# Review Classifier

So the product that I created classifies reviews of a product simply being into a positive or a negative review. So on a broader perspective it’s a problem of binary classification.

Since we are working with linguistic data, We have used NLP(Natural Language Processing) for that purpose.

### NLP

NLP stands for Natural Language Processing. It is a subfield of computer science and artificial intelligence that deals with the interaction between computers and humans using natural language. NLP is used to analyse, understand, and generate human language. It is particularly useful in text analysis and sentiment analysis, which are important for the task of review classification.

Now in this project the source files are :-

- Google colab - review_classifier.ipynb ()
- Html - index.html
- Application - app.py

### We’ll be staring with explaining the code of first file :-

1. **‘review_classifier.ipynb’**

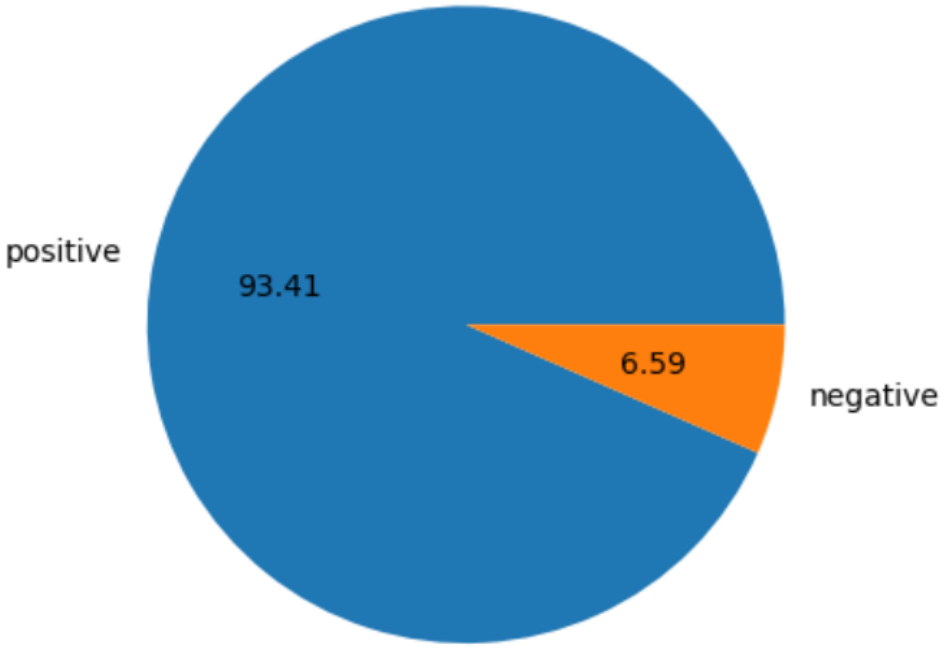
Loaded the two datasets

```
import numpy as np
import pandas as pd

df1 = pd.read_csv('amazon_reviews.csv')
df2 = pd.read_csv('Dataset-SA.csv')
```

```
import matplotlib.pyplot as plt
plt.pie(df1_train['review_type'].value_counts(),labels = ['positive','negative'],autopct="%0.2f")
plt.show()
```

So we started with analysing the first dataset and then we plotted a pie chart of the dataset we got




This pie chart clearly indicates that the data is highly skewed.

To remove this skewness we took some data points from the second data frame and added those data points into the first data frame to create a new data frame called as ‘df_train’.

