# Introduction

This project focuses on sentiment analysis, specifically analyzing movie reviews to classify sentiments as either positive or negative. The primary objective is to determine the emotional tone conveyed in the reviews, helping to gauge audience reactions to films.

To achieve this, we employed a combination of traditional machine learning algorithms (Logistic Regression and Support Vector Machine) and a modern deep learning model (DistilBERT). The dataset consists of cleaned movie reviews, which provide a rich source of textual data for training and evaluating our models.

By implementing these models and assessing their performance using metrics such as accuracy, precision, and recall, the project aims to highlight the effectiveness of each approach in accurately predicting sentiment. The findings will enhance our understanding of sentiment analysis techniques and their practical applications in the entertainment industry.

# 1. Data Cleaning

* First, we define the strip\_html function, which utilizes the BeautifulSoup library to parse and remove HTML tags from the text. This process ensures that any HTML formatting is eliminated and leaves only the plain text content.
* Function remove\_between\_square\_brackets employs regular expressions to identify and remove any content found within square brackets.
* Function remove\_urls is implemented to identify and delete any URLs present in the reviews. This prevents links from being considered during sentiment analysis as they do not contribute meaningful context.
* Function remove\_stopwords is defined to filter out common stopwords using a predefined list. By excluding terms such as "the," "is," and "and," which do not carry significant meaning, the analysis can focus on more impactful words.
* denoise\_text function works as cleaning tool that sequentially applies all previous cleaning methods. This function returns a refined version of the original reviews.
* Next, the code applies the denoise\_text function to the 'review' column of the DataFrame. This processes each review individually and generates a new column named 'cleaned\_review' that contains the sanitized text.

# 1. Models

## 1.1 Logistic Regression

* First, we convert text data into numerical features using TF-IDF (Term Frequency-Inverse Document Frequency). This transforms words into numerical values based on their importance in file.

log\_reg = LogisticRegression(max\_iter=200)

param\_grid = {'C': [0.01, 0.1, 1, 10, 100], 'solver': ['lbfgs', 'liblinear']}

grid\_search = GridSearchCV(log\_reg, param\_grid, cv=5, scoring='accuracy')

* This part sets up a logistic regression model with grid search to find the best hyperparameters. It will try different combinations of regularization strength (C) and optimization algorithms (solver).
* Then line grid\_search.fit(X\_train\_tfidf, y\_train) trains the logistic regression model using cross-validation to find the best hyperparameters from the defined grid.
* After fitting, best\_log\_reg = grid\_search.best\_estimator\_ retrieves the optimal model configuration.
* Finally, y\_pred = best\_log\_reg.predict(X\_test\_tfidf) generates predictions on the test set.

## 1.1.1 Hyperparamter Tuning

* Hyperparameter tuning is chosen for the logistic regression model to optimize its performance by systematically exploring different values for parameters that control the learning process. In logistic regression, hyperparameters such as the regularization strength (C) and the solver algorithm can influence the model's ability to generalize to unseen data. By finding the optimal settings for these hyperparameters, we can enhance the model's accuracy and reduce overfitting.

results = grid\_search.cv\_results\_

results\_df = pd.DataFrame(results)

* Using Grid Search we can identify the optimal settings for the logistic regression model.
* Initially, the grid\_search.fit(X\_train\_tfidf, y\_train) function fits the model to the training data while testing various combinations of hyperparameters defined in the param\_grid.
* After fitting, the best estimator is retrieved using best\_log\_reg = grid\_search.best\_estimator\_, which represents the model configuration that gave the highest cross-validation accuracy.
* Finally, predictions are made on the test set with y\_pred = best\_log\_reg.predict(X\_test\_tfidf)

## 1.2 Support Vector Machine

* For faster calculations, we are limiting the input data by df = df.head(10000) this statement which only takes first 10000 rows in the dataframe.

stop\_words = set(stopwords.words('english'))

# Preprocess the data

def preprocess\_text(text):

    tokens = word\_tokenize(text.lower())

    tokens = [word for word in tokens if word not in string.punctuation and word not in stop\_words]

    return ' '.join(tokens)

* Next, the code initializes a set of stop words using NLTK which helps to filter out common words that do not contribute to the sentiment analysis. function preprocess\_text() is defined to preprocess the text data. Inside this function the text is tokenized into individual words, converted to lowercase, and filtered to remove punctuation and stop words. The cleaned tokens are then joined back into a single string.

tfidf\_df = pd.DataFrame(tfidf\_features.toarray(), columns=tfidf\_vectorizer.get\_feature\_names\_out())

* Then we use TF-IDF (Term Frequency-Inverse Document Frequency) vectorizer from Scikit-learn to convert the cleaned text reviews into numerical features that the SVM (Support Vector Machine) model can process. The TF-IDF vectorizer is fit on the cleaned reviews and the resulting features are stored in tfidf\_features.

