Movie Sentiment Analysis: From Data to Deployment

1. Introduction

In the digital age, opinions about movies are everywhere - from social media to dedicated review platforms. But what if you could harness the power of natural language processing (NLP) to instantly gauge the sentiments expressed in these reviews? This article takes you on a journey through the creation of a Movie Sentiment Analysis application, from its inception to deployment.

2. Review

Why Analyze Movie Sentiments?

The movie industry thrives on audience feedback. Analysing sentiments in movie reviews can provide crucial insights into audience reception. Positive sentiments often indicate a hit, while negative sentiments can signal areas for improvement.

3. Requirements

Before diving into development, we need to outline our requirements:

- **Data**: A dataset of movie reviews with sentiment labels (positive or negative).
- **Libraries**: Python libraries like Transformers, NumPy, and Streamlit as shown below:

```
#Import Libraries
import os

#Data manipulation
import pandas as pd
import numpy as np
import datasets
from sklearn.model_selection import train_test_split
import collections
import emoji
import matplotlib.pyplot as plt
from wordcloud import WordCloud
import nltk
from nltk.corpus import stopwords
import re
import seaborn as sns

#finetuning
from transformers import AutoTokenizer
from transformers import TrainingArguments
```

```
from transformers import AutoModelForSequenceClassification
from transformers import Trainer
import torch
from torch.utils.data import DataLoader

#Evaluation
from datasets import load_metric, load_dataset
from sklearn.metrics import f1_score
from huggingface_hub import notebook_login

# Statistics
from scipy.stats import chi2_contingency
```

- **Pre-trained Models**: NLP models for sentiment analysis (e.g., BERT, DistilBERT).
- **Web Hosting**: A platform to deploy our app (e.g., Hugging Face Transformers Hub).

```
#login to huggingface
notebook_login()
```

4. Exploratory Data Analysis (EDA)

Our journey begins with data exploration. We must understand our dataset, its size, and distribution of sentiments. This helps us make informed decisions during model building. For instance, our main data, the train_df dataset is of the structure below:

```
# Check the train dataset

train_df
```

	review_file	content	sentiment
0	3471_8.txt	Recently shown on cable tv the movie opens wit...	positive
1	9693_8.txt	I was very surprised with this film. I was tou...	positive
2	10801_1.txt	Now, I'm one to watch movies that got poor rev...	negative
3	9592_8.txt	This film came out 12 years years ago, and was...	positive
4	8514_7.txt	When an orphanage manager goes on vacation, hi...	positive
...	...	...	...
24995	2791_3.txt	As with most of the reviewers, I saw this on S...	negative
24996	644_9.txt	A have a female friend who is currently being ...	positive
24997	4921_8.txt	Like A Streetcar Named Desire (also directed b...	positive
24998	5791_1.txt	As a Native film professor, I can honestly say...	negative
24999	8997_10.txt	I've seen this movie on several different occa...	positive

25000 rows × 3 columns

We need also to remove the rows with null values in the sentiment column of the train_df dataset by using the code below:

```
# Remove rows with null values

train_df.dropna(subset=['sentiment'], inplace=True)
```

Fortunately when we review the data, we realize that there were no null values as illustrated below:

```
# View info once again from Train dataset

train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 25000 entries, 0 to 24999
Data columns (total 3 columns):
 #   Column       Non-Null Count  Dtype 
---  ------       --------------  ----- 
 0   review_file  25000 non-null  object
 1   content      25000 non-null  object
 2   sentiment    25000 non-null  object
dtypes: object(3)
     memory usage: 586.1+ KB
```

To visualize the most frequently used words in the 'content' column, we can use the word cloud as shown below:

```
# Generate a word cloud from the message text data

text = " ".join(sentence for sentence in train_df.content)
wordcloud = WordCloud(width=800, height=400, max_font_size=200,
background_color="white")
# Generate the word cloud
wordcloud.generate(text)

# Display the word cloud image
plt.figure(figsize=(16,8))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.show()
```

The Figure 1 below is the visual display of the word cloud, and it enables us to identify that words like 'movie', 'film', 'one', 'time', 'character' are frequently used.

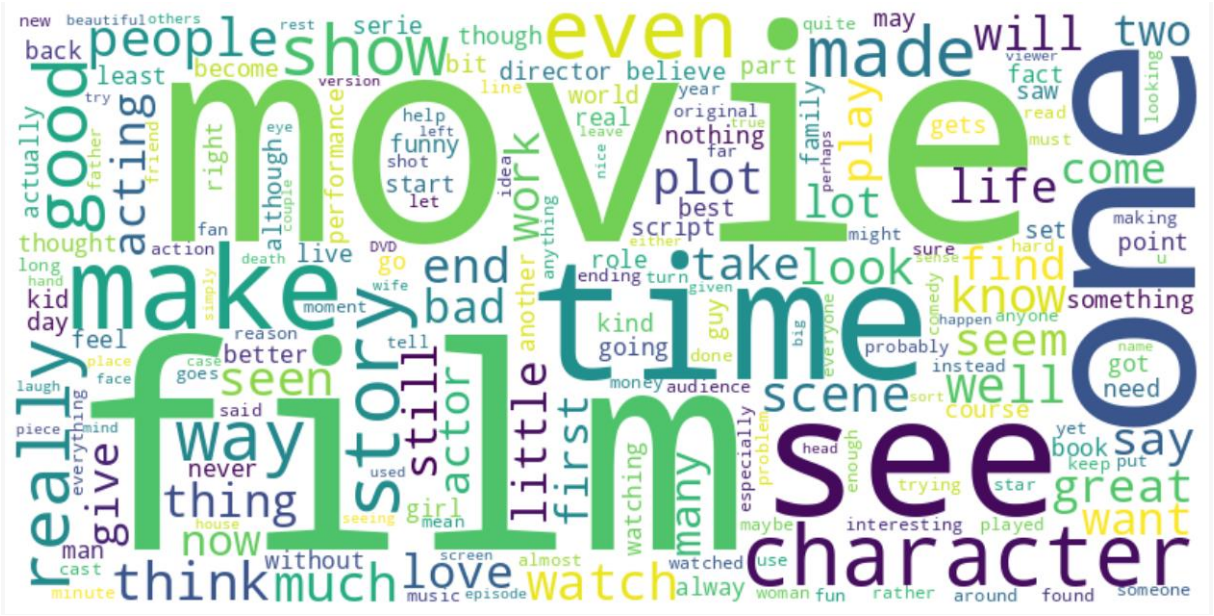


Figure 1

To show these most frequently used words, we need first to remove stop words then visualize the first twenty most frequently used words. Figure 2 shows a bar graph of these twenty words.

```
# Plot the bar chart
```

```
plt.figure(figsize=(10, 6))
plt.bar(top_words.keys(), top_words.values())
plt.xlabel('Words')
plt.ylabel('Frequency')
plt.title('Top 20 Most Common Words')
plt.xticks(rotation=45)
plt.show()
```