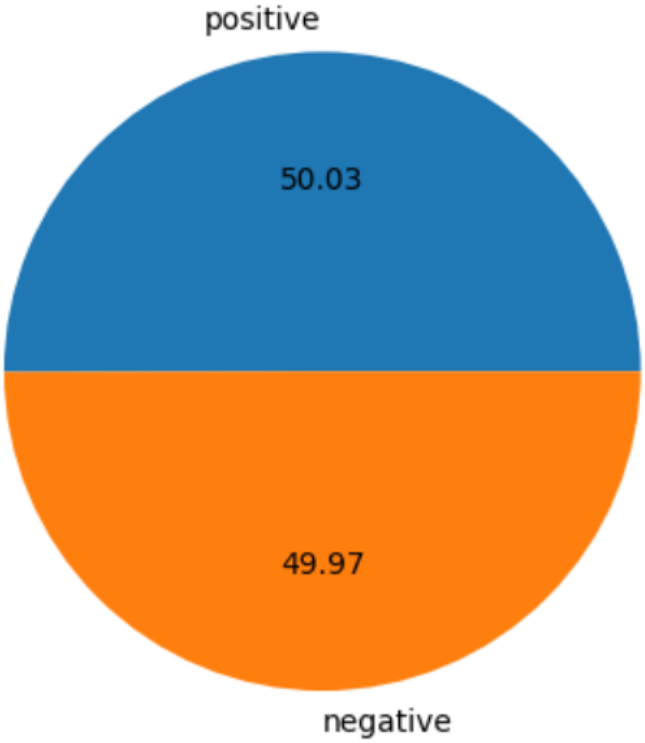
```
df_train = pd.concat([df1_train,positive_sentiments1,negative_sentiments],axis = 0)
```

```
df_train.shape
```

```
(57147, 2)
```

Now the finally we again plotted the pie chart we got the result as shown below

```
plt.pie(df_train['review_type'].value_counts(),labels = ['positive','negative'],autopct="%0.2f")
plt.show()
```




Now we have data free of skewness , therefore we can proceed with this dataset.

Now moving forward first we have to pre-process the dataset before providing it ML models

```
from nltk.corpus import  stopwords
import string
from nltk.stem import PorterStemmer
from nltk.stem import WordNetLemmatizer



def transform_text(text):
  text = text.lower()
  text = nltk.word_tokenize(text)
  y = []
  for i in text:
    if i.isalnum:
      y.append(i)

  text = y[:]
  y.clear()

  for i in text:
    if i not in stopwords.words('english') and i not in string.punctuation:
      y.append(i)

  text = y[:]
  y.clear()
  lemmatizer = WordNetLemmatizer()
  for i in text:
    y.append(lemmatizer.lemmatize(i))
  return " ".join(y)
```

So the general pre-processing steps are :-

- Convert all the words into lowercase.
- Remove all the charachters that are not alpha-numeric.
- Remove the stopwords.
- Perform stemming or lemmatization on the text

## Vocabulary

1. **Stopwords**

   Stopwords are words which do not contain important meaning and are usually removed from texts before processing. Examples of stopwords include "the", "a", "an", "in", "and", "of", etc.

2. **Stemming**

   Stemming is the process of reducing words to their base or root form. For example, the words "running", "runs", and "ran" would all be reduced to the stem "run". This is useful for reducing the number of unique words in a dataset and for simplifying text analysis. There are several algorithms that can be used for stemming, such as the Porter stemming algorithm and the Snowball stemming algorithm.

3. **Lemmatization**

   Lemmatization is the process of reducing words to their base or dictionary form, which is known as the lemma. For example, the words "am", "are", and "is" would all be reduced to the lemma "be". This is useful for reducing the number of unique words in a dataset and for simplifying text analysis. Unlike stemming, lemmatization takes into account the context and part of speech of the word, which can result in more accurate and meaningful lemmas. However, lemmatization can be slower and more computationally expensive than stemming.

Then the part comes of training the ML models

Now we have used ‘Count Vectorizers’ and ‘TF-idf’ to convert our text into numerical data

```
from sklearn.feature_extraction.text import CountVectorizer,TfidfVectorizer
cv = CountVectorizer(max_features = 10625)
tfidf = TfidfVectorizer(max_features=3000)

X = cv.fit_transform(df_train['transformed_text']).toarray()
```

1. **Count Vectorizer**

   Count Vectorizer is a technique for converting text into numerical data, which can be used as input for machine learning models. It works by counting the frequency of each word in a document and creating a vector of those counts. Each document is represented by a vector of counts, and the entire dataset can be represented as a matrix of count vectors. This matrix can then be used as input for machine learning models, such as Naive Bayes or Support Vector Machines (SVMs). The Count Vectorizer technique is simple and effective, but it does not take into account the importance of each word in a document. Therefore, it may not be as effective for tasks such as sentiment analysis, where the meaning of a sentence can depend on the context and the importance of each word.