# Create and train the SVM classifier

svm\_model = SVC(kernel='linear')

svm\_model.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pred = svm\_model.predict(X\_test)

* Now we can split the dataset into features (X) and labels (y), where X contains the TF-IDF features, and y contains the sentiment labels. The data is then further divided into training and testing sets using an 80-20 split.
* An SVM classifier is created with a linear kernel and trained on the training data using svm\_model.fit(). After training, predictions are made on the test set, and the model’s performance is evaluated using accuracy and a classification report.

## 1.3 DistilBERT

* Similar to SVM, we are limiting our dataset to first 2000 rows for faster calculations.
* Using train\_test\_split we divide the dataset into training and test sets.

train\_df, test\_df = train\_test\_split(df, test\_size=0.2, random\_state=42, stratify=df['sentiment'])

* test\_size=0.2 specifies that 20% of the data should be allocated to the testing set, while 80% will be used for training.
* random\_state=42 random seed ensures that the results are reproducible. Using the same seed will yield the same split of the dataset every time you run the code.

tokenizer = DistilBertTokenizer.from\_pretrained('distilbert-base-uncased')

model = DistilBertForSequenceClassification.from\_pretrained('distilbert-base-uncased', num\_labels=2)

* DistilBertTokenizer class is a part of the Hugging Face Transformers library and is specifically designed for the DistilBERT model. It handles the preprocessing of text inputs and converts them into a format suitable for the model.
* from\_pretrained('distilbert-base-uncased') method loads a pre-trained tokenizer associated with the specified model (distilbert-base-uncased). The term "uncased" means that the tokenizer does not differentiate between uppercase and lowercase letters. For example, "Hello" and "hello" will be treated the same.

train\_encodings = tokenizer(train\_df['cleaned\_review'].tolist(), truncation=True, padding=True)

* This statement tokenizes and encodes the cleaned reviews from the training DataFrame (train\_df). It first converts the "cleaned\_review" column into a list of strings, which are then processed by the tokenizer. Padding is added to ensure all tokenized sequences are of equal length.

test\_encodings = tokenizer(test\_df['cleaned\_review'].tolist(), truncation=True, padding=True)

* This part retrieves the "cleaned\_review" column and converts it into a list of strings, which are then processed by the tokenizer. This process breaks each review into tokens, assigns numerical IDs to those tokens, and applies truncation so that any reviews exceeding the model's maximum input length are shortened appropriately.

class SentimentDataset(torch.utils.data.Dataset):

def \_\_init\_\_(self, encodings, labels):

self.encodings = encodings

self.labels = labels

def \_\_getitem\_\_(self, idx):

item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}

item['labels'] = torch.tensor(self.labels[idx])

return item

def \_\_len\_\_(self):

return len(self.labels)

* The SentimentDataset class creates a custom dataset for sentiment analysis in PyTorch. It initializes with tokenized encodings and corresponding labels, providing a way to access individual data samples using the \_\_getitem\_\_ method, which returns the token inputs and their labels as tensors. The \_\_len\_\_ method allows checking the total number of samples in the dataset.
* Then we prepare the training and testing datasets. Then, using the SentimentDataset class, we create train\_dataset with the tokenized training encodings and convert the sentiment labels from strings ('positive' and 'negative') to numerical values (1 and 0). Similarly, we create test\_dataset with the tokenized test encodings and their corresponding numerical labels.

training\_args = TrainingArguments(

output\_dir='./results',

num\_train\_epochs=3,

per\_device\_train\_batch\_size=8,

per\_device\_eval\_batch\_size=8,

warmup\_steps=500,

weight\_decay=0.01,

logging\_dir='./logs',

)

* This part sets up training configuration using the TrainingArguments class. output\_dir='./results' defines where to save model checkpoints and outputs and num\_train\_epochs=3 indicates the model will train for three complete passes over the dataset. The per\_device\_train\_batch\_size=8 and per\_device\_eval\_batch\_size=8 set the batch sizes for training and evaluation, respectively, for simultaneous processing of 8 samples at a time. By warmup\_steps=500, the learning rate gradually increases during the initial 500 steps to stabilize training, and weight\_decay=0.01 applies L2 regularization to help prevent overfitting by penalizing large weights.

