How to create a Naive Bayes text classification model using scikit-learn

practicaldatascience.co.uk/machine-learning/how-to-create-a-naive-bayes-text-classification-model-using-scikit-learn

8 May 2022

Naive Bayes classifiers are commonly used for machine learning text classification problems, such as predicting the sentiment of a tweet, identifying the language of a piece of text, or categorising a support ticket. They're a mainstay of Natural Language Processing or NLP.

There are actually quite a few different Naive Bayes classification algorithms, which all utilise a technique known as Bayes' Theorem. The basic concept of this, and the "naive" part of the name, is that every pair of features is considered independent of each other and is equal.

Obviously, this assumption is often wrong in real-world situations, which is why it's called "naive" Bayes. However, that doesn't stop it being extremely effective. In this simple Python example, I'll show you the code you need to create a basic text classification model using the Multinomial Naive Bayes algorithm via the `MultinomialNB` module in scikit-learn.

Import the packages

```
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
from sklearn.metrics import f1_score

pd.set_option('max_colwidth', 1000)
```

Load the dataset

You can import any text dataset you like for this simple project. I'm using a Microsoft support ticket classification dataset. This contains the text from the support ticket a user sent to the help desk, plus some data on how it's been categorised, it's impact, urgency, the ticket type, and the category.

Since we have lots of training data, we could create a model that could examine the text of past tickets and predict the categorisation of future tickets. Support staff waste lots of time doing this manually, so this could save the support team a lot of time and save money for their business.

```
df = pd.read_csv('all_tickets.csv')
```

```
df.sample(3).T
```

	19510	23310	25833
ticket_type	1	1	1
category	4	4	4
sub_category1	2	3	2
sub_category2	7	21	21
business_service	32	46	40
urgency	3	3	3
impact	4	4	4
text	licenses for wednesday march licenses dear found out interns require accounts please provide licenses kind regards engineer	retrieve library from recycle bin friday pm re retrieve library recycle bin hi asks deleted restore please ask restore site tuesday pm retrieve library recycle bin hello please advised queue kind regards analyst ext hub tuesday pm retrieve library recycle bin hi guys please assist retrieving library old were migrated thank leader	internal transfer request tuesday november pm re transfer form si mine tuesday november re transfer form document tuesday november transfer form hello va transfer form transfer date administration officer

Examine the target variable

This dataset contains support tickets that have been categorised by support engineers using the `ticket_type`, `category`, `sub_category1`, `sub_category2`, `business_service`, `urgency`, and `impact` variables. We could create a model to predict any one of these values from the text data present. Let's take a look at a few of the potential target variables we could predict.

```
df.ticket_type.value_counts()

1    34621
0    13928
Name: ticket_type, dtype: int64

df.category.value_counts()
```

```
4     34061
5      9634
6      2628
7       921
11      612
8       239
9       191
3       137
1        72
12       45
0         4
2         3
10        2
Name: category, dtype: int64
```

```
df.business_service.value_counts()
```

```
32    8174
36    3685
68    3589
67    2858
4     2527
      ...
37       1
69       1
81       1
17       1
0        1
Name: business_service, Length: 103, dtype: int64
```

```
df.urgency.value_counts()
```

```
3    34621
1     6748
2     5528
0     1652
Name: urgency, dtype: int64
```

```
df.impact.value_counts()
```

```
4    34621
3    13184
0      471
2      228
1       45
Name: impact, dtype: int64
```

Let's go with the `ticket_type` variable. If we create a model that can predict the `ticket_type` class for each ticket, we can help staff to prioritise the tickets by their type. Using `value_counts()` shows that we have two values numbered 0-1, making this a binary text classification model problem.

The other variables have more values and are multiclass text classification problems. They can be handled in the same way, using exactly the same model, however, it will be much harder to generate accurate predictions and the model is going to require a lot more tuning, so we'll keep things simple instead.

Preprocess the text for the model

Before we can do anything, we first need to preprocess our text to convert it to a numeric form. Machine learning models can't use text, so the first step is to use a text preprocessing technique called Count Vectorization to turn the text into a vector of numbers via the `CountVectorizer` module in scikit-learn.

`CountVectorizer` is widely used in Natural Language Processing (NLP), both in the model building process and for the analysis of text via n-grams (also called Q-grams or shingles). `CountVectorizer` takes our text and returns a count for each time a word present in the entire "corpus" (that is, the whole dataset of tickets) appears in an individual ticket.

To use `CountVectorizer`, we'll instantiate it, then use `fit_transform()` to pass it the `text` and return a Bag of Words model. We'll then use Numpy to convert that Bag of Words to a dense array that can better be used by the model. Note that there are many other ways to preprocess text for machine learning models - this is one of many methods used.

```
count_vec = CountVectorizer()
bow = count_vec.fit_transform(df['text'])
bow = np.array(bow.todense())
```

Create the train and test data

Next, we'll pass our Bag of Words data stored in `bow` to `X` and use it as the feature set for our model. We'll assign the `impact` column of our dataset as the target variable `y` that we're aiming to predict.

We'll then use the `train_test_split()` function to create our train and test data, allocating 30% for testing or validation and using stratification to ensure that the proportions are split equally across the datasets.

```
X = bow
y = df['ticket_type']

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.3,
                                                    stratify=y)
```

Create a Multinomial Naive Bayes classification model

Now everything is set up, we'll fit a Multinomial Naive Bayes classification model using the `MultinomialNB` module from scikit-learn. We'll use the `fit()` function to pass this our `X_train` and `y_train` data to train the model to predict the `ticket_type` from the vectors of the ticket text.

The Multinomial Naive Bayes model can be used for classification on any data with discrete features, such as our word counts from the count vectorization process. Usually integer counts, such as those from count vectorizer are needed to get good results, but the model can work OK with fractional counts from other text preprocessing techniques such as TF-IDF (Term Frequency Inverse Document Frequency).

```
model = MultinomialNB().fit(X_train, y_train)
```

Evaluate model performance

Finally, in this very basic example, we can generate predictions from our trained model by passing the `X_test` validation dataset to `predict()` and returning the predictions in `y_pred`.

There are various model evaluation metrics we can use. However, since this is a simple binary text classification problem, we'll use `accuracy_score()` and `f1_score`. These both give high scores of over 97%, so the base model performs quite well, without more advanced processes such as cross validation or hyperparameter tuning.

```
y_pred = model.predict(X_test)

print('Accuracy:', accuracy_score(y_test, y_pred))
print('F1 score:', f1_score(y_test, y_pred, average="macro"))
```

```
Accuracy: 0.9796086508753862
F1 score: 0.9754309418775351
```

The `classification_report()` function is also worth running on the data. For each class, this effectively shows how good the model was at predicting each class. The closer to 1 the better the performance.

```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.94      0.99      0.97      4178
           1       1.00      0.98      0.99     10387

    accuracy                           0.98     14565
   macro avg       0.97      0.98      0.98     14565
weighted avg       0.98      0.98      0.98     14565
```

There are, of course, many ways we could further improve our model. We could try different preprocessing approaches, obtain or engineer additional features, utilise cross validation, model selection and hyperparameter tuning among others.

The results from our simple base model show that a basic, un-tuned model is easily capable of making good predictions, but some additional effort will likely see even better results.

Matt Clarke, Sunday, May 08, 2022