Machine Learning Nanodegree Capstone Project

Spam Detection using Naive Bayes and Support Vector Machines

By

Sylvester Ranjith Francis

1.Introduction:

With the rapid growth of online information, text categorization has become one of the key techniques for handling and organizing text data. Text categorization techniques are used to classify news stories, to find interesting information on the WWW, and to guide a user's search through hypertext. Since building text classifiers by hand is difficult and time-consuming, it is advantageous to learn classifiers from examples. In this project we try to build such a Text classifier using Naive Bayes and Support Vector Machines to classify the incoming text messages as either Spam/Not Spam (Ham).

My reason to pick this problem as a Capstone Project:

My motives to picking this problem as a capstone project was due to the fact that Text classification has always intrigued me. I wanted to try something along the same topic .
A Spam/Ham Classifier has been implemented already by many companies but I wanted to try implementing one myself by utilising all what I learnt during the course of this nanodegree. I believe that I have successfully implemented a prediction engine that classifies Ham/Spam in a very accurate way.

My references are stated below:

http://airccse.org/journal/jcsit/0211ijcsit12.pdf
https://www.cs.cmu.edu/~schneide/tut5/node42.html

The below figure explains the working of the Text Classifier.

A basic workflow of this project is as follows

- Step 1:Initial exploration of the dataset is performed
- Step 2:Once the data exploration was performed,We proceed to convert the raw messages into vectors.
- Step 3:The mapping was not 1 to 1 hence the **Bag of Words** approach is used where each unique word in a text is represented by a number
- Step 4: Converting the text into a vector involves three steps namely,Counting the frequency of a word occurring in each message (term frequency),Weighting the counts, so that frequent tokens get lower weight (inverse document frequency),Normalizing the vectors to unit length, to abstract from the original text length (L2 norm)
- Reference : http://scikit-learn.org/stable/modules/feature_extraction.html#the-bag-of-words-representation
- Step 5: The model has to trained using a naive bayes classifier to check if our model works
- Step 6: Split the data into testing and training data using Sklearn train_test_split
- Step 7:The testing data is fed into the designed Naive bayes classifier,to find out that the prediction engine gave an accuracy of 95%
- Step 8: To improve the performance of the prediction engine,I tried designing a Support Vector Machine after many different trials and errors to improve the accuracy
- Step 9: The resulting Support Vector Machine gave an accuracy of 98% which is a huge improvement compared to the Naive bayes classifier

- The result being that we were able to classify the messages successfully as spam and ham with an accuracy of 98%

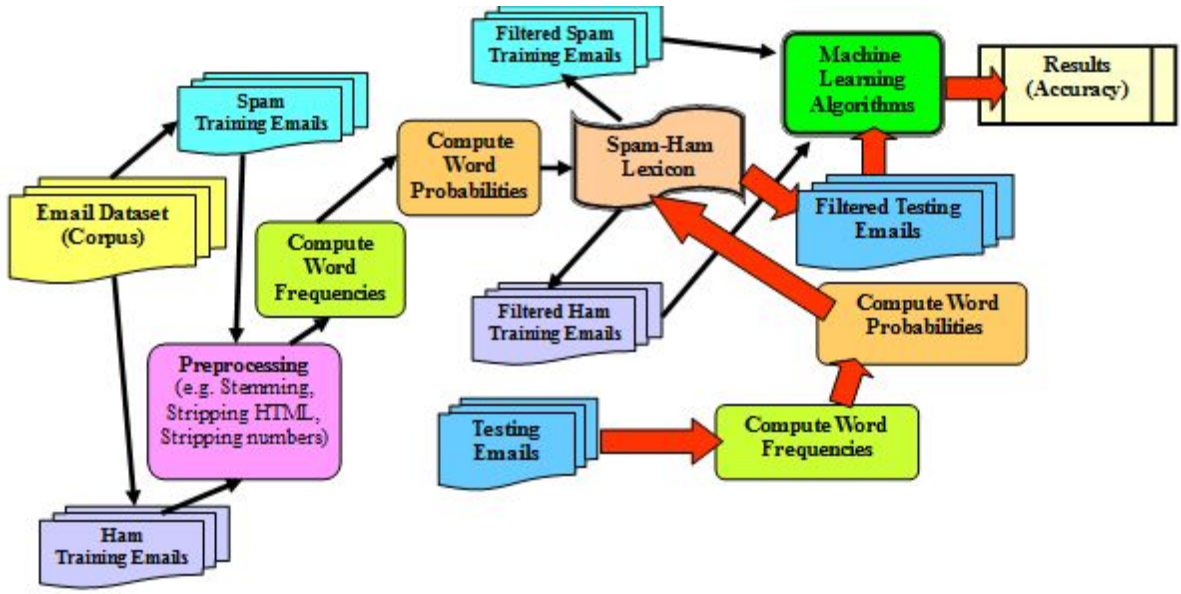


Fig 1 :Basic block diagram.

This block diagram will be explained in detail further as we proceed to the workings of the prediction engine.

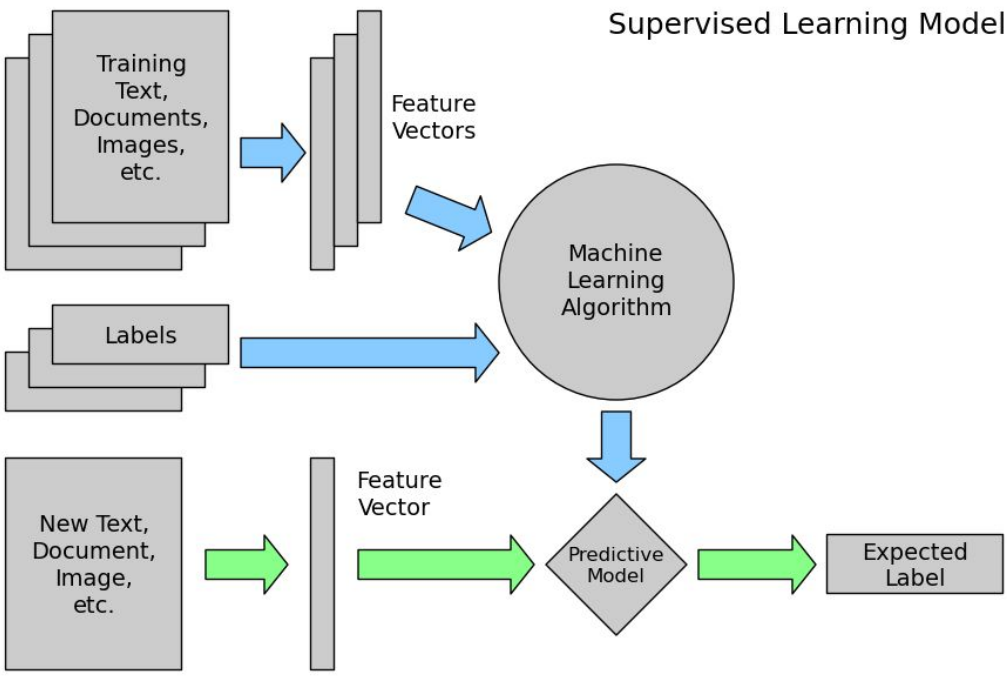
2.Dataset:

In this study, we have used the publicly available text corpus dataset of SMS spam messages found in the UCI Machine Learning Repository There are 5574 total messages in this dataset where 4459 is categorized for training and remaining 1115 messages is categorized for test phases. Figure 2 shows the first 10 messages of the text corpus dataset

```
In [11]: #printing the first 10 messages of the SMS CORPUS
         for message_no, message in enumerate(messages[:10]):
             print message_no, message
```

```
0 ham   Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine there got amore wa
t...
1 ham   Ok lar... Joking wif u oni...
2 spam  Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to 87121 to receive entry questio
n(std txt rate)T&C's apply 08452810075over18's
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 around here though
5 spam  FreeMsg Hey there darling it's been 3 week's now and no word back! I'd like some fun you up for it still? Tb
 ok! XxX std chgs to send, £1.50 to rcv
6 ham   Even my brother is not like to speak with me. They treat me like aids patent.
7 ham   As per your request 'Melle Melle (Oru Minnaminunginte Nurungu Vettam)' has been set as your callertune for al
l Callers. Press *9 to copy your friends Callertune
8 spam  WINNER!! As a valued network customer you have been selected to receivea £900 prize reward! To claim call 090
61701461. Claim code KL341. Valid 12 hours only.
9 spam  Had your mobile 11 months or more? U R entitled to Update to the latest colour mobiles with camera for Free!
 Call The Mobile Update Co FREE on 08002986030
```

We can observe that these are Tab Separated Values where the first column is a label saying whether the given message is a normal message ("ham") or "spam". The second column is the message itself. This corpus will be our labeled training set. Using these ham/spam examples, we'll train a machine learning model to learn to discriminate between ham/spam automatically. Then, with a trained model, we'll be able to classify arbitrary unlabeled messages as ham or spam.



2.1 Data Exploration:

In this section we try to understand the ways of classifying the data on different parameters to classify the data better.By understanding if there are frequent patterns in the different messages such as “Length of a message”,”Addressed to”, “Sender info”,“Content of a message”. We try to correlate these different parameters to see if

there are any specific patterns in a message which may lead it to be classified under the label of spam.

```
In [13]: #Aggregating Statistics
         messages.groupby('label').describe()
```

Out[13]:

		message
label		
ham	**count**	4827
	unique	4518
	top	Sorry, I'll call later
	freq	30
spam	**count**	747
	unique	653
	top	Please call our customer service representativ...
	freq	4

By grouping the data by the label,We observe that there are 4827 messages which are classified as “Ham” and number of messages that were unique are 4518.The most repeated messages were with the text “Sorry,I’ll call later”.Similarly there are 747 “Spam” messages out of which 653 of them were unique and the frequently repeated text was “Please call our customer service representative...”. We don't necessarily infer much by classifying them by labels and describing it.

Next we try to see if we can infer anything about classifying the messages based on the length of the text

```
In [14]: #length of messages
         messages['length'] = messages['message'].map(lambda text: len(text))
         print messages.head()
```

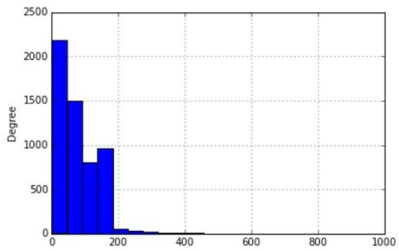
```
  label                                            message  length
0   ham  Go until jurong point, crazy.. Available only ...     111
1   ham                      Ok lar... Joking wif u oni...      29
2  spam  Free entry in 2 a wkly comp to win FA Cup fina...     155
3   ham  U dun say so early hor... U c already then say...      49
4   ham  Nah I don't think he goes to usf, he lives aro...      61
```

```
In [16]: messages.length.describe()
```

```
Out[16]: count    5574.000000
         mean       80.604593
         std        59.919970
         min         2.000000
         25%        36.000000
         50%        62.000000
         75%       122.000000
         max       910.000000
         Name: length, dtype: float64
```

The longest message is 910 characters long and it is found that is a ham message and not spam. Representing them graphically in figure 2.1. We infer that spam messages generally have a lesser character count than the ham messages.

```
In [7]: messages.length.plot(bins=20, kind='hist')
Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x1174a54d0>
```




```
In [8]: messages.length.describe()
Out[8]: count    5574.000000
        mean       80.604593
        std        59.919970
        min         2.000000
        25%        36.000000
        50%        62.000000
        75%       122.000000
        max       910.000000
        Name: length, dtype: float64
```

Fig 2.1 Ham vs Spam (Frequency of words)

```
In [9]: print list(messages.message[messages.length > 900])
["For me the love should start with attraction.i should feel that I need her every time around me.she should be the first thing
 which comes in my thoughts.I would start the day and end it with her.she should be there every time I dream.love will be then
 when my every breath has her name.my life should happen around her.my life will be named to her.I would cry for her.will give
 all my happiness and take all her sorrows.I will be ready to fight with anyone for her.I will be in love when I will be doing
 the craziest things for her.love will be when I don't have to proove anyone that my girl is the most beautiful lady on the who
le planet.I will always be singing praises for her.love will be when I start up making chicken curry and end up makiing sambar.
life will be the most beautiful then.will get every morning and thank god for the day because she is with me.I would like to sa
y a lot..will tell later.."]
```

Fig. Longest message

So through this initial data exploration we find out that “length of the messages” is a factor by which we can classify the messages.

In section 3 let us get into detail about the working of the classifier and how and why two different approaches were used to implement the classifier

3. Implementation :

There are two ways that the Ham/Spam classifier has been implemented in this project namely

1. Naive bayes classification
2. Using Support Vector Machines

We shall discuss in detail about these two approaches in this section

3.1 Data Preprocessing:

Before proceeding with either approach preprocessing of the data is necessary.
Initially the raw messages which is a sequence of characters is converted into vectors by utilising a bag of words approach,since the mapping is not one to one.The bag of words approach assigns a different number to each unique word in the sentence.We will convert each message, represented as a list of tokens into a vector that machine learning models can understand. It involves three steps in the bag of words approach

1.Counting the frequency of a word occurring in each message (term frequency)

2.Weighting the counts, so that frequent tokens get lower weight (inverse document frequency)

3.Normalizing the vectors to unit length, to abstract from the original text length (L2 norm)

Each vector has as many dimensions as there are unique words in the SMS corpus

Text Analysis is a major application field for machine learning algorithms. However the raw data, a sequence of symbols cannot be fed directly to the algorithms themselves as most of them expect numerical feature vectors with a fixed size rather than the raw text documents with variable length. In order to address this, scikit-learn provides utilities for the most common ways to extract numerical features from text content, namely: tokenizing strings and giving an integer id for each possible token, for instance by using white-spaces and punctuation as token separators. counting the occurrences of tokens in each document. normalizing and weighting with diminishing importance tokens that occur in the majority of samples / documents. In this scheme, features and samples are defined as follows: each individual token occurrence frequency (normalized or not) is treated as a feature. the vector of all the token frequencies for a given document is considered a multivariate sample. A corpus of documents can thus be represented by a matrix with one row per document and one column per token (e.g. word) occurring in the corpus. We call vectorization the general process of turning a collection of text documents into

numerical feature vectors. This specific strategy (tokenization, counting and normalization) is called the Bag of Words or “Bag of n-grams” representation. Documents are described by word occurrences while completely ignoring the relative position information of the words in the document.

3.1.1 Bag of Words:

The **bag-of-words** model is a simplifying representation used in natural language processing and information retrieval (IR). In this model, a text (such as a sentence or a document) is represented as the **bag** (multiset) of its **words**, disregarding grammar and even **word** order but keeping multiplicity.
Bag of Words (BoW) is an algorithm that counts how many times a word appears in a document. Those word counts allow us to compare documents and gauge their similarities for applications like search, document classification and topic modeling. BoW is a method for preparing text for input in a deep-learning net.

