Sentiment Analysis for Persian Texts

Mehraneh Moghtadaeefar Mohammadreza Seidgar

Sentiment analysis plays a crucial role in understanding people's opinions and emotions expressed in texts. In this study, we aimed to perform sentiment analysis on Persian texts using various natural language processing (NLP) techniques and deep learning models. The dataset used in this study is comments people submitted on Snappfood app about food and it was preprocessed to handle linguistic nuances in Persian, and several models were trained and evaluated. We compared the performance of Logistic Regression, LSTM, a Simple Neural Network, and Bidirectional LSTM models. Additionally, various preprocessing techniques, including stemming, lemmatization, and stop-word removal, were explored, and two different word embeddings, Word2Vec and FastText, were applied.

Understanding sentiment is valuable in various domains, including social media monitoring, customer feedback analysis, and market research. In this study, we focused on sentiment analysis for Persian texts, considering the unique linguistic characteristics of the Persian language.

Dataset Preprocessing

The dataset underwent extensive preprocessing to enhance the performance of sentiment analysis models. The following steps were taken:

NLP Preprocessing

The data is loaded from a CSV file (`train.csv`) using Pandas. The data has four columns: 'Unnamed: 0', 'comment', 'label', and 'label_id'.

Data Overview:

`raw_data.info()` provides information about the DataFrame, indicating 56,700 entries with columns 'Unnamed: 0', 'comment', 'label', and 'label_id'. The 'label' column contains two unique values: 'SAD' and 'HAPPY', and the 'label_id' column contains corresponding numerical labels 1 and 0.

Label Preprocessing:

- The 'Unnamed: 0' column is dropped as it seems to be an unnecessary index column.
- The 'comment' column is extracted and converted to a Pandas Series called `comments`.

Detecting Non-Persian Comments:

- The `Normalizer` from Hazm library is used to normalize each comment.
- A function `is_non_persian` is defined to check if a comment contains non-Persian (ASCII) characters.
- A new column 'is_non_persian' is created in the DataFrame to store the result of the function applied to each comment.
- Non-Persian comments are extracted into a new DataFrame called `non_persian_comments`.

Handling Finglish (Latin Script) Comments:

- Indices of Finglish comments are stored in `finglish_comment_indices`.
- A subset DataFrame (`specific_comments_df`) is created with only the Finglish comments.
- The `f2p` function is applied to convert Finglish comments to Persian script, and the original DataFrame (`raw_data`) is updated accordingly.

However the transition was not helpful because some of the comments became meaningless in Farsi. They were a total of 175 and were not useful. Because they were a few so they were removed from the whole dataset.

Replacing Persian and Arabic Characters:

Functions (`convert_fa_numbers` and `convert_ar_characters`) are defined to replace Persian and Arabic characters in the text.

Numbers, Arabic letters, emojis, URLs, and other non-textual elements were replaced with specific tokens

Main Preprocessing Function:

The `preprocess` function is defined to perform various preprocessing steps, including URL extraction, emoji removal, number conversion, character conversion, smiley removal, lowercase conversion, and punctuation removal.

The `preprocess` function is applied to the 'comment' column in the original DataFrame.

Loading Preprocessed Data:

The preprocessed DataFrame is saved to a CSV file (`reviews_cleaned_v1.csv`) for future use(because the transition took so much time) and loaded back into the `clean_df` DataFrame.

Null values are checked, and there are no missing values.

Vectorization and Embedding:

Before getting into model training and evaluation, we need to vectorize the comments. This is also a part of preprocessing because the model won't understand the words and letters and we must convert it to numbers.

We use TF-IDF vectorization.

