Preface

In this notebook, I will try to use Amazon food review data found on Kaggle in order to make a model that can predict whether or not a review is a postitive or negative review purely based on the text portion of the review. I will use multiple methods to build different models and compare their effectiveness with each other. This is my first time trying to use NLP methods so a lot of the things I do in this will most likely be learned along the way.

Cleaning the data

Since the dataset was found on Kaggle, there isn't much work needed to clean the data up for modeling, but I'd like to remove the columns I don't plan on using and separate positive reviews from negative ones. For simplicity, scores of 3 or higher are considered positive while scores of 1 and 2 are considered negative. I could make scores of 3 be considered neutral, but adding a 3rd category would likely make the model less effective/more confusing since neutral review are likely to contain words used in both negative and positive statements depending on context.

In [1]:
```
# read downloaded data
import pandas as pd
reviews_df = pd.read_csv('C:\\Users\\beton\\Documents\\Personal Projects\\Food Review N
```

In [2]:
```
reviews_df
```

Out[2]:

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenc
0	1	B001E4KFG0	A3SGXH7AUHU8GW	delmartian	1	
1	2	B00813GRG4	A1D87F6ZCVE5NK	dll pa	0	
2	3	B000LQOCH0	ABXLMWJIXXAIN	Natalia Corres "Natalia Corres"	1	
3	4	B000UA0QIQ	A395BORC6FGVXV	Karl	3	
4	5	B006K2ZZ7K	A1UQRSCLF8GW1T	Michael D. Bigham "M. Wassir"	0	

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenc
...	...	...	...	...	...	
568449	568450	B001EO7N10	A28KG5XORO54AY	Lettie D. Carter	0	
568450	568451	B003S1WTCU	A3I8AFVPEE8KI5	R. Sawyer	0	
568451	568452	B004I613EE	A121AA1GQV751Z	pksd "pk_007"	2	
568452	568453	B004I613EE	A3IBEVCTXKNOH	Kathy A. Welch "katwel"	1	
568453	568454	B001LR2CU2	A3LGQPJCZVL9UC	srfell17	0	

568454 rows × 10 columns

In [3]:

```
import pandas as pd
import numpy as np

# remove unnecessary columns
review_text = reviews_df.drop(['Id', 'ProductId', 'UserId', 'ProfileName', 'Helpfulness

# change scores to positive or negative label
review_text.replace({'Score': {1: 'negative', 2: 'negative', 3: 'positive', 4: 'positiv

# make a trimmed version of the dataset for faster training (needed for SVM model other
np.random.seed(10)
remove_n = 558454
drop_indices = np.random.choice(review_text.index, remove_n, replace=False)
review_trimmed = review_text.drop(drop_indices)
```

In [4]:

```
review_trimmed.reset_index(drop=True, inplace=True)
review_trimmed.head()
```

Out[4]:

	Score	Text
0	positive	I bought these for my husband who is currently...
1	positive	I have lived out of the US for over 7 yrs now,...

	Score	Text
2	positive	Natural Balance Dry Dog Food Lamb Meal and Bro...
3	positive	not what I was expecting in terms of the compa...
4	negative	Terrible! Artificial lemon taste, like Pledge ...

In [5]:
```
print(review_trimmed.iloc[0,1])
```

```
I bought these for my husband who is currently overseas. He loves these, and apparently
his staff likes them also.<br />There are generous amounts of Twizzlers in each 16-ounce
bag, and this was well worth the price. <a href="http://www.amazon.com/gp/product/B001GV
ISJM">Twizzlers, Strawberry, 16-Ounce Bags (Pack of 6)</a>
```

In [6]:
```
print(review_trimmed.iloc[6,1])
```

```
Sir Kensington's did a great job of updating the classic ketchup with this wonderful pro
duct. The refreshed taste of this Ketchup is a great update, and now leaves me disappoin
ted when I'm given Heinz while out at a restaurant.<br /><br />For you Heinz die hard fa
ns out there, this is not the ketchup for you. But for those of you who wish you always
knew what ketchup could be without the chemical aftertaste found in Heinz, be sure to gi
ve this Ketchup a try.<br /><br />Don't forget the spiced variety. Purchasing the pack w
ith both the classic and spiced variety for your first Sir Kensington experience is defi
nitely the way to go.
```

