TextClassificationCleared

November 8, 2020

1 Import Libraries

[ ]:
```
# Installing natural language toolkit
!pip install nltk
import nltk
nltk.download('punkt')
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')
```

[ ]:
```
import numpy as np
import random as rand
import pandas as pd
import seaborn as sns
from scipy import sparse
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import KFold
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB, MultinomialNB
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
from sklearn.linear_model import SGDClassifier
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction import text
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import Normalizer
from nltk.corpus import wordnet
from nltk import word_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.stem import PorterStemmer
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.feature_selection import SelectKBest, chi2, f_classif,␣
↪mutual_info_classif, f_regression, mutual_info_regression, SelectPercentile
import math as ma
import scipy as sp
```

```python
import matplotlib.pyplot as plt
import pandas as pd
import time
print("Finished importing!")
```

2 Importing Data Sets

[ ]:
```python
from google.colab import drive
drive.mount('/content/myDrive')
```

[ ]:
```python
# df_train = pd.read_csv('/content/myDrive/My Drive/ECSE_551_Machine_Learning/
↪train.csv')
# df_test = pd.read_csv('/content/myDrive/My Drive/ECSE_551_Machine_Learning/
↪test.csv')
# df_train = pd.read_csv('C:/Users/AlexG35/Desktop/GitHub/TextClassification/
↪train.csv')
# df_test = pd.read_csv('C:/Users/AlexG35/Desktop/GitHub/TextClassification/
↪test.csv')
df_train = pd.read_csv('/content/myDrive/My Drive/ECSE_551_Machine_Learning/
↪TextClassification/train.csv')
df_test = pd.read_csv('/content/myDrive/My Drive/ECSE_551_Machine_Learning/
↪TextClassification/test.csv')
# df_train = pd.read_csv('https://raw.githubusercontent.com/jhu960213/
↪TextClassification/master/train.csv?token=AEZTLS4FRLKMK3IW2NBIKR27TQ7K6')
# df_test = ped.read_csv('https://raw.githubusercontent.com/jhu960213/
↪TextClassification/master/test.csv?token=AEZTLS6ZNQ6KBUPIGHUS7TS7TRBXQ')
# df_train = pd.read_csv('C:/Users/karan/Desktop/Masters/Codes/CodeRemote/
↪GitHub_codes/Under_Grad_mSpice/DP_teams/TextClassification/train.csv')
# df_test = pd.read_csv('C:/Users/karan/Desktop/Masters/Codes/CodeRemote/
↪GitHub_codes/Under_Grad_mSpice/DP_teams/TextClassification/test.csv')
```

3 Sample Reduction

[ ]:
```python
# Will use the copies of the data frame dont want to change our original data␣
↪frame objects when we do processing down the line
training_set = df_train
test_set = df_test["body"]
```

[ ]:
```python
# Removes samples from the training set until only remaining_samples samples␣
↪remain.
# the training set in is a pandas data frame object
def sampleReduction(training_set, remaining_samples=None):
  if remaining_samples == None:
    totalSamples = len(training_set)
```

```
    original_training_set = training_set.sample(totalSamples).
↪reset_index(drop=True)
    return original_training_set
  else:
    reduced_training_set = training_set.sample(remaining_samples).
↪reset_index(drop=True)
    return reduced_training_set
```

[ ]:
```
#--------Restrict the Number of Training Samples for Testing Purposes**---------
print("Original number of training samples: " + str(len(training_set)))
training_set = sampleReduction(training_set)
print("Reduced number of training samples: " + str(len(training_set)))
#--------Restrict the Number of Training Samples for Testing Purposes**---------
```

[ ]:
```
# Fixing the seed of the rand
# np.random.seed(5)

# Splitting our data into X and Y
training_set.sample(frac=1)
print("Finished shuffling our data sets")
Xtraining = training_set["body"]
ylabels = training_set["subreddit"]

print(f"Xtraining shape: {Xtraining.shape}")
print(f"ylabels shape: {Xtraining.shape}")
```

4 Visualization of Data Sets

[ ]:
```
# Prints out however many rows of our data
def printTrainingSet(X,numRows):
  print(f"Visualizing our sample texts:\n {X[:numRows]}")
```

[ ]:
```
printTrainingSet(Xtraining, 10)
```

[ ]:
```
# print(ylabels[0:8])
# Displays the distribution of our labels
plt.figure(figsize=(15, 10))
sns.countplot(ylabels)
plt.title("Distribution of Labels")
plt.xlabel("Labels")
plt.ylabel("Count")
# plt.xticks(np.arange(8), ylabels, rotation="horizontal")
```

