CS4395: Final Project Report

https://github.com/adibarra/hlt

Group 14	Alec Ibarra adi220000	Syed Kabir snk210004	Ashlee Kang ajk200003	Annette Llanas ajl200006

1. Introduction

Sentiment analysis is a key task in Natural Language Processing (NLP) that focuses on identifying and interpreting the emotional tone within text. With the rapid growth of user-generated content on platforms like Twitter, YouTube, and Amazon, efficient sentiment classification has become increasingly important.

In this project, we compare a range of sentiment analysis models across diverse datasets to evaluate how well they perform under varying contexts and text lengths. Our models span from traditional approaches like Logistic Regression with TF-IDF to deep learning methods such as LSTM and CNN, and modern transformer-based models like TinyBERT.

Our goal is to assess each model's efficiency by analyzing the trade-offs between predictive performance, computational time, and model complexity. This helps identify the most practical models for real-world sentiment analysis tasks.

2. Data

We utilized three publicly available sentiment classification datasets for evaluating the models. Below is a summary of the datasets, including key statistics and sentiment distribution:

Dataset	**Total Entries**	**Sentiment**	**Source**
Amazon Reviews	49,998	Positive: 33.33%, Negative: 33.33%, Neutral: 33.33%	Xiang Zhang's Google Drive
Tweets (Airline Sentiment)	14,640	Positive: 16.14%, Negative: 62.69%, Neutral: 21.17%	Kaggle - Crowdflower
YouTube Comments	18,408	Positive: 62.10%, Negative: 12.70%, Neutral: 25.20%	Kaggle - atifaliak

These datasets cover a range of sentiment expressions, from detailed product reviews to short, informal social media posts.

- The Amazon Reviews dataset features long and descriptive text with a balanced sentiment distribution, making it a suitable test case for more complex models.
- The Tweets (Airline Sentiment) dataset presents challenges like informal language, slang, and a negative sentiment bias, useful for testing models on short, context-heavy texts.
- The YouTube Comments dataset is conversational and skewed positive, helping evaluate how models handle diverse, informal language and varied topics.

Each dataset provides a unique set of challenges that help in assessing the versatility and robustness of different sentiment analysis models across varying contexts and languages.

3. Methodology

3.1 Model Architectures

We evaluated four different model architectures for sentiment analysis, ranging from traditional machine learning to advanced deep learning models:

- **Logistic Regression with TF-IDF:** A classical approach that uses word frequency statistics (TF-IDF) to represent text and a logistic regression classifier to predict sentiment. This model serves as a baseline for performance comparison with more complex models.
- **LSTM with GloVe Embeddings:** A deep learning model based on Long Short-Term Memory (LSTM) networks, which are capable of capturing long-term dependencies in sequences of text. We utilized pretrained GloVe word embeddings to enhance the model's ability to understand word meanings in context.
- **Text CNN:** A convolutional neural network (CNN) designed for text. The model applies convolutional filters to the input text, extracting local n-gram features, which are then aggregated using max-pooling. This approach is particularly effective for capturing spatial hierarchies in text data.
- **TinyBERT:** A distilled transformer-based model (BERT variant) that is pre-trained on a large corpus of text and fine-tuned for specific tasks like sentiment analysis. TinyBERT is designed to provide deep contextual understanding of text with fewer parameters, offering a balance between performance and computational efficiency.