BoW lists words with their word counts per document. In the table where the words and documents effectively become vectors are stored, each row is a word, each column is a document and each cell is a word count. Each of the documents in the corpus is represented by columns of equal length. Those are word count vectors, an output stripped of context.

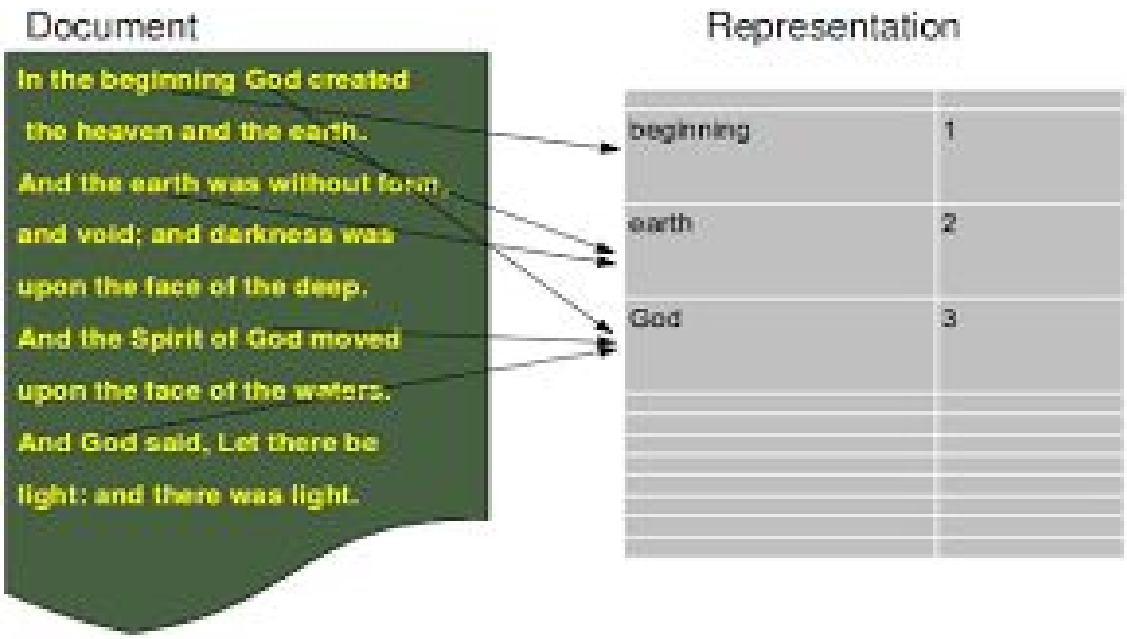


Fig 3.1 Bag of Words - Representation example

After counting, the term weighting and normalization can be done with TF-IDF, using scikit-learn's TfidfTransformer

3.1.2 TF-IDF Algorithm:

Term Frequency - Inverse Document Frequency is a method to judge the topic of an article by the words it contains, With this method words are given weight – TF-IDF measures relevance, not frequency. That is, word counts are replaced with TF-IDF scores across the whole dataset.
First, TF-IDF measures the number of times that words appear in a given document But because words such as “and” or “the” appear frequently in all documents, those are systematically discounted. That’s the inverse-document frequency part. The more documents a word appears in, the less valuable that word is as a signal. That’s intended to leave only the frequent AND distinctive words as markers. Each word’s TF-IDF relevance is a normalized data format that also adds up to one.

TFIDF

For a term i in document j:

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

tf_{ij} = number of occurrences of i in j
df_i = number of documents containing i
N = total number of documents

Fig 3.1.2 TFIDF

Steps for preprocessing:

Step 1:

We decide a function to split the messages into its individual words

```
In [11]: def split_into_tokens(message):
             message = unicode(message, 'utf8')  # convert bytes into proper unicode
             return TextBlob(message).words
```

Here are some of the original texts again:

```
In [12]: messages.message.head()
Out[12]: 0    Go until jurong point, crazy.. Available only ...
         1                        Ok lar... Joking wif u oni...
         2    Free entry in 2 a wkly comp to win FA Cup fina...
         3    U dun say so early hor... U c already then say...
         4    Nah I don't think he goes to usf, he lives aro...
         Name: message, dtype: object
```

...and here are the same messages, tokenized:

```
In [13]: messages.message.head().apply(split_into_tokens)
Out[13]: 0    [Go, until, jurong, point, crazy, Available, o...
         1                        [Ok, lar, Joking, wif, u, oni]
         2    [Free, entry, in, 2, a, wkly, comp, to, win, F...
         3    [U, dun, say, so, early, hor, U, c, already, t...
         4    [Nah, I, do, n't, think, he, goes, to, usf, he...
         Name: message, dtype: object
```

Before we proceed to the next step we need to analyse the split words to check whether the following holds any importance such as

1. Do capital letters carry information?
2. Does distinguishing inflected form ("goes" vs. "go") carry information?
3. Do interjections, determiners carry information?

Hence we want to better normalise the text

Text normalization is the process of transforming text into a single canonical form that it might not have had before. Normalizing text before storing or

processing it allows for separation of concerns, since input is guaranteed to be consistent before operations are performed on it. Text normalization requires being aware of what type of text is to be normalized and how it is to be processed afterwards

For simple, context-independent normalization, such as removing non-alphanumeric characters or diacritical marks, regular expressions would suffice. For example, the sed script sed -e "s/\s+/ /g" *inputfile* would normalize runs of whitespace characters into a single space. More complex normalization requires correspondingly complicated algorithms, including domain knowledge of the language and vocabulary being normalized. Among other approaches, text normalization has been modeled as a problem of tokenizing and tagging streams of text and as a special case of machine translation

We will now tag the split messages using the Parts of speech tag using Text Blob

```
In [14]: TextBlob("Hello world, how is it going?").tags  # list of (word, POS) pairs

Out[14]: [(u'Hello', u'UH'),
          (u'world', u'NN'),
          (u'how', u'WRB'),
          (u'is', u'VBZ'),
          (u'it', u'PRP'),
          (u'going', u'VBG')]
```

The tags used are explained below

Alphabetical list of part-of-speech tags used in the Penn Treebank Project:

Number	Tag	Description
1.	CC	Coordinating conjunction
2.	CD	Cardinal number
3.	DT	Determiner
4.	EX	Existential *there*
5.	FW	Foreign word
6.	IN	Preposition or subordinating conjunction
7.	JJ	Adjective
8.	JJR	Adjective, comparative
9.	JJS	Adjective, superlative
10.	LS	List item marker
11.	MD	Modal
12.	NN	Noun, singular or mass
13.	NNS	Noun, plural
14.	NNP	Proper noun, singular
15.	NNPS	Proper noun, plural
16.	PDT	Predeterminer
17.	POS	Possessive ending
18.	PRP	Personal pronoun
19.	PRP$	Possessive pronoun
20.	RB	Adverb
21.	RBR	Adverb, comparative
22.	RBS	Adverb, superlative
23.	RP	Particle
24.	SYM	Symbol
25.	TO	*to*
26.	UH	Interjection
27.	VB	Verb, base form
28.	VBD	Verb, past tense
29.	VBG	Verb, gerund or present participle
30.	VBN	Verb, past participle
31.	VBP	Verb, non-3rd person singular present
32.	VBZ	Verb, 3rd person singular present
33.	WDT	Wh-determiner
34.	WP	Wh-pronoun
35.	WP$	Possessive wh-pronoun
36.	WRB	Wh-adverb

Example: [(u'Hello', u'UH')] where ‘Hello’ is the **word** and ‘UH’ is the **POS tag** representation for **interjection.**

Step 2:

Next we normalise the text into their lemmas or base form using a process called as **Lemmatisation**

Lemmatisation (or lemmatization) in linguistics is the process of grouping together the inflected forms of a word so they can be analysed as a single item, identified by the word's lemma, or dictionary form.

