**Twitter Sentiment Analysis**

Hi all,

Welcome to the course on **Twitter Sentiment Analysis: Practice Problem**. This is an extensive course on sentiment analysis where our task will be to classify a set of tweets into two categories:

* racist/sexist
* non-racist/sexist

The prerequisites for this course are basic knowledge of both Machine Learning and Python.

**What is Sentiment Analysis?**

Sentiment analysis (also known as opinion mining) is one of the many applications of Natural Language Processing. It is a set of methods and techniques used for extracting subjective information from text or speech, such as opinions or attitudes. In simple terms, it involves classifying a piece of text as positive, negative or neutral.



**Objective of the Course**

The course is designed to give you a hands-on experience in solving a sentiment analysis problem using Python. This course will introduce you to the skills and techniques required to solve text classification/sentiment analysis problems. You will be provided with a sufficient theory and practice material.

**Expectations from the Course**

The course is divided into below modules:

1. Text Preprocessing
2. Data Exploration
3. Feature Extraction
4. Model Building

These sections are supplemented with theory, coding examples, and exercises. Additionally, you will be provided with below resources:

* Dataset - Actual data to work on
* Jupyter Notebook - Complete code for the practical part of the code
* Discussion Forum Support - Your doubts and queries will be addressed by the course instructors

**How to best utilize this course?**

You should follow the below steps to extract maximum benefit out of this course:

1. Study the concepts and give it time to sink in.
2. Go through the practical content, download the dataset, and implement the solution on your own.
3. In case you need advice on something or you get stuck - use the discussion forum to ask the questions. In case of questions outside the scope of this course, please feel free to ask questions on the [discussion portal](https://discuss.analyticsvidhya.com/) of AnalyticsVidhya.

***Python Version****: Python 3.6 has been used for this course*

## Understand the Problem Statement

Let’s go through the problem statement once as it is very crucial to understand the objective before working on the dataset. The problem statement is as follows:

*The objective of this task is to detect hate speech in tweets. For the sake of simplicity, we say a tweet contains hate speech if it has a racist or sexist sentiment associated with it. So, the task is to classify racist or sexist tweets from other tweets.*

Formally, given a training sample of tweets and labels, where label ‘1’ denotes the tweet is racist/sexist and label ‘0’ denotes the tweet is not racist/sexist, your objective is to predict the labels on the given test dataset.

*You can access the problem statement and the data over*[*here*](https://datahack.analyticsvidhya.com/contest/practice-problem-twitter-sentiment-analysis/)*.*

*Note: The evaluation metric from this practice problem is F1-Score.*

## Table of Contents

1. Understand the Problem Statement
2. Tweets Preprocessing and Cleaning
   * Data Inspection
   * Data Cleaning
3. Story Generation and Visualization from Tweets
4. Extracting Features from Cleaned Tweets
   * Bag-of-Words
   * TF-IDF
   * Word Embeddings
5. Model Building: Sentiment Analysis
   * Logistic Regression
   * Support Vector Machine
   * RandomForest
   * XGBoost
6. Model Fine-tuning
7. Summary

## Loading Libraries and Data

Let’s load the libraries which will used in this course.

import re # for regular expressions

import nltk  # for text manipulation

import string

import warnings

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

pd.set\_option("display.max\_colwidth", 200)

warnings.filterwarnings("ignore", category=DeprecationWarning)

%matplotlib inline

Let’s read train and test datasets. Download data from [here](https://datahack.analyticsvidhya.com/contest/practice-problem-twitter-sentiment-analysis/).

train  = pd.read\_csv('train\_E6oV3lV.csv')

test = pd.read\_csv('test\_tweets\_anuFYb8.csv')

## Data Inspection

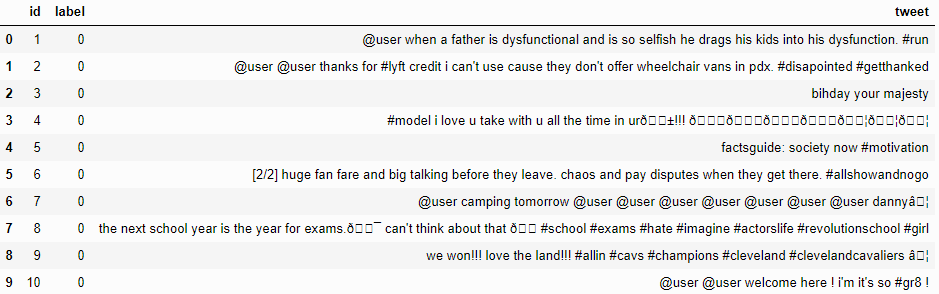
Text is a highly unstructured form of data, various types of noise are present in it and the data is not readily analyzable without any pre-processing. The entire process of cleaning and standardization of text, making it noise-free and ready for analysis is known as text preprocessing. We will divide it into 2 parts:

* Data Inspection
* Data Cleaning

### Data Inspection

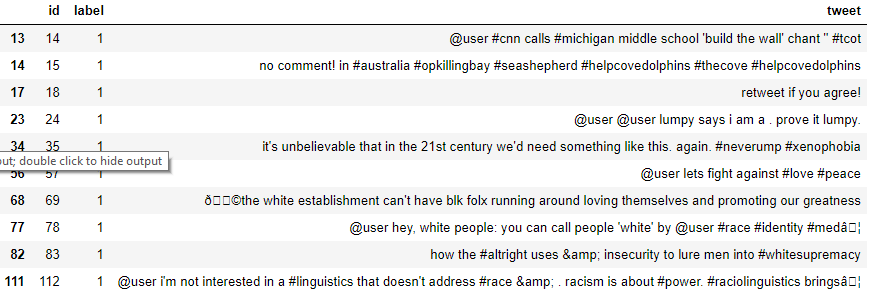
Let’s check out a few non racist/sexist tweets.

train[train['label'] == 0].head(10)



Now check out a few racist/sexist tweets.

train[train['label'] == 1].head(10)



There are quite a many words and characters which are not really required. So, we will try to keep only those words which are important and add value.

Let’s check dimensions of the train and test dataset.

train.shape, test.shape

((31962, 3), (17197, 2))

Train set has 31,962 tweets and test set has 17,197 tweets.

Let’s have a glimpse at label-distribution in the train dataset.

train["label"].value\_counts()

0 29720

1     2242

Name: label, dtype: int64

In the train dataset, we have 2,242 (~7%) tweets labeled as racist or sexist, and 29,720 (~93%) tweets labeled as non racist/sexist. So, it is an imbalanced classification challenge.

Now we will check the distribution of length of the tweets, in terms of words, in both train and test data.

length\_train = train['tweet'].str.len()

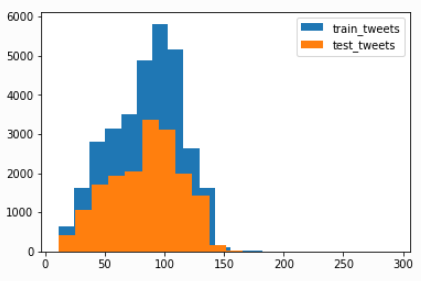
length\_test = test['tweet'].str.len()

plt.hist(length\_train, bins=20, label="train\_tweets")

plt.hist(length\_test, bins=20, label="test\_tweets")

plt.legend()

plt.show()



## Data Cleaning

In any natural language processing task, cleaning raw text data is an important step. It helps in getting rid of the unwanted words and characters which helps in obtaining better features. If we skip this step then there is a higher chance that you are working with noisy and inconsistent data. The objective of this step is to clean noise those are less relevant to find the sentiment of tweets such as punctuation, special characters, numbers, and terms which don’t carry much weightage in context to the text.

Before we begin cleaning, let’s first combine train and test datasets. Combining the datasets will make it convenient for us to preprocess the data. Later we will split it back into train and test data.

combi = train.append(test, ignore\_index=True)

combi.shape

(49159, 3)

Given below is a user-defined function to remove unwanted text patterns from the tweets.

def remove\_pattern(input\_txt, pattern):

    r = re.findall(pattern, input\_txt)

    for i in r:

        input\_txt = re.sub(i, '', input\_txt)

    return input\_txt

We will be following the steps below to clean the raw tweets in out data.

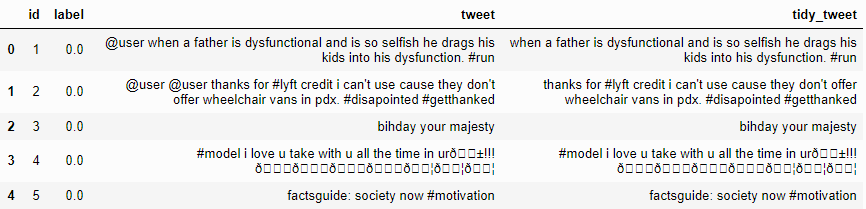
1. We will remove the twitter handles as they are already masked as @user due to privacy concerns. These twitter handles hardly give any information about the nature of the tweet.
2. We will also get rid of the punctuations, numbers and even special characters since they wouldn’t help in differentiating different types of tweets.
3. Most of the smaller words do not add much value. For example, ‘pdx’, ‘his’, ‘all’. So, we will try to remove them as well from our data.
4. Lastly, we will normalize the text data. For example, reducing terms like loves, loving, and lovable to their base word, i.e., ‘love’.are often used in the same context. If we can reduce them to their root word, which is ‘love’. It will help in reducing the total number of unique words in our data without losing a significant amount of information.

