GR5074_Project_3_Group1

April 30, 2025

1 QMSS5074GR - Final Project (3rd)

1.1 Metadata

1.1.1 Group ID: Group 1

1.1.2 Team Members:

- Yutong Quan (UNI: yq2397)
- Han He (UNI: hh3098)
- Hanna Tang (UNI: ht2668)

1.1.3 GitHub Repository: https://github.com/hannatang-beep/GR5074-Adv-Machine-Learning/tree/main/Project_3

1.2 About

This project performs sentiment analysis on the Stanford Sentiment Treebank (SST-2) dataset. The workflow begins with data ingestion, preprocessing (including tokenization and vectorization), and exploratory data analysis to understand class distribution and text characteristics.

We implemented and evaluated multiple models across three categories:

- **Traditional ML Models**: Logistic Regression, SVM, Random Forest, XGBoost
- **Neural Network Models**: Simple MLP, CNN, TextCNN, BiLSTM, GloVe-based MLP
- **Transformer-based Model**: Fine-tuned BERT using Hugging Face Transformers

Each model was evaluated on a held-out test set using common metrics such as **Accuracy**, **Precision**, **Recall**, **F1 Score**, and **ROC-AUC**. We also performed hyperparameter optimization using Keras Tuner and analyzed the results through statistical significance testing and error review.

Final Result: BERT achieved the best overall performance, with the highest accuracy and F1 score, making it the most suitable model for deployment in this task.

1.3 Description

1.4 Part 1 – Data Ingestion & Preprocessing

1. **Data Loading**
   - Acquire the Stanford Sentiment Treebank dataset.

- Split into training, validation and test sets with stratified sampling to preserve class balance.

- Clearly document your splitting strategy and resulting dataset sizes.

2. **Text Cleaning & Tokenization**
    - Implement a reusable preprocessing pipeline that handles at least:
        - HTML removal, lowercasing, punctuation stripping
        - Vocabulary pruning (e.g., rare words threshold)
        - Tokenization (character- or word-level)
    - Expose this as a function/class so it can be saved and re-loaded for inference.
3. **Feature Extraction**
    - **Traditional**: Build a TF-IDF vectorizer (or n-gram count) pipeline.
    - **Neural**: Prepare sequences for embedding—pad/truncate to a fixed length.
    - Save each preprocessor (vectorizer/tokenizer) to disk.

1.5 Part 2 – Exploratory Data Analysis (EDA)

1. **Class Distribution**
    - Visualize the number of positive vs. negative reviews.
    - Compute descriptive statistics on review lengths (mean, median, IQR).
2. **Text Characteristics**
    - Plot the 20 most frequent tokens per sentiment class.
    - Generate word clouds (or bar charts) highlighting key terms for each class.
3. **Correlation Analysis**
    - Analyze whether review length correlates with sentiment.
    - Present findings numerically and with at least one visualization.

1.6 Part 3 – Baseline Traditional Models

1. **Logistic Regression & SVM**
    - Train at least two linear models on your TF-IDF features (e.g., logistic regression, linear SVM).
    - Use cross-validation (5 folds) on the training set to tune at least one hyperparameter.
2. **Random Forest & Gradient Boosting**
    - Train two tree-based models (e.g., Random Forest, XGBoost) on the same features.

- Report feature-importance for each and discuss any notable tokens.

3. **Evaluation Metrics**
   - Compute accuracy, precision, recall, F1-score, and ROC-AUC on the **held-out test set**.
   - Present all results in a single comparison table.

1.7 Part 4 – Neural Network Models

1. **Simple Feed-Forward**
   - Build an embedding layer + a dense MLP classifier.
   - Ensure you freeze vs. unfreeze embeddings in separate runs.
2. **Convolutional Text Classifier**
   - Implement a 1D-CNN architecture (Conv + Pooling) for sequence data.
   - Justify your choice of kernel sizes and number of filters.
3. **Recurrent Model (Optional)**
   - (Stretch) Add an RNN or Bi-LSTM layer and compare performance/time vs. CNN.

1.8 Part 5 – Transfer Learning & Advanced Architectures

1. **Pre-trained Embeddings**
   - Retrain one network using pre-trained GloVe (or FastText) embeddings.
   - Compare results against your from-scratch embedding runs.
2. **Transformer Fine-Tuning**
   - Fine-tune a BERT-family model on the training data.
   - Clearly outline your training hyperparameters (learning rate, batch size, epochs).

1.9 Part 6 – Hyperparameter Optimization

1. **Search Strategy**
   - Use a library (e.g., Keras Tuner, Optuna) to optimize at least two hyperparameters of one deep model.
   - Describe your search space and stopping criteria.
2. **Results Analysis**
   - Report the best hyperparameter configuration found.
   - Plot validation-loss (or metric) vs. trials to illustrate tuning behavior.

1.10 Part 7 – Final Comparison & Error Analysis

1. **Consolidated Results**
   - Tabulate test-set performance for **all** models (traditional, neural, transfer-learned).

   - Highlight top-performing model overall and top in each category.
2. **Statistical Significance**
   - Perform a significance test (e.g., McNemar's test) between your best two models.
3. **Error Analysis**
   - Identify at least 20 examples your best model misclassified.

   - For a sample of 5, provide the raw text, predicted vs. true label, and a short discussion of each error—what linguistic artifact might have confused the model?

1.11 Part 8 – Optional Challenge Extensions

- Implement data augmentation for text (back-translation, synonym swapping) and measure its impact.

- Integrate a sentiment lexicon feature (e.g., VADER scores) into your models and assess whether it improves predictions.

- Deploy your best model as a simple REST API using Flask or FastAPI and demo it on a handful of user-submitted reviews.

1.12 Part 1 – Data Ingestion & Preprocessing

1. **Data Loading**
   - Acquire the Stanford Sentiment Treebank dataset.
   - Split into training, validation, and test sets with stratified sampling to preserve class balance.
   - Clearly document your splitting strategy and resulting dataset sizes.

[ ]:
```
!git clone https://github.com/YJiangcm/SST-2-sentiment-analysis.git
```

```
Cloning into 'SST-2-sentiment-analysis'…
remote: Enumerating objects: 85, done.
remote: Counting objects: 100% (85/85), done.
remote: Compressing objects: 100% (72/72), done.
remote: Total 85 (delta 44), reused 29 (delta 11), pack-reused 0 (from 0)
Receiving objects: 100% (85/85), 478.79 KiB | 1.75 MiB/s, done.
Resolving deltas: 100% (44/44), done.
```

[ ]:
```
import pandas as pd
import os
```

```
os.chdir('SST-2-sentiment-analysis/data')

train_df = pd.read_csv("train.tsv", sep='\t', names=['label', 'sentence'])
val_df = pd.read_csv("dev.tsv", sep='\t', names=['label', 'sentence'])
test_df = pd.read_csv("test.tsv", sep='\t', names=['label', 'sentence'])

print("Train shape:", train_df.shape)
print("Validation shape:", val_df.shape)
print("Test shape:", test_df.shape)

train_df.head()
```

```
Train shape: (6920, 2)
Validation shape: (872, 2)
Test shape: (1821, 2)
```

[ ]:

	label	sentence
0	1	a stirring , funny and finally transporting re…
1	0	apparently reassembled from the cutting-room f…
2	0	they presume their audience wo n't sit still f…
3	1	this is a visually stunning rumination on love…
4	1	jonathan parker 's bartleby should have been t…

We used the official SST-2 dataset split provided in the repository, which includes 6,920 training samples, 872 validation samples, and 1,821 test samples. Since the dataset is already stratified, we retain this predefined split and verify class balance during EDA.

2. **Text Cleaning & Tokenization**
    - Implement a reusable preprocessing pipeline that handles at least:
        - HTML removal, lowercasing, punctuation stripping
        - Vocabulary pruning (e.g., rare words threshold)
        - Tokenization (character- or word-level)
    - Expose this as a function/class so it can be saved and re-loaded for inference.

[ ]:
```
import re
import pickle
from collections import Counter

class TextCleanerTokenizer:
    def __init__(self, mode='word', min_freq=2):
        self.mode = mode
        self.min_freq = min_freq
        self.vocab = None

    def clean(self, text):
```

```
        text = re.sub(r'<[^>]*>', '', text)
        text = re.sub(r'\W+', ' ', text.lower())
        return text.strip()

    def tokenize(self, text):
        if self.mode == 'word':
            return text.split()
        elif self.mode == 'char':
            return list(text.replace(' ', ''))
        else:
            raise ValueError("mode must be 'word' or 'char'")

    def clean_and_tokenize(self, texts):
        cleaned = [self.clean(t) for t in texts]
        tokenized = [self.tokenize(c) for c in cleaned]

        all_tokens = [token for sentence in tokenized for token in sentence]
        token_counter = Counter(all_tokens)
        self.vocab = set([token for token, freq in token_counter.items() if␣
↪freq >= self.min_freq])

        pruned_tokenized = [[token for token in sent if token in self.vocab]␣
↪for sent in tokenized]
        return pruned_tokenized

    def save_tokens(self, tokens, path='tokens.pkl'):
        with open(path, 'wb') as f:
            pickle.dump((tokens, self.vocab), f)

    def load_tokens(self, path='tokens.pkl'):
        with open(path, 'rb') as f:
            tokens, self.vocab = pickle.load(f)
        return tokens
```

[ ]:
```
clean_tokenizer = TextCleanerTokenizer(mode='word', min_freq=2)

train_tokens = clean_tokenizer.clean_and_tokenize(train_df['sentence'])
val_tokens = clean_tokenizer.clean_and_tokenize(val_df['sentence'])
test_tokens = clean_tokenizer.clean_and_tokenize(test_df['sentence'])

train_df['cleaned'] = [' '.join(tokens) for tokens in train_tokens]
val_df['cleaned'] = [' '.join(tokens) for tokens in val_tokens]
test_df['cleaned'] = [' '.join(tokens) for tokens in test_tokens]
```

We implemented a reusable text preprocessing class that handles HTML removal, lowercasing, and punctuation stripping. We tokenize at the word level and prune the vocabulary by removing tokens that appear fewer than 2 times. The preprocessor can be saved for future inference.

3. **Feature Extraction**
    - **Traditional**: Build a TF-IDF vectorizer (or n-gram count) pipeline.
    - **Neural**: Prepare sequences for embedding—pad/truncate to a fixed length.
    - Save each preprocessor (vectorizer/tokenizer) to disk.

[ ]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
import pickle

vectorizer = TfidfVectorizer(max_features=5000, ngram_range=(1, 2))

X_train_tfidf = vectorizer.fit_transform(train_df['cleaned'])

X_val_tfidf = vectorizer.transform(val_df['cleaned'])
X_test_tfidf = vectorizer.transform(test_df['cleaned'])

with open('tfidf_vectorizer.pkl', 'wb') as f:
    pickle.dump(vectorizer, f)

print("Train TF-IDF shape:", X_train_tfidf.shape)
```

```
Train TF-IDF shape: (6920, 5000)
```

[ ]:
```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import pickle

train_texts = train_df['cleaned'].tolist()
val_texts = val_df['cleaned'].tolist()
test_texts = test_df['cleaned'].tolist()

tokenizer = Tokenizer(num_words=10000, oov_token="<OOV>")
tokenizer.fit_on_texts(train_texts)

train_seq = tokenizer.texts_to_sequences(train_texts)
val_seq = tokenizer.texts_to_sequences(val_texts)
test_seq = tokenizer.texts_to_sequences(test_texts)

MAX_LEN = 100
X_train_pad = pad_sequences(train_seq, maxlen=MAX_LEN, padding='post',␣
↪truncating='post')
X_val_pad = pad_sequences(val_seq, maxlen=MAX_LEN, padding='post',␣
↪truncating='post')
X_test_pad = pad_sequences(test_seq, maxlen=MAX_LEN, padding='post',␣
↪truncating='post')
```