2. **TF-idf**

   TF-idf stands for term frequency-inverse document frequency. It is a technique for converting text into numerical data, which can be used as input for machine learning models. It works by counting the frequency of each word in a document and then scaling the counts by the inverse frequency of each word in the entire dataset. This scaling helps to give more weight to words that are important for a particular document, while de-emphasizing words that are common across all documents. The TF-idf technique is more effective than the Count Vectorizer technique for tasks

such as sentiment analysis, where the meaning of a sentence can depend on the context and the importance of each word.

$$TF - IDF = TermFrequency(TF) * InverseDocumentFrequency(IDF)$$

**Term Frequency**

This measures the frequency of a word in a document.

**tf(t,d) = count of t in d / number of words in d**

**Document Frequency**

This measures the importance of documents in a whole set of the corpus.

**df(t) = occurrence of t in N documents**

**Inverse Document Frequency**

IDF is the inverse of the document frequency which measures the informativeness of term t.

**idf(t) = N/df**

**idf(t) = log(N/(df + 1))**

**Final Equation**

$$tf - idf(t,d) = tf(t,d) * log(N/(df + 1))$$

## Model Training

Now the time comes to train the model.

So for training we used Naive Bayes Algorithm since it works best over the text data

```
from sklearn.naive_bayes import GaussianNB,MultinomialNB,BernoulliNB
from sklearn.metrics import accuracy_score,confusion_matrix,precision_score
gnb = GaussianNB()
mnb = MultinomialNB()
bnb = BernoulliNB()
```

**Naive Bayes**

Naive Bayes is a machine learning algorithm that is commonly used for classification tasks, particularly in text analysis. It works by calculating the probability of a particular class (e.g. positive or negative sentiment) given a set of input features (e.g. the words in a text document), using Bayes' theorem. Naive Bayes assumes that the input features are independent of each other, which allows it to work well with high-dimensional datasets such as text data. Naive Bayes is often used for sentiment analysis, spam detection, and document classification, among other tasks.

1. **Gaussian Naive Bayes**

   In Gaussian Naive Bayes, continuous values associated with each feature are assumed to be distributed according to a **Gaussian distribution**. A Gaussian distribution is also called **Normal distribution**.

   ```
   gnb.fit(X_train,y_train)
   y_pred1 = gnb.predict(X_test)
   print(accuracy_score(y_test,y_pred1))
   print(confusion_matrix(y_test,y_pred1))
   ```

```
print(precision_score(y_test,y_pred1))

#Results
0.6072615923009623
[[5412  296]
 [4193 1529]]
0.8378082191780822
```

2. **Multinomial Naive Bayes**

   Feature vectors represent the frequencies with which certain events have been generated by a **multinomial distribution**. This is the event model typically used for document classification.

```
mnb.fit(X_train,y_train)
y_pred2 = mnb.predict(X_test)
print(accuracy_score(y_test,y_pred2))
print(confusion_matrix(y_test,y_pred2))
print(precision_score(y_test,y_pred2))

#Results
0.8825021872265967
[[4895  813]
 [ 530 5192]]
0.8646128226477935
```

3. **Bernoulli Naive Bayes**

   In the multivariate Bernoulli event model, features are independent booleans (binary variables) describing inputs. Like the multinomial model, this model is popular for document classification tasks, where binary term occurrence(i.e. a word occurs in a document or not) features are used rather than term frequencies(i.e. frequency of a word in the document).

```
bnb.fit(X_train,y_train)
y_pred3 = bnb.predict(X_test)
print(accuracy_score(y_test,y_pred3))
print(confusion_matrix(y_test,y_pred3))
print(precision_score(y_test,y_pred3))

#Results
0.8139982502187226
[[5302  406]
 [1720 4002]]
0.9078947368421053
```

## Saving the Model

In the end we used pickle module to save the model as ‘pkl‘ file.

```
import pickle
pickle.dump(bnb,open('model.pkl','wb'))
pickle.dump(cv,open('vectorizer.pkl','wb'))
```

## Deep Learning Aspect

I tried to use the LSTMs to train the model but in the end I got to know that it didn’t work out that well giving the accuracy of just 50%.

Basically in this model we use 3-D matrix to define the dataset in which the third dimension vectorize each word in a sentence making it even more detailed for the computer to understand a sentence.

