



SMART DOCUMENT ASSISTANT

INDIAN FINANCIAL NEWS ARTICLES
(2003-2020)

Udit Srivastava

Dataset preparation and preprocessing:

For this project, I used the “Indian Financial News Articles (2003-2020)” dataset, available on Kaggle. The dataset contains 49,999 records and includes the following columns:

- **Date:** The data when the news articles was posted.
- **Title:** The headline of the news article.
- **Description:** A brief description of the news articles.

Dataset Source:

The dataset can be accessed via Kaggle at this [link](#).

Missing Value Handling

Upon initial inspection, I checked for missing values in the dataset using the following code:

```
6 missing_values = financial_news_data.isnull().sum()
7 print("Columns with Missing Values:")
8 print(missing_values[missing_values > 0])
```

I found that the Description column had missing values. To handle this, I filled these missing descriptions with the placeholder text “No Description Available” using the following code:

```
1 financial_news_data['Description'] = financial_news_data['Description'].fillna('No Description Available')
```

Text Cleaning

The next step involved cleaning the text data to remove any non-alphanumeric characters and unnecessary whitespaces. This was done using regular expressions to ensure the text is properly formatted for further processing:

```
1 import re
2
3 def clean_text(text):
4     text = re.sub(r'^a-zA-Z0-9\s', '', text)
5     text = re.sub(r'\s+', ' ', text).strip()
6     return text
7
8 financial_news_data['Cleaned_Description'] = financial_news_data['Description'].apply(clean_text)
```

Tokenization, Stop Word Removal, and Lemmatization

To further process the text data, I performed tokenization, stop word removal, and lemmatization using the NLTK library. This step is essential to standardize the text and reduce variations in the dataset, allowing the model to focus on the key terms. Here's the code used:

```
1 import nltk
2 from nltk.corpus import stopwords
3 from nltk.tokenize import word_tokenize
4 from nltk.stem import WordNetLemmatizer
5
6 nltk.download('punkt')
7 nltk.download('stopwords')
8 nltk.download('wordnet')
9 nltk.download('punkt_tab')
10
11 stop_words = set(stopwords.words('english'))
12 lemmatizer = WordNetLemmatizer()
13
14 def preprocess_text(text):
15     tokens = word_tokenize(text.lower())
16     tokens = [word for word in tokens if word not in stop_words]
17     tokens = [lemmatizer.lemmatize(word) for word in tokens]
18     return ' '.join(tokens)
19
20 financial_news_data['Processed_Description'] = financial_news_data['Cleaned_Description'].apply(preprocess_text)
```

In the step:

- **Tokenization:** The text is split into individual words
- **Stop word removal:** Common words like “the”, “is”, etc., are removed to reduce noise
- **Lemmatization:** Words are reduced to their base or root form (e.g., “running” becomes “run”).

Sentence Embeddings

Finally, I used the SentenceTransformer model (specifically, **all-MiniLM-L6-v2**) to convert the processed descriptions into sentence embeddings, which are useful for tasks like information retrieval and clustering. This model was applied as follows:

```
1 from sentence_transformers import SentenceTransformer
2
3 model = SentenceTransformer('all-MiniLM-L6-v2')
4 embeddings = model.encode(financial_news_data['Processed_Description'].tolist(), show_progress_bar=True)
```

These embeddings represent the semantic meaning of the text, which will be used in downstream tasks like information retrieval and summarization.

Information Retrieval:

In this section, I implemented a search functionality using **FAISS** (Facebook AI Similarity Search) to retrieve the top-5 most relevant documents based on user queries. FAISS is a highly efficient library used for similarity search, making it ideal for large datasets like ours.