**1. Removing Twitter Handles (@user)**

Let’s create a new column tidy\_tweet, it will contain the cleaned and processed tweets. Note that we have passed “@[]\*” as the pattern to the remove\_pattern function. It is actually a regular expression which will pick any word starting with ‘@’.

combi['tidy\_tweet'] = np.vectorize(remove\_pattern)(combi['tweet'], "@[\w]\*")

combi.head()

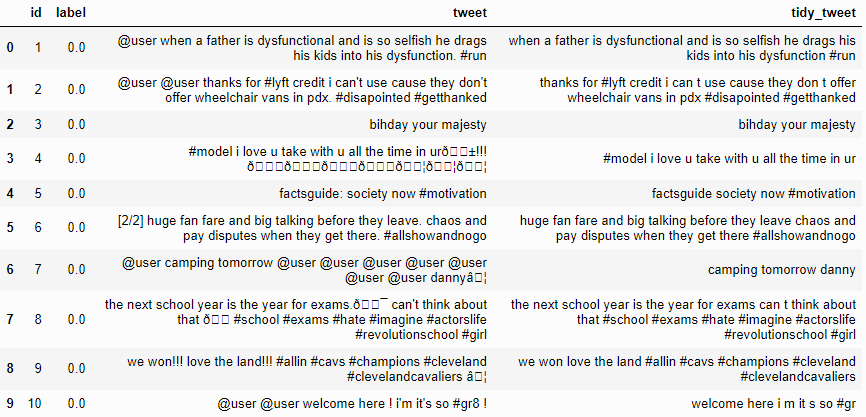


**2. Removing Punctuations, Numbers, and Special Characters**

Here we will replace everything except characters and hashtags with spaces. The regular expression “[^a-zA-Z#]” means anything except alphabets and ‘#’.

combi['tidy\_tweet'] = combi['tidy\_tweet'].str.replace("[^a-zA-Z#]", " ")

combi.head(10)



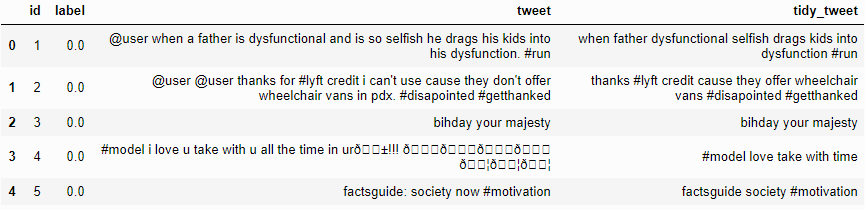
**3. Removing Short Words**

We have to be a little careful here in selecting the length of the words which we want to remove. So, I have decided to remove all the words having length 3 or less. For example, terms like “hmm”, “oh” are of very little use. It is better to get rid of them.

combi['tidy\_tweet'] = combi['tidy\_tweet'].apply(lambda x: ' '.join([w for w in x.split() if len(w)>3]))

Let’s take another look at the first few rows of the combined dataframe.

combi.head()



You can see the difference between the raw tweets and the cleaned tweets (tidy\_tweet) quite clearly. Only the important words in the tweets have been retained and the noise (numbers, punctuations, and special characters) has been removed.

**4. Text Normalization**

Here we will use nltk’s PorterStemmer() function to normalize the tweets. But before that we will have to tokenize the tweets. Tokens are individual terms or words, and tokenization is the process of splitting a string of text into tokens.

tokenized\_tweet = combi['tidy\_tweet'].apply(lambda x: x.split()) # tokenizing

tokenized\_tweet.head()

0 [when, father, dysfunctional, selfish, drags, kids, into, dysfunction, #run]

1    [thanks, #lyft, credit, cause, they, offer, wheelchair, vans, #disapointed, #getthanked]

2                                                                     [bihday, your, majesty]

3                                                            [#model, love, take, with, time]

4                                                          [factsguide, society, #motivation]

Name: tidy\_tweet, dtype: object

Now we can normalize the tokenized tweets.

from nltk.stem.porter import \*

stemmer = PorterStemmer()

tokenized\_tweet = tokenized\_tweet.apply(lambda x: [stemmer.stem(i) for i in x]) # stemming

Now let’s stitch these tokens back together. It can easily be done using nltk’s MosesDetokenizer function.

for i in range(len(tokenized\_tweet)):

    tokenized\_tweet[i] = ' '.join(tokenized\_tweet[i])

combi['tidy\_tweet'] = tokenized\_tweet

## Story Generation and Visualization from Tweets

In this section, we will explore the cleaned tweets. Exploring and visualizing data, no matter whether its text or any other data, is an essential step in gaining insights. Do not limit yourself to only these methods told in this course, feel free to explore the data as much as possible.

Before we begin exploration, we must think and ask questions related to the data in hand. A few probable questions are as follows:

* What are the most common words in the entire dataset?
* What are the most common words in the dataset for negative and positive tweets, respectively?
* How many hashtags are there in a tweet?
* Which trends are associated with my dataset?
* Which trends are associated with either of the sentiments? Are they compatible with the sentiments?

**A) Understanding the common words used in the tweets: WordCloud**

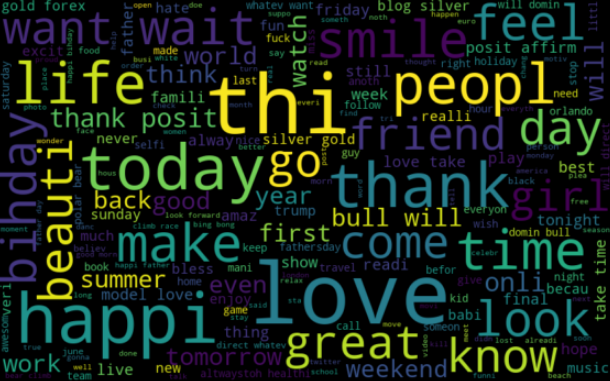
Now I want to see how well the given sentiments are distributed across the train dataset. One way to accomplish this task is by understanding the common words by plotting wordclouds.

A wordcloud is a visualization wherein the most frequent words appear in large size and the less frequent words appear in smaller sizes.

Let’s visualize all the words our data using the wordcloud plot.

all\_words = ' '.join([text for text in combi['tidy\_tweet']]) from wordcloud import WordCloud wordcloud = WordCloud(width=800, height=500, random\_state=21, max\_font\_size=110).generate(all\_words)

plt.figure(figsize=(10, 7)) plt.imshow(wordcloud, interpolation="bilinear") plt.axis('off') plt.show()

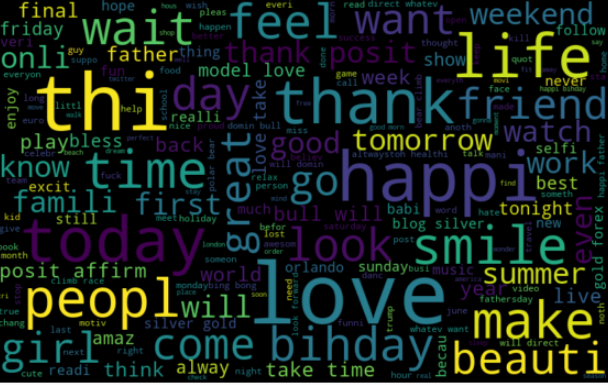


We can see most of the words are positive or neutral. Words like love, great, friend, life are the most frequent ones. It doesn’t give us any idea about the words associated with the racist/sexist tweets. Hence, we will plot separate wordclouds for both the classes (racist/sexist or not) in our train data.

**B) Words in non racist/sexist tweets**

normal\_words =' '.join([text for text in combi['tidy\_tweet'][combi['label'] == 0]])

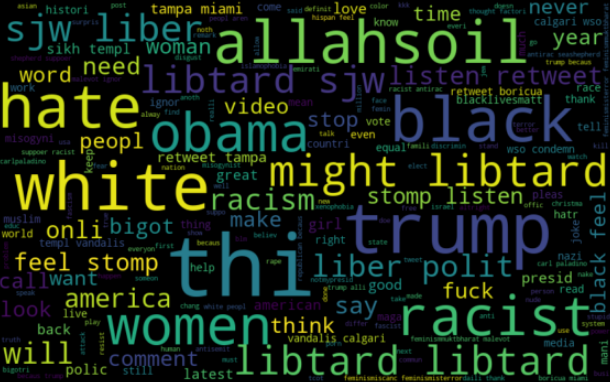
wordcloud = WordCloud(width=800, height=500, random\_state=21, max\_font\_size=110).generate(normal\_words) plt.figure(figsize=(10, 7)) plt.imshow(wordcloud, interpolation="bilinear") plt.axis('off') plt.show()



Most of the frequent words are compatible with the sentiment, i.e, non-racist/sexists tweets. Similarly, we will plot the word cloud for the other sentiment. Expect to see negative, racist, and sexist terms.

