Table of Contents

[Sentiment Analysis 1](#_Toc116626846)

[Tokenize 1](#_Toc116626847)

[Stop Words 3](#_Toc116626848)

[Stemming 6](#_Toc116626849)

[Vectorize (and NGrams) 7](#_Toc116626850)

[Modeling 9](#_Toc116626851)

[Using RMSE to Evaluate a Classification Model 13](#_Toc116626852)

[Hot Words with N-Grams 14](#_Toc116626853)

## Sentiment Analysis

Sentiment analysis attempts to understand and summarize the feeling and intent behind large volumes of text.

Like all data modeling algorithms, measuring sentiment is subject to many biases at different stages of the process. Compared to other types of predictions, sentiment analysis faces unique challenges of interpretability due to sarcasm, different types of negation and word ambiguity. Expecting 100% predictive accuracy when working with sentiment analysis algorithms and a wide rating system is not realistic. However, with careful inspection you can develop summaries of sentiment with reasonable ranges of confidence.

Like any predictive algorithm, sentiment measures are subject to bias at each step in the process. Sentiment relies on the bias of a reviewer. Some reviewers may not believe in offering the highest or lowest score even when they are extremely thrilled or extremely disappointed with the item being reviewed. Other reviewers may default to an extreme rating even though they actually feel more neutral. If a data analyst is manually scoring text sentiment for developing models then the scores are subject to the analyst’s bias. Steps to prepare the data can also omit or distort sentiment and content which may adversely affect the outcome.

This document explains common techniques that are used to prepare text and summarize sentiment in a manner that is objective. The techniques to be discussed include:

* Tokenization
* N-Grams
* Stemming
* Removing Stop Words
* Word ranking

You do not need to memorize the code for implementing each of these techniques but you should be able to explain how these algorithms work.

### Tokenize

Tokenization splits each sentence into an array of words while also removing punctuation.

Example 1: Tokenizing

This example shows how to tokenize sentences by converting two sentences into arrays of words.

|  |
| --- |
| ﻿import nltk  from nltk.tokenize import RegexpTokenizer  sentence1 = "Despite its fresh perspective, Banks's Charlie's Angels update " \  + "fails to capture the energy or style that made it the beloved phenomenon it is."  sentence2 = "This 2019 Charlie's Angels is stupefyingly entertaining and " \  + "hilarious. It is a stylish alternative to the current destructive blockbusters."  sentences = [sentence1, sentence2]  # -------------------------------------------------------------  # Create lower case array of words with no punctuation.  # -------------------------------------------------------------  def createTokenizedArray(sentences):  # Initialize tokenizer and empty array to store modified sentences.  tokenizer = RegexpTokenizer(r'\w+')  tokenizedArray = []  for i in range(0, len(sentences)):  # Convert sentence to lower case.  sentence = sentences[i].lower()  # Split sentence into array of words with no punctuation.  words = tokenizer.tokenize(sentence)  # Append word array to list.  tokenizedArray.append(words)  print(tokenizedArray)  return tokenizedArray # send modified contents back to calling function.  tokenizedList = createTokenizedArray(sentences) |

After passing the array of sentences to our function, the data is returned as an array of arrays which contain words for each sentence. Notice how the punctuation is absent and how all letters are lower case.

|  |
| --- |
| ﻿﻿[[‘despite’, ‘its’, ‘fresh’, ‘perspective’, ‘banks’, ‘s’, ‘charlie’, ‘s’, ‘angels’, ‘update’, ‘fails’, ‘to’, ‘capture’, ‘the’, ‘energy’, ‘or’, ‘style’, ‘that’, ‘made’, ‘it’, ‘the’, ‘beloved’, ‘phenomenon’, ‘it’, ‘is’], [‘this’, ‘2019’, ‘charlie’, ‘s’, ‘angels’, ‘is’, ‘stupefyingly’, ‘entertaining’, ‘and’, ‘hilarious’, ‘it’, ‘is’, ‘a’, ‘stylish’, ‘alternative’, ‘to’, ‘the’, ‘current’, ‘destructive’, ‘blockbusters’]] |

