Stock Market Trend Prediction Models

Provider: Saifullah Shoaib (representing SimplyAI)

Mentor/Client: Dr. Beibei Cheng (Applied Scientist at Microsoft)

Contents

1. Scenario Introduction

(The details of this section are fictitious and used to make the assignment more similar to real-world projects).

We at SimplyAI strive to provide our clients with progressive solutions to their unique problems utilizing the most cutting-edge AI technologies. This report summarizes the findings of the study requested by our client, Dr. Beibei Cheng (Applied Scientist at Microsoft), affiliated with Missouri University of Science and Technology.

The objective of this study was to develop for our client a system that can accurately predict stock market trends as increasing or decreasing. To accomplish this goal, we attempted to analyze daily news articles using Natural Language Processing (NLP) techniques and machine learning models. In this report, we explore various approaches for feature extraction, model building, and evaluation.

2. Data Collection and Preprocessing

2.1 Data Collection

There are many possible paths that can be taken when it comes to addressing the goal of stock market prediction. One particular option that we pursue in this study is analyzing news headlines for indications of stock market growth or decline. The reason for this selection is that news headlines may often contain information regarding major topics, including financial news such as stock market crashes or booms. We attempt to utilize this information to help us predict the stock market trends.

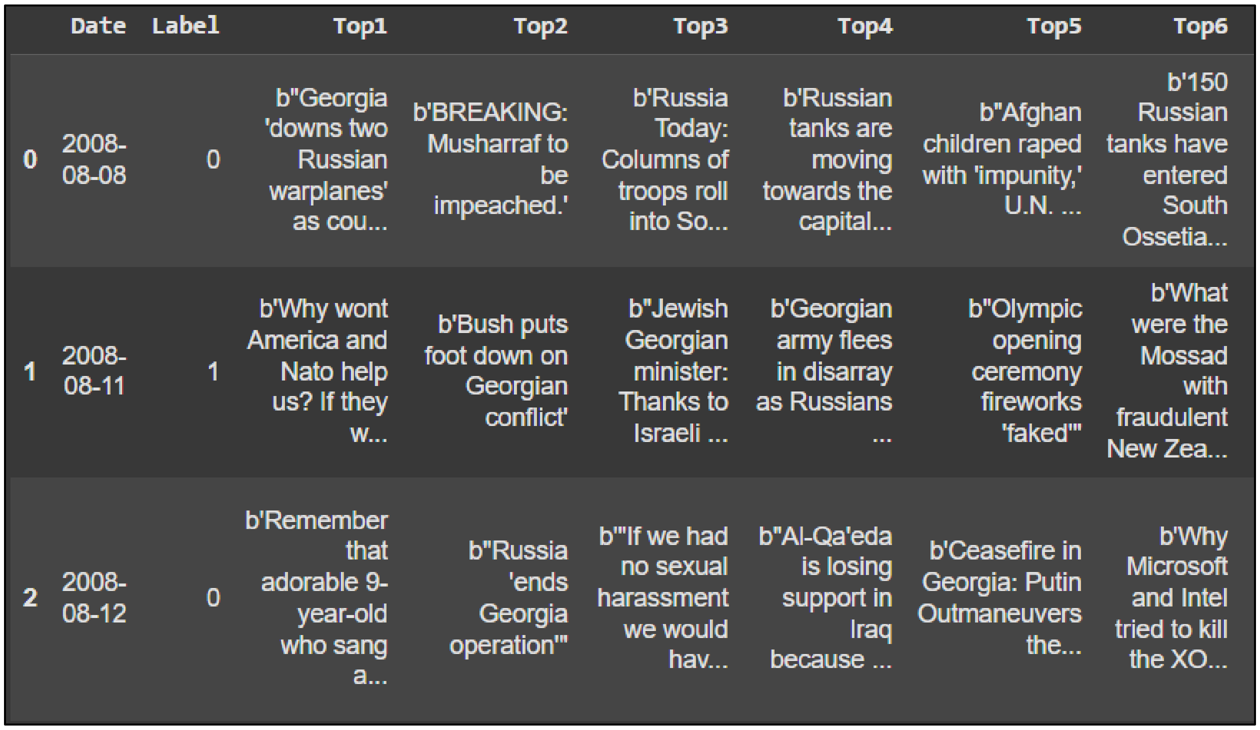
More specifically, the client provided us with the "Combined_News_DJIA.csv" dataset. It is a dataset in the CSV (comma separated values) format containing 27 columns:

(a) Column 1: Date – this column contains the entry date, with values ranging from 2008-08-08 to 2016-07-01. Additionally, only working business days are included (weekends and holidays are excluded).

(b) Column 2: Label – this column is a binary indicator of the stock market trend, with a 1 if the Dow Jones Industrial Average (DJIA) increased or stayed the same and a 0 if it decreased.

(c) Columns 3-27: Top1, Top2, … , Top25 – these columns contain the top 25 news headlines for the particular entry date. It is important to note that there are some null values (NaN) included.

A total of 1,989 entries are included in the dataset. A sample snapshot of the data is included below and the full dataset is included as an attachment with this report submission.