5 Data Preprocessing

Our own data preprocessing functions

[ ]:
```python
#@title Dictionary of irrelevant words.
# Dictionary of irrelevant words.
# Returns true if the *LOWERCASE* word is irrelevant, else returns false.
def isIrrelevant (word):
  switcher = {
    # All pronouns and associated words
    "i": True,
    "i'll": True,
    "i'd": True,
    "i'm": True,
    "i've": True,
    "ive": True,
    "me": True,
    "myself": True,
    "you": True,
    "you'll": True,
    "you'd": True,
    "you're": True,
    "you've": True,
    "yourself": True,
    "he": True,
    "he'll": True,
    "he'd": True,
    "he's": True,
    "him": True,
    "she": True,
    "she'll": True,
    "she'd": True,
    "she's": True,
    "her": True,
    "it": True,
    "it'll": True,
    "it'd": True,
    "it's": True,
    "itself": True,
    "oneself": True,
    "we": True,
    "we'll": True,
    "we'd": True,
    "we're": True,
    "we've": True,
    "us": True,
    "ourselves": True,
    "they": True,
```

```python
    "they'll": True,
    "they'd": True,
    "they're": True,
    "they've": True,
    "them": True,
    "themselves": True,
    "everyone": True,
    "everyone's": True,
    "everybody": True,
    "everybody's": True,
    "someone": True,
    "someone's": True,
    "somebody": True,
    "somebody's": True,
    "nobody": True,
    "nobody's": True,
    "anyone": True,
    "anyone's": True,
    "everything": True,
    "everything's": True,
    "something": True,
    "something's": True,
    "nothing": True,
    "nothing's": True,
    "anything": True,
    "anything's": True,
    # All determiners and associated words
    "a": True,
    "an": True,
    "the": True,
    "this": True,
    "that": True,
    "that's": True,
    "these": True,
    "those": True,
    "my": True,
    #"mine": True,   #Omitted since mine can refer to something else
    "your": True,
    "yours": True,
    "his": True,
    "hers": True,
    "its": True,
    "our": True,
    "ours": True,
    "own": True,
    "their": True,
    "theirs": True,
```

```
    "few": True,
    "much": True,
    "many": True,
    "lot": True,
    "lots": True,
    "some": True,
    "any": True,
    "enough": True,
    "all": True,
    "both": True,
    "half": True,
    "either": True,
    "neither": True,
    "each": True,
    "every": True,
    "certain": True,
    "other": True,
    "another": True,
    "such": True,
    "several": True,
    "multiple": True,
    # "what": True,    #Dealt with later on
    "rather": True,
    "quite": True,
    # All prepositions
    "aboard": True,
    "about": True,
    "above": True,
    "across": True,
    "after": True,
    "against": True,
    "along": True,
    "amid": True,
    "amidst": True,
    "among": True,
    "amongst": True,
    "anti": True,
    "around": True,
    "as": True,
    "at": True,
    "away": True,
    "before": True,
    "behind": True,
    "below": True,
    "beneath": True,
    "beside": True,
    "besides": True,
```

```python
    "between": True,
    "beyond": True,
    "but": True,
    "by": True,
    "concerning": True,
    "considering": True,
    "despite": True,
    "down": True,
    "during": True,
    "except": True,
    "excepting": True,
    "excluding": True,
    "far": True,
    "following": True,
    "for": True,
    "from": True,
    "here": True,
    "here's": True,
    "in": True,
    "inside": True,
    "into": True,
    "left": True,
    "like": True,
    "minus": True,
    "near": True,
    "of": True,
    "off": True,
    "on": True,
    "onto": True,
    "opposite": True,
    "out": True,
    "outside": True,
    "over": True,
    "past": True,
    "per": True,
    "plus": True,
    "regarding": True,
    "right": True,
    #"round": True,    #Omitted
    #"save": True,     #Omitted
    "since": True,
    "than": True,
    "there": True,
    "there's": True,
    "through": True,
    "to": True,
    "toward": True,
```

```python
    "towards": True,
    "under": True,
    "underneath": True,
    "unlike": True,
    "until": True,
    "up": True,
    "upon": True,
    "versus": True,
    "via": True,
    "with": True,
    "within": True,
    "without": True,
    # Irrelevant verbs
    "may": True,
    "might": True,
    "will": True,
    "won't": True,
    "would": True,
    "wouldn't": True,
    "can": True,
    "can't": True,
    "cannot": True,
    "could": True,
    "couldn't": True,
    "should": True,
    "shouldn't": True,
    "must": True,
    "must've": True,
    "be": True,
    "being": True,
    "been": True,
    "am": True,
    "are": True,
    "aren't": True,
    "ain't": True,
    "is": True,
    "isn't": True,
    "was": True,
    "wasn't": True,
    "were": True,
    "weren't": True,
    "do": True,
    "doing": True,
    "don't": True,
    "does": True,
    "doesn't": True,
    "did": True,
```

```
    "didn't": True,
    "done": True,
    "have": True,
    "haven't": True,
    "having": True,
    "has": True,
    "hasn't": True,
    "had": True,
    "hadn't": True,
    "get": True,
    "getting": True,
    "gets": True,
    "got": True,
    "gotten": True,
    "go": True,
    "going": True,
    "gonna": True,
    "goes": True,
    "went": True,
    "gone": True,
    "make": True,
    "making": True,
    "makes": True,
    "made": True,
    "take": True,
    "taking": True,
    "takes": True,
    "took": True,
    "taken": True,
    "need": True,
    "needing": True,
    "needs": True,
    "needed": True,
    "use": True,
    "using": True,
    "uses": True,
    "used": True,
    "want": True,
    "wanna": True,
    "wanting": True,
    "wants": True,
    "let": True,
    "lets": True,
    "letting": True,
    "let's": True,
    "suppose": True,
    "supposing": True,
```

```
    "supposes": True,
    "supposed": True,
    "seem": True,
    "seeming": True,
    "seems": True,
    "seemed": True,
    "say": True,
    "saying": True,
    "says": True,
    "said": True,
    "know": True,
    "knowing": True,
    "knows": True,
    "knew": True,
    "known": True,
    "look": True,
    "looking": True,
    "looked": True,
    "think": True,
    "thinking": True,
    "thinks": True,
    "thought": True,
    "feel": True,
    "feels": True,
    "felt": True,
    "based": True,
    "put": True,
    "puts": True,
    "begin": True,
    "began": True,
    "begun": True,
    "begins": True,
    "wanted": True,
    "like": True,
    "feel": True,
    "believe": True,
    "understand": True,
    "shall": True,
    "regard": True,
    "regards": True,
    "regarding": True,
    # Question words and associated words
    "who": True,
    "who's": True,
    "who've": True,
    "who'd": True,
    "whoever": True,
```

```python
    "whoever's": True,
    "whom": True,
    "whomever": True,
    "whomever's": True,
    "whose": True,
    "whosever": True,
    "whosever's": True,
    "when": True,
    "whenever": True,
    "which": True,
    "whichever": True,
    "where": True,
    "where's": True,
    "where'd": True,
    "wherever": True,
    "why": True,
    "why's": True,
    "why'd": True,
    "whyever": True,
    "what": True,
    "what's": True,
    "whatever": True,
    "whence": True,
    "how": True,
    "how's": True,
    "how'd": True,
    "however": True,
    "whether": True,
    "whatsoever": True,
    # Connector words and irrelevant adverbs
    "and": True,
    "or": True,
    "not": True,
    "because": True,
    "also": True,
    "always": True,
    "never": True,
    "only": True,
    "really": True,
    "very": True,
    "greatly": True,
    "extremely": True,
    "somewhat": True,
    "no": True,
    "nope": True,
    "nah": True,
    "yes": True,
```

```
    "yep": True,
    "yeh": True,
    "yeah": True,
    "maybe": True,
    "perhaps": True,
    "more": True,
    "most": True,
    "less": True,
    "least": True,
    "good": True,
    "great": True,
    "well": True,
    "better": True,
    "best": True,
    "bad": True,
    "worse": True,
    "worst": True,
    "too": True,
    "thru": True,
    "though": True,
    "although": True,
    "yet": True,
    "already": True,
    "then": True,
    "even": True,
    "now": True,
    "sometimes": True,
    "still": True,
    "together": True,
    "altogether": True,
    "entirely": True,
    "fully": True,
    "entire": True,
    "whole": True,
    "completely": True,
    "utterly": True,
    "seemingly": True,
    "apparently": True,
    "clearly": True,
    "obviously": True,
    "actually": True,
    "actual": True,
    "usually": True,
    "usual": True,
    "literally": True,
    "honestly": True,
    "absolutely": True,
```

```
    "definitely": True,
    "generally": True,
    "totally": True,
    "finally": True,
    "basically": True,
    "essentially": True,
    "fundamentally": True,
    "automatically": True,
    "immediately": True,
    "necessarily": True,
    "primarily": True,
    "normally": True,
    "perfectly": True,
    "constantly": True,
    "particularly": True,
    "eventually": True,
    "hopefully": True,
    "mainly": True,
    "typically": True,
    "specifically": True,
    "differently": True,
    "appropriately": True,
    "plenty": True,
    "certainly": True,
    "unfortunately": True,
    "ultimately": True,
    "unlikely": True,
    "likely": True,
    "potentially": True,
    "fortunately": True,
    "personally": True,
    "directly": True,
    "indirectly": True,
    "nearly": True,
    "closely": True,
    "slightly": True,
    "probably": True,
    "possibly": True,
    "especially": True,
    "frequently": True,
    "thankfully": True,
    "often": True,
    "oftentimes": True,
    "seldom": True,
    "rarely": True,
    "sure": True,
    "while": True,
```

```
    "whilst": True,
    "able": True,
    "unable": True,
    "else": True,
    "ever": True,
    "once": True,
    "twice": True,
    "thrice": True,
    "almost": True,
    "again": True,
    "instead": True,
    "next": True,
    "previous": True,
    "unless": True,
    "somehow": True,
    "anyhow": True,
    "anywhere": True,
    "somewhere": True,
    "everywhere": True,
    "elsewhere": True,
    "anytime": True,
    "nowhere": True,
    "further": True,
    "anymore": True,
    "later": True,
    "ago": True,
    "ahead": True,
    "just": True,
    "same": True,
    "different": True,
    "big": True,
    "small": True,
    "little": True,
    "tiny": True,
    "large": True,
    "huge": True,
    "pretty": True,
    "mostly": True,
    "anyway": True,
    "anyways": True,
    "otherwise": True,
    "regardless": True,
    "needless": True,
    "throughout": True,
    "additionally": True,
    "moreover": True,
    "furthermore": True,
```

```python
    "therefore": True,
    "thereof": True,
    "meanwhile": True,
    "likewise": True,
    "afterwards": True,
    "nice": True,
    "nicer": True,
    "nicest": True,
    "glad": True,
    "fine": True,
    # Irrelevant nouns
    "thing": True,
    "thing's": True,
    "things": True,
    "stuff": True,
    "other's": True,
    "others": True,
    "another's": True,
    "total": True,
    "true": True,
    "false": True,
    "none": True,
    "way": True,
    "kind": True,
    # Lettered numbers and order
    "zero": True,
    "zeros": True,
    "zeroes": True,
    "one": True,
    "ones": True,
    "two": True,
    "three": True,
    "four": True,
    "five": True,
    "six": True,
    "seven": True,
    "eight": True,
    "nine": True,
    "ten": True,
    "twenty": True,
    "thirty": True,
    "forty": True,
    "fifty": True,
    "sixty": True,
    "seventy": True,
    "eighty": True,
    "ninety": True,
```

```python
    "hundred": True,
    "hundreds": True,
    "thousand": True,
    "thousands": True,
    "million": True,
    "millions": True,
    "first": True,
    "last": True,
    "second": True,
    "third": True,
    "fourth": True,
    "fifth": True,
    "sixth": True,
    "seventh": True,
    "eigth": True,
    "ninth": True,
    "tenth": True,
    "firstly": True,
    "secondly": True,
    "thirdly": True,
    "lastly": True,
    # Greetings and slang
    "hello": True,
    "hi": True,
    "hey": True,
    "sup": True,
    "yo": True,
    "greetings": True,
    "please": True,
    "okay": True,
    "ok": True,
    "y'all": True,
    "lol": True,
    "rofl": True,
    "thank": True,
    "thanks": True,
    "alright": True,
    "kinda": True,
    "dont": True,
    "sorry": True,
    "idk": True,
    "tldr": True,
    "tl": True,
    "dr": True,  #This means that dr (doctor) is a bad feature because of tl;dr
    "tbh": True,
    "dude": True,
    "dudes": True,
```

```python
    "tho": True,
    "aka": True,
    "plz": True,
    "pls": True,
    "bit": True,
    "don": True,
    "afaik": True,
    "wouldn": True,
    "wouldnt": True,
    "doesnt": True,
    "doesn": True,
    "didn": True,
    "didnt": True,
    "haven": True,
    "havent": True,
    "ugh": True,
    "legit": True,
    "guess": True,
    "bullshit": True,
    "yup": True,
    "yep": True,
    "haha": True,
    "hahaha": True,
    "hahahaha": True,
    "hehe": True,
    "hehehe": True,
    "till": True,
    "sure": True,
    "soon": True,
    "nah": True,
    "meh": True,
    "imo": True,
    "imho": True,
    "ill": True,
    "hella": True,
    "chill": True,
    "btw": True,
    "bro": True,

    # Miscellaneous
    "www": True,
    "https": True,
    "http": True,
    "com": True,
    "etc": True,
    "html": True,
    "reddit": True,
```

```
    "subreddit": True,
    "subreddits": True,
    "comments": True,
    "reply": True,
    "replies": True,
    "thread": True,
    "threads": True,
    "post": True,
    "posts": True,
    "website": True,
    "websites": True,
    "web site": True,
    "web sites": True
  }
  return switcher.get(word,False)
```

[ ]:

```
#@title Dictionary of relevant characters in a word.
#@ Dictionary of relevant characters in a word.
# Dictionary of relevant characters in a word.
# Returns true if the character is a *LOWERCASE* letter, number or accepted
# special character, else returns false.
def isAlphanumeric (char):
  switcher = {

    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ':True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
```

```
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ':True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    ' ': True,
    'a': True,
    'b': True,
    'c': True,
    'd': True,
    'e': True,
    'f': True,
    'g': True,
    'h': True,
    'i': True,
    'j': True,
    'k': True,
    'l': True,
    'm': True,
    'n': True,
    'o': True,
    'p': True,
    'q': True,
    'r': True,
    's': True,
    't': True,
    'u': True,
    'v': True,
    'w': True,
    'x': True,
    'y': True,
    'z': True,
    '0': True,
    '1': True,
    '2': True,
```

```
    '3': True,
    '4': True,
    '5': True,
    '6': True,
    '7': True,
    '8': True,
    '9': True,
    '\'': True,   #Apostrophe
    '$': True,
    #'@': True,    #For email addresses
    #'_': True,
    '-': True,
    #'/': True     #For websites
  }
  return switcher.get(char,False)
```

[ ]:
```
# Joins the characters in a list of characters together to form a string.
def mergeCharacters(char_array):
  # Intialize string to ""
  s = ""

  return(s.join(char_array))
```

[ ]:
```
# Returns true if character is a number, false otherwise.
def isNumber(char):
  switcher = {
    '0': True,
    '1': True,
    '2': True,
    '3': True,
    '4': True,
    '5': True,
    '6': True,
    '7': True,
    '8': True,
    '9': True,
  }
  return switcher.get(char,False)
```

[ ]:
```
# Parses the input string and returns its words in the form of a list of␣
↪strings.
def parseLine(line):
  line = line.lower()     # Converts all letters to lowercase
  char_array = list(line) # Converts the line to an list of characters
  words_array = [[]]      # List of a list of characters that for words
  words = []              # List of words
  word_idx = 0      # Index of the current word
```

```
  # Creates a list of words made out of characters (i.e. a list of lists of␣
↪char).
  for char in char_array:
    if isAlphanumeric(char):
      temp = words_array[word_idx]     # temp is the current word
      temp.append(char)
      #char_idx += 1
    else:
      if words_array[word_idx]:       # True if word contains at least 1 character
        words_array.append([])        # Appending the next empty word to be filled
        word_idx += 1                 # Index into the next word
  # Creates a list of words made out of strings.
  for word in words_array:
    if word:                          # If word is non-empty. Always true␣
↪(hopefully) except for the last one)
      dollar_sign = False
      contains_number = False
      letter_number = False
      for char in word:
        if char == '$':
          dollar_sign = True
        elif contains_number and (not isNumber(char)):
          letter_number = True
        elif isNumber(char):
          contains_number = True
      if dollar_sign:
        words.append("money")         # Substitute any money amount with simply␣
↪"money"
      elif letter_number:
        words.append("alphanum")      # Substitute any number with letters with␣
↪"alphanum"
      elif contains_number:
        words.append("number")        # Substitute any number with simply "number"
      else:
        temp = mergeCharacters(word)
        if not isIrrelevant(temp) and len(temp) > 2:
          words.append(temp)  # Merge the characters of the word into a string
  return words
```