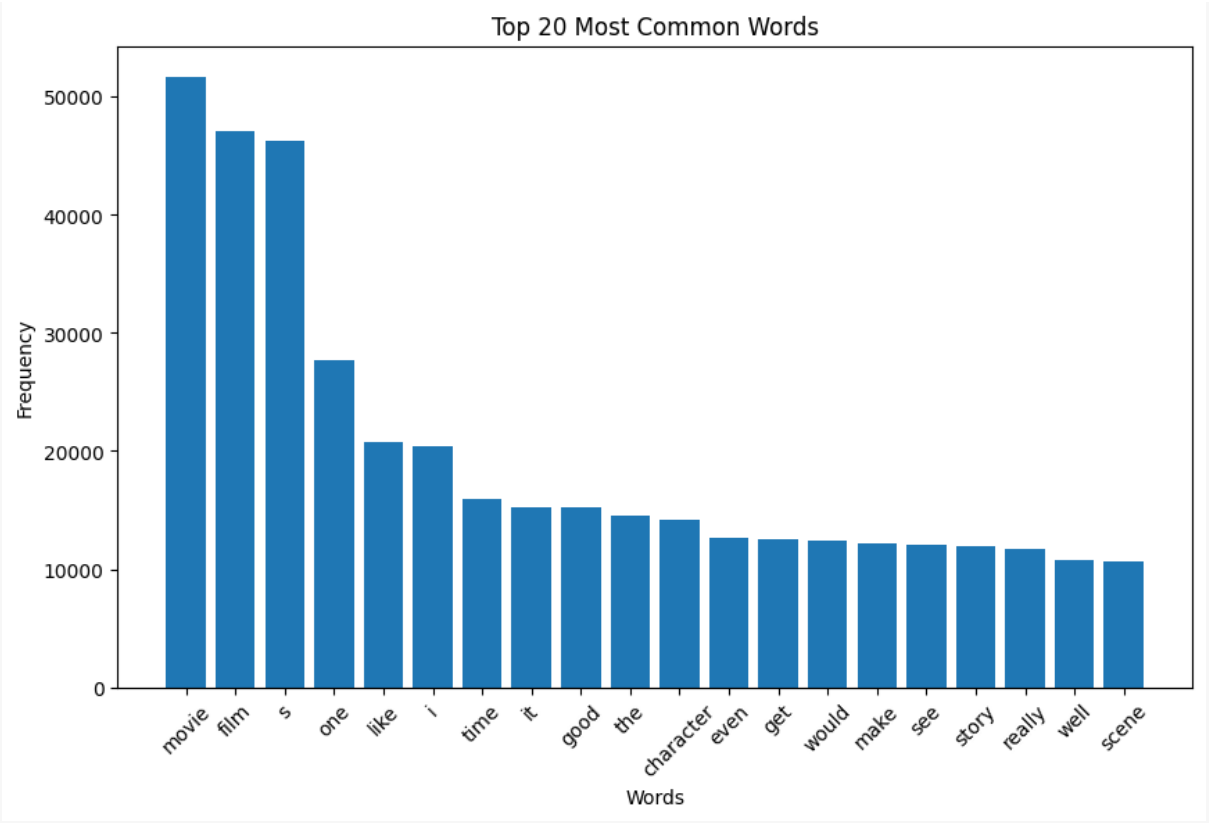


Figure 2

It is also important to show the distribution of the sentiment labels in the dataset. To help us do this, we use the code below to work out this distribution then visualize it as shown in Figure 3 below:

```
# Calculate the frequency of each sentiment label
label_counts = train_df['sentiment'].value_counts()
label_counts

train_df['sentiment'] = train_df['sentiment'].astype('category')
# Create a bar chart to visualize the distribution
plt.figure(figsize=(8, 6))
sns.barplot(x=train_df['sentiment'].value_counts().index,
y=train_df['sentiment'].value_counts().values)
plt.xlabel('Sentiment Label')
plt.ylabel('Frequency')
plt.title('Distribution of Sentiment Labels')
plt.show()
```

