<u>BUILDING A SMARTER AI-POWERED SPAM CLASSIFIER</u>

Team member

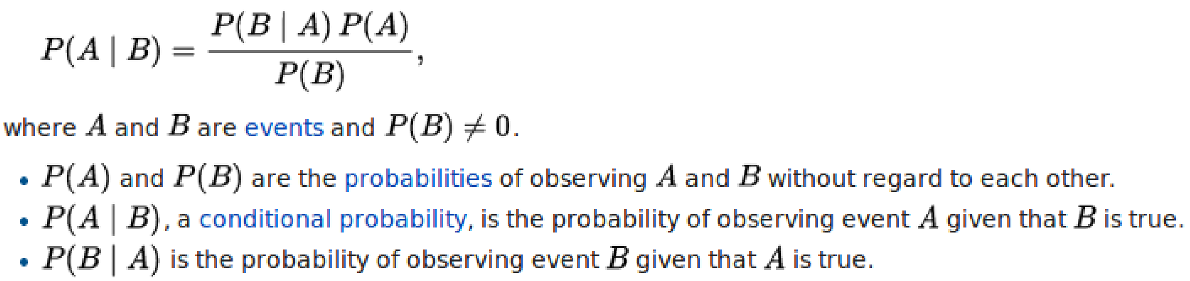
621421104021-Harivishwa.s.p

Phase 3 project submission

Project title: AI Powered Spam Classifier

Introduction:

We all face the problem of spams in our inboxes. Let’s build a spam classifier program in python which can tell whether a given message is spam or not! We can do this by using a simple, yet powerful theorem from probability theory called Baye’s Theorem. It is mathematically expressed as



$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)},$$

where A and B are events and $P(B) \neq 0$.

- $P(A)$ and $P(B)$ are the probabilities of observing A and B without regard to each other.
- $P(A \mid B)$, a conditional probability, is the probability of observing event A given that B is true.
- $P(B \mid A)$ is the probability of observing event B given that A is true.

Baye’s Theorem

Problem Statement

We have a message m = (w*1*, w*2*, ... , w*n*), where (w*1*, w*2*, ... , w*n*) is a set of unique words contained in the message. We need to find

$$P(spam|w1 \cap w2 \cap ... \cap wn) = \frac{P(w1 \cap w2 \cap ... \cap wn|spam).\, P(spam)}{P(w1 \cap w2 \cap ... \cap wn)}$$

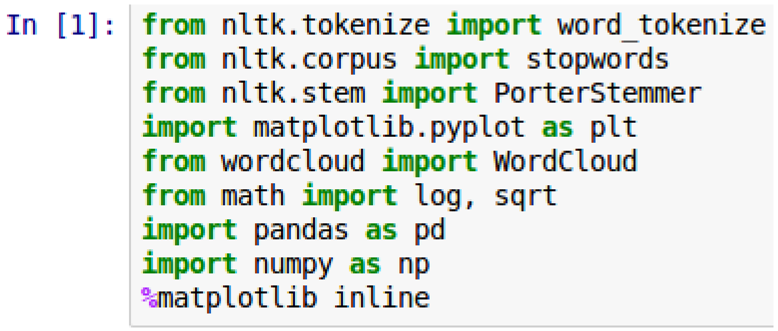
If we assume that occurrence of a word are independent of all other words, we can simplify the above expression to

$$\frac{P(w1|spam).P(w2|spam)\ldots P(wn|spam).P(spam)}{P(w1).P(w2)..P(wn)}$$

In order to classify we have to determine which is greater

$$P(spam|w1 \cap w2 \cap \ldots \cap wn) \ versus \ P(\sim spam|w1 \cap w2 \cap \ldots \cap wn)$$

