Reddit Comment Classification using Subreddits

Abstract

In this mini project, an attempt to classify Reddit comments into appropriate subreddit groups by implementing Bernoulli's Naive Bayes classifier and explore the performance of other widely used classifiers from *scikit-learn* library. The data sets have been subjected to selective data preprocessing to convert raw text to presentable input vectors. We were able to achieve 73.94% cross-validation accuracy using the Bernoulli's Naive Bayes algorithm. A 91.1% validation set accuracy and 93.09% accuracy over 30% of test dataset, is achieved over the stacked classifier (Linear SVC, Multinomial NB and Random Forest stacked over Logistic Regression).

1 Introduction

Text Classification is an important task in supervised Machine Learning. A fair number of classifiers have been suggested in the literature for performing text classification. Sentiment analysis, email spam/non-spam classification, tagging and categorization of articles and comments, etc. are wide applications of natural language processing and is widely in trend. This project has been divided into two independent phases. The first phase aimed at implementation of Bernoulli's Naive Bayes algorithm, from scratch, whereas the second phase targeted comparison of different classifiers for classification of Reddit comments into appropriate subreddit groups.

Bernoulli Naive Bayes (NB) classifier is a generative, probabilistic machine learning model, used for classification tasks. The primary assumption made in this classification approach is that the features/predictors are independent of each other. Thus the name, 'Naive'. The multivariate Bernoulli NB classifier, where l is the total number of class labels, uses the following equation,

$$P(x_i|y) = \prod_{i=1}^{m} P(x_i|y) \tag{1}$$

where $x_1, x_2, ...x_m$ are the m features of a sample x. Given an a new test sample, x_{new}, the probability of the sample belonging to each class k can be calculated using the following equation:

$$P(y = k|x) \propto \delta_k(x) = logP(y = k)P(x|y = k) \tag{2}$$

The final class label, y, can be predicted by finding the class with maximum probability $y = argmax_y P(y = k|x)$.

Laplace smoothing is applied to smooth the categorical data and overcome the problem of zero probability. To assess the strength of the model designed using this algorithm, K-Fold cross validation had been imposed on it. An accuracy of 73.94% was recorded for cross-validation using the Bernoulli NB implementation after limiting the maximum features to 1000.

To spice up the project, the second phase is hosted as a Kaggle[1] competition, where the freedom to use any classifier algorithm is provided. Using a stacked classifier model, we were able to achieve 91.1% validation set accuracy and a 93.09% accuracy over 30% of test dataset.

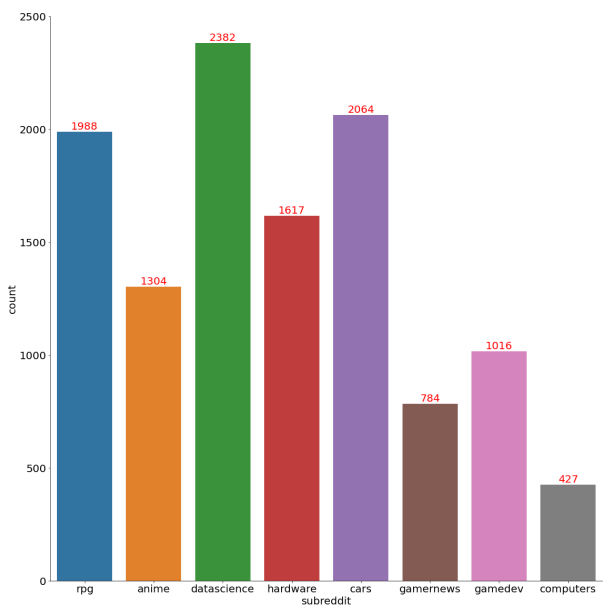
2 Dataset

The training dataset comprised of 11582 post and comments, posted over eight subreddit groups on Reddit, a popular social media forum. The posts are constituted of multiple sentences discussing the domain of a particular subreddit, and are labelled with the appropriate subreddit group to which they belong.

