CS-464: Introduction to Machine Learning

Amazon Product Review Sentiment Analysis

Final Report

Group - 10

Mehmet Berk Türkçapar 21902570
Emir Türkölmez 21901626
M. Haider Akbar 21901132
Afşın Arpad Demirkasımoğlu 21903470
Övgüm Can Sezen 21902418

1. Introduction **3**
2. Background Research **3**
2.1. Word Embedding 3
2.2. Word2vec 3
3. Dataset Analysis **4**
4. Models **5**
4.1. Gaussian Naive Bayes Model 5
4.2. Logistic Regression 5
4.3. Fully Connected Layer 6
5. Implementation and Experiments **8**
5.1. Preprocessing 9
5.2. Dataset Split 10
5.3. Word2Vec Embeddings 11
5.4. Gaussian Naive Bayes 12
5.5. Logistic Regression 14
5.6. Fully Connected Layer 16
5.6.1. Activation Function 18
5.6.2. Learning Rate 19
5.6.3. Hidden Layer Size 20
5.6.4. Number of Hidden Layers 21
6. Evaluation **23**
7. Conclusion **25**

1. Introduction

Amazon.com is one of the most visited online retail websites. A high number of products are purchased every day by customers thus there are millions of product reviews. Sentiment analysis is the process of detecting positive or negative attitudes in a text. Sentiment analysis has become a crucial area for businesses. It helps businesses understand how customers feel about the product or the service. Analysis of customer reviews on Amazon enables businesses to fix or improve their product by reviewing the results of the analysis. As businesses see the data, their decisions on the product become more accurate. The sentiment analysis on products listed on amazon.com by various businesses should be done so that businesses can have a scientific foundation on how to change their product or service as well as how customers feel about their products or services.

In this project, our aim is to automate the process of classification of product reviews. We will be using a data set that contains approximately 586000 product reviews for different products from amazon.com. This project is significant for businesses because it creates a scientific base for their customer feedback.

2. Background Research

2.1. Word Embedding

A word embedding is a representation of words based on semantics aiming to have words that have similar meanings also have a similar representation. Word embeddings are learned models. The main approach is using a dense probability representation, where each word is represented by a real-valued vector in lower dimensionality than sparse word representations [1].

We considered two common word embeddings BERT and word2vec. We selected word2vec embedding as this was the embedding we come across our research most frequently and is intuitive for small text instances (the amazın review set is composed mostly of 1-to-2 sentences) [1].

2.2. Word2vec

Word2vec is a statistical method to generate a standalone word embedding from a BoW representation (more specifically continuous BoW, CBoW) of text. Additionally, word2vec allows vector mathematics to reflect the semantic qualities of the words, described by the famous King - man + woman = Queen example [1].

The word2vec embedding will be used in each sentiment classification model and the deep learning model will be selected with the expectancy of it using word2vec embedding as well [1].

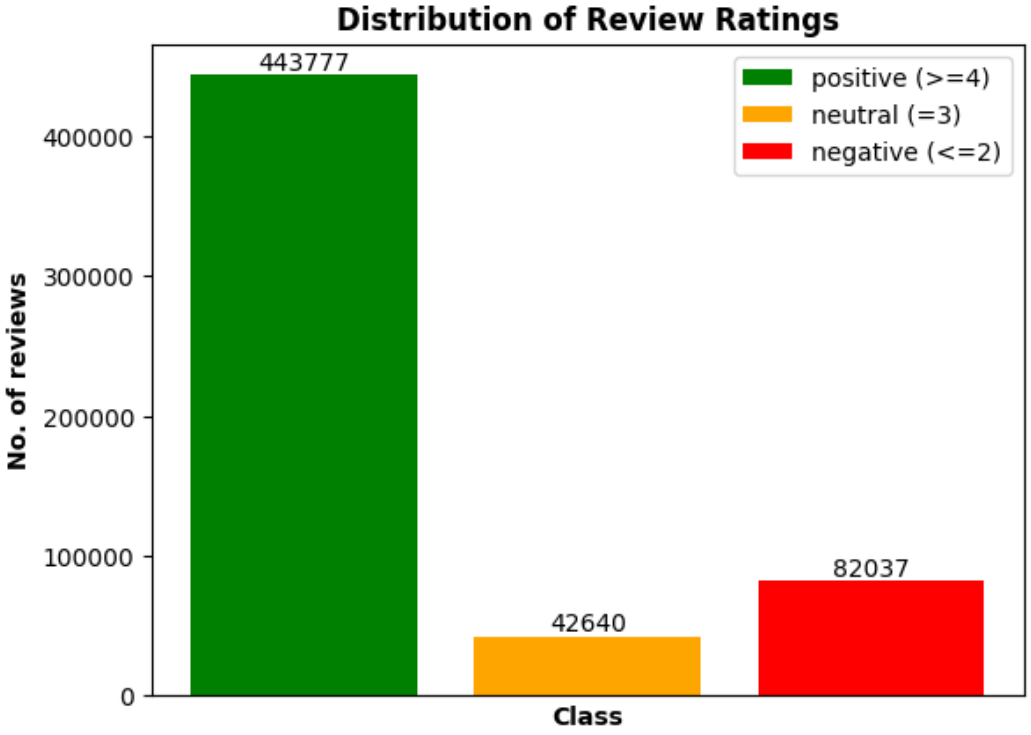
We plan to use a pre-trained mode, from the gensim library, for the word2vec algorithm so the details about how to train the model are not researched.

3. Dataset Analysis

The project involved analyzing a dataset of Amazon product reviews to perform sentiment analysis and gain insights into customer sentiment toward various products. The dataset contained over 568,000 reviews from multiple categories, including electronics, clothing, and home appliances. It is in CSV form and contains a lot of information for each review. This information includes the review text, the score, the review title, the product category, the review date, and more. For the purposes of this project, the review text and the score given (out of 5) will be important. We are planning to categorize each review based on the scores. Reviews with a score 5 or 4 will be considered as positive, with a score 3 will be considered neutral and with a score 2 or 1 will be considered as negative.

Our main focus is on review scores. So, we first checked for any non-integer values or values less than 0 in the review scores column. This was done to ensure that the data was consistent and accurate before performing any further analysis. Any such values were removed from the dataset to ensure that the analysis was not affected by any outliers or errors in the data.

After cleaning the review scores column, we plotted a histogram (Fig 1) to visualize the distribution of review scores in the dataset. The resulting graph revealed that the majority of reviews were positive, with a total of 443,777 reviews and then we have negative reviews of 82,037, and finally neutral reviews with a count of 42,540. The histogram clearly indicates an imbalanced distribution of reviews toward positive sentiment, which is a common challenge in machine learning. To address this issue we will be using F1 Score to measure the performance of our models. F1 Score takes into account both precision and recall and thus provides a better insight into the performance of our models.



4. Models

4.1. Gaussian Naive Bayes Model

Previously we were using Multinomial Naive Bayes. However, after consideration, we have switched to Gaussian Naive Bayes Model since our word2vec embeddings are continuous. Gaussian Naive Bayes is a classification algorithm that assumes the features follow a Gaussian distribution.

