classifier using Naive Bayes

November 5, 2021

0.0.1 Step 0: Introduction to the Naive Bayes Theorem

Bayes theorem is one of the earliest probabilistic inference algorithms developed by Reverend Bayes (which he used to try and infer the existence of God no less) and still performs extremely well for certain use cases.

It's best to understand this theorem using an example. Let's say you are a member of the Secret Service and you have been deployed to protect the Democratic presidential nominee during one of his/her campaign speeches. Being a public event that is open to all, your job is not easy and you have to be on the constant lookout for threats. So one place to start is to put a certain threat-factor for each person. So based on the features of an individual, like the age, sex, and other smaller factors like is the person carrying a bag?, does the person look nervous? etc. you can make a judgement call as to if that person is viable threat.

If an individual ticks all the boxes up to a level where it crosses a threshold of doubt in your mind, you can take action and remove that person from the vicinity. The Bayes theorem works in the same way as we are computing the probability of an event(a person being a threat) based on the probabilities of certain related events(age, sex, presence of bag or not, nervousness etc. of the person).

One thing to consider is the independence of these features amongst each other. For example if a child looks nervous at the event then the likelihood of that person being a threat is not as much as say if it was a grown man who was nervous. To break this down a bit further, here there are two features we are considering, age AND nervousness. Say we look at these features individually, we could design a model that flags ALL persons that are nervous as potential threats. However, it is likely that we will have a lot of false positives as there is a strong chance that minors present at the event will be nervous. Hence by considering the age of a person along with the 'nervousness' feature we would definitely get a more accurate result as to who are potential threats and who aren't.

This is the 'Naive' bit of the theorem where it considers each feature to be independent of each other which may not always be the case and hence that can affect the final judgement.

In short, the Bayes theorem calculates the probability of a certain event happening(in our case, a message being spam) based on the joint probabilistic distributions of certain other events(in our case, the appearance of certain words in a message). We will dive into the workings of the Bayes theorem later in the mission, but first, let us understand the data we are going to work with.

0.0.2 Step 1.1: Understanding our dataset

We will be using a dataset from the UCI Machine Learning repository which has a very good collection of datasets for experimental research purposes. The direct data link is here.

The columns in the data set are currently not named and as you can see, there are 2 columns.

The first column takes two values, 'ham' which signifies that the message is not spam, and 'spam' which signifies that the message is spam.

The second column is the text content of the SMS message that is being classified.

[ ]:
```
import pandas as pd
df = pd.read_table("SMSSpamCollection",names=['label','sms_message'])
df.head()
```

[ ]:
```
  label                                          sms_message
0   ham  Go until jurong point, crazy.. Available only …
1   ham                      Ok lar… Joking wif u oni…
2  spam  Free entry in 2 a wkly comp to win FA Cup fina…
3   ham  U dun say so early hor… U c already then say…
4   ham  Nah I don't think he goes to usf, he lives aro…
```

0.0.3 Step 1.2: Data Preprocessing

[ ]:
```
df['label'] = df.label.map({'ham':0,'spam':1})
df.shape
```

[ ]:
```
(5572, 2)
```

0.0.4 Step 2.1: Bag of words

What we have here in our data set is a large collection of text data (5,572 rows of data). Most ML algorithms rely on numerical data to be fed into them as input, and email/sms messages are usually text heavy.

Here we'd like to introduce the Bag of Words(BoW) concept which is a term used to specify the problems that have a 'bag of words' or a collection of text data that needs to be worked with. The basic idea of BoW is to take a piece of text and count the frequency of the words in that text. It is important to note that the BoW concept treats each word individually and the order in which the words occur does not matter.

Using a process which we will go through now, we can convert a collection of documents to a matrix, with each document being a row and each word(token) being the column, and the corresponding (row,column) values being the frequency of occurrence of each word or token in that document.

For example:

Lets say we have 4 documents as follows:

```
['Hello, how are you!', 'Win money, win from home.', 'Call me now', 'Hello, Call you tomorrow?']
```

Our objective here is to convert this set of text to a frequency distribution matrix, as follows:

Here as we can see, the documents are numbered in the rows, and each word is a column name, with the corresponding value being the frequency of that word in the document.

To handle this, we will be using sklearns count vectorizer method which does the following:

- It tokenizes the string(separates the string into individual words) and gives an integer ID to each token.
- It counts the occurrence of each of those tokens.

