AMAZON SENTIMENT ANALYSIS

BY: ***SIDDHARTH MANDGI***,
SARA NOOR &
SESHASAI CHATURVEDULA

STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

Objective

The goal of the project is to carry out sentiment analysis on Amazon Books dataset based on reviews on a million books and predict whether this review is 'Positive', 'Neutral', 'Negative'. Sentiment analysis is a text analysis method that detects polarity (e.g.
a *positive* or *negative* opinion) within text, whether a whole document, paragraph, sentence, or clause. The reason for selecting this project is because understanding people's emotions is essential for businesses since customers can express their thoughts and feelings more openly than ever before. By automatically analyzing customer feedback, from survey responses to social media conversations, brands can listen attentively to their customers, and tailor products and services to meet their needs. For example, using sentiment analysis to automatically analyze 4,000+ reviews about your product could help you discover if customers are happy about your pricing plans and customer service.

Our Approaches

Approach 1: Using spaCy for tokenization, Lemmatization and Removing stopwords and using scikit-learn to build our models for different batches of data and using Ensemble Techniques to create an aggregate prediction result.

Approach 2: Using NLTK for tokenization, Lemmatization and Removing Stop words and using scikit-learn to build our models for different batches of data.

Approach 3: Using a custom-built Naïve Bayes model

Approach 4: Handling Big Data

Libraries

Following libraries/packages are used: -

spaCy:
spaCy is a library for advanced Natural Language Processing in Python and Cython. It's built on the very latest research and was designed from day one to be used in real products. spaCy comes with pretrained statistical models and word vectors, and currently supports tokenization for **50+ languages**. It features state-of-the-art speed, convolutional **neural network models** for tagging, parsing and **named entity recognition** and easy **deep learning** integration. It's commercial open-source software, released under the MIT license.

Scikit-Learn:

Scikit-learn is probably the most useful library for machine learning in Python. The sklearn library contains a lot of efficient tools for machine learning and statistical modeling including classification, regression, clustering and dimensionality reduction.

Components of scikit-learn:

- Supervised learning algorithms
- Cross-validation
- Unsupervised learning algorithms
- Various toy datasets
- Feature extraction

NLTK:

NLTK abbreviated for Natural Language Toolkit is a famous library helping python programmer to work in the field of Natural Language Processing. It consists of libraries for tokenizing, stemming, lemmatizing, parsing, part of speech word tagging, name entity recognition etc.

The source is available at https://github.com/siddh30/Amazon-Sentiment-Analysis, and be run on jupyter notebook/ google colab. For running please change the directory path accordingly as below

```
review_data_path = 'amazon_reviews_us_Books_v1_02.tsv'

reviews = pd.read_table(review_data_path, error_bad_lines=False, encoding='utf-8')

reviews.head()
```

Approach 1: SPACY and Sci-kit learn

Preprocessing data

- In this approach we have used a custom function to tokenize our feature by using spaCy's in built stop words function to compare and remove our stop words and punctuations along with lemmatization.

```
def spacy_tokenizer(sentence):
    mytokens = parser(sentence)
    mytokens = [word.lemma_.lower() if word.lemma_ != "-PRON-" else word.lower_ for word in mytoke
ns]
    mytokens = [word for word in mytokens if word not in stopwords and word not in punctuations]
    return mytokens
```