Gaussian Naive Bayes is well-suited for text classification tasks, including sentiment analysis, because it can handle a large number of features (words or phrases) efficiently. It assumes independence between the features, which aligns with the assumption that the occurrence of different words in a review is unrelated to each other. While this independence assumption may not be entirely accurate, it often holds reasonably well for sentiment analysis tasks.

Moreover, Gaussian Naive Bayes is computationally efficient and performs well with limited training data, making it suitable when dealing with a large number of Amazon product reviews. Since the algorithm estimates the parameters of the Gaussian distribution for each feature (word), it can effectively capture the probability distribution of sentiment-related words in positive and negative reviews.

However, it is important to acknowledge that the distribution of word frequencies in product reviews may not strictly follow a Gaussian distribution. Reviews often contain skewed or non-Gaussian patterns due to language nuances, slang, or specific domain-related vocabulary. In such cases, the Gaussian assumption may not be fully accurate, potentially leading to suboptimal results. Exploratory data analysis and model evaluation can help determine if the Gaussian assumption holds reasonably well for the given dataset or if alternative methods should be considered.

4.2. Logistic Regression

Logistic regression is a statistical algorithm used for binary classification problems, where the goal is to predict the probability of an instance belonging to a particular class. In the case of sentiment analysis, logistic regression can be employed to classify text as positive or negative based on the sentiment it conveys.

Logistic regression is based on the concept of logistic functions, also known as sigmoid functions, which map input data to a value between 0 and 1. These functions transform the linear regression output into a probability score that can be interpreted as the likelihood of belonging to a specific class.

In logistic regression, the goal is to find a set of weights (coefficients) denoted by **w** that minimizes the error between the predicted probabilities and the true labels in the training dataset. The logistic function, also known as the sigmoid function, is defined as:

$$Sigmoid(x) = 1 \div (1 + e^{-x})$$

Given a feature vector x and the corresponding weight vector w, the logistic regression model computes the linear combination of the features and weights:

$$z = w^T x$$

Here, w^T represents the transpose of the weight vector w.

The output of the logistic regression model is then obtained by applying the sigmoid function to the linear combination:

$$P(y = 1 \mid x, w) = Sigmoid(w^T x)$$
$$P(y = 2 \mid x, w) = Sigmoid(w^T x)$$
$$P(y = 2 \mid x, w) = Sigmoid(w^T x)$$

Where $P(y = 1)$ represents the possibility of belonging to class “positive”, $P(y = 2)$ represents possibility of belonging to the class “neutral” and $P(y = 1)$ represents the possibility of belonging to class “negative”.

Logistic regression offers several advantages when applied to sentiment analysis. It is a simple and interpretable algorithm, making it easy to understand and analyze the results. Additionally, logistic regression can handle large feature spaces and is computationally efficient. However, it has limitations in capturing complex relationships in the data, such as sentiment nuances or context dependencies. In such cases, more advanced techniques like neural network models may be considered.

4.3. Fully Connected Layer

A fully connected layer, also known as a dense layer, is a neural network layer where each neuron is connected to every neuron in the previous layer. It is a good choice for classification tasks because it can learn complex patterns and relationships in the input data, introduce non-linear transformations through activation functions, share parameters to improve generalization and efficiency and work well with backpropagation for effective training. In a fully connected neural network, several key concepts play crucial roles in determining its structure and behavior: activation function, learning rate, hidden layer size, and the number of hidden layers. In order to fine-tune it for our specific task we have experimented with different hyperparameters to achieve a more effective model.

An activation function is a mathematical function applied to the output of each neuron in a neural network. It introduces non-linearity into the network, enabling it to learn complex patterns and make more accurate predictions. Commonly used activation functions include the sigmoid, ReLU (Rectified Linear Unit), and tanh (hyperbolic tangent) functions. Each activation function has different properties and can affect the network's ability to learn and generalize from the data.

The learning rate is a hyperparameter that determines the step size at which the neural network adjusts its weights during training. It controls the speed and stability of the learning process. A higher learning rate allows for faster convergence, but it may also result in overshooting or instability. Conversely, a lower learning rate can provide more precise weight updates but may require more training iterations to achieve convergence. Selecting an appropriate learning rate is crucial for effective training and finding the optimal balance between speed and accuracy.

The hidden layer size refers to the number of neurons or units present in a hidden layer of the neural network. The size of the hidden layer impacts the network's capacity to learn and represent complex relationships within the data. Increasing the hidden layer size allows the network to capture more intricate patterns, but it also increases the number of parameters and the computational complexity of the model. Determining the optimal hidden layer size depends on the complexity of the problem and the amount of available training data. It often requires experimentation and validation to strike the right balance between model complexity and generalization ability.

The number of hidden layers refers to the depth of the neural network, i.e., the number of layers between the input and output layers. Deep neural networks with multiple hidden layers have the potential to learn hierarchical representations of the data, enabling them to extract more abstract and complex features. Adding more hidden layers can enhance the network's ability to capture intricate relationships in the data, leading to improved performance. However, excessively deep networks may suffer from vanishing or exploding gradients, which can hinder training. The appropriate number of hidden layers depends on the complexity of the task and the availability of training data, and it is typically determined through experimentation and performance evaluation.

In summary, the activation function introduces non-linearity, the learning rate controls the speed and stability of learning, the hidden layer size determines the network's capacity to learn complex patterns, and the number of hidden layers affects the depth and representation power of the network. Understanding these concepts and their implications is crucial for designing and training effective fully connected neural networks.

5. Implementation and Experiments

We have used Google Collab for effective teamwork and to be able to eliminate any development environment-based problems we may encounter. After opening and initializing the Google Collab environment, we uploaded the dataset to drive in order to use it on Google Collab. Next, we have mounted the drive to Google Collab. Also, we have imported and downloaded the necessary modules and packages.

```
from google.colab import drive
drive.mount("/content/gdrive/")
```

```
import os
import numpy as np
import pandas as pd
import string
import matplotlib.pyplot as plt
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize, punkt
from nltk.stem import SnowballStemmer
from sklearn.model_selection import train_test_split
import nltk
from sklearn.naive_bayes import MultinomialNB
import numpy as np
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix
from gensim.models import Word2Vec
from nltk.stem import PorterStemmer
nltk.download("punkt")
nltk.download('stopwords')
```

After that, we tested if we could reach the dataset correctly by printing some example data from the set. As the output shows, we were able to read data from the dataset.

```
dataset_dir = '/content/gdrive/MyDrive/CS-464: Introduction to Machine
Learning/Project/'
csv_path = os.path.join(dataset_dir, 'Reviews.csv')
df = pd.read_csv(csv_path)
df.head(3)
```

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenominator	Score	Time	Summary	Text
0	1	B001E4KFG0	A3SGXH7AUHU8GW	delmartian	1	1	5	1303862400	Good Quality Dog Food	I have bought several of the Vitality canned d...
1	2	B00813GRG4	A1D87F6ZCVE5NK	dll pa	0	0	1	1346976000	Not as Advertised	Product arrived labeled as Jumbo Salted Peanut...
2	3	B000LQOCH0	ABXLMWJIXXAIN	Natalia Corres "Natalia Corres"	1	1	4	1219017600	"Delight" says it all	This is a confection that has been around a fe...