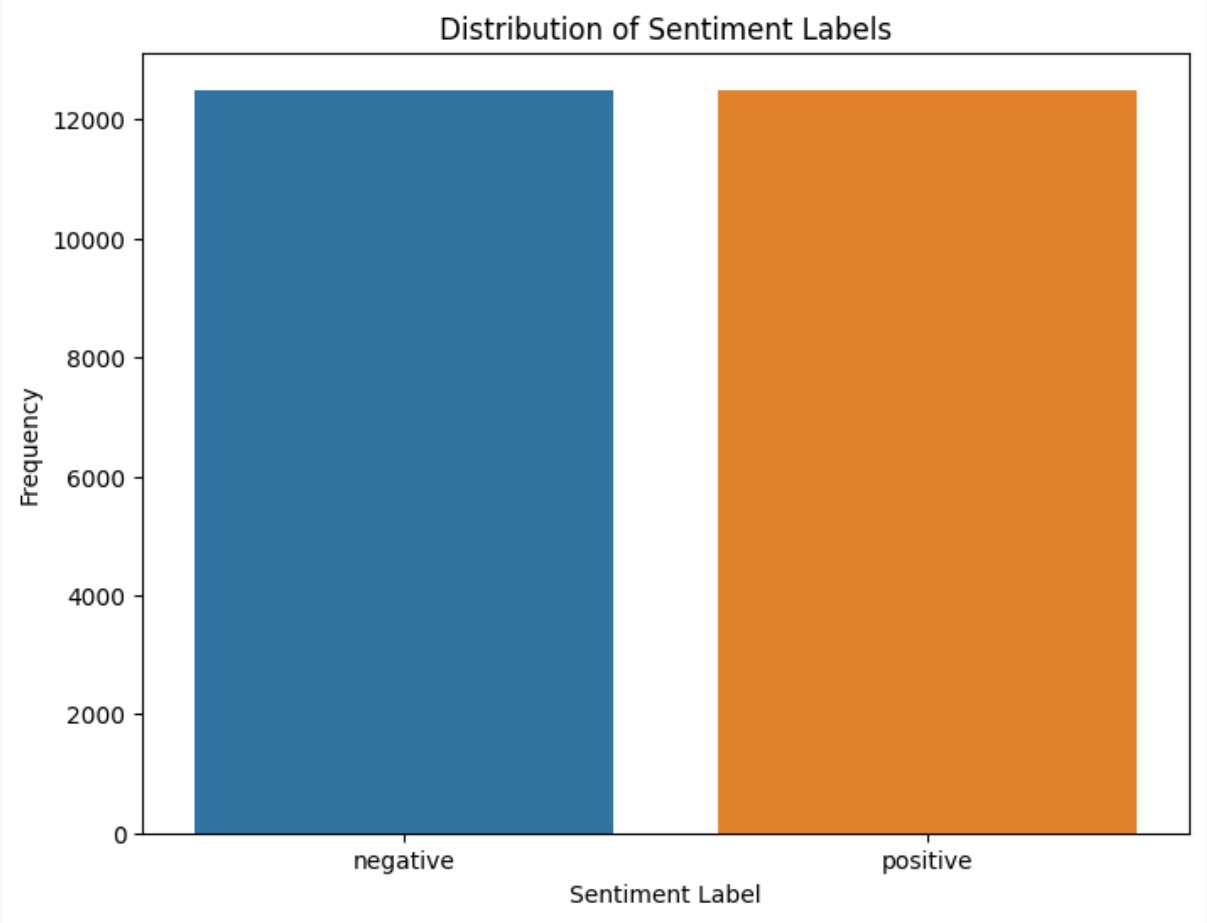


Figure 3

From the bar graph we can see that the sentiments are equally distributed in the dataset.

Another very import EDA aspect to look at is to perform text length analysis on the "content" column. By computing descriptive statistics of text lengths, we are able to tell maximum, minimum, mean, etc., of the text lengths used in the 'content' column in the dataset. The code below helps use to achieve this and then be able to visualize it as shown in Figure 4.

```
# Calculate the length of each text in the "content" column
text_lengths = train_df['content'].apply(len)

# Compute descriptive statistics of text lengths
text_length_stats = text_lengths.describe()

# Print the descriptive statistics
print(text_length_stats)
```

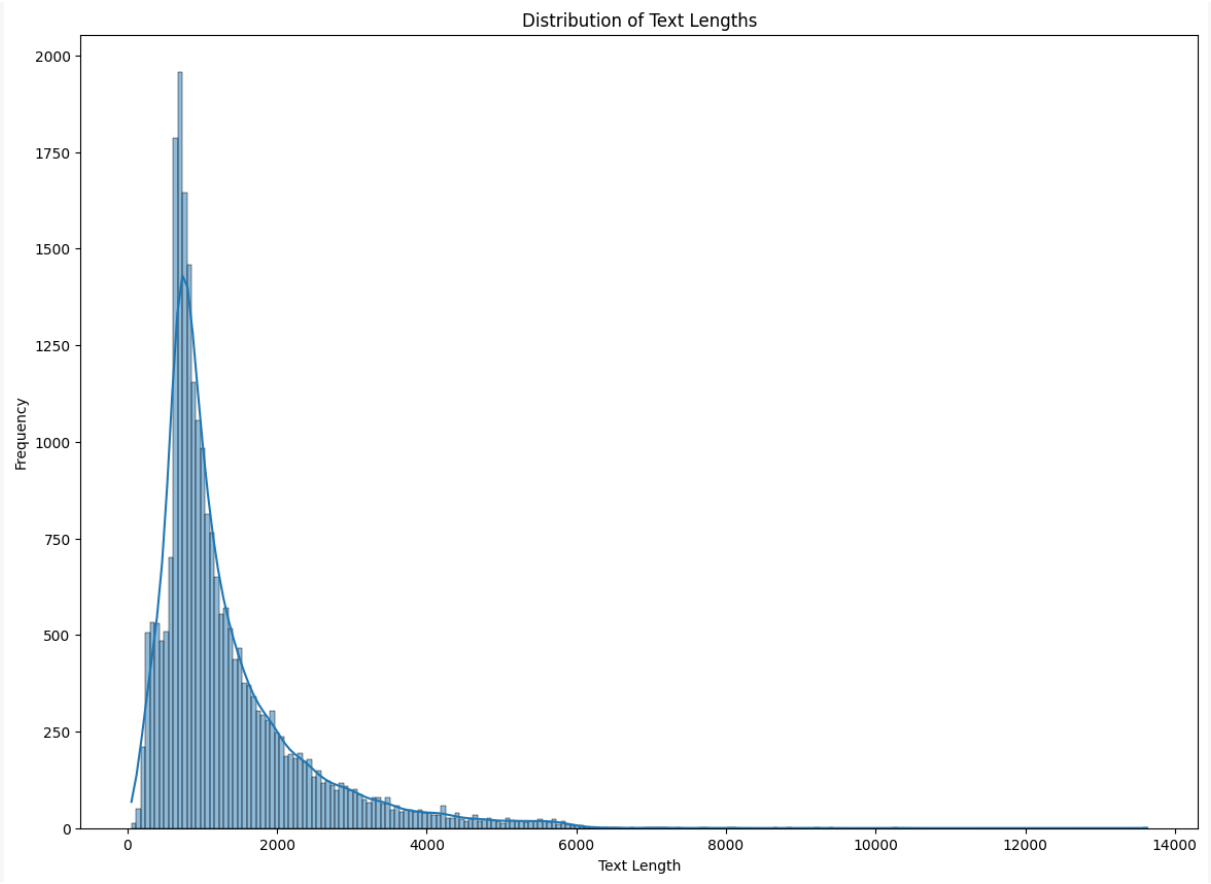


Figure 4

From Figure 4, we can draw the following conclusions:

1. The distribution appears to be right-skewed, indicating that there are relatively few texts with very short lengths and a larger number of texts with longer lengths.
2. The majority of texts seem to fall within the range of around 500 to 2000 characters.
3. As text length increases beyond this range, the frequency of texts decreases significantly.
4. The KDE (kernel density estimation) plot provides additional smoothness to the distribution, allowing us to observe trends and modes more easily.

The distribution of the text length can also be depicted using a box plot as shown in Figure 5 below.