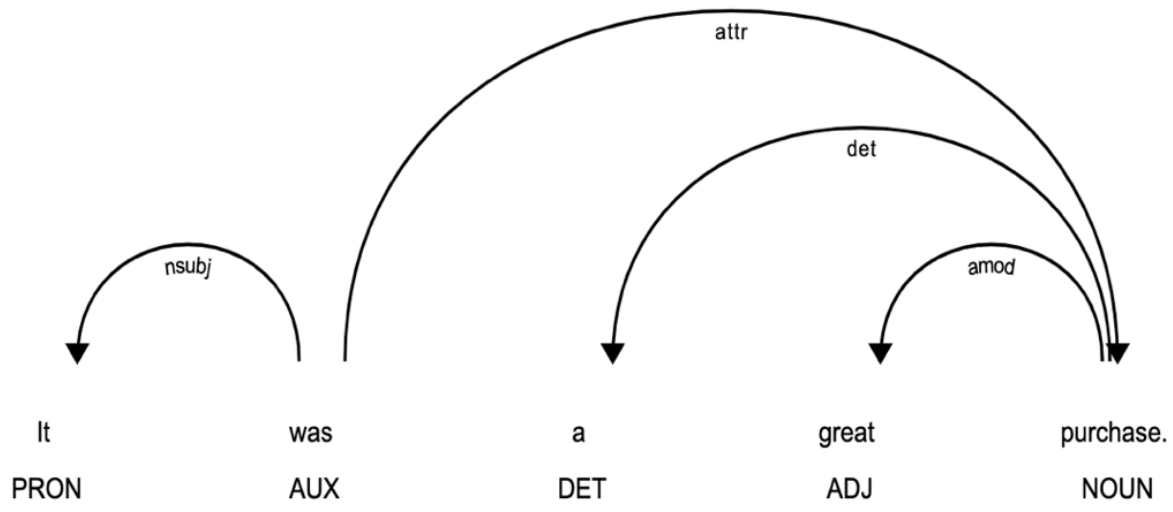
- We have also built a custom transformer using spaCy to transform and our reviews.

```
#Custom Transformer using spaCy
class predictors(TransformerMixin):

    def transform(self, X, **transform_params):
        return [clean_text(text) for text in X]
    def fit(self, X, y=None, **fit_params):
        return self
    def get_params(self,deep=True):
        return{}

# Basic function to clean the text
def clean_text(text):
    return text.strip().lower()
```

- Apart from this we have also created a Tfidf Vectorizer and along with our sklearn ml models created a pipeline.

```
pipe = Pipeline([('cleaner', predictors()),
                 ('tfidfVect', tfdifVect),
                ('classifier',classifier),])
```

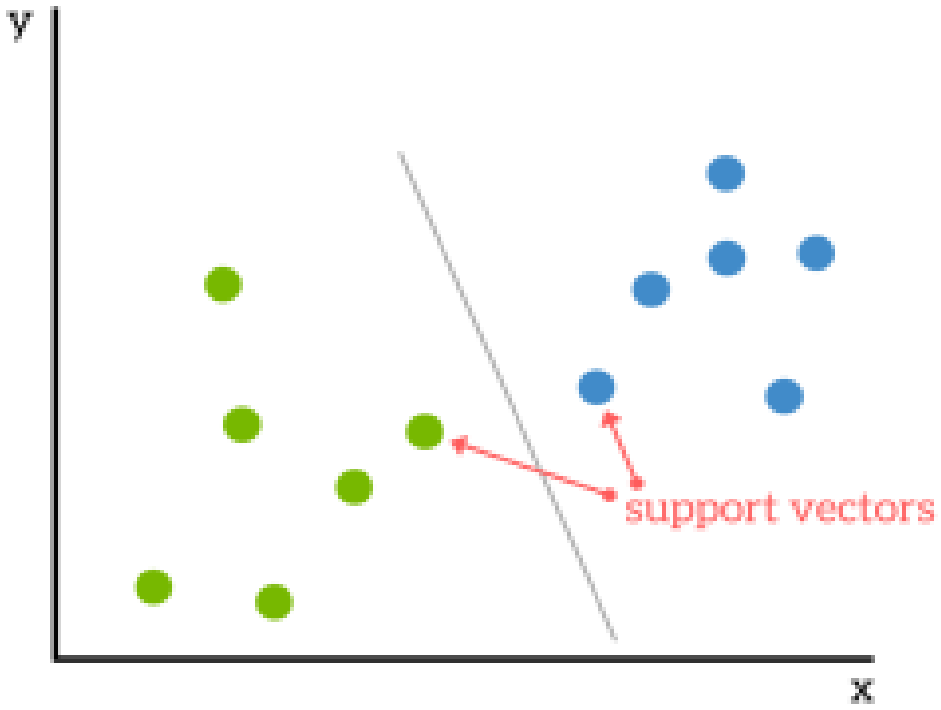
- We have also used spaCy for visualization of different dependencies of our reviews.



Machine Learning Models using Sklearn

Support Vector Machines

A Support Vector Machine (SVM) is a supervised machine learning algorithm that can be employed for both classification and regression purposes. SVMs are more commonly used in classification problems and as such SVMs are based on the idea of finding a hyperplane that best divides a dataset into two classes, as shown in the image below.