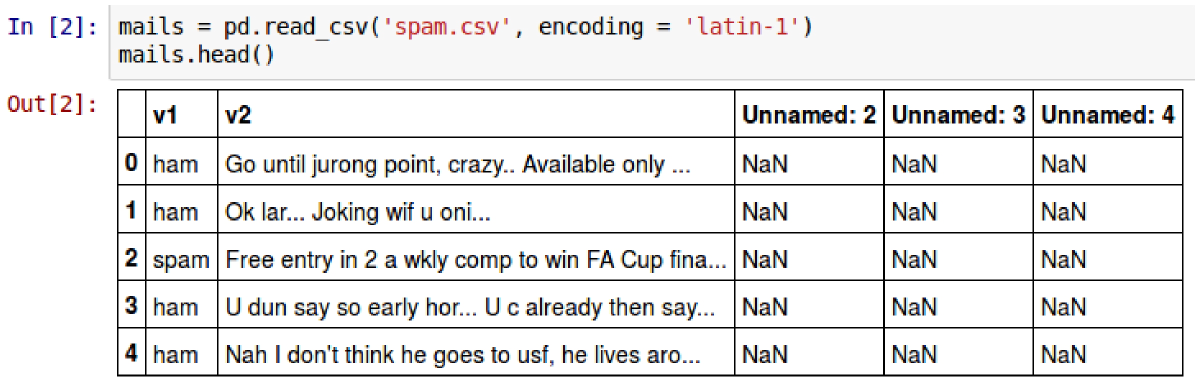
1. Loading dependencies



```
In [1]: from nltk.tokenize import word_tokenize
        from nltk.corpus import stopwords
        from nltk.stem import PorterStemmer
        import matplotlib.pyplot as plt
        from wordcloud import WordCloud
        from math import log, sqrt
        import pandas as pd
        import numpy as np
        %matplotlib inline
```

We are going to make use of NLTK for processing the messages, WordCloud and matplotlib for visualization and pandas for loading data, NumPy for generating random probabilities for train-test split.

2. Loading Data

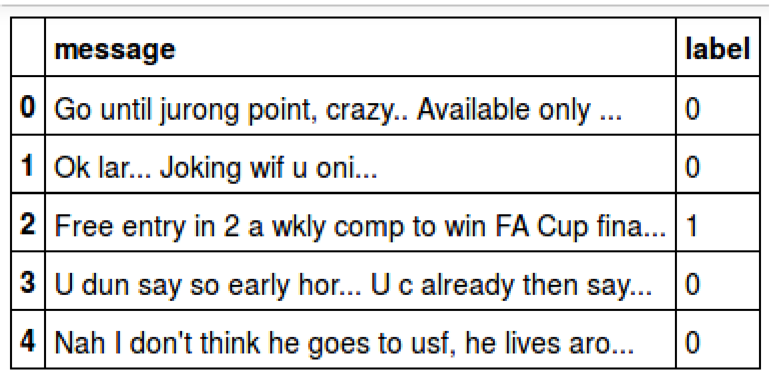


```
In [2]: mails = pd.read_csv('spam.csv', encoding = 'latin-1')
        mails.head()
```

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 fina...	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 to usf, he lives aro...	NaN	NaN	NaN

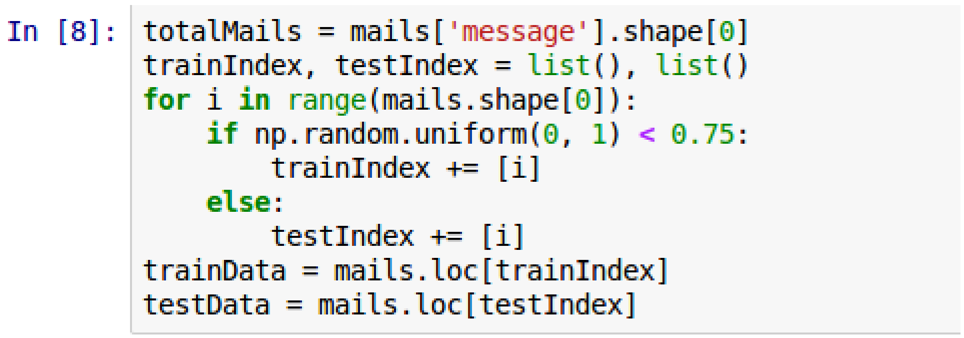
We do not require the columns ‘Unnamed: 2’, ‘Unnamed: 3’ and ‘Unnamed: 4’, so we remove them. We rename the column ‘v1’ as ‘label’ and ‘v2’ as ‘message’. ‘ham’ is replaced by 0 and ‘spam’ is replaced by 1 in the ‘label’ column. Finally we obtain the following dataframe.