3.2 Evaluation Metrics

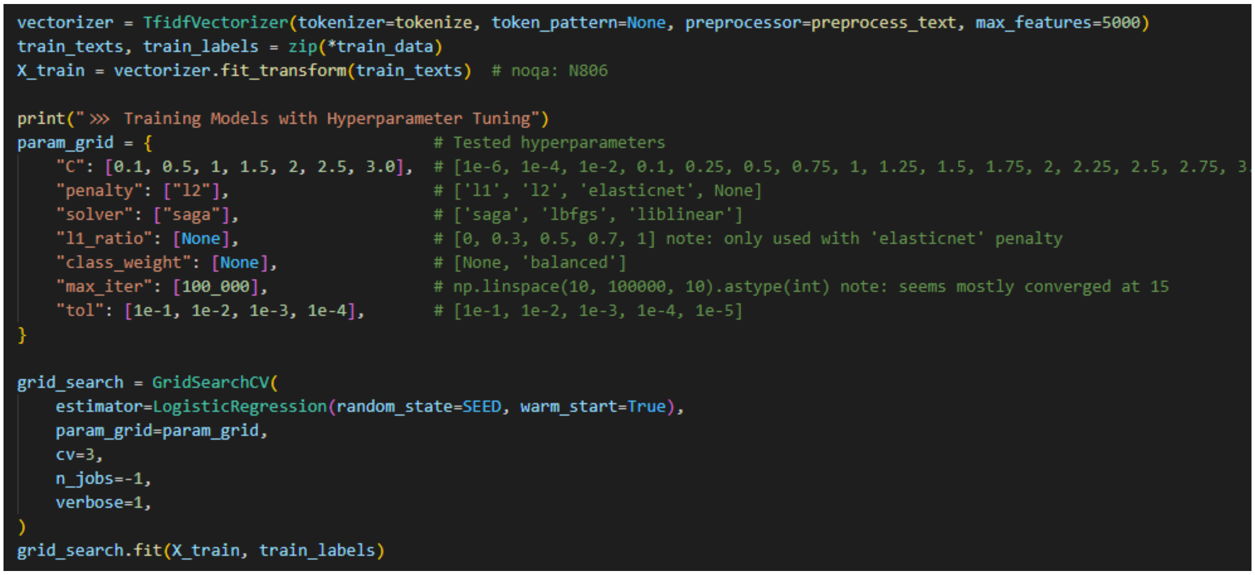
For model evaluation, we primarily used the F1-score, which provides a balance between precision and recall. This metric is particularly important in sentiment analysis where class distribution may be imbalanced, ensuring that the model's performance is assessed across both false positives and false negatives.

4. Implementations

4.1 Logistic Regression with TF-IDF

The Logistic Regression model was implemented using TF-IDF (Term Frequency-Inverse Document Frequency) for feature extraction [1]. The text data was transformed into numerical features using TfidfVectorizer from sklearn, which leverages the frequency of words across documents to capture the most important terms. The vectorizer uses a custom tokenizer and preprocessor, with a maximum of 5000 features considered for training the model.

For training, a Logistic Regression model was used from the sklearn.linear_model package. Hyperparameter tuning was performed through Grid Search with Cross-Validation, where various hyperparameters, such as the regularization parameter C, penalty type, solver, and tolerance levels, were tested. The GridSearchCV function automatically searched for the best combination of hyperparameters based on the training data, helping to optimize the model's performance. Once the best hyperparameters were identified, the model was trained using the optimal configuration.



```
vectorizer = TfidfVectorizer(tokenizer=tokenize, token_pattern=None, preprocessor=preprocess_text, max_features=5000)
train_texts, train_labels = zip(*train_data)
X_train = vectorizer.fit_transform(train_texts)  # noqa: N806

print(" >>> Training Models with Hyperparameter Tuning")
param_grid = {                              # Tested hyperparameters
    "C": [0.1, 0.5, 1, 1.5, 2, 2.5, 3.0],  # [1e-6, 1e-4, 1e-2, 0.1, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2, 2.25, 2.5, 2.75, 3
    "penalty": ["l2"],                      # ['l1', 'l2', 'elasticnet', None]
    "solver": ["saga"],                     # ['saga', 'lbfgs', 'liblinear']
    "l1_ratio": [None],                     # [0, 0.3, 0.5, 0.7, 1] note: only used with 'elasticnet' penalty
    "class_weight": [None],                 # [None, 'balanced']
    "max_iter": [100_000],                  # np.linspace(10, 100000, 10).astype(int) note: seems mostly converged at 15
    "tol": [1e-1, 1e-2, 1e-3, 1e-4],        # [1e-1, 1e-2, 1e-3, 1e-4, 1e-5]
}

grid_search = GridSearchCV(
    estimator=LogisticRegression(random_state=SEED, warm_start=True),
    param_grid=param_grid,
    cv=3,
    n_jobs=-1,
    verbose=1,
)
grid_search.fit(X_train, train_labels)
```