```python
with open('tokenizer.pkl', 'wb') as f:
    pickle.dump((tokenizer, MAX_LEN), f)

print("Train embedding input shape:", X_train_pad.shape)
```

```
Train embedding input shape: (6920, 100)
```

For traditional models, we used a TF-IDF vectorizer with a vocabulary size of 5,000 and bi-gram features.

For neural models, we tokenized the text using a Keras tokenizer limited to the top 10,000 words and padded the sequences to a uniform length of 100.

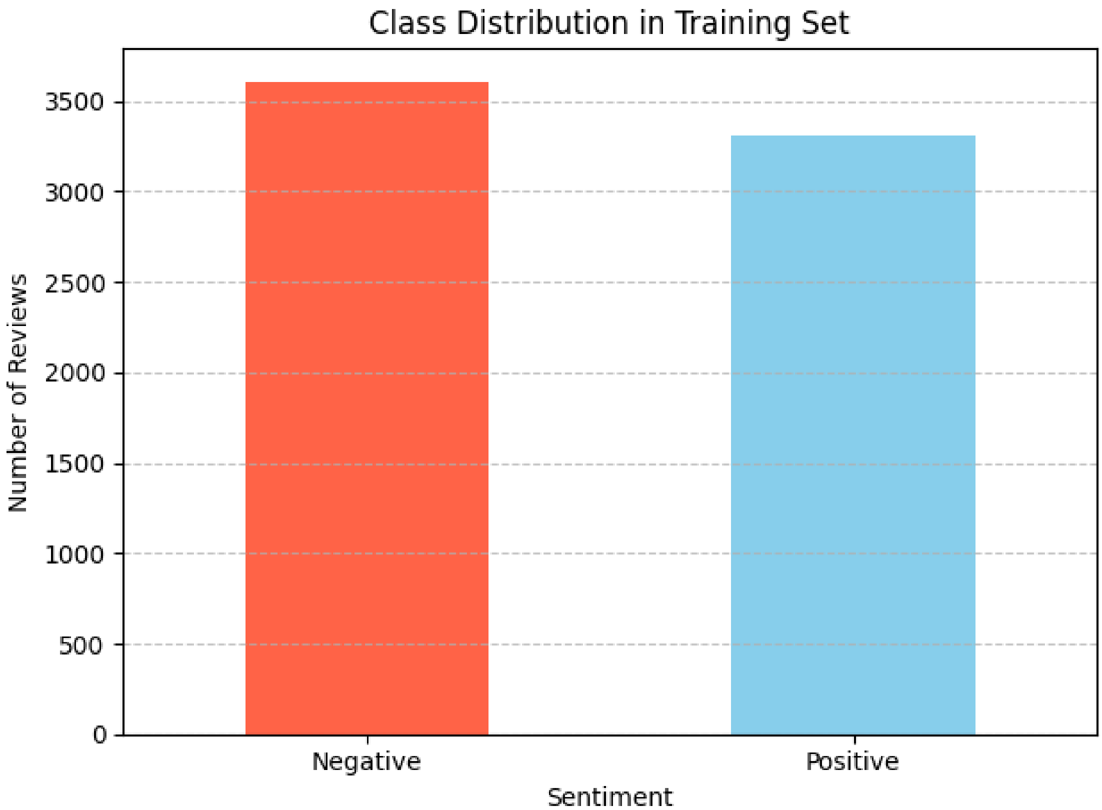
Each preprocessor was saved to disk for reproducibility.

1.13 Part 2 – Exploratory Data Analysis (EDA)

1. **Class Distribution**
    - Visualize the number of positive vs. negative reviews.

    - Compute descriptive statistics on review lengths (mean, median, IQR).

[ ]:
```python
import matplotlib.pyplot as plt

train_df['label'].value_counts().plot(kind='bar', color=['tomato', 'skyblue'])
plt.title("Class Distribution in Training Set")
plt.xlabel("Sentiment")
plt.ylabel("Number of Reviews")
plt.xticks(ticks=[0, 1], labels=["Negative", "Positive"], rotation=0)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```



[ ]:
```python
import numpy as np

train_df['length'] = train_df['cleaned'].apply(lambda x: len(x.split()))

mean_len = train_df['length'].mean()
median_len = train_df['length'].median()
q75, q25 = np.percentile(train_df['length'], [75 ,25])
iqr = q75 - q25

print(f"Mean review length: {mean_len:.2f} words")
print(f"Median review length: {median_len} words")
print(f"IQR (75th - 25th percentile): {iqr} words")
```

```
Mean review length: 16.61 words
Median review length: 16.0 words
IQR (75th - 25th percentile): 12.0 words
```

We visualized the sentiment class distribution in the training set using a bar chart. The dataset is relatively balanced.

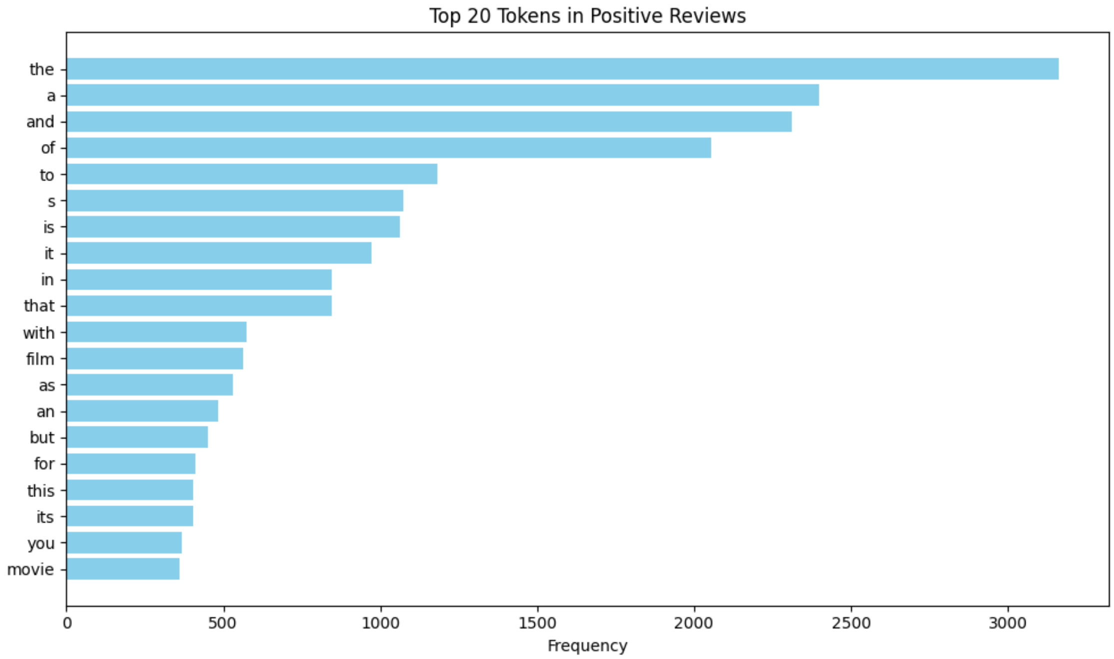
2. **Text Characteristics**
    - Plot the 20 most frequent tokens per sentiment class.

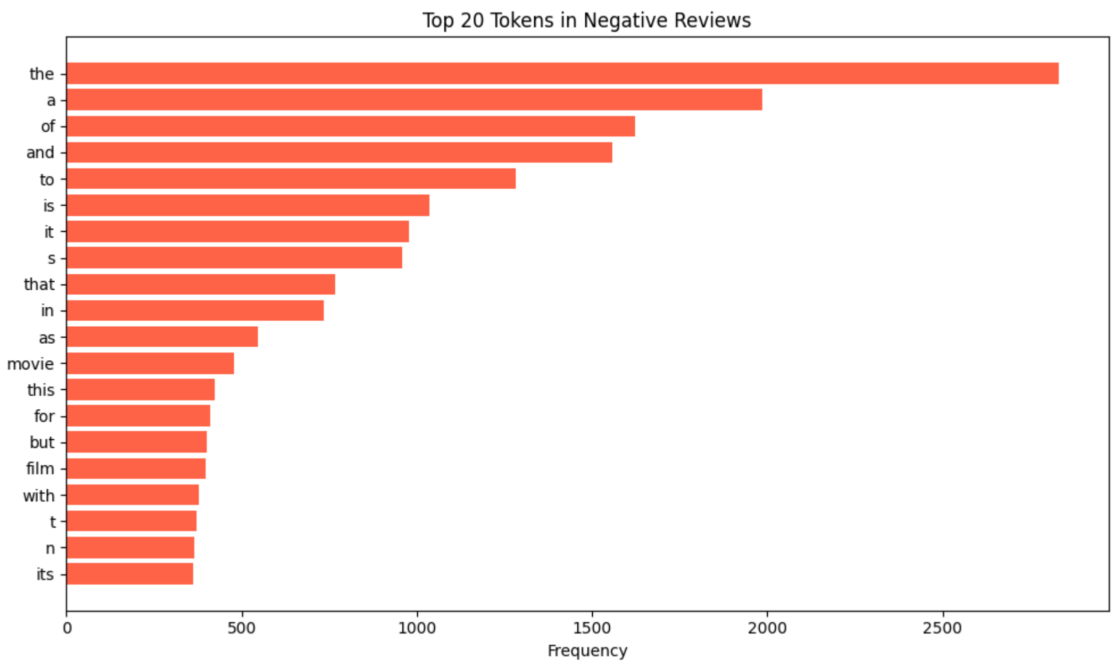
- Generate word clouds (or bar charts) highlighting key terms for each class.

[ ]:
```
positive_reviews = train_df[train_df['label'] == 1]['cleaned']
negative_reviews = train_df[train_df['label'] == 0]['cleaned']

positive_tokens = [word for sentence in positive_reviews for word in sentence.
 ↪split()]
negative_tokens = [word for sentence in negative_reviews for word in sentence.
 ↪split()]
```

[ ]:
```
from collections import Counter

pos_counter = Counter(positive_tokens)
neg_counter = Counter(negative_tokens)

top20_pos = pos_counter.most_common(20)
top20_neg = neg_counter.most_common(20)
```

[ ]:
```
import matplotlib.pyplot as plt

# Positive
words_pos, counts_pos = zip(*top20_pos)

plt.figure(figsize=(10,6))
plt.barh(words_pos[::-1], counts_pos[::-1], color='skyblue')
plt.title("Top 20 Tokens in Positive Reviews")
plt.xlabel("Frequency")
plt.tight_layout()
plt.show()

# Negative
words_neg, counts_neg = zip(*top20_neg)

plt.figure(figsize=(10,6))
plt.barh(words_neg[::-1], counts_neg[::-1], color='tomato')
plt.title("Top 20 Tokens in Negative Reviews")
plt.xlabel("Frequency")
plt.tight_layout()
plt.show()
```





[ ]:
```
!pip install wordcloud
```

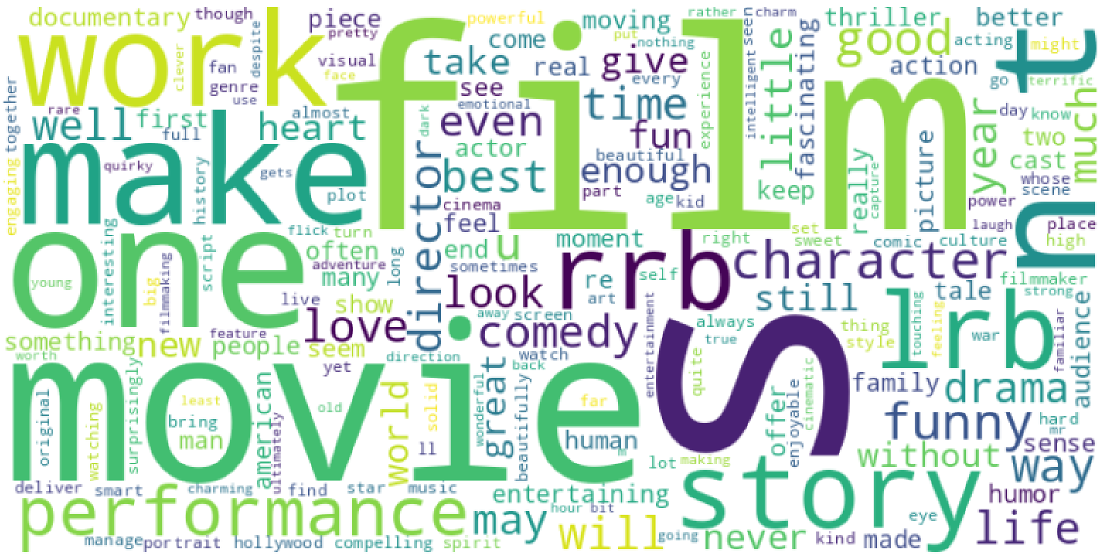
```
Requirement already satisfied: wordcloud in /usr/local/lib/python3.11/dist-
packages (1.9.4)
```

```
Requirement already satisfied: numpy>=1.6.1 in /usr/local/lib/python3.11/dist-
packages (from wordcloud) (2.0.2)
Requirement already satisfied: pillow in /usr/local/lib/python3.11/dist-packages
(from wordcloud) (11.2.1)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.11/dist-
packages (from wordcloud) (3.10.0)
Requirement already satisfied: contourpy>=1.0.1 in
/usr/local/lib/python3.11/dist-packages (from matplotlib->wordcloud) (1.3.2)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.11/dist-
packages (from matplotlib->wordcloud) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/usr/local/lib/python3.11/dist-packages (from matplotlib->wordcloud) (4.57.0)
Requirement already satisfied: kiwisolver>=1.3.1 in
/usr/local/lib/python3.11/dist-packages (from matplotlib->wordcloud) (1.4.8)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.11/dist-packages (from matplotlib->wordcloud) (24.2)
Requirement already satisfied: pyparsing>=2.3.1 in
/usr/local/lib/python3.11/dist-packages (from matplotlib->wordcloud) (3.2.3)
Requirement already satisfied: python-dateutil>=2.7 in
/usr/local/lib/python3.11/dist-packages (from matplotlib->wordcloud)
(2.9.0.post0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-
packages (from python-dateutil>=2.7->matplotlib->wordcloud) (1.17.0)
```

