Multinomial Naive Bayes algorithm

Use scikit-learn to complete a popular text classification task (spam filtering) using Multinomial Naive Bayes

By Eda Kavlakoglu

Published 26 September 2023

The Naive Bayes classifier is a supervised machine learning algorithm, which is commonly applied in use cases involving recommendation systems, text classification, and sentiment analysis. Since it performs well with data sets with high dimensionality, it is a favored classifier for text classification in particular.

Naive Bayes (NB) is also a generative learning algorithm, which means that it models the distribution of data points for a given class or category. This probabilistic classifier is based off of Bayes' Theorem , meaning that this Bayesian classifier uses conditional probabilities and prior probabilities to calculate the posterior probabilities.

Naive Bayes classifiers work differently in that they operate under a couple of key assumptions, earning it the title of *naive*. It assumes that predictors in a Naive Bayes model are conditionally independent, or unrelated to any of the other feature in the model. It also assumes that all features contribute equally to the outcome.

While these assumptions are often violated in real-world scenarios (for example, a subsequent word in an e-mail is dependent upon the word that

In this tutorial, we'll use scikit-learn to walk through different types of Naive Bayes algorithms, focusing primarily on a popular text classification task (spam filtering) using Multinomial Naive Bayes.

Prerequisites

- Create an IBM Cloud® account
- Install scikit-learn

Steps

Step 1: Set up your environment

While there are a number of tools to choose from, we'll walk you through how to set up an IBM account to use a Jupyter notebook. Jupyter notebooks are widely used within data science to combine code, text, images, data visualizations to formulate a well-formed analysis.

1. Log in to watsonx.ai using your IBM Cloud account.

2. Create a watsonx.ai project.

   a. Click the hamburger menu at the top left of the screen, and then select **Projects > View all projects**. b. Click the **New project** button. c. Select **Create an empty project**. d. Enter a project name in the **Name** field. e. Select **Create**.

3. Create a Jupyter notebook.

code from this beginner tutorial to tackle a simple classification problem.

Step 2: Install and import relevant libraries

We'll need a few libraries for this tutorial. Make sure to import the ones below, and if they're not installed, you can resolve this with a quick `pip install`.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import nltk
import seaborn as sns
import re
import os, types

from sklearn.feature_extraction.text import CountVectorizer, TfidfVect
from sklearn.metrics import roc_auc_score, roc_curve, confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB

from nltk.stem import WordNetLemmatizer
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
nltk.download("punkt")
nltk.download('wordnet')
nltk.download('stopwords')
```

Show more

Step 3: Load the data

For this tutorial, we will be using a spam data set from the UCI Machine Learning Repository to walk through a classic spam filtering use case for Naive Bayes.

3. Upload this `.csv` file from your local system to your notebook in watsonx.ai.

4. Read the data in by selecting the **</>** icon in the upper right menu, and then selecting **Read data**.

5. Select **Upload a data file**.

6. Drag your data set over the prompt, **Drop data files here or browse for files to upload**. Within **Selected data**, select your data file (for example, SMSSpamCollection.csv) and load it as a pandas DataFrame.

7. Select **Insert code to cell** or the copy to clipboard icon to manually inject into your notebook.

Step 4: Conduct an exploratory data analysis

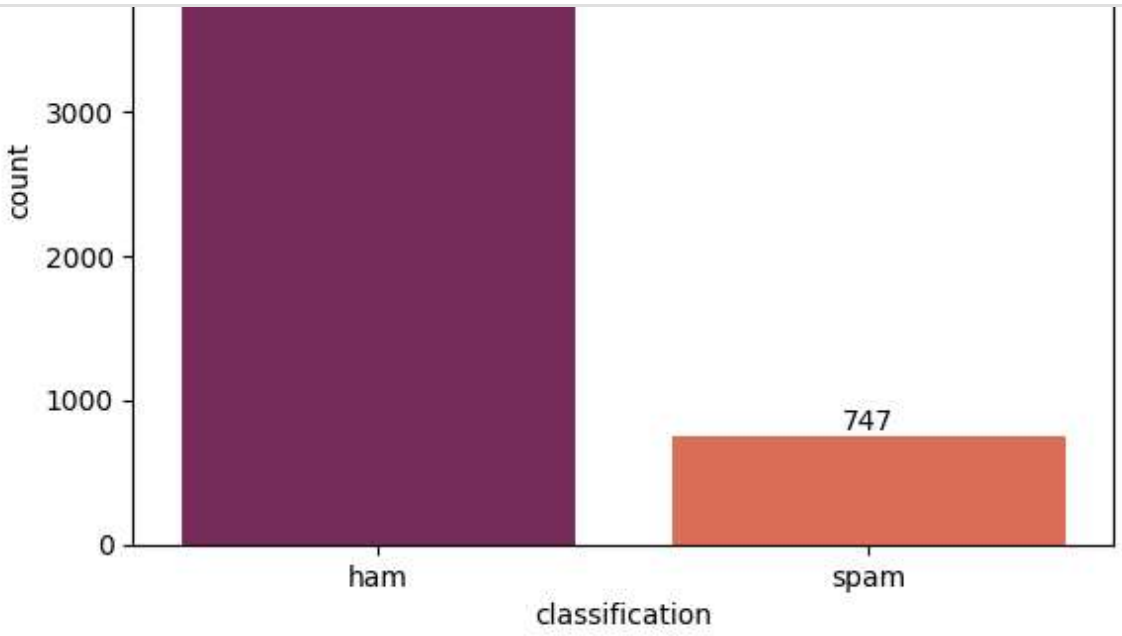
Before preprocessing, it’s always good to organize the data and examine it for any underlying issues, such as missing or duplicate data. Plotting the data can also help us to see if the data is balanced.

Let's start by recoding our class labels from their categorical form, such as "spam" and "ham" to a numerical format using 1's and 0's.

```
df['target'] = np.where(df['classification']=='ham',0, 1)
```

Show more

From there, we can examine and visualize the data with a few pieces of code.



While there is no missing data, there is duplicate data in this data set. Additionally, the plot indicates that the class distribution is uneven with spam representing only 13% of the data. So, we have an imbalanced dataset, which can be a concern as it can lead to overfitting. While it does not mean that the training data will overfit, it is good to be aware of this upfront in case we need to use over-sampling (also known as upsampling) or under-sampling (also known as downsampling) techniques. With this in mind, we'll drop the duplicate data and proceed with training our model on this imbalanced distribution first.

Step 5: Split your data

Next, we will split our data set into two groups, a training set and a test set. The training data will help us train our Naive Bayes model and our test data will help us to evaluate its performance. The test data set will be 30% of our initial data set, but you can adjust this by changing the value of the `test_size` parameter.

```
X_train, X_test, Y_train, Y_test = train_test_split(df_no_dup['text'],
                                                    df_no_dup['target'
                                                    test_size=0.3,
```

Step 6: Preprocess the data

After we split the data, we can start preprocessing it. This includes natural language processing tasks, such as tokenization, stop-word removal, stemming, and lemmatization.

Then, we will use a popular word embedding technique, called *bag-of-words*, to extract features from the text. This technique specifically calculates the frequency of words within a given document, which can help us classify documents, assuming that similar documents have similar content.

We can use scikit-learn's `CountVectorizer` or `TfidfVectorizer` to do the heavy lifting for us here. For the purposes of this tutorial, we will only show the code for `TFidVectorizer`, but you can find the full code in the notebook on GitHub .