Conversion of class labels into binary numbers

[ ]:
```
# Convert the class labels to integers from 0 to 7 using a dictionary
def numberToLabel (num):
  switcher = {
     0:"rpg",
     1: "anime",
     2:"datascience",
```

```
    3:"hardware",
    4:"cars",
    5:"gamernews",
    6:"gamedev",
    7:"computers"
  }
  return switcher.get(num,"Invalid class label")
```

[ ]:
```
# Convert the class labels to integers from 0 to 7 using a dictionary
def labelToNumber (label):
  switcher = {
    "rpg": 0,
    "anime": 1,
    "datascience": 2,
    "hardware": 3,
    "cars": 4,
    "gamernews": 5,
    "gamedev": 6,
    "computers": 7,
  }
  return switcher.get(label,"Invalid class label")
```

[ ]:
```
# Convert the class labels into a numpy array named Y.
# Each entry in Y should still match the corresponding row in X.
Y = np.zeros((ylabels.shape[0],1))
for i in range(0, Y.shape[0]):
  # Convert the class labels to a number between 0 and 7
  label_number = labelToNumber(ylabels[i])
  if label_number != "Invalid class label":
    Y[i,0] = label_number
  else:
    print("Invalid class label!")

# Reshaping our labels
# Y = np.reshape(Y,(Y.shape[0],))
print(f"Y shape as numpy: {Y.shape}")
```

Lemmatization & Stemming

[ ]:
```
# Map the pos tag to first character that lemmatize accepts
def get_wordnet_pos(word):
  # my tag returns an upper case letter representing nouns, verbs, adverbs, etc
  """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)
```

[ ]:
```
# Create a new class for that does word tokenizing combined with word␣
↪lemmatization
class MyLemmaTokenizer:
  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()]
```

[ ]:
```
# Create my stemming object for mophological variants of root/base words␣
↪findings
class MyStemTokenizer:
  # Constructor
  def __init__(self):
    self.myPorterStemmer = PorterStemmer()
  # It does the stemmization
  def __call__(self, document):
    return [self.myPorterStemmer.stem(j) for j in word_tokenize(document) if j.
↪isalpha()]
```

[ ]:
```
# Get all my english stop words
myStopWords = text.ENGLISH_STOP_WORDS
myStopWords = list(myStopWords)
print(myStopWords)
myStopWords.append("_")

# #Adding more stop words to the imported list of stop words
# file_path = "/content/myDrive/My Drive/ECSE_551_Machine_Learning/
↪TextClassification/stopwords.txt"
# with open(file_path,mode='r') as file:
#   content = file.readlines()
#   for word in content:
#     myStopWords.append(str(word))
```

[ ]:
```
# Getting rid of all the bad characters and words
Xtraining_list = Xtraining.to_list()
X_words1 = []
for sample in Xtraining_list:
  tmp = parseLine(sample)
  s = ' '
  s = s.join(tmp)
  X_words1.append(s)
```

```
[ ]: # Converting the list of words back into a pandas object
Xtraining = pd.DataFrame(X_words1)
Xtraining = Xtraining[0]
print(Xtraining[0])
print(Xtraining.shape)
```

```
[ ]: # Getting rid of all bad characters and words in test set
Xtest_list = test_set.to_list()
X_words2 = []
for sample in Xtest_list:
  tmp = parseLine(sample)
  s = ' '
  s = s.join(tmp)
  X_words2.append(s)
```

```
[ ]: # Converting the list of test words into a pandas object
Xtest = pd.DataFrame(X_words2)
Xtest = Xtest[0]
print(Xtest[0])
print(Xtest.shape)
```

6 Naive Bayes Classifier

```
[ ]: # Superclass for Bernouilli Naive Bayes Classifiers
class Bernoulli_NB():

    #Class constructor
    def __init__(self, alpha=0.01):
      self.num_classes = 8
      self.condProb = None
      self.priorProb = None
      self.alpha = alpha
  ␣
 ↪##########################################################################################
  ␣
 ↪##########################################################################################

    def set_params(self, **params):
      # self.alpha = params["alpha"]
      pass

    def get_params(self, deep=False):
      parameters = {"alpha": self.alpha}
      return parameters

    # Calculate the probability of Y = 1.
```

```python
    # Returns a column vector for which the ith entry is P(Yi=1)
    # and i refers to a class label.
    def probY(self, Y):

      Y = np.reshape(Y, (Y.shape[0],1))
      num_labels = self.num_classes
      prob = np.zeros((num_labels,1))  #Probability vector P(Yi=1)

      for i in range(0,Y.shape[0]):
        # Assumption: labels are integers ranging from 0 to num_labels - 1
        for label in range(0, num_labels):
          if Y[i,0] == label:
            prob[label,0] += 1
          elif Y[i,0] > (num_labels - 1):
            print("Y at index " + str(i) + " is an invalid class label")
            break
      # Divide by the total # of labels input labels Y
      prob = prob/Y.shape[0]
      return prob

  ␣
↪##########################################################################################
  ␣
↪##########################################################################################
    # Calculate the probability of X = 1 given a class label Yi (number).
    # Returns a row vector for which the jth entry is P(Xj=1|Yi)
    # and j refers to a feature.
    # Note: not implemented with the bias term in mind.
    def probXGivenYi(self, X, Y, label):

      Y = np.reshape(Y, (Y.shape[0],1))
      prob = np.zeros((1,X.shape[1]))  # Conditional probability vector␣
↪P(Xj=1|Yi)
      denominator = 0                  # Number of times label Yi appears in Y

      for i in range(0, X.shape[0]):
        if Y[i,0] == label:
          denominator += 1
          prob = prob + X[i,:]

      # Laplace smoothing
      if(self.alpha == 1):
        prob = prob + np.ones((1,X.shape[1]))
        denominator += 2
      else:
        prob = prob + self.alpha*np.ones((1,X.shape[1]))
        denominator += self.alpha * 2
```

```
    prob = prob/denominator
    return prob
␣
↪##########################################################################################
␣
↪##########################################################################################
  # Write the Bernoulli Naive  Bayes Method Here
  def fit(self, Xtrain, Y):
      Y = np.reshape(Y, (Y.shape[0],1))
      # print('Starting the fit function::::')
      t1 = time.time()
      self.condProb = np.zeros((self.num_classes,Xtrain.shape[1]))

      t2 =time.time()
      self.priorProb = self.probY(Y)
      # print('Time taken for Pior probs function:::', time.time()-t2 )

      t3 = time.time()
      for c in range(self.num_classes):
          self.condProb[c,:] = self.probXGivenYi(Xtrain, Y, c)
      # print('Time taken for Condtional Prob:::', time.time()-t3)
      # print('Total time by fit function:::', time.time()-t1 )
␣
↪##########################################################################################
␣
↪##########################################################################################
                          #NON OPTIMISED BERNOULLI #
  # def predict(self,Xtest):
  #
  #     predLabel = np.zeros((Xtest.shape[0],1))
  #     prosteriorProb = np.zeros((Xtest.shape[0],8))
  #
  #     for d in range(Xtest.shape[0]):
  #         for c in range(8):
  #             prosteriorProb[d,c] = np.log10(self.priorProb[c,0])
  #             for index in range(Xtest.shape[1]):
  #                 if(Xtest[d,index] == 1):
  #                     prosteriorProb[d,c] += np.log10(self.
↪condProb[c,index])
  #                 else:
  #                     prosteriorProb[d,c] += np.log10(1 - self.
↪condProb[c,index])
  #
  #                 # print(self.condProb[c, index])
  #
  #         predLabel[d,0] = np.argmax(prosteriorProb[d,:])
```

```
    #         # print(prosteriorProb[d, :])
    #         # print(predLabel[d,0])
    #         # if d == 5:
    #         #     return
    #
    #     return predLabel
␣
↪################################################################################
␣
↪################################################################################
                              # OPTIMISED BERNOULLI #
    def predict(self,Xtest):
        print(f'Starting the Predicting Function....')
        start_time = time.time()
        predLabel = np.zeros((Xtest.shape[0],1), dtype=int)
        prosteriorProb = np.zeros((Xtest.shape[0],self.num_classes))
        for d in range(Xtest.shape[0]):
            for c in range(self.num_classes):
                prosteriorProb[d,c] = np.log10(self.priorProb[c,0])
                # for index in range(Xtest.shape[1]):
                #     if(Xtest[d,index] == 1):
                #         prosteriorProb[d,c] += np.log10(self.
↪condProb[c,index])
                #     else:
                #         prosteriorProb[d,c] += np.log10(1 - self.
↪condProb[c,index])
                Z = Xtest[d,:]
                # print(Xtest.shape)
                # print(type(Xtest))
                # print(Z.shape)

                # print(type(Z))

                # print(sp.nonzero(Z))
                # one_ind = np.where(Z==1)[0]
                one_ind = sp.nonzero(Z)[1]
                zero_ind = np.delete(np.arange(0,Xtest.shape[0]),one_ind,0)
                row_ones = np.ones((1,zero_ind.shape[0]))
                prosteriorProb[d,c] += np.sum(np.log10(self.
↪condProb[c,one_ind]))
                prosteriorProb[d,c] +=  np.sum(np.log10(row_ones-self.
↪condProb[c,zero_ind]))


            predLabel[d,0] = np.argmax(prosteriorProb[d,:])
        print(f'Time taken for predict function: {time.time()-start_time}')
```

```
    return predLabel
```

7 Ensemble Bagging

[ ]:
```
def featureVectorizer(myVectorizer, tfidfNormalizer, Xtraining, numFeatures):

  # Making my frequency vectors (either binary or non binary depends on␣
↪CountVectorizer)
  training_vectors = myVectorizer.fit_transform(Xtraining)

  # TFIDF normalizing
  training_vectors_tfidf_normalized = tfidfNormalizer.
↪fit_transform(training_vectors)

  # Using Sklearns function to help us select the top features
  training_vectors_tfidf_normalized_new = SelectKBest(chi2, k=numFeatures).
↪fit_transform(training_vectors_tfidf_normalized, Y)

  # Look at feature names for both and extract and save a list of them for␣
↪viewing
  myFeatures = myVectorizer.get_feature_names()
  path1 = "/content/myDrive/My Drive/ECSE_551_Machine_Learning/
↪TextClassification/features1.txt"
  # Writing the feaure names to a text file
  with open(path1, mode='w') as file:
    for item in myFeatures:
      file.write("%s\n" % item)

  return training_vectors_tfidf_normalized_new, myVectorizer, tfidfNormalizer
```

[ ]:
```
#
def bootstrapWithReplacement(X, Y):
  X = X.toarray()
  Xout = np.zeros((X.shape[0],X.shape[1]))
  Yout = np.zeros((Y.shape[0],Y.shape[1]))
  for i in range(0,X.shape[0]):
    rand_idx = rand.randint(0,X.shape[0]-1)
    Xout[i,:] = X[rand_idx,:]
    Yout[i,:] = Y[rand_idx,:]
  return Xout, Yout
```

