CS 189:289A Introduction to Spring 2017 Machine Learning

Homework 5: Decision Trees Due Mon., March 27

Instructions

- We prefer that you typeset your answers using LaTeX or other word processing software. Neatly handwritten and scanned solutions will also be accepted.
- Please make sure to start **each question on a new page**, as grading (with Gradescope) is much easier that way!
- Deliverables. Submit a **PDF of your writeup** to the Homework 5 assignment on Gradescope. Include your code in your writeup in the appropriate sections. Submit your **code zip** and a README to the Homework 5 Code assignment on Gradescope. Finally, submit **your predictions** for the test sets to Kaggle. Be sure to include your Kaggle display name and score in your writeup.
- Due **Monday, March 27, 2017 at 11:59 PM.**

Decision Trees for Classification

In this homework, you will implement decision trees and random forests for classification on 3 datasets: 1) the spam dataset, 2) a census income dataset to predict whether or not a person makes over 50k in income, and 3) a Titanic dataset to predict Titanic survivors. The data is available on Piazza.

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.

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 (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 criteria, 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. 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.**

3. Describe implementation details. We aren't looking for an essay; 1-2 sentences per question is enough.

   (a) How did you deal with categorical features and missing values?

   (b) What was your stopping criteria?

   (c) Did did do anything special to speed up training?

   (d) How did you implement random forests?

   (e) Anything else cool you implemented?

4. Performance evaluation. For each of the 3 datasets, train a decision tree and random forest and report your training and validation accuracies. You should be reporting 12 numbers (3 datasets $\times$ 2 classifiers $\times$ 2 data splits). In addition, for each of the 3 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 3 numbers.

5. Writeup requirements for the 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. You can implement any custom feature extraction code in `featurize.py`, which will save your features to a `.mat` file.

   (b) 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. Example of what this might look like:

      i. ("viagra") ≥ 2

      ii. ("thanks") < 1

      iii. ("nigeria") ≥ 3

iv. Therefore this email was spam.

i. ("budget") ≥ 2

ii. ("spreadsheet") ≥ 1

iii. Therefore this email was ham.

(c) 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)

6. Writeup requirements for the census dataset:

(a) Do 4a, but for the census dataset

(b) Do 4b, but for the census dataset

(c) Do 4c, but for the census dataset

(d) 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. 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.

7. Writeup requirements for the Titanic dataset:

(a) 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.

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 `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).

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.

Data Processing for Titanic

Here's a brief overview of the fields in the Titanic dataset. Again, you will need to preprocess the dataset.

1. pclass - Measure of socioeconomic status. 1 is upper, 2 is middle, 3 is lower.
2. age - Fractional if less than 1.
3. sex - Male/female
4. sibsp - Number of siblings/spouses aboard the Titanic
5. parch - Number of parents/children aboard the Titanic
6. ticket - Ticket number
7. fare - Fare.
8. cabin - Cabin number.
9. embarked - Port of Embarkation (C = Cherbourg, Q = Queenstown, S = Southampton)

(Approximate) Expected Performance

For spam, using the base features and a regular decision tree, we got 76% testing accuracy. With a random forest on the census data, we got around 78% testing accuracy. We get around 80% 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)
```

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 poorly.
- `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. Again, it's highly recommended to think through design before coding.

Happy hacking!

2 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 three CSV files of predictions to Kaggle.