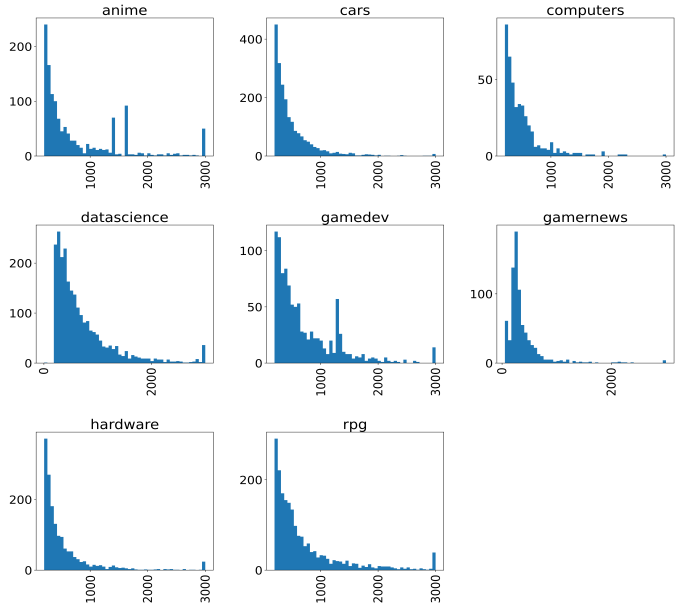
The distribution of the number of observations, per subreddit class, in the training dataset is found to be unbalanced, as shown in Fig. 1a. Due to the unbalanced nature of the dataset, misclassification of the posts or comments belonging to the subreddit class whose number of observations are far less in comparison to the average numbers is suspected. For example *computer* and

[1] https://www.kaggle.com/

gamernews having 427 and 784 sample observations respectively, have a higher chance of being misclassified in comparison to other subreddit classes.



(a) Subreddit class count in training dataset



(b) Length of posts in training dataset

On examining the number of words used in each post, as shown in Fig. 1b, it was observed that most of the posts for each class were between 0-1000 words so the length of the posts cannot be used as a feature for classification.

On analysis of the 'body' of the posts, some posts were found to comprise entirely of hyperlinks or had an *'#NAME?'* error. Since the number of such posts were relatively very low to actually affect performance, we ignored the "noise" and proceeded with data preprocessing.

2.1 Data Preprocessing

For this text classification task, the data preprocessing can be broadly divided into following steps (in order of application):

2.1.1 Text Simplification

The begining of the data preprocessing task starts by removal of stop-words and punctuation marks, tokenization and lemmatization. For the Kaggle competition, including the aforementioned steps, we also remove numbers, accents within the word, choose features based on part of sentence (POS) and limit the number of features wherever necessary. This was achieved using two libraries (applied separately): *nltk*[2] and *spacy*[3]. Preprocessing using both libraries gave similar results hence, for ease of reporting, the results in this report are derived by using *nltk* library.

2.1.2 Vectorization

Vectorization implies essentially creating a bag of words model where a text is represented by a vector of it's constituent words and the number of instances the words occur. This is done using the *CountVectorizer()* function in the sklearn library. The following hyperparameters of the vectorizer were tuned while optimising different ML models:

- max_df: Ignore terms that have a document frequency higher than the given threshold
- min_df: Ignore terms that have a document frequency lower than the given threshold.
- binary: If True, all non zero counts are set to 1.(Set true for Bernoulli Naive Bayes).
- ngram_range: The range of n-values for different word n-grams to be extracted
- strip_accents: Remove accents and perform other character normalization

[2]https://www.nltk.org/
[3]https://spacy.io/

It is expected that the selection of suitable features will be taken care of by selecting the appropriate max_df and min_df. Hence, an explicit constraint has not been imposed by using the *max_features* parameter.

2.1.3 Term Frequency -Inverse Document Frequency

Term Frequency - Inverse Document Frequency (TF-IDF) transformation was used thereafter to take into consideration the importance of a particular word within the document in the corpus. The only parameter tuned was the *norm* which can either be L1 or L2 norm.

2.1.4 Normalisation

The final preprocessing step consisted of normalising the features to unit norm. For this, L2 norm was used.

3 Proposed Approach

After building the Bernoulli NB model from scratch, there was an intuitive idea of the overall performance of a Naive Bayes Classifier. From existing examples of text classification problems, it is expected that Naive Bayes algorithm will work well on the dataset, especially with the bag of words model. It is expect the Linear Support Vector Classifier (SVC) to work well with the problem based on it's ability to create many hyper-planes to classify the data.