[ ]:
```
# Our bootstrap ensemble classifier
def ensembleBagging(myVectorizer, mytfidfTransformer, num_features, alpha,␣
↪Xtraining, Y, Xtest, B, path):
```

```
# feature selector from sklearn
skLearnFeatureSelector = SelectKBest(chi2, k=num_features)

# vectorization of our training and testing data
vectors_train = myVectorizer.fit_transform(Xtraining)
vectors_test = myVectorizer.transform(Xtest)

# tfidf normalization
vectors_train_tfidf_normalized = mytfidfTransformer.
↪fit_transform(vectors_train)
vectors_test_tfidf_normalized = mytfidfTransformer.transform(vectors_test)

# selecting the top features using chi squared evaluation
vectors_train_tfidf_normalized_new = skLearnFeatureSelector.
↪fit_transform(vectors_train_tfidf_normalized, Y)
vectors_test_tfidf_normalized_new = skLearnFeatureSelector.
↪transform(vectors_test_tfidf_normalized)

# accuracies list
accuracies = np.zeros((1,B))

# predicted labels for each of our classifier
predictedLabels = np.zeros((Xtest.shape[0], B), dtype=int)

for i in range(0,B):

  # instantiating a different instance of NB for classification
  nb = Bernoulli_NB(alpha=alpha)

  # bootstrap our data (sample with replacement)
  Xresampled, Yresampled =␣
↪bootstrapWithReplacement(vectors_train_tfidf_normalized_new, Y)

  # convert back from numpy to sparse
  Xresampled = sparse.csr_matrix(Xresampled)

  # # Run k-fold to get the model (classifier) accuracies
  # avgErrorOfThisModel, avgAccuracyOfThisModel, fold_Accuracy_Dict =␣
↪run_K_Fold_CrossValidation(Xresampled, Yresampled, nb, numFolds=3)
  # # accuracies.append(avgAccuracyOfThisModel)
  # accuracies[0,i] = avgAccuracyOfThisModel

  # Evaluating using our classifier
  nb.fit(Xresampled, Yresampled)
  tmp = nb.predict(vectors_test_tfidf_normalized_new)
  predictedLabels[:,i] = np.reshape(tmp, (tmp.shape[0],))
```

```
# Majority voting
finalLabel = np.zeros((predictedLabels.shape[0], 1))
for i in range(0, predictedLabels.shape[0]):
  frequencies = np.zeros((1,8), dtype=int)
  for j in range(B):
    frequencies[0,predictedLabels[i,j]] += 1
  max = np.max(frequencies)
  idx = [] # Classifier indices
  for x in range(0,frequencies.shape[1]):
    if frequencies[0,x] == max:
      idx.append(x)
  if len(idx) > 1:
    print(idx)
    finalLabel[i,0] = idx[1]
  else:
    finalLabel[i,0] = idx[0]


# Convert number labels back into string labels
predictedLabelsStringFormat = []
for labelNum in finalLabel:
  predictedLabelsStringFormat.append(numberToLabel(int(labelNum[0])))

print(f"Length of predicted list: {len(predictedLabelsStringFormat)}")
# print(predictedLabelsStringFormat)
# Convert from list to pandas data frame
dfPredicted = pd.DataFrame(predictedLabelsStringFormat, columns=['subreddit'])
# print(dfPredicted)
dfPredicted.to_csv(path)
return finalLabel
```

8 Pipeline & GridsearchCV

SKLearn SVC Tunning

[ ]:
```
# Defining my pipeline parameters for my gridsearchCV to tweak at run time
pipeline_parameters = {
    "classifier__max_iter": [500000],
    "classifier__multi_class": ['ovr', 'crammer_singer'],
    "classifier__tol": [1e-4],
    "classifier__C": [1.0],
    "classifier__dual": [True],
    "classifier__loss": ['squared_hinge'],
    # "classifier__penalty": ['l1', 'l2'],
    "countVectorizer__binary": [False],
    "countVectorizer__max_df": [1.0],
```

```
    "countVectorizer__min_df": [0.0],
    "countVectorizer__stop_words": [myStopWords],
    "countVectorizer__max_features": [5500],
    "countVectorizer__ngram_range":[(1,1)],
    "countVectorizer__tokenizer": [MyStemTokenizer(), MyLemmaTokenizer()],
    "normalizer__norm": ['l2']
}

# adaboostCLF = AdaBoostClassifier(base_estimator=Bernoulli_NB(),␣
 ↪n_estimators=8, random_state=0, algorithm='SAMME')

# Making my pipeline to have these preprocessing functions
myPipe = Pipeline(
    # [('countVectorizer', CountVectorizer()), ('classifier', LinearSVC())]
    [('countVectorizer', CountVectorizer()), ('normalizer', Normalizer()),␣
 ↪('classifier', LinearSVC())]
)

# Making a grid search object in order to optimize our parameters
myGridSearch = GridSearchCV(myPipe, param_grid=pipeline_parameters,␣
 ↪scoring="accuracy", n_jobs=-2, verbose=1)
print("Performing grid search...")
print("Pipeline: ", [name for name, _ in myPipe.steps])
t0 = time.time()
Ygridsearch = np.reshape(Y, (Y.shape[0],))
print(Xtraining.shape)
myGridSearch.fit(Xtraining, Ygridsearch)
print("Finished in %0.3fs" % (time.time() - t0))
print("\n")
```

[ ]:
```
print("Best score: %0.3f" % myGridSearch.best_score_)
print("Best parameters set:")
bestParameters = myGridSearch.best_estimator_.get_params()
# print(bestParameters)
for paramName in sorted(bestParameters.keys()):
  print("\t%s: %r" % (paramName, bestParameters[paramName]))
```

SKLearn LinearSVC Tuning

[ ]:
```
# Defining my pipeline parameters for my gridsearchCV to tweak at run time
pipeline_parameters = {
    "classifier__max_iter": [10000000],
    "classifier__multi_class": ['ovr'],
    "classifier__tol": [1e-4],
    "classifier__C": [1.0],
    "classifier__dual": [True],
    "classifier__loss": ['squared_hinge'],
```

```
    "classifier__penalty": ['l2'],
    "countVectorizer__binary": [False],
    "countVectorizer__max_df": [1.0],
    "countVectorizer__min_df": [0.0],
    "countVectorizer__stop_words": [myStopWords],
    "countVectorizer__max_features": [15000],
    "countVectorizer__ngram_range":[(1,1)],
    # "countVectorizer__tokenizer": [MyStemTokenizer()],
    "normalizer__norm": ['l2']
}

# adaboostCLF = AdaBoostClassifier(base_estimator=Bernoulli_NB(),␣
 ↪n_estimators=8, random_state=0, algorithm='SAMME')

# Making my pipeline to have these preprocessing functions
myPipe = Pipeline(
    # [('countVectorizer', CountVectorizer()), ('classifier', LinearSVC())]
    [('countVectorizer', CountVectorizer()), ('normalizer', Normalizer()),␣
 ↪('classifier', LinearSVC())]
)

# Making a grid search object in order to optimize our parameters
myGridSearch = GridSearchCV(myPipe, param_grid=pipeline_parameters,␣
 ↪scoring="accuracy", n_jobs=-2, verbose=1)
print("Performing grid search...")
print("Pipeline: ", [name for name, _ in myPipe.steps])
t0 = time.time()
Ygridsearch = np.reshape(Y, (Y.shape[0],))
print(Xtraining.shape)
myGridSearch.fit(Xtraining, Ygridsearch)
print("Finished in %0.3fs" % (time.time() - t0))
print("\n")
```

[ ]:
```
print("Best score: %0.3f" % myGridSearch.best_score_)
print("Best parameters set:")
bestParameters = myGridSearch.best_estimator_.get_params()
# print(bestParameters)
for paramName in sorted(bestParameters.keys()):
  print("\t%s: %r" % (paramName, bestParameters[paramName]))
```

Sklearn Multinomial NB Tuning

[ ]:
```
# Defining my pipeline parameters for my gridsearchCV to tweak at run time
pipeline_Params_Multi_Bayes = {
    "classifier__alpha": [0.06, 0.04, 0.02],
    "countVectorizer__binary": [False],
    "countVectorizer__max_df": [0.1],
```

```
    "countVectorizer__min_df": [1],
    "countVectorizer__stop_words": [myStopWords],
    "countVectorizer__max_features": [5000],
    "countVectorizer__ngram_range":[(1,1)],
    "countVectorizer__tokenizer": [MyStemTokenizer()],
    # "normalizer__norm": ['l2','l1']
}

# adaboostCLF = AdaBoostClassifier(base_estimator=Bernoulli_NB(),␣
↪n_estimators=8, random_state=0, algorithm='SAMME')

# Making my pipeline to have these preprocessing functions
myPipeMultiBayes = Pipeline(
    # [('countVectorizer', CountVectorizer()), ('classifier', Bernoulli_NB())]
    [('countVectorizer', CountVectorizer()), ('classifier', MultinomialNB())]
)

# Making a grid search object in order to optimize our parameters
myGridSearchMultiBayes = GridSearchCV(myPipeMultiBayes,␣
↪param_grid=pipeline_Params_Multi_Bayes, scoring="accuracy", n_jobs=-2,␣
↪verbose=1)
# Xnew = SelectKBest(chi2, k=5000).fit_transform(Xtraining, Y)
print("Performing grid search...")
print("Pipeline: ", [name for name, _ in myPipeMultiBayes.steps])
t0 = time.time()
YMultiBayes = np.reshape(Y, (Y.shape[0],))
myGridSearchMultiBayes.fit(Xtraining, YMultiBayes)
print("Finished in %0.3fs" % (time.time() - t0))
print("\n")
```

[ ]:
```
print("Best score: %0.3f" % myGridSearchMultiBayes.best_score_)
print("Best parameters set:")
bestParameters = myGridSearchMultiBayes.best_estimator_.get_params()
# print(bestParameters)
for paramName in sorted(bestParameters.keys()):
  print("\t%s: %r" % (paramName, bestParameters[paramName]))
```