In computational linguistics, lemmatisation is the algorithmic process of determining the lemma of a word based on its intended meaning. Unlike stemming, lemmatisation depends on correctly identifying the intended part of speech and meaning of a word in a sentence, as well as within the larger context surrounding that sentence, such as neighboring sentences or even an entire document. As a result, developing efficient lemmatisation algorithms is an open area of research.

For instance:

1. The word "better" has "good" as its lemma. This link is missed by stemming, as it requires a dictionary look-up.

2. The word "walk" is the base form for word "walking", and hence this is matched in both stemming and lemmatisation.
3. The word "meeting" can be either the base form of a noun or a form of a verb ("to meet") depending on the context; e.g., "in our last meeting" or "We are meeting again tomorrow". Unlike stemming, lemmatisation attempts to select the correct lemma depending on the context.

We define a function to normalise the words into their lemma or base form.

```
In [15]: def split_into_lemmas(message):
             message = unicode(message, 'utf8').lower()
             words = TextBlob(message).words
             # for each word, take its "base form" = lemma
             return [word.lemma for word in words]

         messages.message.head().apply(split_into_lemmas)

Out[15]: 0    [go, until, jurong, point, crazy, available, o...
         1                      [ok, lar, joking, wif, u, oni]
         2    [free, entry, in, 2, a, wkly, comp, to, win, f...
         3    [u, dun, say, so, early, hor, u, c, already, t...
         4    [nah, i, do, n't, think, he, go, to, usf, he, ...
         Name: message, dtype: object
```

This way the stop words can be filtered from the data

Stop words are words that are particularly common in a text corpus and thus considered as rather uninformative (e.g., words such as so, and, or, the, …”). One approach to stop word removal is to search against a language-specific stop word dictionary. An alternative approach is to create a stop list by sorting all words in the entire text corpus by frequency. The stop list — after conversion into a set of non-redundant words — is then used to remove all those words from the input documents that are ranked among the top n words in this stop list.

Step 3:

Now we'll convert each message, represented as a list of tokens (lemmas) above, into a vector that machine learning models can understand.

Doing that requires essentially three steps, in the bag-of-words model:

1. counting how many times does a word occur in each message (term frequency)
2. weighting the counts, so that frequent tokens get lower weight (inverse document frequency)
3. normalizing the vectors to unit length, to abstract from the original text length (L2 norm)

Each vector has as many dimensions as there are unique words in the SMS corpus:

```
In [16]: bow_transformer = CountVectorizer(analyzer=split_into_lemmas).fit(messages['message'])
         print len(bow_transformer.vocabulary_)

         8874
```

Let's take one text message and get its bag-of-words counts as a vector, putting to use our new bow_transformer:

```
In [17]: message4 = messages['message'][3]
         print message4

         U dun say so early hor... U c already then say...

In [18]: bow4 = bow_transformer.transform([message4])
         print bow4
         print bow4.shape

           (0, 1158)     1
           (0, 1899)     1
           (0, 2897)     1
           (0, 2927)     1
           (0, 4021)     1
           (0, 6736)     2
           (0, 7111)     1
           (0, 7698)     1
           (0, 8013)     2
         (1, 8874)
```

So there are 9 unique words in message 4 and only 2 of them appear twice. It can be inferred that the words “Say” and “U” repeat twice

```
In [19]: print bow_transformer.get_feature_names()[6736]
         print bow_transformer.get_feature_names()[8013]

         say
         u
```

The bag of words counts for this corpus is a large,sparse matrix.

```
In [20]: messages_bow = bow_transformer.transform(messages['message'])
         print 'sparse matrix shape:', messages_bow.shape
         print 'number of non-zeros:', messages_bow.nnz
         print 'sparsity: %.2f%%' % (100.0 * messages_bow.nnz / (messages_bow.shape[0] * messages_bow.shape[1]))

         sparse matrix shape: (5574, 8874)
         number of non-zeros: 80272
         sparsity: 0.16%
```

Now after counting, the term weighting and normalization can be done with TF-IDF

```
In [21]: tfidf_transformer = TfidfTransformer().fit(messages_bow)
         tfidf4 = tfidf_transformer.transform(bow4)
         print tfidf4

           (0, 8013)     0.305114653686
           (0, 7698)     0.225299911221
           (0, 7111)     0.191390347987
           (0, 6736)     0.523371210191
           (0, 4021)     0.456354991921
           (0, 2927)     0.32967579251
           (0, 2897)     0.303693312742
           (0, 1899)     0.24664322833
           (0, 1158)     0.274934159477
```

We now transform the entire bag of words corpus into a TF-IDF corpus

```
In [23]: messages_tfidf = tfidf_transformer.transform(messages_bow)
         print messages_tfidf.shape

         (5574, 8874)
```

There are a multitude of ways in which data can be preprocessed and vectorized. These steps, also called "feature engineering", are typically the most time consuming parts of building a predictive pipeline, but they are very important .

Now we can proceed into the explanation of the Naive Bayes approach in the implementation of the classifier

3.2 Naive Bayes Approach:

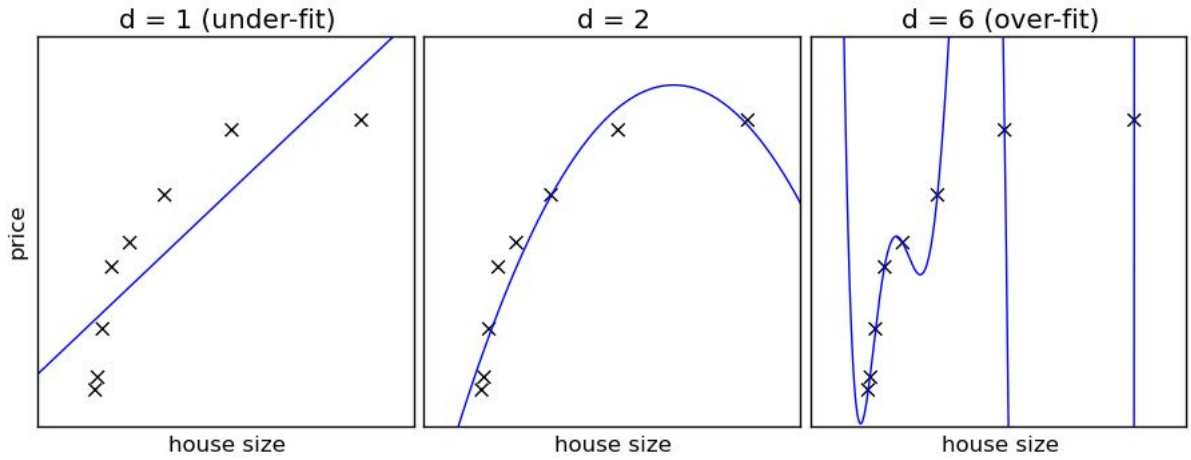
Naive Bayes is a collection of classification algorithms based on Bayes Theorem. It is not a single algorithm but a family of algorithms that all share a common principle, that every feature being classified is independent of the value of any other feature.

MultinomialNB implements the naive Bayes algorithm for multinomial distributed data, and is one of the two classic naive Bayes variants used in text classification (where the data are typically represented as word vector counts, although tf-idf vectors are also known to work well in practice). The distribution is parametrized by vectors $\theta_y = (\theta_{y1}, \ldots, \theta_{yn})$ for each class y, where n is the number of features (in text classification, the size of the vocabulary) and θ_{yi} is the probability $P(x_i \mid y)$ of feature i appearing in a sample belonging to class y.