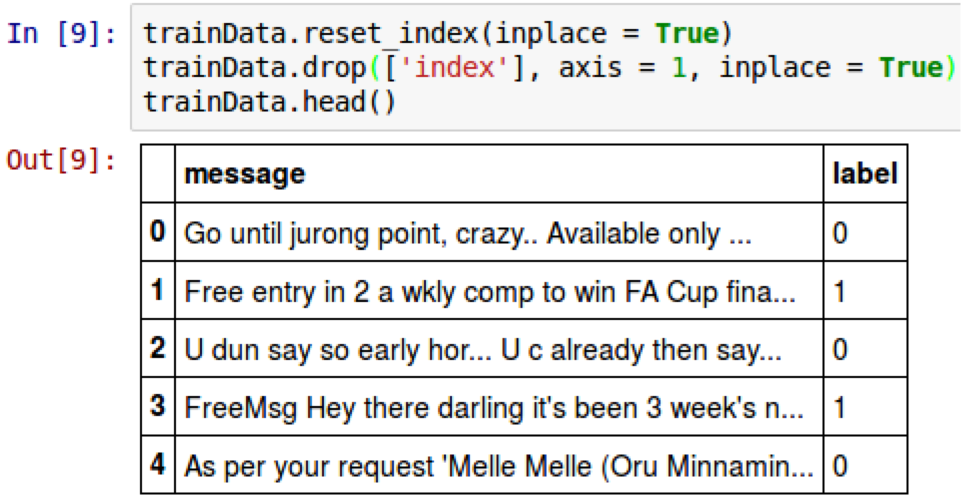
	message	label
0	Go until jurong point, crazy.. Available only ...	0
1	Ok lar... Joking wif u oni...	0
2	Free entry in 2 a wkly comp to win FA Cup fina...	1
3	U dun say so early hor... U c already then say...	0
4	Nah I don't think he goes to usf, he lives aro...	0

3. Train-Test Split

To test our model we should split the data into train dataset and test dataset. We shall use the train dataset t0 train the model and then it will be tested on the test dataset. We shall use 75% of the dataset as train dataset and the rest as test dataset. Selection of this 75% of the data is uniformly random.



```
In [8]: totalMails = mails['message'].shape[0]
        trainIndex, testIndex = list(), list()
        for i in range(mails.shape[0]):
            if np.random.uniform(0, 1) < 0.75:
                trainIndex += [i]
            else:
                testIndex += [i]
        trainData = mails.loc[trainIndex]
        testData = mails.loc[testIndex]
```



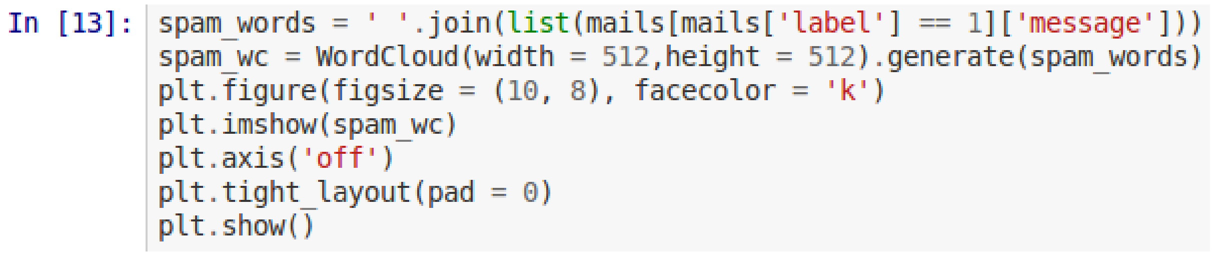
```
In [9]: trainData.reset_index(inplace = True)
        trainData.drop(['index'], axis = 1, inplace = True)
        trainData.head()
```