However, to confirm the hypothesis, a quick quick validation performance test on nine models, *K Nearest Neighbors, KNN with bagging, Decision Tree, Random Forest, AdaBoost Classifier, Multinomial NB, Bernoulli NB, Linear SVC and SVC with bagging*, was performed and recorded their accuracy along with other performance metrics. All models used in Table 1 are from the *sklearn* library

Table 1: Performance of different models on validation set (20% of training data)

	Model	**Accuracy**
1	Linear SVC	0.88908
2	Multinomial NB	0.87440
3	Bernoulli NB	0.85973
4	SVC with bagging	0.85267
5	Random Forests	0.80535
6	KNN	0.77168
7	Decision Tree	0.69054
8	Adaboost	0.67414
9	KNN with bagging	0.34613

After acquiring an initial idea about the performance of these models (refer Table 1 for performance report), the next stage was to proceed with optimisation of the performance of the promising and complementary models by fine tuning the hyper-parameters. This was achieved by performing a grid search on the Multinomial NB, Linear SVC and Random Forest Classifier algorithms. Besides the hyper-parameters of the models itself, we also tuned the word vectorizer and TF-IDF transformer used in preprocessing with the 3 models. Cross validation was used to measure performance of models during grid search.

3.1 Naive Bayes Classifier

It is a classification technique based on Bayes' Theorem with an assumption of independence among predictors. As introduced in Section 1, the first phase of the project records the implementation and application of Bernoulli NB. Another variation of NB, Multinomial NB classifier was explored in phase two of the project. While the base of both Bernoulli NB and Multinomial NB remains same, the only difference is how the input vector space is represented. For Bernoulli NB, only the *presence* or *absence* of a word is taken into account, Multinomial NB keeps a count of the frequency of each word in the text as well. In all models the only parameter used for tuning was ***alpha*** which is the Laplace smoothing parameter.

3.2 Support Vector Classifier

SVC's ability to learn linear threshold functions and the ease with which it can handle high dimensional input vector space appealed the most to us. Additionally, the security of the margin being influenced, only by the most relevant features strengthen the argument. The hyperparameters used for the model were ***C:*** The strength of the regularization is inversely proportional to C (must be positive) and ***tol:*** Tolerance for stopping criterion.

3.3 Ensemble Methods

Ensemble methods is a class of Machine Learning algorithms that combines several base models in order to produce one optimal predictive model. Out of the many ensemble methods endorsed, it is found that Random Forest Classifier and Stacking Classifier to best suit our purpose. Random Forests, a mix of decision trees and Bagging, works with the simple fundamental concept that a large number of relatively uncorrelated models (trees) operating as a committee will outperform any of the individual constituent models. the only parameter used for tuning was ***max_depth*** which is the maximum depth of the tree. Stacking Classifiers use the predictions made by N complementary base estimators and feeds these predictions over a simple classifier to provide final predictions with the assumption that the strengths of each base estimators can be put together to make final classification.

After fine-tuning of the hyper-parameters, the stacking ensemble method with a meta classifier over the three models is explored. The base estimators used for this stacking classifier were Multinomial NB, Linear SVC and Random Forest, which were stacked over the Logistic Regression Classifier with Stochastic Gradient Descent used as meta classifier. It is important here to note that as stated previously we selected these three classifiers for tuning in the first place because they tend to complement each others' performance in our preliminary testing.

