In [81]:
```python
import numpy as np
import pandas as pd
import random
import re
import string
# from sklearn.datasets import fetch_20moviegroups
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import itertools
import warnings
warnings.filterwarnings('ignore')
random_seed = 3116
```

1 Naive Bayes

In this part of the assignment, you will be using the processed dataset from previous Lab Question 1. Using the BOW and Tf-Idf representation, implement a Naive-Bayes classifier for the data. Use Laplace smoothing for the implementation. Compare your implemenation with the sklearn implementation.

In [82]:
```python
#importing the training data
imdb_data=pd.read_csv('imdb_dataset.csv')
imdb_data = imdb_data.iloc[:1000, :] # Taking a subset of data due to high computational requirement of this question
```

In [83]:
```python
# Function to split the dataset into train, test and validation sets
def split_dataset(movie_vectors, targets, train_ratio, validation_ratio):
    combined = list(zip(movie_vectors, targets))
    random.shuffle(combined)
    train_rows = int(len(combined) * train_ratio)
    validation_rows = int(len(combined) * validation_ratio)
    X , y = list(zip(*combined))
    X, y = list(X), list(y)
    X_train, X_val, X_test = X[:train_rows], X[train_rows:train_rows+validation_rows], X[train_rows+validation_rows:]
    y_train, y_val, y_test = y[:train_rows], y[train_rows:train_rows+validation_rows], y[train_rows+validation_rows:]
    return X_train, X_val, X_test, y_train, y_val, y_test
```

In [84]:
```python
# Preprocessing textual data to remove punctuation, stop-words
# Function to Preprocess the data by removing Stopwords, punctuations and tokenizing the data
def preprocess(movie_string):
    # Extracting the English stopwords and converting it into a set
    english_stop_words = set(stopwords.words('english'))
    # Making the data into the lower case string and then tokenizing the data into word list
    movie_string = word_tokenize(movie_string.lower())
    # Removing stopwords and punctuations from the word list
    movie_string = [word for word in movie_string if word not in english_stop_words and word.isalpha()]
    # Returning the final processed data list
    return movie_string
```

In [85]:
```python
# Extracting movie and its Target label
movie_items = imdb_data['review']
movie_target = imdb_data['sentiment']
# Initializing an empty list to store processed movie items
processed_movie = []
# Applying Preprocessing step on all the movie items
# Iterating for each movie items
for n_item in movie_items:
    #Applying preprocessing on current movie item
    processed_movie.append(preprocess(n_item))
```

In [86]:
```python
# bag-of-words feature representation for each text sample
def update_word_freq(data, freq_dict): # Word freq Dictionary from the Provided data
    # Using list comprehension to update word freq in the dictionary
    freq_dict = {word: freq_dict.get(word, 0) + 1 for word in data}
    return freq_dict

# Create a Binary vector for a data based on Bag of Word Representation
def bag_of_words(data, freq_dict):
    #Initializing a vector with zeros having the length equal to total unique words in the corpus
    sent_vector = np.zeros(shape=(len(freq_dict.keys()),))
    for word in data:
        if word in freq_dict.keys():
            sent_vector[list(freq_dict.keys()).index(word)] = 1
    return sent_vector

features = 1500 # Number of features for processing
# Creating a dictionary to store all unique words and there count in the entire corpus
corpus_word_freq = {}
for doc in processed_movie:
    corpus_word_freq = update_word_freq(doc, corpus_word_freq)
corpus_word_freq = dict(sorted(corpus_word_freq.items(), key=lambda item: item[1], reverse=True))
corpus_word_freq = dict(itertools.islice(corpus_word_freq.items(), features))

movie_bog_vectors = []
for doc in processed_movie:
    # Creating a bag of word representation vector and appending it into the final list
    movie_bog_vectors.append(bag_of_words(doc, corpus_word_freq))
```

In [87]:
```python
# Bag of Word Representation
movie_bog_df = pd.DataFrame(movie_bog_vectors, columns=corpus_word_freq.keys())
movie_bog_df.head()
```

Out[87]:

	make	br	see	even	like	appear	film	go	avoid	minutes	...	noises	appropriate	masking	noise	old	israeli	russian	planes	used	us
0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	1.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
3	1.0	1.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	1.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0

5 rows × 219 columns

In [88]:
```
# Function to calculate the Term freq (TF) for a word in a data
def term_freq(data, word):
    return data[word]/sum(data.values())

# Function to calculate the Inverse data freq (IDF) for a word in the entire Corpus
def data_freq_inv(total_doc_freq, word, total_datas):
    return np.log(total_datas/total_doc_freq[word] + 1)

# Function to calculate the TF-IDF of a all the words individually in the entire corpus
def tf_idf(data, total_doc_freq, total_datas):
    data_vector = np.zeros(shape=(len(total_doc_freq.keys()),))
    for word in data.keys():
        if word in total_doc_freq.keys():
            tf_idf = term_freq(data, word) * data_freq_inv(total_doc_freq, word, total_datas)
            data_vector[list(total_doc_freq.keys()).index(word)] = tf_idf
    return data_vector
```

