assignment_1a

September 17, 2018

1 Part 1a: Classification with Decision Trees

DUE September 17th 2018

1.1 Introduction

The code for this project consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore.

1.1.1 Files You'll Edit

`assignment_1b.ipynb`: Will be your edited copy of this notebook pertaining to part 1a of the assignment.
`titanic-features.py`: This contains some functions to help you apply a decision tree to the Titanic dataset.

1.1.2 Files you might want to look at

`binary.py`: Our generic interface for binary classifiers (actually works for regression and other types of classification, too).
`datasets.py`: Where a handful of test data sets are stored.
`util.py`: A handful of useful utility functions: these will undoubtedly be helpful to you, so take a look!
`runClassifier.py`: A few wrappers for doing useful things with classifiers, like training them, generating learning curves, etc.
`mlGraphics.py`: A few useful plotting commands
`data/*`: all of the datasets we'll use.

1.1.3 What to Submit

You will hand in all of the python files listed above under "Files you'll edit". You will also have to answer the written questions in this notebook denoted **Q#:** in the corresponding cells denoted with **A#:**.

Autograding Your code will be autograded for technical correctness. Please **do not** change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder's output – will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

```
In [1]: # Jupyter magic!
        %matplotlib inline
```

1.2 0. Warming up to Classifiers (10%)

Let's begin our foray into classification by looking at some very simple classifiers. There are three classifiers in `dumbClassifiers.py`, one is implemented for you, the other two you will need to fill in appropriately.

The already implemented one is `AlwaysPredictOne`, a classifier that (as its name suggest) always predicts the positive class. We're going to use the `TennisData` dataset from `datasets.py` as a running example. So let's start up python and see how well this classifier does on this data. You should begin by importing `util`, `datasets`, `binary` and `dumbClassifiers`.

```
In [2]: import dumbClassifiers, datasets
        import numpy as np
```

```
In [3]: h = dumbClassifiers.AlwaysPredictOne({})
        h.train(datasets.TennisData.X, datasets.TennisData.Y)
        h.predictAll(datasets.TennisData.X)
```

```
Out[3]: array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Indeed, it looks like it's always predicting one!

Now, let's compare these predictions to the truth. Here's a very clever way to compute accuracies

```
In [4]: np.mean((datasets.TennisData.Y > 0) == (h.predictAll(datasets.TennisData.X) > 0))
```

```
Out[4]: 0.6428571428571429
```

Q1: Why is this computation equivalent to computing classification accuracy?

A1: The second half of the equivalence will always be true. The first half will be true when play tennis is true i.e. can play tennis. This equation will come out to be the average of when to play tennis over the sum of time.

That's training accuracy; let's check test accuracy:

```
In [5]: np.mean((datasets.TennisData.Yte > 0) == (h.predictAll(datasets.TennisData.Xte) > 0))
```

```
Out[5]: 0.5
```

Okay, so it does pretty badly. That's not surprising, it's really not learning anything!!!

Now, let's use some of the built-in functionality to help do some of the grunt work for us. We'll need to import `runClassifier`.

```
In [6]: import runClassifier
```

```
In [7]: runClassifier.trainTestSet(h, datasets.TennisData)
```

```
Training accuracy 0.642857, test accuracy 0.5
```

Very convenient!

Now, your first implementation task will be to implement the missing functionality in `AlwaysPredictMostFrequent`. This actually will "learn" something simple. Upon receiving training data, it will simply remember whether +1 is more common or -1 is more common. It will then always predict this label for future data. **Once you've implemented this**, you can test it:

```
In [8]: h = dumbClassifiers.AlwaysPredictMostFrequent({})
        runClassifier.trainTestSet(h, datasets.TennisData)
```

```
Training accuracy 0.642857, test accuracy 0.5
```

It turns out that its behavior is exactly the same as `AlwaysPredictOne` since +1 is most common in the TennisData training set. If we change to a different dataset: `SentimentData` is the data you've seen before, now Python-ified, AlwaysPredictMostFrequent will still predict the same as AlwaysPredictOne since +1 is also the most common in the SentimentData dataset.

```
In [9]: runClassifier.trainTestSet(dumbClassifiers.AlwaysPredictOne({}), datasets.SentimentDat
```

```
Training accuracy 0.504167, test accuracy 0.5025
```

```
In [10]: runClassifier.trainTestSet(dumbClassifiers.AlwaysPredictMostFrequent({}), datasets.Se
```

```
Training accuracy 0.504167, test accuracy 0.5025
```

The last dumb classifier we'll implement is `FirstFeatureClassifier`. This actually does something slightly non-trivial. It looks at the first feature (i.e., `X[0]`) and uses this to make a prediction. Based on the training data, it figures out what is the most common class for the case when `X[0] > 0` and the most common class for the case when `X[0] <= 0`. Upon receiving a test point, it checks the value of `X[0]` and returns the corresponding class. **Once you've implemented this**, you can check it's performance:

```
In [11]: ################################################################################
         # TODO: Implement FirstFeatureClassifier                                      #
         ################################################################################
         runClassifier.trainTestSet(dumbClassifiers.FirstFeatureClassifier({}), datasets.Tenni
```

```
Training accuracy 0.714286, test accuracy 0.666667
```

```
In [12]: runClassifier.trainTestSet(dumbClassifiers.FirstFeatureClassifier({}), datasets.Senti
```

```
Training accuracy 0.504167, test accuracy 0.5025
```

1.3 1. Decision Trees

Our next task is to implement a decision tree classifier. There is stub code in `dt.py` that you should edit. Decision trees are stored as simple data structures. Each node in the tree has a `.isLeaf` boolean that tells us if this node is a leaf (as opposed to an internal node). Leaf nodes have a `.label` field that says what class to return at this leaf. Internal nodes have: a `.feature` value that tells us what feature to split on; a `.left` *tree* that tells us what to do when the feature value is *less than 0.5*; and a `.right` *tree* that tells us what to do when the feature value is *at least 0.5*. To get a sense of how the data structure works, look at the `displayTree` function that prints out a tree.

Your first task is to implement the training procedure for decision trees. We've provided a fair amount of the code, which should help you guard against corner cases. (Hint: take a look at `util.py` for some useful functions for implementing training. **Once you've implemented the training function**, we can test it on simple data:

```
In [13]: import dt

In [14]: h = dt.DT({'maxDepth': 1})
         h

Out[14]: Leaf 1

In [15]: ###############################################################################
         # TODO: Implement dt.train(...)                                               #
         ###############################################################################
         h.train(datasets.TennisData.X, datasets.TennisData.Y)
         h

Out[15]: Branch 6
           Leaf 1.0
           Leaf -1.0
```

This is for a simple depth-one decision tree (aka a decision stump). If we let it get deeper, we get things like:

```
In [16]: h = dt.DT({'maxDepth': 2})
         h.train(datasets.TennisData.X, datasets.TennisData.Y)
         h

Out[16]: Branch 6
           Branch 7
             Leaf 1.0
             Leaf 1.0
           Branch 1
             Leaf -1.0
             Leaf 1.0

In [17]: h = dt.DT({'maxDepth': 5})
         h.train(datasets.TennisData.X, datasets.TennisData.Y)
         h
```

```
Out[17]: Branch 6
           Branch 7
             Leaf 1.0
             Branch 2
               Leaf 1.0
               Leaf -1.0
           Branch 1
             Branch 5
               Branch 4
                 Leaf -1.0
                 Branch 3
                   Leaf -1.0
                   Leaf 1.0
               Leaf 1.0
             Leaf 1.0
```

We can do something similar on the sentiment data (this will take a bit longer—it takes about 10 seconds on my laptop):

```
In [18]: h = dt.DT({'maxDepth': 2})
         h.train(datasets.SentimentData.X, datasets.SentimentData.Y)
         h
```

```
Out[18]: Branch 626
           Branch 683
             Leaf 1.0
             Leaf -1.0
           Branch 1139
             Leaf -1.0
             Leaf 1.0
```

```
In [19]: h = dt.DT({'maxDepth': 2})
         h.train(datasets.SentimentData.X, datasets.SentimentData.Y)
         h
```

```
Out[19]: Branch 626
           Branch 683
             Leaf 1.0
             Leaf -1.0
           Branch 1139
             Leaf -1.0
             Leaf 1.0
```

The problem here is that words have been converted into numeric ids for features. We can look them up (your results here might be different due to hashing):

```
In [20]: print(2428, datasets.SentimentData.words[2428])
         print(3842, datasets.SentimentData.words[3842])
         print(3892, datasets.SentimentData.words[3892])
```

```
2428 deliver
3842 reportedly
3892 wraps
```

Based on this, we can rewrite the tree (by hand) as:

```
Branch 'bad'
  Branch 'worst'
    Leaf -1.0
    Leaf 1.0
  Branch 'sequence'
    Leaf -1.0
    Leaf 1.0
```

Now, you should **go implement prediction**. This should be easier than training! We can test by (this takes about a minute for me):

```
In [21]: ################################################################################
         # TODO: Implement dt.predict(...)                                              #
         ################################################################################
         runClassifier.trainTestSet(dt.DT({'maxDepth': 1}), datasets.SentimentData)
         runClassifier.trainTestSet(dt.DT({'maxDepth': 3}), datasets.SentimentData)
         runClassifier.trainTestSet(dt.DT({'maxDepth': 5}), datasets.SentimentData)
```

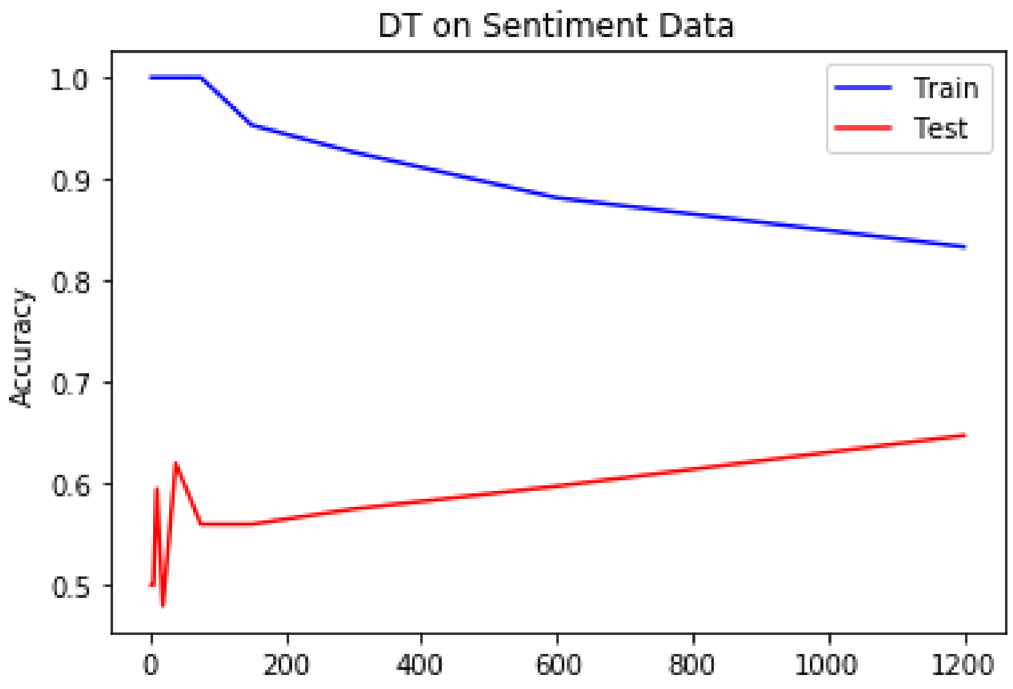
```
Training accuracy 0.630833, test accuracy 0.595
Training accuracy 0.701667, test accuracy 0.6175
Training accuracy 0.765833, test accuracy 0.625
```

Looks like it does better than the dumb classifiers on training data, as well as on test data! Hopefully we can do even better in the future!

We can use more `runClassifier` functions to generate learning curves and hyperparameter curves:

```
In [22]: curve = runClassifier.learningCurveSet(dt.DT({'maxDepth': 9}), datasets.SentimentData
         runClassifier.plotCurve('DT on Sentiment Data', curve)
```