Figure 1. TF-IDF LR Vectorization and Grid Search

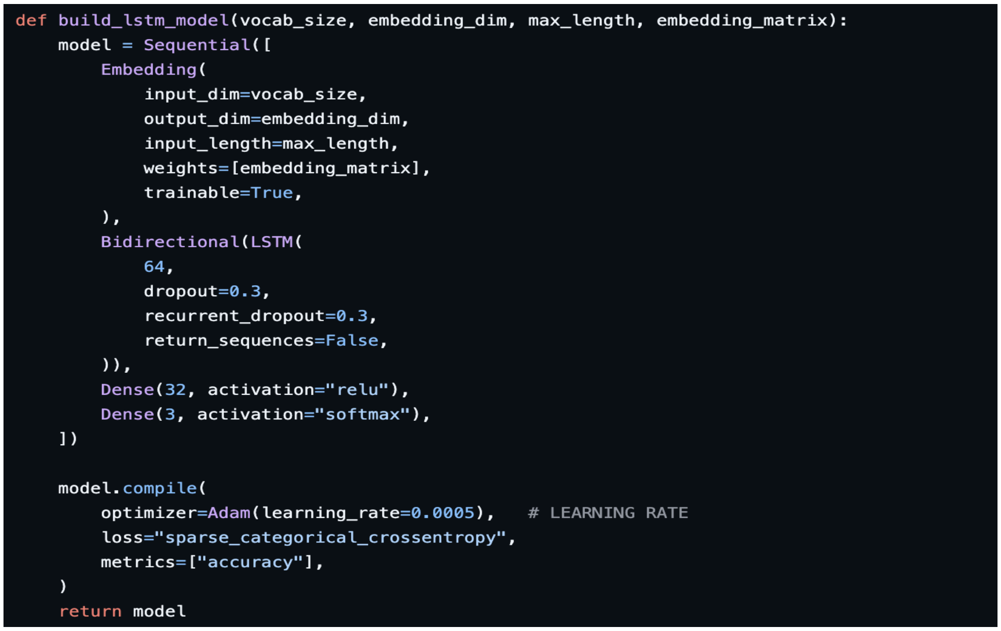
After training, the model was evaluated on the test set using standard classification metrics, including precision, recall, and F1-score, which were computed using the classification_report function from sklearn.metrics. The best model and its hyperparameters were then returned, along with a summary of the evaluation metrics, providing a comprehensive view of the model's performance.

4.2 LSTM with GloVe Embeddings

The LSTM model was implemented on top of 50-dimensional GloVe pretrained embeddings so that raw text was mapped to dense feature vectors. Each dataset was tokenized with a Keras Tokenizer using a 5000-word vocabulary and an Out of Vocabulary or OOV token. Sequences were then padded or shortened to length of 100. An embedding matrix of shape **vocab_size*50** was assembled by loading each word's pretrained GloVe vector when available and initializing others randomly with a fixed seed.

During training, a bidirectional LSTM with 64 units in each direction was used along with a dropout of 30 percent on both inputs and recurrent connections to reduce overfitting. A 32-unit dense layer with ReLU activation was added on top of the LSTM, and was then followed by a three-unit softmax output. The model was compiled with the Adam optimizer with a learning rate of 0.0005 and a sparse categorical_crossentropy loss. To address class imbalance, balanced class weights were computed from the training labels and passed into model.fit. Training ran for up to 10 epochs with a batch size of 128 along with two callbacks:

- EarlyStopping (patience = 3, restore best weights)
- ReduceLROnPlateau (factor = 0.5, patience =2)



```
def build_lstm_model(vocab_size, embedding_dim, max_length, embedding_matrix):
    model = Sequential([
        Embedding(
            input_dim=vocab_size,
            output_dim=embedding_dim,
            input_length=max_length,
            weights=[embedding_matrix],
            trainable=True,
        ),
        Bidirectional(LSTM(
            64,
            dropout=0.3,
            recurrent_dropout=0.3,
            return_sequences=False,
        )),
        Dense(32, activation="relu"),
        Dense(3, activation="softmax"),
    ])

    model.compile(
        optimizer=Adam(learning_rate=0.0005),   # LEARNING RATE
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model
```