Naive Bayes is an example of a high bias - low variance classifier (aka simple and stable, not prone to overfitting). An example from the opposite side of the spectrum would be Nearest Neighbour (kNN) classifiers, or Decision Trees, with their low bias but high variance (easy to overfit). Bagging (Random Forests) as a way to lower variance, by training many (high-variance) models and averaging.

High bias = classifier: is opinionated. Not as much room to change its mind with data, it has its own ideas. On the other hand, not as much room it can fool itself into overfitting either (picture on the left).

Low bias = classifier: more obedient, but also more neurotic. Will do exactly what you ask it to do, which, as everybody knows, can be a real nuisance (picture on the right).



Advantages of Naive Bayes Classifier:

- It is very simple to use,If the NB conditional independence assumption actually holds, a Naive Bayes classifier will converge quicker than discriminative models like logistic regression, so you need less training data
- In the case of the SMS classifier this model gives an accuracy of 95% which is a good result
- It’s not sensitive to irrelevant features

Disadvantages of Naive Bayes Classifier:

- It assumes independence of all features
- We cannot decide with utilising one algorithm that it returns the best result,The accuracy may change with another algorithm. This is the reason we utilise another approach to implement the SMS classifier

```
In [39]: pipeline = Pipeline([
             ('bow', CountVectorizer(analyzer=split_into_lemmas)),  # strings to token integer counts
             ('tfidf', TfidfTransformer()),  # integer counts to weighted TF-IDF scores
             ('classifier', MultinomialNB()),  # train on TF-IDF vectors w/ Naive Bayes classifier
         ])
```

```
In [40]: scores = cross_val_score(pipeline,  # steps to convert raw messages into models
                                  msg_train,  # training data
                                  label_train,  # training labels
                                  cv=10,  # split data randomly into 10 parts: 9 for training, 1 for scoring
                                  scoring='accuracy',  # which scoring metric?
                                  n_jobs=-1,  # -1 = use all cores = faster
                                  )
         print scores
```

```
[ 0.93736018  0.95302013  0.94630872  0.9573991   0.95515695  0.96412556
  0.96404494  0.9505618   0.93258427  0.95505618]
```

```
In [41]: print scores.mean()
         print scores.std()
```

```
0.951561784339
0.00982568625738
```

Thus we get an accuracy of 95% using the naive bayes classifier

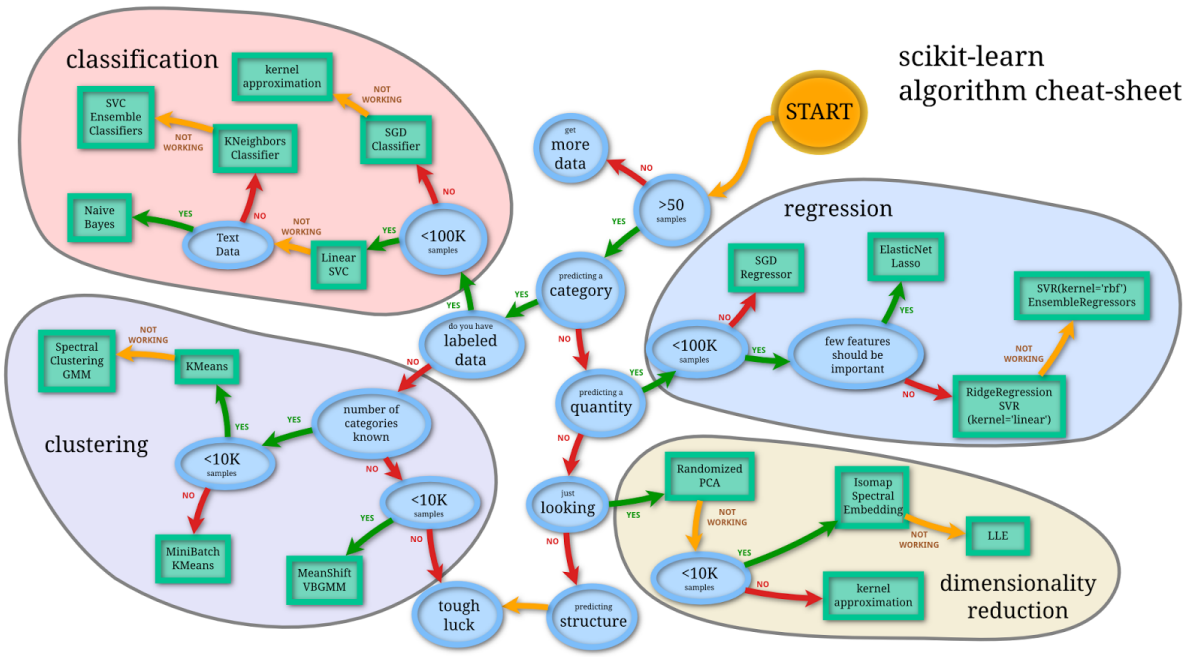


Fig 3.2 Sklearn Algorithm Cheat-Sheet

Let us discuss the second approach that was used to implement this project. One may think, Does IDF weighting have an effect on accuracy? Do we require extra processing cost of lemmatization (vs. just plain words)?

3.3 Support Vector Machine Approach:

In machine learning, support vector machines (SVMs, also support vector networks) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis.“Support Vector Machine” (SVM) is a supervised machine learning algorithm which can be used for both classification or regression challenges.it is mostly used in classification problems. In this algorithm, we plot each data item as a point in n-dimensional space (where n is number of features you have) with the value of each feature being the value of a particular coordinate.

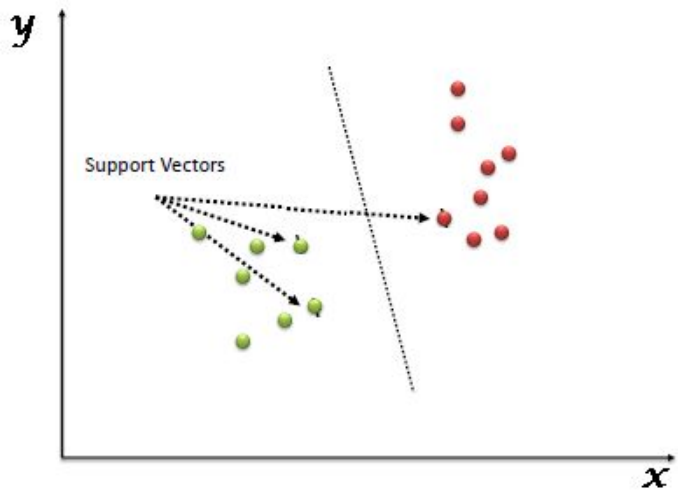


Fig 3.3 Support Vector Machines - SVM

Advantages of Support Vector Machines:

- It works really well with clear margin of separation
- It is effective in high dimensional spaces.
- It is effective in cases where number of dimensions is greater than the number of samples.
- It uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
- It gives an accuracy of 98% in our Ham/Spam classifier which is higher than an accuracy of 95% while using Naive Bayes.

Disadvantages of Support Vector Machines:

- It doesn't perform well, when we have large data set because the required training time is higher
- It also doesn't perform very well, when the data set has more noise i.e. target classes are overlapping
- SVM doesn't directly provide probability estimates, these are calculated using an expensive five-fold cross-validation. It is related SVC method of Python scikit-learn library.