Exercise 1 (1 mark)

Call *createTokenizedArray()* function and pass the following sentence as a parameter:

|  |
| --- |
| reviewSentence = \  "Parents need to know that this classic 1908 children's novel by L.M. " \  + "Montgomery remains a perennial favorite thanks to its memorable heroine: " \  + "irrepressible red-headed orphan Anne Shirley. Anne's adventures are full of " \  + "amusing (and occasionally mildly dangerous) scrapes, but she's quick to learn " \  + "from her mistakes and usually has only the best of intentions. Although Anne " \  + "gets her best friend drunk in one episode (it's an honest mistake), there's very " \  + "little here that's at all iffy for kids -- though younger readers might get a " \  + "bit bogged down in the many descriptions of Anne's Prince Edward Island, Canada, " \  + "home. A sad death may hit some kids hard, but the book's messages about the " \  + "importance of love, friendship, family, and ambition are worth it." |

Show the output here:

|  |
| --- |
|  |

### Stop Words

Stop words are words that do not have any importance in search queries. A stop word is a commonly used word (such as “the”, “a”, “an”, “in”) that a search engine has been programmed to ignore, both when indexing entries for searching and when retrieving them as the result of a search query.

Example 2: Stop Words

This example shows how to remove common insignificant words from a sentence. Extra code has been added to also remove digits and words that are only one letter. To run this example, add this code after the code for Example 1. Remember, you do not need to memorize this code but you should try to be comfortable explaining how it works.

|  |
| --- |
| from nltk.corpus import stopwords  # To get stop words.  nltk.download('stopwords')  #-------------------------------------------------------------  # Create array of words with no punctuation or stop words.  #-------------------------------------------------------------  def removeStopWords(tokenList):  stopWords = set(stopwords.words('english'))  shorterSentences = [] # Declare empty array of sentences.    for sentence in tokenList:  shorterSentence = [] # Declare empty array of words in single sentence.  for word in sentence:  if word not in stopWords:    # Remove leading and trailing spaces.  word = word.strip()    # Ignore single character words and digits.  if(len(word)>1 and word.isdigit()==False):  # Add remaining words to list.  shorterSentence.append(word)  shorterSentences.append(shorterSentence)  return shorterSentences    sentenceArrays = removeStopWords(tokenizedList)  print(sentenceArrays) |

The output after removing stopwords from our tokenized sentences shows an array of arrays of key words:

|  |
| --- |
| ﻿ ﻿[['despite', 'fresh', 'perspective', 'banks', 'charlie', 'angels', 'update', 'fails', 'capture', 'energy', 'style', 'made', 'beloved', 'phenomenon'], ['charlie', 'angels', 'stupefyingly', 'entertaining', 'hilarious', 'stylish', 'alternative', 'current', 'destructive', 'blockbusters']] |

Exercise 2 (1 mark)

Run the tokenized version of the sentence in the solution for Exercise 1 through the removeStopWords() function. What does the output look like?