Figure 2. LSTM Model Setup

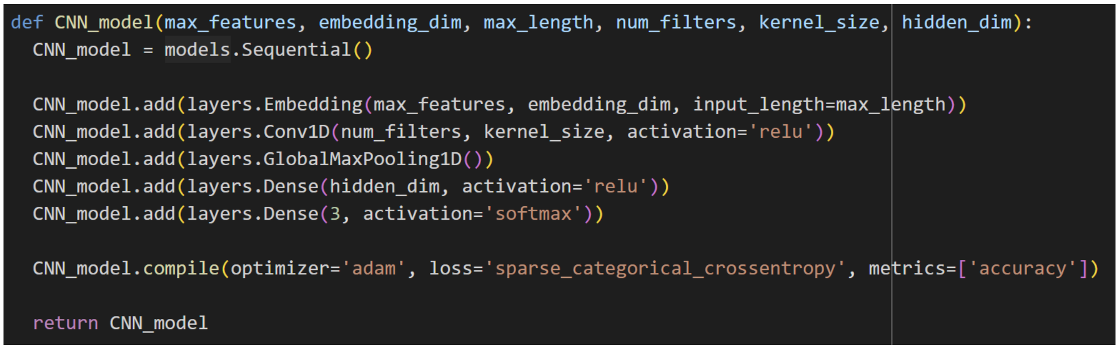
After training, we generated predictions on each test set by taking the arg max over the softmax outputs, then computed precision, recall, and F1-score through scikit-learn's classification_report. All key hyperparameters such as vocab size, embedding dimensions, LSTM units, optimizer type/learning rate, epoch count, batch size, and callback settings were logged for reproducibility. Finally, we summarized average F1 and total runtime across all three datasets.

4.3 Text CNN

The CNN model was implemented using Tensoflow's Sequential model following an architecture similar to the approach proposed by Kim(2014) except we use Conv1D instead of Kim's Conv2D approach. The text was preprocessed and tokenized using Keras' Tokenizer with a vocabulary size of 1000 words and then the sequences were padded to max length of 200 tokens. The model was implemented in Google Colab which uses its own GPUs so training times can vary highly.

For training, the model first converted words into 100 dimensional numerical vectors in the embedding layer where similar words would have similar representations. Next, a 1D convolutional layer with 128 filters and a 3x3 kernel which will capture local n-gram patterns or context features within the text. The ReLU activation function helped the model focus on the strongest patterns by introducing non-linearity. Then the Global Max Pooling layer only selects the most important signal from each filter's results to capture the most relevant sentiment information.

After extracting the features, the dense layer with 128 dimensions will receive the 128 features from the pooling layer and learns which combination of patterns correlates to which specific sentiment. The final layer uses softmax with the 3 sentiment classes to calculate the probability of each. The model uses Adam optimizer and sparse categorical cross entropy. Training was conducted for 10 epochs with batch size of 16 and used validation split data to ensure it wasn't overfitting.



```python
def CNN_model(max_features, embedding_dim, max_length, num_filters, kernel_size, hidden_dim):
  CNN_model = models.Sequential()

  CNN_model.add(layers.Embedding(max_features, embedding_dim, input_length=max_length))
  CNN_model.add(layers.Conv1D(num_filters, kernel_size, activation='relu'))
  CNN_model.add(layers.GlobalMaxPooling1D())
  CNN_model.add(layers.Dense(hidden_dim, activation='relu'))
  CNN_model.add(layers.Dense(3, activation='softmax'))

  CNN_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

  return CNN_model
```

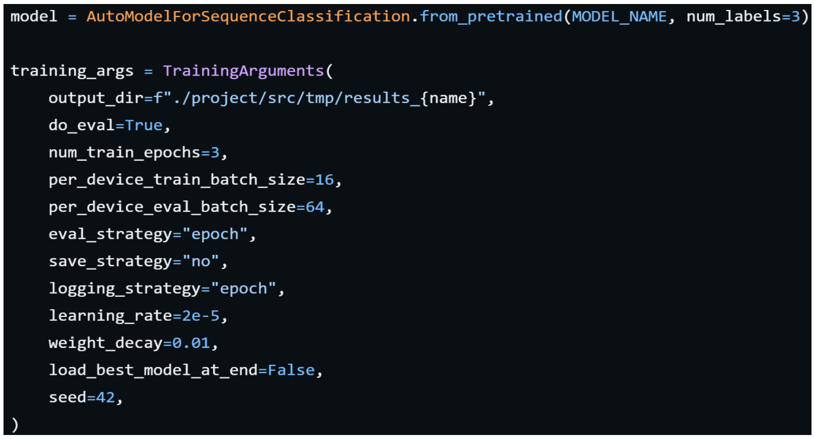
Figure 3. CNN Model Setup

After training, we tested the model's predictions using a testset split and used sklearn's classification_report to get the precision, recall, and F1-score of each of the 3 sentiment classes.