**C) Racist/Sexist Tweets**

negative\_words = ' '.join([text for text in combi['tidy\_tweet'][combi['label'] == 1]]) wordcloud = WordCloud(width=800, height=500, random\_state=21, max\_font\_size=110).generate(negative\_words) plt.figure(figsize=(10, 7)) plt.imshow(wordcloud, interpolation="bilinear") plt.axis('off') plt.show()



As we can clearly see, most of the words have negative connotations. So, it seems we have a pretty good text data to work on. Next we will the hashtags/trends in our twitter data.

**D) Understanding the impact of Hashtags on tweets sentiment**

Hashtags in twitter are synonymous with the ongoing trends on twitter at any particular point in time. We should try to check whether these hashtags add any value to our sentiment analysis task, i.e., they help in distinguishing tweets into the different sentiments.

For instance, given below is a tweet from our dataset:



The tweet seems sexist in nature and the hashtags in the tweet convey the same feeling.

We will store all the trend terms in two separate lists — one for non-racist/sexist tweets and the other for racist/sexist tweets.

# function to collect hashtags def hashtag\_extract(x): hashtags = [] # Loop over the words in the tweet for i in x: ht = re.findall(r"#(\w+)", i) hashtags.append(ht) return hashtags

# extracting hashtags from non racist/sexist tweets

HT\_regular = hashtag\_extract(combi['tidy\_tweet'][combi['label'] == 0])

# extracting hashtags from racist/sexist tweets HT\_negative = hashtag\_extract(combi['tidy\_tweet'][combi['label'] == 1])

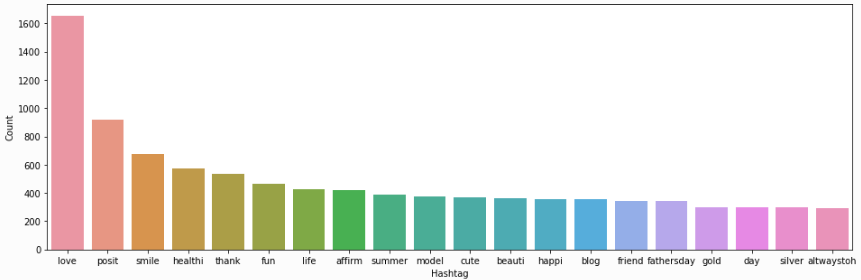
# unnesting list HT\_regular = sum(HT\_regular,[]) HT\_negative = sum(HT\_negative,[])

Now that we have prepared our lists of hashtags for both the sentiments, we can plot the top ‘n’ hashtags. So, first let’s check the hashtags in the non-racist/sexist tweets.

**Non-Racist/Sexist Tweets**

a = nltk.FreqDist(HT\_regular) d = pd.DataFrame({'Hashtag': list(a.keys()), 'Count': list(a.values())})

# selecting top 20 most frequent hashtags d = d.nlargest(columns="Count", n = 20) plt.figure(figsize=(16,5)) ax = sns.barplot(data=d, x= "Hashtag", y = "Count") ax.set(ylabel = 'Count') plt.show()

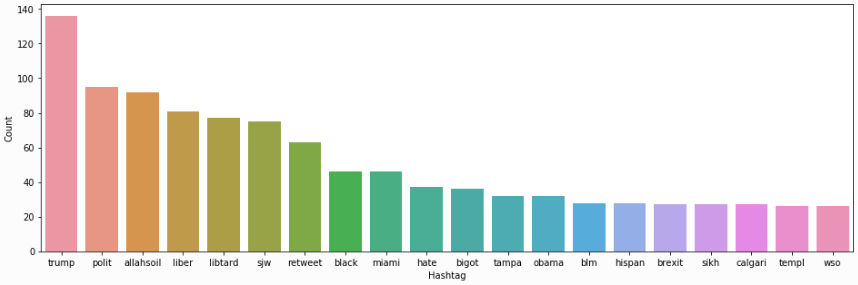


All these hashtags are positive and it makes sense. I am expecting negative terms in the plot of the second list. Let’s check the most frequent hashtags appearing in the racist/sexist tweets.

**Racist/Sexist Tweets**

b = nltk.FreqDist(HT\_negative) e = pd.DataFrame({'Hashtag': list(b.keys()), 'Count': list(b.values())})

# selecting top 20 most frequent hashtags e = e.nlargest(columns="Count", n = 20) plt.figure(figsize=(16,5)) ax = sns.barplot(data=e, x= "Hashtag", y = "Count")



As expected, most of the terms are negative with a few neutral terms as well. So, it’s not a bad idea to keep these hashtags in our data as they contain useful information. Next, we will try to extract features from the tokenized tweets.

## Bag-of-Words Features

To analyse a preprocessed data, it needs to be converted into features. Depending upon the usage, text features can be constructed using assorted techniques – Bag of Words, TF-IDF, and Word Embeddings. Read on to understand these techniques in detail.

from sklearn.feature\_extraction.text import TfidfVectorizer, CountVectorizer import gensim

Let’s start with the **Bag-of-Words** Features.

Consider a Corpus C of D documents {d1,d2…..dD} and N unique tokens extracted out of the corpus C. The N tokens (words) will form a dictionary and the size of the bag-of-words matrix M will be given by D X N. Each row in the matrix M contains the frequency of tokens in document D(i).

Let us understand this using a simple example.

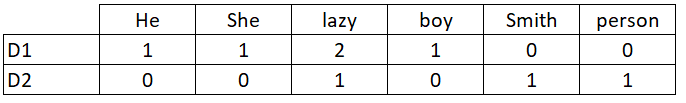
D1: He is a lazy boy. She is also lazy.

D2: Smith is a lazy person.

The dictionary created would be a list of unique tokens in the corpus =[‘He’,’She’,’lazy’,’boy’,’Smith’,’person’]

Here, D=2, N=6

The matrix M of size 2 X 6 will be represented as –



Now the columns in the above matrix can be used as features to build a classification model.

bow\_vectorizer = CountVectorizer(max\_df=0.90, min\_df=2, max\_features=1000, stop\_words='english') bow = bow\_vectorizer.fit\_transform(combi['tidy\_tweet']) bow.shape

(49159, 1000)

## TF-IDF Features

This is another method which is based on the frequency method but it is different to the bag-of-words approach in the sense that it takes into account not just the occurrence of a word in a single document (or tweet) but in the entire corpus.

TF-IDF works by penalising the common words by assigning them lower weights while giving importance to words which are rare in the entire corpus but appear in good numbers in few documents.

Let’s have a look at the important terms related to TF-IDF:

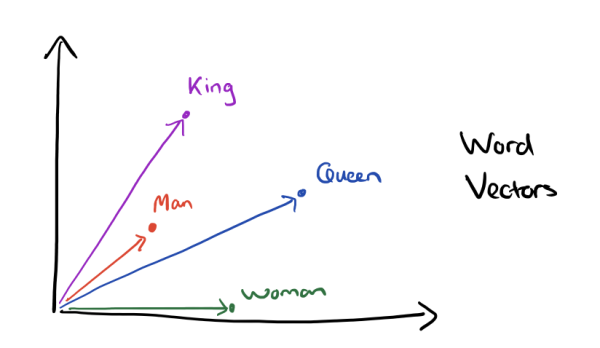
* TF = (Number of times term t appears in a document)/(Number of terms in the document)
* IDF = log(N/n), where, N is the number of documents and n is the number of documents a term t has appeared in.
* TF-IDF = TF\*IDF

tfidf\_vectorizer = TfidfVectorizer(max\_df=0.90, min\_df=2, max\_features=1000, stop\_words='english') tfidf = tfidf\_vectorizer.fit\_transform(combi['tidy\_tweet']) tfidf.shape

(49159, 1000)

## Word2Vec Features

Word embeddings are the modern way of representing words as vectors. The objective of word embeddings is to redefine the high dimensional word features into low dimensional feature vectors by preserving the contextual similarity in the corpus. They are able to achieve tasks like **King -man +woman = Queen**, which is mind-blowing.



The advantages of using word embeddings over BOW or TF-IDF are:

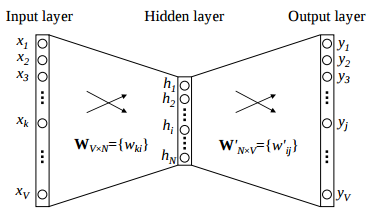
1. Dimensionality reduction - significant reduction in the no. of features required to build a model.
2. It capture meanings of the words, semantic relationships and the different types of contexts they are used in.

**1. Word2Vec Embeddings**

Word2Vec is not a single algorithm but a combination of two techniques – **CBOW (Continuous bag of words)** and **Skip-gram model**. Both of these are shallow neural networks which map word(s) to the target variable which is also a word(s). Both of these techniques learn weights which act as word vector representations.