TF-IDF Vectorization

- The TF-IDF vectorizer is utilized to convert the cleaned text data into TF-IDF representations.
- The data is split into training and testing sets.
- The vectorizer is fitted on the training data, and the resulting TF-IDF matrices (`X_tfidf_train` and `X_tfidf_test`) are obtained.
- The TF-IDF matrices and labels are converted to PyTorch tensors (`X_tfidf_train_tensor`, `y_train_tensor`, `X_tfidf_test_tensor`, `y_test_tensor`).
- PyTorch DataLoaders (`train_loader` and `test_loader`) are created to efficiently load batches of data during training and testing.

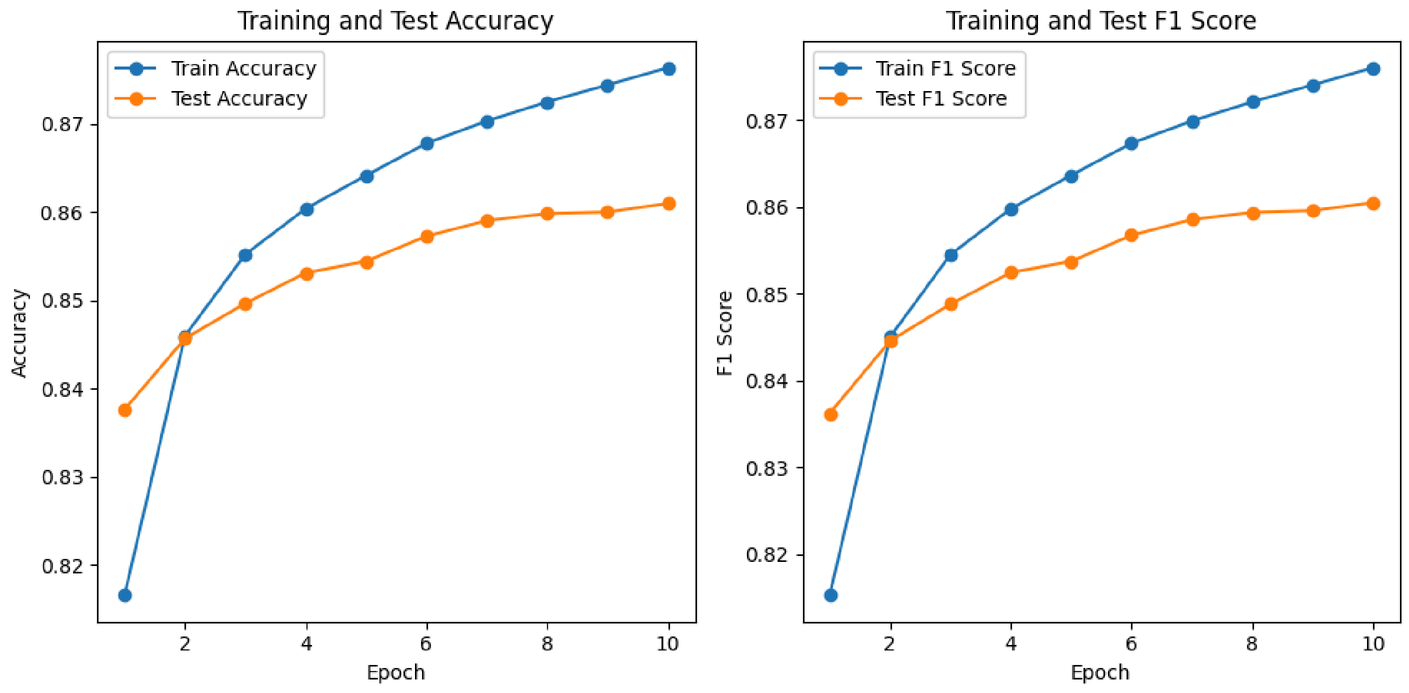
This comprehensive preprocessing pipeline ensures that the text data is cleaned, normalized, and converted into a format suitable for training machine learning models. The use of PyTorch tensors and DataLoaders facilitates the integration of the data into a PyTorch-based model.