Out[9]:

	message	label
0	Go until jurong point, crazy.. Available only ...	0
1	Free entry in 2 a wkly comp to win FA Cup fina...	1
2	U dun say so early hor... U c already then say...	0
3	FreeMsg Hey there darling it's been 3 week's n...	1
4	As per your request 'Melle Melle (Oru Minnamin...	0

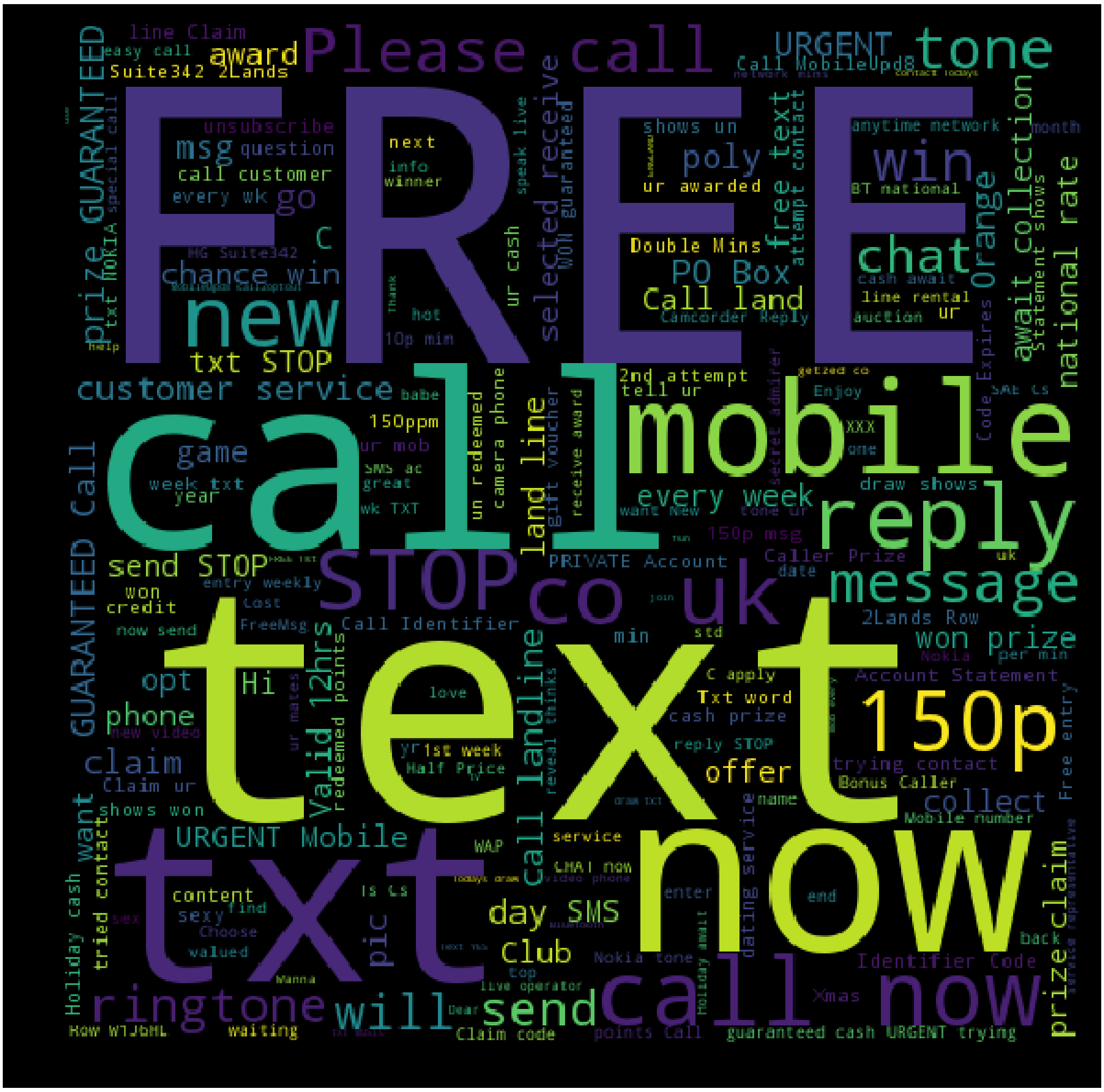
4. Visualizing data

Let us see which are the most repeated words in the spam messages! We are going to use WordCloud library for this purpose.



```
In [13]: spam_words = ' '.join(list(mails[mails['label'] == 1]['message']))
         spam_wc = WordCloud(width = 512,height = 512).generate(spam_words)
         plt.figure(figsize = (10, 8), facecolor = 'k')
         plt.imshow(spam_wc)
         plt.axis('off')
         plt.tight_layout(pad = 0)
         plt.show()
```

This results in the following



As expected, these messages mostly contain the words like 'FREE', 'call', 'text', 'ringtone', 'prize claim' etc.

Similarly the wordcloud of ham messages is as follows:

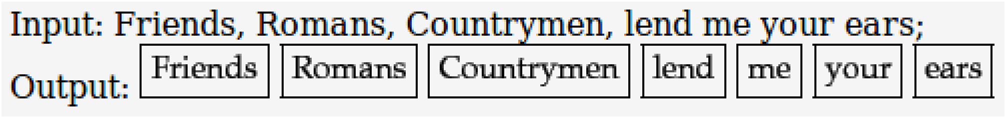
will
love
one
go
lor
call
much
need
got
time
now
come
know
day
get
ok
going
today
think
see
want
Sorry
still
good
tell
home
thing
back
take
work
Thank
make
dont
say
Hey