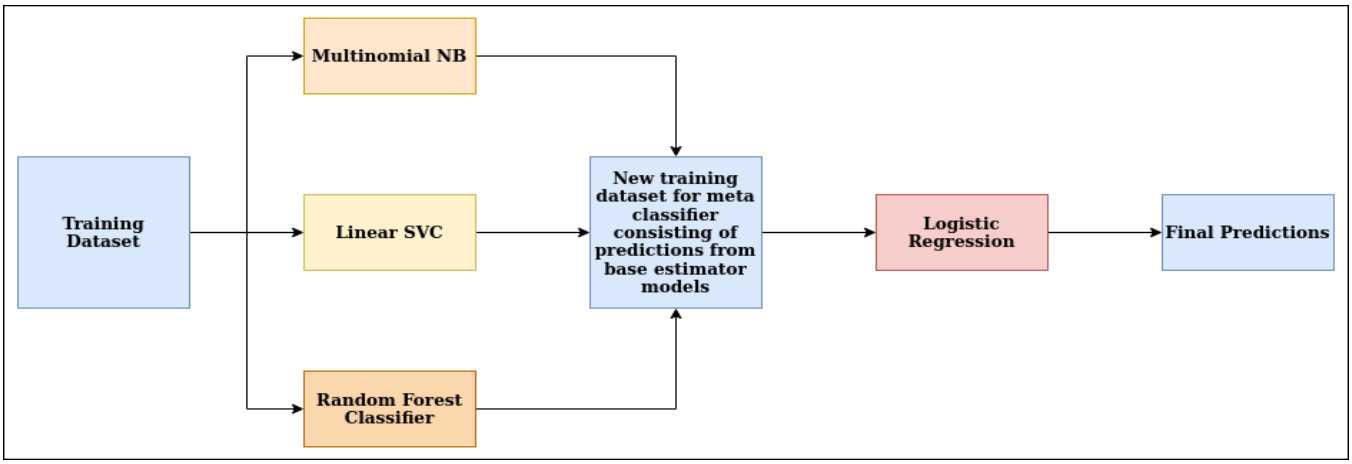


Figure 2: Stacking Classifier using Multinomial NB, Linear SVC and Random Forest over Logistic Regression

4 Results

4.1 Bernoulli Naive Bayes Classifier

Using a Bernoulli NB Classifier model made from scratch, see Appendix A, and using 1000 max features, a mean accuracy of **0.7394** on running 5 fold cross validation is seen. Using more features runs into a bottleneck and runtime takes too long to be viable. The reasons for this will be discussed in section 5.

4.2 Training Optimal Classifier Models

From table 1, we can clearly discern that Multinomial NB and Linear SVC outperform other algorithms on our dataset. Linear SVC leads with an accuracy of 88.90%, closely followed by Multinomial NB giving 87.44% accuracy. Both Bernoulli NB and SVC with Bagging give results in a similar range followed by Random Forests with 80.53% accuracy. Therefore, a Grid-Search Cross-Validation (CV) on these models is performed and the results are tabulated in table 2.

From the hyperparameters listed in table 2, we built our stacked classifier. Refer Table 3 for details.

5 Discussion and Conclusion

Since it is a NLP problem, the dataset included features whose size surpassed the number of given training examples. We noticed that reducing the feature size helped in our algorithms to achieve a better accuracy. For example, in Multinomial NB we reduce the feature set by 75% to achieve our best accuracy. However, Linear SVC was able to outperform these algorithms with 95% of features retained. This can be attributed to its ability to create multi-dimensional hyper-planes and classify. It should be noted here that this feature selection was achieved by tuning the max_df and min_df parameters of *CountVectorizer* API of *scikit-learn*.

Table 2: Optimal Model Hyperparamters and their accuracy on CV and Leaderboard

	Model	**Optimal Hyperparameters: CountVectorizer**	**Optimal Hyperparameters: TFidftransformer**	**Optimal Hyperparameters: Model**	**Accuracy on CV (5 folds)**	**Accuracy on Kaggle Leaderboard**
1	Linear SVC	max_df: 0.95 min_df: 1 strip_accents: 'unicode'	norm: 'L1'	C: 1 random_state: False tol: 1e-5	0.903	0.90909
2	MNB	binary: False max_df: 0.75 min_df: 1 ngram_range: (1, 1) strip_accents: False	norm: 'L1'	alpha: 0.01	0.889	0.90678
3	Random Forest	max_df: 0.5 strip_accents: 'unicode'	norm: 'L1'	max_depth: 100 (From values of (2,5,10,100)) random_state: False	0.795	Not Tested on leaderboard

Table 3: Features of Stacked Classifier

	Model	**Base Estimators**	**Meta Estimator**	**Accuracy on validation set (20% training data)**	**Accuracy on Kaggle Leaderboard**
1	Stacked Model 1	Linear SVC (tol=deafult), Multinomial NB, Random Forest	Logistic Regression (max_iter = 100)	0.911	0.93095
2	Stacked Model 2	Linear SVC, Multinomial NB	Logistic Regression (max_iter = 200)	0.907	Not tested on Leaderboard
3	Stacked Model 3	Linear SVC, Multinomial NB	SGD Classifier	0.894	Not tested on Leaderboard

Furthermore, while vectorizing the features names, it was observed that normalization makes a significant impact because it scales the data to same scale, which in turn, increases the performance of the classifier.

Another noteworthy observation was the time taken for training these classifiers. The Bernoulli NB model (made from scratch) ran into a bottleneck as we increased the number of features. The main issue was calculating all the values for θ_k^j and storing them in a dictionary. This step takes a long time on python.