For classification purposes, we have added a column named ‘Label’ based on the ‘Score’ column. If the ‘Score’ is more than or equal to 4 it is labeled ‘Positive’ if it is 3, ‘Neutral’ and if it is less than or equal to 2 it is labeled as ‘Negative’. We have again printed some sample data to show the labels were correct.

```
# Read in the CSV file and select only the required columns
df = pd.read_csv(csv_path, usecols=["Score", "Text"])

# Add a new 'Label' column based on the 'Score' column
df["Label"] = pd.cut(df["Score"], bins=[-float("inf"), 2, 3,
float("inf")], labels=["negative", "neutral", "positive"])
df.head()
```

5.1. Preprocessing

The data will be handled in a bag-of-words manner and word2vec embedding will be used for models. Prior to that, the text will be run through lowercase-conversion, punctuation removal, stop-word-removal, tokenization and stemming. The lowercase conversion is to standardize the tokens (words) in the text to overcome processing “Love” and “love” as different tokens. Punctuation removal is necessary for the standardization of the tokens for BoW representation as we do not want the punctuation to affect the tokens, similar to the idea of lowercase conversion. Stop word removal is the process of removing words that do not include any semantic information regarding the sentiment of the text, like the words “the”, “a”, and “in”. The existence of such words in the BoW may cause bias for the model as they are in comparison highly frequent to other words. Tokenization is the process of breaking the transformed text into words, the main step that turns the text into the BoW by separating it into units of words. Lastly, stemming is important in data preprocessing as it normalizes text by reducing words to their base form, improves text retrieval by matching variations of the same word, and reduces vocabulary size, which can enhance computational efficiency. The following is the code snippet in which all of this preprocessing was applied. Also, the following figures show our data with and without stemming.

```
stemmer = PorterStemmer()

# Remove punctuation
df['Text'] = df['Text'].apply(lambda x: x.translate(str.maketrans('', '', string.punctuation)))

# Convert all text to lowercase
df['Text'] = df['Text'].apply(lambda x: x.lower())

# Tokenize the text data
df['Text'] = df['Text'].apply(lambda x: word_tokenize(x))

# Remove stop words
stop_words = set(stopwords.words('english'))
df['Text'] = df['Text'].apply(lambda x: [word for word in x if word not in stop_words])

# Stemming
df['Text'] = df['Text'].apply(lambda x: [stemmer.stem(word) for word in x])
```

	Score	Text	Label
0	5	['bought', 'several', 'vitality', 'canned', 'd...	positive
1	1	['product', 'arrived', 'labeled', 'jumbo', 'sa...	negative
2	4	['confection', 'around', 'centuries', 'light',...	positive
3	2	['looking', 'secret', 'ingredient', 'robitussi...	negative
4	5	['great', 'taffy', 'great', 'price', 'wide', '...	positive

	Score	Text	Label
0	5	['bought', 'sever', 'vital', 'can', 'dog', 'fo...	positive
1	1	['product', 'arriv', 'label', 'jumbo', 'salt',...	negative
2	4	['confect', 'around', 'centuri', 'light', 'pil...	positive
3	2	['look', 'secret', 'ingredi', 'robitussin', 'b...	negative
4	5	['great', 'taffi', 'great', 'price', 'wide', '...	positive

5.2. Dataset Split

In this part, we did the train validation test split. Initially, we did a 80-20 split for train and test sets. Hence, we split the data and called 80% as train and 20% of it as test. Afterwards, we split the training part of the data again using a 90-10 split. Hence, 90% of the former training data is now considered as the new training data, and 10% is called the validation data. Then, we printed the train labels as well as the counts of the values (positive-neutral-negative). We obtained 319372 positive reviews, 30685 neutral reviews and 59229 negative reviews. Here is the part from the code from Google Collab showing our work for the split part:

```
# Split into train and test sets (80-20 split)
train_data, test_data, train_labels, test_labels =
train_test_split(df['Text'], df['Label'], test_size=0.2,
random_state=42)

# Split train data into train and validation sets (90-10 split)
train_data, val_data, train_labels, val_labels =
train_test_split(train_data, train_labels, test_size=0.1,
random_state=42)
```

5.3. Word2Vec Embeddings

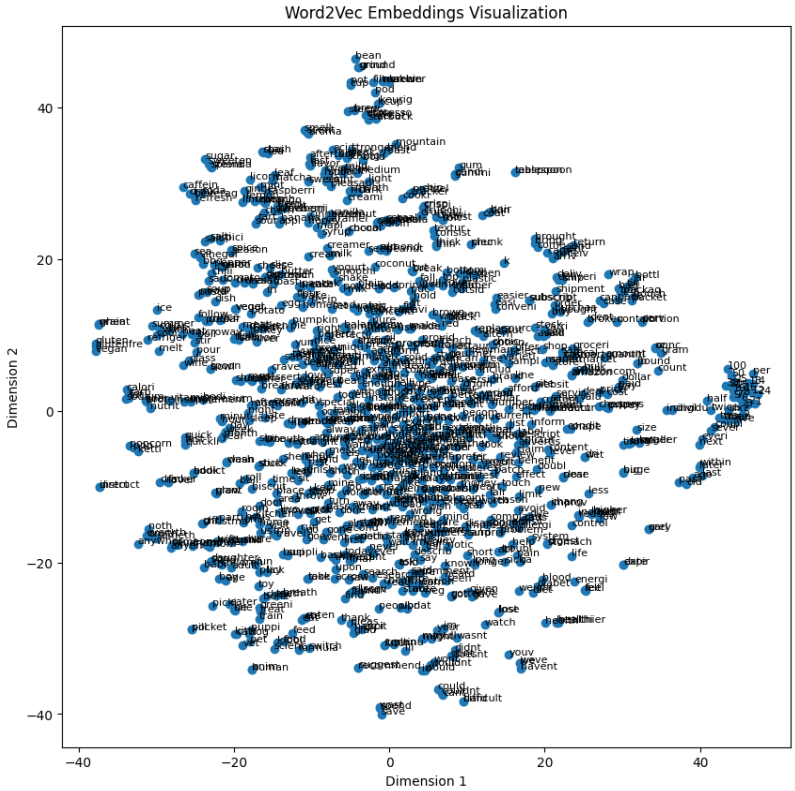
After the data split and preprocessing, we have used Word2vec to obtain the word embeddings. We have used Gensim's Word2vec function. We wrote a function in order to handle any unknown words. If we encounter a word that is not present in our vocabulary we have regarded is a default vector of 0's. Having this function we have obtained the values of texts by taking the mean of the embeddings of the words in the text. We have done this for both our test set and validation set.

```
vector_size = 100  # dimensionality of the word vectors

window_size = 5  # maximum distance between the current and predicted
word within a sentence

min_count = 5  # minimum frequency of a word to be included in the
vocabulary

workers = 4  # number of worker threads to train the model

# Train the Word2Vec model on the sentences

model = Word2Vec(sentences, vector_size=vector_size,
window=window_size, min_count=min_count, workers=workers)
```