really
ur
well
later
send
dear
meet
Oh
hope
ask
please
life
text
leave
finish
friend
Yeah
tomorrow
something
also
phone
Happy
gonna
babe
place
give
great
already
number
free
fine
night
way
Lol
sleep
late
Yes
said
find
Hi
wat

5. Training the model

We are going to implement two techniques: Bag of words and TF-IDF. I shall explain them one by one. Let us first start off with Bag of words.

Preprocessing: Before starting with training we must preprocess the messages. First of all, we shall make all the character lowercase. This is because ‘free’ and ‘FREE’ mean the same and we do not want to treat them as two different words.

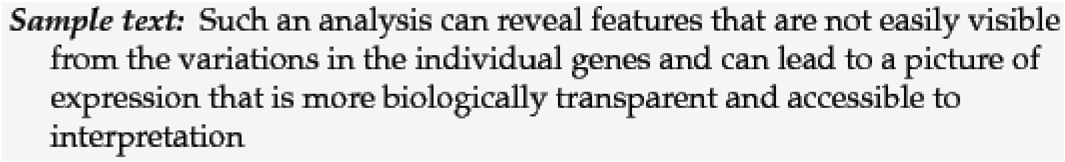
Then we tokenize each message in the dataset. Tokenization is the task of splitting up a message into pieces and throwing away the punctuation characters. For eg.:



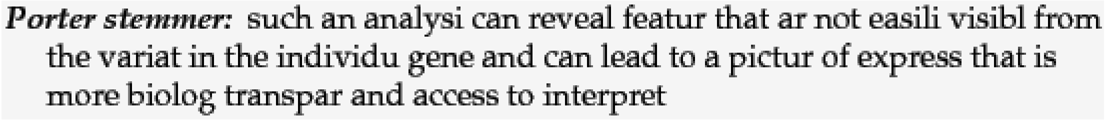
Input: Friends, Romans, Countrymen, lend me your ears;

Output: Friends | Romans | Countrymen | lend | me | your | ears

The words like ‘go’, ‘goes’, ‘going’ indicate the same activity. We can replace all these words by a single word ‘go’. This is called stemming. We are going to use Porter Stemmer, which is a famous stemming algorithm.



