Naive Bayes Tutorial Documentation

Release 1.0

Marc Sabate

Jun 13, 2018

CONTENTS

This practical will explore writing a Naive Bayes classifier in Python.

CHAPTER

ONE

BEFORE YOU START

1.1 Running Python using the Anaconda distribution

Connect to the Data Analytic Cluster:

```
ssh -X [userid]@login.rdf.ac.uk
```

Download and copy the practical material in your home directory.

This practical uses the Ancaonda scientific Python distribution. To access it you shuold load the Anaconda module:

```
module load anaconda
```

Enter the Python's interpreter typing `ipython` on the terminal.

1.2 Required modules for this practical

The practical will use a number of Python modules. Before you use these you need to load them:

```
>>> import os
>>> import pandas as pd
>>> import numpy as np
>>> from sklearn.metrics import roc_curve, roc_auc_score
>>> from sklearn.preprocessing import LabelEncoder
>>> import matplotlib.pyplot as plt
>>> from scipy.stats import norm
```

What do these packages do?

- scikit-learn - Machine Learning library in Python.
  - roc_curve: Compute Receiver operating characteristic (ROC).
  - roc_auc_score: Compute Area Under the Curve.
  - LabelEncoder: Binary label encoder of character features.
- pandas - Python library providing data structures and data anlysis tools.
- os - Python library providing a portable way of using operating system functionality.
- matplotlib: plotting Python library.
- scipy: Python library for scientific computing.
  - stats.norm: normal continuous random variable

CHAPTER

TWO

THE DATA SET

The data set we will use for this practical is the Adult data set from the UCI Machine Learning Repository. This data set can be build a classifier that predicts if adult Americans have an annual salary of less than or equal to $50,000 (<=50K) or above $50,000 (>50K).

Look at the website for the Adult data set to understand it more. You will see that it has both continuous and categorical features. This practical will build a Naive Bayes classifier that uses both these types of features.

Set your working directory to be the tutorial's src directory:

```
>>> os.chdir('/PATH/TO/NaiveBayesClassifierPractical/src')
```

The training and test data frames can be loaded using:

```
>>> execfile("loadData.py")
```

The training data frame is called `training` and the test data frame is called `test`.

Use the summary function to see quick summary of the data:

```
>>> training.describe(include='all')
```

You will notice that some of the data features are categorical and some are continuous. You can count the different values of the categorical data using for example:

```
>>> training['salary'].value_counts()
```

Notice also that the salary class in the training set is heavily biased towards <=50K.

Spend a few minutes further understanding the data.

CHAPTER

THREE

NAIVE BAYES CLASSIFIER

Recall that to implement a Naive Bayes Classifier we wish to use the following equation for each class to determine which class has highest probability of occurring given the feature data:

$$p(c_i|x) \propto p(c_i)p(x_1|c_i)p(x_2|c_i)...p(x_n|c_i)$$

So first we need to determine the a priori probability of each class occurring. Assuming our training set is representative then this is easily done. The `pd.DataFrame.value_counts()` function in pandas will be useful here:

```
>>> training['salary'].value_counts()
```

What is the a priori probability of class '<=50'?

What is the a priori probability of class '>50K'?

Next we need to consider the feature data. Let's start with the education feature. Again we can use pandas' `pd.crosstab()` function to obtain the data we need:

```
>>> pd.crosstab(training['education'], training['salary'])
```

This is the data we need to compute the probability of seeing the various feature values for each class.

Look at the code in classifier1.py and complete the function so that it uses the education feature value when computing the score for each class.

Test the function to ensure it produces the following results for training set instances 1 and 6:

```
>>> execfile('classifier1.py')
>>> clf = classifier1()
>>> clf.fit(training, training['salary'])
>>> clf.predict(training.iloc[0,:])
<=50K: 0.0962501151685, >50K: 0.0682104357974
'<=50K'
>>> clf.predict(training.iloc[5,:])
<=50K: 0.0234636528362, >50K: 0.0294524123952
'>50K'
```

Now we can apply the classifier to the whole test set and analyse the result:

```
>>> predictions = map(lambda i: clf.predict(test.iloc[i,:], printScores = False),␣
↪range(test.shape[0]))
>>> predictions = np.array(predictions, dtype=object)
>>> pd.crosstab(predictions, test['salary'], rownames = ['predicted'], colnames = [
↪'actual'])
actual       <=50K  >50K
predicted
```

```
<=50K      11881  3027
>50K         554   819
```

Here we have applied the classifier to all the test examples and produced a confusion matrix.