Bernoulli Naive Bayes Tunning

[ ]:
```
# Defining my pipeline parameters for my gridsearchCV to tweak at run time
pipeline_parameters = {
    "classifier__alpha": [0.01],
    "countVectorizer__binary": [False],
    "countVectorizer__max_df": [1.0],
    "countVectorizer__min_df": [0.0],
    "countVectorizer__stop_words": [myStopWords],
    "countVectorizer__max_features": [8000],
```

```
    "countVectorizer__ngram_range":[(1,1)],
    # "countVectorizer__tokenizer": [MyStemTokenizer()]
    # "normalizer__norm": ['l2','l1']
}

# adaboostCLF = AdaBoostClassifier(base_estimator=Bernoulli_NB(),␣
 ↪n_estimators=8, random_state=0, algorithm='SAMME')

# Making my pipeline to have these preprocessing functions
myPipe = Pipeline(
    [('countVectorizer', CountVectorizer()), ('tfidf', TfidfTransformer()),␣
 ↪('classifier', Bernoulli_NB())]
    # [('countVectorizer', CountVectorizer()), ('normalizer', Normalizer()),␣
 ↪('classifier', Bernoulli_NB(8))]
)

# Making a grid search object in order to optimize our parameters
myGridSearch = GridSearchCV(myPipe, param_grid=pipeline_parameters,␣
 ↪scoring="accuracy", n_jobs=-2, verbose=1)
# Xnew = SelectKBest(chi2, k=5000).fit_transform(Xtraining, Y)
print("Performing grid search...")
print("Pipeline: ", [name for name, _ in myPipe.steps])
t0 = time.time()
# Ygridsearch = np.reshape(Y, (Y.shape[0],))
print(Xtraining.shape)
myGridSearch.fit(Xtraining, Y)
print("Finished in %0.3fs" % (time.time() - t0))
print("\n")
```

[ ]:
```
print("Best score: %0.3f" % myGridSearch.best_score_)
print("Best parameters set:")
bestParameters = myGridSearch.best_estimator_.get_params()
# print(bestParameters)
for paramName in sorted(bestParameters.keys()):
  print("\t%s: %r" % (paramName, bestParameters[paramName]))
```

9 K-Fold Cross Validation

[ ]:
```
# K-Fold Cross Validation function
def run_K_Fold_CrossValidation(X, Y, classifier, numFolds=None):

    # # Convert
    # X = X.toarray()

    """Starting K-Fold Cross Validation"""
    # Create a dictionary to hold our fold accuracies
```

```
    fold_Accuracy_Dict = {}

    # Create sklearn's K-Fold instance
    kf = KFold(n_splits=numFolds)

    # Find out how many splitting iterations
    print(f"Number of splitting iterations: {kf.get_n_splits(Y)}\n")
    # TO UPDATE: print(f"Number of splitting iterations: {kf.
↪get_n_splits(Y)}\n")

    # Fold Iteration count
    foldCount = 0

    # Fold error sum tracker
    foldErrorSum = 0

    # Fold accuracy tracker
    foldAccuracySum = 0

    # K-Fold loop
    for training_indices, validation_indices in kf.split(X):
    # TO UPDATE: for training_indices, validation_indices in kf.split(Y):

        print(f"Starting fold {foldCount + 1}.....")
        # print("Training: ", training_indices, "Validation: ",␣
↪validation_indices)

        curFoldTrainingLabels = None
        curFoldValidationLabels = None
        curFoldTraining = None
        curFoldValidation = None
        # If we are using an sklearn classifier need to do this
        if (Y.shape == (Y.shape[0],)):
          curFoldTrainingLabels = Y[training_indices]
          curFoldValidationLabels = Y[validation_indices]
          curFoldTraining = X[training_indices,:]
          curFoldValidation = X[validation_indices,:]
        else:
          # Slicing our data to get current training and current validation sets
          curFoldTraining = X[training_indices,:]
          curFoldTrainingLabels = Y[training_indices,:]
          curFoldValidation = X[validation_indices,:]
          curFoldValidationLabels = Y[validation_indices,:]

        # print("Current fold training: \n", curFoldTraining) # if you want to␣
↪see the sliced array for training
        print(f"Current fold training shape: {curFoldTraining.shape}")
```

```
        # print("Current fold validation: \n", curFoldValidation) # if you want␣
↪to see the sliced array for validation
        print(f"Current fold validation shape: {curFoldValidation.shape}")

        # Fit our model with training
        """TODO: use our naive bayes classifier's fit function to fit our model␣
↪to our training set"""
        classifier.fit(curFoldTraining, curFoldTrainingLabels)
        # TO UPDATE: classifier.fit(Xtrain, curFoldTrainingLabels)

        # Predict the labels here
        """TODO: use our predict function to predict the labels on the␣
↪validation set"""
        curFoldPredictedLabels = classifier.predict(curFoldValidation)

        # Calculate accuracy for this fold
        """TODO: find accuracy"""
        count = 0
        for i in range(0, curFoldPredictedLabels.shape[0]):
            if curFoldValidationLabels[i] == curFoldPredictedLabels[i]:
                count += 1
        currentFoldAccuracy = (count/curFoldPredictedLabels.shape[0])*100
        print(f"Accuracy for fold {foldCount + 1}: {currentFoldAccuracy}%\n")


        # Add the accuracy of this fold to the dictionary
        fold_Accuracy_Dict[str(foldCount + 1)] = float(currentFoldAccuracy)

        # Update fold error tracker & fold accuracy tracker
        foldErrorSum = foldErrorSum + (100.0 - currentFoldAccuracy)
        foldAccuracySum = foldAccuracySum + currentFoldAccuracy

        # Update fold number
        foldCount += 1

    # Graph the fold accuracies with the dictionary
    plt.figure(figsize=(7,7))
    plt.bar(fold_Accuracy_Dict.keys(), fold_Accuracy_Dict.values(), 0.3,␣
↪color='b')
    plt.xlabel('Fold Number')
    plt.ylabel('Accuracy %')
    plt.title("K-Fold Accuracy Distribution of Current Model")
    plt.show()

    # Display this model's avg accuracy for the K-Fold
    avgAccuracyOfThisModel = float(float(foldAccuracySum)/float(foldCount))
    print(f"\nAvg accuracy for this model is: {avgAccuracyOfThisModel} %")
```

```
    # Display this model's avg error for the K-Fold
    avgErrorOfThisModel = float(float(foldErrorSum)/float(foldCount))
    print(f"Avg error for this model is: {avgErrorOfThisModel} %\n")

    # Returning the avg error, fold accuracy dictionary, and model accuracy
    return avgErrorOfThisModel, avgAccuracyOfThisModel, fold_Accuracy_Dict
```

10 Selected Pipeline & Estimator

Pipeline Multinomial Naive Bayes

[ ]:
```
# Vectorizer object for linear svc
myVectorizer4 = CountVectorizer(binary=False, max_df=1.0, min_df=0.0,␣
 ↪stop_words=myStopWords, ngram_range=(1,1), max_features=None)

# TFIDF normalizer
tfidfNormalizer4 = TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True,␣
 ↪sublinear_tf=False)

# Normalizer object for  linear svc
normalizer4 = Normalizer()

# Create our non binary training vector of word frequencies
training_vectors4 = myVectorizer4.fit_transform(Xtraining)

# Normalize with l2 so values are between 0 and 1
training_vectors4_tfidf_normalized = tfidfNormalizer4.
 ↪fit_transform(training_vectors4)
training_vectors4_normalized = normalizer4.fit_transform(training_vectors4)

# Now select our best features based on chi squared evaluation
training_vectors4_normalized_new = SelectKBest(chi2, k=7500).
 ↪fit_transform(training_vectors4_normalized, Y)
training_vectors4_tfidf_normalized_new = SelectKBest(chi2, k=8500).
 ↪fit_transform(training_vectors4_tfidf_normalized, Y)
```

Pipeline LinearSVC

[ ]:
```
# Vectorizer object for linear svc
myVectorizer3 = CountVectorizer(binary=False, max_df=1.0, min_df=0.0,␣
 ↪stop_words=myStopWords, ngram_range=(1,1), max_features=None)

# TFIDF normalizer
tfidfNormalizer3 = TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True,␣
 ↪sublinear_tf=False)
```

```
# Normalizer object for  linear svc
normalizer3 = Normalizer()

# Create our non binary training vector of word frequencies
training_vectors3 = myVectorizer3.fit_transform(Xtraining)

# Normalize with l2 so values are between 0 and 1
training_vectors3_tfidf_normalized = tfidfNormalizer3.
↪fit_transform(training_vectors3)
training_vectors3_normalized = normalizer3.fit_transform(training_vectors3)

# Now select our best features based on chi squared evaluation
training_vectors3_normalized_new = SelectKBest(chi2, k=15000).
↪fit_transform(training_vectors3_normalized, Y)
training_vectors3_tfidf_normalized_new = SelectKBest(chi2, k=15000).
↪fit_transform(training_vectors3_tfidf_normalized, Y)
```

Pipeline Bernoulli Naive Bayes

[ ]:
```
# My selected feature vectorization paraemeters and instance
# Keep in mind to have stemming when predicting on test set
# tokenizer=MyStemTokenizer()
myVectorizer1 = CountVectorizer(binary=False, max_df=1.0, min_df=0.0,␣
↪stop_words=myStopWords, ngram_range=(1,1), max_features=None)

# TFIDF normalizer
tfidfNormalizer1 = TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True,␣
↪sublinear_tf=False)

# Making my training binary vectors
training_vectors1 = myVectorizer1.fit_transform(Xtraining)

# TFIDF normalizing
training_vectors1_tfidf_normalized = tfidfNormalizer1.
↪fit_transform(training_vectors1)

# Using Sklearns function to help us select the top features
training_vectors1_tfidf_normalized_new = SelectKBest(chi2, k=8000).
↪fit_transform(training_vectors1_tfidf_normalized, Y)

# Look at feature names for both and extract and save a list of them for viewing
myFeatures1 = myVectorizer1.get_feature_names()
path = "/content/myDrive/My Drive/ECSE_551_Machine_Learning/TextClassification/
↪features.txt"
# Writing the feaure names to a text file
with open(path, mode='w') as file:
```

```
    for item in myFeatures1:
      file.write("%s\n" % item)
print("Finished writing our feature names to the text files for viewing!")
```

[ ]:
```
# myVectorizer = CountVectorizer(binary=False, max_df=1.0, min_df=0.0,␣
↪stop_words=myStopWords, ngram_range=(1,1), max_features=None)
# mytfidfTransformer = TfidfTransformer(norm='l2', use_idf=True,␣
↪smooth_idf=True, sublinear_tf=False)
# path = "/content/myDrive/My Drive/ECSE_551_Machine_Learning/
↪TextClassification/ensembleOut.csv"

# # split into trianing and val (1 fold for our kfold algorithm)
# Xtr = Xtraining[0:9264,]
# Xt = Xtraining[9265:11581,]
# Ytr = np.reshape(Y[0:9264,0], (Y[0:9264,0].shape[0],1))
# Yt = Y[9265:11581,0]

# # Trying to see
# # ensembleBagging(myVectorizer, mytfidfTransformer, 8000, 0.01, Xtraining, Y,␣
↪Xtest, 10, path)
# finalLabel = ensembleBagging(myVectorizer, mytfidfTransformer, 8000, 0.01,␣
↪Xtr, Ytr, Xt, 10, path)
# count = 0
# for i in range(finalLabel.shape[0]):
#   if (finalLabel[i] == Yt[i]):
#     count += 1

# print(f"Ensemble accuracy: {(count/finalLabel.shape[0])*100}")
```