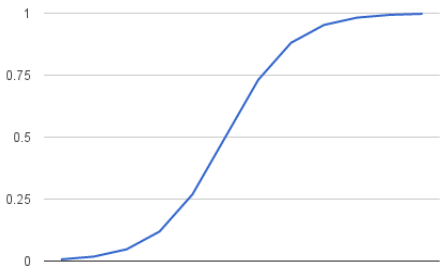
Hyperplane

A hyperplane as a line that linearly separates and classifies a set of data. Intuitively, the further from the hyperplane our data points lie, the more confident we are that they have been correctly classified. We therefore want our data points to be as far away from the hyperplane as possible, while still being on the correct side of it. So, when new testing data is added, whatever side of the hyperplane it lands will decide the class that we assign to it.

Logistic Regression

Logistic regression is named for the function used at the core of the method, the logistic function. The logistic function, also called the sigmoid function was developed by statisticians to describe properties of population growth in ecology, rising quickly and maxing out at the carrying capacity of the environment. It's an S-shaped curve that can take any real-valued number and map it into a value between 0 and 1, but never exactly at those limits

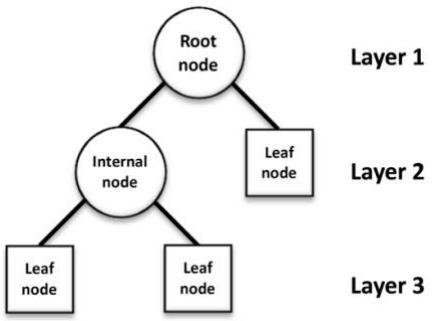
The sigmoid function is given by -> 1 / (1 + e^-value)



Decision Trees

A **decision tree** is a decision support tool that uses a tree-like graph or model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It is one way to display an algorithm that only contains conditional control statements.

A decision tree is a flowchart-like structure in which each internal node represents a "test" on an attribute (e.g. whether a coin flip comes up heads or tails), each branch represents the outcome of the test, and each leaf node represents a class label (decision taken after computing all attributes). The paths from root to leaf represent classification rules.



Results:

Models	Accuracies
Support Vector Machines	0.89605
Logistic Regression	0.89605
Decision Trees	0.84955

Ensemble Learning Algorithms

- Bagging - Bootstrap Aggregation (or Bagging for short), is a simple and very powerful ensemble method. Bagging is the application of the Bootstrap procedure to a high-variance machine learning algorithm, typically decision trees.

- Boosting - Boosting refers to a group of algorithms that utilize weighted averages to make weak learners into stronger learners. Unlike bagging that had each model run independently and then aggregate the outputs at the end without preference to any model. Boosting is all about “teamwork”. Each model that runs, dictates what features the next model will focus on.

- Random Forest - Random forest is a supervised learning algorithm. The "forest" it builds, is an ensemble of decision trees, usually trained with the “bagging” method. The general idea of the bagging method is that a combination of learning models increases the overall result. Put simply: random forest builds multiple decision trees and merges them together to get a more accurate and stable prediction**.** One big advantage of random forest is that it can be used for both classification and regression problems, which form the majority of current machine learning systems.

Results:

Models	Accuracies
Bagging	0.877
Boosting	0.872
Random Forest	0.88475

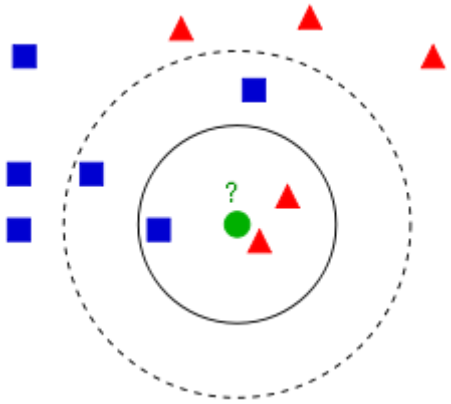
Approach 2: NLTK and Scikit-Learn

KNN

In pattern recognition, the ***k*-nearest neighbors algorithm** (***k*-NN**) is a non-parametric method used for classification and regression. In both cases, the input consists of the *k* closest training examples in the feature space. The output depends on whether *k*-NN is used for classification or regression:

In *k-NN classification*, the output is a class membership. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its *k* nearest neighbors (*k* is a positive integer, typically small). If *k* = 1, then the object is simply assigned to the class of that single nearest neighbor.

The neighbors are taken from a set of objects for which the class (for *k*-NN classification) or the object property value (for *k*-NN regression) is known. This can be thought of as the training set for the algorithm, though no explicit training step is required.

