Open in Colab

Sentiment Classification Using Naïve Bayes

Abstract

In this notebook you'll be implementing the Naïve Bayes algorithm and applying it to the task of classifying the sentiment of a movie review. We're going to take a mostly bottom-up approach here where we directly connect the math we've been learning with the algorithm. At the end we'll compare our results to sklearn's built-in implementation of the algorithm to both sanity check our own implementation and to allow those who desire to run more experiments with an implementation that has more options.

Sentiment Analysis

The Wikipedia Article on Sentiment Analysis provides the following definition for sentiment analysis.

> Sentiment analysis (also known as opinion mining or emotion AI) refers to the use of natural language processing, text analysis, computational linguistics, and biometrics to systematically identify, extract, quantify, and study affective states and subjective information. Sentiment analysis is widely applied to voice of the customer materials such as reviews and survey responses, online and social media, and healthcare materials for applications that range from marketing to customer service to clinical medicine.

In this notebook we'll be focusing on predicting the sentiment of a movie review from IMDB based on the text of the movie review. This dataset is one that was originally used in a Kaggle competition called Bag of Words meets Bag of Popcorn (you'll understand that joke by the end of this notebook!)

The data consists of the following.

> The labeled data set consists of 50,000 IMDB movie reviews, specially selected for sentiment analysis. The sentiment of reviews is binary, meaning the IMDB rating < 5 results in a sentiment score of 0, and rating >=7 have a sentiment score of 1. No individual movie has more than 30 reviews. The 25,000 review labeled training set does not include any of the same movies as the 25,000 review test set. In addition, there are another 50,000 IMDB reviews provided without any rating labels.

Our goal will be to see if we can learn a model, using Naïve Bayes on a training set to accurately estimate sentiment of new reviews.

Without further ado, let's download and parse the data into a data frame.

In [0]:

```
import gdown
import pandas as pd

gdown.download('https://drive.google.com/uc?authuser=0&id=1Z8bwIBa_0gFe9-
C2W0goZ72lQfFMbxjS&export=download',
               'labeledTrainData.tsv',
               quiet=False)
df = pd.read_csv('labeledTrainData.tsv', header=0, delimiter='\t')
df
```

Let's look at the average sentiment to see what we are dealing with (1 is positive sentiment and 0 is negative)

In [0]:

```
df['sentiment'].mean()
```

Looks like we're dealing with a balanced set of positives and negatives.

Next, let's look at a particular review. To make the output look nicer, we'll create a new Pandas series with line wrapping.

In [0]:

```
# this takes a little while to run
```

```
reviews_wrapped = df['review'].str.wrap(80)
```

In [0]:

```
print(reviews_wrapped.iloc[20])
```

The Bag of Words Model

We know that in order to apply Naïve Bayes we need to convert each of our reviews into a vector of features. There are lots of different methods to convert text into vectors. In this notebook we'll be using a pretty basic (but suprisingly powerful) form of vectorization where we construct a feature vector with k entries (where k is the total number of unique words in the dataset) and for any particular review we set the corresponding entry to 1 if that word appears in the review and 0 otherwise. This representation is called ***bag of words*** since the encoding of the review into a vector is independent of where the words occur in the review (you could shuffle the words in the review and still have the same feature vector). The Wikipedia article on Bag of Words has more information.

Instead of writing our own code to convert from text to a bag of words representation we're going to use scikit learn's built-in count vectorizer. Before we apply it to the data, let's apply it to toy dataset to help you better understand the bag of words model.

In [0]:

```
from sklearn.feature_extraction.text import CountVectorizer
import numpy as np

# the binary feature makes it so only the presence or absence of the word is
# returned (rather than the count)
vectorizer = CountVectorizer(binary=True)
toy_data = ['This is review one.  It has some words.', 'This is review two.  It also has some
words.']
vectorizer.fit(toy_data)
X_toy = vectorizer.transform(toy_data)
X_toy
```

The first thing to notice about the data is that it's stored in a sparse matrix format. A sparse matrix is useful when the elements of your matrix are mostly zeros. If you have a lot of unique words, but each piece of text only contains a small fraction of those words, your matrix will be sparse. In this notebook we'll be limiting the size of our vocabulary and converting the matrix to a dense (i.e., typical) format. This is to ease implementation. If you want to leave things as a sparse matrix, please feel free to do so.

In [0]:

```
print("feature vectors", X_toy.todense())
print(vectorizer.get_feature_names())
```

Notebook Exercise 1

Referencing back to what you know about the bag of words model and our toy data, explain why the vectors look the way they do. Make up your own toy data (or add to ours) and see if the results make sense.

Solution

The first entry of the vector corresponds to the word 'also', which occurs only in the second review (thus the first line after feature vectors has a 0 for the first entry and the second line has a 1). The second entry of the vector corresponds to the word 'has', which is in both reviews (so both vectors have a 1 there). The pattern continues as you go through the vector.

Vectorizing the Whole Dataset

Now that you have a general idea what bag of words is all about, let's apply it to our movie reviews. To make our lives easier we're only going to include words in our feature vector if they occur in at least 100 reviews. Doing this will help with overfitting (although next assignment we will be learning another technique to deal with this). While we're at it we'll also convert the sentiment labels to a numpy array.

In [0]:

```
vectorizer = CountVectorizer(binary=True, min_df=100)
vectorizer.fit(df['review'])
```

```
vectorizer.fit(df['review'])
X = vectorizer.transform(df['review']).todense()
y = np.array(df['sentiment'])
print("X.shape", X.shape)
print("y.shape", y.shape)
```

As a quick intuition builder, let's look at a word we think would probably differ across sentiment values.

In [0]:

```
terrible_index = vectorizer.get_feature_names().index('terrible')
print("terrible occurs in", X[y==1, terrible_index].mean(), "for Y=1")
print("terrible occurs in", X[y==0, terrible_index].mean(), "for Y=0")
```

Sorta makes sense (but please someone do some looking into what those reviews are where it terrible appears and it is Y=1)

The General Form of Naïve Bayes

Last Assignment we showed how the Titanic problem could be solved using the Naïve Bayes Model. Specifically we computed what's known as the odds ratio.

$$\frac{p(\mathcal{Y}|\mathcal{S})p(\mathcal{C}=1|\mathcal{S})p(\mathcal{M}|\mathcal{S})p(\mathcal{S})}{p(\mathcal{Y}|\neg\mathcal{S})p(\mathcal{C}=1|\neg\mathcal{S})p(\mathcal{M}|\neg\mathcal{S})p(\neg\mathcal{S})}$$

Recall that $\mathcal{Y}$ meant young passenger, $\mathcal{C}$ represented fare class, $\mathcal{M}$ represented "is male", and $\mathcal{S}$ represented survival.

Hopefully it is pretty clear that while we derived the formula in the specific case of the Titanic model, the logic we applied is completely general. If we assume that we are doing binary classification provided values $x_1, x_2, \ldots, x_d$, then the odds ratio can be written in the following way.

$$\frac{p(Y=1|X_1=x_1, X_2=x_2, \ldots, X_d=x_d)}{p(Y=0|X_1=x_1, X_2=x_2, \ldots, X_d=x_d)} = \frac{p(Y=1)\times p(X_1=x_1|Y=1)\times\ldots\times p(X_d=x_d|Y=1)}{p(Y=0)\times p(X_1=x_1|Y=0)\times\ldots\times p(X_d=x_d|Y=0)}$$

We also learned last assignment that if this odds ratio is greater than 1, we should predict positive. While the odds ratio is a totally valid way to attack the problem, it can be numerically unstable. Therefore, most people use the log odds ratio instead (you just hit both sides with a log).

Notebook Exercise 2

Compute the log odds ratio for the Naïve Bayes algorithm by taking the log of both sides of the preceding equation. Simplify as much as possible using properties of logarithms. Assuming you'd like to predict $Y=1$ whenever your model assigns a probability greater than 0.5 to the output being 1 given the data, what must be true of the log odds ratio in this case?

Solution

The answer is pretty straight forward. We use the fact that log of a ratio is the difference in the log of the numerator and denominator. We also use the fact that log of a product is the sum of the logs of each of the terms.

$$\log\left(\frac{p(Y=1|X_1=x_1, X_2=x_2, \ldots, X_d=x_d)}{p(Y=0|X_1=x_1, X_2=x_2, \ldots, X_d=x_d)}\right) = \log p(Y=1) - \log p(Y=0) + \sum_{i=1}^{d}\left(\log p(X_i=x_i|Y=1) - \log p(X_i=x_i|Y=0)\right)$$

If the log odds ratio is greater than or equal to 0, we should predict $Y=1$ (at least our probability will be at least 0.5).

Fitting the Parameters of the Model

What we see from looking at the log odds ratio equation is that in order to apply the model we must have an estimate of the following probabilities.

- $p(Y=0)$
- $p(Y=1)$
- $p(X_i=x_i|Y=0)$ (for i from 1 to d)
- $p(X_i=x_i|Y=1)$ (for i from 1 to d)

The strategy for fitting these parameters will be the same as was described in the previous assignment (we'll see a more formal

The strategy for fitting these parameters will be the same as was described in the previous assignment (we'll see a more formal justification of why this works in the next assignment). In order to estimate a probability, we'll just count the number of times the event occurs across the dataset.

For instance, if we want to estimate $p(Y = 0)$ we would count the number of instances in the dataset where $Y = 0$ and divide that by the total number of instances in the dataset. If we wanted to esitmated $p(X_i = 1 \mid Y = 0)$ (suppose X_i represents the word "terrible") we would count the number of reviews that included the word terrible and had sentiment 0 and divide that by the number of reviews that were sentiment 0 (hopefully that makes sense given the definition of $p(A \mid B) = \frac{p(A,B)}{p(B)}$.

Notebook Exercise 3

Write a function that takes as input X and y and returns a tuple containing the following elements (in this order)

1. The probability of Y=1
2. The probability of Y=0
3. A vector where the ith entry represents $p(X_i = 1 \mid Y = 1)$
4. A vector where the ith entry represents $p(X_i = 1 \mid Y = 0)$

To help you out we've included a unit test and a function stub.

In [0]:

```
def fit_nb_model(X, y):
    """ Fit the parameters of a Naive Bayes model given a bag of words model
        with binary counts (X) and class labels (y).

        Returns: a tuple with the following components in order
                  The probability of Y=1,
                  The probability of Y=0,
                  A vector where the $i$th entry represents $p(X_i=1|Y=1)$
                  A vector where the $i$th entry represents $p(X_i=1|Y=0)$

    >>> X = np.array([[1, 0, 1], [1, 0, 0], [0, 0, 1], [1, 1, 1]])
    >>> y = np.array([1, 0, 1, 0])
    >>> fit_nb_model(X, y)
    (0.5, 0.5, array([0.5, 0. , 1. ]), array([1. , 0.5, 0.5]))
    """
    return None

import doctest
doctest.testmod()
```

In [0]:

```
# ***Solution***
def fit_nb_model(X, y):
    """ Fit the parameters of a Naive Bayes model given a bag of words model
        with binary counts (X) and class labels (y).

        Returns: a tuple with the following components in order
                  The probability of Y=1,
                  The probability of Y=0,
                  A vector where the $i$th entry represents $p(X_i=1|Y=1)$
                  A vector where the $i$th entry represents $p(X_i=1|Y=0)$

    >>> X = np.array([[1, 0, 1], [1, 0, 0], [0, 0, 1], [1, 1, 1]])
    >>> y = np.array([1, 0, 1, 0])
    >>> fit_nb_model(X, y)
    (0.5, 0.5, array([0.5, 0. , 1. ]), array([1. , 0.5, 0.5]))
    """
    X_1 = X[y == 1, :]
    X_0 = X[y == 0, :]
    return y.mean(), 1 - y.mean(), X_1.mean(axis=0), X_0.mean(axis=0)

import doctest
doctest.testmod()
```

Inference

Once we have the model parameters fitted, we have to be able to do inference on new data. This will amount to computing our log

odds ratio and seeing if it is greater than 0. If the log odds ratio is greater than 0, we return a predicted sentiment of 1, otherwise we return a sentiment of 0.

Before having you implement the inference code, let's split into a train and test set and use your code to fit the model.

In [0]:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y)
p_y_1, p_y_0, p_x_y_1, p_x_y_0 = fit_nb_model(X_train, y_train)
```

Notebook Exercise 4

Implement a function called `get_nb_prediction` that takes as input the NB model (`p_y_1, p_y_0, p_x_y_1, p_x_y_0`) and a dataset (X), computes the log odds ratio and returns a vector of predictions for that data.

To get you started we have a function stub.

Hint: you can write a nested loop with one over the data points and one over the features and compute log probabilities as you go. (you can speed this up using numpy features, but don't worry about computational speed)

In [0]:

```
def get_nb_predictions(p_y_1, p_y_0, p_x_y_1, p_x_y_0, X):
    """ Predict the labels for the data X given the Naive Bayes model """
    return None

# here is some test code that will call your model on the first 100 test points
# (for speed of development).
y_pred = get_nb_predictions(p_y_1, p_y_0, p_x_y_1, p_x_y_0, X_test[:100,:])
```

In [0]:

```
# ***Solution***
# This will be really slow, but we hope it is more readable for folks that are
# less familiar with numpy
def get_nb_predictions(p_y_1, p_y_0, p_x_y_1, p_x_y_0, X):
    """ Predict the labels for the data X given the Naive Bayes model """
    log_odds_ratios = np.zeros(X.shape[0])
    for i in range(X.shape[0]): # loop over data points
        print("progress", i/X.shape[0])
        log_odds_ratios[i] += np.log(p_y_1) - np.log(p_y_0)
        for j in range(X.shape[1]): #loop over words
            if X[i, j] == 1:
                log_odds_ratios[i] += np.log(p_x_y_1[0, j]) - np.log(p_x_y_0[0, j])
            else:
                log_odds_ratios[i] += np.log(1 - p_x_y_1[0, j]) - np.log(1 - p_x_y_0[0, j])
    return (log_odds_ratios >= 0).astype(np.float)

y_pred = get_nb_predictions(p_y_1, p_y_0, p_x_y_1, p_x_y_0, X_test[:100,:])
```

Calculating Accuracy

Now we'll test our model on all of the test data and see how accurate it is.

In [0]:

```
y_pred = get_nb_predictions(p_y_1, p_y_0, p_x_y_1, p_x_y_0, X_test)
print("accuracy is", (y_pred == y_test).mean())
```

Sanity Check

Just to see if we're in the ball park, let's try scikit learn's built-in implementation of Naïve Bayes.

In [0]:

```
from sklearn.naive_bayes import MultinomialNB
```

```
from sklearn.naive_bayes import MultinomialNB

model = MultinomialNB()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
np.mean(y_pred == y_test)
```