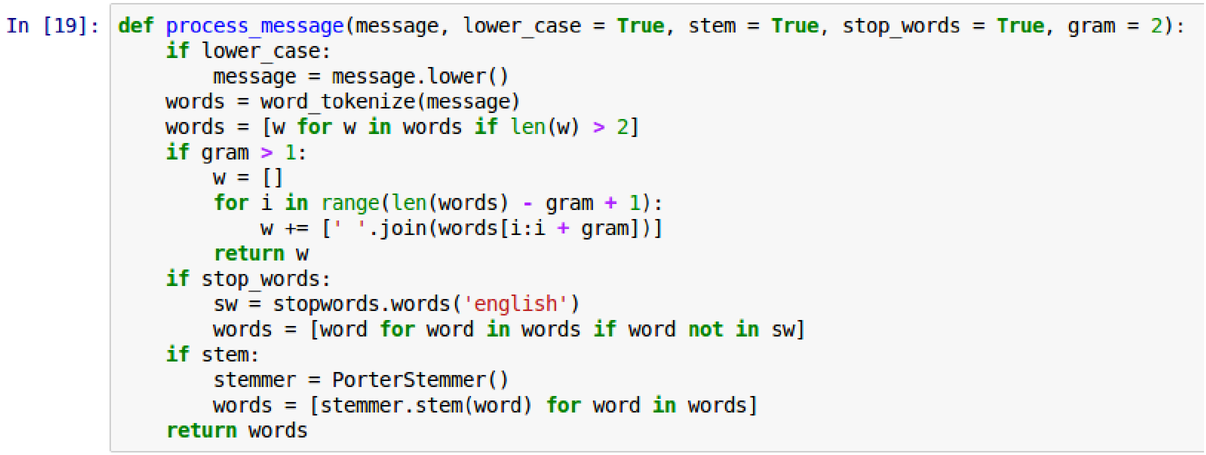
Sample text: Such an analysis can reveal features that are not easily visible from the variations in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpretation



Porter stemmer: such an analysi can reveal featur that ar not easili visibl from the variat in the individu gene and can lead to a pictur of express that is more biolog transpar and access to interpret

We then move on to remove the stop words. Stop words are those words which occur extremely frequently in any text. For example words like ‘the’, ‘a’, ‘an’, ‘is’, ‘to’ etc. These words do not give us any information about the content of the text. Thus it should not matter if we remove these words for the text.

Optional: You can also use n-grams to improve the accuracy. As of now, we only dealt with 1 word. But when two words are together the meaning totally changes. For example, ‘good’ and ‘not good’ are opposite in meaning. Suppose a text contains ‘not good’, it is better to consider ‘not good’ as one token rather than ‘not’ and ‘good’. Therefore, sometimes accuracy is improved when we split the text into tokens of two (or more) words than only word.