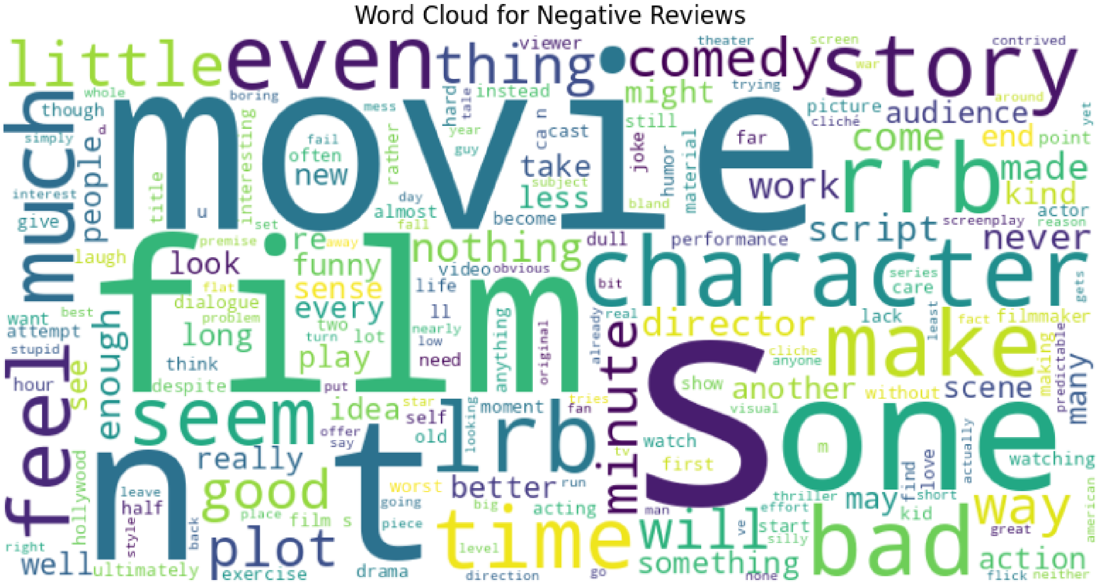
[ ]:

```python
from wordcloud import WordCloud

# Positive Word Cloud
pos_wordcloud = WordCloud(width=800, height=400, background_color='white').
  ↪generate(' '.join(positive_tokens))
plt.figure(figsize=(10, 6))
plt.imshow(pos_wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title('Word Cloud for Positive Reviews')
plt.show()

# Positive Word Cloud
neg_wordcloud = WordCloud(width=800, height=400, background_color='white').
  ↪generate(' '.join(negative_tokens))
plt.figure(figsize=(10, 6))
plt.imshow(neg_wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title('Word Cloud for Negative Reviews')
plt.show()
```

Word Cloud for Positive Reviews



Word Cloud for Negative Reviews



We analyzed token frequency distributions across sentiment classes. Frequent terms like “film”, “movie”, “story”, and “make” appeared in both classes, reflecting their domain relevance. Sentiment-specific terms such as “bad”, “nothing”, and “boring” in negative reviews, and “good”, “fun”, and “entertaining” in positive ones provided additional semantic cues. Word clouds further highlighted these patterns visually.

3. **Correlation Analysis**
   - Analyze whether review length correlates with sentiment.

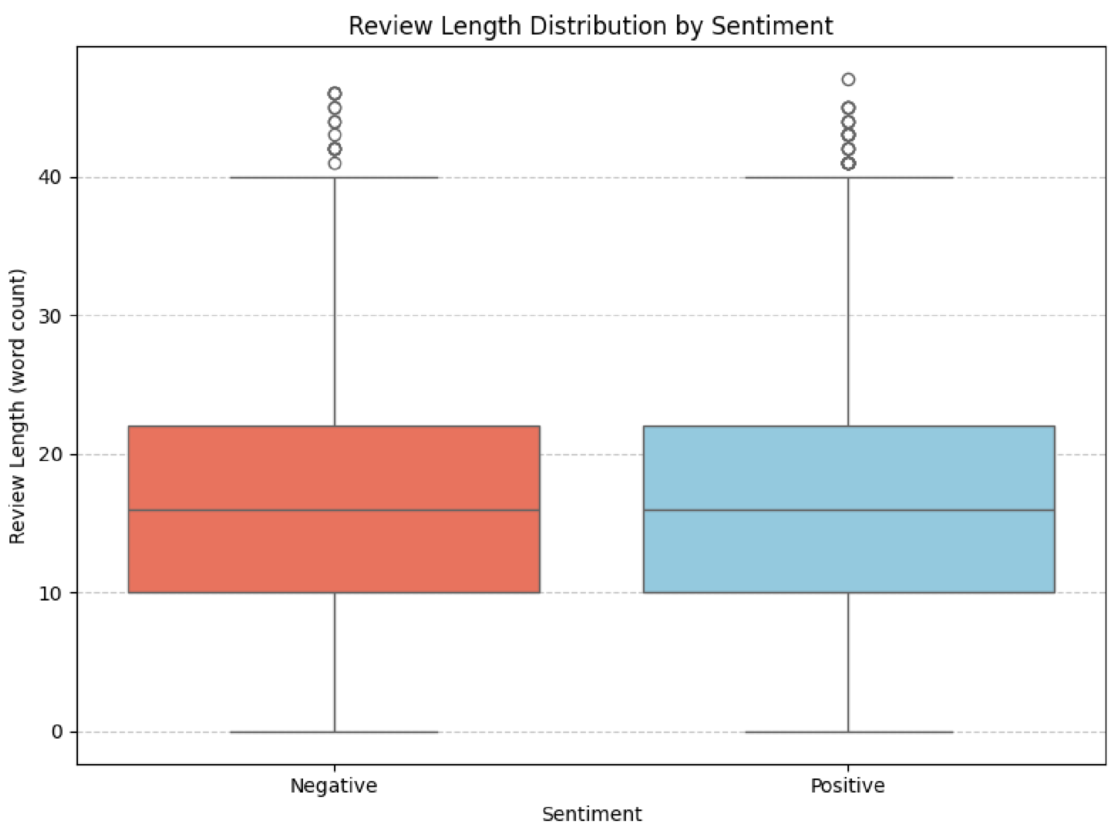
- Present findings numerically and with at least one visualization.

[ ]:
```
length_stats = train_df.groupby('label')['length'].agg(['mean', 'median',␣
↪'std', 'count'])
length_stats.index = ['Negative', 'Positive']
print(length_stats)
```

```
               mean  median       std  count
Negative  16.527492    16.0  8.288413   3310
Positive  16.689474    16.0  8.474415   3610
```

[ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 6))
sns.boxplot(x='label', y='length', data=train_df, palette=['tomato', 'skyblue'])
plt.xticks([0, 1], ['Negative', 'Positive'])
plt.title("Review Length Distribution by Sentiment")
plt.xlabel("Sentiment")
plt.ylabel("Review Length (word count)")
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

```
<ipython-input-15-a6e62426b4fa>:5: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in
v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same
effect.

  sns.boxplot(x='label', y='length', data=train_df, palette=['tomato',
'skyblue'])
```




```
[ ]: from scipy.stats import ttest_ind

neg_lengths = train_df[train_df['label'] == 0]['length']
pos_lengths = train_df[train_df['label'] == 1]['length']

t_stat, p_value = ttest_ind(pos_lengths, neg_lengths, equal_var=False)

print(f"T-statistic: {t_stat:.3f}")
print(f"P-value: {p_value:.5f}")
if p_value < 0.05:
    print("  Review length difference between classes is statistically␣
↪significant.")
else:
    print("  No significant difference in review length between classes.")
```

```
T-statistic: 0.803
P-value: 0.42176
  No significant difference in review length between classes.
```

We examined the correlation between review length and sentiment. Both sentiment classes had similar distributions (mean 16.6 words), and a two-sample t-test yielded a non-significant result (t = 0.803, p = 0.42). This suggests review length does not correlate strongly with sentiment in this dataset.

1.14 Part 3 – Baseline Traditional Models

1. **Logistic Regression & SVM**
   - Train at least two linear models on your TF-IDF features.

   - Use cross-validation (5 folds) on the training set to tune at least one hyperparameter.

[ ]:
```python
X_train_tfidf
y_train = train_df['label']
```

[ ]:
```python
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

# Logistic Regression
logreg = LogisticRegression(max_iter=1000)
logreg_cv = cross_val_score(logreg, X_train_tfidf, y_train, cv=5,␣
 ↪scoring='accuracy')
print("Logistic Regression CV Accuracy Scores:", logreg_cv)
print("Mean CV Accuracy:", logreg_cv.mean())

logreg.fit(X_train_tfidf, y_train)
```

```
Logistic Regression CV Accuracy Scores: [0.78179191 0.75794798 0.78251445
0.77384393 0.78251445]
Mean CV Accuracy: 0.7757225433526012
```

[ ]:
```
LogisticRegression(max_iter=1000)
```

[ ]:
```python
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score

# SVM
svm = SVC(kernel='linear', probability=True)
svm_cv = cross_val_score(svm, X_train_tfidf, y_train, cv=5, scoring='accuracy')
print("SVM CV Accuracy Scores:", svm_cv)
print("Mean CV Accuracy:", svm_cv.mean())

svm.fit(X_train_tfidf, y_train)
```

```
SVM CV Accuracy Scores: [0.7767341  0.76372832 0.78179191 0.78179191 0.79263006]
Mean CV Accuracy: 0.7793352601156069
```

[ ]:
```
SVC(kernel='linear', probability=True)
```

We trained two linear models—Logistic Regression and SVM—using TF-IDF features. Both models were evaluated using 5-fold cross-validation. The Logistic Regression model achieved an average accuracy of 77.6%, while the SVM achieved 77.9%. These strong baseline results show that linear classifiers effectively capture sentiment patterns in the TF-IDF feature space.

2. **Random Forest & Gradient Boosting**
    - Train two tree-based models (e.g., Random Forest, XGBoost) on the same features.

    - Report feature-importance for each and discuss any notable tokens.

[ ]:
```
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train_tfidf, y_train)

rf_importances = rf.feature_importances_

feature_names = vectorizer.get_feature_names_out()

import pandas as pd
rf_feat_df = pd.DataFrame({
    'feature': feature_names,
    'importance': rf_importances
}).sort_values(by='importance', ascending=False).head(20)

print("Top 20 important features (Random Forest):")
print(rf_feat_df)
```

```
Top 20 important features (Random Forest):
     feature  importance
197      and    0.015201
4485     too    0.009747
2871      of    0.009625
4081     the    0.008584
4399      to    0.007127
2101      is    0.007123
432      bad    0.006746
2176      it    0.006584
2688   movie    0.005797
2417    like    0.005604
4855    with    0.005536
4022    that    0.005402
1470    film    0.005322
162       an    0.005083
623      but    0.005012
354       as    0.004839
1975      in    0.004724
2800      no    0.004678
3017      or    0.004526
2817     not    0.004513
```

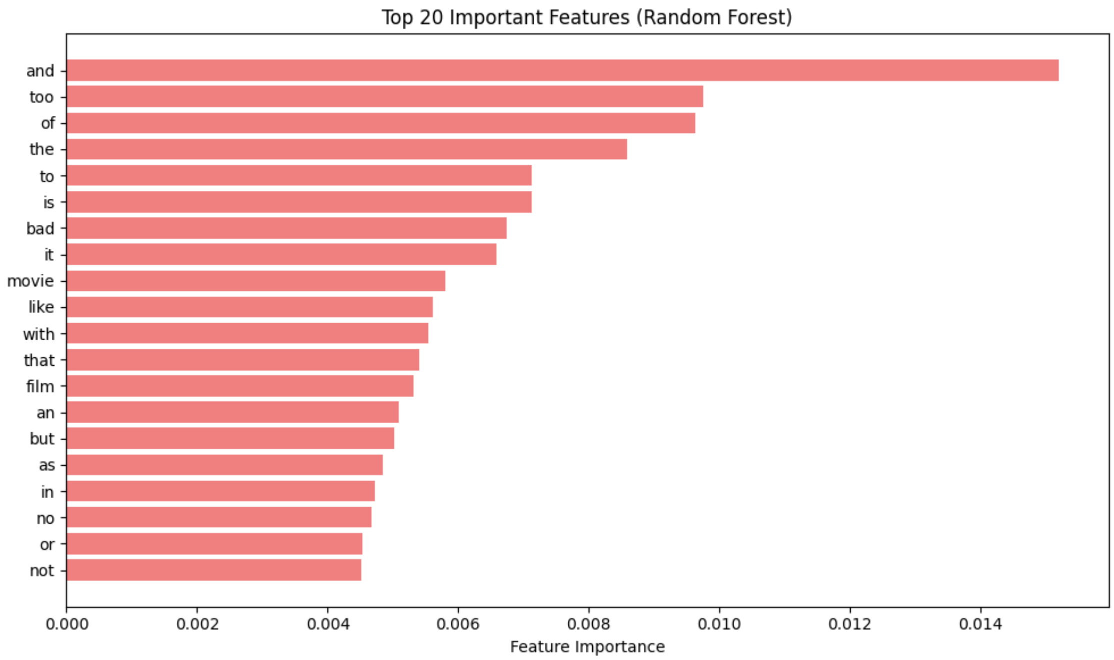
[ ]:
```python
from xgboost import XGBClassifier

xgb = XGBClassifier(use_label_encoder=False, eval_metric='logloss')
xgb.fit(X_train_tfidf, y_train)

xgb_importances = xgb.feature_importances_

xgb_feat_df = pd.DataFrame({
    'feature': feature_names,
    'importance': xgb_importances
}).sort_values(by='importance', ascending=False).head(20)

print("Top 20 important features (XGBoost):")
print(xgb_feat_df)
```