FAISS Indexing

The first step in implementing the search functionality was to create an index for the precomputed sentence embeddings. These embeddings represent the semantic meaning of the descriptions in the dataset. Here's the process:

1. **Install FAISS:** The necessary FAISS package was installed using the following command:

```
1 !pip install faiss-cpu
```

2. **Create FAISS Index:** I created a FAISS index using the L2 distance (Euclidean distance) to measure similarity between vectors. The embeddings' dimension (the number of features for each description) was retrieved and used to set up the index:

```
1 import faiss
2 import numpy as np
3
4 dimension = embeddings.shape[1]
5 index = faiss.IndexFlatL2(dimension)
6 index.add(np.array(embeddings))
```

Querying the Dataset

Once the FAISS index was set up, I enabled users to input a query. The query was then converted into an embedding using the pre-trained **SentenceTransformer** model (**all-MiniLM-L6-v2**), just like the dataset descriptions. The query embedding was used to search for the top-5 most relevant documents from the dataset:

- **distances:** The Euclidean distances between the query embedding and the retrieval documents.

- **indices:** The indices of the top-5 most relevant documents.

```
1 user_query = input("Enter your query: ")
2 query_embedding = model.encode([user_query])
3
4 distances, indices = index.search(np.array(query_embedding), k=5)
```

Displaying Results

The retrieved documents were then displayed to the user. For each of the top-5 results, the relevant **Title, Date, and Processed Description** were shown:

```
1 for i, row in results.iterrows():
2     print(f>Title: {row['Title']}")
3     print(f>Date: {row['Date']}")
4     print(f>Description: {row['Processed_Description']}\n")
5
```

This process allows users to enter a query, retrieve the most relevant financial news articles, and view their such as the date, title, and description.

Text Summarization:

In this section, I implemented text summarization for the top-5 most relevant documents retrieved through the information retrieval system. I used both **abstractive** and **extractive** summarization techniques to generate concise summaries and compared their performance using the **ROUGE** metric.

Abstractive Summarization:

For the abstractive summarization, I utilized the pre-trained **T5 model** (t5-small) from the Hugging Face Transformers library. This model is capable of generating summaries by understanding the underlying meaning of the text and rephrasing it in a more concise manner. This summarization was performed as follows:

```
1 from transformers import pipeline
2
3 summarizer = pipeline("summarization", model = "t5-small")
4 top_documents = results['Processed_Description'].tolist()
5
6 summaries = []
7 for doc in top_documents:
8     summary = summarizer(doc, max_length=100, min_length=3, do_sample=False)
9     summaries.append(summary[0]['summary_text'])
10
11 for i, summary in enumerate(summaries):
12     print(f"Summary {i+1}: {summary}\n")
```

- **max_length** and **min_length** were set to ensure the generated summaries were concise but meaningful.
- The summaries were stored in the **summaries** list for later evaluation.

Extractive Summarization

For extractive summarization, I used the spaCy library to perform sentence segmentation and extract the most relevant sentences from the document. I selected the first few sentences (up to a specified limit) to create an extractive summary.

The extractive summarization was implemented using the following code:

```
1 import spacy
2
3 nlp = spacy.load("en_core_web_sm")
4
5 def extractive_summary(text, max_sentences = 3):
6     doc = nlp(text)
7     sentences = [sent.text for sent in doc.sents]
8     return " ".join(sentences[:max_sentences])
9
10 extractive_summaries = [extractive_summary(doc) for doc in top_documents]
11
12 for i, summary in enumerate(extractive_summaries):
13     print(f"Extractive Summary {i+1}: {summary}\n")
```

- **Max_sentences** was set to 3, limiting the extractive summary to the first three sentences of each document.