CBOW tends to predict the probability of a word given a context. A context may be a single adjacent word or a group of surrounding words. The Skip-gram model works in the reverse manner, it tries to predict the context for a given word.

Below is a diagrammatic representation of a 1-word context window Word2Vec model.



There are three laters: - an input layer, - a hidden layer, and - an output layer.

The input layer and the output, both are one- hot encoded of size [1 X V], where V is the size of the vocabulary (no. of unique words in the corpus). The output layer is a softmax layer which is used to sum the probabilities obtained in the output layer to 1. The weights learned by the model are then used as the word-vectors.

We will go ahead with the Skip-gram model as it has the following advantages:

1. It can capture two semantics for a single word. i.e it will have two vector representations of ‘apple’. One for the company Apple and the other for the fruit.
2. Skip-gram with negative sub-sampling outperforms CBOW generally.

We will train a Word2Vec model on our data to obtain vector representations for all the unique words present in our corpus. There is one more option of using **pre-trained word vectors** instead of training our own model. Some of the freely available pre-trained vectors are:

1. [Google News Word Vectors](https://code.google.com/archive/p/word2vec/)
2. [Freebase names](https://code.google.com/archive/p/word2vec/)
3. [DBPedia vectors (wiki2vec)](https://github.com/idio/wiki2vec#prebuilt-models)

However, for this course, we will train our own word vectors since size of the pre-trained word vectors is generally huge.

Let’s train a Word2Vec model on our corpus.

tokenized\_tweet = combi['tidy\_tweet'].apply(lambda x: x.split()) # tokenizing

model\_w2v = gensim.models.Word2Vec(

            tokenized\_tweet,

            size=200, # desired no. of features/independent variables

            window=5, # context window size

            min\_count=2,

            sg = 1, # 1 for skip-gram model

            hs = 0,

            negative = 10, # for negative sampling

            workers= 2, # no.of cores

            seed = 34)

model\_w2v.train(tokenized\_tweet, total\_examples= len(combi['tidy\_tweet']), epochs=20)

Let’s play a bit with our Word2Vec model and see how does it perform. We will specify a word and the model will pull out the most similar words from the corpus.

model\_w2v.wv.most\_similar(positive="dinner")

[('spaghetti', 0.5658013820648193),

 ('#avocado', 0.5653101801872253),

 ('#cellar', 0.5547047853469849),

 ('cookout', 0.5511012077331543),

 ('noodl', 0.5489310622215271),

 ('melani', 0.547335147857666),

 ('#biall', 0.546588122844696),

 ('gown', 0.5430378913879395),

 ('#foodcoma', 0.5413259267807007),

 ('spinach', 0.5404106378555298)]

model\_w2v.wv.most\_similar(positive="trump")

[('donald', 0.5460224747657776),

 ('phoni', 0.5259741544723511),

 ('unfit', 0.5246831178665161),

 ('unstabl', 0.5200092792510986),

 ('melo', 0.5164598226547241),

 ('potu', 0.515663743019104),

 ('unfavor', 0.5101866126060486),

 ('hillari', 0.5076572299003601),

 ('#delegaterevolt', 0.5051215291023254),

 ('jibe', 0.5032232999801636)]

From the above two examples, we can see that our word2vec model does a good job of finding the most similar words for a given word. But how is it able to do so? That’s because it has learned vectors for every unique word in our data and it uses cosine similarity to find out the most similar vectors (words).

Let’s check the vector representation of any word from our corpus.

model\_w2v['food']

array([ 0.15007606, 0.14879119, -0.51990169, 0.0058126 , 0.55208695,

        0.25509292,  0.07453623, -0.13314515, -0.0254878 , -0.8908332 ,

       -0.02133939,  0.89018846, -0.14146671, -0.34468454, -0.47961965,

        0.03066353,  0.35275963, -0.57058471,  1.03962195,  0.12884547,

        0.59137058, -0.34608129,  0.92952412,  0.653409  ,  0.28881171,

        0.55403763, -0.34932148, -0.73746759, -0.38691986,  0.68097055,

        0.13209412, -0.44750923,  0.29219675,  0.63686401, -0.30297968,

       -0.03825129, 0.62068158, -0.62892973,  0.15462588,  0.1439736 ,

       -0.10439953, 0.02763413, -0.45384923, -0.00450154,  0.20950264,

        0.3756882 , -0.23810436,  0.22051652, -0.30991587,  0.41085938,

        0.03515792,  0.07115516, -0.41318294,  0.62097359,  0.25477788,

       -0.20776244, -0.72082233,  0.13771147, -0.71929592,  0.11429328,

        0.2206036 ,  0.24659269, -0.96678174,  0.98984069, -0.02419505,

        0.41572711, -0.17525066,  0.37907407, -0.21024323, -0.62315828,

        0.39220247,  0.20188518,  0.34157351, -0.53269279,  0.03763888,

       -0.29426393,  0.2054625 , -0.43705827,  0.25044039, -0.24679622,

        0.67416072, -0.67289978, -0.48204809, -0.97117424, -0.30714279,

        0.42861882, -0.06790878, -0.14522843, -0.42536703,  0.47236034,

       -0.63712251, -0.96000892,  0.07974151,  0.25426412,  0.1906969 ,

        0.48031083,  0.17527717,  0.42015558, -0.87493235,  0.24464223,

        1.08253467, -0.17896391, -0.13219987,  0.3883971 ,  0.02994647,

        0.3245416 , -0.29796079, -0.49378172,  0.80830252, -0.10045221,

        0.38219792,  0.27322048,  0.04144816,  0.2422674 ,  0.64382648,

       -0.3711468 ,  0.49309272, -0.60873193, -0.46271139,  0.47515544,

        0.09082295, -0.25910863,  0.05061042,  0.2590718 ,  0.00440796,

       -0.22002698,  0.37606686,  0.34843382, -0.2715351 , -0.08971476,

        0.06571087, -0.18973967, -0.35962474,  0.00556282, -0.16604713,

        0.45029792,  0.32906559,  0.30952054,  0.74116272, -1.25179255,

        0.31527328,  0.419824  , -0.53012145,  0.33158424,  0.77047271,

       -0.54991871,  0.76820081,  0.50645679,  0.49994075,  0.60128528,

       -0.00750624,  0.12363218, -0.6370244 ,  0.29522142, -0.29805934,

       -0.11819029,  0.18606399, -0.49186444,  0.01151775,  0.12069881,

       -0.80829483,  0.13237464,  0.09028237, -0.44283563, -0.20080626,

       -0.69564581,  0.53329039,  0.30282548,  0.08716848,  0.53611708,

       -0.96495646,  0.0116515 ,  0.35164613, -0.67445683,  0.13109499,

       -0.4415434 , -0.12365375, -0.10757735,  0.72350281, -0.15514924,

       -0.26206878,  0.38556293,  0.17972343,  0.36686096, -0.25163049,

       -0.45159557,  0.123597  ,  0.34224105,  0.20177038,  0.38732901,

        0.01403304, -0.02501263,  0.1044465 , -0.38610229,  0.41346085,

        0.59413546, -0.06027097, -0.12144137,  0.48135847,  0.08364052],

 dtype=float32)

len(model\_w2v['food']) #The length of the vector is 200

200

**Preparing Vectors for Tweets**

Since our data contains tweets and not just words, we’ll have to figure out a way to use the word vectors from word2vec model to create vector representation for an entire tweet. There is a simple solution to this problem, we can simply take mean of all the word vectors present in the tweet. The length of the resultant vector will be the same, i.e. 200. We will repeat the same process for all the tweets in our data and obtain their vectors. Now we have 200 word2vec features for our data.

We will use the below function to create a vector for each tweet by taking the average of the vectors of the words present in the tweet.

def word\_vector(tokens, size):

    vec = np.zeros(size).reshape((1, size))

    count = 0.

    for word in tokens:

        try:

            vec += model\_w2v[word].reshape((1, size))

            count += 1.

        except KeyError: # handling the case where the token is not in vocabulary                                     continue

    if count != 0:

        vec /= count

    return vec

Preparing word2vec feature set…

wordvec\_arrays = np.zeros((len(tokenized\_tweet), 200))

for i in range(len(tokenized\_tweet)):

    wordvec\_arrays[i,:] = word\_vector(tokenized\_tweet[i], 200)

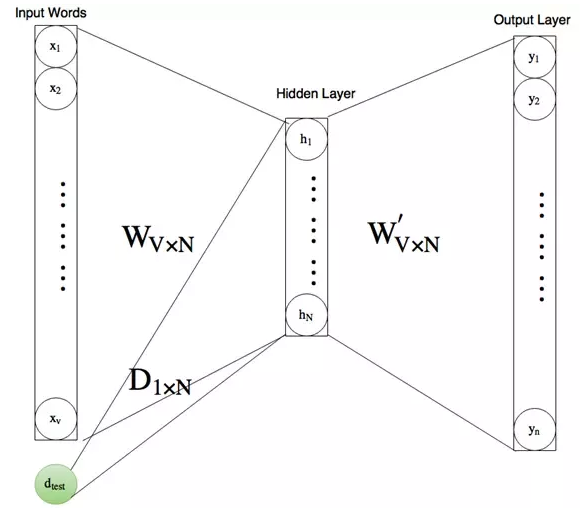
    wordvec\_df = pd.DataFrame(wordvec\_arrays) wordvec\_df.shape

(49159, 200)

Now we have 200 new features, whereas in Bag of Words and TF-IDF we had 1000 features.