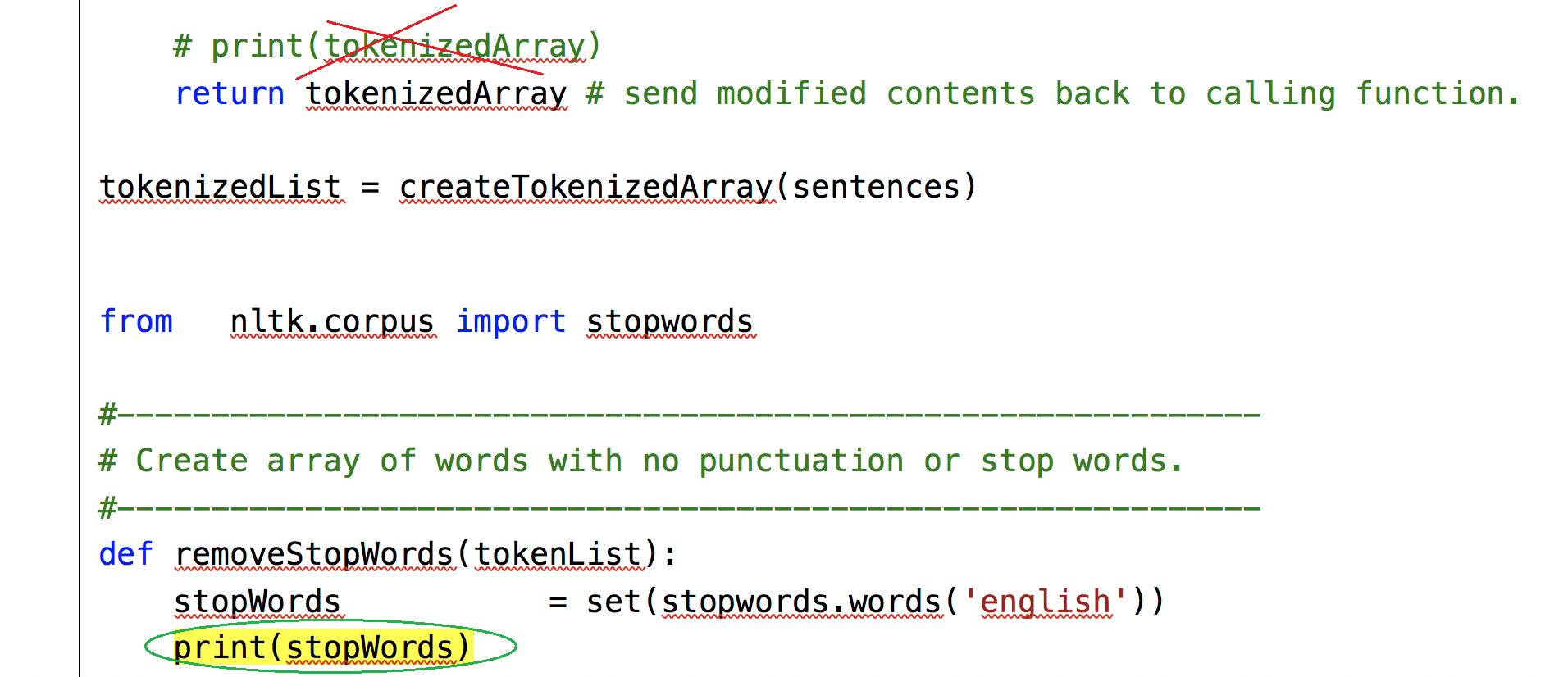
|  |
| --- |
|  |

Exercise 3 (1 marks)

Print out the stop word list using the following instruction from inside the *removeStopWords()* function:

print(stopWords);

Observe the output when adding the highlighted line in removeStopWords(). To get a clearer view of the output remove the print statement that is crossed out in red.



Show your output here:

|  |
| --- |
|  |

Exercise 4 (3 marks)

Create a *modifiedStopWords()* function. Inside this function, create a list with the custom stop words of “charlie” and “angels” in it. Then, take the output from the *removeStopWords()* function in Example 2 and run it through your *modifiedStopWords()* function to also remove the words that exist in your custom stop words list. Print the output to ensure your function worked properly.

Here is some starter code:

|  |
| --- |
| from nltk.tokenize import RegexpTokenizer  sentence1 = "Despite its fresh perspective, Banks's Charlie's Angels update "\  + "fails to capture the energy or style that made it the beloved phenomenon it is."  sentence2 = "This 2019 Charlie's Angels is stupefyingly entertaining and " \  + "hilarious. It is a stylish alternative to the current destructive blockbusters."  sentences = [sentence1, sentence2]  #-------------------------------------------------------------  # Create lower case array of words with no punctuation.  #-------------------------------------------------------------  def createTokenizedArray(sentences):  # Initialize tokenizer and empty array to store modified sentences.  tokenizer = RegexpTokenizer(r'\w+')  tokenizedArray = []  for i in range(0, len(sentences)):  # Convert sentence to lower case.  sentence = sentences[i].lower()    # Split sentence into array of words with no punctuation.  words = tokenizer.tokenize(sentence)    # Append word array to list.  tokenizedArray.append(words)  print(tokenizedArray)  return tokenizedArray # send modified contents back to calling function.  tokenizedList = createTokenizedArray(sentences)  from nltk.corpus import stopwords  #-------------------------------------------------------------  # Create array of words with no punctuation or stop words.  #-------------------------------------------------------------  def removeStopWords(tokenList):  stopWords = set(stopwords.words('english'))  shorterSentences = [] # Declare empty array of sentences.    for sentence in tokenList:  shorterSentence = [] # Declare empty array of words in single sentence.  for word in sentence:  if word not in stopWords:    # Remove leading and trailing spaces.  word = word.strip()    # Ignore single character words and digits.  if(len(word)>1 and word.isdigit()==False):  # Add remaining words to list.  shorterSentence.append(word)  shorterSentences.append(shorterSentence)  return shorterSentences    sentenceArrays = removeStopWords(tokenizedList)  print(sentenceArrays)  def modifiedStopWords(sentenceLists):  updatedList = []  customStopWords = ['charlie', 'angels']  counter = 0  # Loop through list of sentences (there are two sentences)  # Loop through words in the sentence  # Check for words that are not in customStopWords  # append to list if words are not in customStopWords  return updatedList  output = modifiedStopWords(sentenceArrays)  print("The final answer is: ")  print(output) |

Show your revised program here:

|  |
| --- |
|  |

Show the resulting output here:

|  |
| --- |
|  |

### Stemming

The performance of the sentiment algorithms can sometimes be improved if the words are reduced to their root. With stemming, words with similar meaning can be conveyed with just one word. This representative root word can then be weighted more accurately with the algorithm. Stemming is a blunt approach which normalizes the words by removing their suffixes. For example, the following words generally convey the same meaning:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| * connect | * connected | * connecting | * connection | * connections |

Stemming reduces each of these terms to “connect”.

There are other similar types of algorithms for word truncation such as lemmatization which is a more advanced approach that tries to differentiate between verbs and nouns. You can always try to determine if truncation algorithms offer better results but there is no guarantee that they will.

Example 3: Stemming Sentences

This example starts with the code from Example 2. Add the following code after the code that already exists in Example 2. The *stemWords()* function takes the word arrays, stems each word and then joins them back into a sentence.

|  |
| --- |
| from nltk.stem import PorterStemmer  #-------------------------------------------------------------  # Removes suffixes and rebuids the sentences.  #-------------------------------------------------------------  def stemWords(sentenceArrays):  ps = PorterStemmer()  stemmedSentences = []  for sentenceArray in sentenceArrays:  stemmedArray = [] # Declare empty array of words.  for word in sentenceArray:  stemmedArray.append(ps.stem(word)) # Add stemmed word.    # Convert array back to sentence of stemmed words.  delimeter = ' '  sentence = delimeter.join(stemmedArray)  # Append stemmed sentence to list of sentences.  stemmedSentences.append(sentence)  return stemmedSentences  stemmedSentences = stemWords(sentenceArrays)  print(stemmedSentences) |

The output after passing in sentences with no stop words shows one array that contains two truncated sentences:

|  |
| --- |
| ['despit fresh perspect bank charli angel updat fail captur energi style made belov phenomenon',  'charli angel stupefyingli entertain hilari stylish altern current destruct blockbust'] |

Exercise 5 (2 marks)

Run your sentence from Exercise 1 through the *stemWords()* function after it has been tokenized and after stop words have been removed. Show the outcome now:

|  |
| --- |
|  |

### Vectorize (and NGrams)

Vectorization transforms words to numbers so they can be used in different predictive machine learning algorithms. Vectorization is used to create a master number vector that represents unique words for all documents. Then, for each document (sentence in our case), the number of occurrences of each word are recorded with a copy of the master number vector. Remember, you do not need to memorize the code but you will be asked to explain how it works.