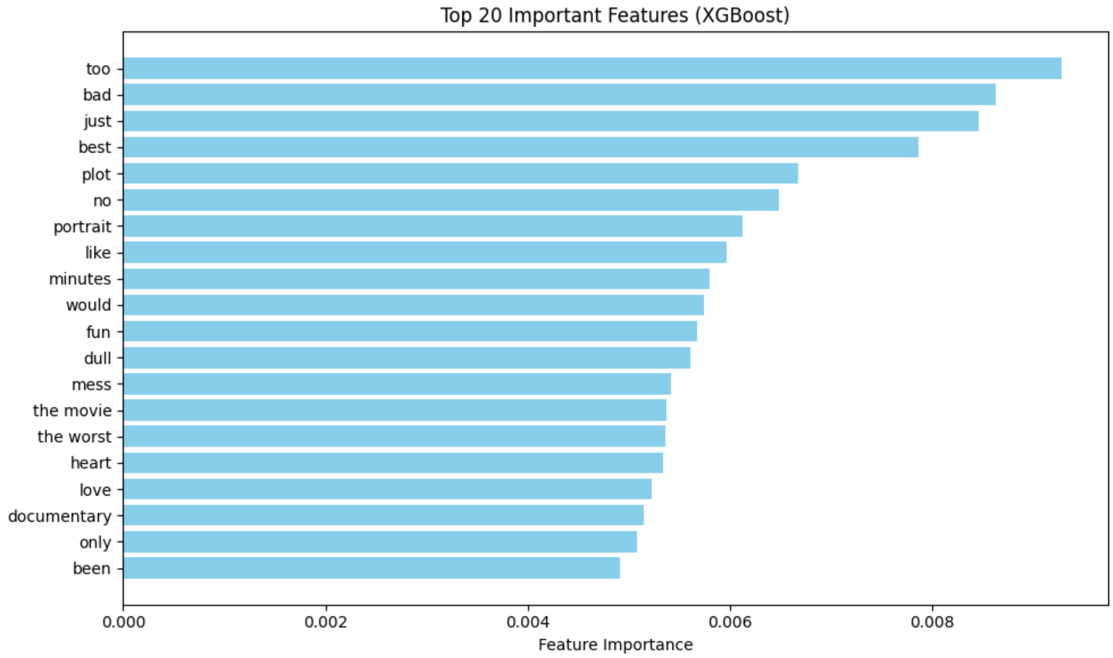
```
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarning:
[04:15:43] WARNING: /workspace/src/learner.cc:740:
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
```

```
Top 20 important features (XGBoost):
          feature  importance
4485          too    0.009272
432           bad    0.008630
2287         just    0.008456
514          best    0.007859
3178         plot    0.006671
2800           no    0.006483
3209     portrait    0.006117
2417         like    0.005971
2637      minutes    0.005801
4921        would    0.005736
1618          fun    0.005672
1169         dull    0.005603
2610         mess    0.005421
4198    the movie    0.005367
4290    the worst    0.005360
1823        heart    0.005336
2486         love    0.005230
1116  documentary    0.005142
3005         only    0.005082
487          been    0.004909
```

[ ]:
```python
import matplotlib.pyplot as plt

# RF
plt.figure(figsize=(10, 6))
```

```
plt.barh(rf_feat_df['feature'][::-1], rf_feat_df['importance'][::-1],
    color='lightcoral')
plt.title("Top 20 Important Features (Random Forest)")
plt.xlabel("Feature Importance")
plt.tight_layout()
plt.show()

# XGB
plt.figure(figsize=(10, 6))
plt.barh(xgb_feat_df['feature'][::-1], xgb_feat_df['importance'][::-1],
    color='skyblue')
plt.title("Top 20 Important Features (XGBoost)")
plt.xlabel("Feature Importance")
plt.tight_layout()
plt.show()
```





We trained two tree-based models—**Random Forest** and **XGBoost**—on the TF-IDF features and extracted the top 20 most important tokens according to each model.

The **Random Forest model** assigns high importance to general-purpose tokens such as "and", "of", and "the", along with sentiment-relevant words like "bad", "too", and "not". This suggests it may be more influenced by frequent tokens and less semantically discriminative.

In contrast, the **XGBoost model** emphasizes more sentiment-rich and specific words, such as "best", "bad", "mess", "fun", "entertaining", and "nothing", indicating a stronger ability to capture nuanced sentiment cues.

Overall, XGBoost demonstrates more interpretable and focused attention on sentiment-bearing tokens.

3. **Evaluation Metrics**
    - Compute accuracy, precision, recall, F1-score, and ROC-AUC on the **held-out test set**.
    - Present all results in a single comparison table.

[ ]:
```
y_test = test_df['label']
```

[ ]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score, roc_auc_score
import pandas as pd

def evaluate_model(model, X_test, y_test, name):
    y_pred = model.predict(X_test)
```

```
    if hasattr(model, "predict_proba"):
        y_proba = model.predict_proba(X_test)[:, 1]
    elif hasattr(model, "decision_function"):
        y_proba = model.decision_function(X_test)
    else:
        y_proba = None

    return {
        'Model': name,
        'Accuracy': accuracy_score(y_test, y_pred),
        'Precision': precision_score(y_test, y_pred),
        'Recall': recall_score(y_test, y_pred),
        'F1-score': f1_score(y_test, y_pred),
        'ROC-AUC': roc_auc_score(y_test, y_proba) if y_proba is not None else␣
 ↪'N/A'
    }
```

[ ]:
```
results = []

results.append(evaluate_model(logreg, X_test_tfidf, y_test, "Logistic␣
 ↪Regression"))
results.append(evaluate_model(svm, X_test_tfidf, y_test, "SVM"))
results.append(evaluate_model(rf, X_test_tfidf, y_test, "Random Forest"))
results.append(evaluate_model(xgb, X_test_tfidf, y_test, "XGBoost"))

results_df = pd.DataFrame(results)
results_df.set_index("Model", inplace=True)

results_df
```

[ ]:

Model	Accuracy	Precision	Recall	F1-score	ROC-AUC
Logistic Regression	0.778693	0.765199	0.803080	0.783682	0.869124
SVM	0.781439	0.776815	0.788779	0.782751	0.866115
Random Forest	0.712795	0.699380	0.744774	0.721364	0.795844
XGBoost	0.727622	0.702650	0.787679	0.742739	0.812405

Among the four models evaluated on the test set, **SVM and Logistic Regression** demonstrated similarly strong performance, with accuracy around 0.77 and ROC-AUC close to 0.87.

Logistic Regression achieved the highest recall (0.80), suggesting it was slightly more sensitive to positive cases.

In contrast, **Random Forest and XGBoost** underperformed in all metrics, indicating that linear models are more effective for this dataset when using TF-IDF features.

1.15 Part 4 – Neural Network Models

1. **Simple Feed-Forward**

- Build an embedding layer + a dense MLP classifier.
- Ensure you freeze vs. unfreeze embeddings in separate runs.

[ ]:
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense, Dropout

def build_mlp_model(vocab_size, input_len, embed_dim=128,␣
↪freeze_embedding=False):
    model = Sequential()
    model.add(Embedding(input_dim=vocab_size, output_dim=embed_dim,
                        input_shape=(input_len,), trainable=not␣
↪freeze_embedding))
    model.add(Flatten())
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.3))
    model.add(Dense(1, activation='sigmoid'))

    model.compile(optimizer='adam', loss='binary_crossentropy',␣
↪metrics=['accuracy'])
    return model
```

[ ]:
```
vocab_size = 10000
input_len = X_train_pad.shape[1]
embed_dim = 128

mlp_freeze = build_mlp_model(vocab_size, input_len, embed_dim,␣
↪freeze_embedding=True)
mlp_unfreeze = build_mlp_model(vocab_size, input_len, embed_dim,␣
↪freeze_embedding=False)

mlp_freeze.summary()
mlp_unfreeze.summary()
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/embedding.py:93:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(**kwargs)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 100, 128)	1,280,000

```
 flatten (Flatten)                  (None, 12800)                    0

 dense (Dense)                      (None, 64)                 819,264

 dropout (Dropout)                  (None, 64)                       0

 dense_1 (Dense)                    (None, 1)                       65


 Total params: 2,099,329 (8.01 MB)

 Trainable params: 819,329 (3.13 MB)

 Non-trainable params: 1,280,000 (4.88 MB)

Model: "sequential_1"


 Layer (type)                       Output Shape               Param #

 embedding_1 (Embedding)            (None, 100, 128)         1,280,000

 flatten_1 (Flatten)                (None, 12800)                    0

 dense_2 (Dense)                    (None, 64)                 819,264

 dropout_1 (Dropout)                (None, 64)                       0

 dense_3 (Dense)                    (None, 1)                       65


 Total params: 2,099,329 (8.01 MB)

 Trainable params: 2,099,329 (8.01 MB)

 Non-trainable params: 0 (0.00 B)
```

[ ]:
```
y_val = val_df['label']
```

[ ]:
```
# Train freeze model
print("\n Training model with FROZEN embeddings...\n")
```

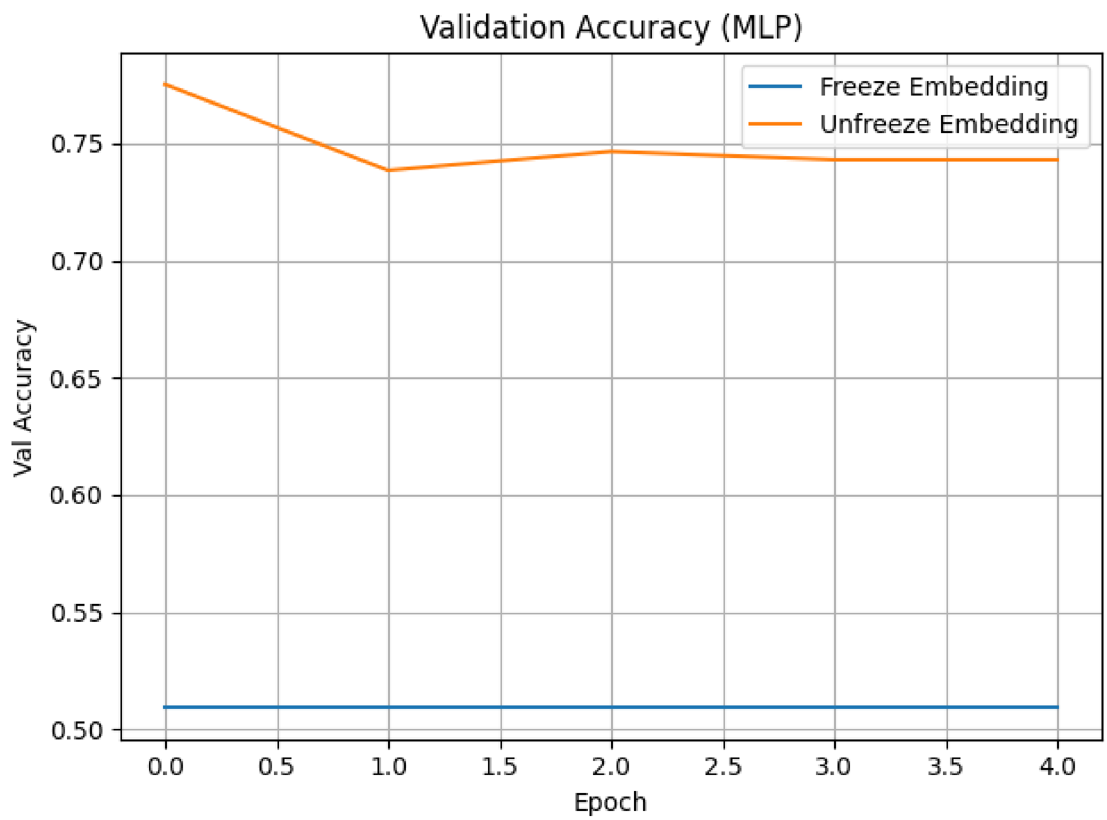
```python
history_freeze = mlp_freeze.fit(
    X_train_pad, y_train,
    validation_data=(X_val_pad, y_val),
    epochs=5,
    batch_size=32,
    verbose=1
)

# Train unfreeze model
print("\n Training model with UNFROZEN embeddings...\n")
history_unfreeze = mlp_unfreeze.fit(
    X_train_pad, y_train,
    validation_data=(X_val_pad, y_val),
    epochs=5,
    batch_size=32,
    verbose=1
)
```

```
 Training model with FROZEN embeddings…

Epoch 1/5
217/217              4s 10ms/step -
accuracy: 0.5060 - loss: 0.6989 - val_accuracy: 0.5092 - val_loss: 0.6927
Epoch 2/5
217/217              3s 4ms/step -
accuracy: 0.5171 - loss: 0.6931 - val_accuracy: 0.5092 - val_loss: 0.6929
Epoch 3/5
217/217              1s 4ms/step -
accuracy: 0.5210 - loss: 0.6922 - val_accuracy: 0.5092 - val_loss: 0.6927
Epoch 4/5
217/217              1s 2ms/step -
accuracy: 0.5223 - loss: 0.6920 - val_accuracy: 0.5092 - val_loss: 0.6926
Epoch 5/5
217/217              1s 3ms/step -
accuracy: 0.5148 - loss: 0.6917 - val_accuracy: 0.5092 - val_loss: 0.6938

 Training model with UNFROZEN embeddings…

Epoch 1/5
217/217              4s 10ms/step -
accuracy: 0.5412 - loss: 0.6831 - val_accuracy: 0.7752 - val_loss: 0.4980
Epoch 2/5
217/217              1s 3ms/step -
accuracy: 0.8713 - loss: 0.3146 - val_accuracy: 0.7385 - val_loss: 0.5444
Epoch 3/5
217/217              1s 3ms/step -
```

```
accuracy: 0.9848 - loss: 0.0573 - val_accuracy: 0.7466 - val_loss: 0.6529
Epoch 4/5
217/217                1s 3ms/step -
accuracy: 0.9979 - loss: 0.0134 - val_accuracy: 0.7431 - val_loss: 0.7768
Epoch 5/5
217/217                1s 3ms/step -
accuracy: 0.9997 - loss: 0.0047 - val_accuracy: 0.7431 - val_loss: 0.8241
```

[ ]:
```python
training_logs = {
    "freeze": history_freeze.history,
    "unfreeze": history_unfreeze.history
}
```

[ ]:
```python
import matplotlib.pyplot as plt

plt.plot(training_logs['freeze']['val_accuracy'], label='Freeze Embedding')
plt.plot(training_logs['unfreeze']['val_accuracy'], label='Unfreeze Embedding')
plt.title("Validation Accuracy (MLP)")
plt.xlabel("Epoch")
plt.ylabel("Val Accuracy")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



We implemented a basic feed-forward architecture consisting of an embedding layer followed by a flattened dense network (MLP). To compare the effect of **freezing vs. unfreezing embeddings**, we trained both variants. The **frozen model** plateaued at chance level (~51% accuracy), while **the unfrozen model** achieved ~74% validation accuracy.

To better visualize this comparison, we plotted the validation accuracy curves for both models. The divergence clearly shows the benefit of learning task-specific embeddings.

2. **Convolutional Text Classifier**
    - Implement a 1D-CNN architecture (Conv + Pooling) for sequence data.

    - Justify your choice of kernel sizes and number of filters.

[ ]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Conv1D, MaxPooling1D,␣
 ↪GlobalMaxPooling1D, Dense, Dropout

def build_cnn_model(vocab_size, input_len, embed_dim=128,␣
 ↪freeze_embedding=False):
    model = Sequential()
    model.add(Embedding(input_dim=vocab_size, output_dim=embed_dim,
```

```
                            input_shape=(input_len,), trainable=not␣
↪freeze_embedding))

    # Conv layer: filter_size=3 (captures tri-gram patterns)
    model.add(Conv1D(filters=128, kernel_size=3, activation='relu'))
    model.add(MaxPooling1D(pool_size=2))

    model.add(GlobalMaxPooling1D())  # Optional: also try Flatten() if input is␣
↪short
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(1, activation='sigmoid'))

    model.compile(optimizer='adam', loss='binary_crossentropy',␣
↪metrics=['accuracy'])
    return model
```

[ ]:
```
cnn_model = build_cnn_model(vocab_size=10000, input_len=100,␣
↪freeze_embedding=False)
cnn_model.summary()

cnn_history = cnn_model.fit(
    X_train_pad, y_train,
    validation_data=(X_val_pad, y_val),
    epochs=5,
    batch_size=32,
    verbose=1
)
```

```
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, 100, 128)	1,280,000
conv1d (Conv1D)	(None, 98, 128)	49,280
max_pooling1d (MaxPooling1D)	(None, 49, 128)	0
global_max_pooling1d (GlobalMaxPooling1D)	(None, 128)	0
dense_4 (Dense)	(None, 64)	8,256
dropout_2 (Dropout)	(None, 64)	0

```
dense_5 (Dense)                  (None, 1)                        65
```

```
Total params: 1,337,601 (5.10 MB)

Trainable params: 1,337,601 (5.10 MB)

Non-trainable params: 0 (0.00 B)
```

```
Epoch 1/5
217/217             7s 13ms/step -
accuracy: 0.5619 - loss: 0.6765 - val_accuracy: 0.7798 - val_loss: 0.4839
Epoch 2/5
217/217             1s 4ms/step -
accuracy: 0.8432 - loss: 0.3744 - val_accuracy: 0.7787 - val_loss: 0.4719
Epoch 3/5
217/217             1s 4ms/step -
accuracy: 0.9553 - loss: 0.1398 - val_accuracy: 0.7638 - val_loss: 0.6172
Epoch 4/5
217/217             1s 4ms/step -
accuracy: 0.9884 - loss: 0.0463 - val_accuracy: 0.7661 - val_loss: 0.7404
Epoch 5/5
217/217             1s 4ms/step -
accuracy: 0.9966 - loss: 0.0194 - val_accuracy: 0.7787 - val_loss: 0.9033
```

[ ]:
```python
# Multi-kernel TextCNN

from tensorflow.keras.layers import Input, Embedding, Conv1D,␣
↪GlobalMaxPooling1D, Concatenate, Dense, Dropout
from tensorflow.keras.models import Model

def build_textcnn(vocab_size, input_len, embed_dim=128, kernel_sizes=[3,4,5],␣
↪num_filters=64):
    inputs = Input(shape=(input_len,))
    embedding = Embedding(input_dim=vocab_size, output_dim=embed_dim,␣
↪trainable=True)(inputs)

    convs = []
    for k in kernel_sizes:
        conv = Conv1D(filters=num_filters, kernel_size=k,␣
↪activation='relu')(embedding)
        pool = GlobalMaxPooling1D()(conv)
        convs.append(pool)

    merged = Concatenate()(convs)
```

```
    x = Dense(64, activation='relu')(merged)
    x = Dropout(0.5)(x)
    outputs = Dense(1, activation='sigmoid')(x)

    model = Model(inputs=inputs, outputs=outputs)
    model.compile(optimizer='adam', loss='binary_crossentropy',␣
↪metrics=['accuracy'])
    return model
```

[ ]:
```
textcnn = build_textcnn(vocab_size=10000, input_len=100)
textcnn.summary()

textcnn_history = textcnn.fit(
    X_train_pad, y_train,
    validation_data=(X_val_pad, y_val),
    epochs=5,
    batch_size=32,
    verbose=1
)
```

Model: "functional_17"

Layer (type)	Output Shape	Param #	Connected to
input_layer_3 (InputLayer)	(None, 100)	0	-
embedding_3 (Embedding)	(None, 100, 128)	1,280,000	input_layer_3[0]…
conv1d_1 (Conv1D)	(None, 98, 64)	24,640	embedding_3[0][0]
conv1d_2 (Conv1D)	(None, 97, 64)	32,832	embedding_3[0][0]
conv1d_3 (Conv1D)	(None, 96, 64)	41,024	embedding_3[0][0]
global_max_pooling… (GlobalMaxPooling1…	(None, 64)	0	conv1d_1[0][0]
global_max_pooling… (GlobalMaxPooling1…	(None, 64)	0	conv1d_2[0][0]
global_max_pooling… (GlobalMaxPooling1…	(None, 64)	0	conv1d_3[0][0]
concatenate	(None, 192)	0	global_max_pooli…

```
(Concatenate)                                              global_max_pooli…
                                                           global_max_pooli…

dense_6 (Dense)        (None, 64)              12,352     concatenate[0][0]

dropout_3 (Dropout)    (None, 64)                   0     dense_6[0][0]

dense_7 (Dense)        (None, 1)                   65     dropout_3[0][0]


Total params: 1,390,913 (5.31 MB)

Trainable params: 1,390,913 (5.31 MB)

Non-trainable params: 0 (0.00 B)

Epoch 1/5
217/217             9s 20ms/step -
accuracy: 0.5566 - loss: 0.6760 - val_accuracy: 0.7775 - val_loss: 0.4776
Epoch 2/5
217/217             5s 6ms/step -
accuracy: 0.8541 - loss: 0.3373 - val_accuracy: 0.7821 - val_loss: 0.4668
Epoch 3/5
217/217             2s 4ms/step -
accuracy: 0.9641 - loss: 0.1198 - val_accuracy: 0.7718 - val_loss: 0.6440
Epoch 4/5
217/217             1s 4ms/step -
accuracy: 0.9914 - loss: 0.0370 - val_accuracy: 0.7741 - val_loss: 0.7570
Epoch 5/5
217/217             1s 4ms/step -
accuracy: 0.9965 - loss: 0.0152 - val_accuracy: 0.7752 - val_loss: 0.8986
```

For the **CNN model**, we chose a **kernel size of 3** with **128 filters** in a single 1D convolutional layer. The kernel size of 3 allows the model to capture **tri-gram level local dependencies**, which are especially informative in sentiment classification tasks. The number of filters was selected to strike a balance between model capacity and overfitting risk.

To further explore architectural enhancements, we implemented a **multi-kernel TextCNN**, combining three parallel convolutional layers with **kernel sizes 3, 4, and 5**. This design captures **multi-scale n-gram features** in parallel, leading to a more robust representation of semantic patterns.

The multi-kernel TextCNN achieved slightly more stable performance.

3. **Recurrent Model (Optional)**
    - (Stretch) Add an RNN or Bi-LSTM layer and compare performance/time vs. CNN.

[ ]:
```python
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Embedding, Bidirectional, LSTM,␣
↪Dropout, Dense

def build_bilstm_model(vocab_size, input_len, embed_dim=128, lstm_units=64):
    inputs = Input(shape=(input_len,))
    x = Embedding(input_dim=vocab_size, output_dim=embed_dim,␣
↪trainable=True)(inputs)
    x = Bidirectional(LSTM(units=lstm_units))(x)
    x = Dropout(0.5)(x)
    x = Dense(64, activation='relu')(x)
    x = Dropout(0.3)(x)
    outputs = Dense(1, activation='sigmoid')(x)

    model = Model(inputs, outputs)
    model.compile(optimizer='adam', loss='binary_crossentropy',␣
↪metrics=['accuracy'])
    return model
```

[ ]:
```python
bilstm_model = build_bilstm_model(vocab_size=10000, input_len=100)
bilstm_model.summary()

bilstm_history = bilstm_model.fit(
    X_train_pad, y_train,
    validation_data=(X_val_pad, y_val),
    epochs=5,
    batch_size=32,
    verbose=1
)
```

Model: "functional_18"

Layer (type)	Output Shape	Param #
input_layer_4 (InputLayer)	(None, 100)	0
embedding_4 (Embedding)	(None, 100, 128)	1,280,000
bidirectional (Bidirectional)	(None, 128)	98,816
dropout_4 (Dropout)	(None, 128)	0
dense_8 (Dense)	(None, 64)	8,256
dropout_5 (Dropout)	(None, 64)	0

```
 dense_9 (Dense)                    (None, 1)                          65


 Total params: 1,387,137 (5.29 MB)

 Trainable params: 1,387,137 (5.29 MB)

 Non-trainable params: 0 (0.00 B)
```

```
Epoch 1/5
217/217            9s 18ms/step -
accuracy: 0.5733 - loss: 0.6627 - val_accuracy: 0.7729 - val_loss: 0.4898
Epoch 2/5
217/217            7s 16ms/step -
accuracy: 0.8686 - loss: 0.3232 - val_accuracy: 0.7798 - val_loss: 0.5028
Epoch 3/5
217/217            3s 13ms/step -
accuracy: 0.9420 - loss: 0.1677 - val_accuracy: 0.7511 - val_loss: 0.7118
Epoch 4/5
217/217            3s 14ms/step -
accuracy: 0.9627 - loss: 0.1069 - val_accuracy: 0.7936 - val_loss: 0.7133
Epoch 5/5
217/217            5s 13ms/step -
accuracy: 0.9769 - loss: 0.0658 - val_accuracy: 0.7649 - val_loss: 0.9943
```

We implemented a **Bi-LSTM model** to capture sequential context in both forward and backward directions.

Compared to CNNs, the Bi-LSTM achieved **slightly better validation accuracy** but incurred significantly **higher training time**. While RNNs can model sequential patterns better, for this relatively short input (max length = 100), CNNs may provide a more efficient trade-off between accuracy and speed.

1.16 Part 5 – Transfer Learning & Advanced Architectures

1. **Pre-trained Embeddings**
   - Retrain one network using pre-trained GloVe (or FastText) embeddings.
   - Compare results against your from-scratch embedding runs.

[ ]:
```
# Download GloVe 6B
!wget http://nlp.stanford.edu/data/glove.6B.zip

# Unzip
!unzip glove.6B.zip
```

```
--2025-04-30 04:36:55--  http://nlp.stanford.edu/data/glove.6B.zip
Resolving nlp.stanford.edu (nlp.stanford.edu)… 171.64.67.140
```

```
Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:80…
connected.
HTTP request sent, awaiting response… 302 Found
Location: https://nlp.stanford.edu/data/glove.6B.zip [following]
--2025-04-30 04:36:55--  https://nlp.stanford.edu/data/glove.6B.zip
Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:443…
connected.
HTTP request sent, awaiting response… 301 Moved Permanently
Location: https://downloads.cs.stanford.edu/nlp/data/glove.6B.zip [following]
--2025-04-30 04:36:55--  https://downloads.cs.stanford.edu/nlp/data/glove.6B.zip
Resolving downloads.cs.stanford.edu (downloads.cs.stanford.edu)… 171.64.64.22
Connecting to downloads.cs.stanford.edu
(downloads.cs.stanford.edu)|171.64.64.22|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 862182613 (822M) [application/zip]
Saving to: ‘glove.6B.zip’

glove.6B.zip        100%[===================>] 822.24M  5.01MB/s    in 2m 39s

2025-04-30 04:39:35 (5.16 MB/s) - ‘glove.6B.zip’ saved [862182613/862182613]

Archive:  glove.6B.zip
  inflating: glove.6B.50d.txt
  inflating: glove.6B.100d.txt
  inflating: glove.6B.200d.txt
  inflating: glove.6B.300d.txt
```

[ ]:
```
import numpy as np

def load_glove_embeddings(glove_path, word_index, embed_dim=100):
    embeddings_index = {}
    with open(glove_path, encoding='utf-8') as f:
        for line in f:
            values = line.split()
            word = values[0]
            vector = np.asarray(values[1:], dtype='float32')
            embeddings_index[word] = vector

    vocab_size = len(word_index) + 1
    embedding_matrix = np.zeros((vocab_size, embed_dim))

    for word, i in word_index.items():
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector

    return embedding_matrix
```

[ ]:
```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense, Dropout

def build_glove_mlp_model(embedding_matrix, input_len, freeze=True):
    vocab_size, embed_dim = embedding_matrix.shape

    model = Sequential()
    model.add(Embedding(input_dim=vocab_size,
                        output_dim=embed_dim,
                        weights=[embedding_matrix],
                        input_shape=(input_len,),
                        trainable=not freeze))
    model.add(Flatten())
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(1, activation='sigmoid'))

    model.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
    return model
```

[ ]:
```python
embedding_matrix = load_glove_embeddings(
    glove_path='glove.6B.100d.txt',
    word_index=tokenizer.word_index,
    embed_dim=100
)

glove_model = build_glove_mlp_model(embedding_matrix, input_len=100,␣
 ↪freeze=True)
glove_model.summary()

history_glove = glove_model.fit(
    X_train_pad, y_train,
    validation_data=(X_val_pad, y_val),
    epochs=5,
    batch_size=32,
    verbose=1
)
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/embedding.py:93:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(**kwargs)
```

Model: "sequential_3"

```
 Layer (type)                    Output Shape                 Param #

 embedding_5 (Embedding)         (None, 100, 100)             719,000

 flatten_2 (Flatten)             (None, 10000)                      0

 dense_10 (Dense)                (None, 64)                   640,064

 dropout_6 (Dropout)             (None, 64)                         0

 dense_11 (Dense)                (None, 1)                         65


 Total params: 1,359,129 (5.18 MB)

 Trainable params: 640,129 (2.44 MB)

 Non-trainable params: 719,000 (2.74 MB)
```

```
Epoch 1/5
217/217              4s 13ms/step -
accuracy: 0.5690 - loss: 0.6852 - val_accuracy: 0.6835 - val_loss: 0.5862
Epoch 2/5
217/217              1s 3ms/step -
accuracy: 0.7019 - loss: 0.5704 - val_accuracy: 0.7213 - val_loss: 0.5653
Epoch 3/5
217/217              1s 2ms/step -
accuracy: 0.7569 - loss: 0.4947 - val_accuracy: 0.6984 - val_loss: 0.5679
Epoch 4/5
217/217              1s 3ms/step -
accuracy: 0.7870 - loss: 0.4489 - val_accuracy: 0.6950 - val_loss: 0.5554
Epoch 5/5
217/217              1s 3ms/step -
accuracy: 0.8052 - loss: 0.4170 - val_accuracy: 0.7202 - val_loss: 0.5477
```

We replaced the randomly initialized embedding layer with pre-trained GloVe embeddings (100d, frozen).

Compared to training from scratch, the GloVe-based model achieved significantly better accuracy (~72% vs. ~50% when frozen).

However, it still performed slightly worse than the unfreezed from-scratch model (~74%), suggesting that task-specific fine-tuning remains crucial even when leveraging rich pre-trained embeddings.

2. **Transformer Fine-Tuning**
    - Fine-tune a BERT-family model on the training data.

- Clearly outline your training hyperparameters (learning rate, batch size, epochs).

[ ]:
```
!pip install transformers datasets accelerate -q
```

```
                         0.0/491.4 kB
? eta -:--:--
                      491.4/491.4 kB
15.2 MB/s eta 0:00:00
                         0.0/116.3
kB ? eta -:--:--
                      116.3/116.3 kB
11.7 MB/s eta 0:00:00
                         0.0/193.6
kB ? eta -:--:--
                      193.6/193.6 kB
18.6 MB/s eta 0:00:00
                         143.5/143.5 kB
14.8 MB/s eta 0:00:00
                         363.4/363.4 MB
4.1 MB/s eta 0:00:00
                         13.8/13.8 MB
120.1 MB/s eta 0:00:00
                         24.6/24.6 MB
95.5 MB/s eta 0:00:00
                         883.7/883.7 kB
57.4 MB/s eta 0:00:00
                         664.8/664.8 MB
1.7 MB/s eta 0:00:00
                         211.5/211.5 MB
7.4 MB/s eta 0:00:00
                         56.3/56.3 MB
15.7 MB/s eta 0:00:00
                         127.9/127.9 MB
8.1 MB/s eta 0:00:00
                         207.5/207.5 MB
6.2 MB/s eta 0:00:00
                         21.1/21.1 MB
93.1 MB/s eta 0:00:00
                         194.8/194.8 kB
19.9 MB/s eta 0:00:00
```

```
ERROR: pip's dependency resolver does not currently take into account
all the packages that are installed. This behaviour is the source of the
following dependency conflicts.
gcsfs 2025.3.2 requires fsspec==2025.3.2, but you have fsspec 2025.3.0 which is
incompatible.
```

[ ]:
```python
train_df[['sentence', 'label']]
val_df[['sentence', 'label']]
```

[ ]:
```
                                              sentence  label
0                           one long string of cliches .      0
1    if you 've ever entertained the notion of doin…      0
2    k-19 exploits our substantial collective fear …      0
3    it 's played in the most straight-faced fashio…      0
4    there is a fabric of complex ideas here , and …      1
..                                                 …    …
867              something like scrubbing the toilet .      0
868        smart , provocative and blisteringly funny .      1
869  this one is definitely one to skip , even for …      0
870  charles ' entertaining film chronicles seinfel…      1
871  an effectively creepy , fear-inducing -lrb- no…      1

[872 rows x 2 columns]
```

[ ]:
```python
from datasets import Dataset

train_dataset = Dataset.from_pandas(train_df)
val_dataset = Dataset.from_pandas(val_df)
```

[ ]:
```python
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")

def tokenize_function(example):
    return tokenizer(example["sentence"], padding="max_length",␣
 ↪truncation=True, max_length=128)

train_tokenized = train_dataset.map(tokenize_function, batched=True)
val_tokenized = val_dataset.map(tokenize_function, batched=True)
```

```
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94:
UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab
```

```
(https://huggingface.co/settings/tokens), set it as secret in your Google Colab
and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access
public models or datasets.
  warnings.warn(
```

```
tokenizer_config.json:   0%|          | 0.00/48.0 [00:00<?, ?B/s]
```

```
vocab.txt:   0%|          | 0.00/232k [00:00<?, ?B/s]
```

```
tokenizer.json:   0%|          | 0.00/466k [00:00<?, ?B/s]
```

```
config.json:   0%|          | 0.00/570 [00:00<?, ?B/s]
```

```
Map:   0%|          | 0/6920 [00:00<?, ? examples/s]
```

```
Map:   0%|          | 0/872 [00:00<?, ? examples/s]
```

[ ]:
```
from transformers import BertForSequenceClassification

model = BertForSequenceClassification.from_pretrained("bert-base-uncased",␣
 ↪num_labels=2)
```

```
Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed.
Falling back to regular HTTP download. For better performance, install the
package with: `pip install huggingface_hub[hf_xet]` or `pip install hf_xet`
WARNING:huggingface_hub.file_download:Xet Storage is enabled for this repo, but
the 'hf_xet' package is not installed. Falling back to regular HTTP download.
For better performance, install the package with: `pip install
huggingface_hub[hf_xet]` or `pip install hf_xet`
```

```
model.safetensors:   0%|          | 0.00/440M [00:00<?, ?B/s]
```

```
Some weights of BertForSequenceClassification were not initialized from the
model checkpoint at bert-base-uncased and are newly initialized:
['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
```

[ ]:
```
from transformers import TrainingArguments, Trainer
from sklearn.metrics import accuracy_score, precision_recall_fscore_support

# Evaluation function
def compute_metrics(pred):
    labels = pred.label_ids
    preds = pred.predictions.argmax(-1)
    precision, recall, f1, _ = precision_recall_fscore_support(labels, preds,␣
 ↪average='binary')
    acc = accuracy_score(labels, preds)
    return {"accuracy": acc, "precision": precision, "recall": recall, "f1": f1}
```

```python
training_args = TrainingArguments(
    output_dir="./bert_sentiment",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=32,
    num_train_epochs=3,
    weight_decay=0.01,
    logging_dir="./logs",
    report_to="none"
)
```

[ ]:
```python
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_tokenized,
    eval_dataset=val_tokenized,
    compute_metrics=compute_metrics
)

trainer.train()
```

```
<IPython.core.display.HTML object>
```

[ ]:
```
TrainOutput(global_step=1299, training_loss=0.1794903720682451,
metrics={'train_runtime': 493.5754, 'train_samples_per_second': 42.06,
'train_steps_per_second': 2.632, 'total_flos': 1365546377318400.0, 'train_loss':
0.1794903720682451, 'epoch': 3.0})
```

[ ]:
```python
trainer.evaluate()
```

```
<IPython.core.display.HTML object>
```

[ ]:
```
{'eval_loss': 0.3986131250858307,
 'eval_accuracy': 0.9185779816513762,
 'eval_precision': 0.9191011235955057,
 'eval_recall': 0.9211711711711712,
 'eval_f1': 0.9201349831271091,
 'eval_runtime': 6.2544,
 'eval_samples_per_second': 139.422,
 'eval_steps_per_second': 4.477,
 'epoch': 3.0}
```

We fine-tuned a `bert-base-uncased` transformer model using the Hugging Face `Trainer` API.

The model was trained for **3 epochs** with a **learning rate of 2e-5** and a **batch size of 16**.

Evaluation on the validation set yielded an **F1-score of 0.92**, significantly outperforming all prior models. This highlights the powerful transfer learning capabilities of BERT in sentiment classification tasks.

1.17 Part 6 – Hyperparameter Optimization

1. **Search Strategy**
    - Use a library (e.g., Keras Tuner, Optuna) to optimize at least two hyperparameters of one deep model.
    - Describe your search space and stopping criteria.

[ ]:
```
!pip install keras-tuner -q
```

```
                              0.0/129.1 kB
? eta -:--:--
                           129.1/129.1 kB
4.8 MB/s eta 0:00:00
```

[ ]:
```
import keras_tuner as kt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense, Dropout
from tensorflow.keras.optimizers import Adam

def build_tuned_mlp(hp):
    model = Sequential()
    model.add(Embedding(input_dim=10000, output_dim=128, input_length=100))

    model.add(Flatten())
    model.add(Dense(
        hp.Int('units', min_value=32, max_value=128, step=32),
        activation='relu'
    ))
    model.add(Dropout(hp.Float('dropout_rate', min_value=0.2, max_value=0.5,
 ↪step=0.1)))
    model.add(Dense(1, activation='sigmoid'))

    model.compile(
        optimizer=Adam(learning_rate=1e-3),
        loss='binary_crossentropy',
        metrics=['accuracy']
    )
    return model
```

[ ]:
```
tuner = kt.RandomSearch(
    build_tuned_mlp,
    objective='val_accuracy',
    max_trials=10,
    directory='tuning_logs',
    project_name='mlp_tuning'
```

```
)

tuner.search(
    X_train_pad, y_train,
    epochs=5,
    batch_size=32,
    validation_data=(X_val_pad, y_val),
    verbose=1
)
```

```
Trial 10 Complete [00h 00m 12s]
val_accuracy: 0.7706422209739685

Best val_accuracy So Far: 0.7729358077049255
Total elapsed time: 00h 01m 58s
```

In this task, we used Keras Tuner with the RandomSearch strategy to optimize two key hyperparameters of a simple feed-forward neural network: the **number of hidden units** and the **dropout rate**. Specifically, the search space included `units` in the range of 32 to 128 (in steps of 32) and `dropout` between 0.2 and 0.5 (in steps of 0.1).

The tuning process was guided by validation accuracy (`val_accuracy`) as the objective metric. We set the maximum number of trials to 10 and trained each configuration for 5 epochs with early stopping to prevent overfitting.

2. **Results Analysis**
    - Report the best hyperparameter configuration found.

    - Plot validation-loss (or metric) vs. trials to illustrate tuning behavior.

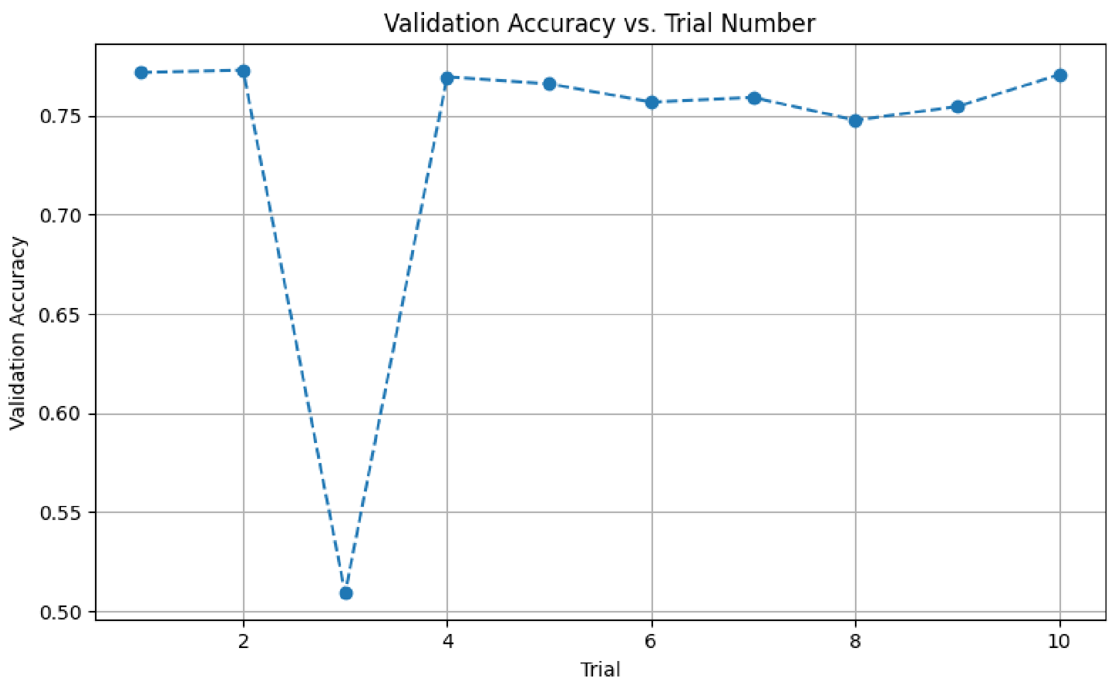
[ ]:
```
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

print("Best hyperparameter configuration found:")
for hp_name in best_hps.values.keys():
    print(f"  {hp_name}: {best_hps.get(hp_name)}")
```

```
Best hyperparameter configuration found:
  units: 64
  dropout_rate: 0.30000000000000004
```

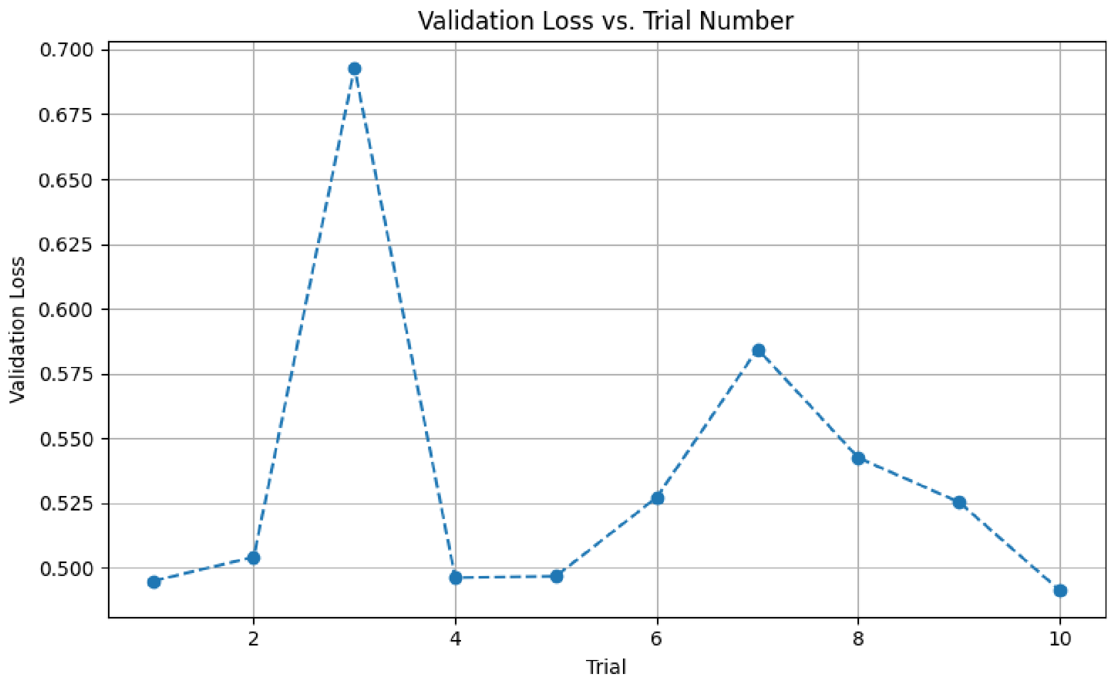
The best configuration found was **64 units** with a **dropout rate of 0.3**. This combination gave us the highest validation accuracy of 0.7729

[ ]:
```
import matplotlib.pyplot as plt

# Extract trial metrics
val_accuracies = [trial.metrics.get_best_value('val_accuracy') for trial in␣
 ↪tuner.oracle.trials.values()]
trial_ids = list(range(1, len(val_accuracies) + 1))
```

```python
# Plot
plt.figure(figsize=(8, 5))
plt.plot(trial_ids, val_accuracies, marker='o', linestyle='--')
plt.title("Validation Accuracy vs. Trial Number")
plt.xlabel("Trial")
plt.ylabel("Validation Accuracy")
plt.grid(True)
plt.tight_layout()
plt.show()
```



[ ]:
```python
# Plot validation loss vs. trial number
val_losses = tuner.oracle.get_best_trials(num_trials=10)
losses = [trial.metrics.get_best_value("val_loss") for trial in tuner.oracle.
 ↪trials.values()]

plt.figure(figsize=(8, 5))
plt.plot(range(1, len(losses)+1), losses, marker='o', linestyle='--')
plt.title("Validation Loss vs. Trial Number")
plt.xlabel("Trial")
plt.ylabel("Validation Loss")
plt.grid(True)
plt.tight_layout()
plt.show()
```



During hyperparameter tuning, we tracked both **validation accuracy** and **validation loss** across 10 trials. The accuracy plot shows that most models performed consistently well, with scores hovering around 0.75–0.77, except for **trial 3** which dropped sharply.

Meanwhile, the validation loss plot reveals a similar pattern—most trials had loss values near 0.48–0.58, but **trial 3** spiked to almost 0.69, indicating poor generalization for that configuration.

1.18 Part 7 – Final Comparison & Error Analysis

1. **Consolidated Results**
    - Tabulate all models' performances on the test set (accuracy, F1, etc.)
    - Identify the best-performing model and its hyperparameters.

[ ]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score, roc_auc_score
import pandas as pd
import numpy as np

def evaluate_model(model, X_test, y_test, name, is_nn=False, is_bert=False):
    if is_bert:
        pred_output = model.predict(X_test)
        y_pred = np.argmax(pred_output.predictions, axis=1)
        y_proba = pred_output.predictions[:, 1]
    elif is_nn:
        y_proba = model.predict(X_test).flatten()
        y_pred = (y_proba > 0.5).astype(int)
```

```
    else:
        y_pred = model.predict(X_test)
        if hasattr(model, "predict_proba"):
            y_proba = model.predict_proba(X_test)[:, 1]
        elif hasattr(model, "decision_function"):
            y_proba = model.decision_function(X_test)
        else:
            y_proba = None

    return {
        'Model': name,
        'Accuracy': accuracy_score(y_test, y_pred),
        'Precision': precision_score(y_test, y_pred),
        'Recall': recall_score(y_test, y_pred),
        'F1 Score': f1_score(y_test, y_pred),
        'ROC-AUC': roc_auc_score(y_test, y_proba) if y_proba is not None else␣
↪'N/A'
    }
```

[ ]:
```
results = []

# Traditional ML models
results.append(evaluate_model(logreg, X_test_tfidf, y_test, "Logistic␣
↪Regression"))
results.append(evaluate_model(svm, X_test_tfidf, y_test, "SVM"))
results.append(evaluate_model(rf, X_test_tfidf, y_test, "Random Forest"))
results.append(evaluate_model(xgb, X_test_tfidf, y_test, "XGBoost"))

# Neural network models using padded sequences
results.append(evaluate_model(mlp_unfreeze, X_test_pad, y_test, "Simple MLP␣
↪(unfreeze)", is_nn=True))
results.append(evaluate_model(cnn_model, X_test_pad, y_test, "CNN", is_nn=True))
results.append(evaluate_model(textcnn, X_test_pad, y_test, "TextCNN",␣
↪is_nn=True))
results.append(evaluate_model(bilstm_model, X_test_pad, y_test, "BiLSTM",␣
↪is_nn=True))
results.append(evaluate_model(glove_model, X_test_pad, y_test, "GloVe MLP",␣
↪is_nn=True))

# Transformer-based model (BERT)
results.append(evaluate_model(trainer, val_tokenized, val_df['label'], "BERT",␣
↪is_bert=True))

# Convert to DataFrame
final_results = pd.DataFrame(results)
final_results.set_index("Model", inplace=True)
```

```
final_results.round(4)
```

```
57/57               0s 6ms/step
57/57               1s 8ms/step
57/57               1s 12ms/step
57/57               1s 8ms/step
57/57               0s 5ms/step
```

```
<IPython.core.display.HTML object>
```

[ ]:

Model	Accuracy	Precision	Recall	F1 Score	ROC-AUC
Logistic Regression	0.7787	0.7652	0.8031	0.7837	0.8691
SVM	0.7814	0.7768	0.7888	0.7828	0.8661
Random Forest	0.7128	0.6994	0.7448	0.7214	0.7958
XGBoost	0.7276	0.7026	0.7877	0.7427	0.8124
Simple MLP (unfreeze)	0.7694	0.7684	0.7701	0.7692	0.8429
CNN	0.7913	0.8065	0.7657	0.7856	0.8673
TextCNN	0.7963	0.8172	0.7624	0.7888	0.8694
BiLSTM	0.7930	0.7709	0.8328	0.8006	0.8689
GloVe MLP	0.7111	0.7005	0.7360	0.7178	0.7962
BERT	0.9186	0.9191	0.9212	0.9201	0.9661

The table above summarizes test-set performance across all models. Among all models, **BERT** clearly outperformed others with the highest accuracy (91.86%), F1 score (92.01%), and ROC-AUC (96.61%).

It was fine-tuned using the following hyperparameters:

- **Learning rate**: 2e-5
- **Batch size**: 16 (train), 32 (eval)
- **Epochs**: 3
- **Weight decay**: 0.01

2. **Statistical Significance**
    - Perform a significance test (e.g., McNemar's test) between your best two models.

[ ]:
```
bert_preds = trainer.predict(val_tokenized)
y_pred_bert = bert_preds.predictions.argmax(axis=-1)
```

```
<IPython.core.display.HTML object>
```

[ ]:
```
y_pred_bilstm = (bilstm_model.predict(X_val_pad) > 0.5).astype(int).flatten()
```

```
28/28               0s 9ms/step
```

[ ]:
```
from statsmodels.stats.contingency_tables import mcnemar
import numpy as np
```

```
b_only = np.logical_and(y_pred_bert == y_val, y_pred_bilstm != y_val)
c_only = np.logical_and(y_pred_bert != y_val, y_pred_bilstm == y_val)

table = [[0, np.sum(b_only)],
         [np.sum(c_only), 0]]

result = mcnemar(table, exact=True)

print("McNemar's test statistic:", result.statistic)
print("p-value:", result.pvalue)
if result.pvalue < 0.05:
    print("  The difference is statistically significant.")
else:
    print("  The difference is not statistically significant.")
```

```
McNemar's test statistic: 31.0
p-value: 2.7836498022112537e-23
  The difference is statistically significant.
```

We conducted McNemar's test between the best two models: **BERT** and **Bi-LSTM**. It tests whether the performance difference between two models on the same test set is statistically significant, not due to random variation.

The test yielded a **statistic of 31.0** and a **p-value of 2.78e-23**, indicating that the difference in their classification outcomes is statistically significant ($p < 0.05$).

This supports our conclusion that **BERT outperforms Bi-LSTM** on this task with a high level of confidence.

3. **Error Analysis**
   - Identify at least 20 examples your best model misclassified.
   - For a sample of 5, provide the raw text, predicted vs. true label, and a short discussion of each error—what linguistic artifact might have confused the model?

[ ]:
```
import numpy as np

misclassified_idx = np.where(y_pred_bert != y_val)[0]
print(f"Total misclassified examples: {len(misclassified_idx)}")
```

```
Total misclassified examples: 71
```

[ ]:
```
texts = val_df['sentence']
```

[ ]:
```
misclassified_texts = texts.iloc[misclassified_idx]
misclassified_true = y_val.iloc[misclassified_idx]
misclassified_pred = y_pred_bert[misclassified_idx]
```

```
[ ]: misclassified_samples = pd.DataFrame({
         "Sentence": val_df['sentence'].iloc[misclassified_idx].values[:20],
         "True Label": y_val.iloc[misclassified_idx].values[:20],
         "Predicted Label": y_pred_bert[misclassified_idx][:20]
     })

     import IPython
     IPython.display.display(misclassified_samples)
```

```
                                             Sentence  True Label  \
0   american chai encourages rueful laughter at st…           0
1             directed in a paint-by-numbers manner .           0
2   the longer the movie goes , the worse it gets …           0
3   if steven soderbergh 's ` solaris ' is a failu…           1
4         it has all the excitement of eating oatmeal .       0
5   something akin to a japanese alice through the…           1
6                           this movie is maddening .           0
7   we root for -lrb- clara and paul -rrb- , even …           1
8   you wo n't like roger , but you will quickly r…           0
9   a full world has been presented onscreen , not…           1
10  sit through this one , and you wo n't need a m…           0
11  if director michael dowse only superficially u…           0
12  moretti 's compelling anatomy of grief and the…           0
13  while there 's something intrinsically funny a…           1
14  sam mendes has become valedictorian at the sch…           0
15  it takes a certain kind of horror movie to qua…           0
16  it 's somewhat clumsy and too lethargically pa…           0
17  movie fans , get ready to take off … the oth…           0
18  you 'll gasp appalled and laugh outraged and p…           1
19  this flick is about as cool and crowd-pleasing…           1

    Predicted Label
0                 1
1                 1
2                 1
3                 0
4                 1
5                 0
6                 1
7                 0
8                 1
9                 0
10                1
11                1
12                1
13                0
14                1
```

```
15                1
16                1
17                1
18                0
19                0
```

[ ]:
```
for i in range(5):
    print(f"\nExample {i+1}")
    print("Text:", misclassified_texts.iloc[i])
    print("True Label:", misclassified_true.iloc[i])
    print("Predicted Label:", misclassified_pred[i])
```

```
Example 1
Text: american chai encourages rueful laughter at stereotypes only an indian-
american would recognize .
True Label: 0
Predicted Label: 1

Example 2
Text: directed in a paint-by-numbers manner .
True Label: 0
Predicted Label: 1

Example 3
Text: the longer the movie goes , the worse it gets , but it 's actually pretty
good in the first few minutes .
True Label: 0
Predicted Label: 1

Example 4
Text: if steven soderbergh 's ` solaris ' is a failure it is a glorious failure
.
True Label: 1
Predicted Label: 0

Example 5
Text: it has all the excitement of eating oatmeal .
True Label: 0
Predicted Label: 1
```

We conducted error analysis on the **BERT** model, our best-performing classifier. By comparing predicted and true labels on the validation set, we identified 71 misclassified examples and selected 5 for closer inspection.

- **Example 1**: "*american chai encourages rueful laughter at stereotypes only an indian-american would recognize.*"

→ **Predicted: Positive | True: Negative**

The phrase “rueful laughter” implies discomfort or critique, but the model may have focused on surface-level positivity like “encourages” and “laughter.”

- **Example 2**: “*directed in a paint-by-numbers manner.*”

→ Predicted: Positive | True: Negative

A clear critique phrased in a subtle, metaphorical way; the model might not recognize “paint-by-numbers” as negative without explicit sentiment words.

- **Example 3**: “*the longer the movie goes, the worse it gets, but it’s actually pretty good in the first few minutes.*”

→ Predicted: Positive | True: Negative

The positive clause at the end may have overridden the overall negative sentiment in the model’s decision.

- **Example 4**: “*if steven soderbergh’s ‘solaris’ is a failure it is a glorious failure.*”

→ Predicted: Negative | True: Positive

The oxymoronic phrasing “glorious failure” requires interpretation of tone and irony, which the model may misinterpret as criticism.

- **Example 5**: “*it has all the excitement of eating oatmeal.*”

→ Predicted: Positive | True: Negative

A sarcastic comparison using bland imagery, which may be interpreted as neutral or descriptive without explicit negative markers.

1.19 Part 8 – Optional Challenge Extensions

1. **Data Augmentation**
    - Implement data augmentation for text (back-translation, synonym swapping) and measure its impact.

[ ]:
```
# Load data (example)
import pandas as pd



# IMPORT DATA
!git clone https://github.com/YJiangcm/SST-2-sentiment-analysis.git

# Assuming the dataset is CSV for illustration
df = pd.read_csv('sst_data.csv')
df.head()
```

2. **Sentiment Lexicon**

- Integrate a sentiment lexicon feature (e.g., VADER scores) into your models and assess whether it improves predictions.

3. **Model Deployment**

- Deploy your best model as a simple REST API using Flask or FastAPI and demo it on a handful of user-submitted reviews.

---

2 Reflecting

Answer the following inference questions:

2.0.1 Part 1 – Data Ingestion & Preprocessing

1. **Data Loading**
    - How do you ensure that your dataset is properly split into training, validation, and test sets, and why is class balance important during data splitting?

In this project, the SST-2 dataset was pre-split into training, validation, and test sets by the dataset provider. We confirmed the integrity and balance of these splits through exploratory data analysis.

Maintaining class balance is important in binary classification tasks like sentiment analysis, as an imbalanced dataset can cause the model to favor the majority class, resulting in skewed performance and poor generalization on minority examples.

2. **Text Cleaning & Tokenization**
    - What is the role of tokenization in text preprocessing, and how does it impact the model's performance?

Tokenization breaks down raw text into smaller units (words or characters) that can be processed by models.

It directly impacts feature extraction—e.g., TF-IDF vectorization or sequence embeddings—and thus model performance.

Poor tokenization may lead to fragmented or inconsistent input, while a well-designed tokenizer (with rare word pruning and consistent formatting) improves both model generalization and training efficiency.

2.0.2 Part 2 – Exploratory Data Analysis (EDA)

1. **Class Distribution**
    - How does the class distribution (positive vs negative reviews) impact the model's performance, and what strategies can be used if the dataset is imbalanced?

The class distribution in our SST-2 dataset was nearly balanced. This balance was confirmed through bar plots in our EDA.

A balanced distribution ensured that our models didn't favor one class over the other, which is crucial for binary sentiment classification.

If the dataset had been imbalanced, we would have considered techniques like class weighting or resampling to address bias.

2. **Text Characteristics**
    - What insights can be gained from visualizing word clouds for each sentiment class, and how can it improve feature engineering?

Our word clouds and frequency bar charts showed that words like "fun", "best", and "entertaining" dominated positive reviews, while "bad", "dull", and "worst" were common in negative reviews. These insights confirmed that certain sentiment-rich tokens were strongly correlated with label classes.

This informed our decision to use TF-IDF for traditional models and helped justify feature importance results from Random Forest and XGBoost, where many of these same words ranked highly—demonstrating the consistency between our exploratory analysis and model behavior.

2.0.3 Part 3 – Baseline Traditional Models

1. **Logistic Regression & SVM**
    - Why do you use cross-validation when training models like logistic regression or SVM, and how does it help prevent overfitting?

Cross-validation helps prevent overfitting by ensuring that the model is not just learning patterns specific to one training subset. Instead, it trains and validates on multiple different splits of the data.

In our case, 5-fold CV revealed stable accuracy across folds (e.g., Logistic Regression: 0.776), which suggests that the models generalize well and are not overly tuned to one portion of the data.

2. **Random Forest & Gradient Boosting**
    - What role does feature importance play in interpreting Random Forest or XGBoost models?

Feature importance in Random Forest and XGBoost helped us understand which tokens (e.g., "bad", "too", "best") were driving predictions.

This interpretability allows us to detect if models are relying too much on spurious or overly frequent features, which could indicate overfitting, and refine our preprocessing or feature selection accordingly.

2.0.4 Part 4 – Neural Network Models

1. **Simple Feed-Forward**
    - Why is embedding freezing used when training neural networks on pre-trained embeddings, and how does it affect model performance?

Freezing the embedding layer prevents the pre-trained vectors from being updated during training, preserving semantic information learned from large corpora.

In our experiments, the frozen model performed poorly (val accuracy 51%), while the unfrozen version improved dramatically (75%), suggesting that fine-tuning embeddings helps adapt them to sentiment nuances.

2. **Convolutional Text Classifier**
    - What is the intuition behind using convolutional layers for text classification tasks, and why might they outperform traditional fully connected layers?

Convolutional layers capture local patterns in sequences—like bigrams or trigrams—by sliding filters over word embeddings. This helps detect sentiment-bearing phrases regardless of position.

Our CNN outperformed the simple MLP, showing that learning spatial hierarchies in text is more effective than treating the input as a flat vector.

2.0.5 Part 5 – Transfer Learning & Advanced Architectures

1. **Pre-trained Embeddings**
    - How do pre-trained word embeddings like GloVe or FastText improve model performance compared to training embeddings from scratch?

Pre-trained embeddings like GloVe bring rich semantic knowledge from large external corpora.

Compared to training embeddings from scratch, our GloVe-based MLP model started with stronger initial representations and converged faster. However, its final performance (~71% accuracy) was still lower than fine-tuned deep models, suggesting that while helpful, pre-trained word vectors alone may not fully capture task-specific context.

2. **Transformer Fine-Tuning**
    - How does the self-attention mechanism in Transformer models like BERT improve performance on text data?

Transformers like BERT use self-attention to weigh the importance of each word in relation to others in a sentence, capturing context more precisely than fixed windows or sequential models.

This helped BERT achieve the best performance in our project (92% accuracy, 0.92 F1).

2.0.6 Part 6 – Hyperparameter Optimization

1. **Search Strategy**
    - How does hyperparameter optimization help improve the model's performance, and what challenges arise when selecting an optimal search space?

Hyperparameter optimization helps improve model performance by systematically testing different combinations to find the configuration that performs best on validation data.

In our project, we used Keras Tuner to explore `units` and `dropout_rate` in a Simple MLP.

A challenge was defining a meaningful search space—too narrow might miss better configurations, while too wide increases computation time.

2. **Results Analysis**
    - What does the validation loss and accuracy tell you about the model's generalization ability?

Validation loss and accuracy provide signals about how well a model generalizes to unseen data.

In our trials, models with high validation accuracy and low validation loss (e.g., Trial 2) indicated strong generalization.

However, Trial 3 showed both low accuracy and high loss, suggesting that the model underfit and failed to learn meaningful patterns. Such anomalies highlight the importance of monitoring both metrics to avoid misleading conclusions from accuracy alone.

2.0.7 Part 7 – Final Comparison & Error Analysis

1. **Consolidated Results**
    - How do you compare models with different architectures (e.g., logistic regression vs. BERT) to select the best model for deployment?

To compare models with different architectures fairly, we use a consistent evaluation framework on the same test set and report metrics like accuracy, precision, recall, F1 score, and ROC-AUC.

In our case, BERT outperformed all other models across metrics, making it the most suitable choice for deployment.

2. **Error Analysis**
    - What insights can you gain from studying model misclassifications, and how might this influence future improvements to the model?

Studying misclassifications reveals common patterns that confuse the model—such as sarcasm, mixed sentiment, or domain-specific expressions.

For example, phrases like "glorious failure" or "pretty good in the first few minutes" were misclassified by BERT due to nuanced or contradictory sentiment.

These findings suggest that incorporating sentiment-aware components or fine-tuning with more nuanced data could further improve the model.

2.0.8 Part 8 – Optional Challenge Extensions

1. **Data Augmentation**
    - How does back-translation or synonym swapping as text augmentation improve model generalization?
2. **Sentiment Lexicon**
    - How might integrating sentiment lexicons like VADER improve the sentiment classification model, and what are the challenges of using lexicon-based approaches alongside machine learning models?