**2. Doc2Vec Embedding**

Doc2Vec model is an unsupervised algorithm to generate vectors for sentence/paragraphs/documents. This approach is an extension of the word2vec. The major difference between the two is that doc2vec provides an additional context which is unique for every document in the corpus. This additional context is nothing but another feature vector for the whole document. This document vector is trained along with the word vectors.



Let’s load the required libraries.

from tqdm import

tqdm tqdm.pandas(desc="progress-bar")

from gensim.models.doc2vec import LabeledSentence

To implement doc2vec, we have to **labelise** or **tag** each tokenised tweet with unique IDs. We can do so by using Gensim’s *LabeledSentence()* function.

def add\_label(twt):

    output = []

    for i, s in zip(twt.index, twt):

        output.append(LabeledSentence(s, ["tweet\_" + str(i)]))

    return output

labeled\_tweets = add\_label(tokenized\_tweet) # label all the tweets

Let’s have a look at the result.

labeled\_tweets[:6]

[LabeledSentence(words=['when', 'father', 'dysfunct', 'selfish', 'drag', 'kid', 'into', 'dysfunct', '#run'], tags=['tweet\_0']),

 LabeledSentence(words=['thank', '#lyft', 'credit', 'caus', 'they', 'offer', 'wheelchair', 'van', '#disapoint', '#getthank'], tags=['tweet\_1']),

LabeledSentence(words=['bihday', 'your', 'majesti'], tags=['tweet\_2']), LabeledSentence(words=['#model', 'love', 'take', 'with', 'time'], tags=['tweet\_3']), LabeledSentence(words=['factsguid', 'societi', '#motiv'], tags=['tweet\_4']), LabeledSentence(words=['huge', 'fare', 'talk', 'befor', 'they', 'leav', 'chao', 'disput', 'when', 'they', 'there', '#allshowandnogo'], tags=['tweet\_5'])]

Now let’s train a **doc2vec** model.

model\_d2v = gensim.models.Doc2Vec(dm=1, # dm = 1 for ‘distributed memory’ model dm\_mean=1, # dm = 1 for using mean of the context word vectors size=200, # no. of desired features

window=5, # width of the context window

negative=7, # if > 0 then negative sampling will be used                                 min\_count=5, # Ignores all words with total frequency lower than 2.

workers=3, # no. of cores

alpha=0.1, # learning rate

seed = 23)

model\_d2v.build\_vocab([i for i in tqdm(labeled\_tweets)])

model\_d2v.train(labeled\_tweets, total\_examples= len(combi['tidy\_tweet']), epochs=15)

**Preparing doc2vec Feature Set**

docvec\_arrays = np.zeros((len(tokenized\_tweet), 200))

for i in range(len(combi)):

    docvec\_arrays[i,:] = model\_d2v.docvecs[i].reshape((1,200))

docvec\_df = pd.DataFrame(docvec\_arrays)

docvec\_df.shape

(49159, 200)

## Modeling

We are now done with all the pre-modeling stages required to get the data in the proper form and shape. We will be building models on the datasets with different feature sets prepared in the earlier sections — Bag-of-Words, TF-IDF, word2vec vectors, and doc2vec vectors. We will use the following algorithms to build models:

1. Logistic Regression
2. Support Vector Machine
3. RandomForest
4. XGBoost

**Evaluation Metric**

**F1 score** is being used as the evaluation metric. It is the weighted average of Precision and Recall. Therefore, this score takes both false positives and false negatives into account. It is suitable for uneven class distribution problems.

The important components of F1 score are:

1. True Positives (TP) - These are the correctly predicted positive values which means that the value of actual class is yes and the value of predicted class is also yes.
2. True Negatives (TN) - These are the correctly predicted negative values which means that the value of actual class is no and value of predicted class is also no.
3. False Positives (FP) – When actual class is no and predicted class is yes.
4. False Negatives (FN) – When actual class is yes but predicted class in no.

Precision = TP/TP+FP

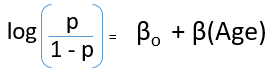
Recall = TP/TP+FN

F1 Score = 2*(Recall*Precision) / (Recall + Precision)

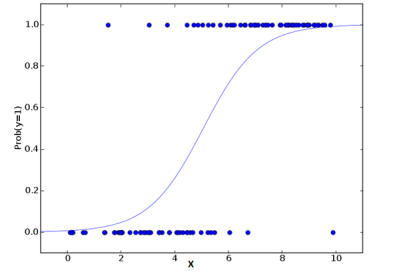
## Logistic Regression

Logistic Regression is a classification algorithm. It is used to predict a binary outcome (1 / 0, Yes / No, True / False) given a set of independent variables. You can also think of logistic regression as a special case of linear regression when the outcome variable is categorical, where we are using log of odds as the dependent variable. In simple words, it predicts the probability of occurrence of an event by fitting data to a logit function.

The following equation is used in Logistic Regression:



A typical logistic model plot is shown below. You can see probability never goes below 0 and above 1.



Read this [article](https://www.analyticsvidhya.com/blog/2015/11/beginners-guide-on-logistic-regression-in-r/) to know more about Logistic Regression.

from sklearn.linear\_model import LogisticRegression from sklearn.model\_selection import train\_test\_split from sklearn.metrics import f1\_score

**Bag-of-Words Features**

We will first try to fit the logistic regression model on the Bag-of\_Words (BoW) features.

# Extracting train and test BoW features train\_bow = bow[:31962,:] test\_bow = bow[31962:,:]

# splitting data into training and validation set xtrain\_bow, xvalid\_bow, ytrain, yvalid = train\_test\_split(train\_bow, train['label'], random\_state=42, test\_size=0.3)

lreg = LogisticRegression()

# training the model lreg.fit(xtrain\_bow, ytrain)

prediction = lreg.predict\_proba(xvalid\_bow) # predicting on the validation set prediction\_int = prediction[:,1] >= 0.3 # if prediction is greater than or equal to 0.3 than 1 else 0 prediction\_int = prediction\_int.astype(np.int)

f1\_score(yvalid, prediction\_int) # calculating f1 score for the validation set

0.531

Now let’s make predictions for the test dataset and create a submission file.

test\_pred = lreg.predict\_proba(test\_bow) test\_pred\_int = test\_pred[:,1] >= 0.3 test\_pred\_int = test\_pred\_int.astype(np.int) test['label'] = test\_pred\_int submission = test[['id','label']] submission.to\_csv('sub\_lreg\_bow.csv', index=False) # writing data to a CSV file

Public Leaderboard F1 Score: 0.567

**TF-IDF Features**

We’ll follow the same steps as above, but now for the TF-IDF feature set.

train\_tfidf = tfidf[:31962,:] test\_tfidf = tfidf[31962:,:]

xtrain\_tfidf = train\_tfidf[ytrain.index] xvalid\_tfidf = train\_tfidf[yvalid.index]

lreg.fit(xtrain\_tfidf, ytrain)

prediction = lreg.predict\_proba(xvalid\_tfidf) prediction\_int = prediction[:,1] >= 0.3 prediction\_int = prediction\_int.astype(np.int)

f1\_score(yvalid, prediction\_int) # calculating f1 score for the validation set

0.544

Public Leaderboard F1 Score: 0.564

**Word2Vec Features**

train\_w2v = wordvec\_df.iloc[:31962,:] test\_w2v = wordvec\_df.iloc[31962:,:]

xtrain\_w2v = train\_w2v.iloc[ytrain.index,:] xvalid\_w2v = train\_w2v.iloc[yvalid.index,:]

lreg.fit(xtrain\_w2v, ytrain)

prediction = lreg.predict\_proba(xvalid\_w2v) prediction\_int = prediction[:,1] >= 0.3 prediction\_int = prediction\_int.astype(np.int) f1\_score(yvalid, prediction\_int)

0.622

Public Leaderboard F1 Score: 0.661

**Doc2Vec Features**

train\_d2v = docvec\_df.iloc[:31962,:] test\_d2v = docvec\_df.iloc[31962:,:]

xtrain\_d2v = train\_d2v.iloc[ytrain.index,:] xvalid\_d2v = train\_d2v.iloc[yvalid.index,:]

lreg.fit(xtrain\_d2v, ytrain)

prediction = lreg.predict\_proba(xvalid\_d2v) prediction\_int = prediction[:,1] >= 0.3 prediction\_int = prediction\_int.astype(np.int) f1\_score(yvalid, prediction\_int)

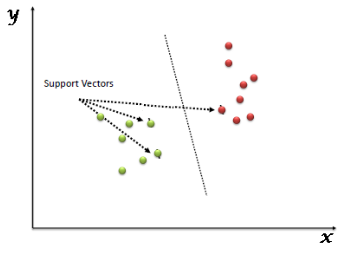
0.367

Public Leaderboard F1 Score: 0.381

Doc2Vec features do not seem to be capturing the right signals as both the F1-scores, on validation set and on public leaderboard are quite low.