A Appendix: Bernoulli Naive Bayes Code

```
# -*- coding: utf-8 -*-
"""Naive_Bayes_from_scratch.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1apQ-mAPAKyJ3DdhSOLOduQx8mIdEFYBC
"""

# Commented out IPython magic to ensure Python compatibility.
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
# %matplotlib inline

from google.colab import drive
drive.mount('/content/drive')

#Text preprocessing . Added as tokenizer in Count_Vectoriser()
def text_process(mess):
    """
    Takes in a string of text, then performs the following:
    1. Remove all punctuation
    2. Remove all stopwords
    3. Lemmatize words
    4. Returns the modified string .
    """
    # Check characters to see if they are in punctuation
    nopunc = [char for char in mess if char not in string.punctuation]

    # Join the characters again to form the string.
    nopunc = ''.join(nopunc)

    # Now just remove any stopwords
    return [wnl.lemmatize(word) for word in nopunc.split() if word.lower()
    not in stopwords.words('english')]

import string
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
nltk.download('stopwords')
nltk.download('wordnet')
wnl=WordNetLemmatizer()

df=pd.read_csv('/content/drive/My Drive/Colab Notebooks/Project 2/train.csv')

df_sm = df.sample(frac=1)#use fraction of data for debugging, (the data is huge)
x = df_sm['body']
y = df_sm['subreddit']
xtrain,xtest,ytrain,ytest =train_test_split(x,y, test_size=0.33,random_state=42)
```

```
#create dictionary with the data and binary word vectors
cv=CountVectorizer(tokenizer=text_process,binary=True,ngram_range=(1,1),
                   max_features=1000)
train_vectors=cv.fit_transform(xtrain)
test_vectors=cv.transform(xtest)
df_train=pd.DataFrame.sparse.from_spmatrix(train_vectors,index=ytrain)

df_train

#calculate theta_k for every class
theta_k= {k:(ytrain == k).sum()/float(ytrain.shape[0]) for k in ytrain.unique()}

theta_k_j = {k:

             {j: ( df_train.loc[k][j].sum() +1) / float( (ytrain ==k).sum() + 2)
             for j in df_train.columns } #calculate theta_k_j for every k

             for k in ytrain.unique()}

#create dataframe for easier vector multplication
df_theta_kj=pd.DataFrame.from_dict(theta_k_j,orient='index')

dict_prob={k:(test_vectors @ np.log(df_theta_kj.loc[k]) +
              (1-test_vectors.toarray()) @ np.log(1-df_theta_kj.loc[k]) +
              theta_k[k] ) for k in ytrain.unique()}
#create dataframe with class probability for every sample
class_prob = pd.DataFrame.from_dict(dict_prob,orient='columns')

#extract the class with max probability for each sample
predictions = class_prob.idxmax(axis=1)
from sklearn.metrics import classification_report,confusion_matrix
from sklearn.metrics import accuracy_score
print( classification_report(ytest,predictions))
print('\n',accuracy_score(ytest,predictions))

"""# **KFold cross validation**"""

from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score
kf =KFold(n_splits=5,shuffle=True)
accuracy_kfold = []
for train, test in kf.split(x):

  #create dictionary with the data and binary word vectors
  cv=CountVectorizer(binary=True,ngram_range=(1,1),
                     max_features=1000)
  train_vectors=cv.fit_transform(x[train])
  test_vectors=cv.transform(x[test])
  df_train=pd.DataFrame.sparse.from_spmatrix(train_vectors,index=y[train])

  #calculate theta_k for every class
  theta_k= {k:(y[train] == k).sum()/float(y[train].shape[0]) for
            k in y[train].unique()}

  theta_k_j = {k:
```

```
            {j: ( df_train.loc[k][j].sum() +1) /float( (y[train] ==k).sum()+2)
            for j in df_train.columns } #calculate theta_k_j for every k

            for k in y[train].unique()}

  #create dataframe for easier vector multplication
  df_theta_kj=pd.DataFrame.from_dict(theta_k_j,orient='index')

  dict_prob={k:(test_vectors @ np.log(df_theta_kj.loc[k]) +
              (1-test_vectors.toarray()) @ np.log(1-df_theta_kj.loc[k]) +
              theta_k[k] ) for k in ytrain.unique()}
  #create dataframe with class probability for every sample
  class_prob = pd.DataFrame.from_dict(dict_prob,orient='columns')

  #extract the class with max probability for each sample
  predictions = class_prob.idxmax(axis=1)
  accuracy_kfold.append(accuracy_score(y[test],predictions))

from statistics import mean
print(mean(accuracy_kfold))



#test_data = pd.read_csv('/content/drive/My Drive/Colab Notebooks/Project 2/test.csv')
#test= test_data['body']
#corpus = pd.concat([x,test],axis=0)
```

