KNN, Naive Bayes and SVM

We will cover the following points in this discussion:

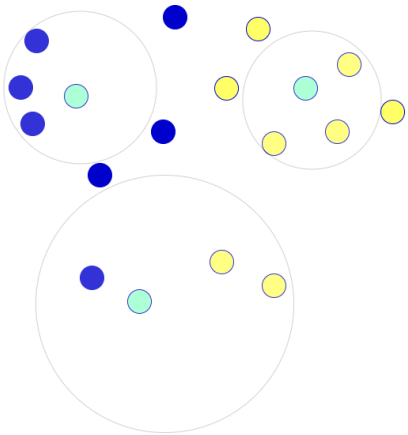
- K-Nearest Neighbours (KNN)
- Naive Bayes
- Support Vector Machines (SVM)
- Implementation in Python using Scikit-learn library - specifically for text data

K-Nearest Neighbours (KNN)

KNN is one of the simplest supervised learning algorithms and it can be used for both, classification as well as regression.

In KNN for classification, an observation is classified by a majority vote of its neighbours. This observation is assigned to a class most common amongst its k nearest neighbours. k is a small positive number which stands for the number of nearest observations from which we wish to take a vote.

For regression, the difference is that instead of taking a vote from the k nearest neighbours, the algorithm considers averages of the k nearest neighbours.



Let us understand this better with an example.

In the diagram above, our objective is to classify the light blue circles i.e. determine whether they should be dark blue or yellow considering its closest neighbours. We choose k=3 i.e we will consider the 3 closest neighbours to determine which class the light blue circle belongs to. In the diagram, you will observe a bigger circle drawn around each of the light blue points which include the three circles closest to it.

Lets consider the top left bigger circle. For the light blue circle within it, we observe that its three closest neighbours are dark blue. Hence, we will consider the majority vote and classify the light blue circle as dark blue.

Now, considering the top right bigger circle, the three closest neighbours for the light blue circle are yellow, hence this light blue circle will be classified as a yellow circle.

However, observing the bottom bigger circle, the light blue circle has two yellow circles and one dark blue circle as its closest neighbours. Since yellow circles are in a majority here, it will be classified as a yellow circle.

Having said that, for the bottom circle, despite the majority being the yellow circle, we observe that the dark blue circle is closer to the light blue circle. What if we want to consider the distance of the nearest neighbours to the light blue circle to determine it class also? To take care of this, we can consider the inverse of the distance as weights to the votes. In this case, the dark blue circle will be given a higher weight, say 0.7 since it is closer to the light blue cirlce, and the two yellow balls will be given lesser weight, say 0.3 and 0.25, since they are further away from the light blue circle. Now when we consider the weights for the two classes, the dark blue circles will have a weight of 0.7 and the yellow circles will have a weight of 0.55 (0.3+0.25). Considering this, the dark blue class will have a higher vote now as compared to the two yellow circles combined and hence the light blue circle will be classified as dark blue one. In short, if we consider a weighted majority vote instead of a simple majority vote, our classification may be different.

One thing to note here is that if we change the value of 'k' i.e. the number of closest neighbours to consider, then the classification of the light blue balls may change depending on the neighbourhood size.

The steps followed in the algorithm are as follows:

- Read the data
- Set the value of k
- To predict the class of an observation from the test dataset, need to iterate through all the training data points
- Calculate the distance between test observation and each training data point. Distance measures that can be used are Euclidean distance, Chebyshev, cosine similarity etc.
- Calculated distances are sorted in ascending order
- Consider only the top k rows from the sorted array
- Get the most frequent class from these k rows

- The most frequent class is the predicted class

Implementation of K-Nearest Neighbours using scikit-learn

```
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
# ignore this bit, its simply there to suppress deprecation warnings from the reading material
```

```
import numpy as np
import pandas as pd
```

```
# load the iris dataset
from sklearn.datasets import load_iris
iris = load_iris()
```

```
# store the feature matrix (X) and response vector (y)
X = iris.data
y = iris.target
```

```
# splitting X and y into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=2)
```