## Support Vector Machine (SVM)

Support Vector Machine (SVM) is a supervised machine learning algorithm which can be used for both classification or regression challenges. However, 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 the number of features you have) with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyper-plane that differentiate the two classes as shown in the plot below:



Refer this [article](https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/) to learn more about SVM. Now we will implement SVM on our data using the scikit-learn library.

from sklearn import svm

**Bag-of-Words Features**

svc = svm.SVC(kernel='linear', C=1, probability=True).fit(xtrain\_bow, ytrain)

prediction = svc.predict\_proba(xvalid\_bow)

prediction\_int = prediction[:,1] >= 0.3

prediction\_int = prediction\_int.astype(np.int)

f1\_score(yvalid, prediction\_int)

0.508

Again let’s make predictions for the test dataset and create another submission file.

test\_pred = svc.predict\_proba(test\_bow)

test\_pred\_int = test\_pred[:,1] >= 0.3

test\_pred\_int = test\_pred\_int.astype(np.int)

test['label'] = test\_pred\_int

submission = test[['id','label']]

submission.to\_csv('sub\_svm\_bow.csv', index=False)

Public Leaderboard F1 Score: 0.554

Here both validation score and leaderboard score are slightly lesser than the Logistic Regression scores for bag-of-words features.

**TF-IDF Features**

svc = svm.SVC(kernel='linear',

C=1, probability=True).fit(xtrain\_tfidf, ytrain)

prediction = svc.predict\_proba(xvalid\_tfidf)

prediction\_int = prediction[:,1] >= 0.3

prediction\_int = prediction\_int.astype(np.int)

f1\_score(yvalid, prediction\_int)

0.51

Public Leaderboard F1 Score: 0.546

**Word2Vec Features**

svc = svm.SVC(kernel='linear', C=1, probability=True).fit(xtrain\_w2v, ytrain)

prediction = svc.predict\_proba(xvalid\_w2v)

prediction\_int = prediction[:,1] >= 0.3

prediction\_int = prediction\_int.astype(np.int) f1\_score(yvalid, prediction\_int)

0.614

Public Leaderboard F1 Score: 0.654

**Doc2Vec Features**

svc = svm.SVC(kernel='linear', C=1, probability=True).fit(xtrain\_d2v, ytrain)

prediction = svc.predict\_proba(xvalid\_d2v)

prediction\_int = prediction[:,1] >= 0.3

prediction\_int = prediction\_int.astype(np.int)

f1\_score(yvalid, prediction\_int)

0.203

Public Leaderboard F1 Score: 0.214

## RandomForest

Random Forest is a versatile machine learning algorithm capable of performing both regression and classification tasks. It is a kind of ensemble learning method, where a few weak models combine to form a powerful model. In Random Forest, we grow multiple trees as opposed to a decision single tree. To classify a new object based on attributes, each tree gives a classification and we say the tree “votes” for that class. The forest chooses the classification having the most votes (over all the trees in the forest).

It works in the following manner. Each tree is planted & grown as follows:

1. Assume number of cases in the training set is N. Then, sample of these N cases is taken at random but with replacement. This sample will be the training set for growing the tree.
2. If there are M input variables, a number m (m<M) is specified such that at each node, m variables are selected at random out of the M. The best split on these m variables is used to split the node. The value of m is held constant while we grow the forest.
3. Each tree is grown to the largest extent possible and there is no pruning.
4. Predict new data by aggregating the predictions of the ntree trees (i.e., majority votes for classification, average for regression).



from sklearn.ensemble import RandomForestClassifier

**Bag-of-Words Features**

First we will train our RandomForest model on the Bag-of-Words features and check its performance on both validation set and public leaderboard.

rf = RandomForestClassifier(n\_estimators=400, random\_state=11).fit(xtrain\_bow, ytrain)

prediction = rf.predict(xvalid\_bow)

# validation score f1\_score(yvalid, prediction)

0.553

Let’s make predictions for the test dataset and create another submission file.

test\_pred = rf.predict(test\_bow) test['label'] = test\_pred submission = test[['id','label']] submission.to\_csv('sub\_rf\_bow.csv', index=False)

Public Leaderboard F1 Score: 0.598

**TF-IDF Features**

rf = RandomForestClassifier(n\_estimators=400, random\_state=11).fit(xtrain\_tfidf, ytrain)

prediction = rf.predict(xvalid\_tfidf) f1\_score(yvalid, prediction)

0.562

Public Leaderboard F1 Score: 0.589

**Word2Vec Features**

rf = RandomForestClassifier(n\_estimators=400, random\_state=11).fit(xtrain\_w2v, ytrain)

prediction = rf.predict(xvalid\_w2v) f1\_score(yvalid, prediction)

0.507

Public Leaderboard F1 Score: 0.549

**Doc2Vec Features**

rf = RandomForestClassifier(n\_estimators=400, random\_state=11).fit(xtrain\_d2v, ytrain)

prediction = rf.predict(xvalid\_d2v) f1\_score(yvalid, prediction)

0.056

Public Leaderboard F1 Score: 0.07

## XGBoost

Extreme Gradient Boosting (xgboost) is an advanced implementation of gradient boosting algorithm. It has both linear model solver and tree learning algorithms. Its ability to do parallel computation on a single machine makes it extremely fast. It also has additional features for doing cross validation and finding important variables. There are many parameters which need to be controlled to optimize the model.

Some key benefits of XGBoost are:

1. **Regularization** - helps in reducing overfitting
2. **Parallel Processing** - XGBoost implements parallel processing and is blazingly faster as compared to GBM.
3. **Handling Missing Values** - It has an in-built routine to handle missing values.
4. **Built-in Cross-Validation** - allows user to run a cross-validation at each iteration of the boosting process

Check out this wonderful [guide](https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/) on XGBoost parameter tuning.

from xgboost import XGBClassifier

**Bag-of-Words Features**

xgb\_model = XGBClassifier(max\_depth=6, n\_estimators=1000).fit(xtrain\_bow, ytrain) prediction = xgb\_model.predict(xvalid\_bow) f1\_score(yvalid, prediction)

0.513

test\_pred = xgb\_model.predict(test\_bow) test['label'] = test\_pred submission = test[['id','label']] submission.to\_csv('sub\_xgb\_bow.csv', index=False)

Public Leaderboard F1 Score: 0.554

**TF-IDF Features**

xgb = XGBClassifier(max\_depth=6, n\_estimators=1000).fit(xtrain\_tfidf, ytrain)

prediction = xgb.predict(xvalid\_tfidf) f1\_score(yvalid, prediction)

Public Leaderboard F1 Score: 0.554

**Word2Vec Features**

xgb = XGBClassifier(max\_depth=6, n\_estimators=1000, nthread= 3).fit(xtrain\_w2v, ytrain)

prediction = xgb.predict(xvalid\_w2v) f1\_score(yvalid, prediction)

0.652

Public Leaderboard F1 Score: 0.698

XGBoost model on word2vec features has outperformed all the previuos models in this course.

**Doc2Vec Features**

xgb = XGBClassifier(max\_depth=6, n\_estimators=1000, nthread= 3).fit(xtrain\_d2v, ytrain)

prediction = xgb.predict(xvalid\_d2v) f1\_score(yvalid, prediction)

0.345

Public Leaderboard F1 Score: 0.374

## FineTuning XGBoost + Word2Vec

XGBoost with Word2Vec model has given us the best performance so far. Let’s try to tune it further to extract as much from it as we can. XGBoost has quite a many tuning parameters and sometimes it becomes tricky to properly tune them. This is what we are going to do in the following steps. You can refer this [guide](https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/) to learn more about parameter tuning in XGBoost.



import xgboost as xgb

Here we will use DMatrices. A DMatrix can contain both the features and the target.

dtrain = xgb.DMatrix(xtrain\_w2v, label=ytrain)

dvalid = xgb.DMatrix(xvalid\_w2v, label=yvalid)

dtest = xgb.DMatrix(test\_w2v)

# Parameters that we are going to tune

params = {

    'objective':'binary:logistic',

    'max\_depth':6,

    'min\_child\_weight': 1,

    'eta':.3,

    'subsample': 1,

    'colsample\_bytree': 1

 }

We will prepare a custom evaluation metric to calculate F1 score.

def custom\_eval(preds, dtrain):

    labels = dtrain.get\_label().astype(np.int)

    preds = (preds >= 0.3).astype(np.int)

    return [('f1\_score', f1\_score(labels, preds))]

**General Approach for Parameter Tuning**

We will follow the steps below to tune the parameters.