4.4 TinyBERT

The Hugging Face platform offers extensive natural language processing tools, including the transformers library, from which the TinyBERT model (huawei-noah/TinyBERT_General 4L_312D) was used. TinyBERT was created using transformer-layer distillation from the original BERT-base as the teacher model and a large scale text corpus as the learning data [2]. The resulting model is a general-purpose TinyBERT intended to be fine-tuned to a specific task.

Each dataset is read into a pandas dataframe, preprocessed, and tokenized using the AutoTokenizer function provided with the TinyBERT model. For fine-tuning and training, the AutoModelForSequenceClassification class loads the TinyBERT model with a classification head with three output labels for the sentiment classes. Training is configured with the TrainingArguments class, with significant specifications including batch sizes of 16 and 64 for training and evaluation, a learning rate of 2e-5 with a weight decay 0.01 for regularization, a training duration of three epochs, evaluation performed at the end of each epoch, and a fixed random seed for reproducibility. The script loops through each dataset, conducting training, testing, and validation independently for each one.



```python
model = AutoModelForSequenceClassification.from_pretrained(MODEL_NAME, num_labels=3)

training_args = TrainingArguments(
    output_dir=f"./project/src/tmp/results_{name}",
    do_eval=True,
    num_train_epochs=3,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=64,
    eval_strategy="epoch",
    save_strategy="no",
    logging_strategy="epoch",
    learning_rate=2e-5,
    weight_decay=0.01,
    load_best_model_at_end=False,
    seed=42,
)
```

Figure 4. TinyBERT TrainingArguments Setup

Precision, recall, and F-1 score are computed using the precision_recall_fscore_support function as well as the classification_report function both from sklearn.metrics.

4.5 Packages and Libraries

The following contains the packages/libraries used for each model:

- Logistic Regression: torch, sklearn
- LSTM + GloVe: numpy, sklearn, tensorflow, colab
- Text CNN: numpy, sklearn, tensorflow, pandas, colab
- TinyBERT: numpy, sklearn, pandas, datasets, transformers

5. Experiments and Results

We evaluated all four models using precision, recall, and F1-score across the three sentiment datasets described earlier. The primary metric reported below is the weighted-averaged F1-score, which accounts for performance across all three sentiment classes (positive, neutral, and negative).

5.1 Results

The table below summarizes the average F1-scores for each model on each dataset, along with their overall average:

Model	**Amazon F1-Score**	**Tweets F1-Score**	**Youtube F1-Score**	**Combined F1-Score**
LR	0.64	0.79	0.75	0.7267
LSTM + GloVe	0.62	0.80	0.74	0.7200
Text CNN	0.59	0.70	0.60	0.6300
TinyBERT	0.69	0.84	0.82	0.7833

These results highlight the general trend that transformer-based models outperform simpler or more traditional architectures, especially on datasets with nuanced or context-rich text.

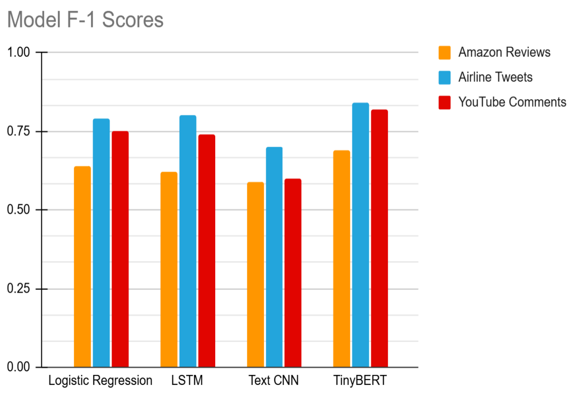


Figure 5. Comparison of F1-Scores Across Datasets

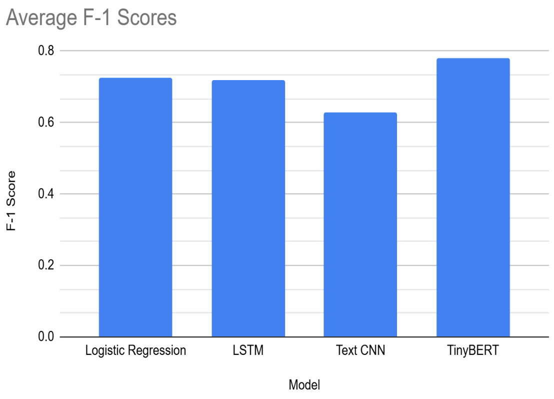


Figure 6. Comparison of Average F1 Scores by Model

5.2 Error Analysis

We observed several consistent error patterns across the models:

- The Amazon Reviews dataset posed more difficulty overall, likely due to longer input lengths, subtler tone, and mixed sentiments within the same review.
- The Text CNN model performed relatively poorly on neutral sentiment, often misclassifying context-dependent or ambiguous text.
- The LSTM model struggled with short and informal text (e.g., tweets), where lack of context or domain-specific slang made classification harder.
- Traditional models (Logistic Regression, LSTM, CNN) were more susceptible to vocabulary limitations and out-of-distribution expressions (especially slang or abbreviations), which were common in tweets and YouTube comments.