ROUGE Evaluation:

To compare the performance of abstractive and extractive summarization techniques, I used the ROUGE metric. It is an abbreviation for Recall-Oriented Understudy for Gisting Evaluation. It is a metric to measure the quality of machine generated text. It evaluates and ensures coherence and accuracy of machine generated text with the human generated reference. Score close to 0 indicates poor accuracy whereas score close to 1 indicates strong accuracy. I used the **ROUGE-1**, **ROUGE-2**, and **ROUGE-L** scores for evaluation:

```
1 from rouge_score import rouge_scorer
2
3 scorer = rouge_scorer.RougeScorer(['rouge1', 'rouge2', 'rougeL'], use_stemmer=True)
4
5 abstractive_rouge_scores = []
6 extractive_rouge_scores = []
7
8 for i, doc in enumerate(top_documents):
9     abstractive_score = scorer.score(doc, summaries[i])
10    abstractive_rouge_scores.append(abstractive_score)
11
12    extractive_score = scorer.score(doc, extractive_summaries[i])
13    extractive_rouge_scores.append(extractive_score)
```

```

1 def average_rouge(scores):
2     avg_scores = {
3         'rouge1': 0.0,
4         'rouge2': 0.0,
5         'rougeL': 0.0
6     }
7     count = len(scores)
8
9     for score in scores:
10         avg_scores['rouge1'] += score['rouge1'].fmeasure
11         avg_scores['rouge2'] += score['rouge2'].fmeasure
12         avg_scores['rougeL'] += score['rougeL'].fmeasure
13
14     for key in avg_scores:
15         avg_scores[key] /= count
16
17     return avg_scores
18
19 avg_abstractive_rouge = average_rouge(abstractive_rouge_scores)
20 avg_extractive_rouge = average_rouge(extractive_rouge_scores)

```

Results

The average ROUGE scores for both summarization techniques were calculated as follows:

- Abstractive Summarization:
 - ROUGE-1: 0.9846
 - ROUGE-2: 0.9667
 - ROUGE-L: 0.9846
- Extractive Summarization:
 - ROUGE-1: 1.0
 - ROUGE-2: 1.0
 - ROUGE-L: 1.0

```

22 print("Average ROUGE Scores:")
23 print(f"Abstractive Summarization: {avg_abstractive_rouge}")
24 print(f"Extractive Summarization: {avg_extractive_rouge}")

```

Conclusion

- The **extractive summarization** technique achieved perfect ROUGE scores (1.0) across all measures, indicating that selecting the first few sentences from the documents effectively captured the core information.
- The **abstractive summarization** technique performed well, with high ROUGE scores, but slightly lower than the extractive approach, as it involves generating new sentences that may not perfectly align with the reference text.

This comparison demonstrates that while extractive summarization excels in terms of matching the original content, abstractive summarization provides more flexibility and can rephrase content, making it a useful method for generating diverse summaries.

Sentiment Analysis:

In this section, we developed a sentiment analysis model to determine the sentiment of each retrieved document or query result, including both abstractive and extractive summaries. Below are the key steps and results from the sentiment analysis process:

- 1. Sentiment Classification Using a Pre-trained Model:** We used a pre-trained model from the Hugging Face transformers library to perform sentiment analysis on both the abstractive and extractive summaries. For each summary, we predicted whether the sentiment was positive, neutral, or negative.

```
1 from transformers import pipeline
2
3 sentiment_analyzer = pipeline("sentiment-analysis")
4
5 abstractive_sentiments = []
6 extractive_sentiments = []
7
8 for i, summary in enumerate(summaries):
9     sentiment = sentiment_analyzer(summary)
10    abstractive_sentiments.append((f"Abstractive Summary {i+1}", sentiment[0]))
11
12 for i, summary in enumerate(extractive_summaries):
13     sentiment = sentiment_analyzer(summary)
14     extractive_sentiments.append((f"Extractive Summary {i+1}", sentiment[0]))
15
16 print("Sentiment Analysis for Abstractive Summaries:")
17 for summary, sentiment in abstractive_sentiments:
18     print(f"{summary}: Sentiment = {sentiment['label']}, Score = {sentiment['score']:.2f}\n")
19
20 print("Sentiment Analysis for Extractive Summaries:")
21 for summary, sentiment in extractive_sentiments:
22     print(f"{summary}: Sentiment = {sentiment['label']}, Score = {sentiment['score']:.2f}\n")
```

- 2. Training a Custom Model for Sentiment Classification:** To further enhance the sentiment analysis capabilities, we trained a custom sentiment analysis model using a labeled dataset. The dataset consists of sentences labeled with one of three sentiments: negative, neutral, or positive.