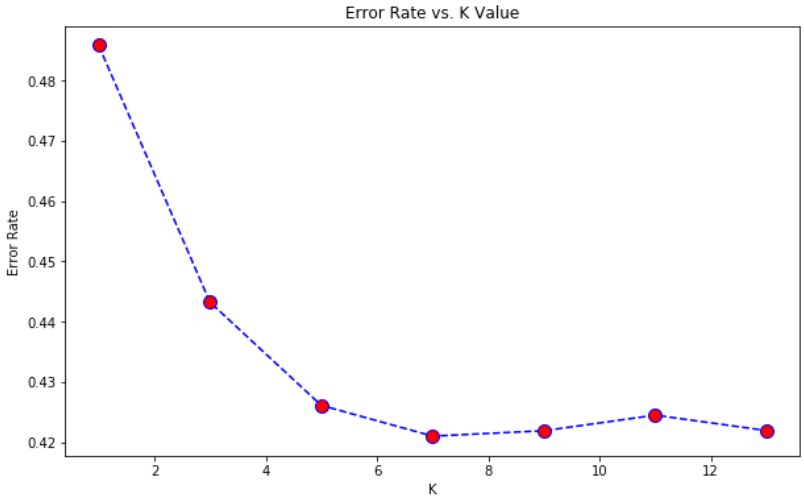


Parameter Selection:

The best choice of *k* depends upon the data; generally, larger values of *k* reduces effect of the noise on the classification,[5] but make boundaries between classes less distinct. A good *k* can be selected by various heuristic techniques. The special case where the class is predicted to be the class of the closest training sample is called the nearest neighbor algorithm.

The accuracy of the *k*-NN algorithm can be severely degraded by the presence of noisy or irrelevant features, or if the feature scales are not consistent with their importance. Much research effort has been put into selecting or scaling features to improve classification.

For our model, we did a grid search for k from 0 to 15 and got best accuracy for k = 7.



Results:

Although we tuned parameters to get the best accuracy possible, we were only able to achieve an accuracy score of 0.578.

```
In [11]: knn = KNeighborsClassifier(n_neighbors=optimal_k, n_jobs=8)
         knn.fit(X_train, y_train)
         pred = knn.predict(X_test)

In [12]: print("Accuracy for KNN model with Bag of words is ",round(accuracy_score(y_test ,pred),3))

         Accuracy for KNN model with Bag of words is  0.578
```

Logistic Regression, Decision Tree and Support Vector Machines

In this we take 100k of training data and separately 100k of test data. we have first done preprocessing of data. We have removed punctuation, extra spaces using replace function in python, stop words using NLTK, lemmatize text using NTLK.

We then applied different classification algorithm including logistic regression, decision tree, Support Vector machine.

At first, we have cleaned the reviews by first removing punctuation, missing values, etc. We have done cleaning of data using mostly python library replace.

```
df_cleaned['star_rating']=df_cleaned['star_rating'].astype(float).astype(int)
df_cleaned['review_body']=df_cleaned['review_body'].str.replace('<br />', r'')
df_cleaned['review_body']=df_cleaned['review_body'].str.replace("[,'!;.-`:?><-]",'')
#df_cleaned['review_body']=df_cleaned['review_body'].str.replace("[!#$%&'()*+, -./:;<=>?@[\]^_`{|}~]",'')
df_cleaned['review_body']=df_cleaned['review_body'].str.lower()
#df_cleaned=df_cleaned[df_cleaned.review_body.str.split(' ').str.len()<300]
df_cleaned['review_body']=df_cleaned.review_body.str.replace('[^a-zA-Z ]', '')
df_cleaned.dropna(inplace=True)
df_cleaned.isna().sum()
```

After preprocessing, we applied logistic regression, decision tree, and support vector machine with different parameters to see which works best. In the table below we have mentioned the

algorithm, their corresponding parameter and accuracy on test data. By observing the table below, we can see among all the algorithms, logistic regression with parameters(max_iter=500, C=0.1, random_state=40, solver='newton-cg')is giving the highest accuracy 0.864429 on test. One of the reasons, is because here I am using Bag of word with ngram approach to convert text into vector. Bag of word using ngram is implemented using Countvectorizer in sklearn library. Bag of word concept is same, It counts how many time a word is appearing in a review and put the count value in a vector accordingly, This approach combine with n grams works better than simple using CountVectorizer because it retrain some context among the sentence. The other approach that gives the highest accuracy was tokenizing data using tfidf and implementing SVM. Tfidf abbreviated for term frequency inverted document frequency is a technique in which it is found and counted of how important a word to a text in a large corpus of text.