```
Training classifier on 2 points...
Training accuracy 1, test accuracy 0.5
Training classifier on 3 points...
Training accuracy 1, test accuracy 0.5
Training classifier on 5 points...
Training accuracy 1, test accuracy 0.5
Training classifier on 10 points...
Training accuracy 1, test accuracy 0.595
Training classifier on 19 points...
Training accuracy 1, test accuracy 0.48
Training classifier on 38 points...
Training accuracy 1, test accuracy 0.62
Training classifier on 75 points...
Training accuracy 1, test accuracy 0.56
Training classifier on 150 points...
Training accuracy 0.953333, test accuracy 0.56
Training classifier on 300 points...
Training accuracy 0.926667, test accuracy 0.575
Training classifier on 600 points...
Training accuracy 0.881667, test accuracy 0.5975
Training classifier on 1200 points...
Training accuracy 0.833333, test accuracy 0.6475
```



This plots training and test accuracy as a function of the number of data points (x-axis) used for training and y-axis is accuracy.

Q2: We should see training accuracy (roughly) going down and test accuracy (roughly) going up. Why does training accuracy tend to go *down?*

A2: At a low testing point, there is a higher probability of getting most predictions right as the dataset is more expaned on all cases and only takes few cases into consideration. It is very likely that the decision tree will fail in training when it is trained on more points.

Q3: Why is test accuracy not monotonically increasing?

A3: The test accuracy will not monotonically increase as n increases on the training set, the accuracy of testing will decrease from it's original, as the size of test and train data size increases with which comes uncertainty about the accuracy of the model.

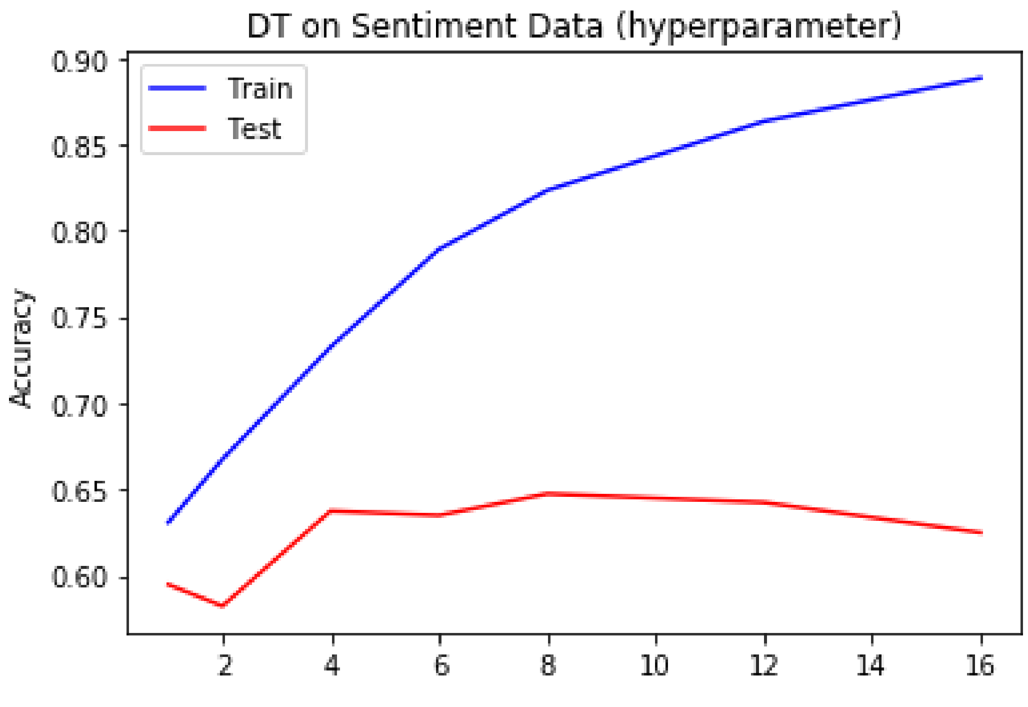
Q4: You should also see jaggedness in the test curve toward the left. Why?

A4: It is jagged in the left as the first few iderations are over small n - 1,2,3,5,10 versus the later where n increases drastically.During the first few iderations there is more jumps in test accuracy which is taking over small increments versus later increments for the later data collected.

We can also generate similar curves by changing the maximum depth hyperparameter:

```
In [23]: curve = runClassifier.hyperparamCurveSet(dt.DT({}), 'maxDepth', [1,2,4,6,8,12,16], da
         runClassifier.plotCurve('DT on Sentiment Data (hyperparameter)', curve)
```

```
Training classifier with maxDepth=1...
Training accuracy 0.630833, test accuracy 0.595
Training classifier with maxDepth=2...
Training accuracy 0.6675, test accuracy 0.5825
Training classifier with maxDepth=4...
Training accuracy 0.7325, test accuracy 0.6375
Training classifier with maxDepth=6...
Training accuracy 0.789167, test accuracy 0.635
Training classifier with maxDepth=8...
Training accuracy 0.823333, test accuracy 0.6475
Training classifier with maxDepth=12...
Training accuracy 0.863333, test accuracy 0.6425
Training classifier with maxDepth=16...
Training accuracy 0.888333, test accuracy 0.625
```



Now, the x-axis is the value of the maximum depth.

Q5: You should see training accuracy monotonically increasing and test accuracy making something like a hill. Which of these is *guaranteed* to happen and which is just something we might expect to happen? Why?

A5: The training accuracy is gauranteed to increase as the depth increases as the data can be better fit into the model as training progresses. But the test accuracy making a hill is something we might expect to happen as the two might input the same result for the data despite being better fit while testing.

assignment_1b

September 17, 2018

1 Part 1b: Predicting survival of Titanic passengers with decision trees

DUE September 17th 2018

1.1 Introduction

The code for this project consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore.

1.1.1 Files You'll Edit

`assignment_1b.ipynb`: Will be your edited copy of this notebook pertaining to part 1a of the assignment
`features.py`: Simple feature engineering function

1.1.2 Files you might want to look at

`binary.py`: Our generic interface for binary classifiers (actually works for regression and other types of classification, too).
`datasets.py`: Where a handful of test data sets are stored.
`util.py`: A handful of useful utility functions: these will undoubtedly be helpful to you, so take a look!
`runClassifier.py`: A few wrappers for doing useful things with classifiers, like training them, generating learning curves, etc.
`mlGraphics.py`: A few useful plotting commands
`data/*`: all of the datasets we'll use.

1.1.3 What to Submit

You will hand in all of the python files listed above under "Files you'll edit". You will also have to answer the written questions in this notebook denoted **Q#:** in the corresponding cells denoted with **A#:**.

Autograding Your code will be autograded for technical correctness. Please **do not** change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder's output – will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

1.2 A quick look at the data

In `data/` you will find the following files: `titanic_train.csv`

\``titanic_test.csv`\`

Let's take a look at the CSV file using the [Pandas] package and import other packages we think we will need.