Next for each train, validation and test data we have obtained the word2vec embeddings. We took the mean of embedding values of each word in a review to obtain its embedding. If an unknown word was encountered we added a default 0 vector as the embedding.

```
def get_embedding(word):
    try:
        return w2v_model.wv[word]
    except KeyError:
        # Return a vector of zeros if the word is not in the model vocabulary
        return np.zeros((100,))

# Transform the text data to Word2Vec embeddings
train_embeddings = np.array([np.mean([get_embedding(word) for word in words], axis=0) for words in train_data])
val_embeddings = np.array([np.mean([get_embedding(word) for word in words], axis=0) for words in val_data])
test_embeddings = np.array([np.mean([get_embedding(word) for word in words], axis=0) for words in test_data])
```

The following is a 2D visualization of the obtained Word2Vec embeddings. On the graph, clusters show words that have similar embeddings and thus similar meanings.



After this point as we had the word2vec embeddings for our dataset we were ready to train our models and use our validation sets to see how well they perform.

5.4. Gaussian Naive Bayes

It is clear to use the Gausian Naive bayes model by the instructions provided in sections 4.3. In short our data is continuous and it makes more sense to use Gaussian rather than multinomial naive bayes classification. As you can see from the implementation section, first we imported ‘GausianNB from sklearn.naive_bayes’. Next, the ‘fit()’ method is called on the GNB model to train it using the ‘train_embeddings’ and ‘train_labels data’. The ‘train_embeddings’ variable represents the Word2Vec embeddings of the training data, while train_labels contains the corresponding labels for each training sample. After training the model, the code proceeds to evaluate its performance on the validation data. The ‘predict()’ method is used to obtain predictions for the ‘val_embeddings’. The predicted labels are stored in the ‘val_predictions’ variable. To assess the model's performance, several

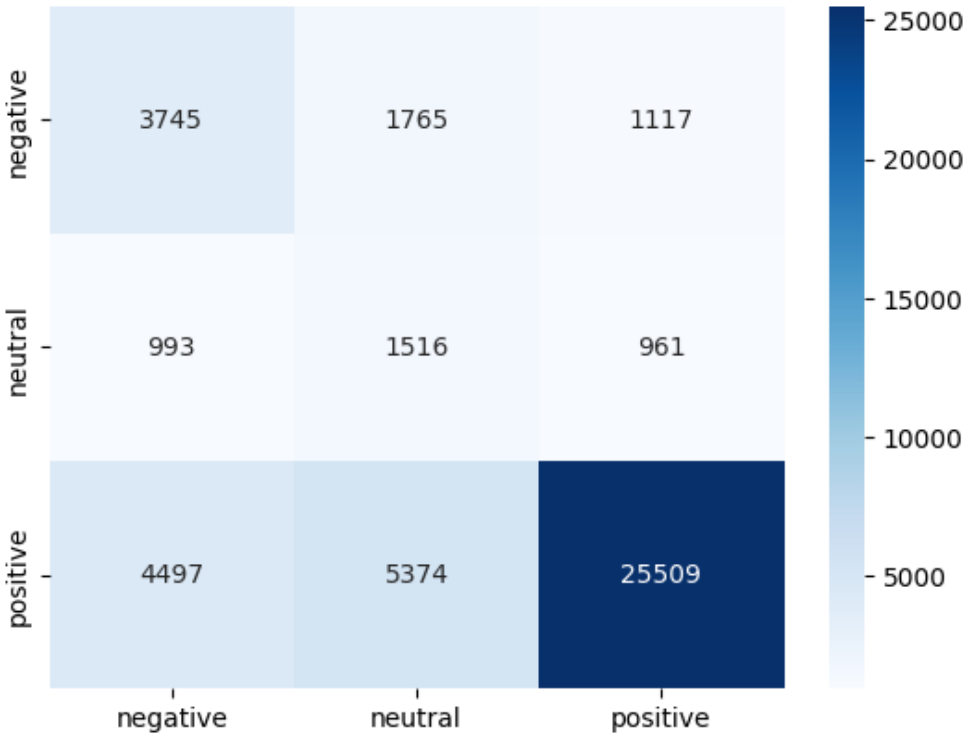
evaluation metrics are calculated. The ‘accuracy_score()’ function from ‘sklearn.metrics’ is used to compute the overall accuracy, comparing the predicted labels (val_predictions) with the true labels (val_labels) of the validation data. The resulting accuracy score is stored in the overall_accuracy variable.

```
# Train a Gaussian Naive Bayes model on the Word2Vec embeddings
gnb_model = GaussianNB()
gnb_model.fit(train_embeddings, train_labels)

# Evaluate the trained model on the validation data
val_predictions = gnb_model.predict(val_embeddings)
overall_accuracy = accuracy_score(val_labels, val_predictions)
f1 = f1_score(val_labels, val_predictions, average='weighted')
confusion_mat = confusion_matrix(val_labels, val_predictions)
```

Then we calculated the evaluation metrics using the confusion matrix.

```
# Print overall accuracy
print(f"Overall accuracy: {overall_accuracy:.4f}")

# Print the F1 score
print(f"F1 score: {f1:.4f}")

cm = confusion_matrix(val_labels, val_predictions)
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=['negative', 'neutral', 'positive']
            , yticklabels=['negative', 'neutral', 'positive'])
```

The model achieved an accuracy of 67.66% and an F1 score of 0.7182. These performance metrics provide an insight into the model's effectiveness in correctly classifying the validation data. The confusion matrix, presented below, offers a detailed breakdown of the model's predictions for each class. The Gaussian Naive Bayes model performed relatively better in predicting the minority classes (neutral and negative) since the imbalanced distribution of the classes had a lesser impact on their predictions. However, it struggled with the positive class due to the imbalance and the biased estimation of parameters.