Model 1: Simple Classifier using sklearn CountVectorizer and an SVM Model

In [7]:
```
# import libraries
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split

# split data into testing and training sets; training set is made small due to how long
x_train, x_test, y_train, y_test = train_test_split(np.array(review_trimmed['Text']), n

# transform data
vectorizer = CountVectorizer()
x_train_vector = vectorizer.fit_transform(x_train)
x_test_vector = vectorizer.transform(x_test)
```

In [8]:
```
x_train.size
```

Out[8]:
```
7000
```

In [9]:
```
# train model
from sklearn import svm

clf_svm = svm.SVC(kernel='linear')
clf_svm.fit(x_train_vector, y_train)
```

Out[9]:
```
SVC(kernel='linear')
```

In [10]:
```
# evaluate model
clf_svm.score(x_test_vector, y_test)
```

Out[10]:
```
0.8716666666666667
```

With the test set, the SVM model has an accuracy of about 87%, which is not bad. However, it's a good idea to test the model with a few made up inputs as well.

In [11]:
```
test = ['This food is amazing! Tastes great!','Not good. The food was lacking in flavor
X = vectorizer.transform(test)
clf_svm.predict(X)
```

Out[11]:
```
array(['positive', 'positive', 'negative', 'positive'], dtype=object)
```

From the test statements above, we can see that the model has a problem with correctly identifying negative sentiments from phrases including multiple words. That's because this model only counts single words when vectorizing and which ignores the effects of modifying words like 'not'. I'll try to account for this using the ngram_range parameter for the CountVectorizer. I'll also see if using the stop_words parameter for further refinement may work.

In [12]:
```
# train and evaluate model with ngram range of 2
vectorizer2 = CountVectorizer(ngram_range=(1,2))
x_train_vector2 = vectorizer2.fit_transform(x_train)
x_test_vector2 = vectorizer2.transform(x_test)

clf_svm2 = svm.SVC(kernel='linear')
clf_svm2.fit(x_train_vector2, y_train)

print('ngram 2 accuracy=', clf_svm2.score(x_test_vector2, y_test))

# train and evaluate model with ngram range of 2 and stop words
vectorizer3 = CountVectorizer(ngram_range=(1,2), stop_words='english')
x_train_vector3 = vectorizer3.fit_transform(x_train)
x_test_vector3 = vectorizer3.transform(x_test)

clf_svm3 = svm.SVC(kernel='linear')
clf_svm3.fit(x_train_vector3, y_train)

print('ngram 2 with stop words accuracy=', clf_svm3.score(x_test_vector3, y_test))

# train and evaluate model with only stop_words accounted for
vectorizer4 = CountVectorizer(stop_words='english')
x_train_vector4 = vectorizer4.fit_transform(x_train)
x_test_vector4 = vectorizer4.transform(x_test)

clf_svm4 = svm.SVC(kernel='linear')
clf_svm4.fit(x_train_vector4, y_train)

print('stop words accuracy=', clf_svm4.score(x_test_vector4, y_test))
```

```
ngram 2 accuracy= 0.897
ngram 2 with stop words accuracy= 0.8883333333333333
stop words accuracy= 0.868
```

Based on the results above, it seems that altering the word combination count increased model accuracy, but removing stop words decreased accuracy. The default 'english' stop words list is known to have issues, so using an alternative list found online or making my own list of stop words may work better. To further optimize the model, I could iterate to find the best-scoring combination of ngram values and stop word lists.

I could also use a different vectorizer to further fit and transform the data.