	Date	Label	Top1	Top2	Top3	Top4	Top5	Top6
0	2008-08-08	0	b"Georgia 'downs two Russian warplanes' as cou...	b'BREAKING: Musharraf to be impeached.'	b'Russia Today: Columns of troops roll into So...	b'Russian tanks are moving towards the capital...	b"Afghan children raped with 'impunity,' U.N. ...	b'150 Russian tanks have entered South Ossetia...
1	2008-08-11	1	b'Why wont America and Nato help us? If they w...	b'Bush puts foot down on Georgian conflict'	b"Jewish Georgian minister: Thanks to Israeli ...	b'Georgian army flees in disarray as Russians ...	b"Olympic opening ceremony fireworks 'faked'"	b'What were the Mossad with fraudulent New Zea...
2	2008-08-12	0	b'Remember that adorable 9-year-old who sang a...	b"Russia 'ends Georgia operation'"	b'"If we had no sexual harassment we would hav...	b"Al-Qa'eda is losing support in Iraq because ...	b'Ceasefire in Georgia: Putin Outmaneuvers the...	b'Why Microsoft and Intel tried to kill the XO...

For this study, we use Google Colab, Google's web service that utilizes GPUs to run hosted Jupyter notebooks. To use the dataset in Google Colab, we must make it accessible as is done in the following code block:

```
# downloading the "Combined_News_DJIA.csv" file
!pip install gdown
!gdown 1tTLkFuUUdJFrwDjjoFpotNsjB9W6y0Od
```

Thereafter, we must convert the csv dataset into a Pandas dataframe for ease of use and manipulation.

```
import pandas as pd

data = pd.read_csv("Combined_News_DJIA.csv")
data.head()
```

2.2 Preprocessing

We applied a number of preprocessing steps to allow the dataset to be more usable and compatible with our machine learning models. More specifically, we completed the following steps.

(A) Combining news headlines

The first step we took was to combine all 25 headlines into a single text string. In the way, we create a single text field from which we can extract various features. This is shown in the code below:

```
# combine all headlines (top1... top25) into 'headlines_combined'
data['headlines_combined'] = data.iloc[:, 2:].apply(lambda row: '
'.join(str(x) for x in row.values if pd.notnull(x)), axis=1)
```

(B) Training/test split

We used the dataset to both train and then evaluate our models. For training, data from 2008-08-08 to 2014-12-31 was used, while for testing and evaluating we utilized data from 2015-01-02 to 2016-07-01. This was following the desires of the client as specified in their instructions to us (assignment specification). The related code is shown below:

```
# split train and test sets according to assignment specifications
train_data = data[(data['Date'] >= '2008-08-08') & (data['Date'] <= '2014-
12-31')]
test_data = data[(data['Date'] >= '2015-01-02') & (data['Date'] <= '2016-
07-01')]
```

(C) Extracting from dataframe

Thereafter, we extract the relevant parts of the dataframe and save them in appropriate variables. We extract the ‘headlines_combined’ column and save it as X_train and X_test, while we save the ‘Label’ column as y_train and y_test, as shown here:

```
# extract features and labels from dataframe
X_train, y_train = train_data['headlines_combined'], train_data['Label']
X_test, y_test = test_data['headlines_combined'], test_data['Label']
```

(D) Cleaning data

The last preprocessing step was to clean and prepare the data for the subsequent analyses. The following data cleaning steps we completed:

- Converting data to all lowercase.
- Removing special characters.
- Tokenizing the data.
- Removing stop words.

These data cleaning steps are illustrated in the code that follows:

```
import re
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import nltk
nltk.download('punkt')
nltk.download('punkt_tab')
nltk.download('stopwords')

stop_words = set(stopwords.words('english'))

def preprocess_text(text):
    text = text.lower() # converts to lowercase
    text = re.sub(r'[^a-zA-Z\s]', '', text) # removes special characters
    words = word_tokenize(text) # tokenizes text
    words = [word for word in words if word not in stop_words] # removes
stopwords
    return ' '.join(words)

# preprocess data
X_train = X_train.apply(preprocess_text)
X_test = X_test.apply(preprocess_text)
```

3. Model Selection and Implementation

SimplyAI utilized several state-of-the-art Natural Language Processing (NLP) techniques to help address the client's needs. In specific, the following steps were taken:

- Step 1 – Bag of Words (BoW) and Simple TF-IDF
- Step 2 – Word Embeddings (GloVe)
- Step 3 – LSTM Model (using word embeddings)
- Step 4 – Open-source LLMs

We will walk through each of these steps in more detail one-by-one.

Step 1 – Bag of Words (BoW) and Simple TF-IDF

(A) Bag of Words (BoW)

In step 1, two related NLP techniques were evaluated. The first was the Bag of Words (BoW) technique which is a simple and widely used technique for text representation in NLP. It converts text into numerical data by creating a vocabulary of all unique words in the dataset and counting the frequency of each word in a given text. The result is a sparse vector where each element corresponds to a word in the vocabulary, and the value represents its occurrence in the text. BoW does not capture word order or context, making it suitable for basic text classification tasks but limited in understanding nuanced language.