To vectorize the words we used ‘glove’.

1. **LSTMs**

   LSTM (Long Short-Term Memory) is a type of recurrent neural network that is useful for processing and classifying sequential data, such as text. LSTMs are designed to overcome the problem of vanishing gradients in traditional recurrent neural networks, which can make it difficult to learn long-term dependencies in the data. LSTMs use a memory cell and a set of gates to selectively remember or forget information from previous time steps. This allows them to learn long-term dependencies in the data and make accurate predictions based on the context of the entire sequence. LSTMs have been used successfully in a variety of natural language processing tasks, including sentiment analysis and language modeling.

2. **Glove**

   GloVe stands for Global Vectors for Word Representation. It is an unsupervised learning algorithm for obtaining vector representations for words. These vector representations are designed to capture the meaning of words based on their co-occurrence statistics in large text corpora. GloVe is similar to other word embedding algorithms such as Word2Vec, but it has been shown to outperform them on many tasks. GloVe is often used in natural language processing tasks such as sentiment analysis and language modeling.

```
import numpy as np
from keras.preprocessing.text import Tokenizer
from keras.utils import pad_sequences
from keras.layers import Embedding, LSTM, Dense
from keras.models import Sequential
from keras.layers import Bidirectional,Dropout

# Load the GloVe embeddings
glove_file = 'glove.6B.300d.txt'
embedding_dim = 300
embeddings_index = {}
with open(glove_file, encoding="utf8") as f:
    for line in f:
        values = line.split()
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs

# Tokenize the text data
texts = df_train['transformed_text']
tokenizer = Tokenizer()
tokenizer.fit_on_texts(texts)
word_index = tokenizer.word_index

num_words = len(word_index) + 1
embedding_matrix = np.zeros((num_words, embedding_dim))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
# Pad the text sequences to a fixed length
maxlen = 10
sequences = tokenizer.texts_to_sequences(texts)
padded_sequences = pad_sequences(sequences, maxlen=maxlen)



#creating model
model_lstm = Sequential()
model_lstm.add(Embedding(len(embedding_matrix), embedding_dim, weights=[embedding_matrix], input_length=maxlen, trainable=False))
model_lstm.add(LSTM(128))
model_lstm.add(Dense(512, activation='relu'))
model_lstm.add(Dropout(0.50))
model_lstm.add(Dense(1, activation='softmax'))
```

```
# Adam Optimiser
model_lstm.compile(loss='binary_crossentropy',optimizer='adam', metrics=['accuracy'])


labels = df_train['review_type']
labels = labels.to_numpy()
labels = labels.reshape(-1,1)

model_lstm.fit(padded_sequences,labels,epochs = 25,batch_size = 500)

#Results
Epoch 1/25
115/115 [==============================] - 13s 96ms/step - loss: -105.3736 - accuracy: 0.5003
```

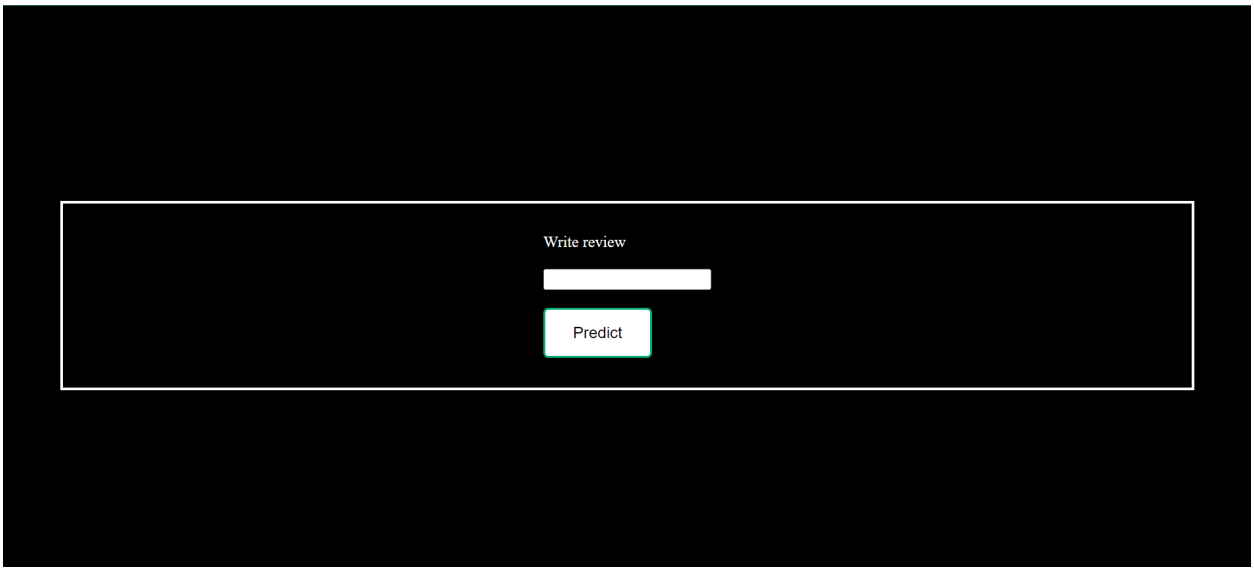
2. **‘index.html**’ (template folder)