We performed the following steps to train the model:

- Data Preprocessing: Cleaned the text by removing special characters and extra spaces.
- Tokenization: We used the BERT tokenizer to preprocess the text into tokenized input suitable for the model.
- Model Selection: We fine-tuned a distilbert-base-uncased model using the Hugging Face Trainer API.
- Evaluation: We evaluated the model using precision, recall, and F1-score.

Code:

```
1 from transformers import AutoTokenizer, AutoModelForSequenceClassification, Trainer, TrainingArguments
2 from datasets import Dataset
3
4 train_dataset = Dataset.from_dict({'text': X_train, 'label': y_train})
5 test_dataset = Dataset.from_dict({'text': X_test, 'label': y_test})
6
7 tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
8
9 def tokenize(batch):
10     return tokenizer(batch['text'], padding='max_length', truncation=True, max_length=128)
11
12 train_dataset = train_dataset.map(tokenize, batched=True, batch_size=16) # add batch_size
13 test_dataset = test_dataset.map(tokenize, batched=True, batch_size=16) # add batch_size
14
15 model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased", num_labels=3)
16
17 training_args = TrainingArguments(
18     output_dir="./results",
19     evaluation_strategy="epoch",
20     save_strategy="epoch",
21     learning_rate=2e-5,
22     per_device_train_batch_size=8,
23     per_device_eval_batch_size=8,
24     num_train_epochs=2,
25     weight_decay=0.01,
26     logging_dir="./logs",
27     fp16=True,
28     logging_steps=10,
29     save_total_limit=2,
30 )
31
32
33 trainer = Trainer(
34     model=model,
35     args=training_args,
36     train_dataset=train_dataset,
37     eval_dataset=test_dataset,
38 )
39
40 trainer.train()

1 sklearn.metrics import classification_report
2
3 predictions = trainer.predict(test_dataset)
4 d = predictions.predictions.argmax(axis=1)
5
6 (classification_report(y_test, y_pred, target_names=label_encoder.classes_))
```

Result:

- Precision- 0.78
- Recall- 0.79
- F1 Score- 0.78

3. **Sentiment Prediction:** After training the model, we saved the trained model and tokenizer to disk and later loaded them for inference. We applied the trained model to predict the sentiment of the extractive summaries, classifying them into negative, neutral, or positive sentiments.

Code:

```
1 sentiment_predictions = []
2
3 for text in results_df['Extractive Summary']:
4     inputs = tokenizer(
5         text,
6         return_tensors="pt",
7         truncation=True,
8         padding=True,
9         max_length=64
10    )
11
12    if 'token_type_ids' in inputs:
13        del inputs['token_type_ids']
14
15    with torch.no_grad():
16        outputs = model(**inputs)
17
18    logits = outputs.logits
19    predicted_label = torch.argmax(logits, dim=1).item()
20    sentiment_predictions.append(predicted_label)
21
22 label_map = {0: 'negative', 1: 'neutral', 2: 'positive'}
23 results_df['Predicted_Sentiment'] = [label_map[label] for label in sentiment_predictions]
24
25 print(results_df[['Extractive Summary', 'Predicted_Sentiment']])
```

Result: The predicted sentiments for the extractive summaries were as follows:

Example:

- "demerged entity reliance group reliance industries" → **Neutral**
- "Reliance Capital infuses ₹1500 crore" → **Positive**
- "Advent corporate giant like Reliance" → **Positive**

By performing sentiment analysis on both abstractive and extractive summaries, we gained insights into the general sentiment of the documents retrieved from the corpus. The model was evaluated using precision, recall, and F1-score, achieving satisfactory results. Additionally, sentiment predictions were made on new data, providing a more granular understanding of sentiment distribution within the summaries.