In [89]:
```
# Converting each data in the dataset into a TF-IDF Representation
movie_tfidf_vectors = []
#Iterating for each datas to generate word freq dictionary and adding tfidf vectors to movie_tfidf_vectors list
for doc in processed_movie:
    current_doc_dict = update_word_freq(doc, {})
    movie_tfidf_vectors.append(tf_idf(current_doc_dict, corpus_word_freq, len(processed_movie)))
```

In [90]:
```
# TF-IDF Representation
movie_tfidf = pd.DataFrame(movie_tfidf_vectors, columns=corpus_word_freq.keys())
movie_tfidf.head()
```

Out[90]:

	make	br	see	even	like	appear	film	go	avoid	minutes	...	noises	appropriate	masking	noise	old	israeli	russian	planes	used	us
0	0.000000	0.043374	0.000000	0.000000	0.000000	0.0	0.000000	0.046393	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
1	0.000000	0.079618	0.079618	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
2	0.000000	0.075482	0.075482	0.075482	0.000000	0.0	0.000000	0.080735	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.089724
3	0.106066	0.116243	0.116243	0.000000	0.124332	0.0	0.124332	0.000000	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
4	0.054673	0.059919	0.059919	0.000000	0.000000	0.0	0.064089	0.000000	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.071224

5 rows × 219 columns

Naive Bayes Classifier for Text Data

In [91]:
```python
class NaiveBayes:
    def __init__(self,feature_representation, dataset, target):
        self.feature_representation = feature_representation
        self.dataset = np.array(dataset)
        self.target = np.array(target)
        self.unique_target = list(set(target))
        self.accuracy = 0
        self.combined = np.concatenate((self.dataset, self.target.reshape(-1,1)), axis = 1)

    # Calculate the Probability for a data represented using Bag of Words
    def get_prob_bog(self, data, target):
        # Calculating the probability for a class itself
        prob_class = len(self.target[np.where(self.target == target)]) / len(self.target)
        # Initializing the prior probability for each Word in the data
        words_prior_prob = 1
        # Iterating over each word in the data
        for i in range(len(data) - 1): # If word exists in the data
            if data[i] == 1:
                # Calculating the Prior probability of the word given the class
                p_word_num = len(self.combined[np.where((self.combined[:,i] == 1) & (self.combined[:,-1] == target))])
                p_word_den = len(self.combined[np.where(self.combined[:,-1] == target)])
                words_prior_prob *= (p_word_num / p_word_den)
        return words_prior_prob * prob_class

    # Calculate the Probability for a data represented using TF-IDF
    def get_prob_tfidf(self, data, target):
        prob_class = len(self.target[np.where(self.target == target)]) / len(self.target)
        words_prior_prob = 1
        for i in range(len(data) - 1):
            if data[i] == 1:
                p_word_num = sum(self.combined[np.where((self.combined[:,i] == 1) & (self.combined[:,-1] == target))])
                p_word_den = sum(self.combined[np.where(self.combined[:,-1] == target)])
                words_prior_prob *= (p_word_num / p_word_den)
        return words_prior_prob * prob_class

    # Predictions on the dataset provided and calculate the Accuracy
    def predict(self):
        for index, doc in enumerate(self.combined):
            tar_prob = []
            # Iterating over all unique target values for calculating there probabilities
            for tar in self.unique_target:
                # calculating the probabiltiy
                if self.feature_representation == 'bog':
                    tar_prob.append(self.get_prob_bog(doc, tar))
                elif self.feature_representation == 'tfidf':
                    tar_prob.append(self.get_prob_tfidf(doc, tar))
            tar_prob = list(map(lambda x : x / (sum(tar_prob) + 1),tar_prob)) # Normalizing
            predicted_class = self.unique_target[tar_prob.index(max(tar_prob))]
            if predicted_class == self.target[index]:
                self.accuracy += 1
        self.accuracy /= len(self.combined)

    # Score
    def score(self):
        if self.feature_representation == 'bog':
            feature_rep = 'Bag of Words'
        else:
            feature_rep = 'TF-IDF'
        return str(feature_rep) + ' is ' + str(np.round(self.accuracy * 100, 2)) + '%'
```