Now this file contains the front-end of the website where I have created  form in which there is a text-input (for review) and ‘predict button’.

Styling is done inside the html file only

```
<body>
    <div class = "center-form">
    <form class = "form" action = "/predict" method = "POST">
      <label style = "color:white">Write review</label><br><br>
      <input type = "text" name = "review" class = "form-control"><br><br>
      <input type = "submit" value = "Predict" class = "button success">
    </form>
  </div>
</body>
```

Interface of the website




3. **‘app.py’**

This python file acts as a connector between the ‘review_classifier.ipynb’ file and ‘index.html’ file.

In this we have used Flask Framework and ‘render template’ function to render the ‘index.html’ file . And we also used ‘request’ to access the http request made by server.

### Flask

Flask is a lightweight web framework for Python that allows developers to easily build web applications. It provides a simple and intuitive API for handling HTTP requests and responses, as well as a templating system for rendering HTML pages. Flask is easy to learn and use, making it a popular choice for building web applications of all sizes.

### Render Template

The `render_template` function is a method provided by Flask that allows you to render a Jinja2 template. This function takes the name of the template file and any variables that you want to pass to the template as arguments. The template file should be located in the `templates` folder of your Flask application. The Jinja2 template language allows you to embed Python code in HTML templates, making it easy to generate dynamic web pages. You can use the `render_template` function to render HTML pages that include dynamic content generated by your Flask application.

```
def home():
    return render_template('index.html')
```