B Appendix: Different Classifiers for Kaggle

```
# -*- coding: utf-8 -*-
"""Copy of 551_2_v5.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1gfp5ADnr1ZD2IjaUzsIjdfq7J7RwUcBY
"""

import pandas as pd
import numpy as np
from google.colab import files
import io
from sklearn.preprocessing import Normalizer
from sklearn.feature_extraction import text
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer, TfidfVectorizer
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.model_selection import GridSearchCV, cross_val_score, train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import MultinomialNB, GaussianNB, BernoulliNB
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier, StackingClassifier, AdaBoostClassif
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import Perceptron, LogisticRegression, SGDClassifier
from sklearn.svm import LinearSVC
from sklearn.metrics import classification_report,accuracy_score
from google.colab import files
!pip install nltk
```

```
import nltk
nltk.download('punkt')
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')
nltk.download('omw')
from nltk import word_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet

"""## **1.Data Preprocessing**"""

"""
1. Selectthe files train and test files to upload. (Modify the file names accordingly; train.csv and test.c
2. Importing stopwords from text.ENGLISH_STOP_WORDS and modifying the set to store in my_stop_words
3. Lemmatization method from class tutorial to take Adjectives, nouns, verbs and adverbs

"""
data_to_load = files.upload()
df_train = pd.read_csv(io.BytesIO(data_to_load['train.csv']))
df_test = pd.read_csv(io.BytesIO(data_to_load['test.csv']))
X_train = df_train['body']
y_train = df_train['subreddit']
X_test = df_test['body']

#############################################################################################

stop_words_set = text.ENGLISH_STOP_WORDS
my_stop_words = ['far','make','u']
for word in stop_words_set:
  my_stop_words.append(word)
def get_wordnet_pos(word):
    """Map POS tag to first character lemmatize() accepts"""
    tag = nltk.pos_tag([word])[0][1][0].upper()
    tag_dict = {"J": wordnet.ADJ,
                "N": wordnet.NOUN,
                "V": wordnet.VERB,
                "R": wordnet.ADV}
    return tag_dict.get(tag, wordnet.NOUN)
class New_LemmaTokenizer:
     def __init__(self):
       self.wnl = WordNetLemmatizer()
     def __call__(self, doc):
       return [self.wnl.lemmatize(t,pos =get_wordnet_pos(t)) for t in word_tokenize(doc) if t.isalpha()]

"""# **2. Models**

# **A. Comparson of all models**

We quickly run suitable classifiers to find their accuracies and other metrics
"""

"""
Using train_test_split to split traing data into two parts: 80% train and 20% test.
This method is preferred to Cross validation to save time.

"""
xtrain,xtest,ytrain,ytest = train_test_split(X_train,y_train, test_size=0.2,random_state=0)
```

```
classifiers = [
    ('KNN', KNeighborsClassifier()),
    ('KNN bagging', BaggingClassifier(base_estimator = KNeighborsClassifier(), max_samples=0.5, max_feature
    ('decision tree', DecisionTreeClassifier()),
    ('random forests', RandomForestClassifier()),
    ('Adaboost', AdaBoostClassifier(n_estimators=50)),
    ('MNB', MultinomialNB(alpha=0.01)),
    ('Bernoulli NB', BernoulliNB(alpha=0.01)),
    ('Linear SVC', LinearSVC(tol=1e-5)),
    ('SVC with bagging', BaggingClassifier(base_estimator = LinearSVC(), max_samples=0.5, max_features=0.5)
    ]
################################################################################
for model_name, model in classifiers:
  classifier = model
  print(model_name)
  pipeline_clf = Pipeline([
                           ('vect', CountVectorizer(tokenizer=New_LemmaTokenizer(),
                                                    stop_words = my_stop_words)),
                           ('tfidf', TfidfTransformer(norm = 'l1')),
                           ('norm', Normalizer()),
                           ('clf', classifier)

  ])
  pipeline_clf.fit(xtrain,ytrain)
  prediction_clf = pipeline_clf.predict(xtest)
  print(classification_report(ytest,prediction_clf))
  print('\n',accuracy_score(ytest,prediction_clf))
  print()