In [19]:

```python
def process_message(message, lower_case = True, stem = True, stop_words = True, gram = 2):
    if lower_case:
        message = message.lower()
    words = word_tokenize(message)
    words = [w for w in words if len(w) > 2]
    if gram > 1:
        w = []
        for i in range(len(words) - gram + 1):
            w += [' '.join(words[i:i + gram])]
        return w
    if stop_words:
        sw = stopwords.words('english')
        words = [word for word in words if word not in sw]
    if stem:
        stemmer = PorterStemmer()
        words = [stemmer.stem(word) for word in words]
    return words
```

Bag of Words: In Bag of words model we find the ‘term frequency’, i.e. number of occurrences of each word in the dataset. Thus for word w,

$$P(w) = \frac{Total\ number\ of\ occurrences\ of\ w\ in\ dataset}{Total\ number\ of\ words\ in\ dataset}$$

and

$$P(w|spam) = \frac{Total\ number\ of\ occurrences\ of\ w\ in\ spam\ messages}{Total\ number\ of\ words\ in\ spam\ messages}$$

TF-IDF: TF-IDF stands for Term Frequency-Inverse Document Frequency. In addition to Term Frequency we compute Inverse document frequency.

$$IDF(w) = \log \frac{Total\ number\ of\ messages}{Total\ number\ of\ messages\ containing\ w}$$

For example, there are two messages in the dataset. ‘hello world’ and ‘hello foo bar’. TF(‘hello’) is 2. IDF(‘hello’) is log(2/2). If a word occurs a lot, it means that the word gives less information.

In this model each word has a score, which is TF(w)*IDF(w). Probability of each word is counted as:

$$P(w) = \frac{TF(w) * IDF(w)}{\sum_{\forall\ words\ x\ \in\ train\ dataset} TF(x) * IDF(x)}$$

$$P(w|spam) = \frac{TF(w|spam) * IDF(w)}{\sum_{\forall\ words\ x\ \in\ train\ dataset} TF(x|spam) * IDF(x)}$$

Additive Smoothing:So what if we encounter a word in test dataset which is not part of train dataset? In that case P(w) will be 0, which will make the P(spam|w) undefined (since we would have to divide by P(w) which is 0. Remember the formula?). To tackle this issue we introduce additive smoothing. In additive smoothing we add a number alpha to the numerator and add alpha times number of classes over which the probability is found in the denominator.

$$P(w|spam) = \frac{TF(w|spam)\ +\ \alpha}{\sum_{\forall\ words\ x\ \in\ spam\ in\ train\ dataset} TF(x)\ +\ \alpha \sum_{\forall\ words\ x\ \in\ spam\ in\ train\ dataset} 1}$$

When using TF-IDF

$$P(w|spam) = \frac{TF(w|spam) * IDF(w)\ +\ \alpha}{\sum_{\forall\ words\ x\ \in\ train\ dataset} TF(x) * IDF(x)\ +\ \alpha \sum_{\forall\ words\ x\ \in\ spam\ in\ train\ dataset} 1}$$