Algorithm	Vectorizer	Parameters	Accuracy on test
Logistic Regression	Bow: CountVectorizer()	max_iter=2000	0.8359
Logistic Regression	Bow: CountVectorizer() Analyzer='word' Ngram_range=(1,2)	Max_iter=500, C=0.1, Random_state=40, Solver='newton-cg'	0.8644
Logistic Regression	Tfidf: TfidfVectorizer()	Max_iter=500, C=0.1, random_state=40, Solver='newton-cg'	0.8216
Logistic Regression	Tfidf: TfidfVectorizer()	Max_iter=500, C=0.1, random_state=40, solver='newton-cg'	0.8050
Decision Tree	Tfidf: TfidfVectorizer()	criterion='gini', min_samples=2	0.7379
Decision Tree	Bow: CountVectorizer()	criterion='gini', min_samples=2	0.7503
Decision Tree	Bow: CountVectorizer()	Criterion='gini'	0.7608
Decision Tree	Bow: CountVectorizer()	Criterion='entropy'	0.7385
Decision Tree	Bow: CountVectorizer()	Max_depth=10, max_leaf_nodes=30	0.8026
Decision Tree	Tfidf: TfidfVectorizer()	Default	0.7397
SVM	Bow:CountVectorizer()	Default	0.815
SVM	Tfidf: TfidfVectorizer()	Default	0.8519

After removing stopwords:

We then removed stop words from my reviews and retrain some of the algorithm to see if there is any improvement in accuracy. I removed the stops using NLTK libraray and using the module "stopwords".

```
df_cleaned['review_body']=df_cleaned['review_body'].apply(lambda x: ' '.join([item for item in x.split() if item not in stop]))
```

We observed that by removing stop words there was not an improvement in accuracy for logistic regression. It slightly decreased. For example, previously, for the same parameter for Logistic Regression it was 0.8349 but with using stop words it became 0.8345. However, for Decision Tree it increased from 0.7608 to 0.770068. So, all together, removal of stop words did not impact the accuracy.

Algorithm	Vectorizer	Parameters	Accuracy on test
Logistic Regression	Bow:CountVectorizer()	max_iter=2000	0.8345
Logistic Regression	Bow:CountVectorizer() analyzer='word', ngram_range=(1,2)	Max_iter=500, C=0.1, randome_state=40, solver='newton-cg'	0.8565
Decision Tree	Bow:CountVectorizer() analyzer='word', ngram_range=(1,2)	Criterion='gini'	0.7700
SVM	Tfidf: TfidfVectorizer()	Default	0.8482

After Lemmatization:

Lemmatization is a linguistic concept for grouping words into its lemma. It is commonly used in text processing Natural Language Processing. We implemented it using NLTK library. Following is the code

```
import nltk
nltk.download('wordnet')
def get_lemmatized_text(corpus):
    lemmatizer = WordNetLemmatizer()
    for i in range(0,len(corpus)):
       corpus.iloc[i] =' '.join([lemmatizer.lemmatize(word) for word in corpus.iloc[i].split()])
get_lemmatized_text(df_cleaned['review_body'])
```

Algorithm	Accuracy on test
Logistic Regression	0.86309
Decision Tree	0.7616
SVM	0.85058

Even after lemmatizing words in the reviews there was not much change in the accuracies.

Approach 3: Custom Built Naive Bayes

Bayes Theorem in Sentiment Analysis:

The basic idea of Naive Bayes technique is to find the probabilities of classes assigned to texts by using the joint probabilities of words and classes.

Given the dependent feature vector ***(x₁, …, xn)*** and the class ***Ck***. Bayes' theorem is stated mathematically as the following relationship:

$$P(C_k \mid x_1, \ldots, x_n) = \frac{P(C_k)P(x_1, \ldots, x_n \mid C_k)}{P(x_1, \ldots, x_n)}$$

According to the "naive" conditional independence assumptions, for the given class *Ck* each feature of vector *xi* is conditionally independent of every other feature *xj* for *i≠j*..

$$P(x_i \mid C_k, x_1, \ldots, x_n) = P(x_i \mid C_k)$$

Thus, the relation can be simplified to,

$$P(C_k \mid x_1, \ldots, x_n) = \frac{P(C_k)\prod_{i=1}^{n} P(x_i \mid C_k)}{P(x_1, \ldots, x_n)}$$