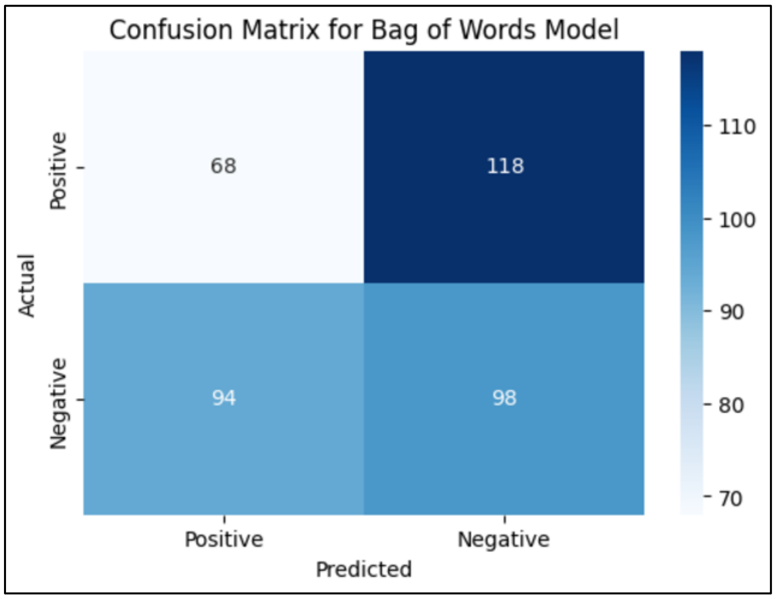
The implementation for this technique is shown below. There are a few key points to note. Firstly, we use scikit-learn's CountVectorizer to vectorize the news headlines. In other words, this converts the data from text to the BoW's feature space. After this, we train a logistic regression model (since it is a binary classification task) based off of this new BoW's feature space. Lastly, we evaluate this model and display related visualizations. In terms of evaluation metrics, we print the (a) accuracy, (b) precision, (c) recall, (d) F1 score, and (e) the confusion matrix along with its visualization. The related code is below:

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# convert data from text to Bag of Words feature space
bow_vectorizer = CountVectorizer()
X_train_bow = bow_vectorizer.fit_transform(X_train)
X_test_bow = bow_vectorizer.transform(X_test)

# building and training logistic regression model
model_bow = LogisticRegression(max_iter=200)
model_bow.fit(X_train_bow, y_train)

# predictions
y_pred_bow = model_bow.predict(X_test_bow)

# evaluation
print("Bag of Words Accuracy:", accuracy_score(y_test, y_pred_bow))
print("Precision:", precision_score(y_test, y_pred_bow))
print("Recall:", recall_score(y_test, y_pred_bow))
print("F1 Score:", f1_score(y_test, y_pred_bow))

conf_matrix = confusion_matrix(y_test, y_pred_bow)
print("Confusion Matrix:\n", conf_matrix)

# visualization
plt.figure(figsize=(6, 4))
```

```
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues",
xticklabels=["Positive", "Negative"], yticklabels=["Positive",
"Negative"])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix for Bag of Words Model")
plt.show()
```

The results of this approach are as follows:

```
Bag of Words Accuracy: 0.4391534391534391 3
Precision: 0.4537037037037037
Recall: 0.5104166666666666
F1 Score: 0.4803921568627451
Confusion Matrix:
 [[ 68 118]
 [ 94  98]]
```



The results are discussed in more detail later, but we will suffice by saying that the performance is very poor.

(B) Simple TF-IDF

The second NLP approach evaluated was the Simple TF-IDF (Term Frequency – Inverse Document Frequency) technique. This approach builds on Bag of Words by weighing words based on their importance. It calculates:

- Term Frequency (TF): The frequency of a word in a specific document.
- Inverse Document Frequency (IDF): A measure of how unique a word is across the entire dataset, reducing the weight of common words like "the". The final

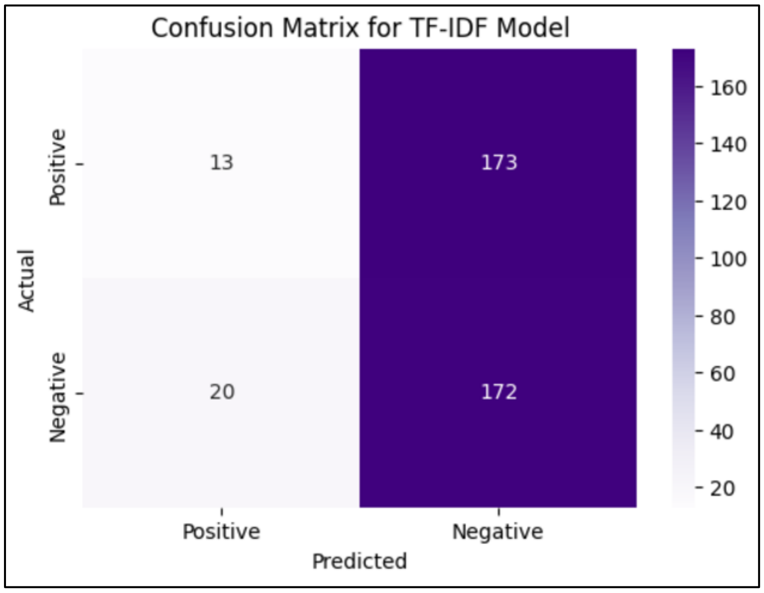
score is the product of TF and IDF. TF-IDF emphasizes terms that are frequent in a document but rare across the dataset, making it more effective for distinguishing relevant words in text classification tasks.

The implementation of this approach is included below. As we can see, it is very similar to the previous approach. One noteworthy difference is that scikit-learn's TfidfVectorizer was used in place of CountVectorizer as was done previously. A logistic regression model was trained after converting the input text into the TF-IDF feature space, followed by evaluation of the model performance.