```
def text_clean(text, method, rm_stop):
    text = re.sub(r"\n","",text)   #remove line breaks
    text = text.lower() #convert to lowercase
    text = re.sub(r"\d+","",text)   #remove digits and currencies
    text = re.sub(r'[\$\d+\d+\$]', "", text)
    text = re.sub(r'\d+[\.\/-]\d+[\.\/-]\d+', '', text)   #remove date
    text = re.sub(r'\d+[\.\/-]\d+[\.\/-]\d+', '', text)
    text = re.sub(r'\d+[\.\/-]\d+[\.\/-]\d+', '', text)
    text = re.sub(r'[^\x00-\x7f]',r' ',text)   #remove non-ascii
    text = re.sub(r'[^\w\s]','',text)   #remove punctuation
    text = re.sub(r'https?:\/\/.*[\r\n]*', '', text)   #remove hyperli

    #remove stop words
    if rm_stop == True:
        filtered_tokens = [word for word in word_tokenize(text) if not
        text = " ".join(filtered_tokens)

    #lemmatization: typically preferred over stemming
    if method == 'L':
        lemmer = WordNetLemmatizer()
        lemm_tokens = [lemmer.lemmatize(word) for word in word_tokeniz
        return " ".join(lemm_tokens)

    #stemming
    if method == 'S':
        porter = PorterStemmer()
        stem_tokens = [porter.stem(word) for word in word_tokenize(tex
        return " ".join(stem_tokens)
```

Step 7: Optimize and evaluate your Multinomial Naive Bayes model

Because we can process the data in a number of ways, we should model different versions of preprocessed data to understand which variation of data provides the optimal results within our model. For this use case, we will be using the most popular NB classifier, Multinomial Naive Bayes, as it is most commonly used for classification tasks, such as document classification. This variant is useful when using discrete data, such as frequency counts, and it is typically applied within natural language processing use cases.

After we apply the Multinomial Naive Bayes model to our different variants of training data, we can evaluate performance of the estimator using the testing data.

```
def transform_model_data_w_tfidf_vectorizer(preprocessed_text, Y_train
    #vectorize dataset
    tfidf = TfidfVectorizer()
    vectorized_data = tfidf.fit_transform(preprocessed_text)

    #define model
    model = MultinomialNB(alpha=0.1)
    model.fit(vectorized_data, Y_train)

    #evaluate model
    predictions = model.predict(tfidf.transform(X_test))

    accuracy = accuracy_score( Y_test, predictions)
    balanced_accuracy = balanced_accuracy_score(Y_test, predictions)
    precision = precision_score(Y_test, predictions)