0.0.5 Step 2.2: Implementing Bag of Words in scikit-learn

[ ]:
```
documents = ['Hello, how are you!',
             'Win money, win from home.',
             'Call me now.',
             'Hello, Call hello you tomorrow?']
```

[ ]:
```
from sklearn.feature_extraction.text import CountVectorizer
count_vector = CountVectorizer()
count_vector.fit(documents)
count_vector.get_feature_names_out()
```

[ ]:
```
array(['are', 'call', 'from', 'hello', 'home', 'how', 'me', 'money',
       'now', 'tomorrow', 'win', 'you'], dtype=object)
```

[ ]:
```
doc_array = count_vector.transform(documents).toarray()
doc_array
```

[ ]:
```
array([[1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1],
       [0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 2, 0],
       [0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0],
       [0, 1, 0, 2, 0, 0, 0, 0, 0, 1, 0, 1]], dtype=int64)
```

Now we have a clean representation of the documents in terms of the frequency distribution of the words in them. To make it easier to understand our next step is to convert this array into a dataframe and name the columns appropriately.

[ ]:
```
columns_names = count_vector.get_feature_names_out()
frequency_matrix = pd.DataFrame(doc_array,columns=columns_names)
frequency_matrix
```

[ ]:

	are	call	from	hello	home	how	me	money	now	tomorrow	win	you
0	1	0	0	1	0	1	0	0	0	0	0	1
1	0	0	1	0	1	0	0	1	0	0	2	0
2	0	1	0	0	0	0	1	0	1	0	0	0
3	0	1	0	2	0	0	0	0	0	1	0	1

We have successfully implemented a Bag of Words problem for a document dataset that we created.

One potential issue that can arise from using this method out of the box is the fact that if our dataset of text is extremely large(say if we have a large collection of news articles or email data), there will be certain values that are more common that others simply due to the structure of the language itself. So for example words like 'is', 'the', 'an', pronouns, grammatical constructs etc could skew our matrix and affect our analyis.

There are a couple of ways to mitigate this. One way is to use the `stop_words` parameter and set its value to `english`. This will automatically ignore all words(from our input text) that are found in a built in list of English stop words in scikit-learn.

0.0.6 Step 3.1: Training and testing sets

[ ]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test =␣
↪train_test_split(df['sms_message'],df['label'],random_state=1)

print('Number of rows in the total set: {}'.format(df.shape[0]))
print('Number of rows in the training set: {}'.format(X_train.shape[0]))
print('Number of rows in the test set: {}'.format(X_test.shape[0]))
```

```
Number of rows in the total set: 5572
Number of rows in the training set: 4179
Number of rows in the test set: 1393
```

0.0.7 Step 3.2: Applying Bag of Words processing to our dataset.

- Firstly, we have to fit our training data (`X_train`) into `CountVectorizer()` and return the matrix.
- Secondly, we have to transform our testing data (`X_test`) to return the matrix.

Note that `X_train` is our training data for the 'sms_message' column in our dataset and we will be using this to train our model.

`X_test` is our testing data for the 'sms_message' column and this is the data we will be using(after transformation to a matrix) to make predictions on. We will then compare those predictions with `y_test` in a later step.

[ ]:
```
count_vector = CountVectorizer()

training_data = count_vector.fit_transform(X_train)
testing_data = count_vector.transform(X_test)
```

0.0.8 Step 4: Naive Bayes implementation using scikit-learn

sklearn has several Naive Bayes implementations that we can use and so we do not have to do the math from scratch. We will be using sklearns `sklearn.naive_bayes` method to make predictions on our dataset.

Specifically, we will be using the multinomial Naive Bayes implementation. This particular classifier is suitable for classification with discrete features (such as in our case, word counts for text

classification). It takes in integer word counts as its input. On the other hand Gaussian Naive Bayes is better suited for continuous data as it assumes that the input data has a Gaussian(normal) distribution.

[ ]:
```
from sklearn.naive_bayes import MultinomialNB
naive_bayes = MultinomialNB()
naive_bayes.fit(training_data,y_train)
```

[ ]:
```
MultinomialNB()
```

[ ]:
```
predictions = naive_bayes.predict(testing_data)
```

Now that predictions have been made on our test set, we need to check the accuracy of our predictions.

0.0.9 Step 5: Evaluating our model

[ ]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score

print('Accuracy score: ', format(accuracy_score(y_test,predictions)))
print('Precision score: ', format(precision_score(y_test,predictions)))
print('Recall score: ', format(recall_score(y_test,predictions)))
print('F1 score: ', format(f1_score(y_test,predictions)))
```

```
Accuracy score:  0.9885139985642498
Precision score:  0.9720670391061452
Recall score:  0.9405405405405406
F1 score:  0.9560439560439562
```

0.0.10 Conclusion

One of the major advantages that Naive Bayes has over other classification algorithms is its ability to handle an extremely large number of features. In our case, each word is treated as a feature and there are thousands of different words. Also, it performs well even with the presence of irrelevant features and is relatively unaffected by them. The other major advantage it has is its relative simplicity. Naive Bayes' works well right out of the box and tuning it's parameters is rarely ever necessary, except usually in cases where the distribution of the data is known. It rarely ever overfits the data. Another important advantage is that its model training and prediction times are very fast for the amount of data it can handle.