Since KNN relies on a distance measure to figure out which class a test observation belongs to. Scaling the data becomes a must here since if we do not scale the data then features with larger values will be dominant.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)

X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

We import the KNeighborsClassifier to classify the data, using 'k' as 5 i.e. the algorithm will consider 5 nearest neighbours to assess class membership.

```
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors=5)
classifier.fit(X_train, y_train)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=5, p=2,
           weights='uniform')
```

```
y_pred = classifier.predict(X_test)
```

```
# comparing actual response values (y_test) with predicted response values (y_pred)
from sklearn import metrics
print("KNN model accuracy:", metrics.accuracy_score(y_test, y_pred))
```

```
KNN model accuracy: 0.9666666666666667
```

Reference: https://stackabuse.com/k-nearest-neighbors-algorithm-in-python-and-scikit-learn/

Points to note:

- The KNN algorithm is purely distance based and variables which have larger values i.e. are high on scale will dominate. In order to avoid this, we will need to standardize the variables such that all the variables are on a single scale.
- Votes can be weighted in different ways, inverse of the the distance is most popular.
- Lower values of k will capture very niche patterns and will tend to overfit.
- Very high values of k will capture broad patterns but may miss local patterns in the data.
- There is no model equation here; the prediction depends completely on the training data. e.g. when we take a new observation to be classified, we determine its class by checking the classes of the training data observations present in its neighbourhood. Hence, training data itself is the model.
- KNN can capture very local patterns in the data whereas other algorithms can capture patterns across the data. Standalone KNN is not a very good algorithm because it relies on simple neighbourhood voting. There is no general pattern extraction

from different variables. So they are rarely used on a standalone basis. But since they can extract very niche patterns, it is very useful in stacking algorithms.

Naive Bayes

Naive Bayes is a classification algorithm based on Bayes Theorem with a naive assumption of independence among features i.e. presence of one feature does not affect the other. These models are easy to build and very useful for high dimensional datasets.

The fundamental Naive Bayes assumption is that each feature makes an independent and equal contribution to the outcome. This assumption is not generally correct in the real world but works quite well in practice.

Lets quickly review some concepts of probability.

Say we have a bag with 10 balls, 6 blue and 4 red. If we choose a ball from this bag, then the probability that it is blue will be 6/10 i.e. the number of blue balls divided by all the possible selections and similarly the probability that it will be red will be 4/10.

Lets say, we have a pack of cards of 52 cards. If we choose a card from this pack at random, then, say for event A the probability that it is a 5 is 4/52 i.e. the number of 5's present in the pack divided by the total number of cards. And, say for event B, the probability that it is red is 26/52 i.e. total number of reds in the pack of cards divided by the total number of cards.

$$P(A) = \frac{|A|}{|U|} \; and \; P(B) = \frac{|B|}{|U|}$$
$$where \; |A| \; = \; number \; of \; times \; 5 \; appears \; in \; the \; pack \; of \; cards \; = \; 4$$
$$and \; |B| \; = \; number \; of \; red \; cards \; in \; the \; pack \; of \; cards \; = \; 26$$
$$and \; |U| \; = \; total \; numer \; of \; cards \; = \; 52$$

Now, instead of seeing two events in isolation, lets look at them together i.e lets consider the probability of selecting a 5 which is red. In other words, we wish to find the probability when both the events A and B happen together.

$$P(AB) \; = \; \frac{|AB|}{|U|}$$

We know that in the pack of cards, we have two 5's that are red as well. So the probability of selecting a 5 which is red as well is 2/52.