```
from sklearn.feature_extraction.text import TfidfVectorizer

# convert data from text to TF-IDF feature space
tfidf_vectorizer = TfidfVectorizer()
X_train_tfidf = tfidf_vectorizer.fit_transform(X_train)
X_test_tfidf = tfidf_vectorizer.transform(X_test)

# building and training logistic regression model
model_tfidf = LogisticRegression(max_iter=200)
model_tfidf.fit(X_train_tfidf, y_train)

# predictions
y_pred_tfidf = model_tfidf.predict(X_test_tfidf)

# evaluation
print("TF-IDF Accuracy:", accuracy_score(y_test, y_pred_tfidf))
print("Precision:", precision_score(y_test, y_pred_tfidf))
print("Recall:", recall_score(y_test, y_pred_tfidf))
print("F1 Score:", f1_score(y_test, y_pred_tfidf))

conf_matrix_tfidf = confusion_matrix(y_test, y_pred_tfidf)
print("Confusion Matrix:\n", conf_matrix_tfidf)

# visualization
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix_tfidf, annot=True, fmt="d", cmap="Purples",
xticklabels=["Positive", "Negative"], yticklabels=["Positive",
"Negative"])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix for TF-IDF Model")
plt.show()
```

We see from the results below that the performance is significantly better, but still severely lacking (more detailed discussion later).

```
TF-IDF Accuracy: 0.4894179894179894
Precision: 0.4985507246376812
Recall: 0.8958333333333334
F1 Score: 0.6405959031657356
Confusion Matrix:
 [[ 13 173]
 [ 20 172]]
```



Step 2 – Word Embeddings (GloVe)

As we have seen, the model performances for Step 1 were very poor. This brings us to Step 2 where we use word embeddings to improve our model. More specifically, we use GloVe (Global Vectors for Word Representation), which is a pre-trained word embedding model that represents words as dense vectors in a continuous vector space. Unlike sparse representations like Bag of Words or TF-IDF, GloVe captures semantic meaning by encoding relationships between words based on their co-occurrence in a large corpus of text.

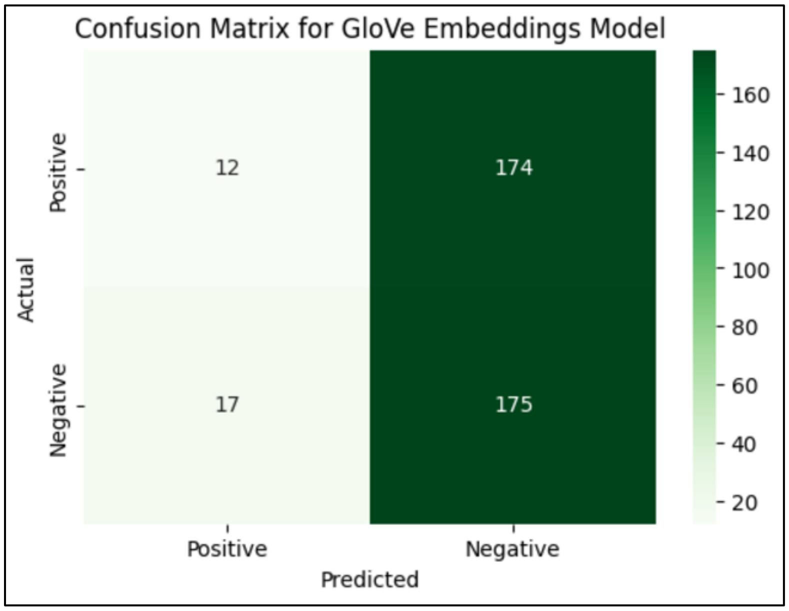
The implementation of Step 2 is shown below. We use the Gensim library to download the pre-trained word embedding model (GloVe). Thereafter, we use it to convert the text to the GloVe embeddings feature space (correcting for number of words by averaging the values). Lastly, we train the logistic regression model and evaluate it.

```
import gensim.downloader as api
import numpy as np
```

```
# load GloVe embeddings
word_vectors = api.load("glove-wiki-gigaword-100")

# convert data from text to glove embeddings feature space
def get_avg_glove(text, model, vector_size):
    words = text.split()
    feature_vec = np.zeros((vector_size,), dtype="float32")
    n_words = 0
    for word in words:
        if word in model:
            n_words += 1
            feature_vec = np.add(feature_vec, model[word])
    if n_words > 0:
        feature_vec = np.divide(feature_vec, n_words)
    return feature_vec

X_train_glove = np.array([get_avg_glove(text, word_vectors, 100) for text
in X_train])
X_test_glove = np.array([get_avg_glove(text, word_vectors, 100) for text
in X_test])

# building and training logistic regression model
model_glove = LogisticRegression(max_iter=200)
model_glove.fit(X_train_glove, y_train)

# predictions
y_pred_glove = model_glove.predict(X_test_glove)

# evaluation
print("GloVe Embeddings Model Accuracy:", accuracy_score(y_test,
y_pred_glove))
print("Precision:", precision_score(y_test, y_pred_glove))
print("Recall:", recall_score(y_test, y_pred_glove))
print("F1 Score:", f1_score(y_test, y_pred_glove))

conf_matrix_glove = confusion_matrix(y_test, y_pred_glove)
print("Confusion Matrix:\n", conf_matrix_glove)

# visualization
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix_glove, annot=True, fmt="d", cmap="Greens",
xticklabels=["Positive", "Negative"], yticklabels=["Positive",
"Negative"])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix for GloVe Embeddings Model")
plt.show()
```

The results are shown below (additional discussion later). We see it performs very similar to the TF-IDF model, which is relatively poor.