  print('.'*80,'\n')

"""# **B. Best hyperparamters: MNB**"""

"""
To find the best hyper paramters
"""

pipeline = Pipeline([
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('norm', Normalizer()),
    ('clf', MultinomialNB()),
])

parameters = {
    'vect__strip_accents' : ('unicode', False),
    'vect__max_df' : (0.5, 0.75,1.0),
    'vect__min_df':(1,2),
    'vect__binary':(True, False),
    'vect__ngram_range': ((1, 1), (1, 2)),
    'tfidf__norm': ('l1', 'l2'),
    'clf__alpha': (0.005, 0.01,0.1, 1),
}
########################################################################

print("Performing grid search")
print('.'*100)
```

```
grid_search = GridSearchCV(pipeline, parameters, cv =5)
grid_search.fit(X_train, y_train)
print("Best score: %0.3f" % grid_search.best_score_)
print("Best parameters set:")
best_parameters = grid_search.best_estimator_.get_params()
for param_name in sorted(parameters.keys()):
  print("\t%s: %r" % (param_name, best_parameters[param_name]))

"""
Checking without TFID transformer
"""

pipeline_MNB_2 = Pipeline([
    ('vect', CountVectorizer(tokenizer=New_LemmaTokenizer(),
                             stop_words=my_stop_words,)),

    ('norm', Normalizer()),
    ('clf', MultinomialNB(alpha = 0.01)),
])

parameters_MNB_2 = {
    'vect__max_df' : (0.5,0.75, 1)
}
##########################################################################
print("Performing grid search")
print('.'*100)
grid_search_MNB_2 = GridSearchCV(pipeline_MNB_2, parameters_MNB_2, cv=5)
grid_search_MNB_2.fit(X_train, y_train)
print("Best score: %0.3f" % grid_search_MNB_2.best_score_)
print("Best parameters set:")
best_parameters_MNB_2 = grid_search_MNB_2.best_estimator_.get_params()
for param_name in sorted(parameters_MNB_2.keys()):
  print("\t%s: %r" % (param_name, best_parameters_MNB_2[param_name]))

"""# **C. Best Hyperparamters: LinearSVC**"""

pipeline_SVC = Pipeline([
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('norm', Normalizer()),
    ('clf', LinearSVC()),
])

parameters_SVC = {
    'vect__strip_accents' : ('unicode', False),
    'vect__max_df' : (0.5,0.75,0.95,1.0),
    'vect__min_df':(1,2),
    'tfidf__norm': ('l1', 'l2'),
    'clf__tol' : (1e-5,1e-6),
    'clf__C': (0.1, 0.5, 1,2,5),
    'clf__random_state' :(False, 0)

}
##########################################################################
print("Performing grid search")
print('.'*100)
grid_search_SVC = GridSearchCV(pipeline_SVC, parameters_SVC, cv = 5)
```

```
grid_search_SVC.fit(X_train, y_train)
print("Best score: %0.3f" % grid_search_SVC.best_score_)
print("Best parameters set:")
best_parameters_SVC = grid_search_SVC.best_estimator_.get_params()
for param_name in sorted(parameters_SVC.keys()):
  print("\t%s: %r" % (param_name, best_parameters_SVC[param_name]))

"""# **D. Best Hyperparamters: Random Forest**"""

pipeline_rf = Pipeline([
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('norm', Normalizer()),
    ('clf', RandomForestClassifier()),
])

parameters_rf = {
    'vect__strip_accents' : ('unicode', False),
    'vect__max_df' : (0.5,0.75,0.95,1.0),
    'clf__max_depth' : (2,5,10,100,200),
    'clf__random_state' :(False, 0)

}
##########################################################################
print("Performing grid search")
print('.'*100)
grid_search_rf = GridSearchCV(pipeline_rf, parameters_rf, cv = 5)
grid_search_rf.fit(X_train, y_train)
print("Best score: %0.3f" % grid_search_rf.best_score_)
print("Best parameters set:")
best_parameters_rf = grid_search_rf.best_estimator_.get_params()
for param_name in sorted(parameters_rf.keys()):
  print("\t%s: %r" % (param_name, best_parameters_rf[param_name]))

"""# **E. Meta Classifiers**"""

"""
estimators: MNB, Linear SVC
Final classifer: Logistic Regression
"""
xtrain,xtest,ytrain,ytest = train_test_split(X_train,y_train, test_size=0.2,random_state=0)
pip_MNB = Pipeline([('vect_MNB',CountVectorizer(tokenizer=New_LemmaTokenizer(),
                                                stop_words=my_stop_words,
                                                max_df = 0.5)),
                    ('tfidf_MNB',TfidfTransformer(norm = 'l1')),
                    ('norm_MNB', Normalizer()),