Now, let say we want to find the probability of selecting a 5 given that we already have the red cards P(A|B) i.e the probability of an event A happening given that event B has happened already. All possible outcomes in this case will be the number of red cards i.e. count of B - 26 red cards. Within this subset we select a 5 that is red i.e. both the events A and B have occured. Therefore the probability that 5 is selected given that we already have the red cards only is 2/26.

$$P(A|B) \; = \; \frac{|AB|}{|B|}$$
$$Dividing \; the \; numerator \; and \; denominator \; by \; total \; number \; of \; cards$$
$$P(A|B) \; = \; \frac{|AB|/|U|}{|B|/|U|}$$
$$Referring \; to \; the \; equations \; above$$
$$P(A|B) \; = \; \frac{P(AB)}{P(B)} \; => \; P(AB) \; = \; P(A|B) * P(B)$$
$$Similarly \; the \; probability \; of \; event \; B \; happening \; given \; event \; A \; has \; happened \; already$$
$$P(B|A) \; = \; \frac{P(AB)}{P(A)} \; => \; P(AB) \; = \; P(B|A) * P(A)$$
$$Equating \; the \; two \; equations$$
$$P(A|B) * P(B) \; = \; P(B|A) * P(A)$$

We now get the Bayes theorem:

$$P(A|B) \; = \; \frac{P(B|A) * P(A)}{P(B)}$$

Now, instead of a single factor B, what if we have multiple factors i.e. B1, B2, B3, etc. We want to know what is the probability of getting an A given multiple factors B1, B2 etc. To handle this we can extend the Bayes Theorem as follows:

$$P(A|B_1 \cap B_2 \cap B_3 \dots) \; = \; \frac{P(B_1 \cap B_2 \cap B_3 \dots |A) * P(A)}{P(B_1 \cap B_2 \cap B_3 \dots)}$$

This basically means that given certain factors B1, B2, B3 etc, what is the probability of getting the outcome A.

Now, when we consider Naive Bayes, we say that each of these factors B1, B2, B3 etc given that A has happened already, are independent of each other, hence we can write the above equation as follows:

$$P(A|B_1 \cap B_2 \cap B_3 \dots) = \frac{P(B_1|A) * P(B_2|A) * P(B_3|A) \dots * P(A)}{P(B_1 \cap B_2 \cap B_3 \dots)}$$

Note: The denominator will stay constant, it will not change with A.

For a detailed example with numbers, please visit the following link: https://stackoverflow.com/questions/10059594/a-simple-explanation-of-naive-bayes-classification or refer to the class presentation.

Types of Naive Bayes Classifiers:

- Gaussian Naive Bayes: In Gaussian Naive Bayes, continuous values associated with each feature are assumed to be distributed according to a Gaussian distribution.
- Multinomial Naive Bayes: Features have discrete values. This is primarily used for document classification. In this case the features used are the frequency of the words present in the document.
- Bernoulli Naive Bayes: This is similar to the multinomial naive bayes but the predictors are boolean variables. This is also popular for document classification tasks.

Implementation of Gaussian Naive Bayes Classifier using scikit-learn

To implement Naive Bayes, we will use the same iris dataset we used earlier.

```
# load the iris dataset
from sklearn.datasets import load_iris
iris = load_iris()
```

```
# store the feature matrix (X) and response vector (y)
X = iris.data
y = iris.target
```

```
# splitting X and y into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)
```

```
# training the model on training set
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
gnb.fit(X_train, y_train)
```

```
GaussianNB(priors=None, var_smoothing=1e-09)
```

```
# making predictions on the testing set
y_pred = gnb.predict(X_test)
```

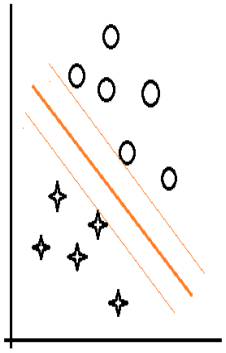
```
# comparing actual response values (y_test) with predicted response values (y_pred)
from sklearn import metrics
print("Gaussian Naive Bayes model accuracy:", metrics.accuracy_score(y_test, y_pred))
```

```
Gaussian Naive Bayes model accuracy: 0.9666666666666667
```