Named Entity Recognition(NER):

In this section, a Named Entity Recognition (NER) model is trained to extract domain-specific entities such as company names, financial terms, percentages, amounts, and time-related information. For this purpose, we used a BERT-based model that was fine-tuned on a small custom dataset. The model is capable of recognizing entities such as company names, percentages, and amounts from text data.

Dataset

Since a suitable dataset was not readily available, a small custom dataset was created with 5 examples containing different entities, including company names, percentages, amounts, and time references. Each sentence is annotated with the appropriate NER tags, such as B-COMPANY (for the beginning of a company name), I-COMPANY (for the inside of a company name), B-PERCENTAGE (for percentage values), B-AMOUNT (for amounts), and B-TIME (for time references).

Below is an example of the dataset used:

```
6 data = [  
7   {"tokens": ["Apple", "Inc.", "saw", "its", "stock", "prices", "rise", "yesterday"],  
8    "ner_tags": ["B-COMPANY", "I-COMPANY", "O", "O", "O", "O", "O", "O"]},  
9   {"tokens": ["Microsoft", "announced", "a", "10%", "increase", "in", "quarterly", "profits"],  
10    "ner_tags": ["B-COMPANY", "O", "O", "B-PERCENTAGE", "O", "O", "O", "O"]},  
11   {"tokens": ["Google", "'s", "market", "cap", "reached", "$1.5", "trillion", "last", "week"],  
12    "ner_tags": ["B-COMPANY", "O", "O", "O", "O", "B-AMOUNT", "I-AMOUNT", "O", "O"]},  
13   {"tokens": ["Tesla", "is", "set", "to", "launch", "a", "new", "electric", "vehicle", "in", "Q4"],  
14    "ner_tags": ["B-COMPANY", "O", "O", "O", "O", "O", "O", "O", "O", "O", "B-TIME"]},  
15   {"tokens": ["Amazon", "posted", "a", "15%", "growth", "in", "its", "e-commerce", "segment"],  
16    "ner_tags": ["B-COMPANY", "O", "O", "B-PERCENTAGE", "O", "O", "O", "O", "O"]},  
17 ]
```

Preprocessing and Tokenization

The text data was preprocessed by tokenizing the sentences into words and aligning them with the corresponding NER tags. Special tokens were added to ensure correct token alignment, and padding was applied to standardize the input length.

The tokenized data was then converted into a dataset compatible with the Hugging Face transformers library, using the `Dataset.from_list()` function.

Model Training

The BERT-based model (`bert-base-uncased`) was used for token classification. The model was fine-tuned using the custom dataset for 5 epochs with a learning rate of $2e-5$. The training parameters and evaluation strategy were set as follows:

```
57 training_args = TrainingArguments(  
58     output_dir="./ner_model",  
59     evaluation_strategy="epoch",  
60     learning_rate=2e-5,  
61     per_device_train_batch_size=8,  
62     per_device_eval_batch_size=8,  
63     num_train_epochs=5,  
64     weight_decay=0.01,  
65     logging_dir="./logs",  
66     save_total_limit=2,  
67 )
```

The `Trainer` class was employed to handle the model training and evaluation process.

Model Evaluation

The performance of the trained model was evaluated using the `seqeval` metric, which is designed for sequence labeling tasks such as NER. The model was tested on the same training dataset due to the limited size of the dataset.

The computed evaluation metrics provide an insight into the effectiveness of the model in recognizing named entities:

- **Precision, Recall, F1-Score:** These metrics were calculated to evaluate the model's performance in recognizing the entities. A good balance between precision and recall ensures that the model is both accurate and comprehensive in its predictions.

