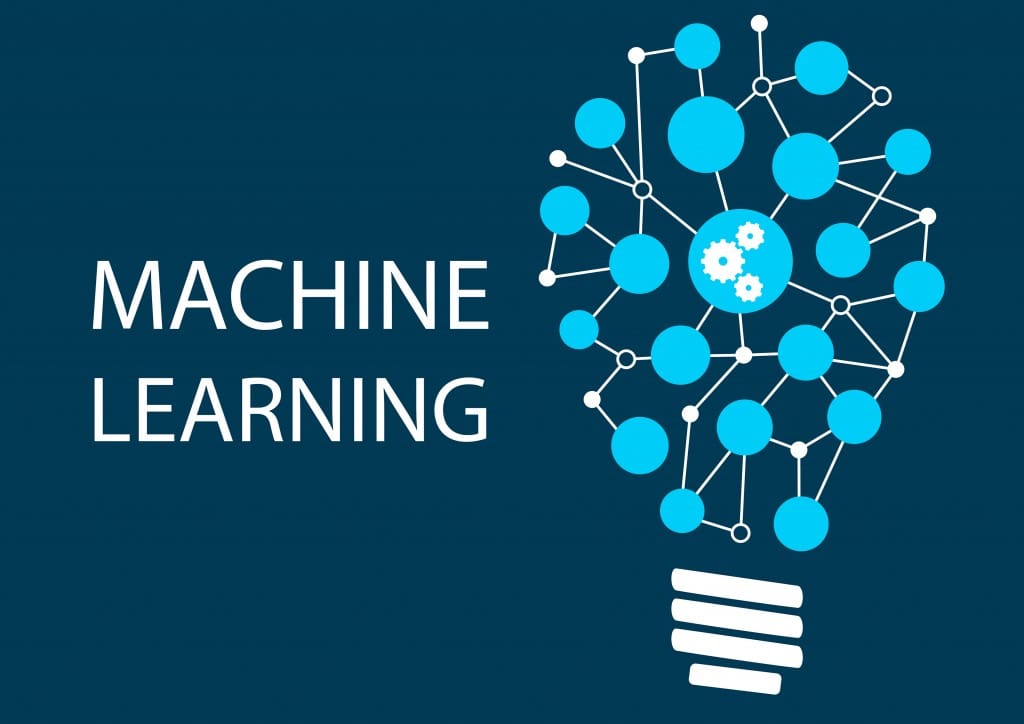
**BUILDING A SMARTER AI-POWERED SPAM CLASSIFIER-DEVELOPMENT PART 02**



**INTRODUCTION:**

Naive Bayes is a popular machine learning algorithm for building spam classifiers due to its simplicity and effectiveness, especially for text classification tasks. Here's how you can use Naive Bayes for a spam classifier:

1. Multinomial Naive Bayes:

* This variant of Naive Bayes is well-suited for text data, which is often represented as a bag of words with counts (such as TF-IDF or term frequencies).
* It assumes that the features (words) are conditionally independent, given the class labels (spam or non-spam).

2. Bernoulli Naive Bayes:

* Bernoulli Naive Bayes is used when the features represent binary data, such as the presence or absence of words in a document.
* It's suitable for situations where you have binary features, like whether a word is present in the text or not.

Here are the general steps to implement a spam classifier using Naive Bayes:

1.Data Preprocessing:

* Preprocess your text data by cleaning, tokenizing, and converting it into a bag of words or TF-IDF representation.

2.Data Splitting:

* Split your dataset into training and testing sets to evaluate the model's performance.

3. Training the Model:

* Use the Multinomial or Bernoulli Naive Bayes algorithm to train the model on the training data.
* Calculate the prior probabilities and likelihoods for each class (spam and non-spam).

4. Model Evaluation:

* Use the testing dataset to evaluate the model's performance.
* Calculate metrics like accuracy, precision, recall, F1-score, and ROC-AUC to assess the classifier's effectiveness.

5. Hyperparameter Tuning:

* You can fine-tune hyperparameters like the smoothing factor (alpha) for Naive Bayes to optimize the model's performance.

6. Threshold Adjustment:

* Experiment with different decision thresholds to balance precision and recall according to your project requirements.

7. Cross-Validation:

* Implement cross-validation to ensure the model's robustness and detect potential overfitting.