Since $P(x_1, \ldots, x_n)$ is constant, if the values of the feature variables are known, the following classification rule can be used:

$$P(C_k \mid x_1, \ldots, x_n) \propto P(C_k)\prod_{i=1}^{n} P(x_i \mid C_k)$$

$$\Downarrow$$

$$\hat{y} = \arg\max_k P(C_k)\prod_{i=1}^{n} P(x_i \mid C_k)$$

To avoid underflow, log probabilities are used.

$$\hat{y} = \arg\max_k (\ln P(C_k) + \sum_{i=1}^{n} \ln P(x_i \mid C_k))$$

The variety of naive Bayes classifiers primarily differs between each other by the assumptions they make regarding the distribution of ***P(xi|Ck)***, while ***P(Ck)*** is usually defined as the relative frequency of class ***Ck*** in the training dataset.

The multinomial distribution is parametrized by vector $\vartheta k=(\vartheta k1,\ldots,\vartheta kn)$ for each class ***Ck***, where ***n*** is the number of features (i.e. the size of the vocabulary) and ϑki is the probability ***P(xi|Ck)*** of feature ***i*** appearing in a sample that belongs to the class ***Ck***.

The parameters ϑk is estimated by a smoothed version of maximum likelihood, i.e. relative frequency counting:

$$\hat{\theta}_{ki} = \frac{N_{ki} + \alpha}{N_k + \alpha n}$$

where ***Nki*** is the number of times feature ***i*** appears in a sample of class ***k*** in the training set ***T***, and ***Ny*** is the total count of all features for class ***Ck***.

The smoothing priors ***α≥0*** accounts for features not present in the learning samples and prevents zero probabilities in further computations. Setting ***α=1*** is called Laplace smoothing, while ***α<1*** is called Lidstone smoothing. Thus, the final decision rule is defined as follows:

$$\hat{y} = \arg\max_{k}(\ln P(C_k) + \sum_{i=1}^{n} \ln \frac{N_{ki} + \alpha}{N_k + \alpha n})$$

Preprocessing data

Texts generated by humans in social media sites contain lots of noise that can significantly affect the results of the sentiment classification process. Moreover, depending on the feature's generation approach, every new term seems to add at least one new dimension to the feature space. That makes the feature space more sparse and high-dimensional. Consequently, the task of the classifier has become more complex. To prepare messages, such text preprocessing techniques as replacing URLs and usernames with keywords, removing punctuation marks and converting to lowercase were used in this program.

```
def ProReviews(review):
    review = ''.join(c for c in review if c not in string.punctuation)
    review = re.sub('((www\S+)|(http\S+))', 'urlsite', review)
    review = re.sub(r'\d+', 'contnum', review)
    review = re.sub(' +',' ', review)
    review = review.lower().strip()
    return review

def rmStopWords(review, stop_words):
    text = review.split()
    text = ' '.join(word for word in text if word not in stop_words)
    return text
```

Naïve Bayes Model:

After preprocessing the data, it is split into train and test data with a ratio of 3:1. A custom naïve Bayes classifier is defined which takes one input, i.e. train data.

Based on the ratings, lists of positive, negative and neutral words are created from the review text. Then, prior probabilities of all the three classes are calculated by dividing the respective length of classes to the total length of unique words in the training data.

And word count for each class is calculated.

Once the prior probabilities and words counts are calculated, the model is ready for predicting class for new reviews.

Prediction is done based on the probability on the number of positive, negative and neutral words in the review and the prior probabilities of those classes. Whichever the probability is high, the review is classified into that respective class.

Results:

On the test data with 100k reviews, Naïve Bayes model has achieved an accuracy score of 0.909.

```
In [12]: rnb = AmazonNBClassifier(df_train)
         rnb = rnb.fit()
         print ('training complete')
         predict = rnb.predict(df_test)
         score = rnb.score(predict,df_test.ratings.tolist())
         print(score)

         training complete
         0.9090909090909091
```

Approach 4: Handling Big Data

So far in all our approaches we have used batches of data instead of our entire dataset. But in this approach, we talk about generating predictions to our entire dataset.