1. Choose a relatively high learning rate. Usually a learning rate of 0.3 is used at this stage.
2. Tune tree-specific parameters such as max\_depth, min\_child\_weight, subsample, colsample\_bytree keeping the learning rate fixed.
3. Tune the learning rate.
4. Finally tune gamma to avoid overfitting.

Tuning max\_depth and min\_child\_weight

gridsearch\_params = [

    (max\_depth, min\_child\_weight)

    for max\_depth in range(6,10)

     for min\_child\_weight in range(5,8)

 ]

max\_f1 = 0. # initializing with 0

best\_params = None

for max\_depth, min\_child\_weight in gridsearch\_params:

    print("CV with max\_depth={}, min\_child\_weight={}".format(

                             max\_depth,

                             min\_child\_weight))

     # Update our parameters

    params['max\_depth'] = max\_depth

    params['min\_child\_weight'] = min\_child\_weight

     # Cross-validation

    cv\_results = xgb.cv(        params,

        dtrain,        feval= custom\_eval,

        num\_boost\_round=200,

        maximize=True,

        seed=16,

        nfold=5,

        early\_stopping\_rounds=10

    )

# Finding best F1 Score

mean\_f1 = cv\_results['test-f1\_score-mean'].max()

boost\_rounds = cv\_results['test-f1\_score-mean'].argmax()

print("\tF1 Score {} for {} rounds".format(mean\_f1, boost\_rounds))

if mean\_f1 > max\_f1:

        max\_f1 = mean\_f1

        best\_params = (max\_depth,min\_child\_weight)

print("Best params: {}, {}, F1 Score: {}".format(best\_params[0], best\_params[1], max\_f1))

CV with max\_depth=6, min\_child\_weight=5

  F1 Score 0.6751088000000001 for 64 rounds

CV with max\_depth=6, min\_child\_weight=6    F1 Score 0.6703884 for 55 rounds

CV with max\_depth=6, min\_child\_weight=7    F1 Score 0.6761038 for 57 rounds

CV with max\_depth=7, min\_child\_weight=5    F1 Score 0.6784994 for 51 rounds

CV with max\_depth=7, min\_child\_weight=6    F1 Score 0.6808281999999999 for 47 rounds

CV with max\_depth=7, min\_child\_weight=7    F1 Score 0.6781346000000001 for 115 rounds

CV with max\_depth=8, min\_child\_weight=5    F1 Score 0.6634426 for 36 rounds

CV with max\_depth=8, min\_child\_weight=6    F1 Score 0.6822174000000001 for 71 rounds

CV with max\_depth=8, min\_child\_weight=7    F1 Score 0.6618758 for 32 rounds

CV with max\_depth=9, min\_child\_weight=5    F1 Score 0.6580265999999999 for 32 rounds

CV with max\_depth=9, min\_child\_weight=6    F1 Score 0.673238 for 43 rounds

CV with max\_depth=9, min\_child\_weight=7    F1 Score 0.6738529999999999 for 54 rounds

Best params: 8, 6, F1 Score: 0.6822174000000001

Updating max\_depth and min\_child\_weight parameters.

params['max\_depth'] = 8

params['min\_child\_weight'] = 6

Tuning subsample and colsample

gridsearch\_params = [

    (subsample, colsample)

    for subsample in [i/10. for i in range(5,10)]

    for colsample in [i/10. for i in range(5,10)] ]

max\_f1 = 0.

best\_params = None

for subsample, colsample in gridsearch\_params:

    print("CV with subsample={}, colsample={}".format(

                             subsample,

                             colsample))

     # Update our parameters

    params['colsample'] = colsample

    params['subsample'] = subsample

    cv\_results = xgb.cv(

        params,

        dtrain,

        feval= custom\_eval,

        num\_boost\_round=200,

        maximize=True,

        seed=16,

        nfold=5,

        early\_stopping\_rounds=10

    )

     # Finding best F1 Score

    mean\_f1 = cv\_results['test-f1\_score-mean'].max()

    boost\_rounds = cv\_results['test-f1\_score-mean'].argmax()

    print("\tF1 Score {} for {} rounds".format(mean\_f1, boost\_rounds))

    if mean\_f1 > max\_f1:

        max\_f1 = mean\_f1

        best\_params = (subsample, colsample)

print("Best params: {}, {}, F1 Score: {}".format(best\_params[0], best\_params[1], max\_f1))

CV with subsample=0.5, colsample=0.5 F1 Score 0.6651868000000001 for 78 rounds

CV with subsample=0.5, colsample=0.6    F1 Score 0.6651868000000001 for 78 rounds

CV with subsample=0.5, colsample=0.7    F1 Score 0.6651868000000001 for 78 rounds

CV with subsample=0.5, colsample=0.8    F1 Score 0.6651868000000001 for 78 rounds

CV with subsample=0.5, colsample=0.9    F1 Score 0.6651868000000001 for 78 rounds

CV with subsample=0.6, colsample=0.5    F1 Score 0.6776694000000001 for 70 rounds

CV with subsample=0.6, colsample=0.6    F1 Score 0.6776694000000001 for 70 rounds

CV with subsample=0.6, colsample=0.7    F1 Score 0.6776694000000001 for 70 rounds

CV with subsample=0.6, colsample=0.8    F1 Score 0.6776694000000001 for 70 rounds

CV with subsample=0.6, colsample=0.9    F1 Score 0.6776694000000001 for 70 rounds

CV with subsample=0.7, colsample=0.5    F1 Score 0.673453 for 58 rounds

CV with subsample=0.7, colsample=0.6    F1 Score 0.673453 for 58 rounds

CV with subsample=0.7, colsample=0.7    F1 Score 0.673453 for 58 rounds

CV with subsample=0.7, colsample=0.8    F1 Score 0.673453 for 58 rounds

CV with subsample=0.7, colsample=0.9    F1 Score 0.673453 for 58 rounds

CV with subsample=0.8, colsample=0.5    F1 Score 0.6776856 for 56 rounds

CV with subsample=0.8, colsample=0.6    F1 Score 0.6776856 for 56 rounds

CV with subsample=0.8, colsample=0.7    F1 Score 0.6776856 for 56 rounds

CV with subsample=0.8, colsample=0.8    F1 Score 0.6776856 for 56 rounds

CV with subsample=0.8, colsample=0.9    F1 Score 0.6776856 for 56 rounds

CV with subsample=0.9, colsample=0.5    F1 Score 0.6830936000000001 for 45 rounds

CV with subsample=0.9, colsample=0.6    F1 Score 0.6830936000000001 for 45 rounds

CV with subsample=0.9, colsample=0.7    F1 Score 0.6830936000000001 for 45 rounds

CV with subsample=0.9, colsample=0.8    F1 Score 0.6830936000000001 for 45 rounds

CV with subsample=0.9, colsample=0.9    F1 Score 0.6830936000000001 for 45 rounds Best

params: 0.9, 0.5, F1 Score: 0.6830936000000001

Updating subsample and colsample\_bytree

params['subsample'] = .9

params['colsample\_bytree'] = .5

Now let’s tune the learning rate.

max\_f1 = 0.

best\_params = None

for eta in [.3, .2, .1, .05, .01, .005]:

    print("CV with eta={}".format(eta))

     # Update ETA

    params['eta'] = eta

     # Run CV

    cv\_results = xgb.cv(

        params,

        dtrain,

        feval= custom\_eval,

        num\_boost\_round=1000,

        maximize=True,

        seed=16,

        nfold=5,

        early\_stopping\_rounds=20

    )

     # Finding best F1 Score

    mean\_f1 = cv\_results['test-f1\_score-mean'].max()

    boost\_rounds = cv\_results['test-f1\_score-mean'].argmax()

    print("\tF1 Score {} for {} rounds".format(mean\_f1, boost\_rounds))

    if mean\_f1 > max\_f1:

        max\_f1 = mean\_f1

        best\_params = eta

print("Best params: {}, F1 Score: {}".format(best\_params, max\_f1))

CV with eta=0.3

F1 Score 0.6849056000000001 for 84 rounds

CV with eta=0.2    F1 Score 0.684616 for 93 rounds

CV with eta=0.1    F1 Score 0.6864700000000001 for 211 rounds

CV with eta=0.05    F1 Score 0.6846697999999999 for 200 rounds

CV with eta=0.01    F1 Score 0.1302024 for 0 rounds

CV with eta=0.005    F1 Score 0.1302024 for 0 rounds

Best params: 0.1, F1 Score: 0.6864700000000001

params['eta'] = .1

Let’s have a look at the final list of tuned parameters.

params

{'colsample': 0.9,

 'colsample\_bytree': 0.5, 'eta': 0.1,

 'max\_depth': 8, 'min\_child\_weight': 6,

 'objective': 'binary:logistic',

 'subsample': 0.9}

Finally we can now use these tuned parameters in our xgboost model. We have used early stopping of 10 which means if the model’s performance doesn’t improve under 10 rounds, then the model training will be stopped.

xgb\_model = xgb.train(

    params,

    dtrain,

    feval= custom\_eval,

    num\_boost\_round= 1000,

    maximize=True,

    evals=[(dvalid, "Validation")],

    early\_stopping\_rounds=10

 )

[0] Validation-error:0.065909 Validation-f1\_score:0.133165 Multiple eval metrics have been passed: 'Validation-f1\_score' will be used for early stopping.