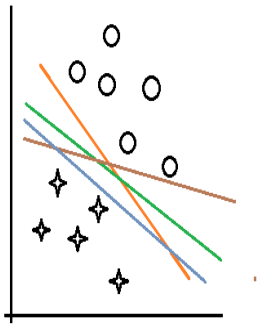
Reference: https://www.geeksforgeeks.org/naive-bayes-classifiers/

Support Vector Machines (SVM)

SVM is one of the supervised learning algorithms using training data to learn a model and make predictions. It learns a linear model. In two dimensions, the algorithm outputs a line which categorizes the data whereas in higher dimensions it generates a hyperplane to categorize the data.

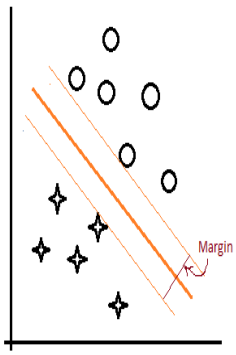


Considering the diagram above, we observe that the orange line classifies the two classes i.e. stars and circles. Any observation that lies to the left of the line falls in the star class and any observation to the right of the line falls under the circle class. In short, SVM separates the classes using a line or a hyperplane (for higher dimensions).

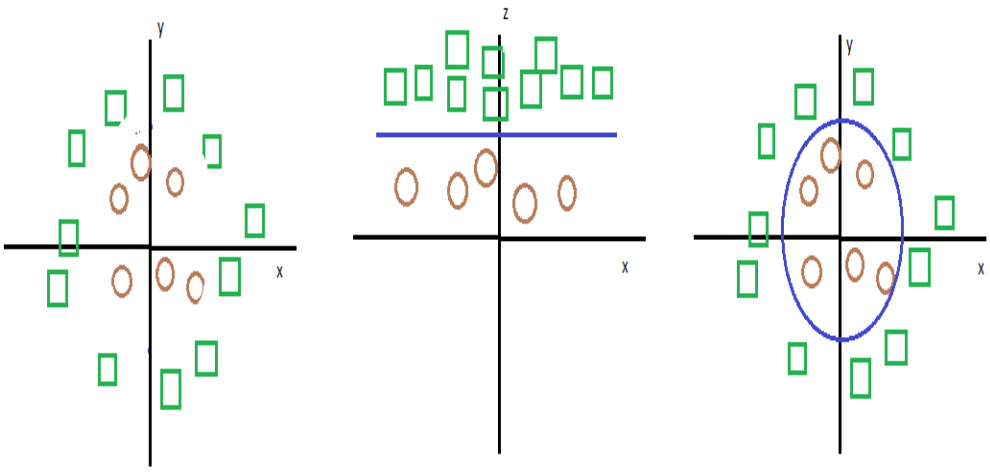


Many hyperplanes can be considered to categorize the two classes completely. However, using SVM, our objective is to find the hyperplane which has maximum distance from points of either class. Maximizing the distance increases the confidence that the future data points will be categorized correctly as well.

This distance between the hyperplane and the nearest data points is called 'margin'. Margin helps the algorithm decide the optimal hyperplane.



Now, lets consider what happens when the classes cannot be separated by a line/hyperplane.



No linear boundary separates the two classes

Transformation applied and one more dimension 'z' added; data now linearly separable

Transformed back to the xy plane; linear separator becomes a circle

Considering the leftmost diagram above, we can see that a linear boundary cannot categorize the two classes in the x-y plane. In order to be able to categorize the classes, another dimension z is added as can be observed from the middle figure. We can see that a clear separation is visible and a line can be used to categorize the two classes. When we now transform the data back into the x-y plane, the linear boundary becomes a circle as can be observed from the right most figure. These transformations are called 'kernels'. Using this 'kernel trick', the algorithm takes a low dimensional input and converts it into a higher dimensional space; thereby converting a non-separable problem to a separable one.