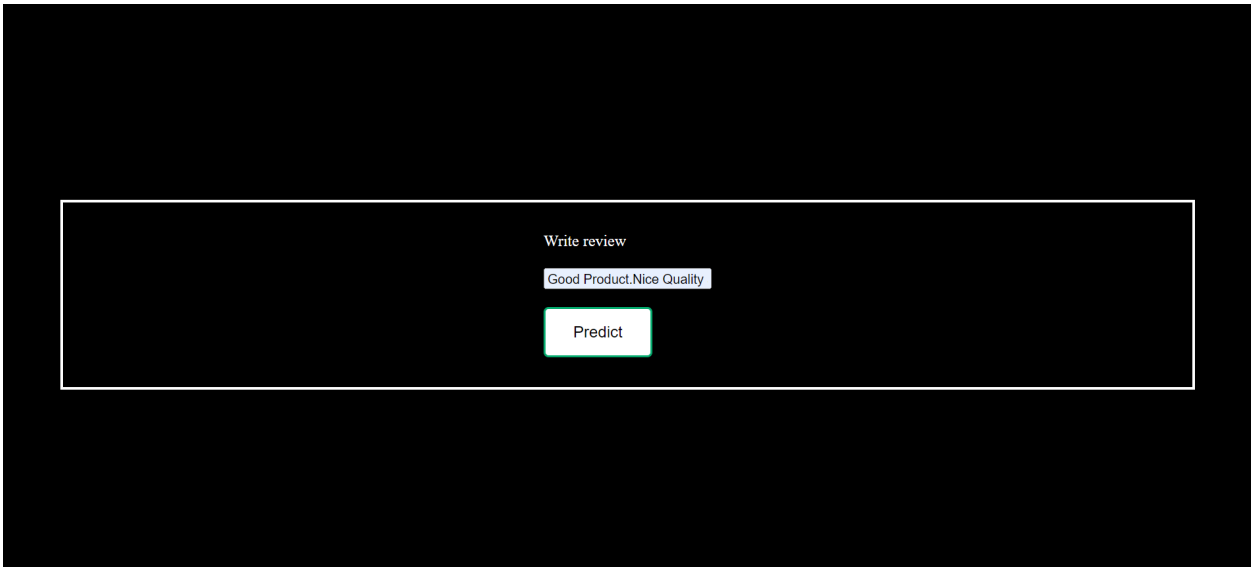
### Request

`request` is a module in the Flask library that allows you to access the HTTP request made by the client. This module provides various attributes and methods to access the data and information sent by the client in the request. Some of the commonly used methods and attributes in the `request` module include `request.form`, `request.args`, `request.files`, `request.method`, `request.cookies`, and `request.headers`. These methods and attributes can be used to extract data from the request and use it in your Flask application.

```
def predict():
    review = request.form.get('review')
    review = transform_text(review)
    vector_input = vectorizer.transform([review])
    vector_input = vector_input.reshape(1,-1)
    result1 = model.predict(vector_input)[0]
    result2 = multinomial_model.predict(vector_input)[0]
    if result2 == 1:
        return "positive review"
    else:
        return "negative review"
```

### Testing Results:

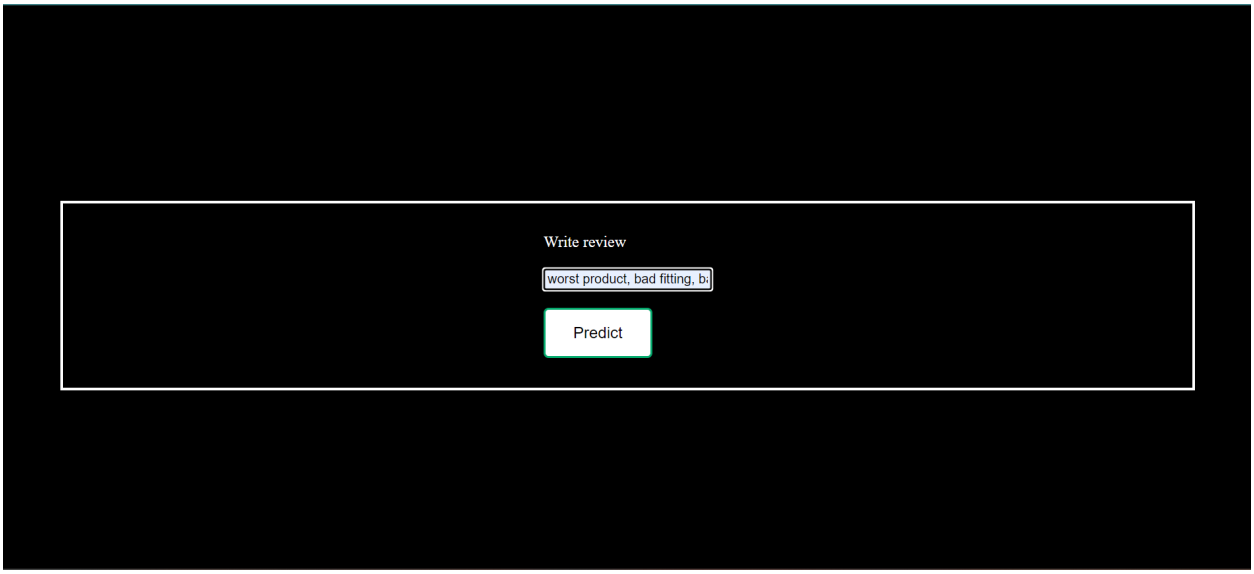
Review_1:




Result:

positive review

Review 2:




Result:

negative review

## Refrences

https://rb.gy/ayya3

https://rb.gy/ty1qg

https://rb.gy/kgl22

https://rb.gy/wsnf5

https://rb.gy/tt38h

https://rb.gy/5pjkp

https://rb.gy/opsp2