5.5. Logistic Regression

In the provided code, the scikit-learn library is used to implement logistic regression for sentiment analysis using Word2Vec embeddings. The logistic regression model is trained by calling the `fit()` method on an instance of the LogisticRegression class, with the training embeddings and corresponding labels as inputs. The trained model is then used to make predictions on the validation data by calling the `predict()` method, generating the predicted labels. Evaluation metrics such as overall accuracy, weighted F1 score, and the confusion matrix can be calculated using functions from the sklearn.metrics module. This approach enables sentiment classification and performance assessment using logistic regression with the scikit-learn library. Here is the code snippet for the above explanation:

```
from sklearn.linear_model import LogisticRegression

# Train a Logistic Regression model on the Word2Vec embeddings
lr_model = LogisticRegression()
lr_model.fit(train_embeddings, train_labels)
```

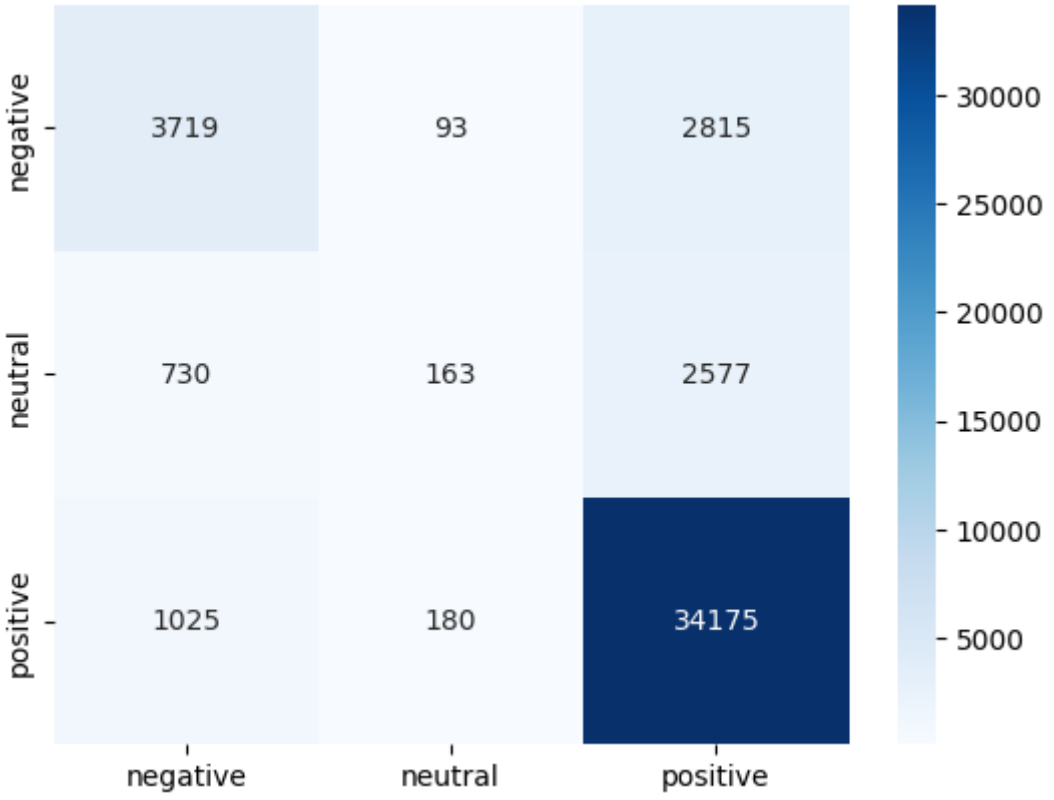
The evaluation of the trained logistic regression model on the validation data is done as follows. First, the `predict()` method is used on the logistic regression model (lr_model) with the validation embeddings (val_embeddings) to generate predictions (val_predictions) for the sentiment labels. The `accuracy_score()` function calculates the overall accuracy by comparing the predicted labels (val_predictions) with the actual labels (val_labels) of the validation data. The `f1_score()` function calculates the weighted F1 score, which considers both precision and recall, using the predicted and actual labels. Finally, the

`confusion_matrix()` function generates a confusion matrix, providing a detailed breakdown of the model's predictions in terms of true positive, true negative, false positive, and false negative instances. These evaluation metrics help assess the performance of the logistic regression model in sentiment classification on the validation data. Here is the code snippet for the above explanation:

```
# Evaluate the trained model on the validation data
val_predictions = lr_model.predict(val_embeddings)
overall_accuracy = accuracy_score(val_labels, val_predictions)
f1 = f1_score(val_labels, val_predictions, average='weighted')
confusion_mat = confusion_matrix(val_labels, val_predictions)
```

The confusion matrix, accuracy score and the F1 score for the evaluation data is as below:

```
Overall accuracy: 0.8368
F1 score: 0.8054
```



5.6. Fully Connected Layer

For the implementation of our neural network model we have used PyTorch. First we have imported the necessary libraries and encoded our labels.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from sklearn.preprocessing import LabelEncoder
# encode the labels as
# 0 -> negative
# 1 -> neutral
# 2 -> positive
label_encoder = LabelEncoder()
train_labels = label_encoder.fit_transform(train_labels)
val_labels = label_encoder.transform(val_labels)
test_labels = label_encoder.transform(test_labels)
```

Since the Word2vec embeddings we have obtained are 100 dimensional the input layer of the neural network we have trained had 100 nodes. We have experimented with a variety of different configurations and hyperparameters. Initially, we started with the following model and at each step changed only the specific configurations.

```
# Prepare the data
train_embeddings_torch = torch.tensor(train_embeddings, dtype=torch.float32)
train_labels_torch = torch.tensor(train_labels, dtype=torch.long)
val_embeddings_torch = torch.tensor(val_embeddings, dtype=torch.float32)
val_labels_torch = torch.tensor(val_labels, dtype=torch.long)

train_dataset = TensorDataset(train_embeddings_torch, train_labels_torch)
val_dataset = TensorDataset(val_embeddings_torch, val_labels_torch)

# Create data loaders
batch_size = 32
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)

# Define the model architecture
input_size = train_embeddings_torch.shape[1]
hidden_size = 128
num_classes = 3
model = nn.Sequential(
    nn.Linear(input_size, hidden_size),
    nn.ReLU(),
    nn.Linear(hidden_size, num_classes)
)

# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
# Train the model
num_epochs = 25
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

print("Activation Function: ReLU")
print("Learning Rate: 0.001")
for epoch in range(num_epochs):
    model.train()
    train_loss = 0.0
    for embeddings, labels in train_loader:
        embeddings = embeddings.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()

        outputs = model(embeddings)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        train_loss += loss.item()

    # Validate the model
    model.eval()
    val_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for embeddings, labels in val_loader:
            embeddings = embeddings.to(device)
            labels = labels.to(device)

            outputs = model(embeddings)
            loss = criterion(outputs, labels)
            val_loss += loss.item()

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    # Print training and validation metrics for each epoch
    print(f"Epoch {epoch+1}/{num_epochs} | Train Loss:
{train_loss/len(train_loader):.4f} | Val Loss:
{val_loss/len(val_loader):.4f} | Val Acc: {(correct/total)*100:.2f}%")

from sklearn.metrics import f1_score
val_predictions = []
with torch.no_grad():
```

```
    for embeddings, labels in val_loader:
        embeddings = embeddings.to(device)
        labels = labels.to(device)

        outputs = model(embeddings)
        _, predicted = torch.max(outputs.data, 1)
        val_predictions.extend(predicted.cpu().numpy())
val_labels = val_labels_torch.cpu().numpy()
f1 = f1_score(val_labels, val_predictions, average='weighted')
print(f"F1 Score: {f1:.4f}")
```

As can be seen from the code, the configurations which were the same throughout all trials were the number of epochs which was 25, batch size which was 32, loss function which was Cross Entropy Loss, and the optimizer which was Adam.

5.6.1. Activation Function

The first configuration we experimented with was the activation function. In this experimantation the Learning rate was 0.001, hidden layer size was 128 and number of hidden layers was 1. We have tested three different activation functions: ReLU, Sigmoid and Tanh. First, for the ReLU the results were as follows.