11 Feature Selection Evaluations

Various Feature Selection Methods including MUTUAL INFORMATION for different classifiers

[ ]:
```
# My selected feature vectorization paraemeters and instance
# Keep in mind to have stemming when predicting on test set
# tokenizer=MyStemTokenizer()
myVectorizer = CountVectorizer(binary=False, max_df=1.0, min_df=0.0,␣
↪stop_words=myStopWords, ngram_range=(1,1), max_features=None)

# TFIDF normalizer
tfidfNormalizer = TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True,␣
↪sublinear_tf=False)

# Making my training binary vectors
training_vectors = myVectorizer.fit_transform(Xtraining)
```

```
# TFIDF normalizing
training_vectors_tfidf_normalized = tfidfNormalizer.
↪fit_transform(training_vectors)

# Using Sklearns feature selection functions to help us select the top features
featureSelectionMethods = [chi2, f_classif, mutual_info_classif, f_regression,␣
↪mutual_info_regression]
accuracyList = []
for selectionMethod in featureSelectionMethods:
  print(f"Starting: {str(selectionMethod)}.......")
  training_vectors_tfidf_normalized_new =␣
↪SelectKBest(score_func=selectionMethod, k=8000).
↪fit_transform(training_vectors_tfidf_normalized, np.ravel(Y))
  _,avgAccuracy,_ =␣
↪run_K_Fold_CrossValidation(training_vectors_tfidf_normalized_new, Y,␣
↪Bernoulli_NB(alpha=0.01), numFolds=5)
  accuracyList.append(avgAccuracy)
```

[ ]:
```
featureSelectionNames = ["chi2", "f_classif", "mutual_info_classif",␣
↪"f_regression", "mutual_info_regression"]
plt.figure(figsize=(8,8))
plt.plot(featureSelectionNames, accuracyList)
plt.title("Naive Bayes Feature Selection Method Tunning with SelectKBest")
plt.xlabel("Feature Selection Method")
plt.ylabel("Average Model Accuracy")
plt.show()
```

Bernouilli Naive Bayes & Best Pipeline & Various Feature Selection Methods & SelectPercentile

[ ]:
```
# My selected feature vectorization paraemeters and instance
# Keep in mind to have stemming when predicting on test set
# tokenizer=MyStemTokenizer()
myVectorizer = CountVectorizer(binary=False, max_df=1.0, min_df=0.0,␣
↪stop_words=myStopWords, ngram_range=(1,1), max_features=None)

# TFIDF normalizer
tfidfNormalizer = TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True,␣
↪sublinear_tf=False)

# Making my training binary vectors
training_vectors = myVectorizer.fit_transform(Xtraining)

# TFIDF normalizing
training_vectors_tfidf_normalized = tfidfNormalizer.
↪fit_transform(training_vectors)

# Using Sklearns feature selection functions to help us select the top features
```

```
featureSelectionMethods = [chi2, f_classif, mutual_info_classif, f_regression]
accuracyList = []
for selectionMethod in featureSelectionMethods:
  print(f"Starting: {str(selectionMethod)}.......")
  training_vectors_tfidf_normalized_new =␣
 ↪SelectPercentile(score_func=selectionMethod, percentile=25).
 ↪fit_transform(training_vectors_tfidf_normalized, np.ravel(Y))
  _,avgAccuracy,_ =␣
 ↪run_K_Fold_CrossValidation(training_vectors_tfidf_normalized_new, Y,␣
 ↪Bernoulli_NB(alpha=0.01), numFolds=5)
  accuracyList.append(avgAccuracy)
```

[ ]:
```
featureSelectionNames = ["chi2", "f_classif", "mutual_info_classif",␣
 ↪"f_regression"]
# Plotting feature selection methods vs accuracy
plt.figure(figsize=(8,8))
plt.plot(featureSelectionNames, accuracyList)
plt.title("Naive Bayes Feature Selection Method Tunning with SelectPercentile")
plt.xlabel("Feature Selection Method")
plt.ylabel("Average Model Accuracy")
plt.show()
```

12 SKLearn Ensemble Bagging

[ ]:
```
# Vectorizer object for linear svc
myVectorizer4 = CountVectorizer(binary=False, max_df=1.0, min_df=0.0,␣
 ↪stop_words=myStopWords, ngram_range=(1,1), max_features=None)

# TFIDF normalizer
tfidfNormalizer4 = TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True,␣
 ↪sublinear_tf=False)

# Normalizer object for  linear svc
normalizer4 = Normalizer()

# Create our non binary training vector of word frequencies
training_vectors4 = myVectorizer4.fit_transform(Xtraining)

# Normalize with l2 so values are between 0 and 1
training_vectors4_tfidf_normalized = tfidfNormalizer4.
 ↪fit_transform(training_vectors4)
training_vectors4_normalized = normalizer4.fit_transform(training_vectors4)

# Now select our best features based on chi squared evaluation
```

```
training_vectors4_normalized_new = SelectKBest(chi2, k=7500).
 ↪fit_transform(training_vectors4_normalized, Y)
training_vectors4_tfidf_normalized_new = SelectKBest(chi2, k=8500).
 ↪fit_transform(training_vectors4_tfidf_normalized, Y)

# Defining ensemble classifier from sklearn
clf2 = BaggingClassifier(base_estimator=MultinomialNB(alpha=0.02,␣
 ↪fit_prior=False), n_estimators=20, random_state=0)

# Run k fold cross validation
run_K_Fold_CrossValidation(training_vectors4_tfidf_normalized_new, np.ravel(Y),␣
 ↪clf2, numFolds=5)
```

[ ]:
```
# My selected feature vectorization paraemeters and instance
# Keep in mind to have stemming when predicting on test set
# tokenizer=MyStemTokenizer()
myVectorizer1 = CountVectorizer(binary=False, max_df=1.0, min_df=0.0,␣
 ↪stop_words=myStopWords, ngram_range=(1,1), max_features=None)

# TFIDF normalizer
tfidfNormalizer1 = TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True,␣
 ↪sublinear_tf=False)

# Making my training binary vectors
training_vectors1 = myVectorizer1.fit_transform(Xtraining)

# TFIDF normalizing
training_vectors1_tfidf_normalized = tfidfNormalizer1.
 ↪fit_transform(training_vectors1)

# Using Sklearns function to help us select the top features
training_vectors1_tfidf_normalized_new = SelectKBest(chi2, k=8000).
 ↪fit_transform(training_vectors1_tfidf_normalized, Y)

# Defining ensemble classifier from sklearn
clf1 = BaggingClassifier(base_estimator=Bernoulli_NB(alpha=0.01),␣
 ↪n_estimators=20, random_state=0)

# Run k fold cross validation
run_K_Fold_CrossValidation(training_vectors1_tfidf_normalized_new, np.ravel(Y),␣
 ↪clf1, numFolds=5)
```

Classifier Instances

[ ]:
```
# Instantiating our classifier instances
nb1 = Bernoulli_NB(0.01)
nb2 = Bernoulli_NB(0.05)
```

```
linearsvc = LinearSVC(max_iter=1000000)
mnb = MultinomialNB(alpha=0.02, fit_prior=False)
```

1) K-Fold Cross Validation: SKLearn Feature Selection & Max_features = None & Bernoulli NB & TFIDF Transfromer normalized & Binary=False

[ ]:
```
# # Run our K_fold to see what our training accuracy is like
run_K_Fold_CrossValidation(training_vectors1_tfidf_normalized_new, Y, nb1,␣
↪numFolds=5)
```

2) K-Fold Cross Validation: SKLearn Feature Selection & Max_features = None & LinearSVC & TFIDF Transfromer normalized & Binary=False

[ ]:
```
Ysvc = np.reshape(Y, (Y.shape[0],))
run_K_Fold_CrossValidation(training_vectors3_tfidf_normalized_new, Ysvc,␣
↪linearsvc, numFolds=5)
```

3) K-Fold Cross Validation: SKLearn Feature Selection & Max_features = None & Multinomial Naive Bayes & TFIDF Transfromer normalized & Binary=False

[ ]:
```
Ymnb = np.reshape(Y, (Y.shape[0],))
run_K_Fold_CrossValidation(training_vectors4_tfidf_normalized_new, Ymnb, mnb,␣
↪numFolds=5)
```

4) K-Fold Cross Validation: SKLearn Feature Selection & Max_features = None & Multinomial Naive Bayes with Normalizer() & Binary=False

[ ]:
```
Ymnb = np.reshape(Y, (Y.shape[0],))
run_K_Fold_CrossValidation(training_vectors4_normalized_new, Ymnb, mnb,␣
↪numFolds=5)
```

13 Model Selection & Evaluation on Test set

[ ]:
```
# My model selection and eval on test function
def selectSKLearnModelEvalOnTest(myVectorizer, mytfidfTransformer, classifier,␣
↪numFeatures, Xtest, Xtraining, Y, path):

  # count vectorization
  skLearnFeatureSelector = SelectKBest(chi2, k=numFeatures)
  vectors_train = myVectorizer.fit_transform(Xtraining)
  vectors_test = myVectorizer.transform(Xtest)

  # tfidf normalization
  vectors_train_tfidf_normalized = mytfidfTransformer.
↪fit_transform(vectors_train)
  vectors_test_tfidf_normalized = mytfidfTransformer.transform(vectors_test)
```

```
    # selecting the top features using chi squared evaluation
    vectors_train_tfidf_normalized_new = skLearnFeatureSelector.
↪fit_transform(vectors_train_tfidf_normalized, Y)
    vectors_test_tfidf_normalized_new = skLearnFeatureSelector.
↪transform(vectors_test_tfidf_normalized)

    # Evaluating using our classifier
    classifier.fit(vectors_train_tfidf_normalized_new, Y)
    predictedLabels = classifier.predict(vectors_test_tfidf_normalized_new)

    # Convert number labels back into string labels
    predictedLabelsStringFormat = []
    for labelNum in predictedLabels:
      # print(labelNum)
      predictedLabelsStringFormat.append(numberToLabel(labelNum))

    print(f"Length of predicted list: {len(predictedLabelsStringFormat)}")
    # print(predictedLabelsStringFormat)

    # Convert from list to pandas data frame
    dfPredicted = pd.DataFrame(predictedLabelsStringFormat, columns=['subreddit'])
    # print(dfPredicted)
    dfPredicted.to_csv(path)
```