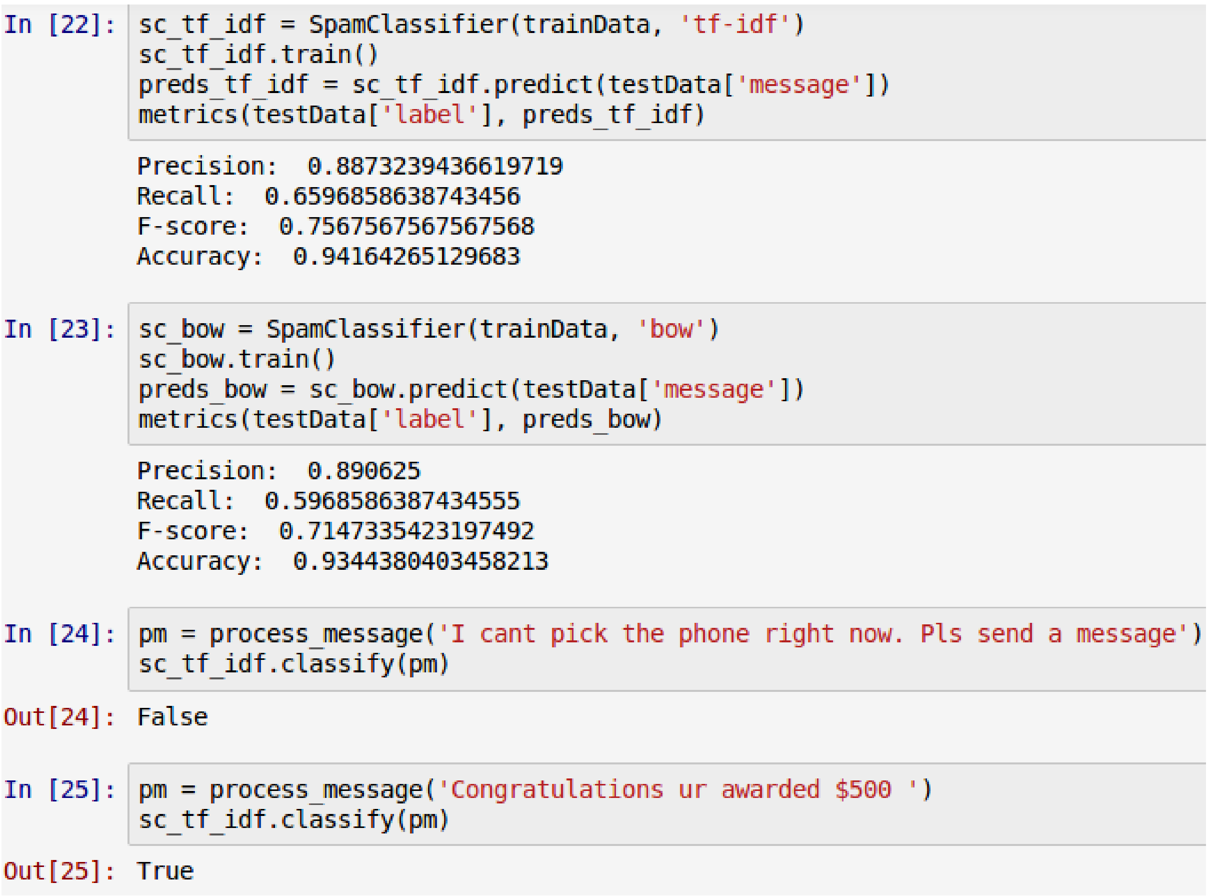
This is done so that the least probability of any word now should be a finite number. Addition in the denominator is to make the resultant sum of all the probabilities of words in the spam emails as 1.

When alpha = 1, it is called Laplace smoothing.

6. Classification

For classifying a given message, first we preprocess it. For each word w in the processed messaged we find a product of P(w|spam). If w does not exist in the train dataset we take TF(w) as 0 and find P(w|spam) using above formula. We multiply this product with P(spam) The resultant product is the P(spam|message). Similarly, we find P(ham|message). Whichever probability among these two is greater, the corresponding tag (spam or ham) is assigned to the input message. Note than we are not dividing by P(w) as given in the formula. This is because both the numbers will be divided by that and it would not affect the comparison between the two.

7. Final result



```
In [22]: sc_tf_idf = SpamClassifier(trainData, 'tf-idf')
         sc_tf_idf.train()
         preds_tf_idf = sc_tf_idf.predict(testData['message'])
         metrics(testData['label'], preds_tf_idf)

         Precision:  0.8873239436619719
         Recall:  0.6596858638743456
         F-score:  0.7567567567567568
         Accuracy:  0.94164265129683

In [23]: sc_bow = SpamClassifier(trainData, 'bow')
         sc_bow.train()
         preds_bow = sc_bow.predict(testData['message'])
         metrics(testData['label'], preds_bow)

         Precision:  0.890625
         Recall:  0.5968586387434555
         F-score:  0.7147335423197492
         Accuracy:  0.9344380403458213

In [24]: pm = process_message('I cant pick the phone right now. Pls send a message')
         sc_tf_idf.classify(pm)

Out[24]: False

In [25]: pm = process_message('Congratulations ur awarded $500 ')
         sc_tf_idf.classify(pm)

Out[25]: True
```

THANK YOU!!