Using Bag of Word Representation

In [92]:
```
# Splitting the Dataset with Bag of Word Representation into Train, Validation and Test sets
X_train, X_val, X_test, y_train, y_val, y_test = split_dataset(movie_bog_vectors, movie_target, 0.8, 0.1)

# Creating and Fitting the Naive Bayes model on the training, Validation and Test Sets
NaiveBayes_train = NaiveBayes('bog', X_train, y_train)
NaiveBayes_train.predict()
NaiveBayes_val = NaiveBayes('bog', X_val, y_val)
NaiveBayes_val.predict()
NaiveBayes_test = NaiveBayes('bog', X_test, y_test)
NaiveBayes_test.predict()

# Calculating and Displaying the Training, Validation and Test Accuracies
print('Training Accuracy: {}'.format(NaiveBayes_train.score()))
print('Validation Accuracy: {}'.format(NaiveBayes_val.score()))
print('Test Accuracy: {}'.format(NaiveBayes_test.score()))
```

```
Training Accuracy: Bag of Words is 51.25%
Validation Accuracy: Bag of Words is 57.0%
Test Accuracy: Bag of Words is 52.0%
```

Using TF-IDF Representation

In [93]:
```
X_train, X_val, X_test, y_train, y_val, y_test = split_dataset(movie_tfidf_vectors, movie_target, 0.8, 0.1)
# Creating and Fitting the Naive Bayes model on the training, Validation and Test Sets
NaiveBayes_train = NaiveBayes('tfidf', X_train, y_train)
NaiveBayes_train.predict()
NaiveBayes_val = NaiveBayes('tfidf', X_val, y_val)
NaiveBayes_val.predict()
NaiveBayes_test = NaiveBayes('tfidf', X_test, y_test)
NaiveBayes_test.predict()

# Calculating and Displaying the Training, Validation and Test Accuracies
print('Training Accuracy: {}'.format(NaiveBayes_train.score()))
print('Validation Accuracy: {}'.format(NaiveBayes_val.score()))
print('Test Accuracy: {}'.format(NaiveBayes_test.score()))
```

```
Training Accuracy: TF-IDF is 50.25%
Validation Accuracy: TF-IDF is 56.0%
Test Accuracy: TF-IDF is 55.0%
```

SVM Classifier via Scikit-Learn

Bag of Word Representation

In [94]:
```
hyperparameter_grid = {'C' : [0.01,0.02,0.03], 'kernel': ['linear', 'rbf'], 'gamma': ['scale','auto']}
X_train, X_val, X_test, y_train, y_val, y_test = split_dataset(movie_bog_vectors, movie_target, 0.8, 0.1)
#  Creating and Fitting the SVM model on the training set using Grid Search and different Hyperparameter combination
svm = SVC(random_state=random_seed)
#Creating a Grid Seach object with the SVM model and K-fold Cross validation
grid_model = GridSearchCV(svm, hyperparameter_grid, n_jobs=-1, cv=5, scoring='accuracy', return_train_score=True)
```

```
grid_model.fit(X_train, y_train)
print('Best Hyperparameters for Bag of Word Representation with Grid Search: ', grid_model.best_params_)
print('Validation Accuracy on best Hyperparameters:', np.round(accuracy_score(y_val, grid_model.predict(X_val)) * 100), 2, '%')
print('Test Accuracy on best Hyperparameters: ',np.round(accuracy_score(y_test, grid_model.predict(X_test)) * 100),2, '%')
```

```
Best Hyperparameters for Bag of Word Representation with Grid Search:  {'C': 0.02, 'gamma': 'scale', 'kernel': 'linear'}
Validation Accuracy on best Hyperparameters: 66.0 2 %
Test Accuracy on best Hyperparameters:  63.0 2 %
```

TF-IDF Representation

In [95]:
```
X_train, X_val, X_test, y_train, y_val, y_test = split_dataset(movie_tfidf_vectors, movie_target, 0.8, 0.1)
# Creating and Fitting the SVM model on the training set using Grid Search and different Hyperparameter combination
svm = SVC(random_state=random_seed)
grid_model = GridSearchCV(svm, hyperparameter_grid, n_jobs=-1, cv=5, return_train_score=True)
grid_model.fit(X_train, y_train)
print('Best Hyperparameters for TF-IDF Representation with Grid Search: ', grid_model.best_params_)
print('Validation Accuracy on best Hyperparameters: ', np.round(accuracy_score(y_val, grid_model.predict(X_val)) * 100),2, '%')
print('Test Accuracy on best Hyperparameters: ', np.round(accuracy_score(y_test, grid_model.predict(X_test)) * 100),2, '%')
```

```
Best Hyperparameters for TF-IDF Representation with Grid Search:  {'C': 0.01, 'gamma': 'scale', 'kernel': 'rbf'}
Validation Accuracy on best Hyperparameters:  49.0 2 %
Test Accuracy on best Hyperparameters:  49.0 2 %
```

In [ ]:
```

```