In order to understand the math behind SVM, you may want to refer to the following excellent link: https://www.svm-tutorial.com/2014/11/svm-understanding-math-part-1/

Next we will discuss about parameter used in SVM. There are three parameters we primarily tune in this algorithm:

- kernel - It defines the a distance measure between new data and the support vectors i.e. observations closest to the hyperplane. The dot product is the similarity measure used for a linear kernel since the distance is a linear combination of the inputs. When we consider higher dimensions other kernels such as a Polynomial Kernel and a Radial Kernel can be used that transform the input space into higher dimensions.
- C (Regularization parameter) - When the value of C is large, smaller-margin hyperplane will be considered since it stresses on getting all the training points classified correctly. On the other hand, a small value of C will consider a larger margin hyperplane, even if some points are misclassified by the hyperplane.
- gamma - The gamma parameter defines how far the influence of a each training observation affects the calculation of the optimal hyperplane. Low gamma values consider points even if they far away from the plausible hyperplane and high gamma values consider the points which are closer to the plausible hyperplane to get the optimal hyperplane.

```
# load the iris dataset
from sklearn.datasets import load_iris
iris = load_iris()
```

```
# store the feature matrix (X) and response vector (y)
X = iris.data
y = iris.target
```

```
# splitting X and y into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)
```

```
# training the model on training set
from sklearn import svm
# we create an instance of SVM
C = 1.0 # SVM regularization parameter
svc = svm.SVC(kernel='linear', C=1, gamma='auto')
```

```
svc.fit(X_train, y_train)
```

```
SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='linear',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

```
# making predictions on the testing set
y_pred = svc.predict(X_test)
```

```
# comparing actual response values (y_test) with predicted response values (y_pred)
from sklearn import metrics
print("SVM model accuracy:", metrics.accuracy_score(y_test, y_pred))
```

```
SVM model accuracy: 1.0
```

KNN, Naive Bayes and SVM implementation in Python for text data

The data we consider now is different than the structured data we have used till now. The data are multiple text files and each text file contains an article. The articles belong to different categories like crude, money and foreign exchange etc. What do we do if we are given a new article and we need to classify it into any of these categories? There are two issues to be handled here: first, we do not have our data in a single file. We need to bring data into a single file which can be used for further analysis. Secondly, the only data we have here is the text in the articles. How do we convert the text into a format which can be used by the machine learning algorithms? We need to convert the data into features such that we can make use of the algorithms for classification.

Lets start with reading in the data.

```
import os
path=r"\Users\anjal\Dropbox\PDS V3\Data\reuters_data"
```

We will start with collecting all the data in a single file first.

```
files=os.listdir(path)
```

The os.listdir(path) function returns all the files present in the folder specified by the 'path' argument.

We can observe the file names as follows:

```
files[0:10]
```

```
['.ipynb_checkpoints',
 '.RData',
 '.Rhistory',
 'training_crude_10011.txt',
 'training_crude_10078.txt',
 'training_crude_10080.txt',
 'training_crude_10106.txt',
 'training_crude_10168.txt',
 'training_crude_10190.txt',
 'training_crude_10192.txt']
```

We want content only from the .txt files; however, we can see that some other files are present here as well. We can clean them up i.e. keep only those files which have a .txt in their file name.

```
files=[x for x in files if '.txt' in x]
```

Now if we look at files, the unecessary files are not there anymore.

```
files[0:10]
```

```
['training_crude_10011.txt',
 'training_crude_10078.txt',
 'training_crude_10080.txt',
 'training_crude_10106.txt',
 'training_crude_10168.txt',
 'training_crude_10190.txt',
 'training_crude_10192.txt',
 'training_crude_10200.txt',
 'training_crude_10228.txt',
 'training_crude_1026.txt']
```

We clean the text for all the files as follows:

```
target=[]
article_text=[]
for file in files:
    if '.txt' not in file:continue # do not read the content from a file not having .txt in its name
    f=open(path+'\\'+file,encoding='latin-1') # for every file a handle is created
    article_text.append(" ".join([line.strip() for line in f if line.strip()!=""])) # removes lines without
text using strip()
                                                                                    # and returns a single
string for each article
    if "crude" in file: # if the file name has crude, then target list is appended with 'crude' else with
'money'
        target.append("crude")
    else:
        target.append("money")
    f.close()
```

Now we bind the two lists created above into a dataframe.