```
Epoch 20/25 | Train Loss: 0.3598 | Val Loss: 0.3954 | Val Acc: 85.50%
Epoch 21/25 | Train Loss: 0.3587 | Val Loss: 0.3945 | Val Acc: 85.38%
Epoch 22/25 | Train Loss: 0.3580 | Val Loss: 0.3896 | Val Acc: 85.75%
Epoch 23/25 | Train Loss: 0.3574 | Val Loss: 0.3901 | Val Acc: 85.47%
Epoch 24/25 | Train Loss: 0.3569 | Val Loss: 0.3925 | Val Acc: 85.59%
Epoch 25/25 | Train Loss: 0.3561 | Val Loss: 0.3914 | Val Acc: 85.63%
F1 Score: 0.8402
```

Second, for the Sigmoid the results were as follows.

```
Epoch 20/25 | Train Loss: 0.3596 | Val Loss: 0.3838 | Val Acc: 85.74%
Epoch 21/25 | Train Loss: 0.3585 | Val Loss: 0.3851 | Val Acc: 85.63%
Epoch 22/25 | Train Loss: 0.3572 | Val Loss: 0.3873 | Val Acc: 85.81%
Epoch 23/25 | Train Loss: 0.3562 | Val Loss: 0.3872 | Val Acc: 85.72%
Epoch 24/25 | Train Loss: 0.3549 | Val Loss: 0.3872 | Val Acc: 85.66%
Epoch 25/25 | Train Loss: 0.3540 | Val Loss: 0.3844 | Val Acc: 85.80%
F1 Score: 0.8387
```

Lastly, for the Tanh function the results were as follows.

```
Epoch 20/25 | Train Loss: 0.3767 | Val Loss: 0.3969 | Val Acc: 85.28%
Epoch 21/25 | Train Loss: 0.3762 | Val Loss: 0.3988 | Val Acc: 85.06%
Epoch 22/25 | Train Loss: 0.3755 | Val Loss: 0.3964 | Val Acc: 85.13%
Epoch 23/25 | Train Loss: 0.3750 | Val Loss: 0.3968 | Val Acc: 85.21%
Epoch 24/25 | Train Loss: 0.3745 | Val Loss: 0.3970 | Val Acc: 85.27%
Epoch 25/25 | Train Loss: 0.3742 | Val Loss: 0.3967 | Val Acc: 85.26%
F1 Score: 0.8286
```

To sum up, here is the F1 Scores obtained for each different activation functions.

Activation Function	F1 Score
ReLU	0.8402
Sigmoid	0.8387
Tanh	0.8286

Based on these results, it can be observed that the choice of activation function had a slight impact on the model's performance. The ReLU activation function achieved the highest F1 score, indicating that it was more effective in capturing the underlying patterns in the data. The sigmoid activation function also performed well, closely following ReLU in terms of F1 score. On the other hand, the tanh activation function yielded a slightly lower F1 score compared to the other two. These results suggest that ReLU and sigmoid activation functions are well-suited for this classification task, while the tanh activation function might be less suitable. Since it performed the best, we have used ReLU function as the activation function for the following experiments.

5.6.2. Learning Rate

The second parameter we experimented with was the learning rate. The learning rate refers to the step size to be taken during the optimization. In this experiment the activation function was ReLU, the hidden layer size was 128, and the number of hidden layers was 1. We have tested three different learning rates: 0.01, 0.001 and 0.0001. First, for the LR = 0.0001 the results were as follows.

```
Epoch 20/25 | Train Loss: 0.3799 | Val Loss: 0.3925 | Val Acc: 85.27%
Epoch 21/25 | Train Loss: 0.3788 | Val Loss: 0.3917 | Val Acc: 85.26%
Epoch 22/25 | Train Loss: 0.3779 | Val Loss: 0.3912 | Val Acc: 85.32%
Epoch 23/25 | Train Loss: 0.3769 | Val Loss: 0.3918 | Val Acc: 85.34%
Epoch 24/25 | Train Loss: 0.3759 | Val Loss: 0.3911 | Val Acc: 85.34%
Epoch 25/25 | Train Loss: 0.3752 | Val Loss: 0.3933 | Val Acc: 85.30%
F1 Score: 0.8344
```

Second, for the LR = 0.001 the results were as follows.

```
Epoch 20/25 | Train Loss: 0.3598 | Val Loss: 0.3954 | Val Acc: 85.50%
Epoch 21/25 | Train Loss: 0.3587 | Val Loss: 0.3945 | Val Acc: 85.38%
Epoch 22/25 | Train Loss: 0.3580 | Val Loss: 0.3896 | Val Acc: 85.75%
Epoch 23/25 | Train Loss: 0.3574 | Val Loss: 0.3901 | Val Acc: 85.47%
Epoch 24/25 | Train Loss: 0.3569 | Val Loss: 0.3925 | Val Acc: 85.59%
Epoch 25/25 | Train Loss: 0.3561 | Val Loss: 0.3914 | Val Acc: 85.63%
F1 Score: 0.8402
```

Lastly, for the LR = 0.0001 the results were as follows.

```
Epoch 20/25 | Train Loss: 0.4298 | Val Loss: 0.4446 | Val Acc: 84.01%
Epoch 21/25 | Train Loss: 0.4302 | Val Loss: 0.4381 | Val Acc: 84.15%
Epoch 22/25 | Train Loss: 0.4306 | Val Loss: 0.4422 | Val Acc: 83.64%
Epoch 23/25 | Train Loss: 0.4307 | Val Loss: 0.4379 | Val Acc: 84.13%
Epoch 24/25 | Train Loss: 0.4298 | Val Loss: 0.4399 | Val Acc: 84.03%
Epoch 25/25 | Train Loss: 0.4309 | Val Loss: 0.4561 | Val Acc: 83.57%
F1 Score: 0.7942
```

To sum up, here are the F1 Scores obtained for each different learning rates.

Learning Rate	F1 Score
0.0001	0.8344
0.001	0.8402
0.01	0.7942

Based on these results, it can be observed that the choice of learning rate had a noticeable effect on the model's performance. The learning rate of 0.001 achieved the highest F1 score, indicating that it facilitated better convergence and learning of the underlying patterns in the data. The learning rate of 0.0001 also yielded a relatively high F1 score, suggesting that it allowed the model to make meaningful progress during training. However, the learning rate of 0.01 resulted in a lower F1 score, indicating that it might have been too large and caused the model to overshoot the optimal parameters. These results suggest that a learning rate of 0.001 is well-suited for this particular classification task with the chosen architecture and dataset. It strikes a balance between making sufficient progress in learning and avoiding convergence issues. Since the 0.001 learning rate performed the best we have set the learning rate to this value for the following trials.

5.6.3. Hidden Layer Size

The third parameter we experimented with was the hidden layer size. The hidden layer size refers to the number of nodes in the fully connected layer. In this experiment the activation function was ReLU and the learning rate was 0.01. We have tested three different hidden layer sizes: 64, 128 and 256. First, for the 64 neurons the results were as follows.