Example 4: Vectorization

This example demonstrates how to vectorize two documents:

['the sky is blue', 'the day is bright']

Place the code for this example after the code in Example 3. Then run the project and examine the output from this new *vectorizeList()* function:

|  |
| --- |
| from sklearn.feature\_extraction.text import CountVectorizer  #-------------------------------------------------------------  # Creates a matrix of word vectors.  #-------------------------------------------------------------  def vectorizeList(stemmedList):  #cv = CountVectorizer(binary=True, ngram\_range=(1, 4))  cv = CountVectorizer(binary=True)  cv.fit(stemmedList)  X = cv.transform(stemmedList)  print("\nNumber vector size: " + str(X.shape))  return X  sampleSentences = ['the sky is blue', 'the day is bright']  sampleOutput = vectorizeList(sampleSentences)  # Assigns numbers to words.  print("\nTransformed words: \n" + str(sampleOutput))  # Shows number of times each word appears in the list.  print("Encoded list: \n" + str(sampleOutput.toarray())) |

When you examine the output, you can see the words for each sentence have been transformed to integers. The integers 3 and 5 appear twice. A total of six unique words exist across two documents. The encoded vectors for each document show the number of occurrences of words.

|  |
| --- |
| ﻿﻿Number vector size: (2, 6)  Transformed words:  (0, 0) 1  (0, 3) 1  (0, 4) 1  (0, 5) 1  (1, 1) 1  (1, 2) 1  (1, 3) 1  (1, 5) 1  Shape: (2, 6)  Encoded list:  [[1 0 0 1 1 1]  [0 1 1 1 0 1]] |

Exercise 6 (1 mark)

Which words do 3 and 5 represent?

|  |
| --- |
|  |

Exercise 7 (1 mark)

Observe the four 0’s and four 1’s that located at the left side of the number pairs that are displayed within parenthesis under the title “Transformed words”. What do these 0’s and 1’s represent?

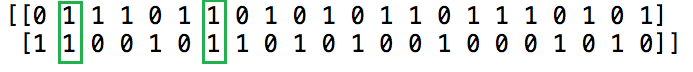
|  |
| --- |
|  |

Exercise 8 (2 marks)

To vectorize the two “Charlie’s Angel’s” reviews, run these two instructions:

|  |
| --- |
| # Vectorize Charlie's Angel's content.  vectorizedSentences = vectorizeList(stemmedSentences)  # Shows number of times each word appears in the list.  print("Encoded list: \n" + str(vectorizedSentences.toarray())) |

Which words are represented by the two highlights in the number vector?



|  |
| --- |
|  |

How do the total rows and columns in the vector output reflect the total words in the stemmed Charlie’s Angel’s reviews?

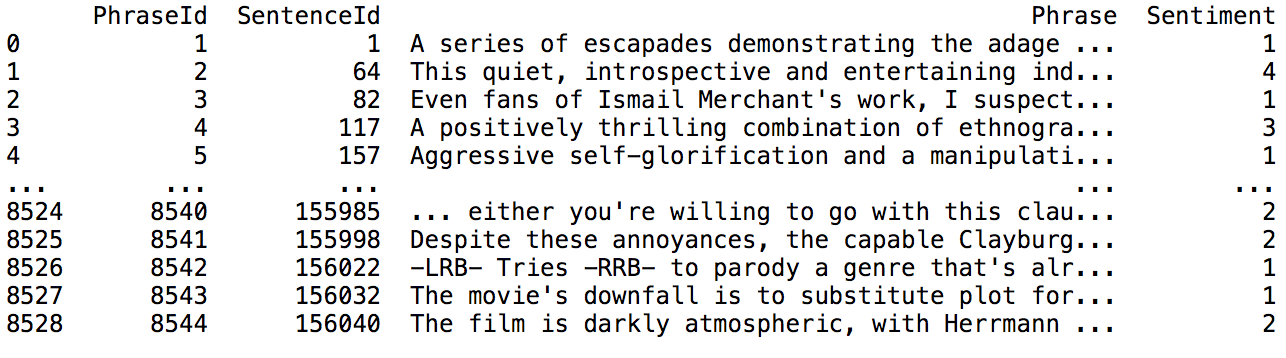
|  |
| --- |
|  |

### Modeling

Today we will look at using logistic regression to build predict the rating classification. In the predictive machine learning course, we will look at using long short-term memory (LSTM) neural networks to build a model for sentiment prediction.