```
mydata=pd.DataFrame({'target':target,'article_text':article_text})
```

The dataframe 'mydata' consists of two columns, the 'article_text' column containing the text and the 'target' column consisting of the topic of the text.

```
mydata.head()
```

	target	article_text
0	crude	CANADA OIL EXPORTS RISE 20 PCT IN 1986 Canadia...
1	crude	BP <BP> DOES NOT PLAN TO HIKE STANDARD <...
2	crude	BP<BP> OFFER RAISES EXPECTATIONS FOR OIL VA...
3	crude	USX <X> SAYS TALKS ENDED WITH BRITISH PETRO...
4	crude	BP <BP> MAY HAVE TO RAISE BID - ANALYSTS B...

```
mydata.shape
```

```
(927, 2)
```

The data has 927 rows and 2 columns.

We can access any of the articles as follows:

```
mydata['article_text'][0]
```

```
'CANADA OIL EXPORTS RISE 20 PCT IN 1986 Canadian oil exports rose 20 pct in 1986 over the previous year to
33.96 mln cubic meters, while oil imports soared 25.2 pct to 20.58 mln cubic meters, Statistics Canada said.
Production, meanwhile, was unchanged from the previous year at 91.09 mln cubic feet. Natural gas exports
plunged 19.4 pct to 21.09 billion cubic meters, while Canadian sales slipped 4.1 pct to 48.09 billion cubic
meters. The federal agency said that in December oil production fell 4.0 pct to 7.73 mln cubic meters, while
exports rose 5.2 pct to 2.84 mln cubic meters and imports rose 12.3 pct to 2.1 mln cubic meters. Natural gas
exports fell 16.3 pct in the month 2.51 billion cubic meters and Canadian sales eased 10.2 pct to 5.25
billion cubic meters.'
```

```
mydata['target'].value_counts()
```

```
money    538
crude    389
Name: target, dtype: int64
```

We see that out of the 927 articles, 538 belong to the 'money' category and 389 belong to the 'crude' category.

To find the tentative performance on our model we will break the dataset into training and validation parts.

```
from sklearn.model_selection import train_test_split
```

```
article_train,article_test= train_test_split(mydata,test_size=0.2,random_state=2)
```

```
article_train.head()
```

	target	article_text
280	crude	MALAYSIA TO CUT OIL OUTPUT FURTHER, TRADERS SA...
688	money	BANKERS OPPOSE STRICT TAIWAN CURRENCY CONTROLS...
375	crude	OPEC WITHIN OUTPUT CEILING, SUBROTO SAYS Opec ...
665	money	U.K. MONEY MARKET SHORTAGE FORECAST AT 300 MLN...
589	money	CURRENCY FUTURES TO KEY OFF G-5, G-7 MEETINGS ...

```
article_train.reset_index(drop=True,inplace=True)
```

```
article_train.head()
```

	target	article_text
0	crude	MALAYSIA TO CUT OIL OUTPUT FURTHER, TRADERS SA...
1	money	BANKERS OPPOSE STRICT TAIWAN CURRENCY CONTROLS...
2	crude	OPEC WITHIN OUTPUT CEILING, SUBROTO SAYS Opec ...
3	money	U.K. MONEY MARKET SHORTAGE FORECAST AT 300 MLN...
4	money	CURRENCY FUTURES TO KEY OFF G-5, G-7 MEETINGS ...