[ ]:

```
# My model selection and eval on test function
def selectModelEvalOnTest(myVectorizer,mytfidfTransformer, classifier,␣
↪numFeatures, Xtest, Xtraining, Y, path):

    # count vectorization
    skLearnFeatureSelector = SelectKBest(chi2, k=numFeatures)
    vectors_train = myVectorizer.fit_transform(Xtraining)
    vectors_test = myVectorizer.transform(Xtest)

    # tfidf normalization
    vectors_train_tfidf_normalized = mytfidfTransformer.
↪fit_transform(vectors_train)
    vectors_test_tfidf_normalized = mytfidfTransformer.transform(vectors_test)

    # selecting the top features using chi squared evaluation
    vectors_train_tfidf_normalized_new = skLearnFeatureSelector.
↪fit_transform(vectors_train_tfidf_normalized, Y)
    vectors_test_tfidf_normalized_new = skLearnFeatureSelector.
↪transform(vectors_test_tfidf_normalized)

    # Evaluating using our classifier
    classifier.fit(vectors_train_tfidf_normalized_new, Y)
```

```
predictedLabels = classifier.predict(vectors_test_tfidf_normalized_new)

# Convert number labels back into string labels
predictedLabelsStringFormat = []
for labelNum in predictedLabels:
  # print(labelNum)
  predictedLabelsStringFormat.append(numberToLabel(labelNum[0]))

print(f"Length of predicted list: {len(predictedLabelsStringFormat)}")
# print(predictedLabelsStringFormat)

# Convert from list to pandas data frame
dfPredicted = pd.DataFrame(predictedLabelsStringFormat, columns=['subreddit'])
# print(dfPredicted)
dfPredicted.to_csv(path)
```

Models Selected to try on test set

[ ]:
```
# Final vectorizer pipeline set up
finalVectorizer = CountVectorizer(binary=False, max_df=1.0, min_df=0.0,␣
 ↪stop_words=myStopWords, ngram_range=(1,1), max_features=None)
# Final TFIDF normalizer object
tfidfNormalizerFinal = TfidfTransformer(norm='l2', use_idf=True,␣
 ↪smooth_idf=True, sublinear_tf=False)
# Final nb with tuned alpha
nbFinal = Bernoulli_NB(alpha=0.01)
# File path output csv
csvPath = "/content/myDrive/My Drive/ECSE_551_Machine_Learning/
 ↪TextClassification/output.csv"
selectModelEvalOnTest(finalVectorizer, tfidfNormalizerFinal, nbFinal, 8000,␣
 ↪Xtest, Xtraining, Y, csvPath)
```

[ ]:
```
# Final vectorizer pipeline set up
finalVectorizerMNB = CountVectorizer(binary=False, max_df=1.0, min_df=0.0,␣
 ↪stop_words=myStopWords, ngram_range=(1,1), max_features=None)
# Final mnb with tuned alpha
mnbFinal = MultinomialNB(alpha=0.02)
# File path output csv
csvPath = "/content/myDrive/My Drive/ECSE_551_Machine_Learning/
 ↪TextClassification/output.csv"
# Final TFIDF normalizer object
tfidfNormalizerFinal = TfidfTransformer(norm='l2', use_idf=True,␣
 ↪smooth_idf=True, sublinear_tf=False)
# Final model evaluation using a sklearn classifier
Ymnbfinal = np.reshape(Y, (Y.shape[0],))
selectSKLearnModelEvalOnTest(finalVectorizerMNB, tfidfNormalizerFinal,␣
 ↪mnbFinal, 7500, Xtest, Xtraining, Ymnbfinal, csvPath)
```

```
[ ]: # Final vectorizer pipeline set up
finalVectorizerMNB = CountVectorizer(binary=False, max_df=1.0, min_df=0.0,␣
 ↪stop_words=myStopWords, ngram_range=(1,1), max_features=None)
# Final mnb with tuned alpha
clf = BaggingClassifier(base_estimator=Bernoulli_NB(alpha=0.01),␣
 ↪n_estimators=20, random_state=0)
# File path output csv
csvPath = "/content/myDrive/My Drive/ECSE_551_Machine_Learning/
 ↪TextClassification/output.csv"
# Final TFIDF normalizer object
tfidfNormalizerFinal = TfidfTransformer(norm='l2', use_idf=True,␣
 ↪smooth_idf=True, sublinear_tf=False)
# Final model evaluation using a sklearn classifier
Ymnbfinal = np.reshape(Y, (Y.shape[0],))
selectSKLearnModelEvalOnTest(finalVectorizerMNB, tfidfNormalizerFinal, clf,␣
 ↪8000, Xtest, Xtraining, Ymnbfinal, csvPath)
```

14 Experimentation

Experiment Classifiers

```
[ ]: nbExp = Bernoulli_NB(0.01)
mbExp = MultinomialNB(alpha=0.02, fit_prior=False)
```

1) Stemming Experiment using Bernoulli NB

```
[ ]: # Vectorizer object for linear svc
myVectorizer4 = CountVectorizer(binary=True, max_df=1.0, min_df=0.0,␣
 ↪stop_words=myStopWords, ngram_range=(1,1), max_features=None,␣
 ↪tokenizer=MyLemmaTokenizer())

# # TFIDF normalizer
# tfidfNormalizer4 = TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True,␣
 ↪sublinear_tf=False)

# Normalizer object for  linear svc
normalizer4 = Normalizer()

# Create our non binary training vector of word frequencies
training_vectors4 = myVectorizer4.fit_transform(Xtraining)

# # Normalize with l2 so values are between 0 and 1
# training_vectors4_tfidf_normalized = tfidfNormalizer4.
 ↪fit_transform(training_vectors4)
# training_vectors4_normalized = normalizer4.fit_transform(training_vectors4)
```

```
# # Now select our best features based on chi squared evaluation
# training_vectors4_normalized_new = SelectKBest(chi2, k=8000).
 ↪fit_transform(training_vectors4_normalized, Y)
# training_vectors4_tfidf_normalized_new = SelectKBest(chi2, k=8000).
 ↪fit_transform(training_vectors4_tfidf_normalized, Y)

# run kfold
run_K_Fold_CrossValidation(training_vectors4, Y, nbExp, numFolds=5)
```

2) Lemmatization Experiment with Bernoulli NB

[ ]:
```
# Vectorizer object for linear svc
myVectorizer4 = CountVectorizer(binary=True, max_df=1.0, min_df=0.0,␣
 ↪stop_words=myStopWords, ngram_range=(1,1), max_features=None,␣
 ↪tokenizer=MyLemmaTokenizer())

# # TFIDF normalizer
# tfidfNormalizer4 = TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True,␣
 ↪sublinear_tf=False)

# Normalizer object for  linear svc
normalizer4 = Normalizer()

# Create our non binary training vector of word frequencies
training_vectors4 = myVectorizer4.fit_transform(Xtraining)

# # Normalize with l2 so values are between 0 and 1
# training_vectors4_tfidf_normalized = tfidfNormalizer4.
 ↪fit_transform(training_vectors4)
# training_vectors4_normalized = normalizer4.fit_transform(training_vectors4)

# # Now select our best features based on chi squared evaluation
# training_vectors4_normalized_new = SelectKBest(chi2, k=8000).
 ↪fit_transform(training_vectors4_normalized, Y)
# training_vectors4_tfidf_normalized_new = SelectKBest(chi2, k=8000).
 ↪fit_transform(training_vectors4_tfidf_normalized, Y)

# run kfold
run_K_Fold_CrossValidation(training_vectors4, Y, nbExp, numFolds=5)
```

3) Raw data with no tokenizer using Bernoulli NB

[ ]:
```
# Vectorizer object for linear svc
myVectorizer4 = CountVectorizer(binary=True, max_df=1.0, min_df=0.0,␣
 ↪stop_words=myStopWords, ngram_range=(1,1), max_features=None)

# # TFIDF normalizer
```

```
# tfidfNormalizer4 = TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True,␣
↪sublinear_tf=False)

# Normalizer object for  linear svc
normalizer4 = Normalizer()

# Create our non binary training vector of word frequencies
training_vectors4 = myVectorizer4.fit_transform(Xtraining)

# # Normalize with l2 so values are between 0 and 1
# training_vectors4_tfidf_normalized = tfidfNormalizer4.
↪fit_transform(training_vectors4)
# training_vectors4_normalized = normalizer4.fit_transform(training_vectors4)

# # Now select our best features based on chi squared evaluation
# training_vectors4_normalized_new = SelectKBest(chi2, k=8000).
↪fit_transform(training_vectors4_normalized, Y)
# training_vectors4_tfidf_normalized_new = SelectKBest(chi2, k=8000).
↪fit_transform(training_vectors4_tfidf_normalized, Y)

# run kfold
run_K_Fold_CrossValidation(training_vectors4, Y, nbExp, numFolds=5)
```

4) Raw data with SKLearn feature selection (SelectKBest - Chi Squared Test)

[ ]:
```
# Vectorizer object for linear svc
myVectorizer4 = CountVectorizer(binary=True, max_df=1.0, min_df=0.0,␣
↪stop_words=myStopWords, ngram_range=(1,1), max_features=None)

# # TFIDF normalizer
# tfidfNormalizer4 = TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True,␣
↪sublinear_tf=False)

# Normalizer object for  linear svc
normalizer4 = Normalizer()

# Create our non binary training vector of word frequencies
training_vectors4 = myVectorizer4.fit_transform(Xtraining)

# # Normalize with l2 so values are between 0 and 1
# training_vectors4_tfidf_normalized = tfidfNormalizer4.
↪fit_transform(training_vectors4)
# training_vectors4_normalized = normalizer4.fit_transform(training_vectors4)

# # Now select our best features based on chi squared evaluation
# training_vectors4_normalized_new = SelectKBest(chi2, k=8000).
↪fit_transform(training_vectors4_normalized, Y)
```

```
# training_vectors4_tfidf_normalized_new = SelectKBest(chi2, k=8000).
 ↪fit_transform(training_vectors4_tfidf_normalized, Y)
training_vectors4_feature_selected = SelectKBest(chi2, k=8000).
 ↪fit_transform(training_vectors4, Y)

# run kfold
run_K_Fold_CrossValidation(training_vectors4_feature_selected, Y, nbExp,␣
 ↪numFolds=5)
```

5) Raw data and TFIDF normalization with SKLearn feature selection with different classifiers

[ ]:
```
# Vectorizer object for linear svc
myVectorizer4 = CountVectorizer(binary=True, max_df=1.0, min_df=0.0,␣
 ↪stop_words=myStopWords, ngram_range=(1,1), max_features=None)

# TFIDF normalizer
tfidfNormalizer4 = TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True,␣
 ↪sublinear_tf=False)

# Normalizer object for  linear svc
normalizer4 = Normalizer()

# Create our non binary training vector of word frequencies
training_vectors4 = myVectorizer4.fit_transform(Xtraining)

# Normalize with l2 so values are between 0 and 1
training_vectors4_tfidf_normalized = tfidfNormalizer4.
 ↪fit_transform(training_vectors4)
training_vectors4_normalized = normalizer4.fit_transform(training_vectors4)

# Now select our best features based on chi squared evaluation
training_vectors4_normalized_new = SelectKBest(chi2, k=8000).
 ↪fit_transform(training_vectors4_normalized, Y)
training_vectors4_tfidf_normalized_new = SelectKBest(chi2, k=8000).
 ↪fit_transform(training_vectors4_tfidf_normalized, Y)
# training_vectors4_feature_selected = SelectKBest(chi2, k=8000).
 ↪fit_transform(training_vectors4, Y)

# run kfold
run_K_Fold_CrossValidation(training_vectors4_normalized_new, Y, nbExp,␣
 ↪numFolds=5)
```