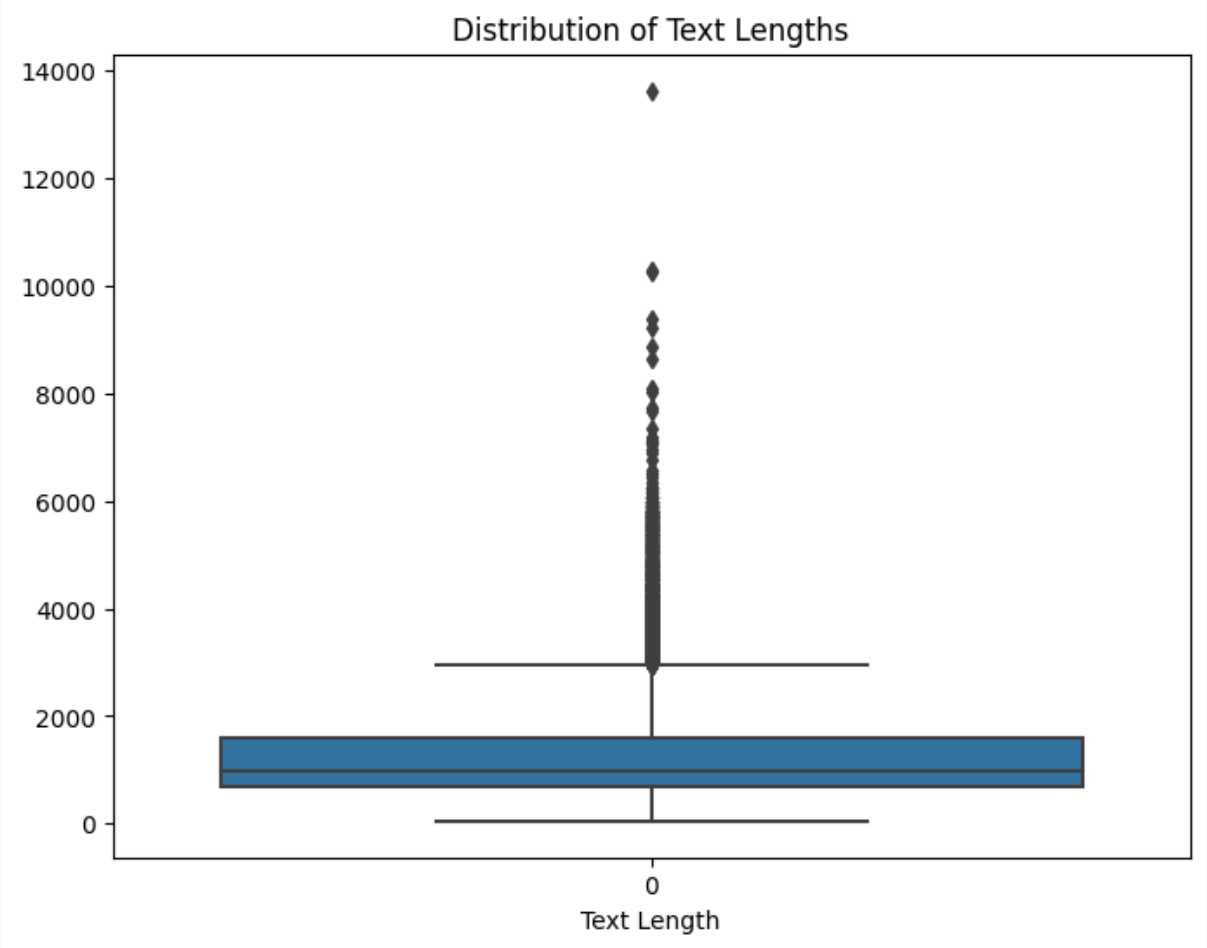


Figure 5

From the boxplot we can observe that:

- The box represents the interquartile range (IQR), which contains the middle 50% of the data.
- The bottom edge of the box indicates the 25th percentile (Q1), and the top edge indicates the 75th percentile (Q3).
- The horizontal line inside the box represents the median (50th percentile).
- The "whiskers" extend from the edges of the box to the minimum and maximum values within a certain range. Data points outside this range are considered potential outliers and are plotted individually as dots.
- The dots above the whiskers are individual data points that are considered outliers.

The chart shown in Figure 6 helps us to compare the distribution and relative frequencies of keywords across different sentiment categories.

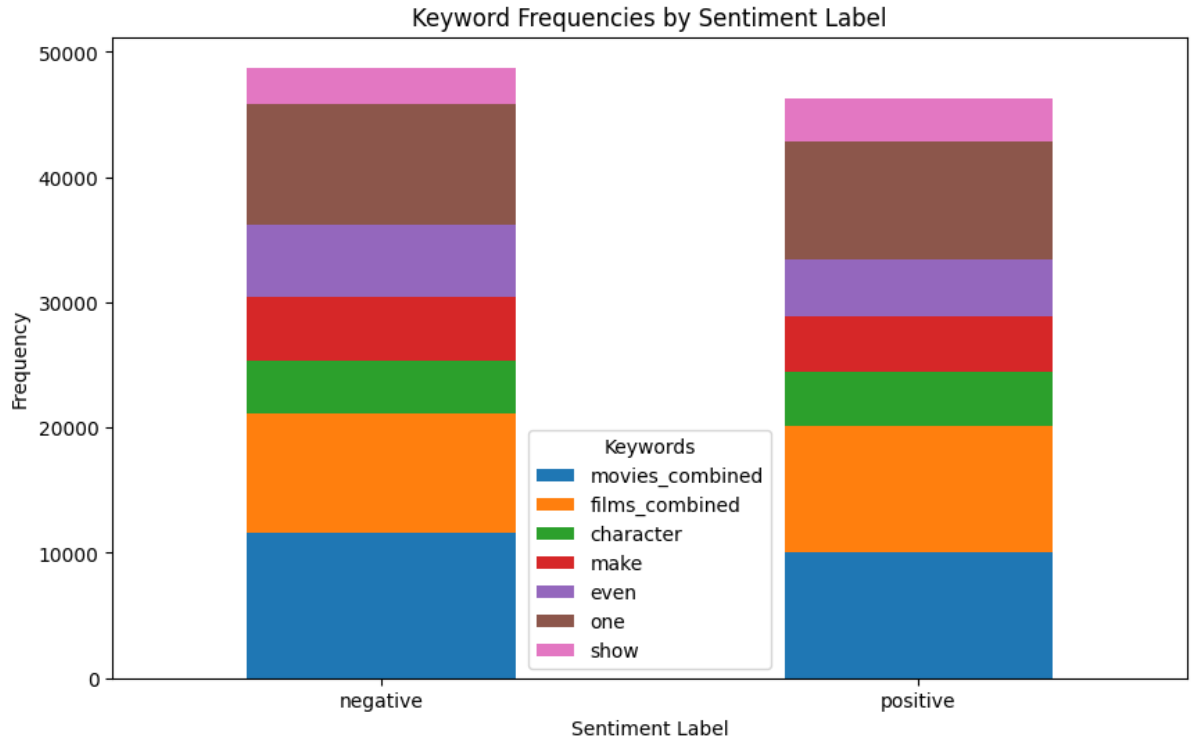


Figure 6

We can conclude that these commonly used words in the sentences are almost uniformly distributed in their usage between the two sentiments with a slightly more usage of the word(s) movie(s) in negative sentiment.

5. Model Building

We delve into model building, selecting from the vast world of Natural Language Processing (NLP) models. BERT, DistilBERT, and RoBERTa are popular choices. Each has its strengths, and our choice depends on factors like model size and accuracy.