```
y_train=(article_train['target']=='money').astype(int)
y_test=(article_test['target']=='money').astype(int)
```

Now we have the data in a column format in a single file. However, we have still not created features that can be used by the different machine learning algorithms for classification of articles.

One simple idea is that we consider every word across all the articles i.e. create a dictionary containing all the distinct words present across all the articles. We can then consider a word from the dictionary and count the number of times it is present in each article; e.g. considering the word 'currency', we can count the number of times it is present in each article and store this count. The count of different words can be used as features i.e. count features.

When doing this, we will come across many words that do not contribute to the article belonging to one category or the other. These words are called as stopwords e.g the, to, a etc. Such words are not useful for differentiating the articles and can be removed.

Also, we may want all words to be converted to their base form i.e. both 'playing' and 'played' should be converted to 'play'. This is referred to as lemmatization. We will use the WordNetLemmatizer for this.

We also want to remove punctuation from the text else punctuations will be considered as separate features.

```
from nltk.stem.wordnet import WordNetLemmatizer
from nltk.corpus import stopwords
from string import punctuation
from nltk.tokenize import word_tokenize # used to break sentences into words
lemma = WordNetLemmatizer()
my_stop=set(stopwords.words('english')+list(punctuation)) # we may want to add/remove words depending on the
business context
```

```
# Function to clean the text data
def split_into_lemmas(message):
    message=message.lower() # converts all text to lowercase
    words = word_tokenize(message)
    words_sans_stop=[]
    for word in words :
        if word in my_stop:continue
        words_sans_stop.append(word) # gets all words from the message except the stopwords
    return [lemma.lemmatize(word) for word in words_sans_stop] # lemmatized words returned
```

```
from sklearn.feature_extraction.text import CountVectorizer
```

CountVectorizer tokenizes the text and builds a vocabulary of words which are present in the text body.

The CountVectorizer function below uses the following parameters:

- 'analyzer = split_into_lemmas' sends the text to the function split_into_lemmas() and then gets every word which is cleaned.
- 'min_df = 20' argument is used so that only those words are considered which are present in the data at least 20 times.
- 'max_df = 500' indicates that the words which are present more than 500 times in the data will not be considered.
- 'stop_words = my_stop' argument takes the stop words defined earlier as its input. CountVectorizer just counts the occurrences of each word in its vocabulary. Hence common words like 'the', 'and', 'a' etc. become very important features since

their frequency is high even though they add little meaning to the text. The words considered as stop words are not used as features.

```
tf= CountVectorizer(analyzer=split_into_lemmas,min_df=20,max_df=500,stop_words=my_stop)
```

The fit() function is used to learn the vocabulary from the training text and the transform() function is used to encode each article text as a vector. This encoded vector has a length of the whole vocabulary and an integer count for the number of times each word appeared in the article text is returned for each article in this vector.

```
tf.fit(article_train['article_text'])
```

```
CountVectorizer(analyzer=<function split_into_lemmas at 0x000001E5DDAF3D08>,
        binary=False, decode_error='strict', dtype=<class 'numpy.int64'>,
        encoding='utf-8', input='content', lowercase=True, max_df=500,
        max_features=None, min_df=20, ngram_range=(1, 1),
        preprocessor=None,
        stop_words={'him', 'they', "weren't", 'won', 'were', 'didn', 'just', 'itself', 'against', '#',
"you'll", 'off', "hadn't", ',', 'weren', 'now', 'while', ']', 'and', 'other', 'ours', 'herself', 'because',
"you'd", 'into', "wasn't", 'below', 'my', 'if', 'on', 'his', 'theirs', 'our', 'here', 'before', '... 'its',
"shan't", 'where', 'isn', '{', 'does', 'once', 'over', "couldn't", ';', 'from', 'been', 'y'},
        strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
        tokenizer=None, vocabulary=None)
```

```
# print(tf.vocabulary_)
```

```
train_tf=tf.transform(article_train['article_text'])
```

```
train_tf
```

```
<741x677 sparse matrix of type '<class 'numpy.int64'>'
    with 36338 stored elements in Compressed Sparse Row format>
```