In [88]:
```
pipeline_svm = Pipeline([
    ('bow', CountVectorizer(analyzer=split_into_lemmas)),
    ('tfidf', TfidfTransformer()),
    ('classifier', SVC()),  # <== change here
])

# pipeline parameters to automatically explore and tune
param_svm = [
  {'classifier__C': [1, 10, 100, 1000], 'classifier__kernel': ['linear']},
  {'classifier__C': [1, 10, 100, 1000], 'classifier__gamma': [0.001, 0.0001], 'classifier__kernel': ['rbf']},
]

grid_svm = GridSearchCV(
    pipeline_svm,  # pipeline from above
    param_grid=param_svm,  # parameters to tune via cross validation
    refit=True,  # fit using all data, on the best detected classifier
    n_jobs=-1,  # number of cores to use for parallelization; -1 for "all cores"
    scoring='accuracy',  # what score are we optimizing?
    cv=StratifiedKFold(label_train, n_folds=5),  # what type of cross validation to use
)
```

In [89]:
```
%time svm_detector = grid_svm.fit(msg_train, label_train) # find the best combination from param_svm
print svm_detector.grid_scores_
```

```
In [90]: print svm_detector.predict(["Hi mom, how are you?"])[0]
         print svm_detector.predict(["WINNER! Credit for free!"])[0]
         print svm_detector.predict(["Hey!!! you won 25000$ on a lottery click link to claim your reward"])[0]

         ham
         spam
         spam

In [91]: print confusion_matrix(label_test, svm_detector.predict(msg_test))
         print classification_report(label_test, svm_detector.predict(msg_test))

         [[975   2]
          [ 13 125]]
                      precision    recall  f1-score   support

                 ham       0.99      1.00      0.99       977
                spam       0.98      0.91      0.94       138

         avg / total       0.99      0.99      0.99      1115
```

Thus we get an accuracy of 98% using Support Vector Machines.

4. Performance Metrics:

To test the performance of our model we utilise the following performance metrics:

Precision: The precision is the ratio `tp / (tp + fp)` where `tp` is the number of true positives and `fp` the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

Recall: The recall is the ratio `tp / (tp + fn)` where `tp` is the number of true positives and `fn` the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

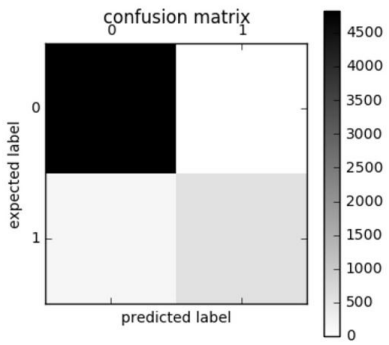
F1-Score: The F-beta score can be interpreted as a weighted harmonic mean of the precision and recall, where an F-beta score reaches its best value at 1 and worst score at 0.

The F-beta score weights recall more than precision by a factor of `beta`. `beta == 1.0` means recall and precision are equally important.

Support: The support is the number of occurrences of each class in y_true.

Confusion Matrix:A confusion matrix is a table that is often used to describe the performance of a classification model (or "classifier") on a set of test data for which the true values are known.

```
In [75]: plt.matshow(confusion_matrix(messages['label'], all_predictions), cmap=plt.cm.binary, interpolation='nearest')
         plt.title('confusion matrix')
         plt.colorbar()
         plt.ylabel('expected label')
         plt.xlabel('predicted label')
Out[75]: <matplotlib.text.Text at 0x121ee74d0>
```




```
In [76]: #From this confusion matrix, we can compute precision and recall, or their combination (harmonic mean) F1:
         print classification_report(messages['label'], all_predictions)
```

```
             precision    recall  f1-score   support

        ham       0.97      1.00      0.98      4827
       spam       1.00      0.77      0.87       747

avg / total       0.97      0.97      0.97      5574
```

In the above confusion matrix the rows are the expected predictions while the columns are the predicted labels .

Reason for utilising Confusion Matrix:

A confusion matrix helps elucidate how the model performed for individual classes. It helps figure and compare the performance of stated models (Naive bayes & SVM)

Text classification rules are typically evaluated using performance measures from information retrieval. Common metrics for text categorization evaluation include recall, precision, accuracy and error rate and F1. Given a test set of N documents, a two-by-two contingency table with four cells can be constructed for each binary classification problem. The cells contain the counts for true positive (TP), false positive (FP), true negative (TN) and false negative (FN), respectively. Clearly, N = TP + FP + TN + FN. The metrics for binary-decisions are defined as:

- Precision = TP / (TP + FP)
- Recall = TP / (TP + FN)
- Accuracy = (TP + TN)/N
- Error = (FP + FN)/N
- F1 = 2*Recall*Precision/(Recall + Precision)

From the above Confusion Matrix I can summarise:

	Precision	Recall	F1-Score	Support
Ham	0.97	1.00	0.98	4827
Spam	1.00	0.77	0.87	747
Avg/Total	0.97	0.97	0.97	5574

Naive Bayes Confusion Matrix Summary:

```
In [41]: predictions = nb_detector.predict(msg_test)
         print confusion_matrix(label_test, predictions)
         print classification_report(label_test, predictions)

         [[973   0]
          [ 46  96]]
                      precision    recall  f1-score   support

                 ham       0.95      1.00      0.98       973
                spam       1.00      0.68      0.81       142

         avg / total       0.96      0.96      0.96      1115
```

Inference:

	Precision	Recall	F1-Score	Support
Ham	0.95	1.00	0.98	973
Spam	1.00	0.68	0.81	142
Avg/Total	0.96	0.96	0.96	1115

Support Vector Machine Confusion Matrix Summary:

```
In [45]: print confusion_matrix(label_test, svm_detector.predict(msg_test))
         print classification_report(label_test, svm_detector.predict(msg_test))

         [[965   8]
          [ 13 129]]
                      precision    recall  f1-score   support

                 ham       0.99      0.99      0.99       973
                spam       0.94      0.91      0.92       142

         avg / total       0.98      0.98      0.98      1115
```

Inference:

	Precision	Recall	F1-Score	Support
Ham	0.99	0.99	0.99	973
Spam	0.94	0.91	0.92	142
Avg/Total	0.98	0.98	0.98	1115

Cross Validation: In k-fold cross-validation, the original sample is randomly partitioned into k equal sized subsamples. Of the k subsamples, a single subsample is retained as the validation data for testing the model, and the remaining k − 1 subsamples are used as training data. **K-fold cross validation** is one way to improve over the holdout method. The data set is divided into *k* subsets, and the holdout method is repeated *k* times. Each time, one of the *k* subsets is used as the test set and the other *k-1* subsets are put together to form a training set. Then the average error across all *k* trials is computed. The advantage of this method is that it matters less how the data gets divided. Every data point gets to be in a test set exactly once, and gets to be in a training set *k-1* times. The variance of the resulting estimate is reduced as *k* is increased. The disadvantage of this method is that the training algorithm has to be rerun from scratch *k* times, which means it takes *k* times as much computation to make an evaluation. A variant of this method is to randomly divide the data into a test and training set *k* different times. The advantage of doing this is that you can independently choose how large each test set is and how many trials you average over.