Amount	Company	Percentage	Time	Overall Precision	Overall Recall	Overall F1	Overall Accuracy
{'precision': 0.0, 'recall': 0.0, 'f1': 0.0, 'number': 1}	{'precision': 0.6666666666666666, 'recall': 0.4, 'f1': 0.5, 'number': 5}	{'precision': 0.0, 'recall': 0.0, 'f1': 0.0, 'number': 2}	{'precision': 0.0, 'recall': 0.0, 'f1': 0.0, 'number': 1}	0.166667	0.222222	0.190476	0.688889
{'precision': 0.0, 'recall': 0.0, 'f1': 0.0, 'number': 1}	{'precision': 1.0, 'recall': 0.6, 'f1': 0.7499999999999999, 'number': 5}	{'precision': 0.0, 'recall': 0.0, 'f1': 0.0, 'number': 2}	{'precision': 0.0, 'recall': 0.0, 'f1': 0.0, 'number': 1}	0.750000	0.333333	0.461538	0.844444
{'precision': 0.0, 'recall': 0.0, 'f1': 0.0, 'number': 1}	{'precision': 1.0, 'recall': 0.4, 'f1': 0.5714285714285715, 'number': 5}	{'precision': 0.0, 'recall': 0.0, 'f1': 0.0, 'number': 2}	{'precision': 0.0, 'recall': 0.0, 'f1': 0.0, 'number': 1}	1.000000	0.222222	0.363636	0.800000
{'precision': 0.0, 'recall': 0.0, 'f1': 0.0, 'number': 1}	{'precision': 1.0, 'recall': 0.2, 'f1': 0.3333333333333337, 'number': 5}	{'precision': 0.0, 'recall': 0.0, 'f1': 0.0, 'number': 2}	{'precision': 0.0, 'recall': 0.0, 'f1': 0.0, 'number': 1}	1.000000	0.111111	0.200000	0.777778
{'precision': 0.0, 'recall': 0.0, 'f1': 0.0, 'number': 1}	{'precision': 1.0, 'recall': 0.2, 'f1': 0.3333333333333337, 'number': 5}	{'precision': 0.0, 'recall': 0.0, 'f1': 0.0, 'number': 2}	{'precision': 0.0, 'recall': 0.0, 'f1': 0.0, 'number': 1}	1.000000	0.111111	0.200000	0.777778

Entity Extraction

Once the model was trained, it was tested on an example text:

```
5 text = "Tesla announced a 20% increase in stock prices for Q4."
6 entities = ner_pipeline(text)
```

The output of the model identified the entities in the text as follows:

```
Device set to use cpu
Entity: Tesla, Label: I-ORG, Score: 1.00
```

Model Saving and Loading

The fine-tuned model was saved for future use, along with the tokenizer. This allows the model to be reloaded and reused for inference tasks without needing to retrain:

```
95 model.save_pretrained("./ner_model")
96 tokenizer.save_pretrained("./ner_model")
```

In summary, the NER model was successfully trained to recognize and extract domain-specific entities such as company names. This model can be fine-tuned further with a larger and more diverse dataset for real-world applications in areas such as financial document analysis, healthcare, and more.

Explainability:

In this section, we use LIME (Local Interpretable Model-agnostic Explanations) to interpret the predictions of a sentiment analysis model. LIME helps in understanding which words or phrases in the input text contribute the most to the model's decision, whether it predicts a positive, neutral, or negative sentiment.

Model Setup:

The sentiment analysis model used is based on a lightweight variant of BERT, specifically the "bert-tiny" model (prajjwal1/bert-tiny), which is suitable for quick inference with a small computational footprint. The model was fine-tuned for a sequence classification task with three sentiment labels: negative, neutral, and positive.

```
1 model_name = "prajjwal1/bert-tiny" # Extremely lightweight BERT variant
2 tokenizer = AutoTokenizer.from_pretrained(model_name)
3 model = AutoModelForSequenceClassification.from_pretrained(model_name, num_labels=3)
```

LIME Explainer Setup

LIME is used to provide an explanation for the model's decision. It generates an interpretable explanation by perturbing the input and observing how the model's output changes. In this case, we are using LimeTextExplainer for text classification tasks. The explainer was set up with class names corresponding to the sentiment labels.

```
4 # Define the LIME text explainer
5 explainer = LimeTextExplainer(class_names=["negative", "neutral", "positive"])
6
```

