Emotion Detection Persian Texts

Mobina Kargar
dymamsijhidjj@gmail.com

Abstract

Emotion detection from text is one of the challenging problems in Natural Language Processing. The reason is the unavailability of labeled dataset and the multi-class nature of the problem. Humans have a variety of emotions and it is difficult to collect enough records for each emotion and hence the problem of class imbalance arises. Here we have a labeled data for emotion detection and the objective is to build an efficient model to detect emotion

1 Introduction

This project considers variety of emotions such as “HAPPY”,”SAD”,”ANGRY”,”Hate”,”FEAR”, “SUPRISE”and if none of them then it will be ”OTHER”. I use a Persian text dataset annotated with various emotion categories. The goal is to develop and evaluate machine learning models that can efficiently and accurately predict the emotional content of Persian text helping improve applications like sentiment analysis in Persian language contexts.

2 Features

The data is composed of 2 columns and 6125 entries (The data has 6125rows and 2 columns). We can see all 2 dimensions of our dataset by printing out the first 3 entries:

Table 1: train dataset (3 rows x 2 columns)

	Text	Emotion
0	همش دوربین ثبت‌شده ایا میشه اعتراض زد اصن تاثیر داره اگه اطلاعی داره ممنون میشم راهنمایی	Other
1	صدای پرنده دم دمای صبح متنفرم متنفرم متنفرم	Hate

We can inspect the types of feature columns:

Table 2: Data columns (total 29 columns):

```
<bound method DataFrame.info of

Text   Emotion
0     ... خیلی کوچیک هستن و سایزشون بدرد نمیخوره میخوام        SAD
1        از صدای پرنده دم دمای صبح متنفرم متنفرم متنفرم      HATE
2      ...کیفیتش خیلی خوبه با شک خریدم ولی واقعا راضیم"       SAD
3     ...چون همش با دوربین ثبت شده ، ایا میشه اعتراض زد     OTHER
4                ... این وضع ب طرز خنده داري گریه داره      SAD
...                                            ...      ...
6120  ...مرحوم پیش بینی آبکی زیاد میکرد      مرحوم عجب آ  SURPRISE
6121  ...کلا عین اعتقادات و توئیت زدناتون ... !!   در ق     ANGRY
6122  ...خب وقتی میگی کسی بیاد مارو بگیره یارو ترس میکن       FEAR
6123  ... همون هارو      مگه آهنگ جدیدای خوانندههای دهه  SURPRISE
6124                           نیم دگیرش چطور حل نیشد     OTHER

[6125 rows x 2 columns]>memory usage: 507.6+ KB
```

Check the balance :

```
      Emotion
   OTHER      1681
   ANGRY       923
   SAD         896
   FEAR        757
   SURPRISE    739
   HAPPY       618
   HATE        511
Name: count, dtype: int64
```

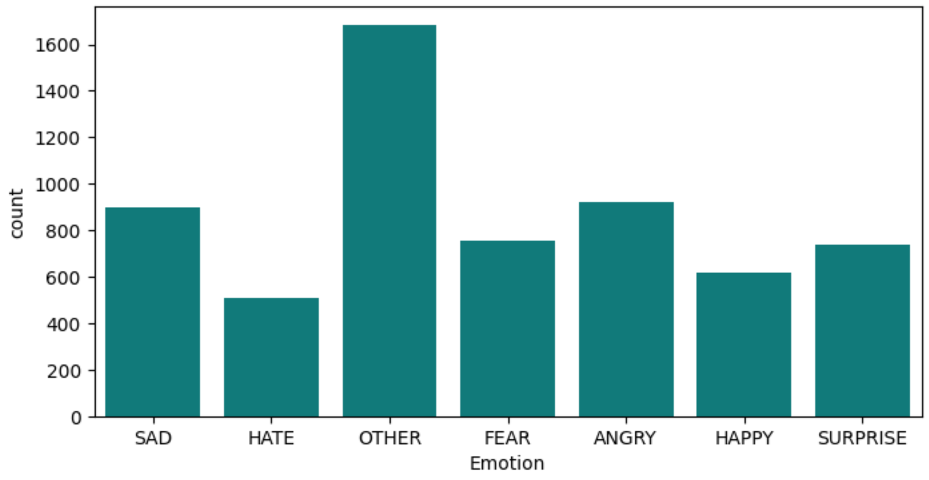
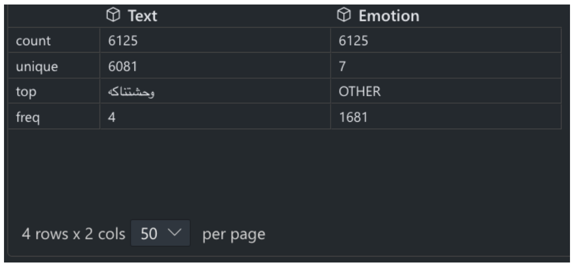


Figure1: The graph indicates that the emotion with the highest intensity is "Other", followed by "Sad" and "Hate". The emotions with the lowest intensities are "Happy" and "Surprise".

Description of Customer Personality:

Table 3: It provides a summary of each column including key statistical metrics such as (counts, unique values , top of them , frequences)



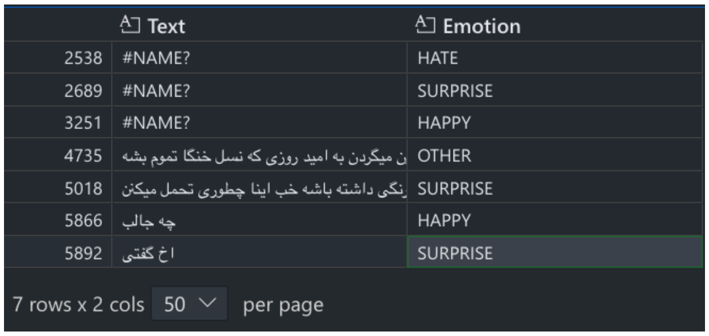
	Text	Emotion
count	6125	6125
unique	6081	7
top	وحشتناک	OTHER
freq	4	1681

4 rows x 2 cols 50 per page

Checking missing value:

```
Text       0
Emotion    0
dtype: int64
37
(6088, 2)
```

We can check the rows which duplicated in the text but with different emotions:



	Text	Emotion
2538	#NAME?	HATE
2689	#NAME?	SURPRISE
3251	#NAME?	HAPPY
4735	ن میگردن به امید روزی که نسل خنگا تموم بشه	OTHER
5018	رنگی داشته باشه خب اینا چطوری تحمل میکنن	SURPRISE
5866	چه جالب	HAPPY
5892	اخ گفتی	SURPRISE

7 rows x 2 cols 50 per page

Count the number of stopwords in the data & Distribution of them:

Stopwords are common words in a language that carry little meaningful content on their own and are often removed during text preprocessing in NLP.For Examples in persain include words like:

"از" ,"به" ,"در" ,"برای" ,"که" ,"با"

```
stop_words
1     743
0     626
2     618
3     522
4     503
5     433
7     366
6     362
8     300
9     281
10    255
11    217
12    200
13    189
14    125
15    105
16     81
17     56
18     44
19     23
20     12
21      6
22      4
24      3
23      3
25      2
87      1
40      1
Name: count, dtype: int64
```

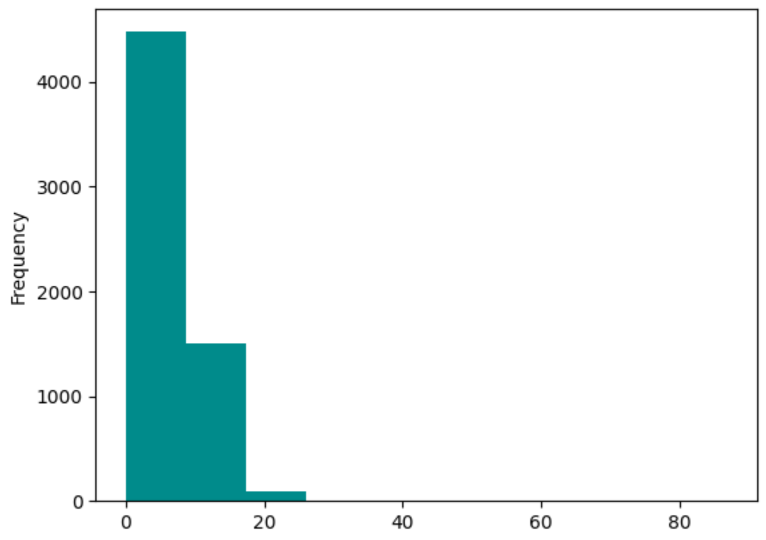


Figure 2: The emotion with the highest intensity or frequency is "Other", represented by the tallest bar on the graph. The next highest intensity emotions are "Sad" and "Hate". The emotions with the lowest intensities are "Happy" and "Surprise". The remaining emotions, "Fear" and "Angry", fall somewhere in the middle in terms of intensity.

Again, let's check if data is balanced or not:

```
Emotion
HAPPY       275
SAD         262
OTHER       193
ANGRY       154
SURPRISE    145
HATE         65
FEAR         57
Name: count, dtype: int64
```

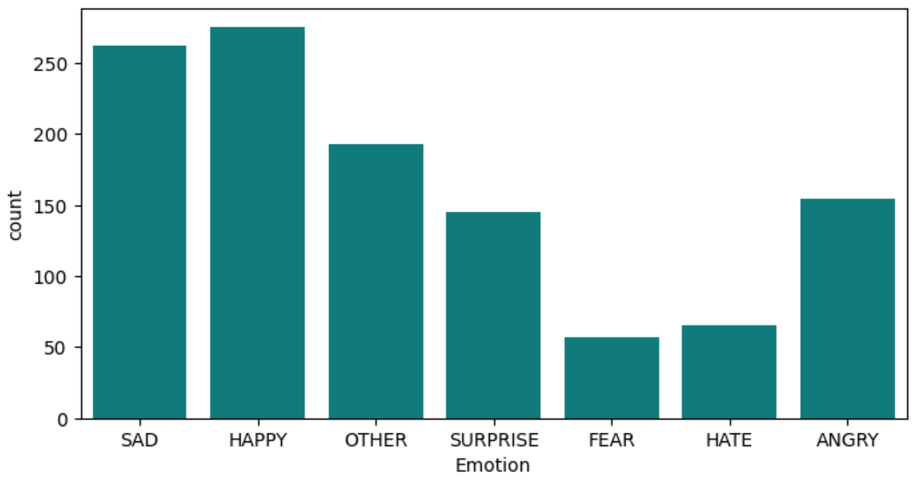
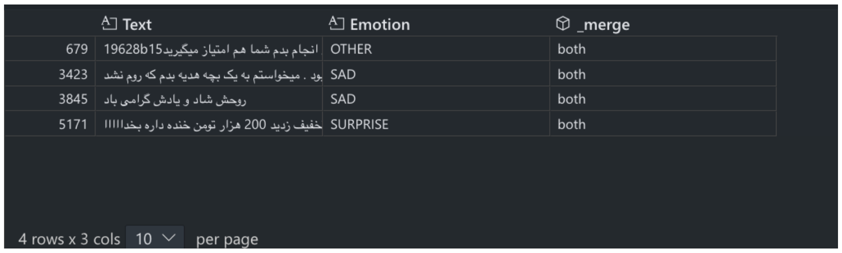


Figure 3: This data suggests that the emotional states are not perfectly balanced, with "Happy" and "Sad" having the highest counts, and "Hate" and "Fear" having the lowest counts. It can be useful for understanding the distribution and prominence of various emotions, which could be relevant for psychological, behavioral, or sentiment analysis research.

Find rows which are different between two DataFrames :

Combine the two DataFrames using a merge operation, with the indicator parameter set to True. This adds a column called _merge to the resulting DataFrame, which indicates the source of each row. Filter the merged DataFrame based on the value of _merge. If which is not specified, return all rows where _merge is not 'both'. Otherwise, return all rows where _merge has the specified value.



	Text	Emotion	_merge
679	انجام بدم شما هم امتیاز میگیرید19628b15	OTHER	both
3423	بود . میخواستم به یک بچه هدیه بدم که روم نشد	SAD	both
3845	روحش شاد و یادش گرامی باد	SAD	both
5171	خفیف زدید 200 هزار تومن خنده داره بخدا!!!!!	SURPRISE	both

4 rows x 3 cols 10 per page

3 Data Exploration and Visualization

In this part I used the Hazm library for Persian text preprocessing. I removed stop words, numbers, and urls to eliminate irrelevant or uninformative content that could negatively impact the model's performance. I also converted all text to lowercase, as Persian words can be harder to read in capital form and consistent casing helps with normalization. Additionally I applied Hazm's normalization functions to standardize the text and used lemmatization to reduce words to their root forms based on Persian morphology. This helps improve the model's ability to generalize across different word forms.

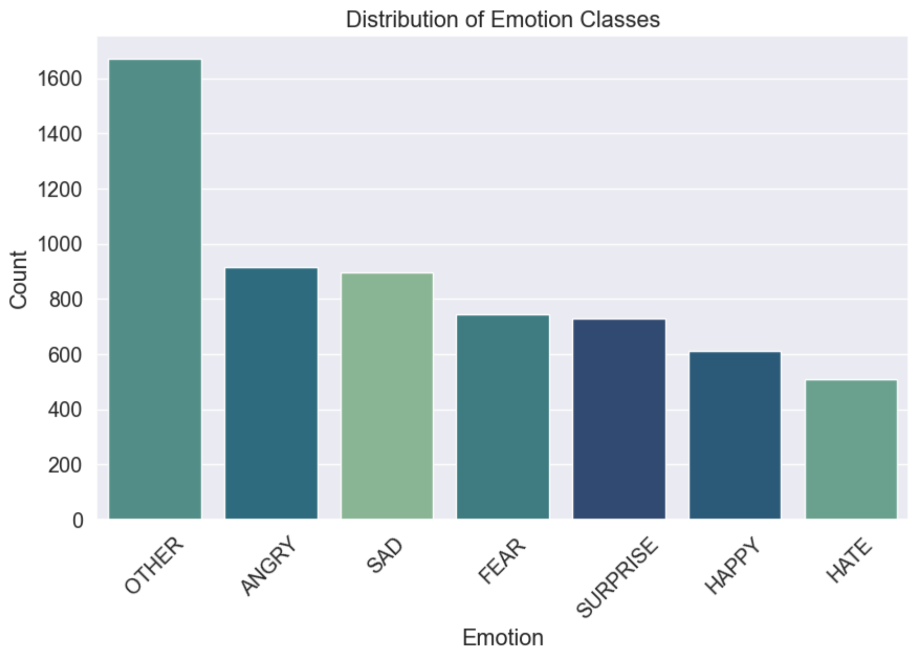


Figure 4: The data exploration and visualization steps described in the image aim to prepare the Persian text data for more effective model training and generalization across different word forms.

In the next step I used Hazm's Normalizer and WordTokenizer to preprocess the Persian text. First, the text was normalized to correct spacing issues, unify characters, and standardize the format. Then, the WordTokenizer was used to split the text into individual tokens (words), preparing it for further analysis or vectorization. The resulting tokens were stored in a new column for later use in the modeling pipeline.

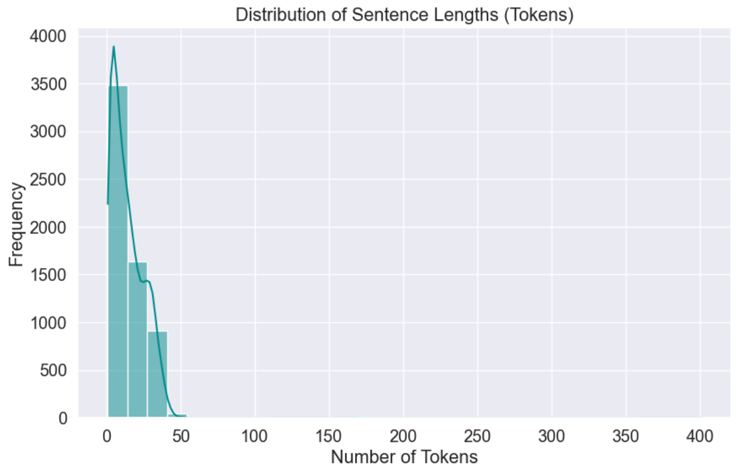


Figure 5: The graph shows the distribution of the number of tokens (words) per sentence in the preprocessed Persian text which distribution has a clear peak, indicating that the majority of sentences contain around 50-100 tokens. And there are also a significant number of sentences with fewer than 50 tokens.The distribution tapers off towards longer sentences, with very few sentences containing more than 300 tokens.

After that I extracted all tokens from the tokens column and counted their frequency using Python's Counter class. Then, I identified the 20 most common words across the dataset. Finally, I visualized their frequencies using a horizontal barplot with Seaborn and Matplotlib. This helps to understand which words appear most frequently in the corpus, providing insight into dominant vocabulary and potential stop words that might need further filtering.

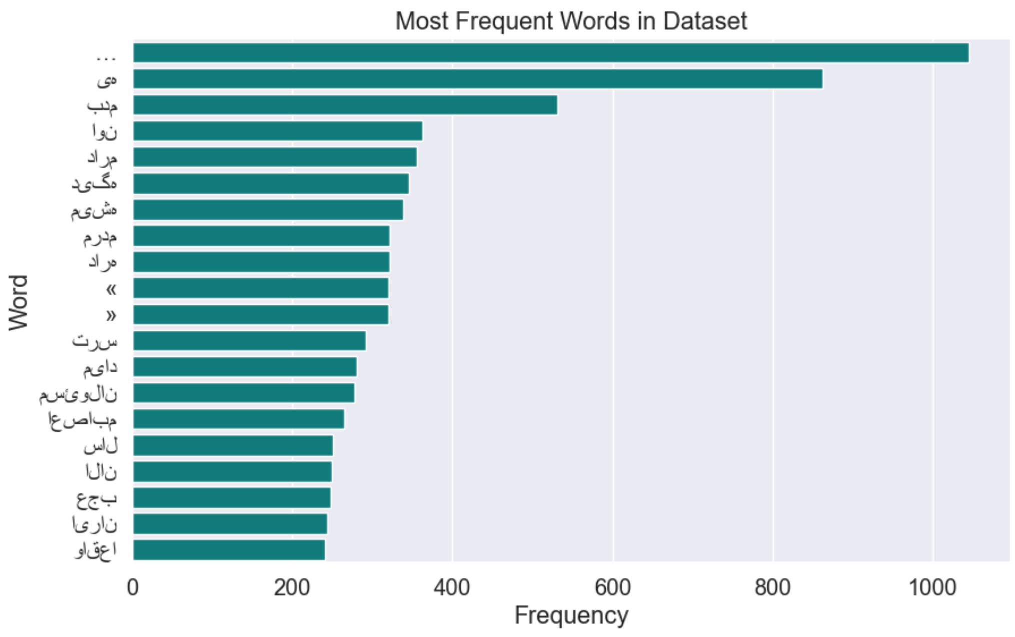


Figure 6: As we see the rate of "…" is more than other words in dataset

4 Modeling

4.1 BaseLine Models

After preprocessing the texts, I encode labels because many machine learning models (like SVM, Logistic Regression and …) require numerical labels, not strings. So I used LabelEncoder to convert the emotion classes ("HAPPY", "SAD",…) into numeric form (0, 1, 2...). I fit the encoder on the training labels and then used the same encoder to transform validation and test labels, ensuring consistency across all splits.

Next, I used TF-IDF vectorization to convert Persian text into numerical feature vectors, capturing both unigrams and bigrams while filtering out overly common and rare terms (max_df=0.9, min_df=5). I then applied SMOTE to balance the training set by generating synthetic samples for minority classes. This helps reduce bias in classifiers trained on imbalanced datasets.

Train a model on the given data and targets:

Parameters:

model (sklearn model): The model to be trained.

data (list of str): The input data.

targets (list of str): The targets.

Returns:

Pipeline: The trained model as a Pipeline object.

Get the F1 score for the given model on the given data and targets:

Parameters:

trained_model (sklearn model): The trained model.

X (list of str): The input data.

y (list of str): The targets.

Returns:

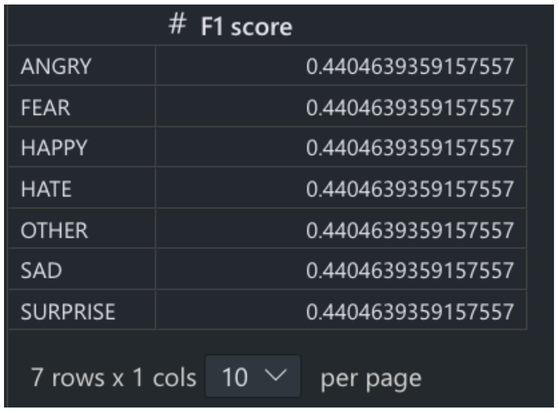
array: The F1 score for each class.

As a summary the F1 function calculates the weighted F1-score to evaluate the model's performance, especially useful for imbalanced class distributions. And in the train function, I defined a simple pipeline that trains any given scikit-learn model on the input features and labels.

4.1.1 Logistic Regression Model

I trained a logistic regression model using resampled TF-IDF features to handle class imbalance. After transforming the test data, I evaluated the model using accuracy and the weighted F1-score to assess its overall and class-wise performance.

Accuracy: 0.4474370112945265



#	F1 score
ANGRY	0.4404639359157557
FEAR	0.4404639359157557
HAPPY	0.4404639359157557
HATE	0.4404639359157557
OTHER	0.4404639359157557
SAD	0.4404639359157557
SURPRISE	0.4404639359157557

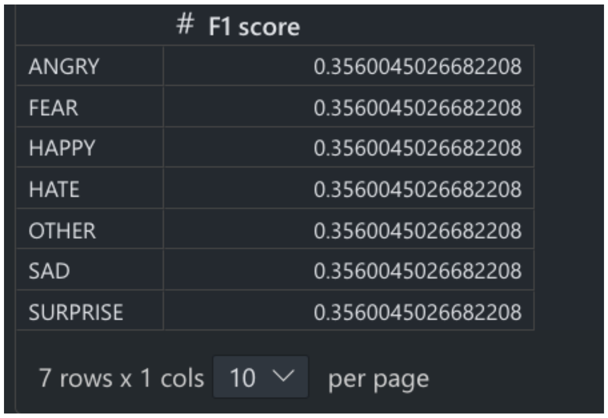
7 rows x 1 cols 10 per page

```
              precision    recall  f1-score   support

       ANGRY       0.45      0.31      0.36       154
        FEAR       0.62      0.67      0.64        57
       HAPPY       0.59      0.37      0.45       275
        HATE       0.22      0.11      0.14        65
       OTHER       0.30      0.66      0.41       193
         SAD       0.53      0.60      0.56       262
    SURPRISE       0.59      0.27      0.37       145

    accuracy                           0.45      1151
   macro avg       0.47      0.42      0.42      1151
weighted avg       0.49      0.45      0.44      1151
```

4.1.2 Decision Tree Model

Int this section I use tfidf transform to convert the training text data into TF-IDF feature vectors using the previously fitted vectorizer. This ensures consistent feature representation across models without refitting the vectorizer, preventing data leakage. The transformed data is then used to train the Decision Tree model, and predictions are made on similarly transformed test data to evaluate performance.

```
Accuracy:  0.3631624674196351
```



#	F1 score
ANGRY	0.3560045026682208
FEAR	0.3560045026682208
HAPPY	0.3560045026682208
HATE	0.3560045026682208
OTHER	0.3560045026682208
SAD	0.3560045026682208
SURPRISE	0.3560045026682208

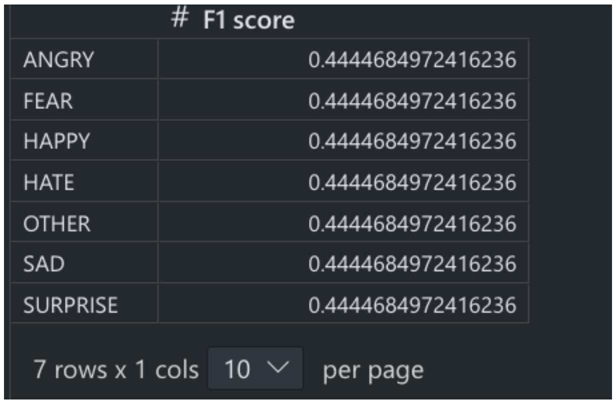
7 rows x 1 cols 10 per page

```
            precision    recall  f1-score   support

   ANGRY       0.35      0.31      0.33       154
    FEAR       0.50      0.63      0.56        57
   HAPPY       0.48      0.21      0.30       275
    HATE       0.26      0.17      0.21        65
   OTHER       0.26      0.59      0.36       193
     SAD       0.43      0.43      0.43       262
SURPRISE       0.51      0.27      0.35       145

    accuracy                           0.36      1151
   macro avg       0.40      0.37      0.36      1151
weighted avg       0.41      0.36      0.36      1151
```

4.1.3 Support Vector Machine Model

I train SVM too with a linear kernel and balanced class weights to handle class imbalance in the data. After vectorizing the test data using the previously fitted TF-IDF vectorizer, I predicted the labels and evaluated the model. The SVM achieved an accuracy of approximately 45%, and the weighted F1 score provides insight into its performance across all classes, accounting for imbalanced data distribution.

```
Accuracy:  0.45004344048653344
```



	# F1 score
ANGRY	0.4444684972416236
FEAR	0.4444684972416236
HAPPY	0.4444684972416236
HATE	0.4444684972416236
OTHER	0.4444684972416236
SAD	0.4444684972416236
SURPRISE	0.4444684972416236

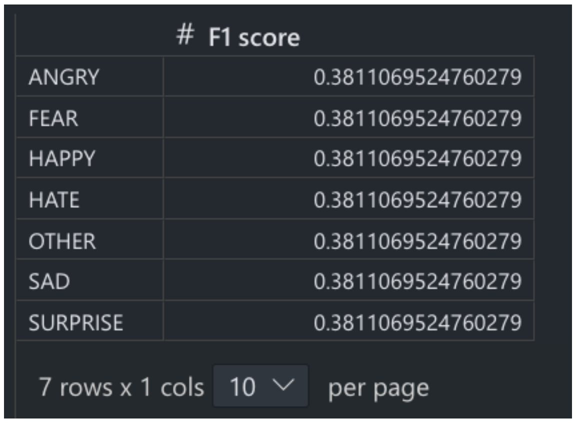
7 rows x 1 cols 10 per page

```
            precision    recall  f1-score   support

   ANGRY       0.48      0.39      0.43       154
    FEAR       0.69      0.61      0.65        57
   HAPPY       0.62      0.39      0.48       275
    HATE       0.19      0.09      0.12        65
   OTHER      0.30     0.65     0.41      193
     SAD      0.50     0.58     0.54      262
SURPRISE      0.60     0.23     0.34      145

    accuracy                    0.45     1151
   macro avg      0.48     0.42     0.42     1151
weighted avg      0.50     0.45     0.44     1151
```

4.1.4 Random Forest Classifier Model

I trained a Random Forest with 200 trees and a maximum depth of 50, using balanced class weights to address class imbalance. The model was trained on TF-IDF vectorized data and evaluated on the test set. The accuracy and weighted F1 scores reflect its overall classification performance, and the detailed classification report shows precision, recall, and F1 for each emotion class.

Accuracy: 0.38922675933970463



	# F1 score
ANGRY	0.3811069524760279
FEAR	0.3811069524760279
HAPPY	0.3811069524760279
HATE	0.3811069524760279
OTHER	0.3811069524760279
SAD	0.3811069524760279
SURPRISE	0.3811069524760279

7 rows x 1 cols 10 per page

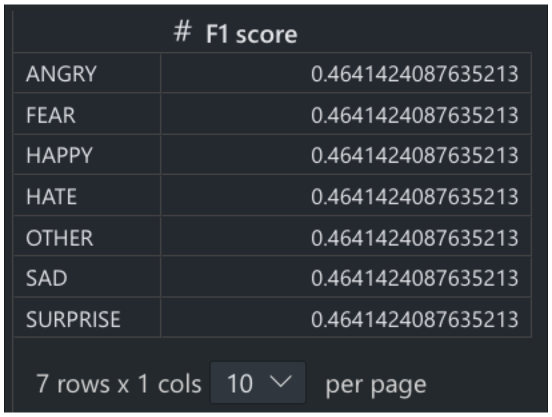
```
          precision    recall  f1-score   support

   ANGRY      0.47     0.28     0.35      154
    FEAR      0.64     0.67     0.66       57
   HAPPY      0.46     0.20     0.28      275
    HATE      0.29     0.15     0.20       65
   OTHER      0.26     0.79     0.39      193
     SAD      0.59     0.43     0.50      262
SURPRISE      0.61     0.25     0.35      145

    accuracy                    0.39     1151
   macro avg      0.48     0.40     0.39     1151
```

	precision	recall	f1-score	support
weighted avg	0.48	0.39	0.38	1151

4.1.5 XGBoost Model (Best Model)

This powerful gradient boosting algorithm configured with multiclass as the evaluation metric and a fixed random state for reproducibility. The model was trained on the SMOTE resampled TF-IDF features to better handle class imbalance. The results show an improved accuracy and weighted F1 score, indicating effective performance in emotion classification.

Accuracy: 0.46481320590790615

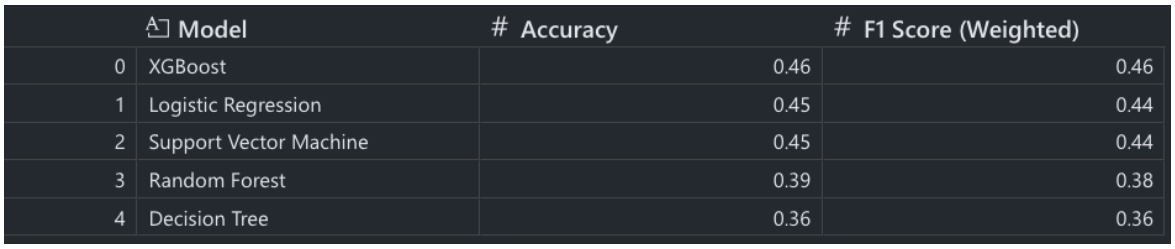


	# F1 score
ANGRY	0.4641424087635213
FEAR	0.4641424087635213
HAPPY	0.4641424087635213
HATE	0.4641424087635213
OTHER	0.4641424087635213
SAD	0.4641424087635213
SURPRISE	0.4641424087635213

7 rows x 1 cols 10 per page

	precision	recall	f1-score	support
ANGRY	0.51	0.37	0.43	154
FEAR	0.61	0.72	0.66	57
HAPPY	0.63	0.40	0.49	275
HATE	0.34	0.28	0.31	65
OTHER	0.32	0.68	0.43	193
SAD	0.53	0.52	0.52	262
SURPRISE	0.56	0.30	0.39	145
accuracy			**0.46**	1151
macro avg	0.50	0.47	0.46	1151
weighted avg	0.51	0.46	0.46	1151

It shows the accuracy and weighted F1 score for Logistic Regression, Decision Tree, Support Vector Machine, Random Forest, and XGBoost models. The results are rounded to two decimal places and sorted by accuracy, providing a clear comparison of each model's effectiveness in detecting emotions from Persian text.



	Model	Accuracy	F1 Score (Weighted)
0	XGBoost	0.46	0.46
1	Logistic Regression	0.45	0.44
2	Support Vector Machine	0.45	0.44
3	Random Forest	0.39	0.38
4	Decision Tree	0.36	0.36

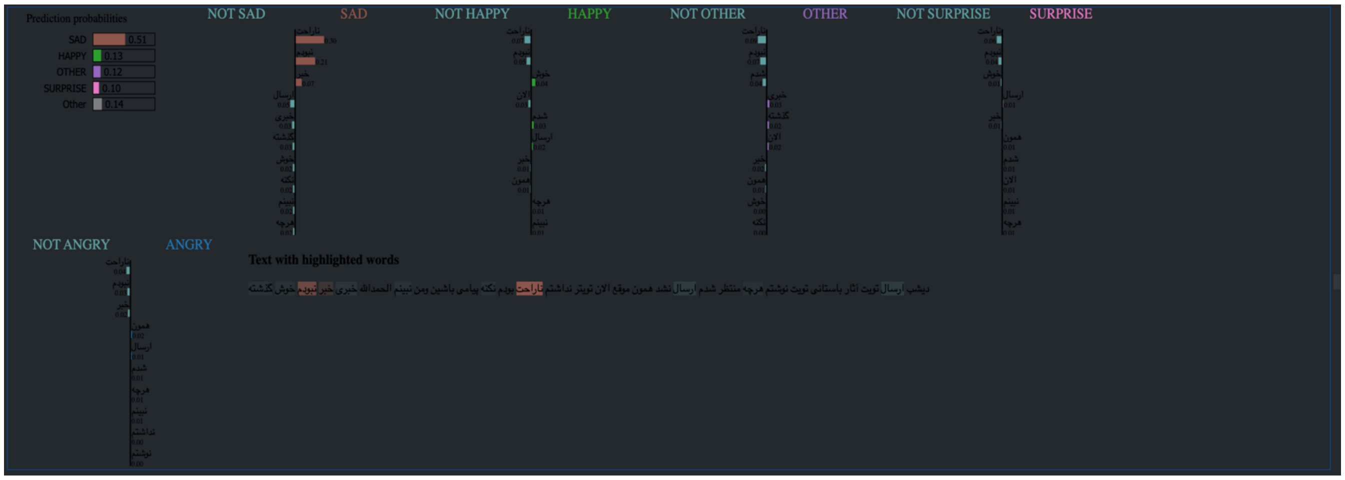
LIME, the acronym for local interpretable model-agnostic explanations, is a technique that approximates any black box machine learning model with a local, interpretable model to explain each individual predictionwe need which words contributed the most in the predicition

For example)

Actual Text : دیشب ارسال تویت آثار باستانی تویت نوشتم هرچه منتظر شدم ارسال نشد همون موقع الان تویتر نداشتم ناراحت بودم نکنه پیامی باشین ومن نبینم الحمدالله خبری خبر نبودم خوش گذشته

Prediction : SAD

Actual : HAPPY



4.2 New Model

In this section after training the data I use class weight because I couldn't use SMOTE to balance the dataset because TF-IDF creates sparse data, and SMOTE doesn't work well with that. Instead, I used class weights to handle class imbalance. I calculated the weights using compute_class_weight to give more importance to rare classes. Then I used these weights to create sample_weights, so that the model focuses more on the underrepresented classes during training.

4.2.1 Tokenize & Padding

I normalize Persian text to unify characters (e.g., converting Arabic "ي" to Persian "ی") and remove diacritics for consistency. Then, I fit a tokenizer with an oov_token='UNK' to handle unseen words.

Finally, I convert the normalized text into integer sequences for model input. This ensures cleaner data and better model generalization, especially in deep learning tasks like LSTM.

```
Test shape is like: (1151, 2)
```

After that, I calculated the maximum sequence length from my training data and used pad_sequences to ensure all sequences had the same length by padding or truncating from the beginning. This step is essential for feeding data into LSTM models. I also computed the vocabulary size from the tokenizer to define the input dimension for the embedding layer.

```
Vocabulary size = 22837
```

Before padding, sequences_train[0] was a variable-length list of token indices for the first training sample. After applying pad_sequences, X_train[0] became a fixed-length, padded version of that list, making it compatible with neural network input requirements.

Before

```
[1933,157,43,8727,8728,201,8729,424,972,250,2909,138,1361,2515,587,1707,731,2910,5680,138,2
50,14,327,817,262,2, 424,588,424,1934,908,424,1708,8730]
```

After

```
array([0, 0, ..., 0, 1933, 157, 43, ..., 1708, 8730], dtype=int32)
```

4.2.2 Word Embedding

To incorporate semantic information from Persian words, I used a pre-trained FastText embedding model (taesiri/PersianWordVecs) and created an embedding matrix. Each word in my tokenizer's vocabulary was mapped to a 100-dimensional vector using fasttext.get_word_vector, forming a matrix of shape (vocabSize, 100) for use in deep learning models.

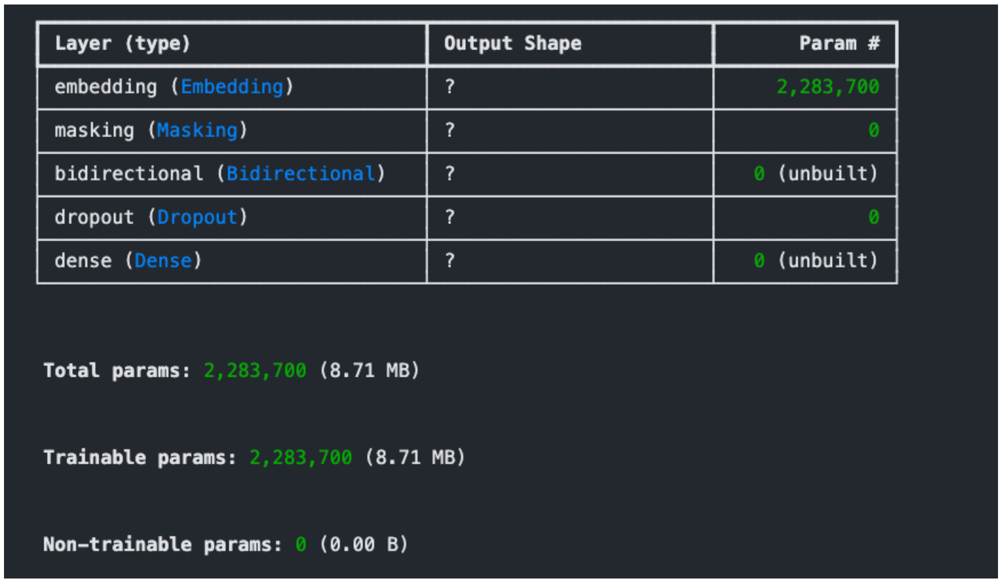
```
Embedding matrix shape: (22837, 100)
```

I checked how many words from my tokenizer's vocabulary were found in the FastText embeddings. For each word, I attempted to retrieve its embedding vector. If successful, the vector was added to the embedding matrix (counted as a hit) it was considered a miss. This helped me track coverage and ensure the quality of the embedding matrix.

```
Converted 22836 words (0 misses) into embedding vectors.
```

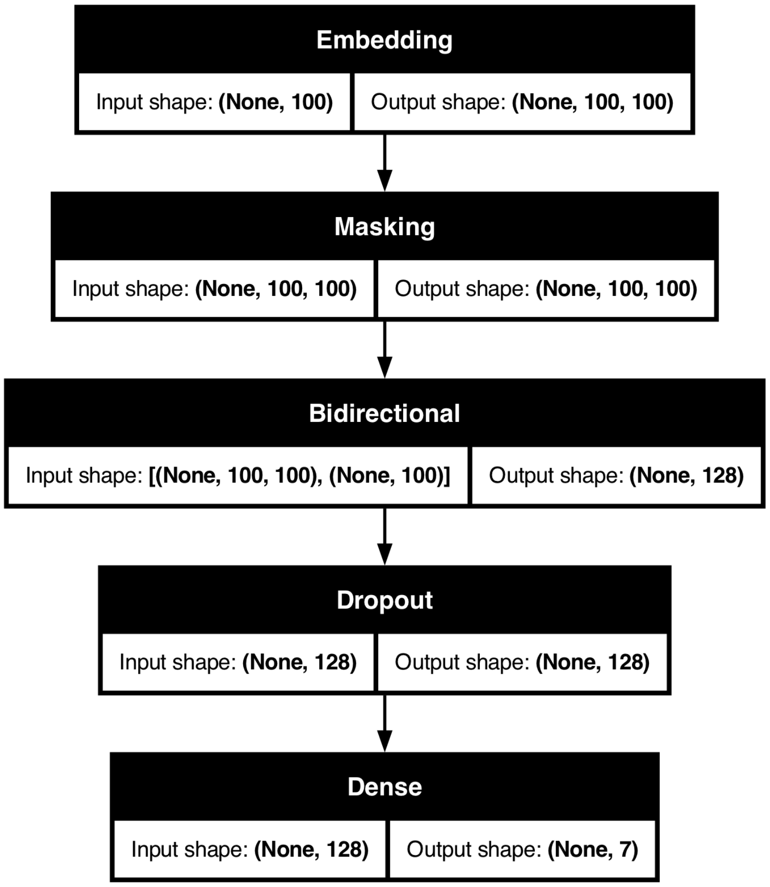
4.2.3 Deep Learning Model

I built a deep learning model using Keras Sequential API with an embedding layer initialized by pre-trained FastText vectors of dimension 100, input length fixed at 100, and trainable embeddings. A masking layer handles padded zeros, followed by a bidirectional LSTM with 64 units and 0.2 dropout and recurrent dropout to capture context and prevent overfitting. After that, a dropout layer with rate 0.5 further reduces overfitting. The output layer is a dense layer with softmax activation matching the number of emotion classes, with L2 regularization applied. The model is compiled using the Adam optimizer with a 0.001 learning rate, categorical crossentropy loss, and metrics including accuracy and a custom F1 score. The summary confirms the architecture and parameters.



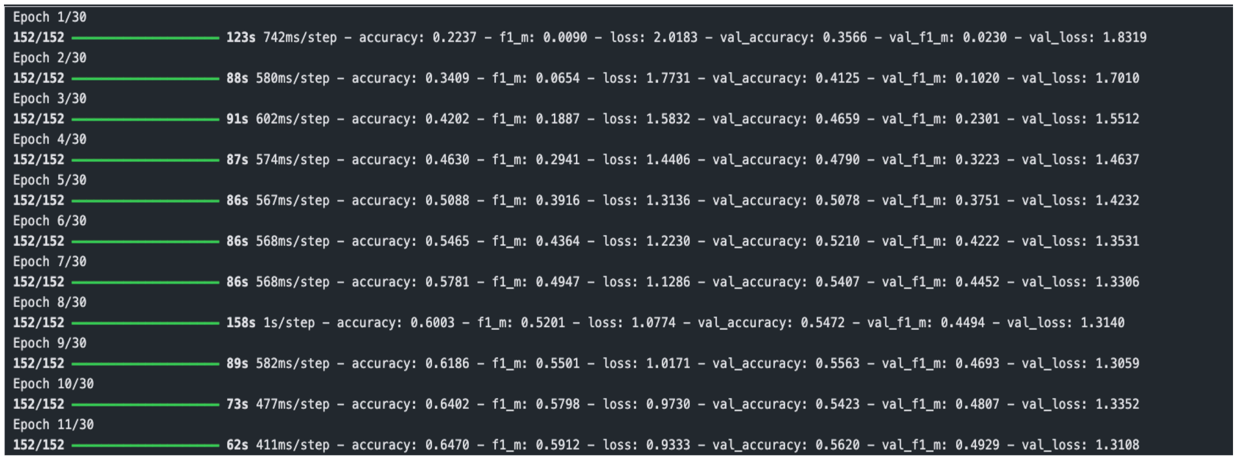
```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ embedding (Embedding)           │ ?                      │     2,283,700 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ masking (Masking)               │ ?                      │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ bidirectional (Bidirectional)   │ ?                      │   0 (unbuilt) │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout (Dropout)               │ ?                      │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense (Dense)                   │ ?                      │   0 (unbuilt) │
└─────────────────────────────────┴────────────────────────┴───────────────┘


 Total params: 2,283,700 (8.71 MB)


 Trainable params: 2,283,700 (8.71 MB)


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

I explicitly built the model by specifying an input shape of (None, 100), where None allows for any batch size and 100 is the fixed sequence length. Then, I visualized the model architecture with TensorFlow's plot_model utility, which displays a diagram of the layers along with their input and output shapes for better understanding and debugging.



Embedding
Input shape: (None, 100)
Output shape: (None, 100, 100)
Masking
Input shape: (None, 100, 100)
Output shape: (None, 100, 100)
Bidirectional
Input shape: [(None, 100, 100), (None, 100)]
Output shape: (None, 128)
Dropout
Input shape: (None, 128)
Output shape: (None, 128)
Dense
Input shape: (None, 128)
Output shape: (None, 7)

I used an EarlyStopping callback to halt training when the validation loss stops improving, with a patience of 2 epochs to avoid stopping too early. This helps prevent overfitting by restoring the best model weights. Then, I trained the model on X_train and y_train, validating on (X_val, y_val) for up to 30 epochs, using a batch size of 32 and applying class weights to handle class imbalance, while showing progress with verbose output.
Then I fit model:



```
Epoch 1/30
152/152 ━━━━━━━━━━━━━━━━━━━━ 123s 742ms/step - accuracy: 0.2237 - f1_m: 0.0090 - loss: 2.0183 - val_accuracy: 0.3566 - val_f1_m: 0.0230 - val_loss: 1.8319
Epoch 2/30
152/152 ━━━━━━━━━━━━━━━━━━━━ 88s 580ms/step - accuracy: 0.3409 - f1_m: 0.0654 - loss: 1.7731 - val_accuracy: 0.4125 - val_f1_m: 0.1020 - val_loss: 1.7010
Epoch 3/30
152/152 ━━━━━━━━━━━━━━━━━━━━ 91s 602ms/step - accuracy: 0.4202 - f1_m: 0.1887 - loss: 1.5832 - val_accuracy: 0.4659 - val_f1_m: 0.2301 - val_loss: 1.5512
Epoch 4/30
152/152 ━━━━━━━━━━━━━━━━━━━━ 87s 574ms/step - accuracy: 0.4630 - f1_m: 0.2941 - loss: 1.4406 - val_accuracy: 0.4790 - val_f1_m: 0.3223 - val_loss: 1.4637
Epoch 5/30
152/152 ━━━━━━━━━━━━━━━━━━━━ 86s 567ms/step - accuracy: 0.5088 - f1_m: 0.3916 - loss: 1.3136 - val_accuracy: 0.5078 - val_f1_m: 0.3751 - val_loss: 1.4232
Epoch 6/30
152/152 ━━━━━━━━━━━━━━━━━━━━ 86s 568ms/step - accuracy: 0.5465 - f1_m: 0.4364 - loss: 1.2230 - val_accuracy: 0.5210 - val_f1_m: 0.4222 - val_loss: 1.3531
Epoch 7/30
152/152 ━━━━━━━━━━━━━━━━━━━━ 86s 568ms/step - accuracy: 0.5781 - f1_m: 0.4947 - loss: 1.1286 - val_accuracy: 0.5407 - val_f1_m: 0.4452 - val_loss: 1.3306
Epoch 8/30
152/152 ━━━━━━━━━━━━━━━━━━━━ 158s 1s/step - accuracy: 0.6003 - f1_m: 0.5201 - loss: 1.0774 - val_accuracy: 0.5472 - val_f1_m: 0.4494 - val_loss: 1.3140
Epoch 9/30
152/152 ━━━━━━━━━━━━━━━━━━━━ 89s 582ms/step - accuracy: 0.6186 - f1_m: 0.5501 - loss: 1.0171 - val_accuracy: 0.5563 - val_f1_m: 0.4693 - val_loss: 1.3059
Epoch 10/30
152/152 ━━━━━━━━━━━━━━━━━━━━ 73s 477ms/step - accuracy: 0.6402 - f1_m: 0.5798 - loss: 0.9730 - val_accuracy: 0.5423 - val_f1_m: 0.4807 - val_loss: 1.3352
Epoch 11/30
152/152 ━━━━━━━━━━━━━━━━━━━━ 62s 411ms/step - accuracy: 0.6470 - f1_m: 0.5912 - loss: 0.9333 - val_accuracy: 0.5620 - val_f1_m: 0.4929 - val_loss: 1.3108
```

Overall loss and accuracy:

[1.305948257446289, 0.5562859773635864, 0.46928703784942627]

Overall loss and accuracy:

[1.402864694595337, 0.5273675322532654, 0.45652881264686584]

```
              precision    recall  f1-score   support

           0       0.60      0.52      0.56       154
           1       0.61      0.60      0.60        57
           2       0.64      0.52      0.57       275
           3       0.38      0.23      0.29        65
           4       0.42      0.65      0.51       193
           5       0.49      0.64      0.56       262
           6       0.68      0.28      0.40       145

    accuracy                           0.53      1151
   macro avg       0.55      0.49      0.50      1151
weighted avg       0.55      0.53      0.52      1151
```

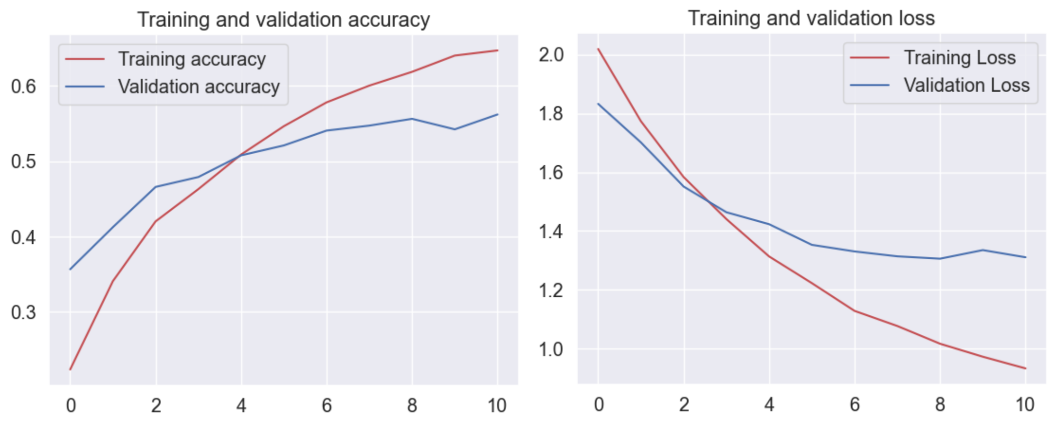


Figure7: The model achieved a moderate overall accuracy of around 53-56% with a corresponding loss between 1.3 and 1.4, indicating there is room for improvement. The precision, recall, and F1-scores vary across classes, with some classes like 0, 1, and 2 performing better, while others such as class 3 show weaker recall and F1, suggesting class imbalance or difficulty in detecting certain emotions. The weighted average F1-score of about 0.52 reflects moderate predictive performance. Overall, the results demonstrate the model's ability to capture general patterns but highlight the need for further tuning or data augmentation to improve classification, especially for underperforming classes.

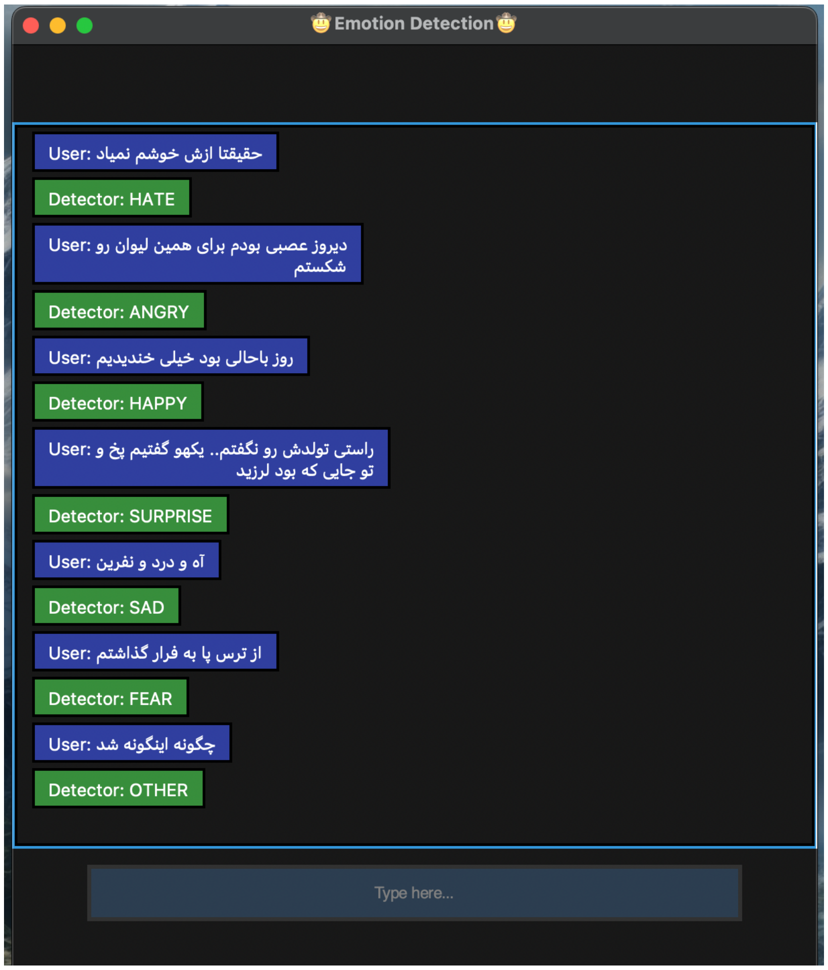
4 Conclusion

In this project, I developed a Persian text emotion detection system using various natural language processing and machine learning techniques. I started by preprocessing the data with the Hazm library to normalize, remove stop words, numbers, punctuation, and URLs, and lemmatize the texts to create clean and standardized input. I then tokenized and padded the sequences to prepare them for deep learning models.

I experimented with multiple machine learning models including Logistic Regression, Decision Trees, SVM, Random Forest, and XGBoost, evaluating them using accuracy and weighted F1-scores. To address class imbalance, I applied both SMOTE oversampling and class weighting strategies. The best traditional models achieved accuracy around 45-46%, showing moderate performance.

For deep learning, I used pretrained Persian FastText embeddings and built a bidirectional LSTM model with dropout and regularization to prevent overfitting. I trained the model with early stopping based on validation loss. The deep learning model reached approximately 53-56% accuracy and reasonable precision and recall across most emotion classes, although some classes remain challenging due to data imbalance.

At The End, the project highlights the complexity of Persian emotion detection due to multi-class imbalance and limited data. While the models show promising results, further improvements could be achieved by increasing dataset size, enhancing feature engineering, or applying more advanced architectures. This work lays a foundation for building effective emotion recognition systems in Persian language text.



User: حقیقتا ازش خوشم نمیاد

Detector: HATE

User: دیروز عصبی بودم برای همین لیوان رو شکستم

Detector: ANGRY

User: روز باحالی بود خیلی خندیدیم

Detector: HAPPY

User: راستی تولدش رو نگفتم.. یکهو گفتیم پخ و تو جایی که بود لرزید

Detector: SURPRISE

User: آه و درد و نفرین

Detector: SAD

User: از ترس پا به فرار گذاشتم

Detector: FEAR

User: چگونه اینگونه شد

Detector: OTHER

References

[1] Arman-Rayan-Sharif “arman-text-emotion” 2022. https://github.com/Arman-Rayan-Sharif/arman-text-emotion

[2] mohamedabdelmohsen “emotion-analysis-and-classification-using-lstm-93”2022. https://www.kaggle.com/code/mohamedabdelmohsen/emotion-analysis-and-classification-using-lstm-93