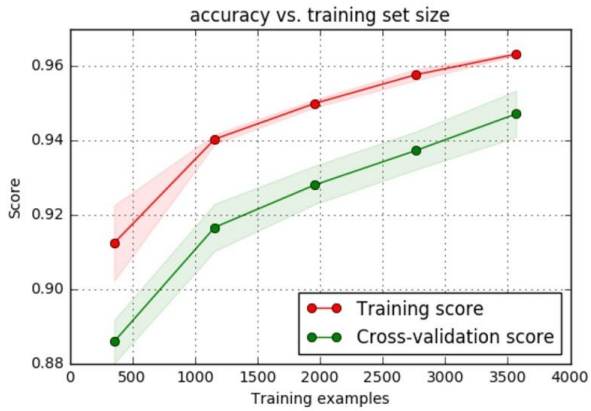
```
In [79]: scores = cross_val_score(pipeline,  # steps to convert raw messages into models
                                  msg_train,  # training data
                                  label_train,  # training labels
                                  cv=10,  # split data randomly into 10 parts: 9 for training, 1 for scoring
                                  scoring='accuracy',  # which scoring metric?
                                  n_jobs=-1,  # -1 = use all cores = faster
                                  )
         print scores

         [ 0.94170404  0.9573991   0.96412556  0.9529148   0.94170404  0.94618834
           0.9529148   0.94843049  0.94394619  0.94831461]

In [80]: print scores.mean()
         print scores.std()

         0.9497641961
         0.00681792912038
```

```
Out[82]: <module 'matplotlib.pyplot' from '/Users/sylvester/anaconda/lib/python2.7/site-packages/matplotlib/pyplot.pyc'>
```



(We're effectively training on 64% of all available data: we reserved 20% for the test set above, and the 5-fold cross validation reserves another 20% for validation sets => 0.8*0.8*5574=3567 training examples left.) Since performance keeps growing, both for training and cross validation scores, we see our model is not complex/flexible enough to capture all nuance, given little data. In this particular case, it's not very pronounced, since the accuracies are high anyway. At this point, we have two options:

- Use more training data, to overcome low model complexity
- Use a more complex (lower bias) model to start with, to get more out of the existing data

Over the last years, as massive training data collections become more available, and as machines get faster, approach 1. is becoming more and more popular (simpler algorithms, more data). Straightforward algorithms, such as Naive Bayes, also have the added benefit of being easier to interpret (compared to some more complex, black-box models, like neural networks).

There are quite a few possible metrics for evaluating model performance. For example, the cost of mispredicting "spam" as "ham" is probably much lower than mispredicting "ham" as "spam".

Performance Study:

Comparing the performance of the two models:

Let us get back to addressing the question What is the effect of IDF weighting on accuracy? Does the extra processing cost of lemmatization (vs. just plain words) really help?

```
In [37]: params = {
             'tfidf__use_idf': (True, False),
             'bow__analyzer': (split_into_lemmas, split_into_tokens),
         }

         grid = GridSearchCV(
             pipeline,  # pipeline from above
             params,  # parameters to tune via cross validation
             refit=True,  # fit using all available data at the end, on the best found param combination
             n_jobs=-1,  # number of cores to use for parallelization; -1 for "all cores"
             scoring='accuracy',  # what score are we optimizing?
             cv=StratifiedKFold(label_train, n_folds=5),  # what type of cross validation to use
         )

In [38]: %time nb_detector = grid.fit(msg_train, label_train)
         print nb_detector.grid_scores_

         CPU times: user 4.09 s, sys: 291 ms, total: 4.38 s
         Wall time: 20.2 s
         [mean: 0.94752, std: 0.00357, params: {'tfidf__use_idf': True, 'bow__analyzer': <function split_into_lemmas at 0x1131e8668>}, m
         ean: 0.92958, std: 0.00390, params: {'tfidf__use_idf': False, 'bow__analyzer': <function split_into_lemmas at 0x1131e8668>}, me
         an: 0.94528, std: 0.00259, params: {'tfidf__use_idf': True, 'bow__analyzer': <function split_into_tokens at 0x11270b7d0>}, mea
         n: 0.92868, std: 0.00240, params: {'tfidf__use_idf': False, 'bow__analyzer': <function split_into_tokens at 0x11270b7d0>}]
```

Grid Search helps me to get the best parameter combinations . These combinations are displayed first i.e.use_idf=True and analyzer=split_into_lemmas .

A quick check on this theory

```
In [39]: print nb_detector.predict_proba(["Hi mom, how are you?"])[0]
         print nb_detector.predict_proba(["WINNER! Credit for free!"])[0]

         [ 0.99383955  0.00616045]
         [ 0.29663109  0.70336891]
```

The predict_proba returns the predicted probability for each class (ham, spam). In the first case, the message is predicted to be ham with > 99% probability, and spam with < 1%. So if forced to choose, the model will say "ham":

```
In [40]: print nb_detector.predict(["Hi mom, how are you?"])[0]
         print nb_detector.predict(["WINNER! Credit for free!"])[0]

         ham
         spam
```

The overall scores on the test set are listed below

```
In [41]: predictions = nb_detector.predict(msg_test)
         print confusion_matrix(label_test, predictions)
         print classification_report(label_test, predictions)

         [[973   0]
          [ 46  96]]
                      precision    recall  f1-score   support

                 ham       0.95      1.00      0.98       973
                spam       1.00      0.68      0.81       142

         avg / total       0.96      0.96      0.96      1115
```

This Naive bayes classifier is our **benchmark** model.Let us try to improvise on this model by using a different approach of using a Support Vector Machine classifier.

```
In [42]: pipeline_svm = Pipeline([
             ('bow', CountVectorizer(analyzer=split_into_lemmas)),
             ('tfidf', TfidfTransformer()),
             ('classifier', SVC()),  # <== change here
         ])

         # pipeline parameters to automatically explore and tune
         param_svm = [
           {'classifier__C': [1, 10, 100, 1000], 'classifier__kernel': ['linear']},
           {'classifier__C': [1, 10, 100, 1000], 'classifier__gamma': [0.001, 0.0001], 'classifier__kernel': ['rbf']},
         ]

         grid_svm = GridSearchCV(
             pipeline_svm,  # pipeline from above
             param_grid=param_svm,  # parameters to tune via cross validation
             refit=True,  # fit using all data, on the best detected classifier
             n_jobs=-1,  # number of cores to use for parallelization; -1 for "all cores"
             scoring='accuracy',  # what score are we optimizing?
             cv=StratifiedKFold(label_train, n_folds=5),  # what type of cross validation to use
         )
```

```
In [43]: %time svm_detector = grid_svm.fit(msg_train, label_train) # find the best combination from param_svm
         print svm_detector.grid_scores_

CPU times: user 5.24 s, sys: 170 ms, total: 5.41 s
Wall time: 1min 8s
[mean: 0.98677, std: 0.00259, params: {'classifier__kernel': 'linear', 'classifier__C': 1}, mean: 0.98654, std: 0.00100, param
s: {'classifier__kernel': 'linear', 'classifier__C': 10}, mean: 0.98654, std: 0.00100, params: {'classifier__kernel': 'linear',
 'classifier__C': 100}, mean: 0.98654, std: 0.00100, params: {'classifier__kernel': 'linear', 'classifier__C': 1000}, mean: 0.8
6432, std: 0.00006, params: {'classifier__gamma': 0.001, 'classifier__kernel': 'rbf', 'classifier__C': 1}, mean: 0.86432, std:
 0.00006, params: {'classifier__gamma': 0.0001, 'classifier__kernel': 'rbf', 'classifier__C': 1}, mean: 0.86432, std: 0.00006,
 params: {'classifier__gamma': 0.001, 'classifier__kernel': 'rbf', 'classifier__C': 10}, mean: 0.86432, std: 0.00006, params:
 {'classifier__gamma': 0.0001, 'classifier__kernel': 'rbf', 'classifier__C': 10}, mean: 0.97040, std: 0.00587, params: {'classi
fier__gamma': 0.001, 'classifier__kernel': 'rbf', 'classifier__C': 100}, mean: 0.86432, std: 0.00006, params: {'classifier__gam
ma': 0.0001, 'classifier__kernel': 'rbf', 'classifier__C': 100}, mean: 0.98722, std: 0.00280, params: {'classifier__gamma': 0.0
01, 'classifier__kernel': 'rbf', 'classifier__C': 1000}, mean: 0.97040, std: 0.00587, params: {'classifier__gamma': 0.0001, 'cl
assifier__kernel': 'rbf', 'classifier__C': 1000}]
```