```
GloVe Embeddings Model Accuracy: 0.4947089947089947
Precision: 0.501432664756447
Recall: 0.9114583333333334
F1 Score: 0.6469500924214417
Confusion Matrix:
 [[ 12 174]
 [ 17 175]]
```



Step 3 – LSTM Model (using word embeddings)

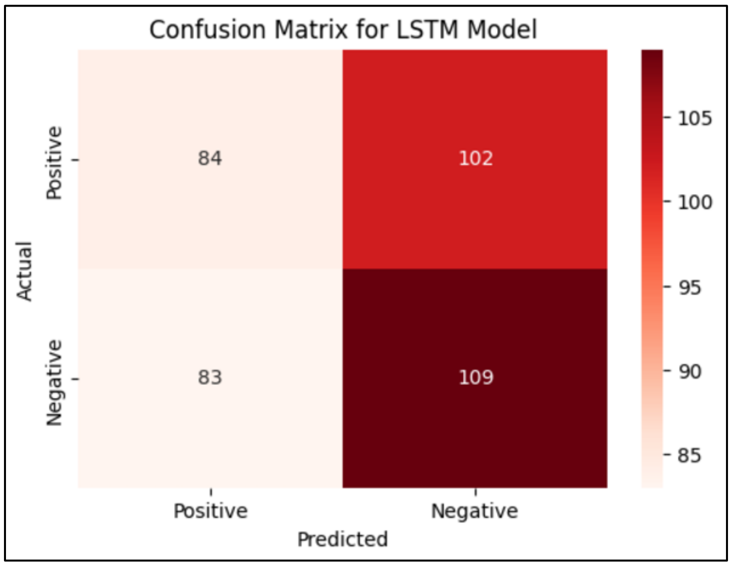
We unfortunately see that the results from both Step 1 and Step 2, are unsatisfactory. Therefore, we progress to another approach which would be to use a neural network as opposed to the simple logistic regression models that we have been using thus far. The model we selected is the LSTM (Long Short-Term Memory) model which is ideal for sequential data with long term dependencies as is the case in text data.

The implementation of this approach is shown below. There are many things to note. Firstly, with regards to the LSTM model itself, we see that an embeddings layer is included. This transforms words into dense numerical vectors, capturing semantic information. The embeddings layer learns word embeddings during training and is limited to a vocabulary size of 5,000 words, each represented as a 100-dimensional vector. There is also a dropout layer to help prevent overfitting. Also, before the LSTM model itself, we see in the code tokenization and padding of the sequences. These two steps properly prepare the data for the model. Tokenization converts words into integers while padding ensures a fixed length of 200 as the input size for the model. This architecture allows for semantic representation of the text, along with sequential and contextual understanding of the data. This means that the LSTM model can take into account the order of words, which was not in the case in Bag of Words and TF-IDF.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense,
SpatialDropout1D
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

# prepare sequences (tokenize and pad)
tokenizer = Tokenizer(num_words=5000)
tokenizer.fit_on_texts(X_train)
X_train_seq = tokenizer.texts_to_sequences(X_train)
X_test_seq = tokenizer.texts_to_sequences(X_test)
X_train_pad = pad_sequences(X_train_seq, maxlen=200)
X_test_pad = pad_sequences(X_test_seq, maxlen=200)

# build LSTM model (embeddings included in the embeddings layer)
model_lstm = Sequential([
    Embedding(5000, 100, input_length=200), # word embeddings included in
this layer
    SpatialDropout1D(0.2),
    LSTM(100, dropout=0.2, recurrent_dropout=0.2),
    Dense(1, activation='sigmoid')
])

model_lstm.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
model_lstm.fit(X_train_pad, y_train, epochs=5, batch_size=64,
validation_data=(X_test_pad, y_test))

# predictions
y_pred_lstm = model_lstm.predict(X_test_pad)
y_pred_lstm = (y_pred_lstm > 0.5).astype("int32")

# evaluation
print("LSTM Accuracy:", accuracy_score(y_test, y_pred_lstm))
print("Precision:", precision_score(y_test, y_pred_lstm))
print("Recall:", recall_score(y_test, y_pred_lstm))
print("F1 Score:", f1_score(y_test, y_pred_lstm))

conf_matrix_lstm = confusion_matrix(y_test, y_pred_lstm)
print("Confusion Matrix:\n", conf_matrix_lstm)

# visualization
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix_lstm, annot=True, fmt='d', cmap='Reds',
xticklabels=["Positive", "Negative"], yticklabels=["Positive",
"Negative"])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix for LSTM Model")
plt.show()
```

The results are shown below, and sadly they are also unsatisfactory.