# Trainer

trainer = Trainer(

model=model,

args=training\_args,

train\_dataset=train\_dataset,

eval\_dataset=test\_dataset,

)

# Train the model

trainer.train()

# Evaluate the model

predictions = trainer.predict(test\_dataset)

preds = torch.argmax(torch.tensor(predictions.predictions), axis=1)

* Next we initialize a Trainer instance from the Hugging Face Transformers library to manage the training and evaluation of the sentiment analysis model. It passes the pre-trained model, the previously defined training arguments, and the training and evaluation datasets to the Trainer.
* The trainer.train() method is then called to start the training process, allowing the model to learn from the training data. After training, the model is evaluated on the test dataset using trainer.predict(test\_dataset), which generates predictions for the test samples.
* The predicted class labels are obtained by applying torch.argmax() to the output logits by selecting the class with the highest score for each sample.

# 2. Model Performance and Error Analysis

## 2.1 Model Performance

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Model** | **Accuracy** | **Precision (Negative)** | **Recall (Negative)** | **F1-Score (Negative)** | **Precision (Positive)** | **Recall (Positive)** | **F1-Score (Positive)** | **Macro Avg** | **Weighted Avg** |
| Logistic Regression | 0.8595 | 0.88 | 0.84 | 0.86 | 0.84 | 0.88 | 0.86 | 0.86 | 0.86 |
| SVM | 0.8555 | 0.87 | 0.83 | 0.85 | 0.84 | 0.88 | 0.86 | 0.86 | 0.86 |
| DistilBERT | 0.855 | 0.9 | 0.8 | 0.85 | 0.82 | 0.91 | 0.86 | 0.86 | 0.85 |

* Logistic Regression achieved the highest accuracy at 0.8595, with balanced precision and recall for both negative (precision: 0.88, recall: 0.84) and positive (precision: 0.84, recall: 0.88) classes, resulting in F1-scores of 0.86 for both.
* SVM closely followed with an accuracy of 0.8555. It shows good precision for the negative class (0.87) but slightly lower recall (0.83). The positive class had a precision of 0.84 and recall of 0.88.
* DistilBERT had an accuracy of 0.8550, with the highest precision for the negative class at 0.90. However, its recall for negatives was lower at 0.80 with an F1-score of 0.85. It performed well in positive recall at 0.91, with a precision of 0.82.

## 2.2 Error Analysis with Confusion Matrix

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Model** | **True Negative (TN)** | **False Positive (FP)** | **Actual Positive (Total)** | **False Negative (FN)** | **True Positive (TP)** |
| **Logistic Regression** | 832 | 164 | 1004 | 117 | 887 |
| **SVM** | 827 | 169 | 1004 | 120 | 884 |
| **DistilBERT** | 167 | 32 | 201 | 24 | 177 |

**Logistic Regression:**

True Negatives (TN): The model correctly identified 832 negative instances.

False Positives (FP): It misclassified 164 negative instances as positive.

False Negatives (FN): There were 117 instances where positive sentiments were incorrectly classified as negative. This meas that while the model performs well overall, it struggles with certain positive reviews.

True Positives (TP): The model correctly identified 887 positive instances, reflecting solid performance in recognizing positive sentiments.

**SVM:**

True Negatives (TN): The SVM model had 827 true negatives which is slightly lower than Logistic Regression.

False Positives (FP): It recorded 169 false positives, this shows a tendency to misidentify negative instances as positive more frequently than Logistic Regression.

False Negatives (FN): The model misclassified 120 positive instances as negative, which indicates a similar challenge in identifying positive sentiments as Logistic Regression.

True Positives (TP): With 884 true positives, the SVM model performed well but slightly less effectively in identifying positive sentiments compared to Logistic Regression.

**DistilBERT:**

True Negatives (TN): DistilBERT demonstrated a strong ability to classify true negatives with 167 true negatives.