Will train until Validation-f1\_score hasn't improved in 10 rounds.

[1] Validation-error:0.058922   Validation-f1\_score:0.133165

[2] Validation-error:0.056523   Validation-f1\_score:0.133165

[3] Validation-error:0.055793   Validation-f1\_score:0.133165

[4] Validation-error:0.054229   Validation-f1\_score:0.133165

[5] Validation-error:0.054437   Validation-f1\_score:0.371141

[6] Validation-error:0.053916   Validation-f1\_score:0.45042

[7] Validation-error:0.053082   Validation-f1\_score:0.5158

[8] Validation-error:0.053082   Validation-f1\_score:0.565517

[9] Validation-error:0.052873   Validation-f1\_score:0.577889

[10]    Validation-error:0.052039   Validation-f1\_score:0.592938

[11]    Validation-error:0.05183    Validation-f1\_score:0.601116

[12]    Validation-error:0.051309   Validation-f1\_score:0.603037

[13]    Validation-error:0.052039   Validation-f1\_score:0.607988

[14]    Validation-error:0.051935   Validation-f1\_score:0.597892

[15]    Validation-error:0.051726   Validation-f1\_score:0.601688

[16]    Validation-error:0.050162   Validation-f1\_score:0.602484

[17]    Validation-error:0.050057   Validation-f1\_score:0.600624

[18]    Validation-error:0.049953   Validation-f1\_score:0.616352

[19]    Validation-error:0.050266   Validation-f1\_score:0.611465

[20]    Validation-error:0.049849   Validation-f1\_score:0.617343

[21]    Validation-error:0.049953   Validation-f1\_score:0.613909

[22]    Validation-error:0.049223   Validation-f1\_score:0.608347

[23]    Validation-error:0.049327   Validation-f1\_score:0.611557

[24]    Validation-error:0.04964    Validation-f1\_score:0.613893

[25]    Validation-error:0.049327   Validation-f1\_score:0.61576

[26]    Validation-error:0.049223   Validation-f1\_score:0.61251

[27]    Validation-error:0.048806   Validation-f1\_score:0.616626

[28]    Validation-error:0.048493   Validation-f1\_score:0.618856

[29]    Validation-error:0.048284   Validation-f1\_score:0.625202

[30]    Validation-error:0.048076   Validation-f1\_score:0.620243

[31]    Validation-error:0.048076   Validation-f1\_score:0.622871

[32]    Validation-error:0.047972   Validation-f1\_score:0.624493

[33]    Validation-error:0.048076   Validation-f1\_score:0.628757

[34]    Validation-error:0.047867   Validation-f1\_score:0.626526

[35]    Validation-error:0.047972   Validation-f1\_score:0.626623

[36]    Validation-error:0.048076   Validation-f1\_score:0.626016

[37]    Validation-error:0.048702   Validation-f1\_score:0.624898

[38]    Validation-error:0.048076   Validation-f1\_score:0.627642

[39]    Validation-error:0.047763   Validation-f1\_score:0.629268

[40]    Validation-error:0.047659   Validation-f1\_score:0.632006

[41]    Validation-error:0.04745    Validation-f1\_score:0.62987

[42]    Validation-error:0.04672    Validation-f1\_score:0.634225

[43]    Validation-error:0.046824   Validation-f1\_score:0.630894

[44]    Validation-error:0.047033   Validation-f1\_score:0.62996

[45]    Validation-error:0.046824   Validation-f1\_score:0.631408

[46]    Validation-error:0.046512   Validation-f1\_score:0.633631

[47]    Validation-error:0.046407   Validation-f1\_score:0.634818

[48]    Validation-error:0.04599    Validation-f1\_score:0.633117

[49]    Validation-error:0.046199   Validation-f1\_score:0.635332

[50]    Validation-error:0.045782   Validation-f1\_score:0.640259

[51]    Validation-error:0.045886   Validation-f1\_score:0.637751

[52]    Validation-error:0.046303   Validation-f1\_score:0.636071

[53]    Validation-error:0.046199   Validation-f1\_score:0.635559

[54]    Validation-error:0.046094   Validation-f1\_score:0.635634

[55]    Validation-error:0.045782   Validation-f1\_score:0.638911

[56]    Validation-error:0.04526    Validation-f1\_score:0.638911

[57]    Validation-error:0.045364   Validation-f1\_score:0.641026

[58]    Validation-error:0.045052   Validation-f1\_score:0.6416

[59]    Validation-error:0.045886   Validation-f1\_score:0.644338

[60]    Validation-error:0.045052   Validation-f1\_score:0.645883

[61]    Validation-error:0.044947   Validation-f1\_score:0.646965

[62]    Validation-error:0.044947   Validation-f1\_score:0.64687

[63]    Validation-error:0.045573   Validation-f1\_score:0.647528

[64]    Validation-error:0.045364   Validation-f1\_score:0.644177

[65]    Validation-error:0.045364   Validation-f1\_score:0.646918

[66]    Validation-error:0.045364   Validation-f1\_score:0.646302

[67]    Validation-error:0.045156   Validation-f1\_score:0.650602

[68]    Validation-error:0.045052   Validation-f1\_score:0.654429

[69]    Validation-error:0.04526    Validation-f1\_score:0.65655

[70]    Validation-error:0.045052   Validation-f1\_score:0.656

[71]    Validation-error:0.044947   Validation-f1\_score:0.653815

[72]    Validation-error:0.045052   Validation-f1\_score:0.651685

[73]    Validation-error:0.04526    Validation-f1\_score:0.650641

[74]    Validation-error:0.044843   Validation-f1\_score:0.653846

[75]    Validation-error:0.045156   Validation-f1\_score:0.652209

[76]    Validation-error:0.045156   Validation-f1\_score:0.652209

[77]    Validation-error:0.044947   Validation-f1\_score:0.653784

[78]    Validation-error:0.045573   Validation-f1\_score:0.655367

[79]    Validation-error:0.045364   Validation-f1\_score:0.653226

Stopping. Best iteration:

[69]    Validation-error:0.04526    Validation-f1\_score:0.65655

Let’s prepare one final submission file.

test\_pred = xgb\_model.predict(dtest) test['label'] = (test\_pred >= 0.3).astype(np.int) submission = test[['id','label']]

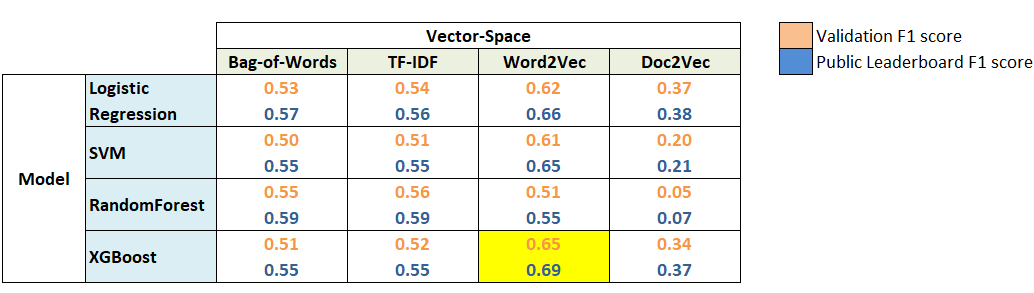
submission.to\_csv('sub\_xgb\_w2v\_finetuned.csv', index=False)

Public Leaderboard F1 Score: 0.703

Our tuning worked! This is our best score on the public leaderboard.

## Summary

Now it’s time to wrap-up things. Let’s quickly revisit what we have learned in this course, initially we cleaned our raw text data, then we learned about 4 different types of feature-set that we can extract from any text data, and finally we used these feature-sets to build models for sentiment analysis. Below is a summary table showing F1 scores for different models and feature-sets.



Word2Vec features turned out to be most useful. Whereas **XGBoost with Word2Vec features** was the best model for this problem. This clearly shows the power of word embeddings in dealing with NLP problems.

## WHAT ELSE CAN BE TRIED?

We have covered a lot in this Sentiment Analysis course, but still there is plenty of room for other things to try out. Given below is a list of tasks that you can try with this data.

1. We have built so many models in this course, we can definitely try model ensembling. A simple ensemble of all the submission files (maximum voting) yielded an F1 score of **0.55** on the public leaderboard.
2. Use Parts-of-Speech tagging to create new features.
3. Use stemming and/or lemmatization. It might help in getting rid of unnecessary words.
4. Use bi-grams or tri-grams (tokens of 2 or 3 words respectively) for Bag-of-Words and TF-IDF.
5. We can give pretrained word-embeddings models a try.