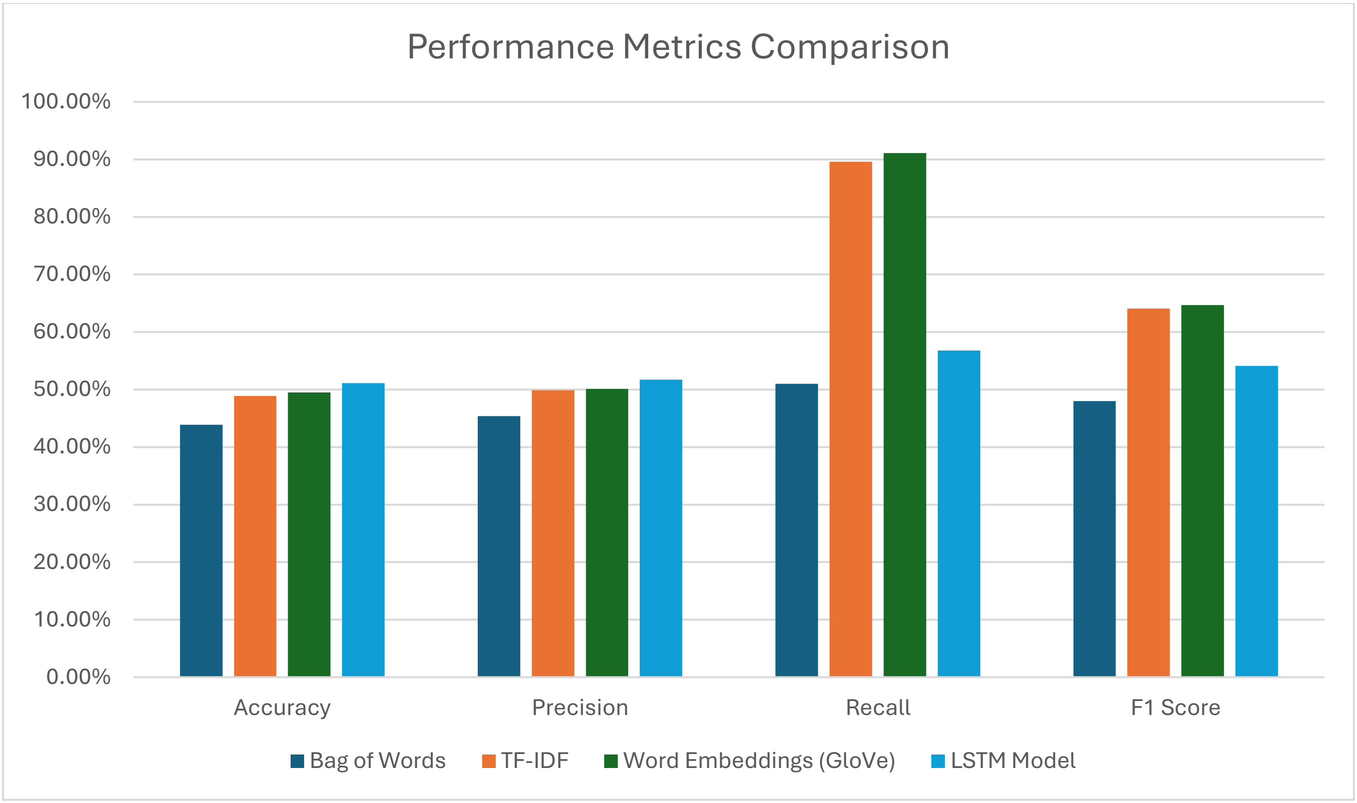
```
LSTM Accuracy: 0.5105820105820106
Precision: 0.5165876777251185
Recall: 0.5677083333333334
F1 Score: 0.5409429280397022
Confusion Matrix:
 [[ 84 102]
 [ 83 109]]
```

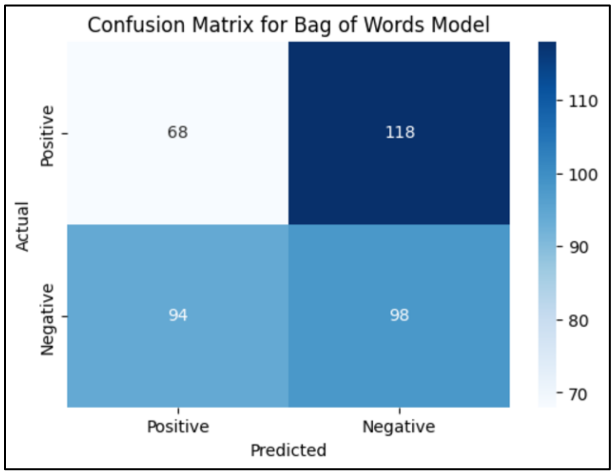


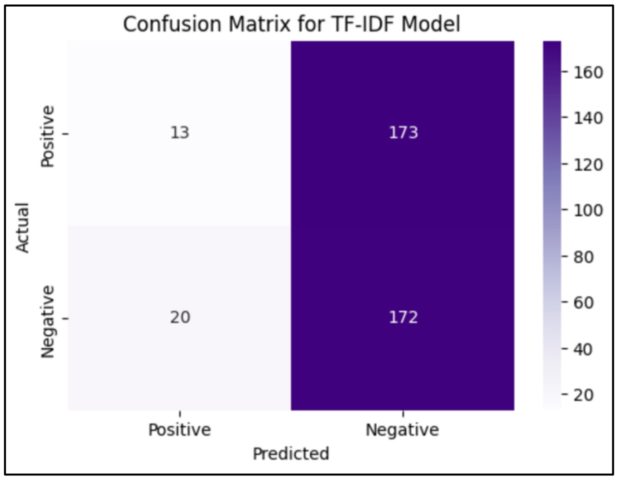
4. Results

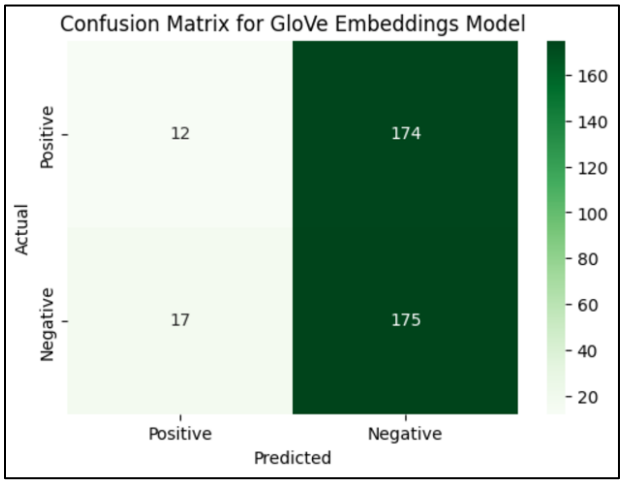
Thus far, we have implemented Steps 1 to 3 and reviewed the results superficially. Now we look at the results in more detail. The table below summarizes the performance metric results for each model.

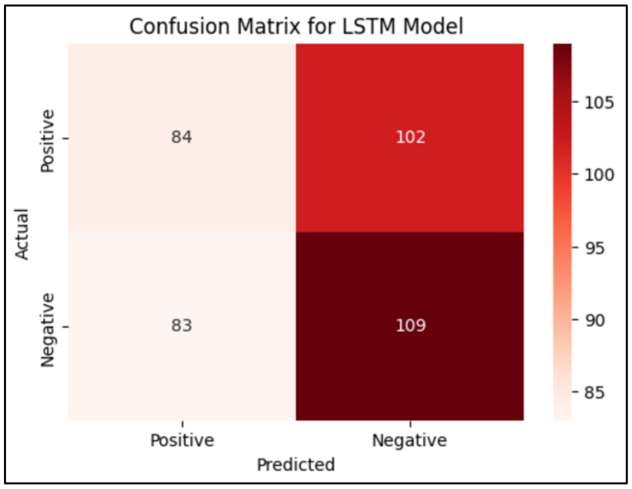
	Bag of Words	TF-IDF	Word Embeddings (GloVe)	LSTM Model
Accuracy	43.9%	48.9%	49.5%	51.1%
Precision	45.4%	49.9%	50.1%	51.7%
Recall	51.0%	89.6%	91.1%	56.8%
F1 Score	48.0%	64.1%	64.7%	54.1%











Observing the performance metric results, we see that the best model based on the F1 Score would be the word embeddings model (GloVe), followed closely by the TF-IDF model. However, even these models don't perform as well as we would want. Additionally, it's noteworthy to point out that both of these models have very good recall scores, but the precision scores are much less leading to a lower F1 score. This means that when the label is 1 (DJIA is increasing), then the model will do a good job of catching that. However, this comes at the cost of predicting an output of 1 too much, often times even when the true label is 0 (low precision). We can see these same observations from the confusion matrix visualizations. Additionally, the Bag of Words model along with the LSTM model do have higher values for recall than precision, but they are much more balanced and closer to each other.

The biggest take away though from analyzing the results is that all the models perform subpar to our desired performance.

5. Challenges and Solutions

As mentioned above, the poor performance of the models was a significant roadblock and a central challenge in this study. As a solution to this challenge, we move on to Step 4.

Step 4 – Open-source LLMs

Until yet, we haven't been able to build a model that performs at the desired level of performance. However, we also haven't truly investigated the underlying cause of such low performance. In this section, we will do exactly that: we use open-source LLMs to help us analyze the issue at hand.

We built state-of-the-art models in Steps 1 to 3, yet they didn't perform well, and this begs the question: where is the problem? To investigate this question, I manually inspected the original dataset file ("Combined_News_DJIA.csv"). Upon closer inspection, it became clear that many of the new headlines didn't relate to the stock market. Since these are general new headlines, much of it was related to other topics such as politics, healthcare, education, etc. So I had a preliminary hypothesis that the lack of related data in the dataset was the cause of such low performance. However, the dataset is far too large to manually inspect thoroughly, and what I did in fact manually inspect was only a small fraction of the total data, so it is a weak hypothesis. To aid in testing my theory, I utilized an open-source LLM: "cross-encoder/nli-distilroberta-base". This was accessed through Hugging Face's transformer library. It is a compact, efficient version of the RoBERTa model fine-tuned on a Natural Language Inference (NLI) task. It excels at comparing pairs of sentences and determining their relationship (e.g., entailment, contradiction, or neutrality). This model was used for zero-shot classification in which we essentially ask the model to give us a ranking (from 0 to 1), of how related a headline is to any given topic. In our case, the topic at hand is the stock market, so we are basically asking the model, how related is this text to the topic of the stock market? The model uses semantic similarity and matching to return a confidence score, ranging from 0 to 1, with the higher values meaning a stronger correlation between the headline and the topic.

This was truly one of the most exciting parts of this project, to apply modern NLP techniques to a real world problem.

The implementation is shown below. First we had to prepare the data by manipulating the dataframe and forcing the headlines to be one long column as opposed to a rectangle like shape. This is done here:

```
import pandas as pd

# reshape the dataframe
df_melted = pd.melt(
    data,
    id_vars=["Date", "Label"],  # freeze these columns
    value_vars=[col for col in data.columns if col.startswith("Top")],  # 
unfreeze these columns
    var_name="headline",  # new column name for the old heading
    value_name="actual_headline"  # new column name for the actual content
)

# re-sort the dataframe
df_melted = df_melted.sort_values(by=["Date", 
"headline"]).reset_index(drop=True)

print(df_melted)
```

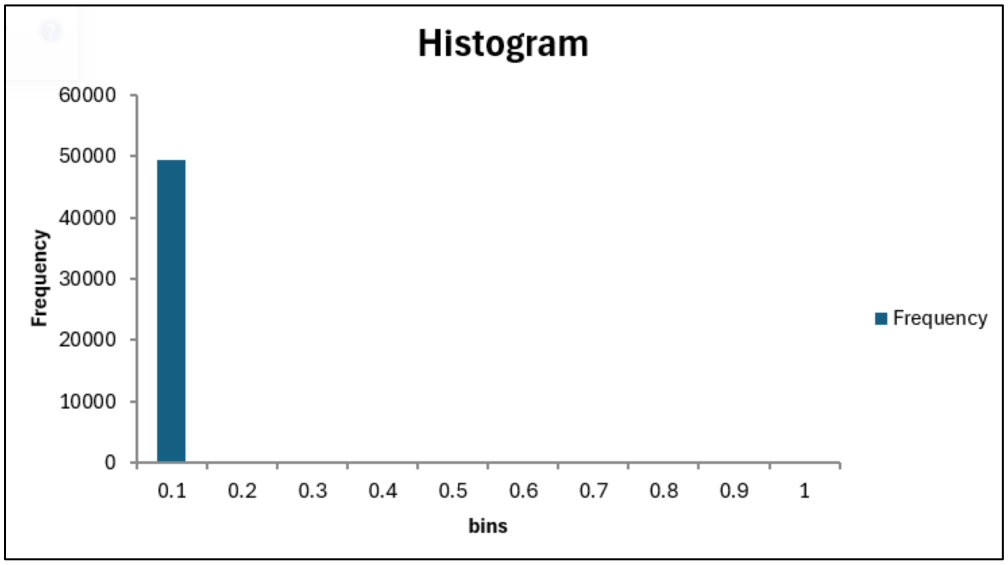
Thereafter, we actually build the classifier itself. We start by defining the classifier and the topic of interest (stock market). We create a new csv file to save our outputs. We continue to process the data in batches, assigning confidence scores depending on how strong the correlation is between the headline and the topic. After all the data has been processed, it is finally saved in a file named: “confidence_scores_with_batches.csv”. This file is attached to this report for reference.

```
import pandas as pd
from transformers import pipeline
import time

classifier = pipeline("zero-shot-classification", model="cross-
encoder/nli-distilroberta-base", device=0)
topic = "stock market"

output_file = "confidence_scores_with_batches.csv"

def compute_confidence_scores_batch(headlines):
    scores = []
    for headline in headlines:
        if pd.isna(headline) or headline.strip() == "":
            scores.append(0.0)
        else:
            # process valid headline
            result = classifier(headline, candidate_labels=[topic], 
multi_label=False)
```

```
            scores.append(result['scores'][0])  # extract "stock market"
confidence
    return scores

# use partially saved file if it exists
try:
    df_melted = pd.read_csv(output_file)
    print(f"Resuming from saved file: {output_file}")
except FileNotFoundError:
    print(f"No saved progress found. Starting fresh.")

# initialize 'confidence_score' column if it doesn't exist
if 'confidence_score' not in df_melted.columns:
    df_melted['confidence_score'] = None

batch_size = 1000
start_time = time.time()

for start in range(0, len(df_melted), batch_size):
    end = start + batch_size
    batch = df_melted[start:end]

    # skip rows that are processed
    if batch['confidence_score'].notnull().all():
        continue

    print(f"Processing batch {start}-{end}...")

    # compute confidence scores
    headlines = batch['actual_headline'].tolist()
    scores = compute_confidence_scores_batch(headlines)

    df_melted.loc[start:end - 1, 'confidence_score'] = scores

    # save progress periodically
    df_melted.to_csv(output_file, index=False)
    print(f"Batch {start}-{end} saved to {output_file}")

# final save
df_melted.to_csv(output_file, index=False)
print(f"All batches processed and saved. Total time: {time.time() -
start_time:.2f} seconds")
```

Once the final processed output file was saved, I did further analysis to see if my initial hypothesis held true or not. I saved this CSV file as an excel file and did additional post-processing (file name: "confidence scores – analysis.xlsx" also attached for reference). I ordered the data based on confidence scores and generated a frequency histogram shown below.

Lower Limit	Upper Limit	Frequency
0.0	0.1	49542
0.1	0.2	70
0.2	0.3	19
0.3	0.4	13
0.4	0.5	5
0.5	0.6	6
0.6	0.7	11
0.7	0.8	9
0.8	0.9	20
0.9	1.0	30



We can see from both the table as well as the graph of the histogram that the vast majority of news headlines have a confidence score of 0 to 0.1 (10% or less). This is a very weak confidence score indicating that most likely this headline does not relate to the stock market. This would mean that more than 99.5% of the headlines are not strongly related to the stock market. If this is truly the case, then it explains the results we were getting before regarding the other models (Steps 1 to 3). If the vast majority of the headlines don't related to the stock market, than the models don't have real data to learn patterns from, and would therefore have poor performance. Based on these findings, it would seems to be the case that the initial hypothesis is in fact true.

6. Conclusion

This was an extremely eye-opening study, where we used modern NLP techniques to solve a real world problem. We attempted to predict stock market trends based on news headlines. We concluded that the dataset was not correlated enough with the topic of interest for the models to learn any meaningful patterns. However, there were some modest gains in performance. More specifically, the TF-IDF model along with the Word Embeddings model (GloVe), did show some improvements with F1 Scores near 64%. However, the biggest conclusion from this study is that better quality data is needed for the models to have better performance. This opens up a number of possible future works. One possibility it to generate a new dataset with headlines only related to finances. Another possibility is to generate a much larger dataset with many more headlines, and in this way, more information regarding the stock market can be captured. Other possible future works include exploring real-time prediction pipelines using streaming news data. Also, various other models may be evaluated to see if there are additional performance gains.