**IMPLEMENTATION:**

In [1] :

*# This Python 3 environment comes with many helpful analytics libraries installed*

*# It is defined by the kaggle/python docker image: https://github.com/kaggle/docker-python*

*# For example, here's several helpful packages to load in*

import numpy as np *# linear algebra*

import pandas as pd *# data processing, CSV file I/O (e.g. pd.read\_csv)*

import nltk

*# Input data files are available in the "../input/" directory.*

*# For example, running this (by clicking run or pressing Shift+Enter) will list the files in the input directory*

import os

print(os.listdir("../input"))

*# Any results you write to the current directory are saved as output.*

['spam.csv']

CHECKING THE LENGTH OF THE SMS

In [2]

import pandas

df\_sms = pd.read\_csv('../input/spam.csv',encoding='latin-1')

df\_sms.head()

Out[2]:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | V1 | V2 | Unnamed:2 | Unnamed:3 | Unnamed:4 |
| 0 | ham | Go until jurong point,crazy…Available only… | NaN | NaN | NaN |
| 1 | ham | Ok lar…  Joking wif u oni… | NaN | NaN | NaN |
| 2 | spam | Free entry in 2 a wkly comp to win FA Cup fins… | NaN | NaN | NaN |
| 3 | ham | U dun say so early hor… U c already then say… | NaN | NaN | NaN |
| 4 | ham | Nah I don’t think he goes usf,he lives aro… | NaN | NaN | NaN |

**Dropping the unwanted columns Unnamed:2, Unnamed: 3 and Unnamed:4**

**In [3] :**

df\_sms = df\_sms.drop(["Unnamed: 2", "Unnamed: 3", "Unnamed: 4"], axis=1)

df\_sms = df\_sms.rename(columns={"v1":"label", "v2":"sms"})

In [4]:

df\_sms.head()

out[4]:

|  |  |  |
| --- | --- | --- |
|  | label | Sms |
| 0 | ham | Go until jurong point,crazy…Available only… |
| 1 | ham | Ok lar…Joking wif u oni… |
| 2 | Spam | Free entry in 2 awkly comp to win FA Cup fins… |
| 3 | ham | U dun say so eaely hor… U c already then say… |
| 4 | Ham | Nah I don’t think he goes to usf,he lives aro |

In [5] :

*#Checking the maximum length of SMS*

print (len(df\_sms))

5572

In [6]:

linkcode

df\_sms.tail()

out[6] :

|  |  |  |
| --- | --- | --- |
|  | label | Sms |
| 5567 | Spam | This is the 2nd time we have tried 2 contact u… |
| 5568 | Ham | Will I\_b going to esplanade fr home? |
| 5569 | Ham | Pity,\*was in mood for that. So…any other s… |
| 5570 | Ham | The guy did some bitching but I acted like I’d… |
| 5571 | ham | Rofl. Its true to its name |

In [7] :

*#Number of observations in each label spam and ham*

df\_sms.label.value\_counts()

Out[7]:

ham 4825

spam 747

Name: label, dtype: int64

In [8]:

df\_sms.describe()

|  |  |  |
| --- | --- | --- |
|  | label | Sms |
| Count | 5572 | 5572 |
| Uinique | 2 | 5169 |
| Top | Ham | Sorry, I’ll call later |
| freq | 4825 | 30 |

In [9] :

df\_sms['length'] = df\_sms['sms'].apply(len)

df\_sms.head()

out[9] :

|  |  |  |  |
| --- | --- | --- | --- |
|  | label | sms | Length |
| 0 | Ham | Go until jurong point,crazy…Available only… | 111 |
| 1 | Ham | Ok lar…joking wif u oni… | 29 |
| 2 | spam | Free entry in 2 wkly comp to win FA Cup fina… | 155 |
| 3 | Ham | U dun say so early hor… U c already then say… | 49 |
| 4 | Ham | Nah I don’t think he goes to usf,he lives aro… | 61 |

In [10]:

import matplotlib.pyplot as plt

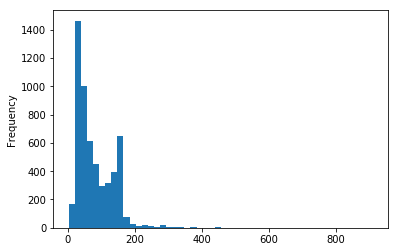
import seaborn as sns

%matplotlib inline

df\_sms['length'].plot(bins=50, kind='hist'

out[10]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x7fc56cbdbdd8>



In [11] :