The first method of how we have implemented this approach is by creating a custom loop which 'pickles' a logistic regression model over 5 batches of data and generates predictions. These predictions are then stored in Pandas DataFrame.

```
count = 1
sample_predictions_list_final = []
loaded_models_final = []
for i in data_list:
    X = i['review_body']
    y = i['star_rating']
    X_train, X_test, y_train, y_test = train_test_split(X, y,test_size=0.2, random_state=42)
    classifier = LogisticRegression(multi_class='auto')
    pipe = Pipeline([('cleaner', predictors()),
                     ('tfidfVect', tfdifVect),
                     ('classifier', classifier)])
    pipe.fit(X_train,y_train)
    filename = str(count) + 'sentimental_model.sav'
    pickle.dump(pipe, open(filename, 'wb'))
    loaded_models_final.append(pickle.load(open(filename, 'rb')))
    sample_predictions = loaded_models_final[count-1].predict(X_test)
    sample_predictions_list_final.append(sample_predictions)
    print(count)
    print("Accuracy for the Batch:", pipe.score(X_test,y_test))
    count += 1
```

This approach gives us 5 different predictions for different batches with our highest accuracy reaching 90%.

The second method is by implementing Pyspark. Pyspark is the API for APACHE Spark, used for handling big data. We have used this since our entire reviews was very big and we need this technology to process our large dataset. We have used CountVectorizer (Bag of word) for converting the tokens of reviews headline into numerical feature and the baseline model of Logistic Regression to predict the classes. Below is the snippet of code for modelling Logistic Regression.

```
[ ]  1 result.printSchema()

root
 |-- star_rating: integer (nullable = true)
 |-- tokenized_headlines: string (nullable = true)
 |-- features: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- tok_features: vector (nullable = true)
```

```
[ ]  1 train, test = result.randomSplit([0.9, 0.1], seed=12345)
```

```
[ ]  1 from pyspark.ml.classification import LogisticRegression
     2 lr = LogisticRegression(labelCol="star_rating", featuresCol="tok_features",maxIter=10)
     3 model=lr.fit(train)
     4 predict_train=model.transform(train)
```

By implementing this model, we were able to achieve 64.3935 accuracy on our test data

Conclusion

Overall, implementing logistic regression (0.89605) using SPACY provided the highest accuracy and the least compilation time and it was the most feasible machine learning model for Approach 1. We have also implemented ensemble learning and though Random Forest did provide a higher accuracy (0.88475) as compared to Decision Trees (0.84955), it did not improve the overall performance.

In approach 2, amongst Logistic regression, Decision Tree and Support vector the highest accuracy of 0.8644 was achieved when Bag of words were implemented using Countvectorizer() with ngram=(1,2) with Logistics Regression having parameters maximum iteration=500, C=0.1, random_state=40, and solver='newton-cg.' Whereas the KNN model is comparatively faster but was only able to achieve a score of 0.578.

Our custom function of Naïve Bayes implementation in approach 3 was able to achieve highest accuracy score of 0.909 among all the models and in all approaches. But it is highly time complex and took almost a day to train on 200k samples and predict 100k reviews.

In our last approach we find ways to deal with our dataset which consists of more than 3 million records of books and reviews. Using Pyspark, we trained logistic regression and was able to achieve 66.667% accuracy on our test data and using a custom loop we generated batch wise accuracies with one of the batches hitting an accuracy of above 90%.

References:

https://medium.com/@yoni.levine/how-to-grid-search-with-a-pipeline-93147835d916

https://becominghuman.ai/ensemble-learning-bagging-and-boosting-d20f38be9b1e

https://machinelearningmastery.com/large-data-files-machine-learning/

https://towardsdatascience.com/countvectorizer-hashingtf-e66f169e2d4e

https://blog.dask.org/2019/01/13/dask-cudf-first-steps

http://ijcsit.com/docs/Volume%206/vol6issue06/ijcsit2015060652.pdf

https://www.learndatasci.com/tutorials/sentiment-analysis-reddit-headlines-pythons-nltk/http://cs229.stanford.edu/proj2017/final-reports/5163147.pdf

https://www.analyticsvidhya.com/blog/2015/01/scikit-learn-python-machine-learning-tool/

https://www.kdnuggets.com/2016/07/support-vector-machines-simple-explanation.html

https://pypi.org/project/spacy/

https://analyticsindiamag.com/primer-ensemble-learning-bagging-boosting/

https://medium.com/greyatom/decision-trees-a-simple-way-to-visualize-a-decision-dc506a403aeb