    print("Accuracy:",round(100*accuracy,2),'%')
    print("Balanced accuracy:",round(100*balanced_accuracy,2),'%')
    print("Precision:", round(100*precision,2),'%')
    return predictions
```

Show more

could be a misleading metric for evaluation. Precision, on the other hand, will help us minimize the number of false positives (that is, the number of non-spam texts that end up in spam).

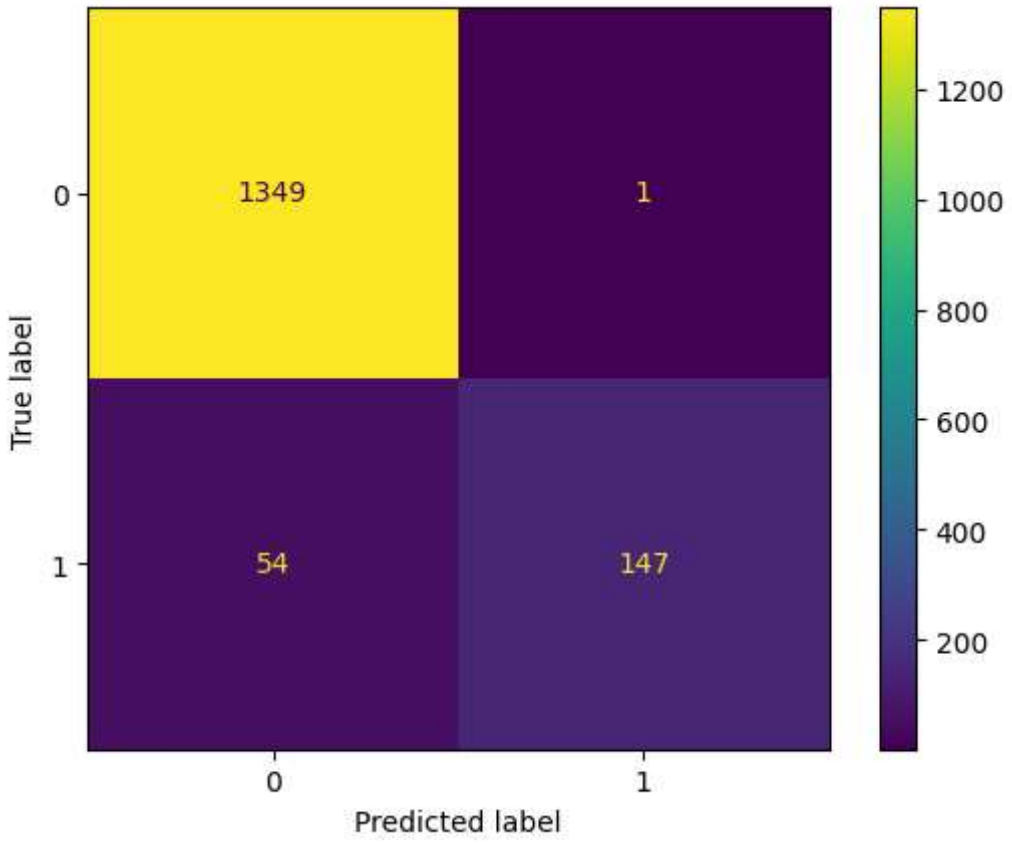
Accuracy score is defined as:

```
[true negative + true positive]/ [true negative + false positive
+ true positive + false negative]
```

Precision is defined as:

```
[true positive] / [true positive + false positive]
```

A confusion matrix can help us visualize these metrics more easily.



Other Naive Bayes classification models

Other Naive Bayes classifiers are used for other types of data distributions. To make predictions using these types of Naive Bayes models, you can use this code, but you should make sure that your data set is appropriate.

Gaussian Naive Bayes

As the name suggests, the data for each of the feature values is drawn from a Gaussian distribution, also known as a normal distribution. The code in this tutorial would be very similar for this approach, but would differ based on the use case. The code would also need to be updated to reflect the Gaussian Naive Bayes model from `scikit-learn`.

```
from sklearn.naive_bayes import GaussianNB
model = GaussianNB()
model.fit(X, y)
```

Show more

Bernoulli Naive Bayes

Similarly, to use Bernoulli Naive Bayes, the data for each of the feature values is drawn from a Bernoulli distribution. The relevant code for this model can be found here:

```
from sklearn.naive_bayes import BernoulliNB
model = BernoulliNB()
```

Summary and next steps

In this tutorial, you used a Naive Bayes classifier from `scikit-learn` to complete a popular text classification task, spam filtering.

Build an AI strategy for your business on one collaborative AI and data platform called IBM watsonx, which brings together new generative AI capabilities, powered by foundation models, and traditional machine learning into a powerful platform spanning the AI lifecycle. With watsonx.ai, you can train, validate, tune, and deploy models with ease and build AI applications in a fraction of the time with a fraction of the data.

Try watsonx.ai, the next-generation studio for AI builders. Explore more articles and tutorials about watsonx on IBM Developer.

To continue learning, we recommend exploring this content:

- Tutorial: Learn classification algorithms using Python and scikit-learn
- Tutorial: Learn regression algorithms using Python and scikit-learn
- What is linear regression?
- What is logistic regression?
- What is a decision tree?
- What are neural networks?

Interested in generative AI?

Learn generative AI skills →

Table of Contents

Implementing logistic regression from scratch in Python

Learn clustering algorithms using Python and scikit-learn

Tutorial

Learn classification algorithms using Python and scikit-learn

Tutorial

Learn regression algorithms using Python and scikit-learn