Example 5: Modeling Sentiment

This example uses a movie review data set with reviews from Rotten Tomatoes. The reviews range from 0 (Very Poor) to 4 (Excellent). Here is a snapshot of the DataFrame head and tail.



You will notice the punctuation, mixed case and stop words. Before we train our model, we will run our data through the methods that we made earlier to tokenize, remove stop words, stem and vectorize the data.

There is really nothing new in the code aside from the use of the LogisticRegression class to build our model along with code to evaluate the model.

|  |
| --- |
| import nltk  from sklearn.model\_selection import train\_test\_split  from sklearn.linear\_model import LogisticRegression  from nltk.tokenize import RegexpTokenizer  from sklearn.metrics import accuracy\_score  from nltk.stem import PorterStemmer  from nltk.corpus import stopwords  from sklearn.feature\_extraction.text import CountVectorizer  import math  # To get stop words.  nltk.download('stopwords')  # -------------------------------------------------------------  # Create lower case array of words with no punctuation.  # -------------------------------------------------------------  def createTokenizedArray(sentences):  # Initialize tokenizer and empty array to store modified sentences.  tokenizer = RegexpTokenizer(r'\w+')  tokenizedArray = []  for i in range(0, len(sentences)):  # Convert sentence to lower case.  sentence = sentences[i].lower()  # Split sentence into array of words with no punctuation.  words = tokenizer.tokenize(sentence)  # Append word array to list.  tokenizedArray.append(words)  print(tokenizedArray)  return tokenizedArray # send modified contents back to calling function.  # -------------------------------------------------------------  # Create array of words with no punctuation or stop words.  # -------------------------------------------------------------  def removeStopWords(tokenList):  stopWords = set(stopwords.words('english'))  shorterSentences = [] # Declare empty array of sentences.  for sentence in tokenList:  shorterSentence = [] # Declare empty array of words in single sentence.  for word in sentence:  if word not in stopWords:  # Remove leading and trailing spaces.  word = word.strip()  # Ignore single character words and digits.  if (len(word) > 1 and word.isdigit() == False):  # Add remaining words to list.  shorterSentence.append(word)  shorterSentences.append(shorterSentence)  return shorterSentences  # -------------------------------------------------------------  # Removes suffixes and rebuids the sentences.  # -------------------------------------------------------------  def stemWords(sentenceArrays):  ps = PorterStemmer()  stemmedSentences = []  for sentenceArray in sentenceArrays:  stemmedArray = [] # Declare empty array of words.  for word in sentenceArray:  stemmedArray.append(ps.stem(word)) # Add stemmed word.  # Convert array back to sentence of stemmed words.  delimeter = ' '  sentence = delimeter.join(stemmedArray)  # Append stemmed sentence to list of sentences.  stemmedSentences.append(sentence)  return stemmedSentences  #-------------------------------------------------------------  # Creates a matrix of word vectors.  #-------------------------------------------------------------  def vectorizeList(stemmedList):  #cv = CountVectorizer(binary=True, ngram\_range=(1, 4))  cv = CountVectorizer(binary=True)  cv.fit(stemmedList)  X = cv.transform(stemmedList)  print("\nNumber vector size: " + str(X.shape))  return X  import pandas as pd  from sklearn import metrics  from sklearn.metrics import classification\_report  # -------------------------------------------------------------  # Build model and predict scores.  #  # Parameters:  # X - X contains the stemmed and vectorized sentences.  # target - The target is the known rating (0 to 4).  # Returns X\_test, y\_test, and y\_predicted values.  # -------------------------------------------------------------  def modelAndPredict(X, target):  # Create training set with 75% of data and test set with 25% of data.  X\_train, X\_test, y\_train, y\_test = train\_test\_split(  X, target, train\_size=0.75  )  # Build the model with the training data.  model = LogisticRegression(solver='newton-cg').fit(X\_train, y\_train)  # Predict target values.  y\_prediction = model.predict(X\_test)  accuracy = accuracy\_score(y\_test, y\_prediction)  print("\n\n\*\*\* The accuracy score is: " + str(accuracy))  print(classification\_report(y\_test, y\_prediction))  rmse2 = math.sqrt(metrics.mean\_squared\_error(y\_test, y\_prediction))  print("RMSE: " + str(rmse2))  # Your Python functions can return multiple values.  return X\_test, y\_test, y\_prediction  # Read in the file.  PATH = "/Users/pm/Desktop/DayDocs/data/"  CLEAN\_DATA = "cleanedMovieReviews.tsv"  df = pd.read\_csv(PATH + CLEAN\_DATA, skiprows=1,  sep='\t', names=('PhraseId', 'SentenceId', 'Phrase', 'Sentiment'))  # Prepare the data.  df['PhraseAdjusted'] = createTokenizedArray(df['Phrase'])  df['PhraseAdjusted'] = removeStopWords(df['PhraseAdjusted'])  df['PhraseAdjusted'] = stemWords(df['PhraseAdjusted'])  vectorizedList = vectorizeList(df['PhraseAdjusted'])  # Get predictions and scoring data.  # Target is the rating that we want to predict.  X\_test, y\_test, y\_predicted = modelAndPredict(vectorizedList, df['Sentiment']) |