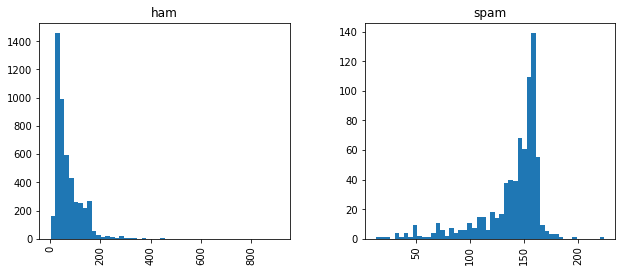
df\_sms.hist(column='length', by='label', bins=50,figsize=(10,4))

out[11]:

array([<matplotlib.axes.\_subplots.AxesSubplot object at 0x7fc56c87a208>,

<matplotlib.axes.\_subplots.AxesSubplot object at 0x7fc56c833c88>],

dtype=object)



In [12] :

df\_sms.loc[:,'label'] = df\_sms.label.map({'ham':0, 'spam':1})

print(df\_sms.shape)

df\_sms.head()

(5572, 3)

**Bag of Words Approach:**

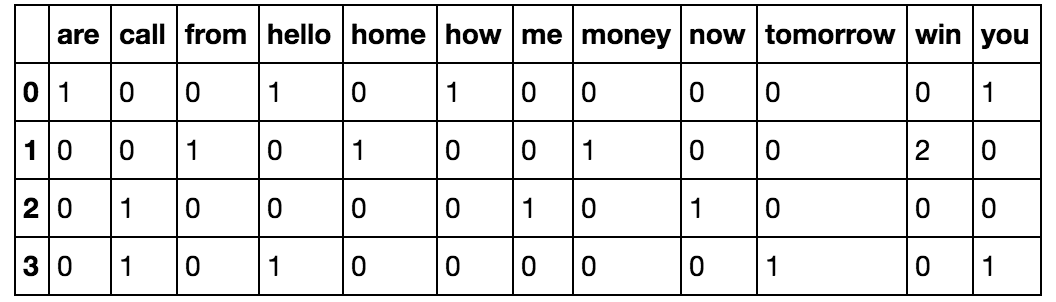
What we have here in our data set is a large collection of text data (5,572 rows of data). Most ML algorithms rely on numerical data to be fed into them as input, and email/sms messages are usually text heavy. We need a way to represent text data for machine learning algorithm and the bag-of-words model helps us to achieve that task. It is a way of extracting features from the text for use in machine learning algorithms. In this approach, we use the tokenized words for each observation and find out the frequency of each token. 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:



**Implementation of Bag of Words Approach**

**Step 1: Convert all strings to their lower case form.**

In [13]:

documents = ['Hello, how are you!',

'Win money, win from home.',

'Call me now.',

'Hello, Call hello you tomorrow?']

lower\_case\_documents = []

lower\_case\_documents = [d.lower() for d **in** documents]

print(lower\_case\_documents)

['hello, how are you!', 'win money, win from home.', 'call me now.', 'hello, call hello you tomorrow?']

**Step 2: Removing all punctuations**

In [14]:

sans\_punctuation\_documents = []

import string

for i **in** lower\_case\_documents:

sans\_punctuation\_documents.append(i.translate(str.maketrans("","", string.punctuation)))

sans\_punctuation\_documents

Out[14]:

['hello how are you',

'win money win from home',

'call me now',

'hello call hello you tomorrow']

Step 3: Tokenization

In [15]:

preprocessed\_documents = [[w for w **in** d.split()] for d **in** sans\_punctuation\_documents]

preprocessed\_documents

Out[15]:

[['hello', 'how', 'are', 'you'],

['win', 'money', 'win', 'from', 'home'],

['call', 'me', 'now'],

['hello', 'call', 'hello', 'you', 'tomorrow']]

Step 4: Count frequencies

In [16]:

frequency\_list = []

import pprint

from collections import Counter

frequency\_list = [Counter(d) for d **in** preprocessed\_documents]

pprint.pprint(frequency\_list)

[Counter({'hello': 1, 'how': 1, 'are': 1, 'you': 1}),

Counter({'win': 2, 'money': 1, 'from': 1, 'home': 1}),

Counter({'call': 1, 'me': 1, 'now': 1}),

Counter({'hello': 2, 'call': 1, 'you': 1, 'tomorrow': 1})]

**IMPLEMENTING BAG OF WORDS IN SCIKIT-LEARN**

''' Here we will look to create a frequency matrix on a smaller document set to make sure we understand how the document-term matrix generation happens. We have created a sample document set 'documents'. ''' documents = ['Hello, how are you!', 'Win money, win from home.', 'Call me now.', 'Hello, Call hello you tomorrow?']