Model Training and Evaluation

1. Logistic Regression

A Logistic Regression model was implemented using TF-IDF vectorization. The model was trained for 10 epochs, the Adam optimizer was used and learning rate = 0.001, for criterion the loss was cross entropy and the following results were obtained:

Train Accuracy: 87.63%, Train F1 Score: 0.8760 | Test Accuracy: 86.09%, Test F1 Score:

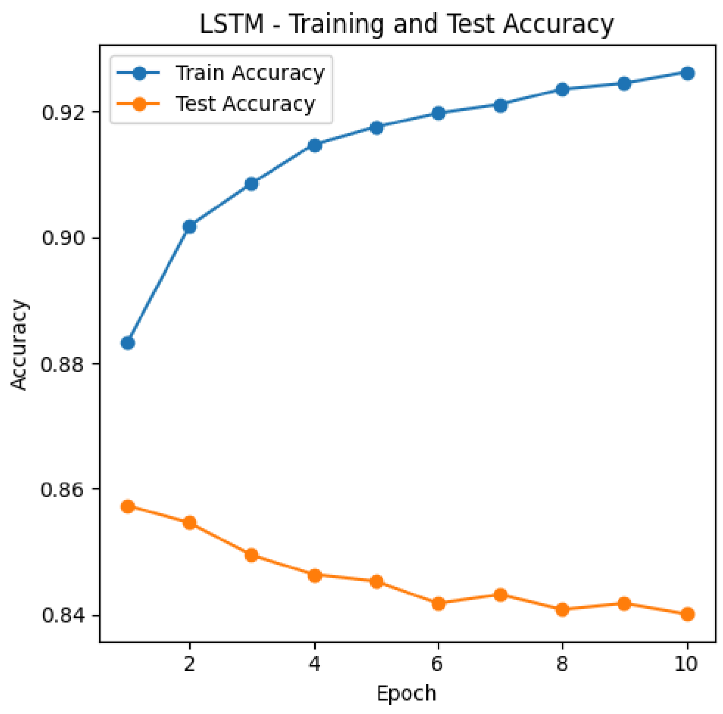


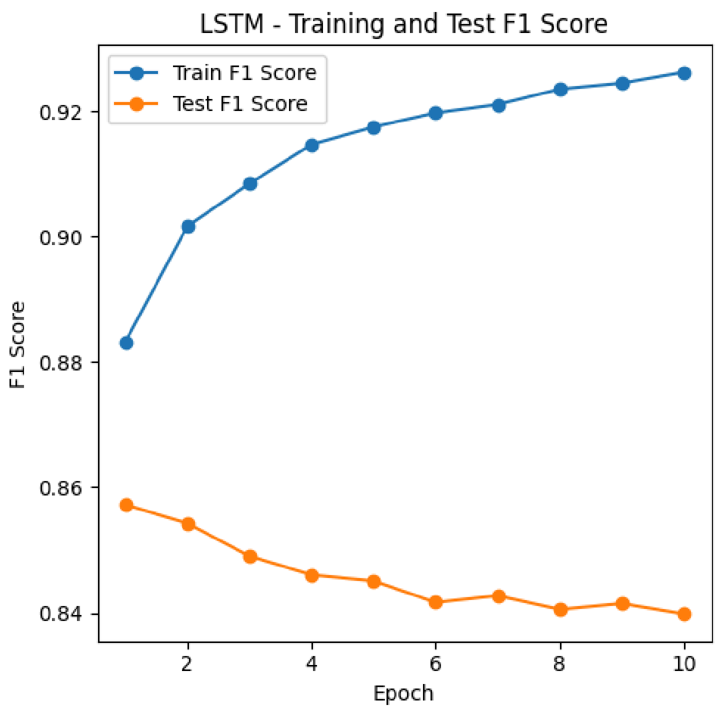
0.8605

2. LSTM

An LSTM model was trained using TF-IDF vectorization. The model was trained for 10 epochs, other parameters were the same as Logistic Regression: CrossEntropy loss, Adam Optimizar and Learning rate= 0.001 and the following results were obtained:

Train Accuracy: 0.9263, Train F1 Score: 0.9262 | Test Accuracy: 0.8401, Test F1 Score: 0.8398





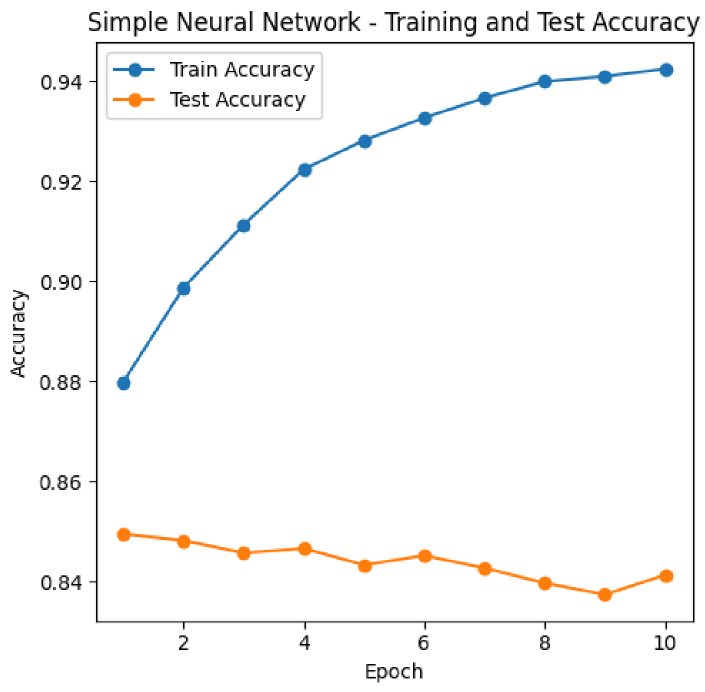
3. Simple Neural Network

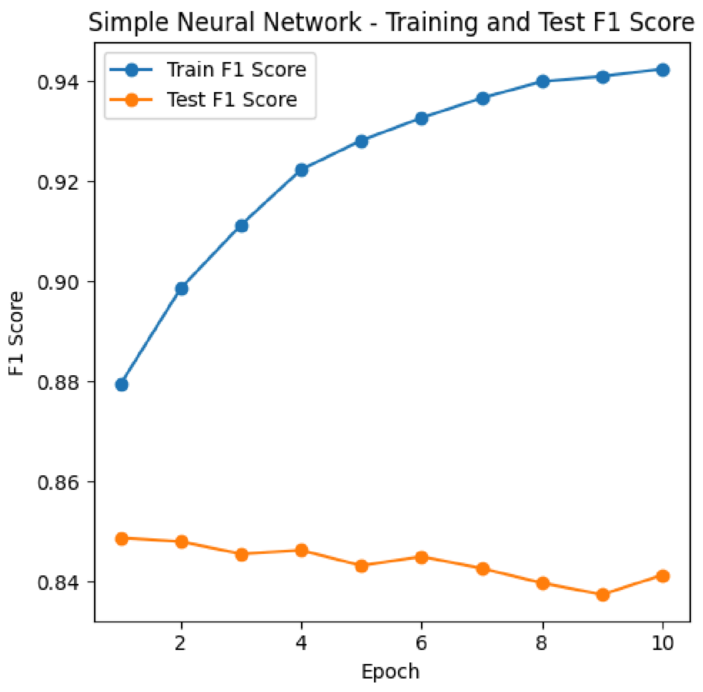
A Simple Neural Network was trained with multiple layers. It has three hidden layers and one output layer, where each layer is followed by a ReLU activation function.

The number of neurons in each hidden layer is 1000, 500, and 50, respectively. Also Dropout layers are added after each activation function to prevent overfitting.

The model achieved the following results after 10 epochs:

Train Accuracy: 0.9424, Train F1 Score: 0.9424 | Test Accuracy: 0.8413, Test F1 Score: 0.8413





4. Bidirectional LSTM

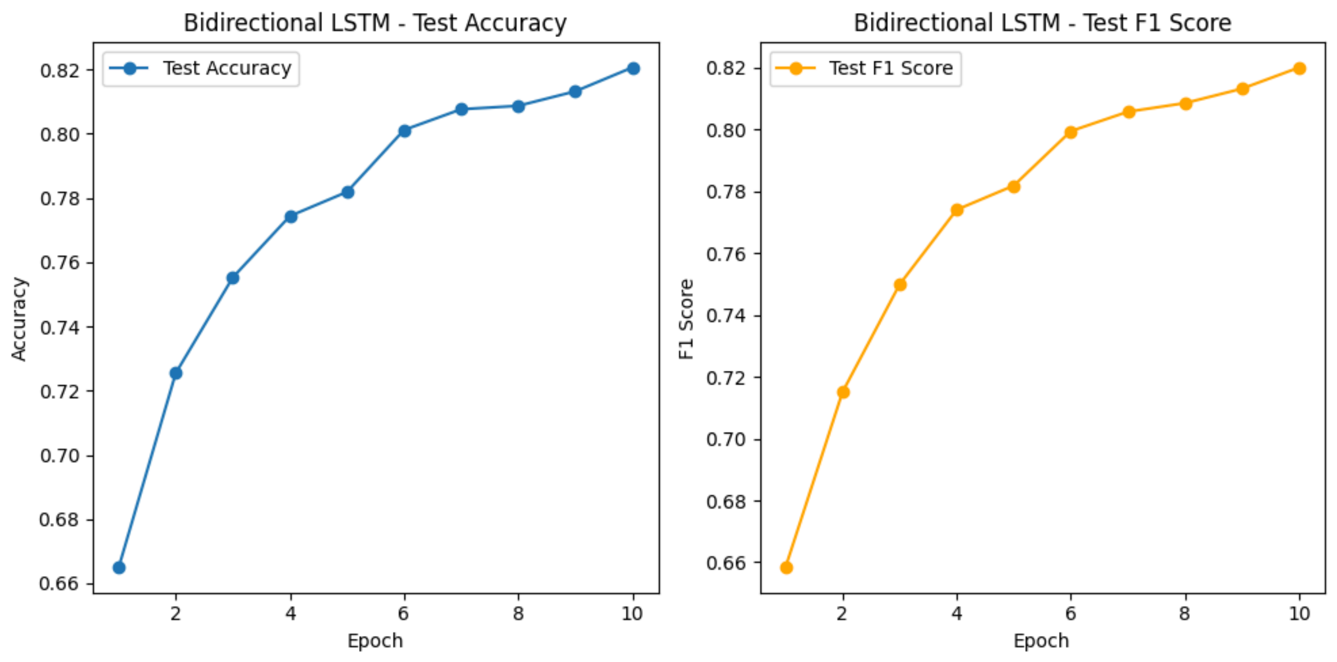
A Bidirectional LSTM model was trained for 10 epochs. The `BiLSTMModel` class defines the architecture of the BiLSTM model.

- It consists of an embedding layer, a bidirectional LSTM layer, two fully connected layers with ReLU activation, and a softmax layer for classification.
- The input size is set to 20,000 (vocabulary size), embedding dimension is 32, hidden size is 128, and there are 2 output classes.

The model is trained using cross-entropy loss and the Adam optimizer.

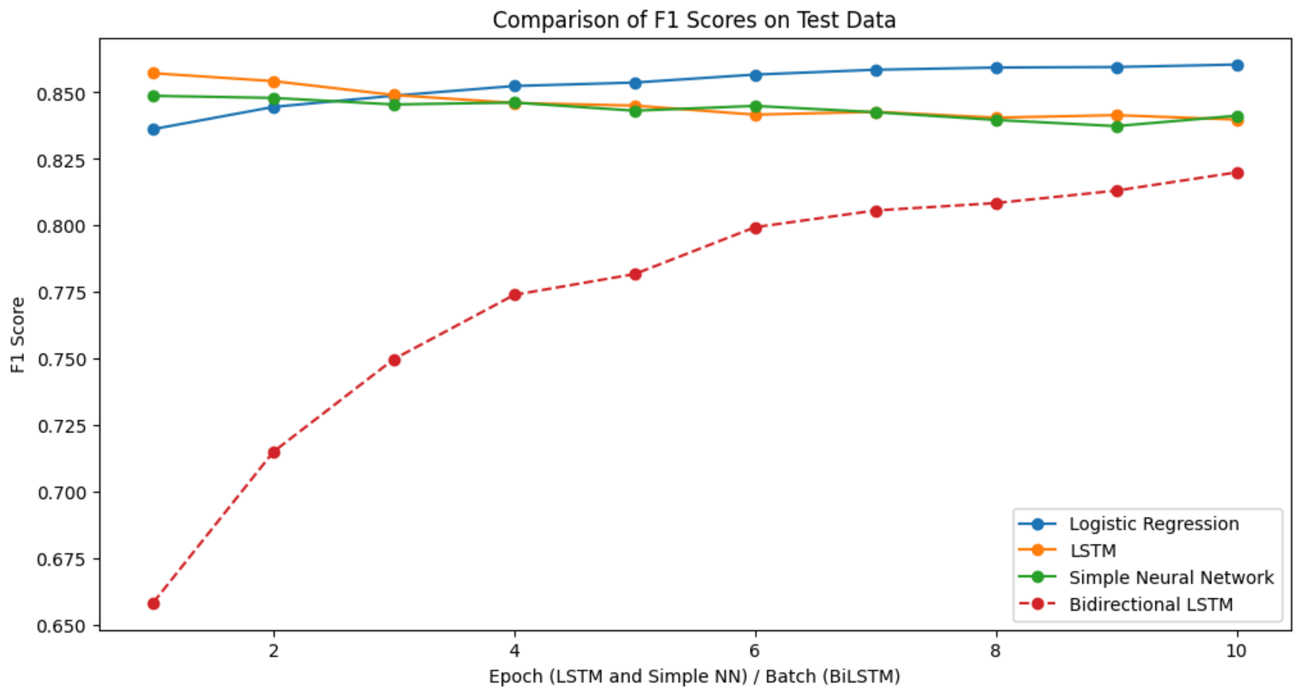
The results are as follows:

Test Accuracy: 82.06% | F1 Score: 0.8200



Overall, comparing F1 scores across different models:

- Logistic Regression: F1-score = 0.8605
- LSTM: F1-score = 0.8398
- Simple Neural Network: F1-score = 0.8413
- Bidirectional LSTM: F1-score = 0.8200



The Logistic Regression model demonstrated the highest F1-score, indicating its effectiveness in capturing sentiment in Persian texts. LSTM and NN models also performed well, highlighting the suitability of these models for sentiment analysis in Persian.

Other Preprocessings

Additional preprocessing steps were explored, including stemming, lemmatization, and stop-word removal. However, the results indicated that the initial preprocessing provided optimal performance for the models.

1. Normalization (hazm_normalizer):

    The `Normalizer` from Hazm library is used to normalize the text data. Normalization involves converting characters with different forms or representations into a standard form. This step helps in reducing variations in the text.

2. Tokenization (hazm_tokenizer):

    The `WordTokenizer` from Hazm library is applied to tokenize the normalized text. Tokenization involves breaking down the text into individual words. The tokenizer used here can replace numbers and IDs during tokenization.

3. Stemming (hazm_stemmer):

    The `Stemmer` from Hazm library is used for stemming. Stemming involves reducing words to their base or root form. This is done to simplify the words and group together different inflected forms.

4. Lemmatization (hazm_lemmatizer):

    The `Lemmatizer` from Hazm library is applied for lemmatization. Lemmatization is similar to stemming but aims to reduce words to their base or dictionary form (lemma). It provides a more meaningful representation compared to stemming.

5. Stopword Removal:

A list of stopwords is obtained from the `stopwords_list` function, and stopwords are removed from the tokenized and lemmatized text. Stopwords are common words (e.g., "برای", "که", "و") that are often removed as they may not contribute much to the meaning.

The preprocessed tokens are then joined back into a single string for further processing.

Word Embeddings

Two different embeddings, Word2Vec and FastText, were applied. However, the TF-IDF vectorization performed better for the LSTM models

Word2Vec:

The `Word2Vec` model from the Gensim library is trained on the preprocessed training data. It learns vector representations of words based on their context in the given text.

A function (`transform`) is defined to convert comments into Word2Vec embeddings. It computes the average vector of all word vectors in a comment, forming a representation for the entire comment.

The transformed Word2Vec embeddings are converted to PyTorch tensors and loaded into DataLoader for training and testing.

FastText:

The `FastText` model from Gensim is trained on the preprocessed training data. FastText also considers subword information, making it effective for handling out-of-vocabulary words.

Similar to Word2Vec, a function is defined to convert comments into FastText embeddings.

The transformed FastText embeddings are converted to PyTorch tensors and loaded into DataLoader for training and testing.

LSTM Model Results:

The results are after new preprocessing which 2 different embeddings:

Word2Vec Embeddings:

Epochs 1-10:

Test Accuracy: 81.48% to 83.03% | F1 Score: 0.8135 to 0.8298

FastText Embeddings:

Epochs 1-10:

Test Accuracy: 83.44% to 84.34% | F1 Score: 0.8335 to 0.8427

FastText embeddings show better performance with higher accuracy and F1 scores compared to Word2Vec embeddings.

Conclusion:

In this sentiment analysis study for Persian texts, a comprehensive pipeline was implemented, covering dataset preprocessing, model training, and evaluation. The dataset underwent meticulous NLP preprocessing, including handling non-Persian comments, converting characters, and applying advanced text cleaning techniques. Four models were explored: Logistic Regression, LSTM, Simple Neural Network, and Bidirectional LSTM, each offering unique approaches to sentiment analysis.

The Logistic Regression model, leveraging TF-IDF vectorization, emerged as a standout performer with an accuracy of 86.09% and an F1 score of 0.8605. The LSTM model, designed to capture sequential patterns, demonstrated a competitive performance with an accuracy of 84.01% and an F1 score of 0.8398.

The Bidirectional LSTM model, incorporating both forward and backward contexts, showcased its efficacy in handling the complexities of Persian sentiment analysis, achieving an accuracy of 82.06% and an F1 score of 0.8200. These findings highlight the promise of more intricate architectures in tackling the nuances of sentiment expressed in Persian.

Comparisons among models revealed that while simpler models like Logistic Regression can yield robust results, the Bidirectional LSTM model, despite a slightly lower F1 score, demonstrated competitive performance. The study also explored additional preprocessing steps, such as stemming and lemmatization, yet found that the initial preprocessing pipeline provided optimal results for the chosen models.

Incorporating Word2Vec and FastText embeddings brought an additional layer of depth to the analysis. The results for the LSTM model with FastText embedding was better than TF-IDF vectorizing and that's because of the structure of LSTM which works better with FastText.

The conclusion suggests that while straightforward models can yield strong results, more intricate architectures, such as the Bidirectional LSTM, exhibit promise in handling the complexities of sentiment analysis for Persian texts.

To further enhance the models, future research could delve into hyperparameter tuning and explore even more advanced architectures. This study, combining state-of-the-art NLP preprocessing techniques with diverse models, establishes a robust foundation for sentiment analysis in Persian, opening avenues for continual improvement and exploration in this domain.