```
Epoch 20/25 | Train Loss: 0.3790 | Val Loss: 0.3956 | Val Acc: 85.32%
Epoch 21/25 | Train Loss: 0.3788 | Val Loss: 0.3969 | Val Acc: 85.19%
Epoch 22/25 | Train Loss: 0.3784 | Val Loss: 0.3959 | Val Acc: 85.28%
Epoch 23/25 | Train Loss: 0.3781 | Val Loss: 0.3971 | Val Acc: 85.24%
Epoch 24/25 | Train Loss: 0.3778 | Val Loss: 0.3937 | Val Acc: 85.42%
Epoch 25/25 | Train Loss: 0.3771 | Val Loss: 0.4002 | Val Acc: 85.05%
F1 Score: 0.8304
```

Second, for 128 neurons, the results were as follows.

```
Epoch 20/25 | Train Loss: 0.3598 | Val Loss: 0.3954 | Val Acc: 85.50%
Epoch 21/25 | Train Loss: 0.3587 | Val Loss: 0.3945 | Val Acc: 85.38%
Epoch 22/25 | Train Loss: 0.3580 | Val Loss: 0.3896 | Val Acc: 85.75%
Epoch 23/25 | Train Loss: 0.3574 | Val Loss: 0.3901 | Val Acc: 85.47%
Epoch 24/25 | Train Loss: 0.3569 | Val Loss: 0.3925 | Val Acc: 85.59%
Epoch 25/25 | Train Loss: 0.3561 | Val Loss: 0.3914 | Val Acc: 85.63%
F1 Score: 0.8402
```

Lastly, for the 256 neurons, the results were as follows.

```
Epoch 20/25 | Train Loss: 0.3303 | Val Loss: 0.3932 | Val Acc: 85.96%
Epoch 21/25 | Train Loss: 0.3284 | Val Loss: 0.3963 | Val Acc: 85.78%
Epoch 22/25 | Train Loss: 0.3274 | Val Loss: 0.3973 | Val Acc: 85.75%
Epoch 23/25 | Train Loss: 0.3258 | Val Loss: 0.3971 | Val Acc: 86.14%
Epoch 24/25 | Train Loss: 0.3248 | Val Loss: 0.3971 | Val Acc: 86.11%
Epoch 25/25 | Train Loss: 0.3237 | Val Loss: 0.3971 | Val Acc: 86.12%
F1 Score: 0.8482
```

To sum up, here are the F1 Scores obtained for each different hidden layer size.

Hidden Layer Size (number of neurons)	F1 Score
64	0.8304
128	0.8402
256	0.8482

The experiments with different hidden layer sizes in the fully connected neural network showed that the choice of layer size has a noticeable impact on the F1 score. Increasing the layer size from 64 to 128 resulted in an improvement in the F1 score, indicating the model's ability to capture more intricate patterns. However, further increasing the layer size to 256 did not significantly enhance the F1 score compared to the model with a layer size of 128. This suggests diminishing returns in performance as the layer size grows. However, since 256 neurons performed the best, for the following trials, hidden layer size will be set to 256.

5.6.4. Number of Hidden Layers

The final parameter we experimented with was the number of hidden layers. The hidden layer size refers to how many fully connected layers are between the input and output layers. In this experiment, the activation function for each layer was ReLU, the learning rate was 0.01 and the hidden layer size was 256. We have tested three different hidden layer sizes: 1, 2 and 3. First, for the 1 hidden layer, the results were as follows.

Second, for 2 hidden layers, the results were as follows.

```
Epoch 20/25 | Train Loss: 0.2619 | Val Loss: 0.4098 | Val Acc: 87.11%
Epoch 21/25 | Train Loss: 0.2583 | Val Loss: 0.4206 | Val Acc: 87.06%
Epoch 22/25 | Train Loss: 0.2542 | Val Loss: 0.4168 | Val Acc: 87.15%
Epoch 23/25 | Train Loss: 0.2512 | Val Loss: 0.4224 | Val Acc: 87.12%
Epoch 24/25 | Train Loss: 0.2478 | Val Loss: 0.4215 | Val Acc: 87.08%
Epoch 25/25 | Train Loss: 0.2450 | Val Loss: 0.4354 | Val Acc: 87.19%
F1 Score: 0.8635
```

Lastly, for 3 hidden layers, the results were as follows.

```
Epoch 20/25 | Train Loss: 0.2671 | Val Loss: 0.4090 | Val Acc: 86.82%
Epoch 21/25 | Train Loss: 0.2639 | Val Loss: 0.4094 | Val Acc: 87.14%
Epoch 22/25 | Train Loss: 0.2596 | Val Loss: 0.4222 | Val Acc: 87.00%
Epoch 23/25 | Train Loss: 0.2569 | Val Loss: 0.4211 | Val Acc: 87.57%
Epoch 24/25 | Train Loss: 0.2534 | Val Loss: 0.4396 | Val Acc: 87.05%
Epoch 25/25 | Train Loss: 0.2503 | Val Loss: 0.4148 | Val Acc: 87.59%
F1 Score: 0.8639
```

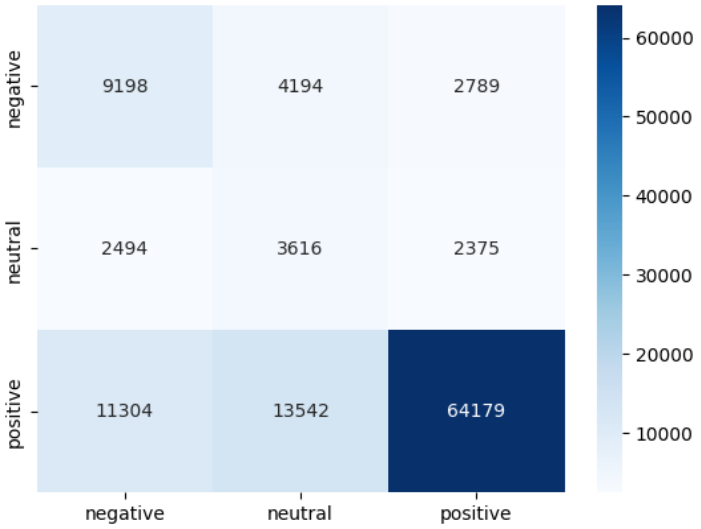
To sum up, here are the F1 Scores obtained for each different number of hidden layers.

Number of Hidden Layers	F1 Score
1	0.8482
2	0.8635
3	0.8639

The experiments with different numbers of hidden layers in the fully connected neural network showed that increasing the number of hidden layers from 1 to 2 resulted in a slight improvement in the F1 score. However, adding a third hidden layer did not lead to a significant improvement in performance compared to the model with two hidden layers. These results suggest that increasing the depth of the neural network can enhance its ability to learn complex representations up to a certain point, but there might be diminishing returns beyond a certain number of hidden layers.

6. Evaluation

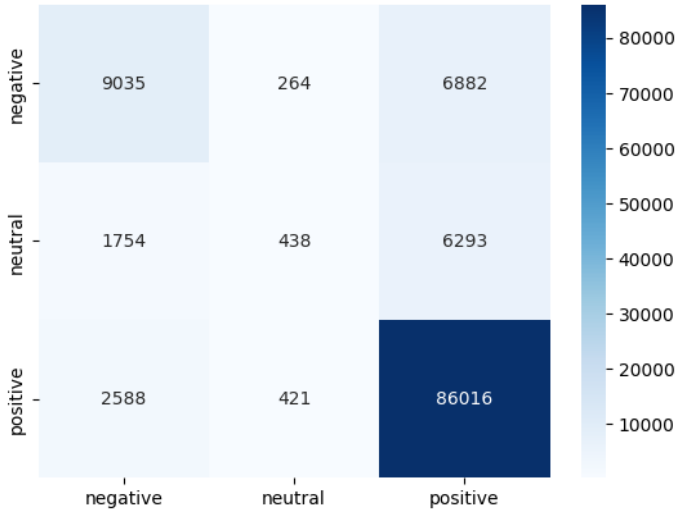
After having our fine-tuned models ready we have tested their performance on our test data. First for the Gaussian Naive Bayes model the accuracy, F1 score and the confusion matrix were as follows.