5.2 Speed Analysis

Training time varied significantly across models, reflecting a trade-off between model complexity and computational efficiency:

- Logistic Regression with TF-IDF was the fastest, with an average training time of only 40ms, making it ideal for real-time or low-resource settings.
- Text CNN trained in approximately 53.6 seconds using Colab's T4 GPU, offering a balance between speed and deep feature extraction.
- TinyBERT, though highly accurate, required 571.5 seconds on average—still efficient for a transformer, but demanding relative to classical methods.
- LSTM with Pretrained Embeddings was the slowest, averaging 725.7 seconds, due to sequential processing and overhead from embedding initialization.

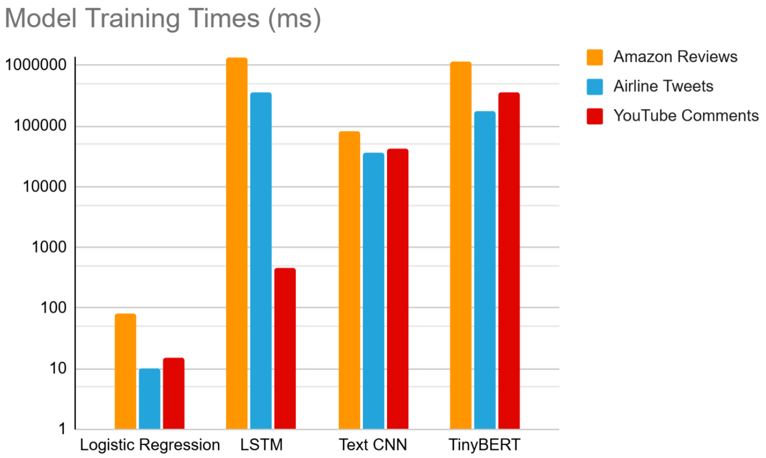


Figure 7. Training Time by Model Across Datasets

These results underscore a common trend in NLP: higher accuracy often comes at the expense of computational cost, whereas simpler models offer faster predictions but may struggle with complex linguistic patterns.

6. Conclusions

Our experiments demonstrate that transformer-based models like TinyBERT consistently achieve the highest F1-scores across diverse sentiment analysis datasets, thanks to their contextual understanding and deep language representations. However, simpler models such as Logistic Regression with TF-IDF offer strong performance at a fraction of the computational cost, making them practical for real-time applications or resource-constrained environments.

Each model excels in different scenarios:

- TinyBERT is best suited for high-accuracy requirements where training time and hardware resources are less restrictive.
- Logistic Regression provides a fast and lightweight alternative with surprisingly robust results.
- LSTM and Text CNN offer intermediate options with trade-offs in speed, context handling, and performance.

There are several directions for future work. One possibility is to explore ensemble models that combine the strengths of different approaches. Another is to fine-tune transformer models using data from specific domains to improve their handling of subtle or informal language. We could also look into using better word embeddings or adapting the models for other languages. Finally, doing more experiments with different hyperparameters could help improve both accuracy and efficiency.

Ultimately, the best model depends on the specific needs of the task, especially when balancing accuracy, speed, and available computing resources.

7. References

[1] G. Salton and C. Buckley, ‘Term-weighting approaches in automatic text retrieval’, Information Processing & Management, vol. 24, no. 5, pp. 513–523, 1988. doi:10.1016/0306-4573(88)90021-0

[2] X. Jiao et al., “TinyBERT: Distilling Bert for Natural language understanding,” Findings of the Association for Computational Linguistics: EMNLP 2020, Sep. 2019. doi:10.18653/v1/2020.findings-emnlp.372

[3] Y. Kim, “Convolutional Neural Networks for Sentence Classification,” Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 1746-1751. https://doi.org/10.3115/v1/D14-1181