The output when running this code is:

|  |
| --- |
| \*\*\* The accuracy score is: 0.36802625410220347  precision recall f1-score support  0 0.34 0.22 0.26 273  1 0.37 0.43 0.40 542  2 0.22 0.22 0.22 386  3 0.43 0.48 0.45 601  4 0.45 0.36 0.40 331  accuracy 0.37 2133  macro avg 0.36 0.34 0.35 2133  weighted avg 0.37 0.37 0.36 2133  RMSE: 1.2630936355292084 |

You may not be very impressed with 36.8% accuracy but we will analyze this further. The metric actually is not bad.

### Using RMSE to Evaluate a Classification Model

Since there are multiple ratings and they are sequential you can use a RMSE to analyse the model. **However, NEVER use RMSE to evaluate binary models. Also, NEVER use RMSE for evaluating classification models which do not provide sequential outcomes.** For example,you would never use RMSE to evaluate a classification model which predicts flower type.

The RMSE 1.2630936355292084 suggests that the model is on average within 1.26 ratings of the actual rating.

#### Confusion Matrices

If we examine the confusion matrix in Figure 1 we can see a path of relatively accurate predictions along the diagonal from top left to the bottom right. The darker colours indicate that the predictions are within a decent range of the actual score.

Figure 1: Confusion Matrix

|  |  |
| --- | --- |
|  | ﻿[[ 68 127 38 28 5]  [ 57 249 86 109 25]  [ 21 165 89 142 33]  [ 13 97 90 269 93]  [ 3 36 38 136 116]] |

Example 6: Printing a Confusion Matrix

Here is the code to show the confusion matrix. Add this code onto the end of Example 5.

|  |
| --- |
| from sklearn import metrics  # Draw the confusion matrix.  def showConfusionMatrix(y\_test, y\_predicted):  # You can print a simple confusion matrix with no formatting – this is easiest.  cm = metrics.confusion\_matrix(y\_test.values, y\_predicted)  print(cm)  showConfusionMatrix(y\_test, y\_predicted) |

### Hot Words with N-Grams

Word counts help analysts identify words that frequently appear for specific ratings. Hot words can highlight the issues that are being discussed most frequently in large amounts of text.

Often, the true meaning of a sentiment can only be understood when viewing phrases. N-Grams allow you to generate phrases that consist of two or more word groupings. Likely during your analysis you will want to examine the highest recurring word and N-gram collections for your high and low ratings.