```
In [1]: import pandas as pd
        import dt
        import features
        import runClassifier
        import pandas as pd
        import numpy as np
        from sklearn.model_selection import train_test_split
```

Pandas lets us take read CSVs easily and allows us to manipulate the data with ease. So lets take a look at the data!

```
In [2]: train_df = pd.read_csv('data/titanic_train.csv')
        train_df.head()
```

```
Out[2]:    PassengerId  Survived  HighClassTicket  IsOld  hasLargeFamily  isSingle  \
        0          128         1                0      0               0         1
        1         1037         0                0      0               0         0
        2          633         1                1      0               0         1
        3         1051         1                0      0               0         0
        4          727         1                1      0               0         0

           hadNiceCabin  isAristocrat     Fare     Sex
        0             0             0   7.1417    male
        1             0             0  18.0000    male
        2             1             0  30.5000    male
        3             0             0  13.7750  female
        4             0             0  21.0000  female
```

Each passenger is identified with a unique PassengerId and is labeled with whether or not she survived the Titanic accident. We can also see that we have some simple information about each of them. In each column, 1 signifies True and 0 False. Since the decision tree we have implemented is quite simple and knows to split on only binary features (either 1 or 0), we have preprocessed the data and have already binaraized some features for you. They are as follows: - `HighClassTicket`: Signifies whether or not the passenger bought a ticket with some extra perks - `IsOld`: Signifies whether or not the passenger is older than 22 - `hasLargeFamily`: Signifies whether the passenger had more than 4 other family members on board - `isSingle`: Signifies whether the passenger had no other family members on board - `hadNiceCabin`: Signifies whether the passenger purchased an upgraded cabin - `isAristocrat`: Signifies whether the passenger had an aristocratic title in his/her name (E.g. Sir, Lord, Dutchess etc.)

However, you have to do some **feature engineering** and 'binarize' the remaining columns.

Unfortunately, the simple decision tree that we implemented does not know how to find partitions in features that are strings or features that are continuous. We will have to do some **feature engineering** to solve this. Binarizing the `Sex` feature is simple. However you will have to figure out a reasonable threshold for binarizing the `Fare` feature.

Do some data analysis below to find a reasonbable. **Plot a chart** and **explain** why you chose the threshold you chose. (Hint: Use histograms, analyze the survival rates and make a reasobable guess. Or find the threshold that minimizes impurity!)

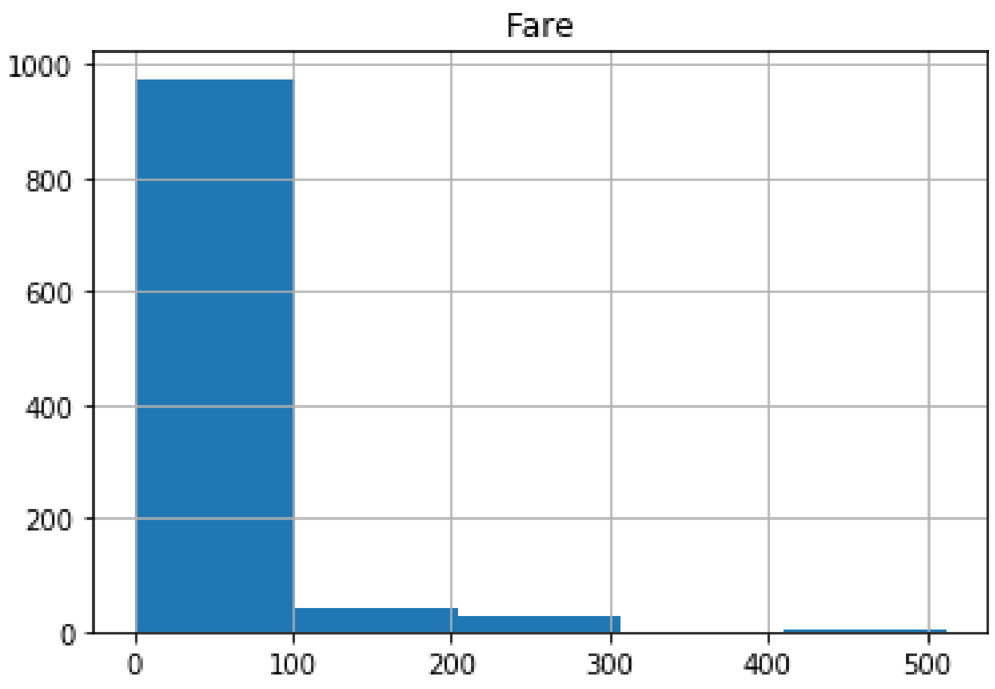
Also, **complete** the `binarize_features` function in `binarize.py`, this function should return a Pandas dataframe with the same number of columns, and binarize the `Fare` and `Sex` columns to `ints`.

```
In [3]: import statistics
        print("Mean: " + str(train_df['Fare'].mean()))
        print("Median: " + str(train_df['Fare'].median()))
        print("Max: " + str(train_df['Fare'].max()))
        print("Stdev: " + str(statistics.stdev(train_df['Fare'])))
```

```
Mean: 33.75423810888252
Median: 14.4542
Max: 512.3292
Stdev: 51.996485547945035
```

```
In [4]: train_df.hist(column = 'Fare', bins = 5)
```

```
Out[4]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x10d2e7b00>]],
              dtype=object)
```



Q1: Why did you threshhold/binarize the Fare featuer at that value?

A1: The theshold fare is set to at 25 which is a roughly good approximation to the middle point in our dataset. As most ~90% of the fare is under 100 from the graph and the Mean is 33, 25 is roughly in the middle to clasify as cheap (0) and expensive (1)

```
In [5]: train_df = features.binarize_features(train_df, 25)
        train_df.head()
```