If we consider >50K to the positive then the true positive rate is 819/(3027+819) = 21.3% and the false positive rate is 554/(11881+554) = 4.5%. This seems a very low true positive rate. Maybe using more features will improve matters.

Edit the `classifier1` class so that it now includes the workclass feature. When completed you should see the following values for training set instances 1 and 6:

```
>>> execfile('classifier1.py')
>>> clf = classifier1()
>>> clf.fit(training, training['salary'])
>>> clf.predict(training.iloc[0,:])
<=50K: 0.00367946435413, >50K: 0.00307081798705
'<=50K'
>>> clf.predict(training.iloc[5,:])
<=50K: 0.0168317538732, >50K: 0.0186420511054
'>50K'
```

Now applying this to the whole test set gives:

```
>>> predictions = map(lambda i: clf.predict(test.iloc[i,:], printScores = False),␣
↪range(test.shape[0]))
>>> predictions = np.array(predictions, dtype=object)
>>> pd.crosstab(predictions, test['salary'], rownames = ['predicted'], colnames = [
↪'actual'])
actual     <=50K  >50K
predicted
<=50K      11764  2854
>50K         671   992
```

Is this better? We have more true positives (992 compared with 819) but we also have more false positives (671 compared with 554). Our true positive rate is now 25.8% and our false positive rate is now 5.4%. It is hard to tell if this is really any better.

CHAPTER

FOUR

ROC CURVES

Choosing the best classifier requires considering a trade-off between between the true positive rate and the false positive rate. Our trained classifier has given as a single pair of true positive and false positive rates (TPR=25.8%, FPR=5.4%) but in fact it can support many other rate pairs if we simply adjust our sensitivity to predicting high salaries. We can adjust the rate simply by adding a weight to the score for high salaries. Let's add a weight of 2.0 to high salaries. This should result in more true positives at the expense of more false positives.

Add a weight of 2.0 to the high salary score and see how well the classifier works now. The true positive rate should now be 53.4% with a false positive rate of 18.8%.

If we adjust the weight for high salary we can get get a whole range of true positive rates for 0.0 to 1.0 (100%). If we alter our classifier code to return the ratio of high salary score to low salary score then we can easily adjust the weighting after we have applied the score all our training examples:

- Make of copy of your classifier1.py file and call it classifier2.py
- Remove any code you added to adjust the weight for the high salaries.
- Change the name of the class from `classifier1` to `classifier2`.
- Change the predict method to return `highSalaryScore/lowSalaryScore`

Now the threshold can be altered effectively adjust the weighting for a high salary. If the threshold is 1.0 we get the same results are before:

```
>>> clf = classifier2()
>>> clf.fit(training, training['salary'])    # we train the classifier
>>> ratios = map(lambda i: clf.predict(test.iloc[i,:], printScores = False),␣
↪range(test.shape[0]))
>>> predictions = map(lambda x: ">50K" if x>=1 else "<=50K", ratios)
>>> predictions = np.array(predictions, dtype=object)
>>> pd.crosstab(predictions, test['salary'], rownames = ['predicted'], colnames = [
↪'actual'])
actual      <=50K  >50K
predicted
<=50K       11764  2854
>50K          671   992
>>> pd.crosstab(predictions, test['salary'], rownames = ['predicted'], colnames = [
↪'actual']).apply(lambda r: r/r.sum(), axis=0)
actual         <=50K     >50K
predicted
<=50K       0.946039  0.74207
>50K        0.053961  0.25793
```

But if we set a lower threshold then we effectively increase the weight of the high salary class. Setting a threshold of 0.5 is equivalent to a high salary weighting of 2.0:

```
>>> predictions = map(lambda x: ">50K" if x>=0.5 else "<=50K", ratios)
>>> predictions = np.array(predictions, dtype=object)
>>> pd.crosstab(predictions, test['salary'], rownames = ['predicted'], colnames = [
↪'actual'])
actual      <=50K  >50K
predicted
<=50K       10100  1792
>50K         2335  2054
```

Graphs showing all the possible true positive and false positive rate combinations are known as receiver operating characteristic (ROC) curves. They always start in the bottom left corner (TPR=FPR=0) and end in the top right corner (TPR=FTP=1.0). Typically the closer the curve gets to the top left corner (0,1) the better.