                    ('clf_MNB', MultinomialNB(alpha = 0.01))
])
pip_SVC = Pipeline([('vect_SVC',CountVectorizer(strip_accents='unicode',
                                                tokenizer=New_LemmaTokenizer(),
                                                stop_words=my_stop_words,
                                                max_df = 0.95)),
                    ('tfidf_SVC',TfidfTransformer(norm = 'l1')),
                    ('norm_SVC', Normalizer()),
                    ('clf_SVC', LinearSVC())
])
```

```
estimators = [
              ('MNB', pip_MNB),
              ('Linear SVC', pip_SVC),
]
meta_clf = StackingClassifier(
                              estimators = estimators,
                              final_estimator=LogisticRegression(max_iter=200)
)
meta_clf_score = meta_clf.fit(xtrain, ytrain).score(xtest, ytest)
print(meta_clf_score)

"""
estimators: MNB, Linear SVC
Final Classifier: SGD
"""
xtrain,xtest,ytrain,ytest = train_test_split(X_train,y_train, test_size=0.2,random_state=0)
pip_MNB = Pipeline([('vect_MNB',CountVectorizer(tokenizer=New_LemmaTokenizer(),
                                                stop_words=my_stop_words,
                                                max_df = 0.5)),
                    ('tfidf_MNB',TfidfTransformer(norm = 'l1')),
                    ('norm_MNB', Normalizer()),
                    ('clf_MNB', MultinomialNB(alpha = 0.01))
])
pip_SVC = Pipeline([('vect_SVC',CountVectorizer(tokenizer=New_LemmaTokenizer(),
                                                stop_words=my_stop_words,
                                                max_df = 0.95)),
                    ('tfidf_SVC',TfidfTransformer(norm = 'l1')),
                    ('norm_SVC', Normalizer()),
                    ('clf_SVC', LinearSVC())
])

estimators = [
              ('MNB', pip_MNB),
              ('Linear SVC', pip_SVC),
]
meta_clf = StackingClassifier(
                              estimators = estimators,
                              final_estimator=SGDClassifier()
)
meta_clf_score = meta_clf.fit(xtrain, ytrain).score(xtest, ytest)
print(meta_clf_score)

"""
estimators: MNB, Linear SVC, Random Forest
Final classifer: Logistic Regression(max_iter = 200)
"""
xtrain,xtest,ytrain,ytest = train_test_split(X_train,y_train, test_size=0.2,random_state=0)
pip_MNB = Pipeline([('vect_MNB',CountVectorizer(tokenizer=New_LemmaTokenizer(),
                                                stop_words=my_stop_words,
                                                max_df = 0.75)),
                    ('tfidf_MNB',TfidfTransformer(norm = 'l1')),
                    ('norm_MNB', Normalizer()),
                    ('clf_MNB', MultinomialNB(alpha = 0.01))
])
pip_SVC = Pipeline([('vect_SVC',CountVectorizer(tokenizer=New_LemmaTokenizer(),
                                                stop_words=my_stop_words,
                                                max_df = 0.95)),
```

```
                    ('tfidf_SVC',TfidfTransformer(norm = 'l1')),
                    ('norm_SVC', Normalizer()),
                    ('clf_SVC', LinearSVC())
])
pip_rfC = Pipeline([('vect_rf',CountVectorizer(strip_accents='unicode',
                                               tokenizer=New_LemmaTokenizer(),
                                               stop_words=my_stop_words,
                                               max_df = 0.5)),
                    ('tfidf_rf',TfidfTransformer(norm = 'l1')),
                    ('norm_rf', Normalizer()),
                    ('clf_rf', RandomForestClassifier(max_depth=200))
])

estimators = [
              ('MNB', pip_MNB),
              ('Linear SVC', pip_SVC),
              ('Random Forest', pip_rfC)
]
meta_clf = StackingClassifier(
                             estimators = estimators,
                             final_estimator=LogisticRegression(max_iter=100)
)
meta_clf_score = meta_clf.fit(xtrain, ytrain).score(xtest, ytest)
print(meta_clf_score)
final_fit = meta_clf.fit(X_train, y_train)
final_predict = meta_clf.predict(X_test)

"""# **3. Final Prediction**"""

meta_clf_predict = pd.DataFrame(final_predict)
meta_clf_predict.to_csv('meta_clf_predict_0.75_2.csv')
files.download('meta_clf_predict_0.75_2.csv')
```