In [17]:

linkcode

from sklearn.feature\_extraction.text import CountVectorizer

count\_vector = CountVectorizer()

**IMPLEMENTATION OF NAIVE BAYES MACHINE LEARNING ALGORITHM:**

I will use 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(alpha=1.0, class\_prior=None, fit\_prior=True)

predictions = naive\_bayes.predict(testing\_data)

TRAINING OUR MODEL

Training a Naive Bayes spam classifier involves several steps. Here's how to train the model using Python and scikit-learn:

python

# Import necessary libraries

from sklearn.feature\_extraction.text import CountVectorizer

from sklearn.naive\_bayes import MultinomialNB

from sklearn.model\_selection import train\_test\_split

# Sample dataset (replace with your spam and non-spam messages)

messages = [

("Free entry to win a million dollars", "spam"),

("Hello, how are you today?", "not spam"),

# Add more data here

]

# Split messages into text and labels

texts, labels = zip(\*messages)

# Create a feature vector using CountVectorizer

vectorizer = CountVectorizer()

X = vectorizer.fit\_transform(texts)

# Split data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, labels, test\_size=0.2, random\_state=42)

# Initialize and train the Multinomial Naive Bayes classifier

classifier = MultinomialNB()

**classifier.fit(X\_train, y\_train)**

**In this code:**

1. Import the necessary libraries, including scikit-learn.

2. Define your dataset in the `messages` list, where each entry is a tuple containing a message and its corresponding label ("spam" or "not spam"). Replace this sample data with your actual dataset.

3. Split the data into text and labels.

4. Create a feature vector using CountVectorizer. This step converts the text data into a numerical representation.

5. Split the data into training and testing sets using `train\_test\_split` to evaluate the model's performance.

6. Initialize a Multinomial Naive Bayes classifier and train it using the training data.

Now, your Naive Bayes spam classifier is trained and ready for making predictions. You can use it to classify new messages or evaluate its performance on the testing set.

**EVALUATING OUR MODEL:**

Now that we have made predictions on our test set, our next goal is to evaluate how well our model is doing. There are various mechanisms for doing so, but first let's do quick recap of them.

**Accuracy** measures how often the classifier makes the correct prediction. It’s the ratio of the number of correct predictions to the total number of predictions (the number of test data points).

**Precision** tells us what proportion of messages we classified as spam, actually were spam. It is a ratio of true positives(words classified as spam, and which are actually spam) to all positives(all words classified as spam, irrespective of whether that was the correct classification), in other words it is the ratio of

**[True Positives/(True Positives + False Positives)]**

**Recall(sensitivity)** tells us what proportion of messages that actually were spam were classified by us as spam. It is a ratio of true positives(words classified as spam, and which are actually spam) to all the words that were actually spam, in other words it is the ratio of

**[True Positives/(True Positives + False Negatives)]**

For classification problems that are skewed in their classification distributions like in our case, for example if we had a 100 text messages and only 2 were spam and the rest 98 weren't, accuracy by itself is not a very good metric. We could classify 90 messages as not spam(including the 2 that were spam but we classify them as not spam, hence they would be false negatives) and 10 as spam(all 10 false positives) and still get a reasonably good accuracy score. For such cases, precision and recall come in very handy. These two metrics can be combined to get the F1 score, which is weighted average of the precision and recall scores. This score can range from 0 to 1, with 1 being the best possible F1 score.

We will be using all 4 metrics to make sure our model does well. For all 4 metrics whose values can range from 0 to 1, having a score as close to 1 as possible is a good indicator of how well our model is doing.

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: 0.9847533632286996

Precision score: 0.9420289855072463

Recall score: 0.935251798561151

F1 score: 0.9386281588447652

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.

**CONCLUSION:**

In conclusion, the Naive Bayes algorithm is a highly effective and popular choice for spam classification. Its simplicity, efficiency, and ability to handle high-dimensional data make it well-suited for this task. By utilizing probabilistic calculations and assuming independence between features (hence "naive"), Naive Bayes can accurately identify spam messages by comparing the likelihood of a message being spam or not. While it may not always achieve perfect results, it remains a robust and widely-used solution for spam classification, and its performance can be further enhanced with feature engineering and data preprocessing.