Example 7: Showing Word Counts and N-Grams

This example shows how to generate N-grams. The top 80 recurring results are selected but this can be adjusted as desired. To build this example, add this code onto the end of Example 6.

|  |
| --- |
| ﻿from collections import Counter  from nltk.util import ngrams  def generateWordList(wordDf, scoreStart, scoreEnd, n\_gram\_size):  resultDf=wordDf[(wordDf['Sentiment']>=scoreStart) & \  (wordDf['Sentiment']<=scoreEnd)]    sentences = [sentence.split() for sentence in resultDf['PhraseAdjusted']]  wordArray = []  for i in range(0, len(sentences)):  wordArray += sentences[i]  counterList = Counter(ngrams(wordArray, n\_gram\_size)).most\_common(80)    print("\n\*\*\*N-Gram")  for i in range(0,len(counterList)):  print("Occurrences: ", str(counterList[i][1]), end=" ")  delimiter = ' '  print(" N-Gram: ", delimiter.join(counterList[i][0]))  return counterList  # Create two column matrix.  dfSub = df[['Sentiment','PhraseAdjusted']]  SCORE\_RANGE\_START = 4  SCORE\_RANGE\_END = 4  SIZE = 1  counterList = generateWordList(dfSub,SCORE\_RANGE\_START, SCORE\_RANGE\_END, SIZE)  SIZE = 3  counterList = generateWordList(dfSub,SCORE\_RANGE\_START, SCORE\_RANGE\_END, SIZE) |

The output shows N-Grams for movies that are rated excellent. Single word N-Grams are shown on the left and three-word N-Grams are on the right. In both cases, the descriptions are not very useful since the N-Grams are for many movies.

**Note:** Unfortunately, these n-grams apply to many different movies so the summary they present is not really representative of one movie. However, if the n-grams were generated from many samples for one specific movie they would reveal significant insight into what people are really saying about it.

|  |  |
| --- | --- |
| ﻿Occurrences: 237 N-Gram: film  Occurrences: 168 N-Gram: movi  Occurrences: 121 N-Gram: one  Occurrences: 87 N-Gram: perform  Occurrences: 77 N-Gram: make  Occurrences: 72 N-Gram: best  Occurrences: 66 N-Gram: year  Occurrences: 65 N-Gram: work  Occurrences: 61 N-Gram: well  Occurrences: 60 N-Gram: comedi | ﻿Occurrences: 5 N-Gram: one year best  Occurrences: 4 N-Gram: movi ever made  Occurrences: 4 N-Gram: best film year  Occurrences: 3 N-Gram: come age tale  Occurrences: 3 N-Gram: year best film  Occurrences: 3 N-Gram: come age stori  Occurrences: 2 N-Gram: one best film  Occurrences: 2 N-Gram: film year film  Occurrences: 2 N-Gram: kiss jessica stein  Occurrences: 2 N-Gram: romant comedi genr |

Exercise 9 (1 mark)

Show the N-Grams of 2 which appear for scores between 0 and 1.

|  |
| --- |
|  |

Exercise 10 (1 mark)

Show the N-Grams of 3 which appear for scores between 0 and 1.

|  |
| --- |
|  |

Exercise 11 (1 mark)

This exercise will help to show how N-GRAMS group items. Add this code after the code from Example 7.

|  |
| --- |
| # Create DataFrame.  simpleDataSet = {'PhraseAdjusted': ['the sky is blue'], 'Sentiment': [4] }  dfSimple = pd.DataFrame(simpleDataSet, columns= ['Sentiment',  'PhraseAdjusted'])  SIZE = 2  newNGrams = generateWordList(dfSimple,SCORE\_RANGE\_START, SCORE\_RANGE\_END,  SIZE)  print(newNGrams) |

Show the *newNGrams* list values that is printed by the code above.

Show the output after running this code block. After observing the N-grams, explain in your own words how N-Gram pairs are generated.

|  |
| --- |
|  |

Exercise 12

Modify Example 7 so the code loads and evaluates hotel reviews.

df = pd.read\_csv("/Users/pm/Downloads/tripadvisor\_hotel\_reviews.csv")

Remember to adjust the target variable name as needed.

Additional code modifications are required. Once the code is working properly.

Show the confusion matrix here (3 marks):

|  |
| --- |
|  |

Show the precision, recall, f1-score metrics report here (2 marks):

|  |
| --- |
|  |

Show the RMSE here:

|  |
| --- |
|  |

Show the top 15 most common 3-word ngrams for the poorest rating here:

|  |
| --- |
|  |

Show the top 15 most common 4-word ngrams for the highest rating here:

|  |
| --- |
|  |