The scikit-learn library includes the `roc_curve` function to get the TPR and FPR different values depending on the ratio threshold:

```
>>> fpr, tpr, thresholds = roc_curve(test['salary'], ratios, pos_label = ">50K")
>>> plt.figure(1)
>>> plt.plot([0, 1], [0, 1], 'k--')
>>> plt.plot(fpr, tpr, label='NB')
>>> plt.xlabel('False positive rate')
>>> plt.ylabel('True positive rate')
>>> plt.title('ROC curve')
>>> plt.legend(loc='best')
>>> plt.show()
```

This curve should contain the TPR and FPR pairs identified above. How would you find the threshold for which the corresponding point is the closest to (0,1)? Hint: compute the distance vector for all pairs (fpr[i], tpr[i]) and get the minimum:

```
>>> distances = np.array[(fpr[i])**2+(1-tpr[i])**2 for i in range(fpr.shape[0])])
>>> t = thresholds[np.argmin(distances)]
```

The area under the ROC curve is a useful single measure of the of how good the classifier is. The `roc_auc_score` function can be used to return the area under the curve. In order to use this function we must previously encode to salary feature to a binary variable:

```
>>> le = LabelEncoder()
>>> le.fit(test['salary'])
>>> le.classes_
>>> salary_binary = le.transform(test['salary'])
>>> auc = roc_auc_score(salary_binary, ratios)
>>> auc
0.72513553055190183
```

So the area under the ROC curve for a Naive Bayes classifier using just the education and workclass features is 0.72.

CHAPTER

FIVE

ADDING CONTINUOUS FEATURES

For categorical features such as the ones we have used so we can determine the required probability models simply by counting. This approach does not work for continuous data. For continuous data we must model the data using a appropriate distribution. Consider the age data in our data set. We can plot the density histogram of this feature for the low and high salary groups using the drawAgeHistograms.py script:

```
>>> execfile('drawAgeHistograms.py')
```

You can see that there is a clear difference between the two groups although there is significant overlap.

We wish to model this distributions using the normal distribution (also called the Gaussian distribution). The normal distribution is specified by two parameters: mean and standard deviation.

Using the pandas' functions `pd.DataFrame.mean()` and `pd.DataFrame.std()` we can easily calculate the mean and standard deviation for the two groups:

```
>>> data1 = training.loc[training['salary']=="<=50K",:]
>>> data2 = training.loc[training['salary']==">50K",:]
>>> mu1, sd1 = data1['age'].mean(), data1['age'].std()
>>> mu2, sd2 = data2['age'].mean(), data2['age'].std()
```

To see how well the normal distributions match the data we can plot them on top of the data:

```
>>> execfile('drawAgeHistogramsWithNormalModels.py')
```

The normal model fits well for the high salary group but is less good a fit for the low salary group. Nonetheless we will carry on with these values and add the age feature into our classifier.

Adjust your classifier2 to use the age feature:

- Use the `scipy.stats.norm` function for the normal probability density function

Applying your classifier to the test set should produce the following confusion matrix:

```
>>> execfile('classifier2.py')
>>> clf = classifier2()
>>> clf.fit(training, training['salary'])
>>> ratios = map(lambda i: clf.predict(test.iloc[i,:], printScores = False),␣
↪range(test.shape[0]))
>>> predictions = map(lambda x: ">50K" if x>=1 else "<=50K", ratios)
>>> predictions = np.array(predictions, dtype=object)
>>> pd.crosstab(predictions, test['salary'], rownames = ['predicted'], colnames = [
↪'actual'])
actual      <=50K  >50K
predicted
<=50K       11509  2471
>50K          926  1375
```

Re-plot the ROC curve for the new classifier. Does it look better? What is the area under the curve now.

CHAPTER

SIX

OPTIONAL ADDITIONAL WORK

Add more features to your Naive Bayes classifier. Adding categorical features education number, marital status, occupation, relationship, race, sex and native country increases the area under the ROC curve to 0.88.