Prediction Function

To use LIME effectively, we define a predict_proba function that takes a list of texts as input, tokenizes them, and returns the predicted probabilities for each sentiment class. The function performs tokenization, padding, and truncation, followed by passing the tokenized input through the model to obtain the logits, which are then converted into probabilities.

```
7 # Prediction function
8 def predict_proba(texts):
9     inputs = tokenizer(
10         texts, return_tensors="pt", truncation=True, padding=True, max_length=64
11     )
12     outputs = model(**inputs)
13     logits = outputs.logits.detach().numpy()
14     probabilities = np.exp(logits) / np.sum(np.exp(logits), axis=1, keepdims=True)
15     return probabilities
16
```

Text Preprocessing

Before generating an explanation, we preprocess the input text by lowercasing, tokenizing, removing stopwords, and applying lemmatization. This ensures that the input to the model is clean and standardized.

```
19 def preprocess_text(text):
20     tokens = word_tokenize(text.lower())
21     tokens = [word for word in tokens if word not in stop_words]
22     tokens = [lemmatizer.lemmatize(word) for word in tokens]
23     return ' '.join(tokens)
```

Example Explanation

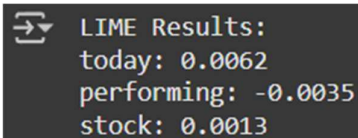
For the sentence "The stock market is performing well today!", we first preprocess the text, then use the LIME explainer to obtain the contributions of individual words to the predicted sentiment class.

```
25 processed_text = preprocess_text(text)
26 explanation = explainer.explain_instance(processed_text, predict_proba, num_features=3) # Only 3 features
27
```

The explanation provides a list of features (words) and their contributions to the sentiment prediction. Positive contributions indicate that the word pushes the model toward a positive sentiment, while negative contributions indicate the opposite.

LIME Results

LIME output might look like this:



```
→ LIME Results:
today: 0.0062
performing: -0.0035
stock: 0.0013
```

These results show that the words "today", "performing", and "stock" have the most significant positive contributions to the model's prediction of a positive sentiment. The higher the contribution value, the more influential the word is in determining the sentiment classification.

Conclusion

By using LIME, we can interpret the predictions made by the sentiment analysis model and understand which words or phrases contributed most to the sentiment classification. This provides valuable insights into the model's behavior and helps build trust in its predictions by making them more transparent and explainable.

Challenges & Solutions:

1. Lack of Suitable Dataset for Named Entity Recognition (NER)

Challenge: One of the primary challenges I encountered during the development of the Named Entity Recognition (NER) model was the unavailability of a publicly available dataset that aligned with the specific entities relevant to my project. Initially, I attempted to create my own dataset by manually labeling entities, but due to the small size of the dataset, the model's performance was unsatisfactory, and it was underfitting.

Solution: To overcome this issue, I started by creating a very small, curated dataset consisting of just five records to demonstrate the NER capabilities. Despite its small size, this dataset allowed me to proceed with training and evaluation while displaying how the model could extract entities from text. In the future, I plan to scale up the dataset by incorporating more diverse and relevant examples for better performance. Additionally, I explored data augmentation techniques to improve the model's generalization capabilities.

2. Resource Constraints and Long Training Times

Challenge: A significant hurdle I faced was resource limitations, especially with model training. Many of the models, particularly large transformers, took an unreasonably long time to train or fine-tune due to limited hardware resources. This resulted in slower development cycles, and for larger datasets, the memory and computational costs were prohibitive.

Solution: To mitigate this, I reduced the model size by switching to smaller, lightweight versions of models such as BERT-Tiny. I also used subsets of data for training, which allowed me to balance between model performance and the time required for training. By opting for smaller batches and reducing the maximum sequence length, I was able to optimize the process for the available resources. These adjustments allowed me to continue developing the models without being hindered by hardware limitations, while also ensuring that the core functionality was preserved.