The first step before training the models is to create a tokenizer instance. The line of code below initializes a tokenizer for the "sentence_sentiments_analysis_bert" model using the transformers library. The tokenizer is an essential component for processing text data and is used to convert input text into tokens that can be fed into the model for sentiment analysis.

```
#Create a tokenizer instance

tokenizer=AutoTokenizer.from_pretrained('bert-base-cased')
```

Since some sequences might be longer than the length of 512 allowable for BERT, DistilBERT, and RoBERTa models, it is important to divide then into chunks of 512 as shown in the code below.

```
    # Chunk size for dividing longer sequences
    chunk_size = 512  # Adjust this based on the length of the tokens
```

The second step is to create functions to tokenize text and transform labels. We shall need two functions to do the following:

1. **tokenize_data(data)**: This function takes a dictionary-like data object as input, which is assumed to have a 'content' field containing the text to be tokenized. Here's how the function works:
    - **content = data['content']**: This line extracts the 'content' field from the input dictionary and assigns it to the variable content.
    - **tokens = tokenizer(content, padding='max_length', truncation=True, max_length=chunk_size, return_tensors='pt')**: This line uses the Hugging Face tokenizer to tokenize the content text. The padding='max_length' argument ensures that the sequences are padded to the specified max_length (assuming max_length is defined elsewhere in the code). The truncation=True argument truncates sequences that exceed the max_length. The max_length=chunk_size argument specifies the maximum length of the tokenized sequence. The return_tensors='pt' argument indicates that the output should be returned as PyTorch tensors.
    - **return tokens**: The function returns the tokenized representation of the text content as a dictionary of PyTorch tensors, which includes the token IDs, attention masks, and other relevant information.
2. **transfom_label(data)**: This function takes a dictionary-like data object as input, which is assumed to have a 'sentiment' field containing the sentiment label ('positive' or 'negative'). Here's how the function works:
    - **label = data['sentiment']**: This line extracts the 'sentiment' field from the input dictionary and assigns it to the variable label.
    - The function then uses a simple conditional logic to transform the sentiment label into a numeric label:
        - If the label is 'negative', the function assigns the value 0 to the variable num.

        - If the label is 'positive', the function assigns the value 1 to the variable num.
    - The function then returns a dictionary with the label information in the format expected by the model. In this case, the key is 'labels', and the value is the numeric label (0 for 'negative' and 1 for 'positive').

This is illustrated in the code snippet below:

```
      def tokenize_data(data):
    content = data['content']
    tokens = tokenizer(content, padding='max_length', truncation=True,
max_length=chunk_size, return_tensors='pt')
    return tokens


#function to transform labels
def transfom_label(data):

    #extract label
    label=data['sentiment']
    num=0
    #create conditions
    if label=='negative':
        num=0
    else:
        num=1       #Positive
    return {'labels':num}
```

The third step is to tokenizer text and transform original sentiment labels. This is achieved by the code below:

```
      #The following columns will be removed after tokenization
      remove_columns = ['review_file', 'content', 'sentiment']

      #Tokenize the text data
      datasets = datasets.map(tokenize_data, batched=True)

      #transform the labels
      datasets=datasets.map(transfom_label,remove_columns=remove_column
      s)
```

The fourth step is to extract the train and eval datasets from datasets as shown in the code snippet below:

```
#extract train datasets
```

```
train_dataset=datasets['train'].shuffle(seed=42)
#extract eval datasets
eval_dataset=datasets['eval'].shuffle(seed=42)
```

The next step is to define metrics function as shown below:

```
# Define the function to compute F1-score
```

```
def compute_metrics(eval_preds):
    logits, labels = eval_preds
    predictions = np.argmax(logits, axis=-1)
    f1 = f1_score(labels, predictions, average='weighted')
    return {"f1-score": f1}
```

The above function works as follows:

1. **logits**: This array contains the predicted scores or probabilities for each class produced by the model. Each row corresponds to a data point, and each column corresponds to a class. The values in the logits array indicate how strongly the model believes each class is present in the data point.
2. **labels**: This array contains the true labels or ground truth for each data point. These are the actual classes to which the data points belong.
3. **predictions**: The np.argmax function is used to find the index of the highest predicted score (probability) in the logits array for each data point. This index corresponds to the predicted class label for that data point.
4. **f1_score(labels, predictions, average='weighted')**: This computes the F1-score using the f1_score function from the sklearn.metrics module. The F1-score is a metric that combines precision and recall to provide a balanced measure of a model's accuracy. It's particularly useful when dealing with imbalanced classes.
5. **return {"f1-score": f1}**: The function returns a dictionary containing the calculated F1-score with the key "f1-score". This format is suitable for reporting and tracking multiple metrics during model evaluation.

Three models shall be trained in turns as follows:

1. 'bert-base-cased' and be named as: 'sentence_sentiments_analysis_bert'
2. 'distilbert-base-cased' and be named as: 'sentence_sentiments_analysis_distilbert'
3. 'roberta-base' and be named as: 'sentence_sentiments_analysis_roberta'

Just before we start training we need to do the following:

(i) Set the training arguments. The **TrainingArguments** object is used to define various training configurations and hyperparameters, such as the number of training epochs, evaluation and saving strategies, and whether to push the model to the Hugging Face Model Hub after training. This object is then passed to the Trainer class for model training using the specified settings. This is shown in the code snippet below:

```
#set the training arguments
trainargs=TrainingArguments('sentence_sentiments_analysis_bert',
                            num_train_epochs=5,
                            evaluation_strategy="epoch",
                            save_strategy='epoch',
                            load_best_model_at_end=True,
                            push_to_hub=True)
```

(ii) Create instance of the model. The code below creates a pre-trained sequence classification model with two output labels and initializes it with the pre-trained weights from the model. This model can be fine-tuned for a specific classification task, such as sentiment analysis, with the appropriate training data.

```
# Create an instance of the BERT model
model_name = 'bert-base-cased'
num_labels = 2
model =
AutoModelForSequenceClassification.from_pretrained(model_name,
num_labels=num_labels)
```

(iii) Create trainer instance. The code below creates a Trainer instance with the necessary components, including the model, training and evaluation datasets, tokenizer, and a function for computing evaluation metrics. The Trainer instance can be used to train and evaluate the pre-trained sequence classification model on the specified datasets using the defined training arguments.

```
# Create a Trainer instance
trainer = Trainer(
    model=model,
    args=trainargs,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics,
    )
```

6. Pushing Trained Models to Hugging Face

Once we've trained our models, we can contribute them to the Hugging Face Transformers Hub. This not only shares our models with the community but also enables seamless integration into web applications. The trained models cards from hugging face are illustrated below:

sentence_sentiments_analysis_bert

This model is a fine-tuned version of bert-base-cased on an unknown dataset. It achieves the following results on the evaluation set:

- Loss: 0.3401
- F1-score: 0.9007

Model description

More information needed

Intended uses & limitations

More information needed

Training and evaluation data

More information needed

Training procedure

Training hyperparameters

The following hyperparameters were used during training:

- learning_rate: 5e-05
- train_batch_size: 8
- eval_batch_size: 8
- seed: 42
- optimizer: Adam with betas=(0.9,0.999) and epsilon=1e-08
- lr_scheduler_type: linear
- num_epochs: 5

Training results

Training Loss	Epoch	Step	Validation Loss	F1-score
0.3194	1.0	2500	0.3401	0.9007
0.232	2.0	5000	0.3748	0.9126
0.1364	3.0	7500	0.4277	0.9186
0.0537	4.0	10000	0.4881	0.9214
0.0238	5.0	12500	0.5175	0.9240

Framework versions

- Transformers 4.32.1
- Pytorch 2.0.1+cu118

- Datasets 2.14.4
- Tokenizers 0.13.3

sentence_sentiments_analysis_distilbert

This model is a fine-tuned version of distilbert-base-cased on an unknown dataset. It achieves the following results on the evaluation set:

- Loss: 0.2850
- F1-score: 0.8986

Model description

More information needed

Intended uses & limitations

More information needed

Training and evaluation data

More information needed

Training procedure

Training hyperparameters

The following hyperparameters were used during training:

- learning_rate: 5e-05
- train_batch_size: 8
- eval_batch_size: 8
- seed: 42
- optimizer: Adam with betas=(0.9,0.999) and epsilon=1e-08
- lr_scheduler_type: linear
- num_epochs: 5

Training results

Training Loss	Epoch	Step	Validation Loss	F1-score

0.3178	**1.0**	**2500**	**0.2850**	**0.8986**
0.2157	**2.0**	**5000**	**0.3249**	**0.9046**
0.1018	**3.0**	**7500**	**0.4707**	**0.9080**
0.0295	**4.0**	**10000**	**0.5929**	**0.9090**
0.0133	**5.0**	**12500**	**0.6411**	**0.9120**

Framework versions

- Transformers 4.32.1
- Pytorch 2.0.1+cu118
- Datasets 2.14.4
- Tokenizers 0.13.3

sentence_sentiments_analysis_roberta

This model is a fine-tuned version of roberta-base on an unknown dataset. It achieves the following results on the evaluation set:

- Loss: 0.2736
- F1-score: 0.9119

Model description

More information needed

Intended uses & limitations

More information needed

Training and evaluation data

More information needed

Training procedure

Training hyperparameters

The following hyperparameters were used during training:

- learning_rate: 5e-05
- train_batch_size: 8
- eval_batch_size: 8
- seed: 42
- optimizer: Adam with betas=(0.9,0.999) and epsilon=1e-08
- lr_scheduler_type: linear
- num_epochs: 5

Training results

Training Loss	Epoch	Step	Validation Loss	F1-score
0.3477	1.0	2500	0.3307	0.9112
0.2345	2.0	5000	0.2736	0.9119
0.175	3.0	7500	0.3625	0.9161
0.1064	4.0	10000	0.3272	0.9358
0.07	5.0	12500	0.3291	0.9380

Framework versions

- Transformers 4.32.1
- Pytorch 2.0.1+cu118
- Datasets 2.14.4
- Tokenizers 0.13.3

7. Testing the Models with Test Data

Testing is crucial. We use a separate test dataset to evaluate the models' performance. Metrics like accuracy, precision, and recall are vital in assessing how well our models can predict sentiments. We shall use test_df dataset to see how the trained models would predict the sentiments.

Let us look at the results of the sentiments outcome from one of the three models, Roberta using the code snippet below:

```
      # Already predicted sentiments added to 'predicted_sentiment'
column
```

```
      selected_columns = ['review_file', 'predicted_sentiment']
      result_df = test_df.loc[:, selected_columns]

      # Display 'result_df' contents of only the 'review_file' and
      'predicted_sentiment' columns
      print('Roberta Model:  Predicted Sentiments on Test Data')
      print('==============================================')
      result_df.head(10)
```

```
Roberta Model:  Predicted Sentiments on Test Data
==============================================
```

	review_file	predicted_sentiment
0	0_10.txt	positive
1	0_2.txt	negative
2	10000_4.txt	negative
3	10000_7.txt	positive
4	10001_1.txt	negative
5	10001_9.txt	positive
6	10002_3.txt	negative
7	10002_8.txt	positive
8	10003_3.txt	negative
9	10003_8.txt	positive

The Distribution of predicted_sentiment for Roberta model can be shown with the code snippet below:

```
# Calculate the frequency of each predicted sentiment label
sentiment_counts = test_df['predicted_sentiment'].value_counts()
sentiment_counts
```

```
positive 12997
negative 12003
Name: predicted_sentiment, dtype: int64
```

The Visualization of sentiment_counts for Roberta Model is shown in Figure 7 below:

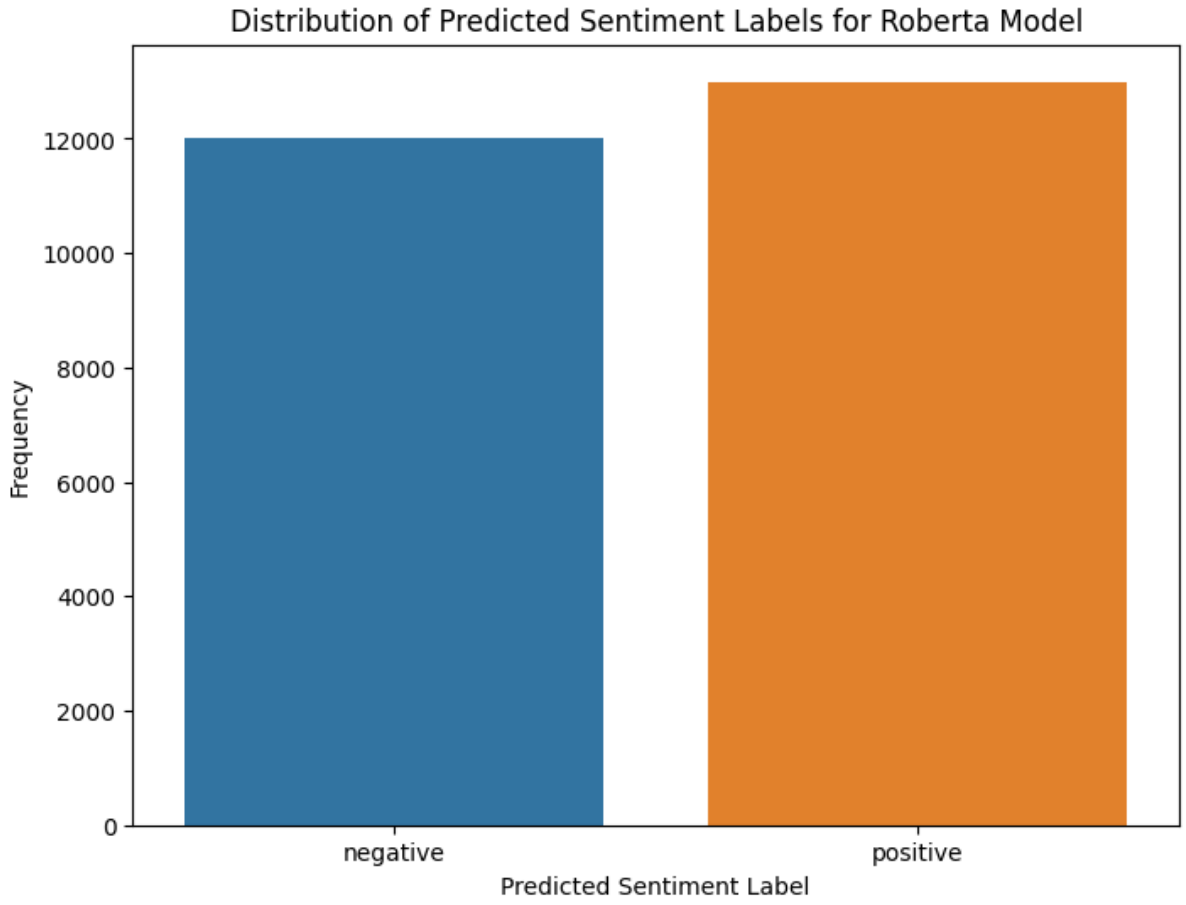


Figure 7

8. Hypothesis Testing on Models' Performance

To ensure we've selected the right model, we can conduct hypothesis testing. Statistical tests help us compare the models' performance and determine if the models are significantly different or not in their performances.

The common statistical test for comparing proportions or success rates between two or more groups is the chi-squared test (χ^2 test). In our case, you want to compare the proportions of positive and negative predictions made by each model.

So we shall formulate our hypothesis as follows:

1. **Null Hypothesis (Ho)**: There is NO significant difference between the models
2. **Alternative Hypothesis (H1)**: There is significant difference between the models

The code snippet below works out the χ^2 test:

```
# Create a contingency table (observed frequencies)
observed = df[["Positive", "Negative"]].values

# Perform the chi-squared test
chi2, p, _, _ = chi2_contingency(observed)

# Degrees of freedom
degrees_of_freedom = (observed.shape[0] - 1) * (observed.shape[1] - 1)

# Print the results
print(f"Chi-squared statistic: {chi2:.2f}")
print(f"P-value: {p:.4f}")
print(f"Degrees of freedom: {degrees_of_freedom}")

# Interpret the results
alpha = 0.05  # Set the significance level
if p < alpha:
    print("There is evidence of a significant difference between the models.")
else:
    print("There is no evidence of a significant difference between the models.")
```

The output of the above code is here below:

```
Chi-squared statistic: 97.98
P-value: 0.0000
Degrees of freedom: 2
There is evidence of a significant difference between the models.
```

From the result, we therefore reject the Null hypothesis and accept the Alternative Hypothesis (H1): that there is significant difference between the models. Meaning that each model behaves differently when predicting the sentiments.

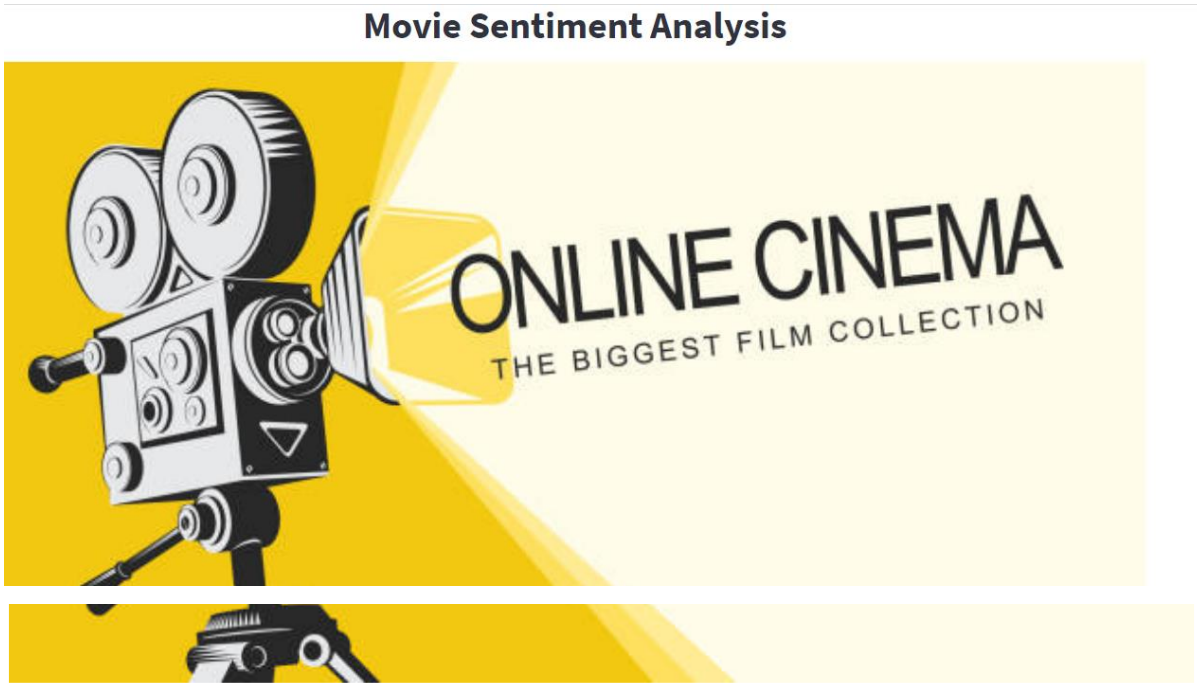
9. App Development in Streamlit

With a trained model in hand, it's time to build the Movie Sentiment Analysis app. Streamlit, a fantastic tool for web app development, simplifies this process. Users can input a sentence or choose from predefined examples and select the model they prefer.