The best parameter combination in the case of SVM will be a **Linear kernel with C=1**

To cross check the output

```
In [44]: print svm_detector.predict(["Hi mom, how are you?"])[0]
         print svm_detector.predict(["WINNER! Credit for free!"])[0]

ham
spam
```

The overall scores are listed below:

```
In [45]: print confusion_matrix(label_test, svm_detector.predict(msg_test))
         print classification_report(label_test, svm_detector.predict(msg_test))

[[965   8]
 [ 13 129]]
             precision    recall  f1-score   support

        ham       0.99      0.99      0.99       973
       spam       0.94      0.91      0.92       142

avg / total       0.98      0.98      0.98      1115
```

Compared to our NB benchmark this SVM model performs better than the previous model giving a precision score of 0.98 which is better than the precision score of 0.96 which the naive bayes model gave

5.Results:

In the end we have designed a Ham/Spam Classifier using two different approaches

1.**Naive Bayes Classifier with TF-IDF** which results in an accuracy of 95%. This has been used as a benchmark for testing the performance of my second implementation as discussed above

```
In [41]: predictions = nb_detector.predict(msg_test)
         print confusion_matrix(label_test, predictions)
         print classification_report(label_test, predictions)

         [[973   0]
          [ 46  96]]
                      precision    recall  f1-score   support

                 ham       0.95      1.00      0.98       973
                spam       1.00      0.68      0.81       142

         avg / total       0.96      0.96      0.96      1115
```

2.**Support Vector Machine** which results in an accuracy of 98% is the final model I ended up with while experimenting with various algorithms(random forests) with which the Support Vector Machine gave one of the best examples . The Resulting model is also made into a production level predictor by serialising it to the disk so that we can reuse the model anytime.

```
In [46]: # store the spam detector to disk after training
         with open('sms_spam_detector.pkl', 'wb') as fout:
             cPickle.dump(svm_detector, fout)

         # ...and load it back, whenever needed, possibly on a different machine
         svm_detector_reloaded = cPickle.load(open('sms_spam_detector.pkl'))
```

Results Tabulated

Models used	Final Accuracy	Precision	Recall	F1-score	Type
Naive bayes	96%	0.96	0.96	0.96	Multinomial NB
SVM	98%	0.98	0.98	0.98	Linear Kernel with c=1

Challenges I faced while Implementing this project:

I learnt a lot while implementing this predictor. I tried implementing the same with a Bernoulli Naive bayes model before settling in for the Multinomial Naive Bayes Model as a benchmark to work with since it had a better accuracy compared to the previous implementation

Challenges:

1. Cleaning the dataset and figuring out what I could use to help train the learning algorithm properly was another challenge I faced while coding this project.
2. One of the main challenges that I faced during this project was filtering the stop words and pronouns and figuring out the base forms of each word. For example the word “Best” has the lemma “good” and the lemma form for “walking” is “Walk”
3. To handle and automate base word generation I had written a function as shown below
4. This function splits the given sentences into lemmas

```
In [15]: def split_into_lemmas(message):
             message = unicode(message, 'utf8').lower()
             words = TextBlob(message).words
             # for each word, take its "base form" = lemma
             return [word.lemma for word in words]

         messages.message.head().apply(split_into_lemmas)

Out[15]: 0    [go, until, jurong, point, crazy, available, o...
         1                      [ok, lar, joking, wif, u, oni]
         2    [free, entry, in, 2, a, wkly, comp, to, win, f...
         3    [u, dun, say, so, early, hor, u, c, already, t...
         4    [nah, i, do, n't, think, he, go, to, usf, he, ...
         Name: message, dtype: object
```

5. I enjoyed implementing the various algorithms and exploring the possibilities of different results when I tried a different classifier

What I learned during this project:

TF-IDF:(Inferences)
Advantage:
TF-IDF is very easy to compute and can be used as a basic metric to extract the most descriptive terms in a document
Disadvantage:
TF-IDF is based on the bag-of-words (BoW) model, therefore it does not capture position in text, semantics, co-occurrences in different documents, etc.

Other Inferences(Learning Experiences):

1. The practical meaning of utilising a high bias-low variance classifier
2. The disadvantages of overfitting data
3. The problems in accuracy of the model if we use the same training data to train our model instead of splitting the data into testing and training sets and then training the model with the testing data and then calculating the performance of the model
4. The process of tagging the words using the Parts of Speech tags and understanding how to classify sentences with their “interjections” removed

There were lot of blogs which I referred to while implementing this project. Some of which have been listed below.

References:

Text Feature Extraction- Bag of Words Representation:
http://scikit-learn.org/stable/modules/feature_extraction.html#the-bag-of-words-representation
http://scikit-learn.org/stable/auto_examples/model_selection/grid_search_text_feature_extraction.html
http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html
Naive Bayes - Sklearn Documentation:
http://scikit-learn.org/stable/modules/naive_bayes.html
Support Vector Machines -Sklearn Documentation:
http://scikit-learn.org/stable/modules/svm.html
http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
Pipelines-Sklearn Documentation:
http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html
Cross validation-Sklearn Documentation:
http://scikit-learn.org/stable/modules/cross_validation.html
http://scikit-learn.org/0.17/modules/generated/sklearn.grid_search.GridSearchCV.html
http://scikit-learn.org/0.17/modules/generated/sklearn.grid_search.GridSearchCV.html
Model Selection-Sklearn Documentation:
http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html#sklearn.model_selection.StratifiedKFold
http://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_curve.html
Accuracy score/Metrics-Sklearn Documentation:
http://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html#sklearn.metrics.accuracy_score
Decision Tree Classifier-Sklearn Documentation:
http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

Other References:
https://www.thinkful.com/projects/building-a-text-classifier-using-naive-bayes-499/
http://blog.fliptop.com/blog/2015/03/02/bias-variance-and-overfitting-machine-learning-overview/
https://deeplearning4j.org/bagofwords-tf-idf
http://blog.aylien.com/naive-bayes-for-dummies-a-simple-explanation/
https://www.analyticsvidhya.com/blog/2015/10/understaing-support-vector-machine-example-code/
http://www.dataschool.io/simple-guide-to-confusion-matrix-terminology/
https://www.cs.cmu.edu/~schneide/tut5/node42.html
http://zacstewart.com/2015/04/28/document-classification-with-scikit-learn.html
https://www.crowdflower.com/understanding-your-model-statistics/
http://www.ling.upenn.edu/courses/Fall_2007/ling001/penn_treebank_pos.html
https://en.wikipedia.org/wiki/Lemmatisation

https://en.wikipedia.org/wiki/Text_normalization
https://en.wikipedia.org/wiki/Stop_words
http://stackoverflow.com/questions/19335165/cross-validation-and-grid-search
http://sebastianraschka.com/Articles/2014_naive_bayes_1.html

Affirmation:

I hereby confirm that this submission is my work. I have cited above the origins of any parts of the submission that were taken from Websites, books, forums, blog posts, github repositories, etc.

Sylvester Ranjith Francis