```
Out[5]:    PassengerId  Survived  HighClassTicket  IsOld  hasLargeFamily  isSingle  \
        0          128         1                0      0               0         1
        1         1037         0                0      0               0         0
        2          633         1                1      0               0         1
        3         1051         1                0      0               0         0
        4          727         1                1      0               0         0

           hadNiceCabin  isAristocrat  Fare  Sex
        0             0             0     0    0
        1             0             0     0    0
        2             1             0     1    0
        3             0             0     0    1
        4             0             0     0    1
```

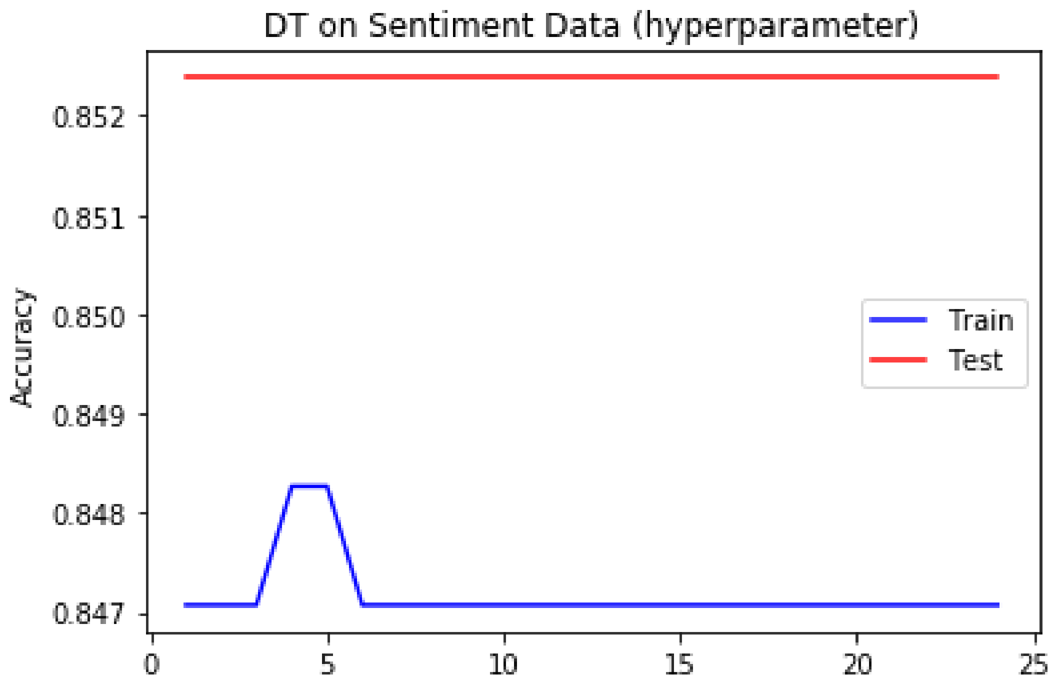
Although there is a test csv, we won't always have access to labels in our test data. Instead, we hold out a portion (20%) of our training data to help us measure how generalizable the trained model is.

```
In [6]: X, y = train_df.iloc[:,2:].values, train_df.iloc[:,1].values
        X_train, X_holdout, y_train, y_holdout = train_test_split(X, y, test_size=0.2, random_s
```

Using `hyperparameterCurve` in `runClassifier.py`, **plot** a corresponding chart of tree depth vs accuracy and **choose** the best tree depth.

```
In [7]: curve = runClassifier.hyperparamCurve(dt.DT({}), 'maxDepth', [1,2,3,4,5,6,7,8,12,16,20
        runClassifier.plotCurve('DT on Sentiment Data (hyperparameter)', curve)
```

```
Training classifier with maxDepth=1...
Training accuracy 0.847073, test accuracy 0.852381
Training classifier with maxDepth=2...
Training accuracy 0.847073, test accuracy 0.852381
Training classifier with maxDepth=3...
Training accuracy 0.847073, test accuracy 0.852381
Training classifier with maxDepth=4...
Training accuracy 0.848268, test accuracy 0.852381
Training classifier with maxDepth=5...
Training accuracy 0.848268, test accuracy 0.852381
Training classifier with maxDepth=6...
Training accuracy 0.847073, test accuracy 0.852381
Training classifier with maxDepth=7...
Training accuracy 0.847073, test accuracy 0.852381
Training classifier with maxDepth=8...
Training accuracy 0.847073, test accuracy 0.852381
Training classifier with maxDepth=12...
Training accuracy 0.847073, test accuracy 0.852381
Training classifier with maxDepth=16...
Training accuracy 0.847073, test accuracy 0.852381
Training classifier with maxDepth=20...
Training accuracy 0.847073, test accuracy 0.852381
Training classifier with maxDepth=24...
Training accuracy 0.847073, test accuracy 0.852381
```



Q2: According to your analysis, what is the best tree depth? Why?

A2: The best accuracy is around 4~5 which gives the highest train accuracy but it is mostly the same.

Now let's retrain on all the data...

```
In [8]: dt = dt.DT({'maxDepth': 5})
        dt.train(X, y)
```

Q3: Why would we want to retrain a decision tree on all the data (X and y) and not just X_train and y_train?

A3: From the above experiments we know the best depth to run our simulation, now we must train it with more samples. The data is also scattered, like how the fare price was consentrated in the first 100 but was still scattered for later values. Also the data size increases by 1.25 times which is better for training.

We can now test our decision tree the test data!

```
In [9]: test_df = pd.read_csv('data/titanic_test.csv')
        test_df = features.binarize_features(test_df, 25 )
        X_test, y_test = test_df.iloc[:,2:].values, test_df.iloc[:,1].values
```

```
In [10]: y_predicted = dt.predictAll(X_test)
         acc = np.mean(y_test == y_predicted)
         print("Test accuracy:", acc)
```

```
Test accuracy: 0.8854961832061069
```

assignment_1c

September 17, 2018

1 Part 1c: Information Gain for Decision Trees

DUE September 17th 2018

1.1 Introduction

The code for this project consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore.

1.1.1 Files You'll Edit

`assignment_1c.ipynb`: Will be your edited copy of this notebook pertaining to part 1c of the assignment.

1.1.2 Files you might want to look at

`binary.py`: Our generic interface for binary classifiers (actually works for regression and other types of classification, too).
`datasets.py`: Where a handful of test data sets are stored.
`util.py`: A handful of useful utility functions: these will undoubtedly be helpful to you, so take a look!
`runClassifier.py`: A few wrappers for doing useful things with classifiers, like training them, generating learning curves, etc.
`mlGraphics.py`: A few useful plotting commands
`data/*`: all of the datasets we'll use.

1.1.3 What to Submit

You will hand in all of the python files listed above under "Files you'll edit". You will also have to answer the written questions in this notebook denoted **Q#:** in the corresponding cells denoted with **A#:**.

Autograding Your code will be autograded for technical correctness. Please **do not** change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder's output – will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