```
Overall accuracy: 0.6772
F1 score: 0.7196
```

As can be seen from the data, it performed similarly to the validation set. It was again better at predicting neutral and negative samples when compared to positive samples. Since there are many more positive samples in our dataset, and the model performed better with the minority classes, it had a better F1 score than accuracy.

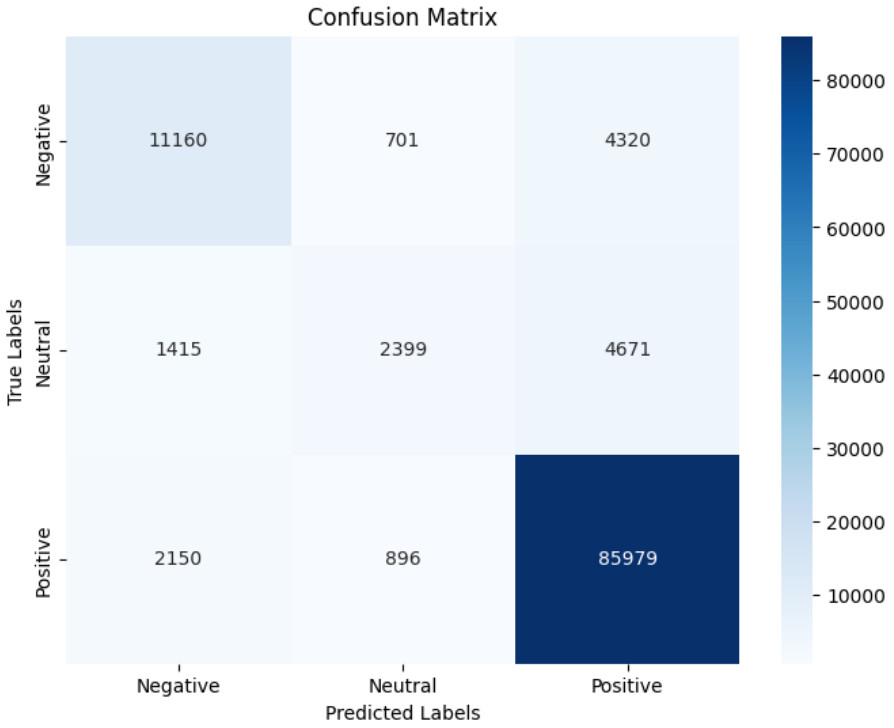
Next for the Logistic Regression, the confusion matrix, accuracy score and the F1 score for the test data is as below:




```
Overall accuracy: 0.8399
F1 score: 0.8095
```

As can be seen from the data, it performed similarly to the validation set. It was good at predicting positive samples and due to our imbalance dataset this resulted in a high accuracy. However, since it was not as good for the neutral and negative classes, it had a lower F1 Score than the accuracy.

Lastly, for the fine-tuned Fully Connected Layer Neural Network the accuracy, f1 score, and the confusion matrix were as follows. According to the experiments we have done previously we have set the activation function for each layer as ReLU, the learning rate as 0.01, the hidden layer size as 256, and the number of hidden layers as 3.




```
Overall accuracy: 0.8755
F1 score: 0.8633
```

As can be seen from the data, it performed similarly to the validation set. It was good at predicting positive samples and due to our imbalanced dataset, this resulted in a high accuracy. It was also the best-performing model for predicting the negative samples. On the other hand, it also performed moderately with the neutral class. So, overall, it has a similar accuracy and F1 Score.

To sum up, the following table shows the different accuracy and F1 Score values obtained for the different models.

Model	Accuracy	F1 Score
Gaussian Naive Bayes	0.6772	0.7196
Logistic Regression	0.8399	0.8095
Fully Connected Layer	0.8755	0.8633

7. Conclusion

This project aimed to automate the process of classification of product reviews, using a data set including 586000 product reviews from amazon.com. The aspect of creating a scientific base for the customer feedback made the project significant for businesses. We have included a total of three methods towards achieving our aim of automated classification: Gaussian Naive Bayes model, Logistic Regression, and Fully Connected Layer.

The Gaussian Naive Bayes addressed a large number of features efficiently, assuming a condition of independence between the features, such that the occurrence of different words are unrelated to each other. It performed highly especially for neutral and negative classes while poorly with the positive class. Since it fell short in addressing the skewed or non-Gaussian patterns in colloquial language, it achieved an accuracy of 67.66% and an F1 score of 0.7182, which are low compared to both of the other two approaches.

The Logistic Regression model achieved an accuracy of 0.8399 and an F1 score of 0.8095 on the imbalanced dataset. The high accuracy indicates overall good performance, but the lower F1 score suggests difficulties in correctly predicting the positive class, which was overrepresented in the dataset.

The Fully Connected Layer method was the best choice in terms of accuracy and F1 score since it can learn complex patterns and relationships in input data due to its introduction of nonlinearity and it has each of its neurons connected to every neuron in its previous layer. We changed many parameters for this method. The best activation function was ReLu (with F1 score = 0.8402), the best learning rate was 0.001 (with F1 score = 0.8402), the best hidden layer size (number of neurons in 1 layer) was 256 (with F1 score = 0.8482), the optimal number of hidden layers was 3 (with F1 score = 0.863). Also, the Fully Connected Layer approach had an accuracy score of 0.8755.

Hence, the Fully Connected Layer method proved to be the most optimal method in this project, providing the highest accuracy and F1 scores.

References:

[1] Brownlee, J. (2019, August 7). *What are word embeddings for text?* MachineLearningMastery.com. Retrieved April 28, 2023, from https://machinelearningmastery.com/what-are-word-embeddings/