```
print(train_tf.shape)
print(type(train_tf))
print(train_tf.toarray())
```

```
(741, 677)
<class 'scipy.sparse.csr.csr_matrix'>
[[0 5 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [2 2 0 ... 0 0 0]
 ...
 [0 1 0 ... 0 0 0]
 [0 1 0 ... 0 0 0]
 [1 6 0 ... 0 1 0]]
```

We can then see that the encoded vector is a sparse matrix.

We observe that the array version of the encoded vector shows counts for different words.

```
x_train_tf = pd.DataFrame(train_tf.toarray(), columns=tf.get_feature_names())
print(x_train_tf.head())
```

```
   ''  's  --  ...  1  1.5  10  100  15  15.8  ...   work  working  world  \
0   0   5   0    0  1    0   2    0   0     0  ...      0        0      0
1   0   0   0    0  0    0   0    0   0     0  ...      0        0      0
2   2   2   0    0  0    0   0    0   0     2  ...      0        0      0
3   0   0   0    0  0    0   0    1   0     0  ...      0        0      0
4   7   1   2    0  0    0   0    0   0     0  ...      0        0      0

   worth  would  year  yen  yesterday  yet  york
0      0      3     1    0          0    0     0
1      0      1     0    0          0    0     0
2      0      3     0    0          0    0     0
3      0      3     0    0          0    0     0
4      0      5     0    7          0    0     0

[5 rows x 677 columns]
```

```
test_tf=tf.transform(article_test['article_text'])
```

```
test_tf.toarray()
```

```
array([[2, 1, 0, ..., 0, 1, 0],
       [0, 1, 0, ..., 0, 0, 0],
       [2, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 3, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [3, 1, 0, ..., 0, 0, 0]], dtype=int64)
```

```
x_test_tf=pd.DataFrame(test_tf.toarray(), columns=tf.get_feature_names())
```

```
x_test_tf.head()
```

	''	's	- -	...	1	1.5	10	100	15	15.8	...	work	working	world	worth	would	year	yen	yesterd
0	2	1	0	0	0	0	1	1	0	0	...	0	0	0	0	2	0	0	0
1	0	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0
2	2	0	0	0	0	0	0	0	0	0	...	0	0	1	0	1	0	0	0
3	0	3	0	0	0	0	0	0	0	0	...	0	0	0	0	0	7	0	0
4	0	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0

5 rows × 677 columns

In the code below you will notice that the number of columns created are 677 which is quite a lot. Text based features usually end up being too many.

```
print(x_train_tf.shape)
print(x_test_tf.shape)
```

```
(741, 677)
(186, 677)
```

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.svm import SVC
from sklearn.naive_bayes import MultinomialNB
```

KNN

```
knn=KNeighborsClassifier(n_neighbors=10)
```

```
knn.fit(x_train_tf,y_train)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=10, p=2,
           weights='uniform')
```

```
predictions=knn.predict(x_test_tf)
```

```
accuracy_score(y_test,predictions)
```

```
0.9623655913978495
```

SVM

```
clf_svm=SVC()
```

```
clf_svm.fit(x_train_tf,y_train)
```

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)
```

```
accuracy_score(y_test,clf_svm.predict(x_test_tf))
```

```
0.978494623655914
```

Naive Bayes

```
clf_nb=MultinomialNB()
```

```
clf_nb.fit(x_train_tf,y_train)
```

```
MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)
```

```
accuracy_score(y_test,clf_nb.predict(x_test_tf))
```

```
0.989247311827957
```

We observe that Naive Bayes performs better than the rest of the algorithms in text classification for this example. This does not mean that this will be the case for every problem. We should anyway use multiple algorithms and chose the one which performs best for our particular problem in discussion. Another point for this particular discussion, notice that we haven't tuned parameters. You can follow similar processes as used in earlier modules to do so.