10. Deployment of Streamlit App in Hugging Face

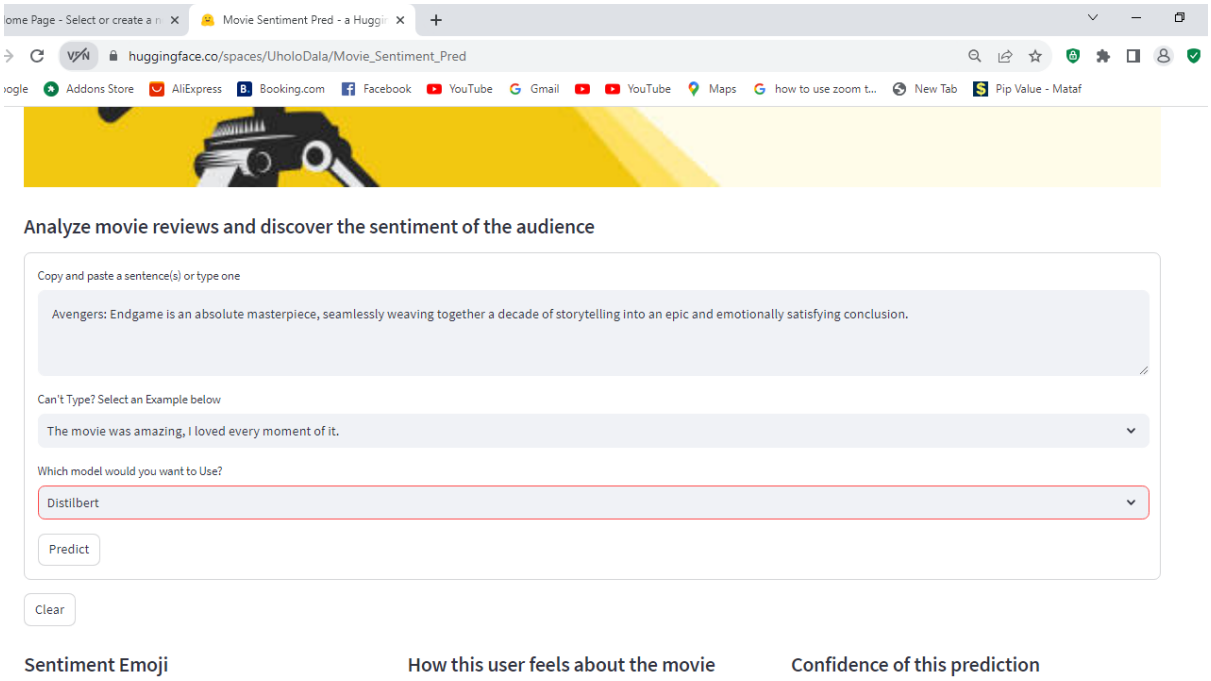
We aim to make our app accessible to a broader audience. Deploying it on Hugging Face's cloud platform is a fantastic way to achieve this. Users can access our app effortlessly through their web browsers.

Below is how the App appears in huggingface.

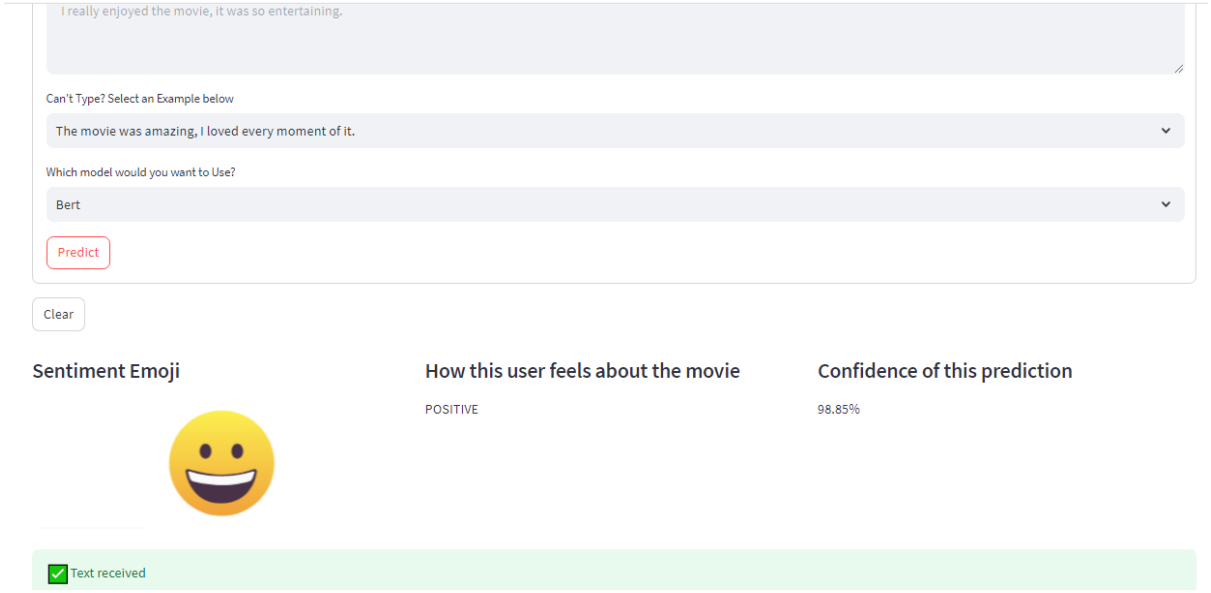


The use can then type in a sentence(s) or choose one of the option given then predict the sentiment about the movie. For example, if a use types in a statement like:

"Avengers: Endgame is an absolute masterpiece, seamlessly weaving together a decade of storytelling into an epic and emotionally satisfying conclusion." The App input frontend will appear showing the typed sentence and the chosen model, here Distilbert has been picked as in the figure below:



And the output will appear as shown below:



11. Evaluation of App

The app isn't complete without evaluating its performance. We consider factors like user-friendliness, response time, and the accuracy of sentiment predictions.

The app truly shines in terms of user-friendliness, boasting an intuitive interface that ensures a smooth user experience. Additionally, the app excels in response time, delivering prompt and seamless interactions. Furthermore, it achieves remarkable accuracy in sentiment predictions, enhancing its overall performance and utility.

12. Conclusion

The Movie Sentiment Analysis app is a testament to the power of NLP and machine learning in understanding audience sentiments. It provides movie enthusiasts, critics, and curious individuals with quick insights into how the public perceives films.

13. Recommendations

To continue improving our app and exploring the world of NLP, we have several recommendations:

- **Multi-Language Support**: Extend the app's capabilities to analyze sentiments in multiple languages.
- **Visualization**: Incorporate interactive data visualizations to display sentiment trends over time or compare sentiments across different movies.
- **Feedback Mechanism**: Implement a feedback mechanism to collect user input and continuously improve the application.
- **Custom Models**: Allow users to upload and use their custom sentiment analysis models, expanding the range of choices.
- **Error Handling**: Enhance error handling to provide users with more informative messages in case of input errors.

With these recommendations, we can create a robust, versatile, and user-centric tool for exploring sentiments in movie reviews. The world of NLP is ever-evolving, and the possibilities for innovative applications are limitless. So, dive in, analyse your favourite movie reviews, and uncover the sentiments of the audience with the Movie Sentiment Analysis app. Enjoy exploring the world of sentiment analysis and NLP!

I thank you for taking your time to read this article. Please don’t hesitate to reach me with your suggestions and ideas at:

jaroyajo@gmail.com

https://www.linkedin.com/in/jjaroya/