6) Raw data and TFIDF normalization with SKLearn feature selection with different classifiers and Stemming

[ ]:
```
# Vectorizer object for linear svc
```

```
myVectorizer4 = CountVectorizer(binary=True, max_df=1.0, min_df=0.0,␣
↪stop_words=myStopWords, ngram_range=(1,1), max_features=None,␣
↪tokenizer=MyStemTokenizer())

# TFIDF normalizer
tfidfNormalizer4 = TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True,␣
↪sublinear_tf=False)

# Normalizer object for  linear svc
normalizer4 = Normalizer()

# Create our non binary training vector of word frequencies
training_vectors4 = myVectorizer4.fit_transform(Xtraining)

# Normalize with l2 so values are between 0 and 1
training_vectors4_tfidf_normalized = tfidfNormalizer4.
↪fit_transform(training_vectors4)
training_vectors4_normalized = normalizer4.fit_transform(training_vectors4)

# Now select our best features based on chi squared evaluation
training_vectors4_normalized_new = SelectKBest(chi2, k=8000).
↪fit_transform(training_vectors4_normalized, Y)
training_vectors4_tfidf_normalized_new = SelectKBest(chi2, k=8000).
↪fit_transform(training_vectors4_tfidf_normalized, Y)
# training_vectors4_feature_selected = SelectKBest(chi2, k=8000).
↪fit_transform(training_vectors4, Y)

# run kfold
run_K_Fold_CrossValidation(training_vectors4_normalized_new, Y, nbExp,␣
↪numFolds=5)
```

7) Bernouilli NB with best pipeline

[ ]:
```
# Vectorizer object for linear svc
myVectorizer4 = CountVectorizer(binary=True, max_df=1.0, min_df=0.0,␣
↪stop_words=myStopWords, ngram_range=(1,1), max_features=None)

# TFIDF normalizer
tfidfNormalizer4 = TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True,␣
↪sublinear_tf=False)

# Normalizer object for  linear svc
normalizer4 = Normalizer()

# Create our non binary training vector of word frequencies
training_vectors4 = myVectorizer4.fit_transform(Xtraining)
```

```
# Normalize with l2 so values are between 0 and 1
training_vectors4_tfidf_normalized = tfidfNormalizer4.
↪fit_transform(training_vectors4)
training_vectors4_normalized = normalizer4.fit_transform(training_vectors4)

# Now select our best features based on chi squared evaluation
training_vectors4_normalized_new = SelectKBest(chi2, k=8000).
↪fit_transform(training_vectors4_normalized, Y)
training_vectors4_tfidf_normalized_new = SelectKBest(chi2, k=8000).
↪fit_transform(training_vectors4_tfidf_normalized, Y)
# training_vectors4_feature_selected = SelectKBest(chi2, k=8000).
↪fit_transform(training_vectors4, Y)

# # run kfold
# run_K_Fold_CrossValidation(training_vectors4_normalized_new, Y, nbExp,␣
↪numFolds=5)
```

Feature Selection for Bernouilli NB using Chi2

[ ]:
```
# Plotting feature size vs avgAccuracy
featureList = [4000,5000, 6000, 7000, 8000, 9000, 10000]
accuracies = []
for f in featureList:
  training_vectors4_normalized_new = SelectKBest(chi2, k=f).
↪fit_transform(training_vectors4_normalized, Y)
  _,avgAccuracy,_ =␣
↪run_K_Fold_CrossValidation(training_vectors4_normalized_new, Y,␣
↪Bernoulli_NB(alpha=0.01), numFolds=5)
  accuracies.append(avgAccuracy)
plt.figure(figsize=(8,8))
plt.plot(featureList, accuracies)
plt.title("Bernouilli Naive Bayes Feature Size Tunning")
plt.xlabel("Feature Size")
plt.ylabel("Average Model Accuracy")
plt.show()
```

Hyper-parameter Tunning for Bernouilli NB

[ ]:
```
# Plotting alphas vs avg accuracy of the model
alpha = [0.1,0.08,0.06,0.04,0.02,0.01]
accuracies = []
for a in alpha:
  _,avgAccuracy,_ =␣
↪run_K_Fold_CrossValidation(training_vectors4_normalized_new, Y,␣
↪Bernoulli_NB(alpha=a), numFolds=5)
  accuracies.append(avgAccuracy)
plt.figure(figsize=(8,8))
```

```
plt.plot(alpha, accuracies)
plt.title("Bernouilli Naive Bayes Alpha Tunning")
plt.xlabel("Alpha")
plt.ylabel("Average Model Accuracy")
plt.show()
```

8) Bernouilli NB with best pipeline binary=False

```
[ ]: # Vectorizer object for linear svc
myVectorizer4 = CountVectorizer(binary=False, max_df=1.0, min_df=0.0,␣
 ↪stop_words=myStopWords, ngram_range=(1,1), max_features=None)

# TFIDF normalizer
tfidfNormalizer4 = TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True,␣
 ↪sublinear_tf=False)

# Normalizer object for  linear svc
normalizer4 = Normalizer()

# Create our non binary training vector of word frequencies
training_vectors4 = myVectorizer4.fit_transform(Xtraining)

# Normalize with l2 so values are between 0 and 1
training_vectors4_tfidf_normalized = tfidfNormalizer4.
 ↪fit_transform(training_vectors4)
training_vectors4_normalized = normalizer4.fit_transform(training_vectors4)

# Now select our best features based on chi squared evaluation
training_vectors4_normalized_new = SelectKBest(chi2, k=8000).
 ↪fit_transform(training_vectors4_normalized, Y)
training_vectors4_tfidf_normalized_new = SelectKBest(chi2, k=8000).
 ↪fit_transform(training_vectors4_tfidf_normalized, Y)
# training_vectors4_feature_selected = SelectKBest(chi2, k=8000).
 ↪fit_transform(training_vectors4, Y)

# run kfold
run_K_Fold_CrossValidation(training_vectors4_tfidf_normalized_new, Y, nbExp,␣
 ↪numFolds=5)
```

9) Multinomial NB with best pipeline

```
[ ]: # Vectorizer object for linear svc
myVectorizer4 = CountVectorizer(binary=False, max_df=1.0, min_df=0.0,␣
 ↪stop_words=myStopWords, ngram_range=(1,1), max_features=None)

# TFIDF normalizer
```

```
tfidfNormalizer4 = TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True,␣
↪sublinear_tf=False)

# Normalizer object for  linear svc
normalizer4 = Normalizer()

# Create our non binary training vector of word frequencies
training_vectors4 = myVectorizer4.fit_transform(Xtraining)

# Normalize with l2 so values are between 0 and 1
training_vectors4_tfidf_normalized = tfidfNormalizer4.
↪fit_transform(training_vectors4)
training_vectors4_normalized = normalizer4.fit_transform(training_vectors4)

# Now select our best features based on chi squared evaluation
training_vectors4_normalized_new = SelectKBest(chi2, k=7500).
↪fit_transform(training_vectors4_normalized, Y)
training_vectors4_tfidf_normalized_new = SelectKBest(chi2, k=8500).
↪fit_transform(training_vectors4_tfidf_normalized, Y)

run_K_Fold_CrossValidation(training_vectors4_normalized_new, Y, mbExp,␣
↪numFolds=5)
```

Feature Selection using Chi2 for Multinomial and SVC

[ ]:
```
# Plotting feature size vs avgAccuracy
featureList = [4000,5000, 6000, 7000, 8500, 9000, 10000]
accuracies = []
for f in featureList:
  training_vectors4_tfidf_normalized_new = SelectKBest(chi2, k=f).
↪fit_transform(training_vectors4_tfidf_normalized, Y)
  _,avgAccuracy,_ =␣
↪run_K_Fold_CrossValidation(training_vectors4_tfidf_normalized_new, Y,␣
↪MultinomialNB(alpha=0.04), numFolds=5)
  accuracies.append(avgAccuracy)
plt.figure(figsize=(8,8))
plt.plot(featureList, accuracies)
plt.title("Multinomial Naive Bayes Feature Size Tunning")
plt.xlabel("Feature Size")
plt.ylabel("Average Model Accuracy")
plt.show()
```

Hyper-parameter Tunning for Multinomial NB

[ ]:
```
# Plotting alphas vs avg accuracy of the model
alpha = [0.1,0.08,0.06,0.04,0.02,0.01]
accuracies = []
```

```
for a in alpha:
  _,avgAccuracy,_ =␣
↪run_K_Fold_CrossValidation(training_vectors4_tfidf_normalized_new, Y,␣
↪MultinomialNB(alpha=a), numFolds=5)
  accuracies.append(avgAccuracy)
plt.figure(figsize=(8,8))
plt.plot(alpha, accuracies)
plt.title("Multinomial Naive Bayes Alpha Tunning")
plt.xlabel("Alpha")
plt.ylabel("Average Model Accuracy")
plt.show()
```

10) LinearSVC with best pipeline

[ ]:
```
# Vectorizer object for linear svc
myVectorizer3 = CountVectorizer(binary=False, max_df=1.0, min_df=0.0,␣
↪stop_words=myStopWords, ngram_range=(1,1), max_features=None)

# TFIDF normalizer
tfidfNormalizer3 = TfidfTransformer(norm='l2', use_idf=True, smooth_idf=True,␣
↪sublinear_tf=False)

# Normalizer object for  linear svc
normalizer3 = Normalizer()

# Create our non binary training vector of word frequencies
training_vectors3 = myVectorizer3.fit_transform(Xtraining)

# Normalize with l2 so values are between 0 and 1
training_vectors3_tfidf_normalized = tfidfNormalizer3.
↪fit_transform(training_vectors3)
training_vectors3_normalized = normalizer3.fit_transform(training_vectors3)

# Now select our best features based on chi squared evaluation
training_vectors3_normalized_new = SelectKBest(chi2, k=15000).
↪fit_transform(training_vectors3_normalized, Y)
training_vectors3_tfidf_normalized_new = SelectKBest(chi2, k=15000).
↪fit_transform(training_vectors3_tfidf_normalized, Y)

Ysvc = np.reshape(Y, (Y.shape[0],))
run_K_Fold_CrossValidation(training_vectors3_tfidf_normalized_new, Ysvc,␣
↪linearsvc, numFolds=5)
```