 5172 training examples and 5857 items in the test set. A data matrix produced using basic feature extractors in `featurize.py` is provided to you in `spam_data.mat`. You may optionally process the raw data differently using the `spam`, `ham`, and `test` folders to improve your accuracy. If you do, please explain what you did in your report.

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 `train_data.csv`. Test data for Kaggle is in `test_data.csv`.

Look at the first line of `train_data.csv` for a description of all the features in the data. Note that the last column has the binary label values, where 1 corresponds to a person with an annual income $\geq$ \$50k. We wish to predict the label.

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 many possibilities 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 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).

   Here is a 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 (**this is good as a start but you must submit something more sophisticated**).

   (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 more powerful). Use kNN to impute feature values based on the nearest neighbors of a data point. You will need to define in your distance metric what the distance to a missing value is

   (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).

It is HIGHLY 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.

(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 label values to their frequencies) that count the frequencies of labels on the "left" and "right"

side of that split. The method calculates and outputs a scalar value representing the impurity (i.e. the "badness") of the specified split on the input data.

- `segmenter(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 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 impurity value. The final split rule uses the split feature with the lowest impurity 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 badly.
- `train(data, labels)`: Grows a decision tree by constructing nodes. Using the impurity and segmenter 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 (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!