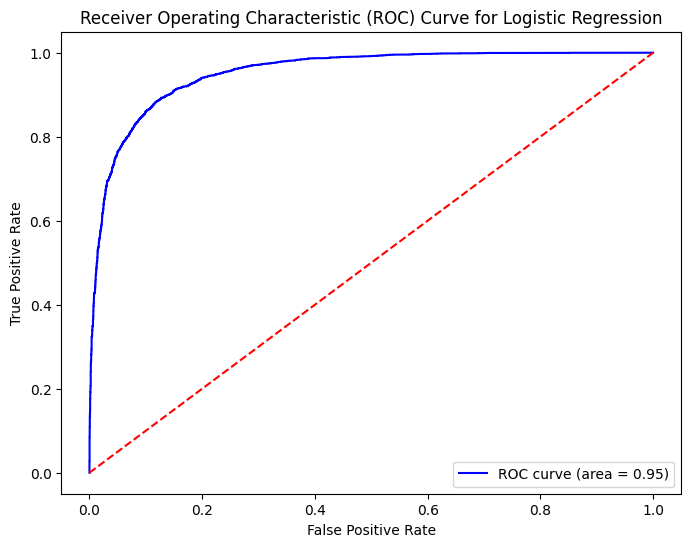
False Positives (FP): The model had 32 false positives, which is a good performance in avoiding misclassifications of negative sentiments as positive.

False Negatives (FN): It misclassified 24 positive instances as negative, which shows its strength in recognizing positive sentiments compared to the other models.

True Positives (TP): With 177 true positives, DistilBERT performed effectively in identifying positive sentiments.

# 3. ROC Visualization

## 3.1 Logistic Regression



The curve (shown in blue) has an AUC (Area Under the Curve) of 0.95, which is excellent for our logistic regression model. AUC ranges from 0 to 1, and 0.95 indicates strong predictive performance.

The diagonal red dashed line represents random chance (AUC = 0.5). Our model's curve is well above this line which means it performs much better than random guessing.

The curve rises steeply at first, indicating the model achieves a high true positive rate while maintaining a low false positive rate in the early thresholds.

## 3.2 Support Vector Machine

A graph of a curve

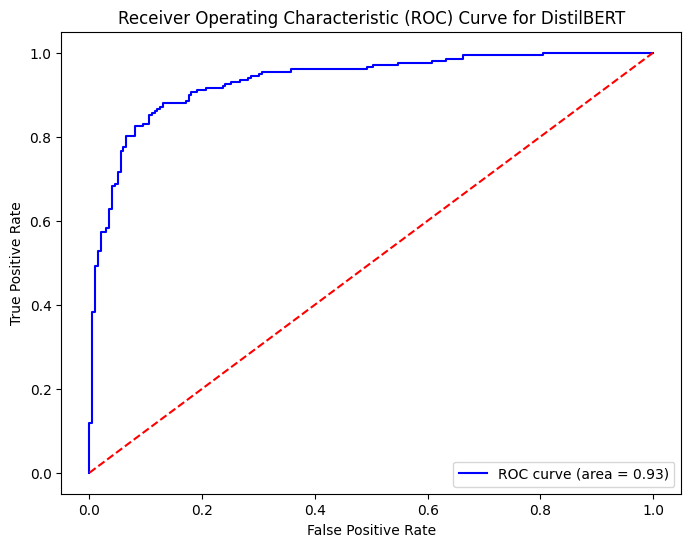
Description automatically generated with medium confidence

The Support Vector Machine (SVM) model demonstrates strong performance with an Area Under the Curve (AUC) of 0.94, which is a 94% probability that the model will correctly rank a randomly chosen positive instance higher than a negative one.

This represents a slight lower AUC score compared to logistic regression by 1%.

Both models exhibit smilar predictive capabilities, with the SVM's ROC curve featuring a characteristic steep initial ascent, which indicates the model achieves a high true positive rate while maintaining a low false positive rate at early classification thresholds.

## 3.3 DistilBERT



DistilBERT ROC curve analysis shows an AUC score of 0.93 slightly behind the SVM's score of 0.94.

The curve's most distinctive feature is its pronounced stepped or staircase pattern, particularly noticeable in the early portion where it makes several sharp turns, contrasting with the smoother curves observed in the other models.

The stepped pattern observed could be influenced by the fact that it was trained on a limited dataset of only the first 2000 rows, unlike the other models that utilized the full dataset. This restriction may have led to a lack of variation in the data and prevented the model from capturing the complete distribution of confidence scores. Consequently, the limited sample size could result in less stable predictions, which can cause the output probability to cluster around specific thresholds and creating distinct jumps in the ROC curve.

Despite these discrete jumps, the curve demonstrates a steep initial ascent (between 0.0 and 0.2 on the x-axis), indicating strong discriminative ability at strict classification thresholds.