In [13]:
```
# Make a model using TfidfTransformer based on the original model
from sklearn.feature_extraction.text import TfidfTransformer
tf_transformer = TfidfTransformer()
x_train_tfidf = tf_transformer.fit_transform(x_train_vector)
x_test_tfidf = tf_transformer.fit_transform(x_test_vector)

# Train model
clf_svm_tfidf = svm.SVC(kernel='linear')
clf_svm_tfidf.fit(x_train_tfidf, y_train)
print('TfidfTransformer score=', clf_svm_tfidf.score(x_test_tfidf, y_test))
```

```
TfidfTransformer score= 0.8893333333333333
```

Using TfidfTransformer, which weighs the frequency of tokens in order to better determine how important they are, also improved the score from the original model.

Model 2: Simple Classifier using sklearn CountVectorizer and a Naive Bayes Model

Now I'll do the same thing using a Naive Bayes Model

In [14]:
```
# make a pipeline to simplify the fitting and modeling process
from sklearn.pipeline import Pipeline
from sklearn.naive_bayes import MultinomialNB

NB_pipeline = Pipeline([('vect', CountVectorizer()),('clf', MultinomialNB())])

# train and test model using the same training and testing sets
NB_pipeline.fit(x_train, y_train)

# evaluate model
NB_pipeline.score(x_test, y_test)
```

Out[14]:
```
0.879
```

In [ ]:
```
# train and evaluate model with ngram range of 2
NB_pipeline2 = Pipeline([('vect', CountVectorizer(ngram_range=(1,2))),('clf', Multinomi
NB_pipeline2.fit(x_train, y_train)

print('NB ngram 2 accuracy=', NB_pipeline2.score(x_test, y_test))

# train and evaluate model with ngram range of 2 and stop words
NB_pipeline3 = Pipeline([('vect', CountVectorizer(ngram_range=(1,2), stop_words='englis
NB_pipeline3.fit(x_train, y_train)

print('NB ngram 2 with stop words accuracy=', NB_pipeline3.score(x_test, y_test))

# train and evaluate model with only stop_words accounted for
NB_pipeline4 = Pipeline([('vect', CountVectorizer(stop_words='english')),('clf', Multin
NB_pipeline4.fit(x_train, y_train)

print('NB stop words accuracy=', NB_pipeline4.score(x_test, y_test))

# Make a model using TfidfTransformer
NB_tfidf_pipeline = Pipeline([('vect', CountVectorizer()),('tfidf', TfidfTransformer())
NB_tfidf_pipeline.fit(x_train,y_train)

print('NB tfidf=', NB_tfidf_pipeline.score(x_test, y_test))
```

With the Naive Bayes model, the model that implements only stop words performs the best, though not quite as well as the most optimized SVM model.

Model 3: Classifier using Spacy Word Vectors and an SVM Model

In [16]:
```
import spacy
```

In [17]:
```
# Load trained spacy pipeline for English
nlp = spacy.load("en_core_web_md")

# apply pipeline to training data and store Doc
docs = [nlp(text) for text in x_train]
x_train_wv = [x.vector for x in docs]

docs2 = [nlp(text) for text in x_test]
x_test_wv = [x.vector for x in docs2]
```

In [18]:
```
clf_svm_wv = svm.SVC(kernel='linear')
clf_svm_wv.fit(x_train_wv, y_train)
```

Out[18]:
```
SVC(kernel='linear')
```

In [19]:
```
print('SVM using spacy accuracy=', clf_svm_wv.score(x_test_wv, y_test))
```

```
SVM using spacy accuracy= 0.8733333333333333
```

Using Spacy, this SVM model performs roughly the same as the original SVM model using CountVectorizer

Clean Up Text Using Regex

Some of the text reviews contain html tags which end up being used to train the model despite having no meaning. Using regular expression, I'll try to get rid of the main tags (line breaks and hyperlinks) that I found here and there when skimming though the texts.

In [20]:
```
import re

review_trimmed.reset_index()
for i in range(len(review_trimmed.index)):
    text = review_trimmed.iloc[i]['Text']
    regexp = re.compile(r"\<a.*a\>|\<br.*/>")
    text2 = re.sub(regexp, '', text)
    review_trimmed.at[i,'Text'] = text2
```

Using Lemmatization and Stop Word Removal to Clean Data Further

With some of the extraneous tags gone, I can do some more cleaning. The main changes I'll make to the texts are lemmatization and stop word removal (using nltk this time). (*I originally wanted to use spell check with TextBlob but using spell check on long statements in a loop of 10000 statements took far too long)*. Stop word removal and spell correction are relatively self-explanatory.