```
In [1]: # Jupyter magic!
        %matplotlib inline
```

Your first task is to write code to support the use of the information gain splitting criterion for decision tree learning. **You should now complete all lines marked `TODO` in `dt.py`, so that our code handles both splitting criteria (misclassification rate and information gain).** Once you've done that, we can test our code for the new splitting criterion:

```
In [2]: import dumbClassifiers, datasets
        import numpy as np
        import runClassifier
        import dt as dt
```

```
In [3]: h = dt.DT({'maxDepth': 1, 'criterion': 'ig'})
        h

Out[3]: Leaf 1
```

```
In [4]: ################################################################################
        # TODO: Implement dt.train(...)                                                #
        ################################################################################
        h.train(datasets.TennisData.X, datasets.TennisData.Y)
        h

Out[4]: Branch 5 [Gain='1.6629']
          Leaf 1.0
          Leaf 1.0
```

```
In [5]: from numpy import *

        from binary import *
        c = 0.6428571428571429
        (c*(math.log(c,2))) + ((1-c)*math.log((1-c),2))

Out[5]: -0.9402859586706311
```

This is for a simple depth-one decision tree (aka a decision stump). Notice how we print the information gain corresponding to each branch in the tree.

If we let it get deeper, we get things like:

```
In [6]: h = dt.DT({'maxDepth': 2, 'criterion': 'ig'})
        h.train(datasets.TennisData.X, datasets.TennisData.Y)
        h

Out[6]: Branch 5 [Gain='1.6629']
          Branch 2 [Gain='1.7286']
            Leaf 1.0
            Leaf 1.0
          Branch 6 [Gain='1.6226']
            Leaf 1.0
            Leaf 1.0
```

```
In [7]: h = dt.DT({'maxDepth': 5, 'criterion': 'ig'})
        h.train(datasets.TennisData.X, datasets.TennisData.Y)
        h

Out[7]: Branch 5 [Gain='1.6629']
          Branch 2 [Gain='1.7286']
            Branch 7 [Gain='1.7320']
              Branch 6 [Gain='1.6085']
                Leaf 1.0
                Branch 4 [Gain='1.5305']
                  Leaf -1.0
                  Leaf -1.0
              Branch 1 [Gain='1.5305']
                Branch 0 [Gain='2.0000']
                  Leaf 1.0
                  Leaf -1.0
                Leaf 1.0
            Branch 3 [Gain='1.8366']
              Leaf 1.0
              Leaf 1.0
          Leaf 1.0
```

We can do something similar on the sentiment data (this will take a bit longer—it takes about 10 seconds on my laptop):

```
In [8]: h = dt.DT({'maxDepth': 2, 'criterion': 'ig'})
        h.train(datasets.SentimentData.X, datasets.SentimentData.Y)
        h

Out[8]: Branch 8160 [Gain='1.9991']
          Branch 8151 [Gain='1.9991']
            Leaf 1.0
            Leaf 1.0
          Leaf 1.0
```

We can look up the words (your results here might be different due to hashing):

```
In [9]: print(626, datasets.SentimentData.words[626])
        print(683, datasets.SentimentData.words[683])
        print(1627, datasets.SentimentData.words[1627])

626 bad
683 worst
1627 stupid
```

Based on this, we can rewrite the tree (by hand) as:

```
Branch 'bad'
  Branch 'worst'
    Leaf 1.0
    Leaf -1.0
  Branch 'stupid'
    Leaf -1.0
    Leaf -1.0
```

Now, we will test prediction (this takes about a minute for me):

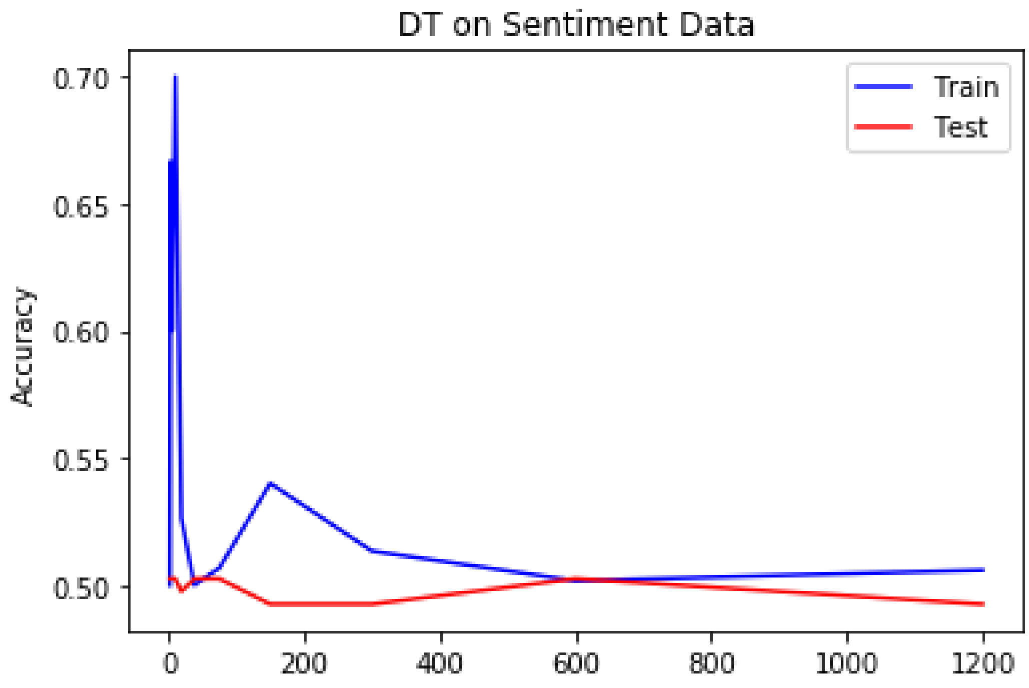
```
In [10]: ################################################################################
         # TODO: Implement dt.predict(...)                                              #
         ################################################################################
         runClassifier.trainTestSet(dt.DT({'maxDepth': 1, 'criterion': 'ig'}), datasets.Sentim
         runClassifier.trainTestSet(dt.DT({'maxDepth': 3, 'criterion': 'ig'}), datasets.Sentim
         runClassifier.trainTestSet(dt.DT({'maxDepth': 5, 'criterion': 'ig'}), datasets.Sentim
```

```
Training accuracy 0.504167, test accuracy 0.5025
Training accuracy 0.505, test accuracy 0.4975
Training accuracy 0.505833, test accuracy 0.5025
```

We can use more `runClassifier` functions to generate learning curves and hyperparameter curves:

```
In [11]: curveIG = runClassifier.learningCurveSet(dt.DT({'maxDepth': 9, 'criterion': 'ig'}), da
         runClassifier.plotCurve('DT on Sentiment Data', curveIG)
```

```
Training classifier on 2 points...
Training accuracy 0.5, test accuracy 0.5025
Training classifier on 3 points...
Training accuracy 0.666667, test accuracy 0.5025
Training classifier on 5 points...
Training accuracy 0.6, test accuracy 0.5025
Training classifier on 10 points...
Training accuracy 0.7, test accuracy 0.5025
Training classifier on 19 points...
Training accuracy 0.526316, test accuracy 0.4975
Training classifier on 38 points...
Training accuracy 0.5, test accuracy 0.5025
Training classifier on 75 points...
Training accuracy 0.506667, test accuracy 0.5025
Training classifier on 150 points...
Training accuracy 0.54, test accuracy 0.4925
Training classifier on 300 points...
Training accuracy 0.513333, test accuracy 0.4925
Training classifier on 600 points...
Training accuracy 0.501667, test accuracy 0.5025
Training classifier on 1200 points...
Training accuracy 0.505833, test accuracy 0.4925
```

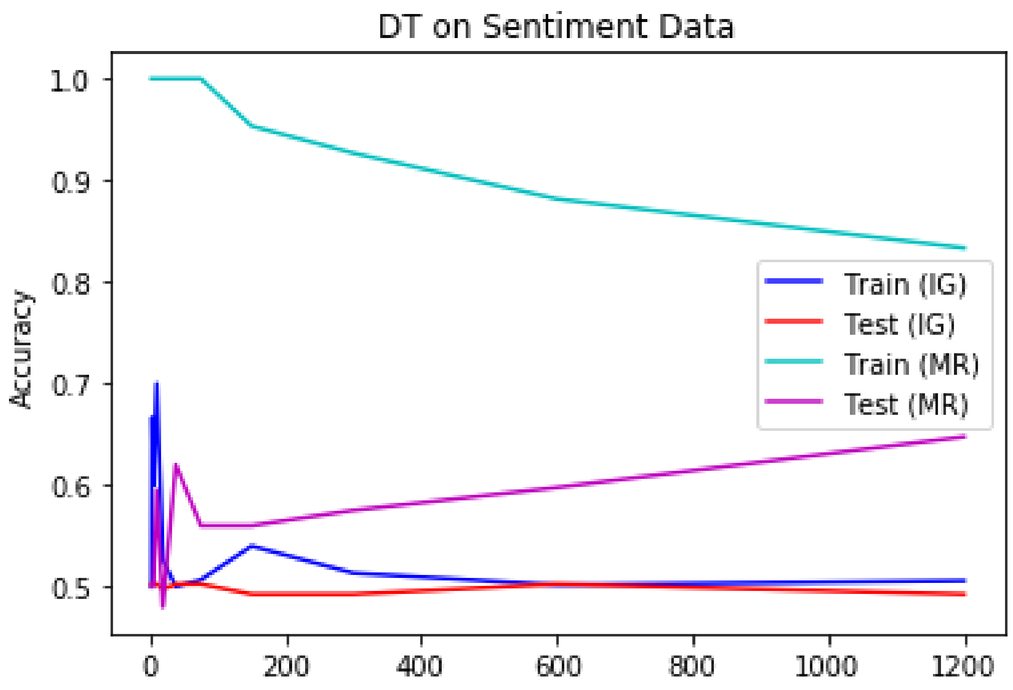


This plots training and test accuracy as a function of the number of data points (x-axis) used for training and y-axis is accuracy.

Now let's compare information gain with misclassification rate. First we'll generate the learning curve for misclassification rate again. Then we'll plot the curves on the same graph.

```
In [12]: curveMR = runClassifier.learningCurveSet(dt.DT({'maxDepth': 9, 'criterion': 'mr'}), da
         runClassifier.plotCurvePair('DT on Sentiment Data', curveIG, 'IG', curveMR, 'MR')
```

```
Training classifier on 2 points...
Training accuracy 1, test accuracy 0.5
Training classifier on 3 points...
Training accuracy 1, test accuracy 0.5
Training classifier on 5 points...
Training accuracy 1, test accuracy 0.5
Training classifier on 10 points...
Training accuracy 1, test accuracy 0.595
Training classifier on 19 points...
Training accuracy 1, test accuracy 0.48
Training classifier on 38 points...
Training accuracy 1, test accuracy 0.62
Training classifier on 75 points...
Training accuracy 1, test accuracy 0.56
Training classifier on 150 points...
Training accuracy 0.953333, test accuracy 0.56
Training classifier on 300 points...
Training accuracy 0.926667, test accuracy 0.575
Training classifier on 600 points...
Training accuracy 0.881667, test accuracy 0.5975
Training classifier on 1200 points...
Training accuracy 0.833333, test accuracy 0.6475
```



Q1: Briefly compare the two **training** curves. Does either splitting criterion perform better than the other for small dataset size (say, N<200)? Why or why not? How about as N increases to 1200? Use your understanding of both criteria to answer these questions.

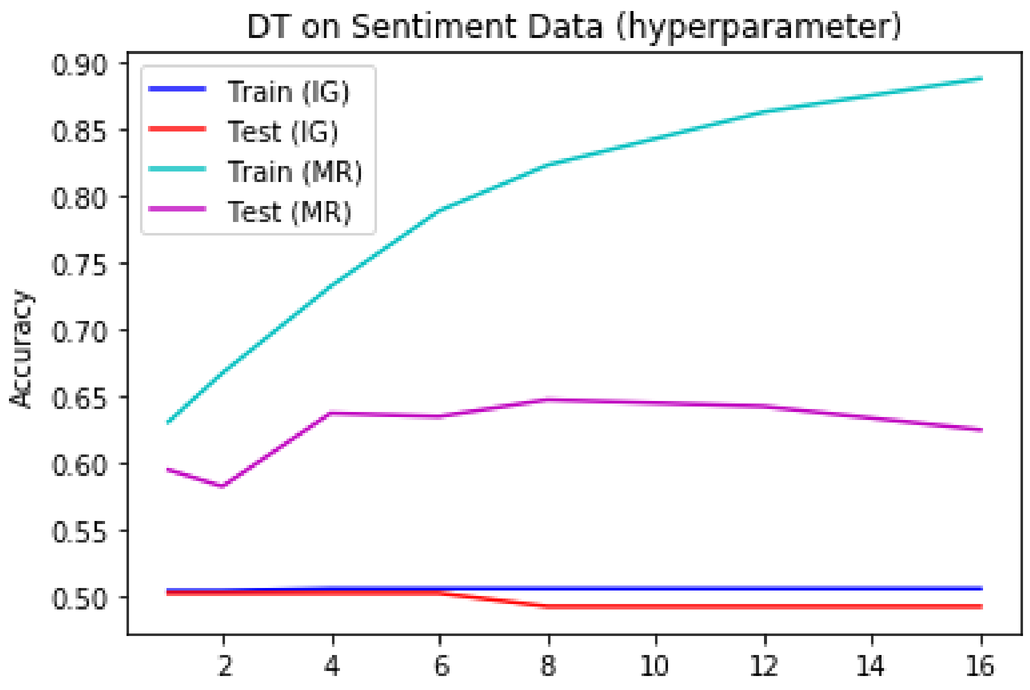
A1: For training the results are unusual. It shows higher training accuracy than when N is incresed. Which is similar to part 1a where for less data the model is very well unfitted. This covers less cases and the training accuracy is around 1.

Q2: Briefly compare the two **test** curves. Does either splitting criterion lead to better generalization than the other for small dataset size (say, N<200)? Why or why not? How about as N increases to 1200? Use your understanding of both criteria to answer these questions.

A2: When the dataset size is low, the splititng criterion doesn't perform well as there is low test accuracy. It improves as N is increased as it increases the dataset for both testing and training, which allows a more comprehensive learning.

We can also generate similar curves by changing the maximum depth hyperparameter:

```
In [13]: curveIG = runClassifier.hyperparamCurveSet(dt.DT({'criterion': 'ig'}), 'maxDepth', [1
         curveMR = runClassifier.hyperparamCurveSet(dt.DT({'criterion': 'mr'}), 'maxDepth', [1
         runClassifier.plotCurvePair('DT on Sentiment Data (hyperparameter)', curveIG, 'IG', cu
Training classifier with maxDepth=1...
Training accuracy 0.504167, test accuracy 0.5025
Training classifier with maxDepth=2...
Training accuracy 0.504167, test accuracy 0.5025
Training classifier with maxDepth=4...
Training accuracy 0.505833, test accuracy 0.5025
Training classifier with maxDepth=6...
Training accuracy 0.505833, test accuracy 0.5025
Training classifier with maxDepth=8...
Training accuracy 0.505833, test accuracy 0.4925
Training classifier with maxDepth=12...
Training accuracy 0.505833, test accuracy 0.4925
Training classifier with maxDepth=16...
Training accuracy 0.505833, test accuracy 0.4925
Training classifier with maxDepth=1...
Training accuracy 0.630833, test accuracy 0.595
Training classifier with maxDepth=2...
Training accuracy 0.6675, test accuracy 0.5825
Training classifier with maxDepth=4...
Training accuracy 0.7325, test accuracy 0.6375
Training classifier with maxDepth=6...
Training accuracy 0.789167, test accuracy 0.635
Training classifier with maxDepth=8...
Training accuracy 0.823333, test accuracy 0.6475
Training classifier with maxDepth=12...
Training accuracy 0.863333, test accuracy 0.6425
Training classifier with maxDepth=16...
Training accuracy 0.888333, test accuracy 0.625
```



Now, the x-axis is the value of the maximum depth.

Q3: Briefly compare the two **training** curves. Does either splitting criterion perform better than the other for shallow depth (say, `maxDepth`<10)? Why or why not? How about as `maxDepth` increases to 16? Use your understanding of both criteria to answer these questions.

A3: For training curves, the accuracy increases as there is more depth for the data to be fit in which allows for better training accuracy. Data can be better fitted when there is more space but while testing it gives the same result.

Q4: Briefly compare the two **test** curves. Does either splitting criterion lead to better generalization than the other for shallow depth (say, `maxDepth`<10)? Why or why not? How about as `maxDepth` increases to 16? Use your understanding of both criteria to answer these questions.

A4: The test curve depict that the accuracy is slightly higher when maxDepth <10. This is due to the fact that the model has similar end points.

Now we will display a tree trained using information gain. Beside each branch, we print out the information gain corresponding to the split.

```
In [14]: h = dt.DT({'maxDepth': 5, 'criterion': 'ig'})
         h.train(datasets.SentimentData.X, datasets.SentimentData.Y)
         h

Out[14]: Branch 8160 [Gain='1.9991']
           Branch 8151 [Gain='1.9991']
             Branch 8152 [Gain='1.9991']
               Branch 8150 [Gain='1.9991']
                 Branch 8159 [Gain='1.9982']
                   Leaf 1.0
                   Leaf 1.0
```

```
                Leaf -1.0
              Leaf -1.0
            Leaf 1.0
          Leaf 1.0
```

Let's print some of the features, so we can see which words this tree uses to make decisions. I've looked up word indexed at 626, but you can go ahead and edit as needed.

```
In [15]: print(626, datasets.SentimentData.words[626])
```

```
626 bad
```

Q5: Look up some words used in the tree. Find a few representative words that seem *helpful* (contributing to higher accuracy) in this classification task, and a few representative words that seem *unhelpful*. Do you notice a correlation between information gain and the quality of the word in this task? Why or why not?

A5: Yes, there is a correlation between information gain and quality of the word in the task. The information gain will be higher for words which help in classification because there is actual information gained with correlation to that word than some other word.

Q6: Should we expect a significant change in test accuracy if we prune subtrees rooted at nodes corresponding to low information gain? Why or why not?

A6: There won't be much significant change in test accuracy if we prune subtrees at low information gain as long as their info gain >= 0. The won't affect the computation but negative info gain can cause results which are far from the truth.