Lemmatization is the process of grouping together the different forms a single word can have into one group so that they can be analyzed as a single entity. Words grouped up through lemmatization should allow NLP models to be better trained since they should, ideally, no long treat different forms of the same word as different words.

In [21]:
```
import nltk

nltk.download('wordnet')
nltk.download('stopwords')
nltk.download('punkt')
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\beton\AppData\Roaming\nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\beton\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\beton\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Out[21]: True

In [22]:
```
# use nltk lemmatizer to lemmatize
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords
import string

lemmatizer = WordNetLemmatizer()
stop_words = stopwords.words('english')
review_cleaned = pd.DataFrame()


for i in range(len(review_trimmed.index)):
    text = review_trimmed.iloc[i]['Text']
    words = word_tokenize(text)

    # make text lower case
    words = [w.lower() for w in words]

    #remove puncuation
    table = str.maketrans('', '', string.punctuation)
    words_stripped = [word.translate(table) for word in words]

    # remove non-alpha words
    words_no_punct = [word for word in words_stripped if word.isalpha()]

    # filter out stop words
    words_cleaned = [word for word in words_no_punct if not word in stop_words]

    new_text = " ".join(words_cleaned)
    review_cleaned.at[i,'Score'] = review_trimmed.iloc[i]['Score']
    review_cleaned.at[i,'Text'] = new_text
```

In [23]:
```
# split data into testing and training sets; training set is made small due to how long
x_clean_train, x_clean_test, y_clean_train, y_clean_test = train_test_split(np.array(re

# transform data
```

```
vectorizer = CountVectorizer()
x_clean_train_vector = vectorizer.fit_transform(x_clean_train)
x_clean_test_vector = vectorizer.transform(x_clean_test)

clf_svm = svm.SVC(kernel='linear')
clf_svm.fit(x_clean_train_vector, y_clean_train)

clf_svm.score(x_clean_test_vector, y_clean_test)
```

Out[23]: 0.877

In [24]:

```
# train and evaluate model with ngram range of 2
vectorizer2 = CountVectorizer(ngram_range=(1,2))
x_train_vector2 = vectorizer2.fit_transform(x_clean_train)
x_test_vector2 = vectorizer2.transform(x_clean_test)

clf_svm2 = svm.SVC(kernel='linear')
clf_svm2.fit(x_train_vector2, y_clean_train)

print('ngram 2 accuracy=', clf_svm2.score(x_test_vector2, y_clean_test))

# train and evaluate model with ngram range of 2 and stop words
vectorizer3 = CountVectorizer(ngram_range=(1,2), stop_words='english')
x_train_vector3 = vectorizer3.fit_transform(x_clean_train)
x_test_vector3 = vectorizer3.transform(x_clean_test)

clf_svm3 = svm.SVC(kernel='linear')
clf_svm3.fit(x_train_vector3, y_clean_train)

print('ngram 2 with stop words accuracy=', clf_svm3.score(x_test_vector3, y_clean_test)

# train and evaluate model with only stop_words accounted for
vectorizer4 = CountVectorizer(stop_words='english')
x_train_vector4 = vectorizer4.fit_transform(x_clean_train)
x_test_vector4 = vectorizer4.transform(x_clean_test)

clf_svm4 = svm.SVC(kernel='linear')
clf_svm4.fit(x_train_vector4, y_clean_train)

print('stop words accuracy=', clf_svm4.score(x_test_vector4, y_clean_test))
```

```
ngram 2 accuracy= 0.8845
ngram 2 with stop words accuracy= 0.885
stop words accuracy= 0.8665
```

When comparing classifier accuracy using the default settings, the SVM model using the cleaned data performed slightly better than the original SVM model, but when using differnt ngram and stop_words parameters, the model performs worse than the previous SVM model.

In [ ]: