Deep Learning for NLP

Konstantinos Malonas
sdi: 7115112200020

Fall Semester 2023

University of Athens Department of Informatics and Telecommunications

Artificial Intelligence II (M138, M226, M262, M325)

Abstract

For this project we will build a sentiment classifier that classifies tweets written in Greek language about the Greek elections. The classifier will classify the tweets as NEGATIVE, POSITIVE and NEUTRAL classes. First, we will start by exploring my data, which includes plotting barplots, word clouds, etc., to recognize potential patterns, understand the predominant sentiment classifications for tweets associated with each party, and so on. Then we will continue with experiments to find the best parameters for our RNN classifier, that we will build with the use of the PyTorch framework, and also we will plot the f1-recall-precision curves from the training of our model in order to depict its performance.

Data processing and analysis

Pre-processing

First we start with the pre-processing of our dataset. Initially we check if there are any null values in any column with the **info** method. We observe that there are no null values.

```
RangeIndex: 36630 entries, 0 to 36629
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   New_ID     36630 non-null  int64
 1   Text       36630 non-null  object
 2   Sentiment  36630 non-null  object
 3   Party      36630 non-null  object
dtypes: int64(1), object(3)
memory usage: 1.1+ MB
```

Turn categorical values to numerical with LabelEncoder

It is a best practice to turn the categorical values of our classes from our dataset to numerical. To achieve that, we use the **LabelEncoder** object. We turn our classes from NEGATIVE to 0, from NEUTRAL to 1, and lastly from POSITIVE to 2.

```
df_train_set['Sentiment'] = le.fit_transform(df_train_set['Sentiment'])
df_valid_set['Sentiment'] = le.fit_transform(df_valid_set['Sentiment'])
```

The preprocess_tweet function

In the data preprocessing stage, the **preprocess_tweet** function plays a crucial role in standardizing and cleaning the tweet data. This function performs several key operations to ensure that the text data is in a suitable format for analysis and modeling.

Firstly, we convert all text to lowercase, which helps in maintaining consistency as text data often contains a mix of uppercase and lowercase letters, and in most contexts, these variations are not meaningful. By standardizing the case, we reduce the complexity of the text and avoid treating the same words in different cases as different tokens.

Next, we remove mentions, which are words starting with the '@' character. Mentions in tweets usually refer to usernames and do not contribute meaningful information for our analysis. Similarly, URLs are removed since they often act as noise in the text analysis, being unique and not contributing to the overall understanding of the tweet's sentiment.

We also remove all non-Greek characters, focusing our analysis strictly on Greek text. This step is crucial as it eliminates irrelevant characters or symbols that might be present in the tweets but do not contribute to their semantic meaning.

Lastly, we remove Greek stop words using the **stopwords_el_2.json** file. Stopwords are common words that appear frequently in the text but do not carry significant meaning and are often filtered out before processing text data. Removing these words helps in reducing the dimensionality of the data and focuses the analysis on the words that carry more meaning and sentiment.

Below is the code of preprocess_tweet function:

```
with open('/kaggle/input/stopwords/stopwords_el_2.json', 'r',
encoding='utf-8') as file:
    greek_stopwords = json.load(file)

def preprocess_tweet(tweet):
    tweet = tweet.lower().replace('_', ' ')
    tweet = re.sub(r'@\w+', '', tweet)
    tweet = re.sub(r'http\S+', '', tweet)
    tweet =
re.sub(r'[^αβγδεζηθικλμνξοπρστυφχψωςάέίόώύήΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟΠΡΣΤΥΦΧΨΩ0
-9\s]', '', tweet)

    tweet_words = tweet.split()
    cleaned_words = [word for word in tweet_words if word not in
greek_stopwords]
    tweet = ' '.join(cleaned_words)
    tweet = tweet.strip()

    return tweet
```

Below is the code from lemmatize_tokenize_text function that we used for tokenization and lemmatization:

```
nlp =
spacy.load('/kaggle/input/el-core-news-lg-4/el_core_news_lg_3/el_core_ne
ws_lg-3.7.0')

def lemmatize_tokenize_text(text):
    doc = nlp(text)
    return ' '.join([token.lemma_ for token in doc])


df_train_set['Text'] =
df_train_set['Text'].apply(lemmatize_tokenize_text)
df_test_set['Text'] = df_test_set['Text'].apply(lemmatize_tokenize_text)
df_valid_set['Text'] =
df_valid_set['Text'].apply(lemmatize_tokenize_text)
```

If we print the values of the **Text** column of the train,test,validation sets we will notice that the words are lemmatized.

```
0    απολυμανση κοριοι απεντομωση κοριος απολυμανσε...
1    έξι νέος επιστολή μακεδονία καίνε νδ μητσοτάκη...
2     ισχυρός κκε δύναμη λαός βουλή καθημερινός αγώνας
3            μνημονιακότατο μερα25 εκλογες 2019 8 κκε
4                  συγκλονιστικός ψυχασθένεια τσίπρας
Name: Text, dtype: object

0    κυριάκος μητσοτάκης ξέρω μουσείο βεργίνας μέσω...
1    συνέντευξη υποψήφιος βουλευτής νέος δημοκρατία...
2    εκλογή μαθητής φοιτητής ψηφίζω ίδιος τρόπος αγ...
3    γεννηματά κιναλ γίνομαι δεκανίκι κανενός ενδια...
4    κυριακός εκλογή οκτώβρης 1993 ξημερώματα δευτέ...
Name: Text, dtype: object

0    θελεις μιλησεις βοσκοτοπια αιγιαλος παραγραφή ...
1    τσίπρας ζητήζω αντιπολίτευση συμμετέχω διαδικα...
2    σωστος ελληνας δημοκρατης ελληνας εξωτερικου ε...
3    βλέπεις ενδιαφέρω μητσοτακηδας γιατί πήγε κότε...
4    συνέντευξη μητσοτάκης αίρεση 13ος σύνταξη αύξη...
Name: Text, dtype: object
```

The unique_words_num function

After we pre-processed our text we printed the unique words of each set with the use of **unique_words_num** function

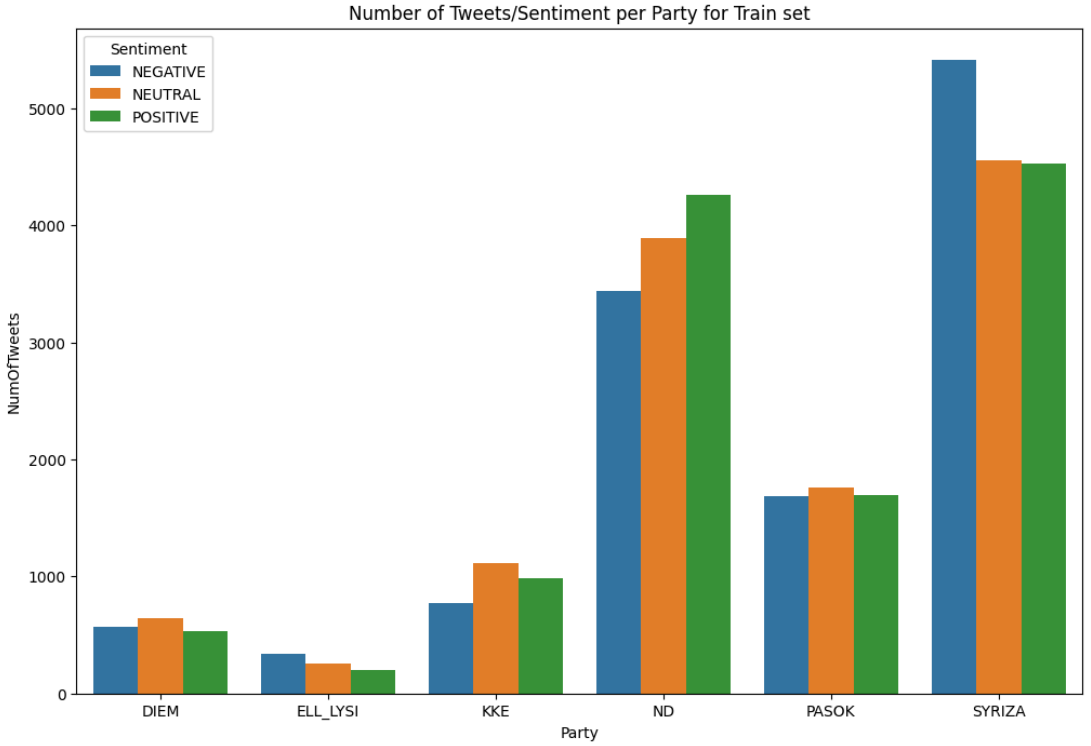
```
def unique_words_num(tweets):
    # Function that counts the number of the unique words from the Text
column of each dataframe
    words = set()
    for tweet in tweets:
        words.update(tweet.split())
    return len(words)

Result:
Num of unique words in df_train_set: 58389
Num of unique words in df_test_set: 26263
Num of unique words in df_valid_set: 16639
```

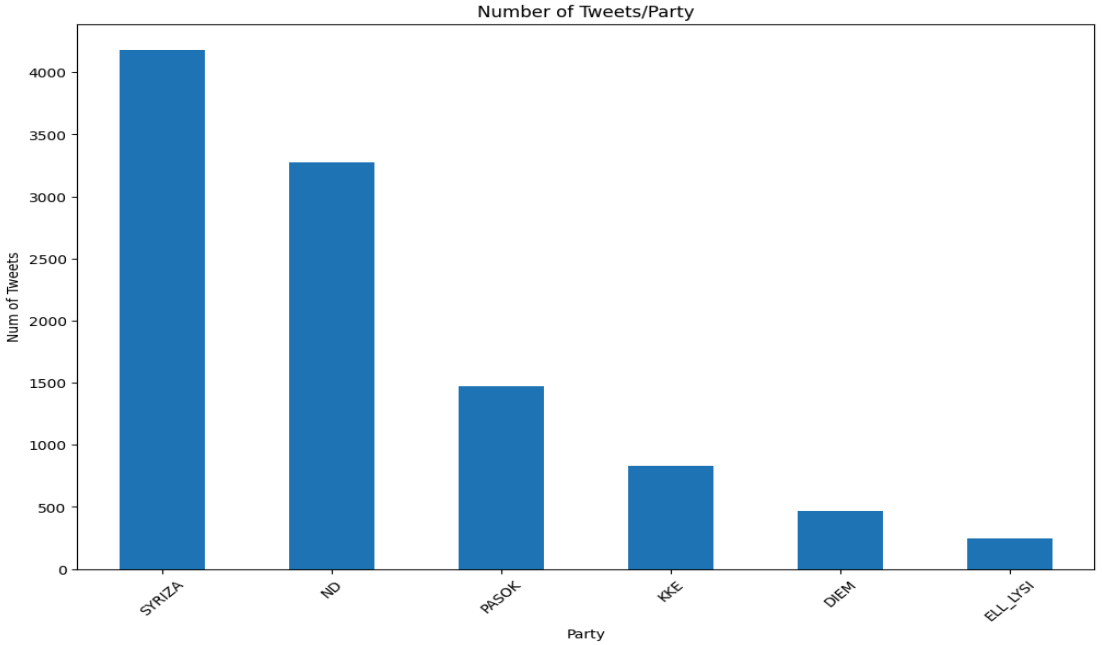
Analysis

For each set (train, validation and test) we plotted some barplots to visualize the number of tweets about each party and the emotion of these tweets. Also we plot the total number of tweets for each party. Also after lemmatization and the general preprocessing of the tweets we plotted the corresponding word cloud for each set.

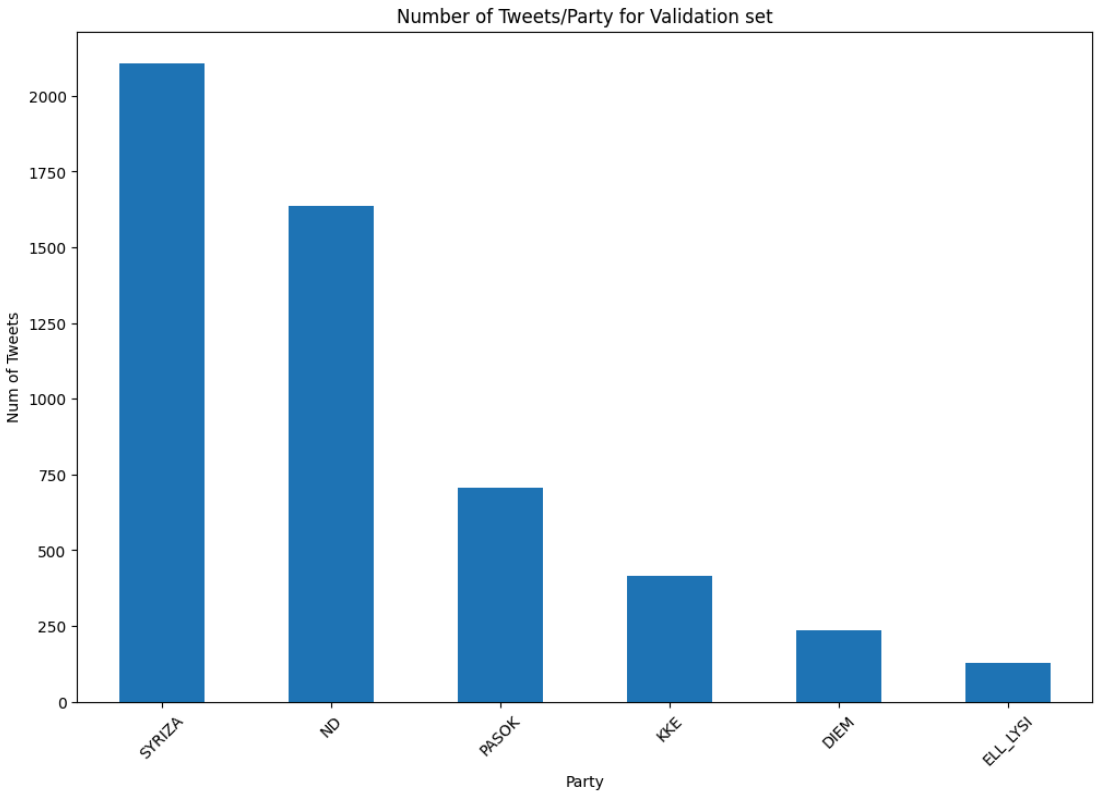
Number of tweets and sentiment for each party



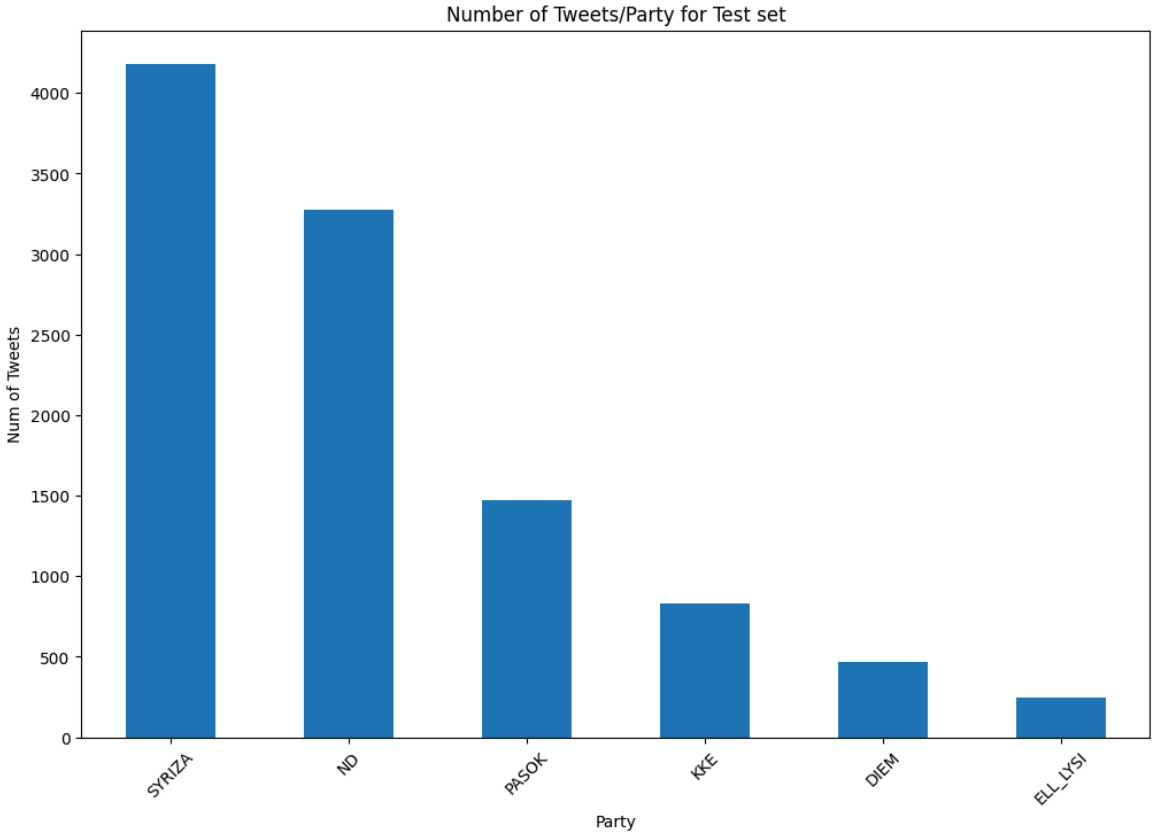
Number of tweets for each party for the train set



Number of tweets for each party from the validation set



Number of tweets for each party from the test set

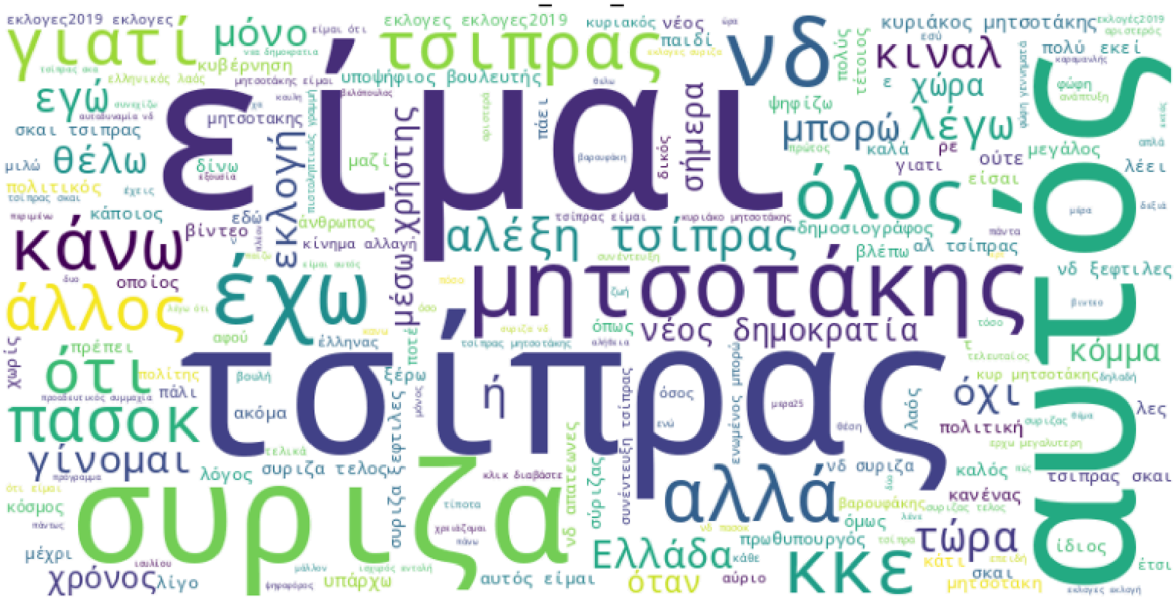


Word cloud for train set

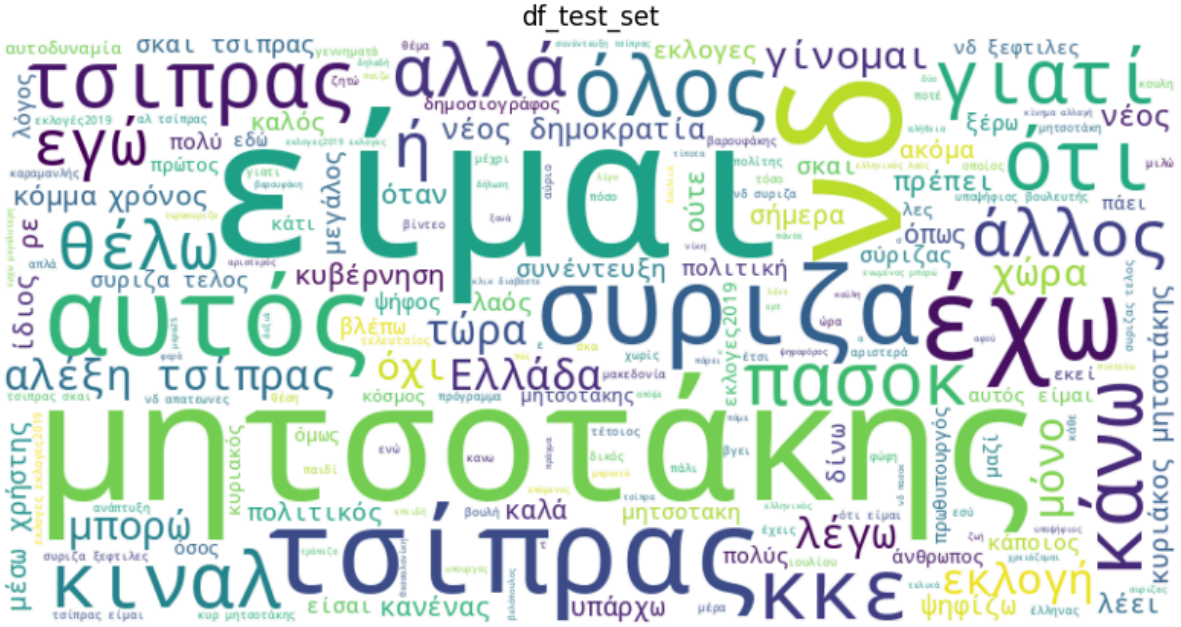
In our analysis, we have utilized word clouds as a visual tool to represent the data from our train, test, and validation sets. Word clouds, or tag clouds, are graphical representations where the frequency of each word in the text data is depicted with various font sizes. The more frequently a word appears in the dataset, the larger and bolder it appears in the word cloud. This visualization technique is particularly useful for quickly identifying key themes and terms that are most prominent in large volumes of text.

The word clouds generated for our datasets reveal significant insights, especially concerning the political context of the tweets. Given that our data is centered around the Greek general elections, it is not surprising to find that the names of the two dominant parties at the time, Syriza (Σύριζα) and Nea Dimokratia (Νέα Δημοκρατία), along with the names of their party leaders, are among the most prominently featured words in the tweets. This observation aligns with the expected discourse during an election period, where discussion and commentary are often focused on the major political parties and their leaders.

df_train_set



Word cloud for test set



Word cloud for validation set

Data partitioning for train, test and validation

In our project, we initially worked with the given datasets as they were originally partitioned into separate training, validation, and testing sets. This conventional partitioning is a standard practice in machine learning, ensuring that models are trained, tuned, and tested on distinct data subsets.

Vectorization

In order for our data to be used from the neural networks we will be building, they must be transformed into vectors of numbers. For that we have used the **Word2Vec** model.

Word2Vec
Word2Vec is used in natural language processing to transform words into numerical vectors. It's based on neural networks and aims to capture the **contextual relationships** between words.

Word Embeddings: Word2Vec produces word embeddings, which are numerical representations of words in a high-dimensional space. In this space, semantically similar words are located close to each other.

Resulting Vectors: After training, each word in the model's vocabulary is associated with a fixed-size vector (e.g., 100 dimensions). These vectors capture semantic information about the words.

Vectorize function steps:

Sentence Splitting:
- Each sentence is split into its constituent words.

Word Vectorization:
- For each word in the sentence, the function retrieves its corresponding vector from the Word2Vec model.
- The function limits the number of word vectors to max_length to maintain a consistent sequence length. If a sentence contains more words than max_length, it truncates the sequence; otherwise, it includes all word vectors.

Padding for Consistent Length:
- A two-dimensional array padded_vecs is initialized with zeros, having dimensions max_length x vector_size (where vector_size is the dimensionality of the Word2Vec embeddings).
- The word vectors (words_vecs) are placed at the beginning of this array.
- If the sentence is shorter than max_length, the remaining part of the array remains zero-padded. This ensures all output sequences have the same length, a requirement for batching in RNNs.

Handling Sentences with Unrecognized Words:

- If a word in the sentence is not present in the Word2Vec vocabulary, it is ignored, and no vector is retrieved for it.
- If none of the words in a sentence are in the Word2Vec vocabulary (i.e., words_vecs is empty), the function returns an array of zeros. This represents a sentence with no recognized words.

Output:

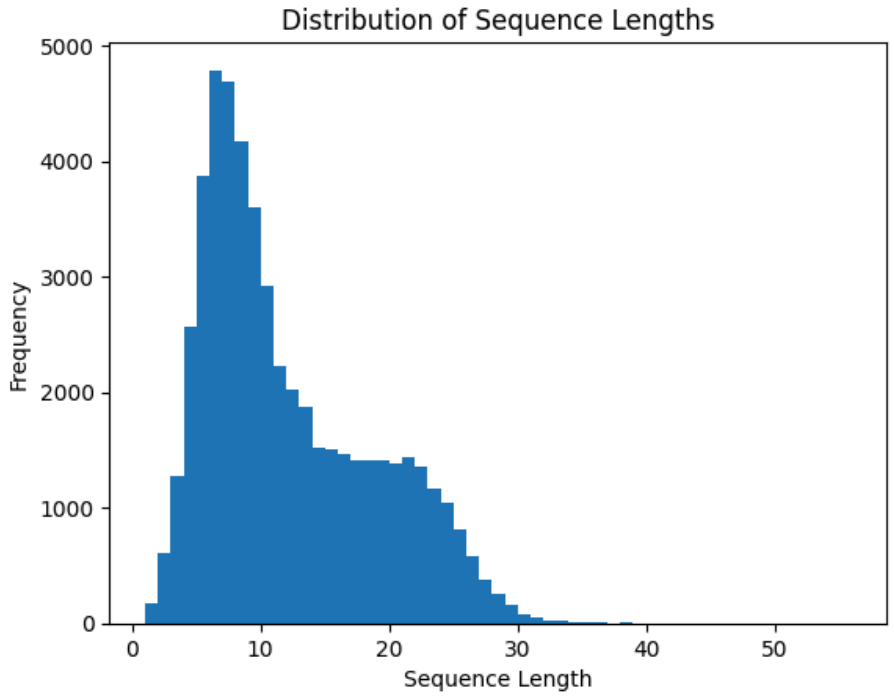
- The function outputs padded_vecs, a two-dimensional numpy array. Each row corresponds to a word vector, and the array length is equal to max_length. This array is the vectorized representation of the sentence, ready to be fed into an RNN.

```
def vectorize(sentence, w2v_model, max_length):
    words = sentence.split()
    words_vecs = [w2v_model.wv[word] for word in words if word in
w2v_model.wv][:max_length]
    padded_vecs = np.zeros((max_length, w2v_model.vector_size))

    if len(words_vecs) > 0:
        padded_vecs[:len(words_vecs)] = words_vecs

    return padded_vecs
```

From the following graph where we have plotted the distribution of the lengths of the sequences, we observe that in order not to lose too much information it is a good approach to choose **max_length** equal to 30 in order to include all the sequences.



Below we can observe the five first rows of our vectorized training set:

```
0    [[-0.16882067918777466, 0.15165138244628906, 0...
1    [[0.04984358325600624, 0.22559648752212524, 0....
2    [[1.3850412368774414, 1.6175470352172852, 0.39...
3    [[-0.0017342357896268368, -0.00815312471240758...
4    [[-0.011479035019874573, 0.19349101185798645, ...
```

Algorithms and Experiments

Experiments with Layers

We conducted experiments for the number of layers equal to 1, 2, 4, 7, 9 and 10.

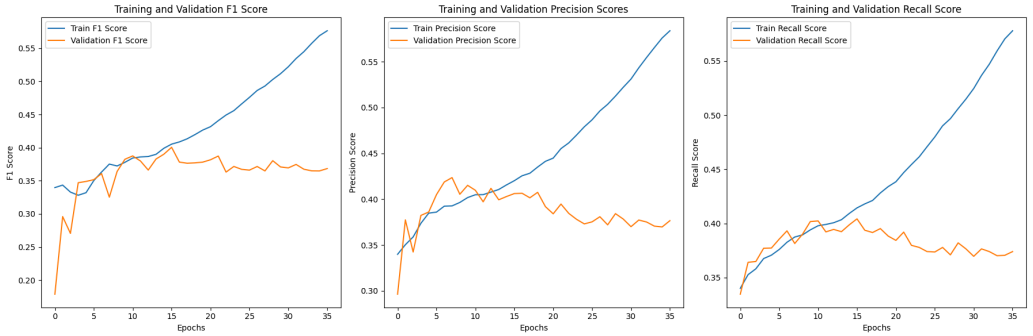
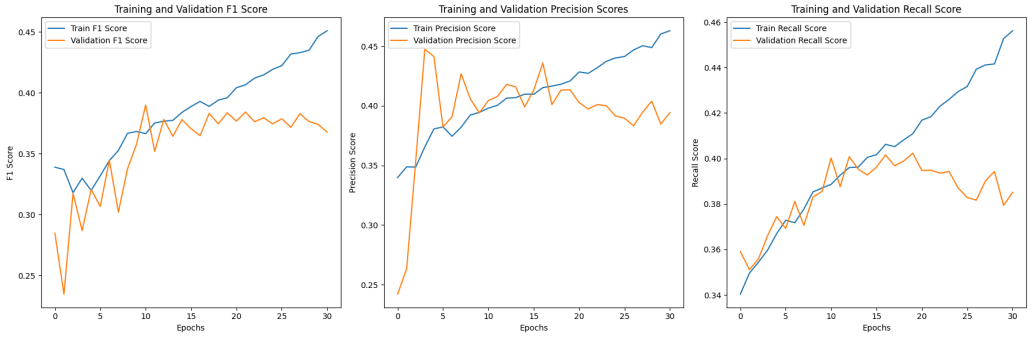
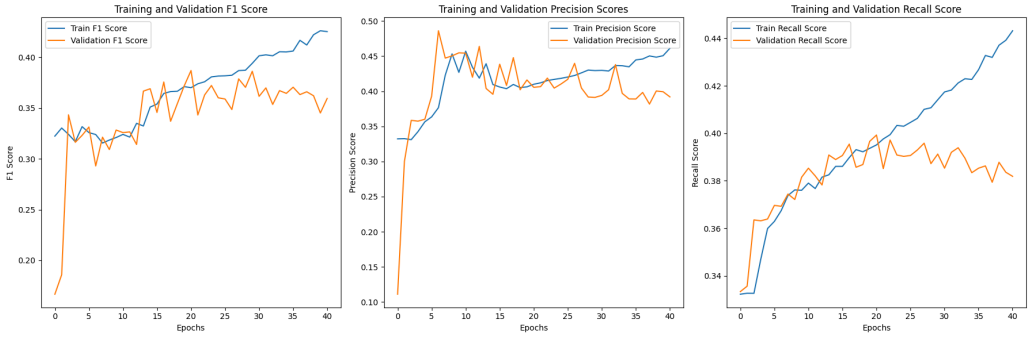
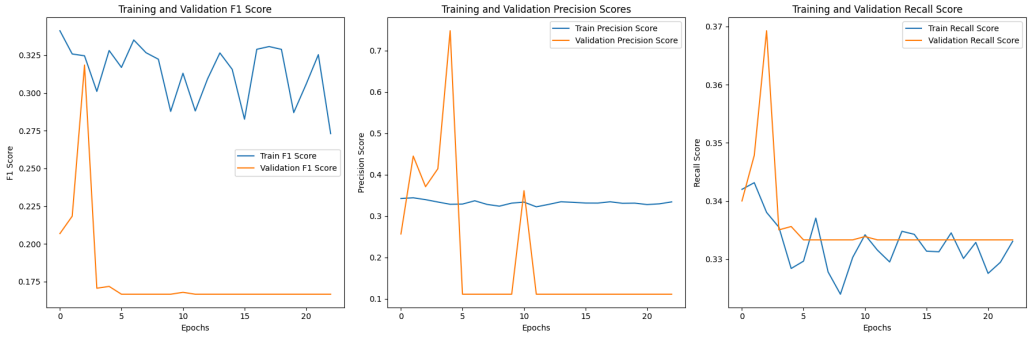
1 layer: With 1 layer we achieved decent performance with the highest mean training f1 score (0.43) among all configurations. A single-layer model can capture patterns effectively, but there's a notable gap between training and validation scores, indicating some overfitting.

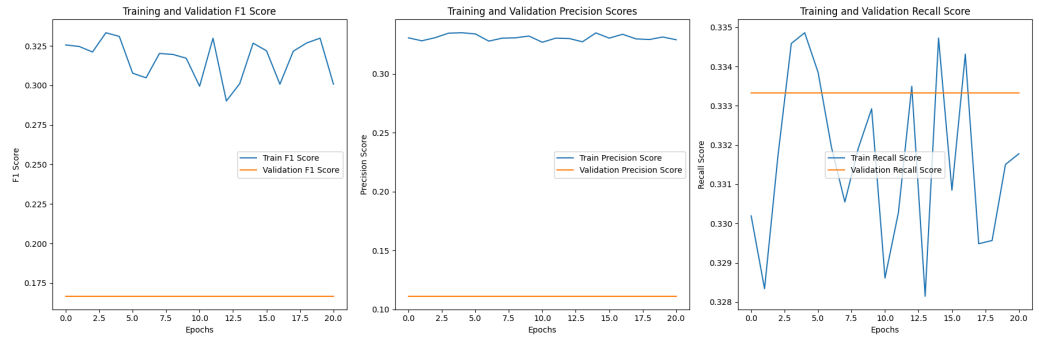
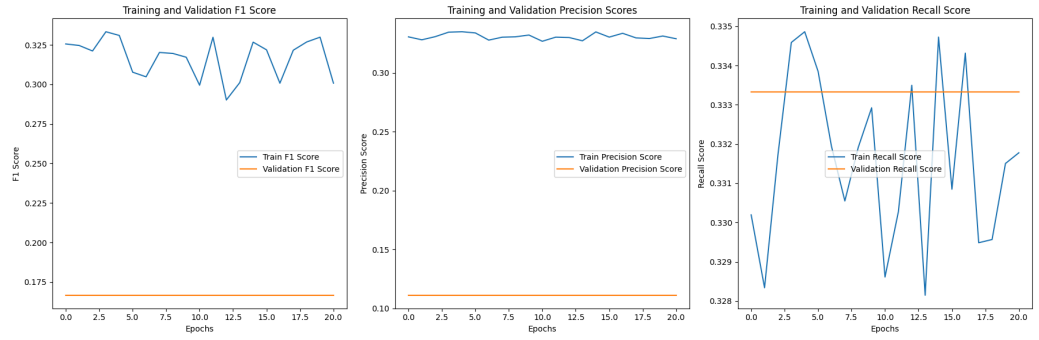
2 layers: Slightly lower mean training f1 score (0.38) but comparable validation f1 score (0.35) to the 1-layer model. The 2-layer model seems to balance training and validation scores better, possibly capturing more complex patterns than the one-layer model.

4 layers: Similar validation f1 score to the 2-layer model, but a slightly lower training f1 score. Increasing layers to four doesn't seem to offer significant benefits in terms of validation performance.

7, 9, 10 layers:
Performance: Significant drop in both training and validation f1, precision, and recall scores. High number of layers leads to significantly worse performance. This could be due to increased model complexity, making it difficult for the model to learn effectively (overfitting on training data or not generalizing well).

Below we can observe the plots for f1, precision and recall scores.

F1 - Precision - Recall	Layers
	1
	2
	4
	7

	9
	10

Mean values of training-validation f1, precision, recall scores

Mean training f1	Mean validation f1	Mean training precision	Mean validation precision	Mean training recall	Mean validation recall	Hidden size
0.38	0.36	0.41	0.40	0.40	0.39	**2**
0.32	0.17	0.33	0.11	0.33	0.33	7
0.37	0.34	0.41	0.40	0.39	0.38	4
0.32	0.17	0.33	0.12	0.33	0.33	10
0.32	0.17	0.33	0.11	0.33	0.33	9
0.43	0.36	0.45	0.39	0.44	0.38	1

Experiments with hidden size

We conducted experiments for number of hidden size equal to 36, 39, 72, 88, 104, 179, 191, 233, 239 and 250.

Hidden size equal to 233, 250, 179, 191, 239: We observed generally higher training f1 scores, indicating stronger learning capacity. Although the notable gaps between training and validation scores suggest a tendency to overfit, especially for the highest hidden sizes.

Hidden size equal to 88: According to optuna framework that is the optimal hidden size. We can observe balanced training and validation f1 scores (0.38 and 0.35, respectively). Also better generalization is indicated by closer training and validation scores.

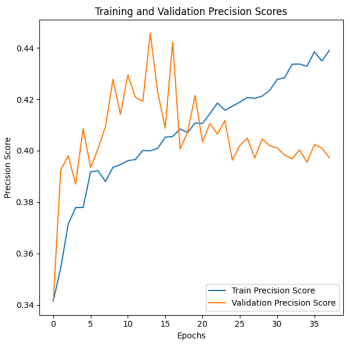
Hidden Size equal to 72, 104, 39: Here we observed moderate training and validation f1 scores, not far from those observed in the optimal hidden size. Additionally the consistent performance across both training and validation, suggesting good model capacity without significant overfitting.

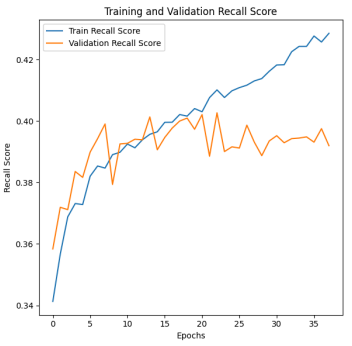
Hidden size 36: Lower training F1 score (0.37) and comparable validation F1 score (0.35) to moderate hidden sizes.
Limited Capacity: Potentially limited learning capacity due to smaller hidden size.
Early Stopping: Later stopping (40 epochs) could indicate slower convergence due to limited capacity.

Overall larger hidden sizes increase the model's capacity to learn complex patterns but also raise the risk of overfitting, as seen in the larger gap between training and validation scores. A hidden size of 88 provides a good balance between learning capacity and generalization ability. This size allows the model to capture patterns effectively without overfitting significantly.

F1 - Precision - Recall	**Hidden size**
	36

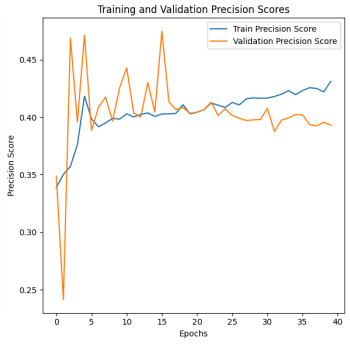

Training and Validation F1 Score
Train F1 Score
Validation F1 Score
F1 Score
Epochs

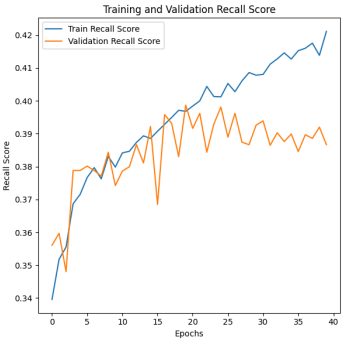

Training and Validation Precision Scores
Train Precision Score
Validation Precision Score
Precision Score
Epochs


Training and Validation Recall Score
Train Recall Score
Validation Recall Score
Recall Score
Epochs

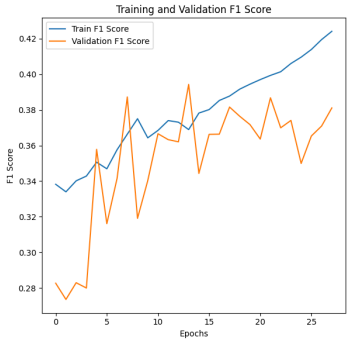
39

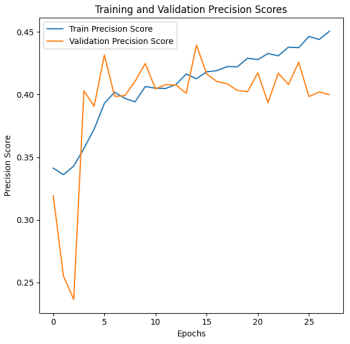

Training and Validation F1 Score
Train F1 Score
Validation F1 Score
F1 Score
Epochs

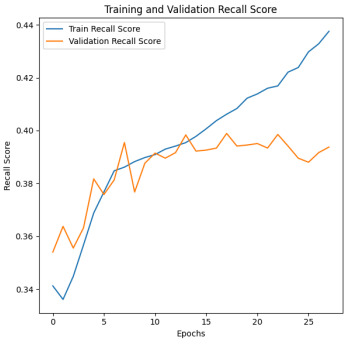

Training and Validation Precision Scores
Train Precision Score
Validation Precision Score
Precision Score
Epochs


Training and Validation Recall Score
Train Recall Score
Validation Recall Score
Recall Score
Epochs

72

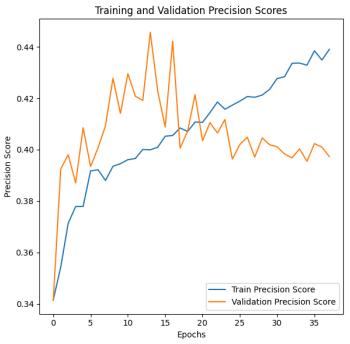

Training and Validation F1 Score
Train F1 Score
Validation F1 Score
F1 Score
Epochs

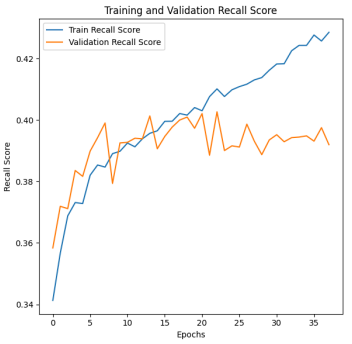

Training and Validation Precision Scores
Train Precision Score
Validation Precision Score
Precision Score
Epochs


Training and Validation Recall Score
Train Recall Score
Validation Recall Score
Recall Score
Epochs

88


Training and Validation F1 Score
Train F1 Score
Validation F1 Score
F1 Score
Epochs


Training and Validation Precision Scores
Train Precision Score
Validation Precision Score
Precision Score
Epochs

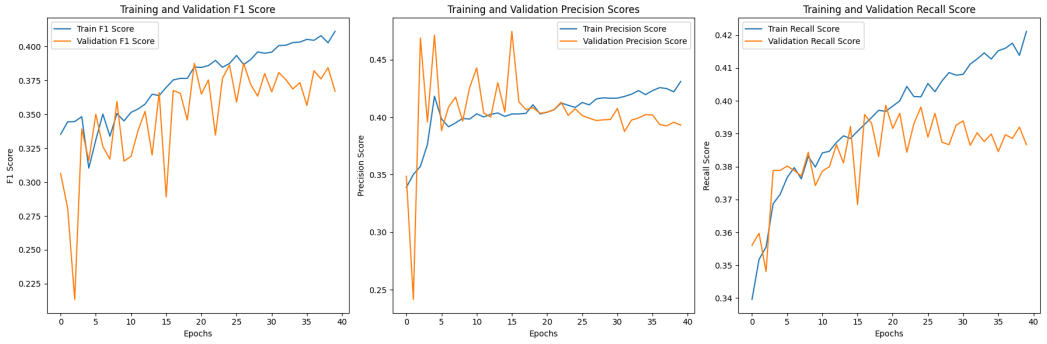
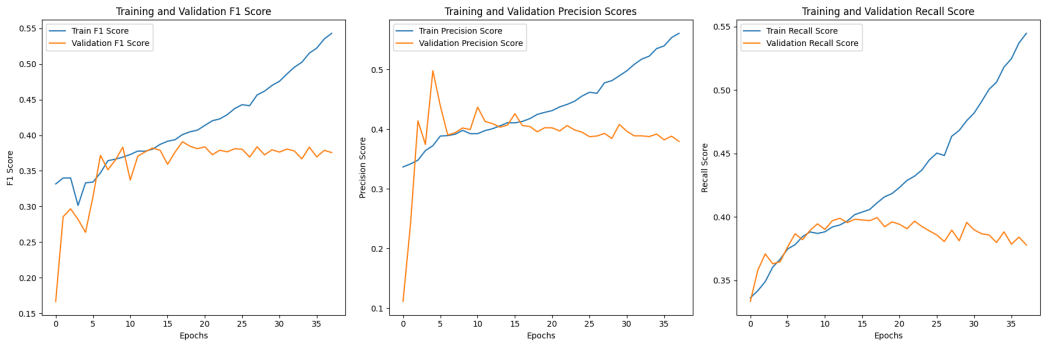
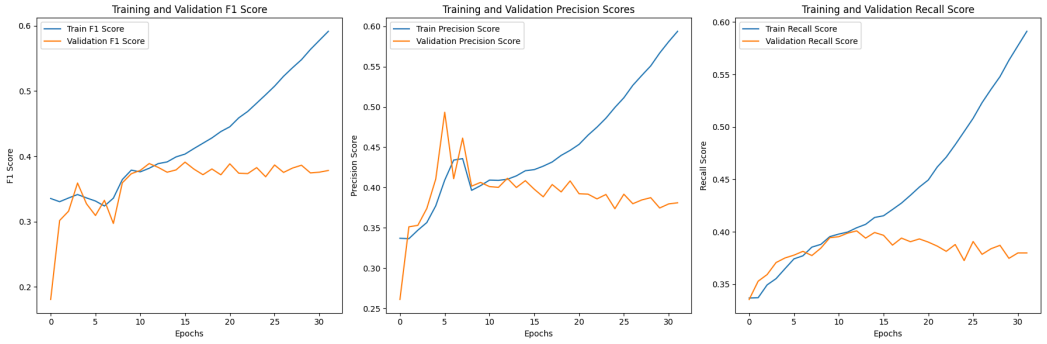
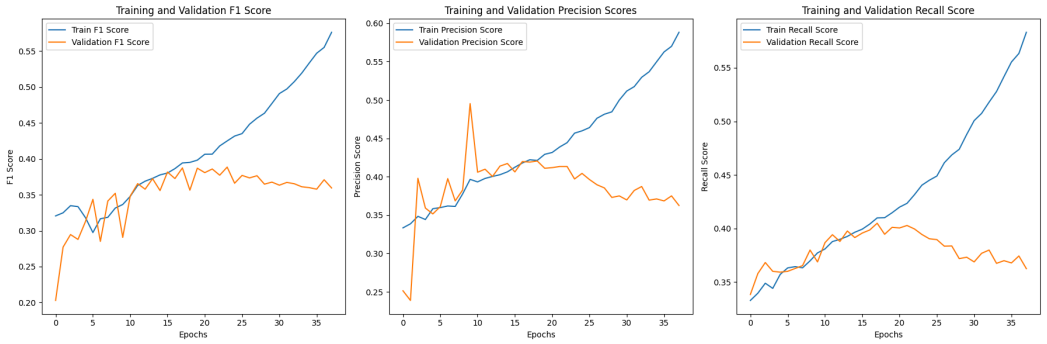

Training and Validation Recall Score
Train Recall Score
Validation Recall Score
Recall Score
Epochs

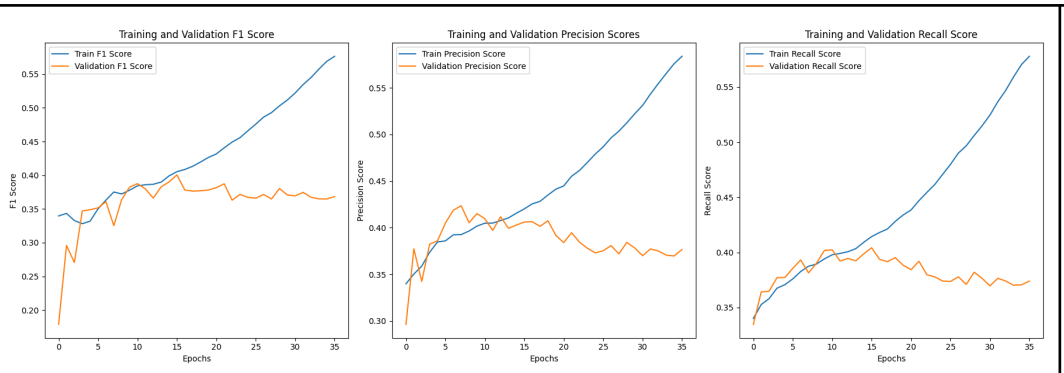
104

Training and Validation F1 Score
Train F1 Score
Validation F1 Score
F1 Score
Epochs

Training and Validation Precision Scores
Train Precision Score
Validation Precision Score
Precision Score
Epochs

Training and Validation Recall Score
Train Recall Score
Validation Recall Score
Recall Score
Epochs

	179
	191
	233
	239

	250

Mean values of training-validation f1, precision, recall scores

Mean training f1	Mean validation f1	Mean training precision	Mean validation precision	Mean training recall	Mean validation recall	Hidden size
0.42	0.36	0.43	0.39	0.43	0.38	250
0.43	0.36	0.45	0.39	0.44	0.38	233
0.38	0.35	0.41	0.39	0.40	0.39	88
0.37	0.35	0.40	0.40	0.39	0.38	36
0.43	0.35	0.46	0.39	0.45	0.38	179
0.39	0.36	0.41	0.39	0.40	0.39	72
0.42	0.36	0.44	0.39	0.43	0.39	191
0.41	0.35	0.44	0.39	0.43	0.38	239
0.39	0.37	0.41	0.41	0.40	0.39	39
0.38	0.36	0.41	0.40	0.40	0.39	104

Experiments with the cell type

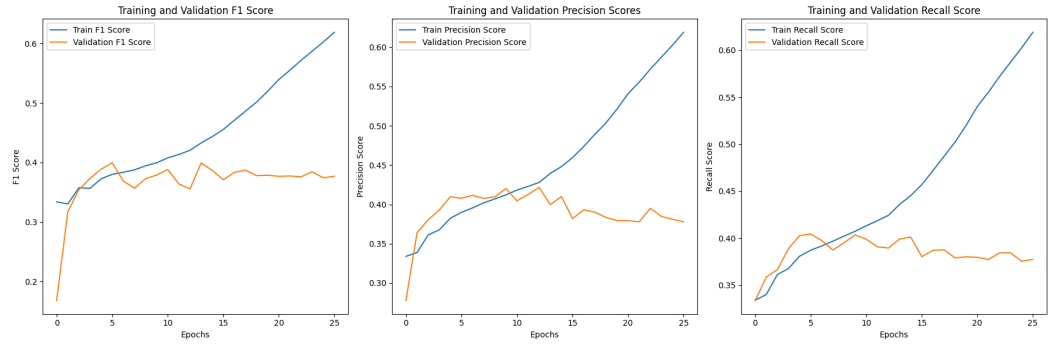
Next we experimented with different cell types like LSTM and GRU.

Performance similarity: Both LSTM and GRU models seem to perform similarly in terms of f1 score, precision, and recall on your validation data. This suggests that for our specific dataset and task, both cell types are capable of capturing the relevant patterns in the data to a similar degree.

Slight edge for LSTM: The LSTM model has a marginally higher mean validation f1 score and precision compared to the GRU model. Although this difference is small, it indicates that the LSTM might be slightly better at balancing precision and recall for our dataset.

Early stopping: The LSTM model was trained for a slightly higher number of epochs (30) before early stopping was triggered, compared to the GRU model (27 epochs). This could imply that the LSTM model was able to continue improving for a bit longer, possibly due to its more complex architecture which includes an additional gate compared to GRU.

Contextual suitability: The LSTM's additional gate (forget gate) gives it more control over the flow of information, which can be an advantage in tasks where understanding the context and retaining information over longer sequences is crucial. If the task involves long-term dependencies LSTM is more suitable.

F1 - Precision - Recall	**Cell type**
	LSTM
	GRU

Mean values of training-validation f1, precision, recall scores

Mean training f1	Mean validation f1	Mean training precision	Mean validation precision	Mean training recall	Mean validation recall	Cell type
0.38	0.36	0.40	0.39	0.40	0.39	LSTM
0.44	0.37	0.45	0.39	0.45	0.39	GRU

Experiments with skip connection

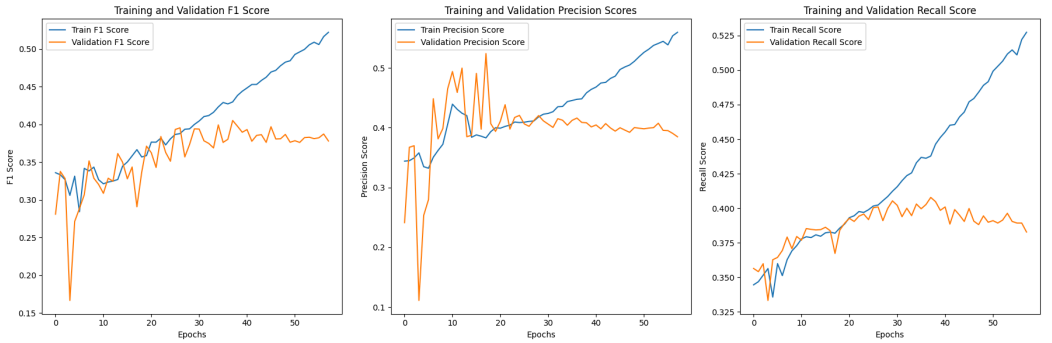
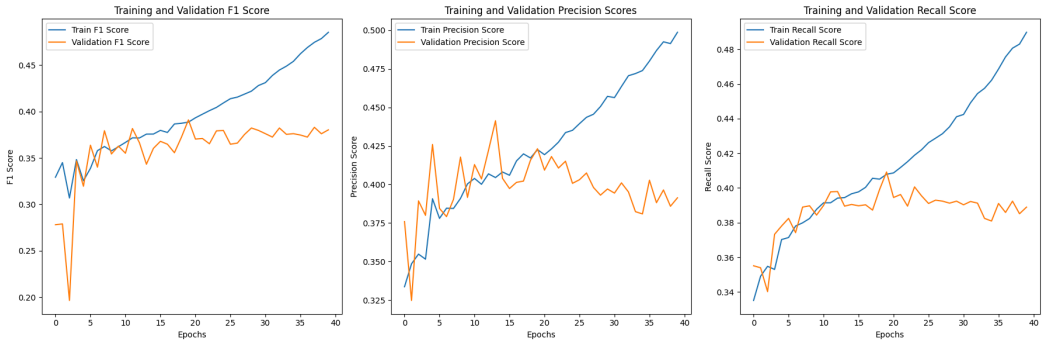
Skip connections: Skip connections, also known as residual connections, are techniques used in deep learning where the output of one layer is added to the output of a later layer. This allows the network to learn an identity function, ensuring that the deeper layer can perform at least as well as the shallower layer. It also helps to mitigate the vanishing gradient problem by providing an alternative pathway for gradients during backpropagation.

The results from our experiments indicate that using skip connections in our model does not lead to improved performance for this specific task. In fact, optuna framework which we used suggested that not using skip connections yields better results. Skip connections technique may not improve the model for the following reasons:

Nature of data: The dataset for tweets sentiment analysis might not have long-term dependencies that require preserving information from earlier layers or time steps. Hence, skip connections may not be particularly beneficial.

Model complexity: Skip connections add more complexity to the model. If the dataset or the problem at hand is not very complex, this additional complexity might not be necessary and could even hinder performance.

Risk of overfitting: More complex models with additional pathways for information flow might overfit on the training data, leading to lower generalization performance on validation or test sets (although we did not observe something like that to our model).

F1 - Precision - Recall	Skip conne-ctions
	True
	False

Mean values of training-validation f1, precision, recall scores

Mean training f1	Mean validation f1	Mean training precision	Mean validation precision	Mean training recall	Mean validation recall	Skip connectio-ns
0.35	0.34	0.42	0.42	0.39	0.38	True
0.40	0.36	0.42	0.40	0.41	0.39	False

Experiments with gradient clipping

Gradient clipping is a technique used to prevent the so-called “exploding gradients” problem in neural networks, which can occur when large error gradients accumulate and result in very large updates to neural network model weights during training.

At an extreme, the values of weights can become so large as to overflow and result in NaN values, often referred to as an “exploding” network. More commonly, the network can become unstable, and the loss value used to train the network can oscillate during training, causing the training process to fail.

This is particularly problematic for recurrent neural networks (RNNs), like LSTMs and GRUs, when trained on long sequences, as the accumulation of gradients through many time steps can be very large.

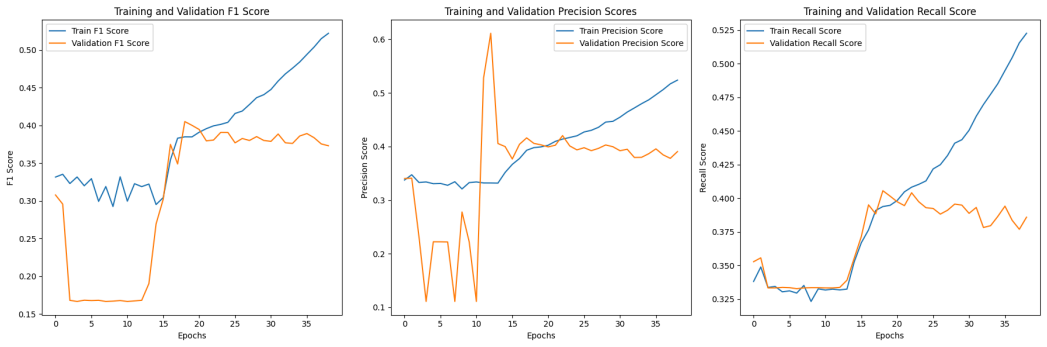
The concept behind gradient clipping is pretty simple: by limiting (or “clipping”) the maximum value of the gradient, we can prevent the problems caused by large gradients without significantly impacting the learning process.

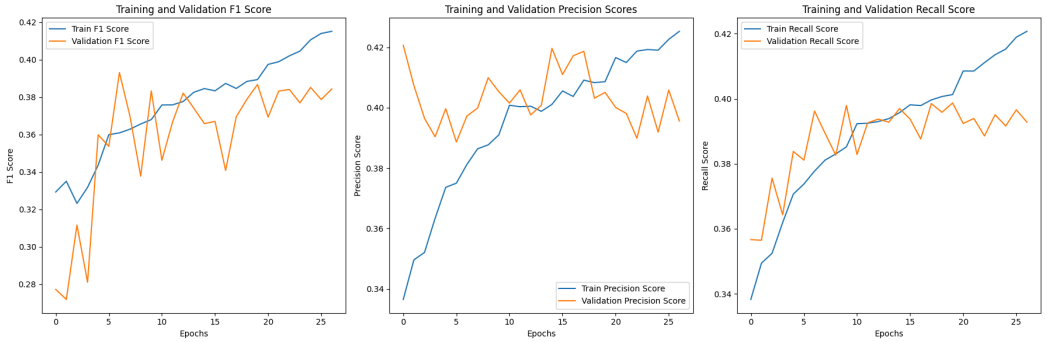
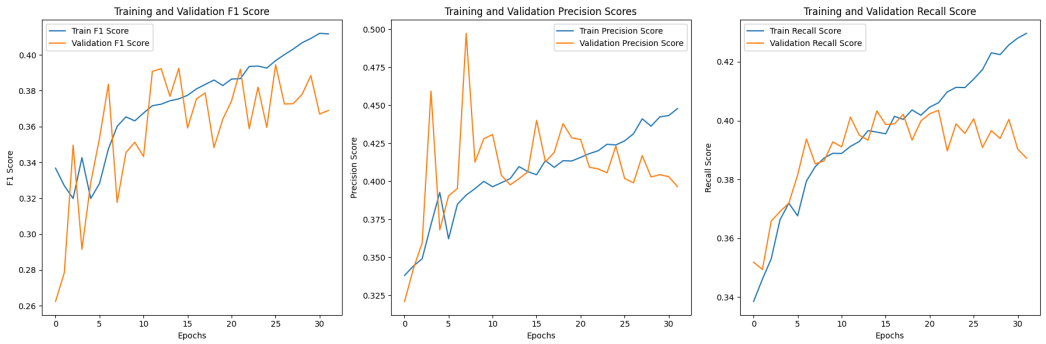
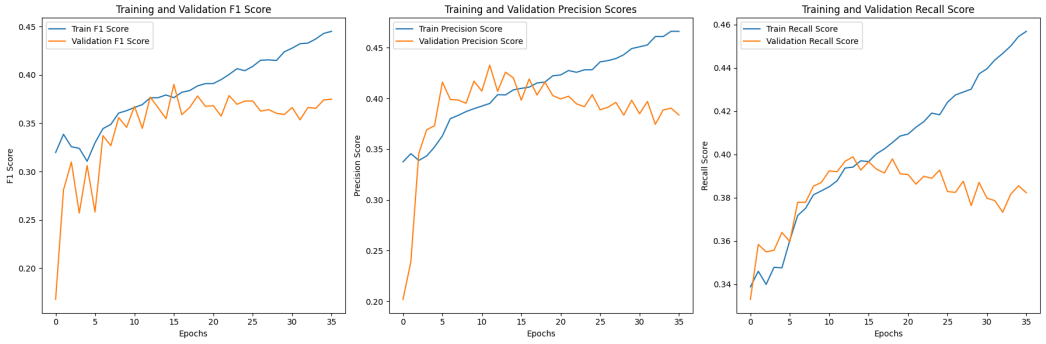
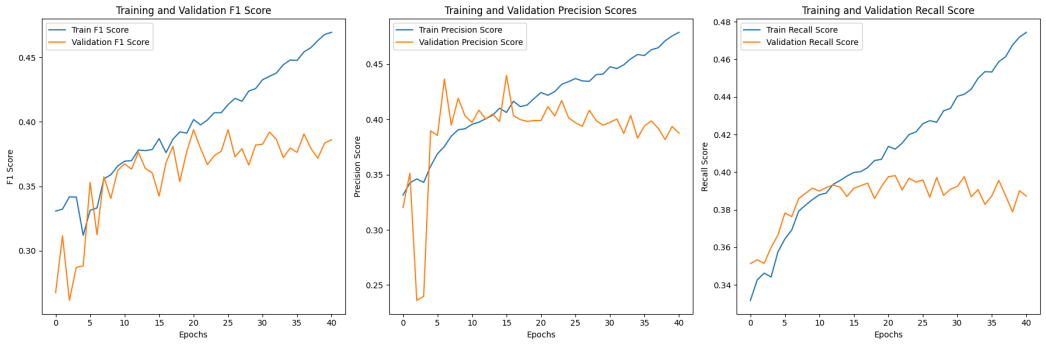
Gradient clipping comes in two main types:

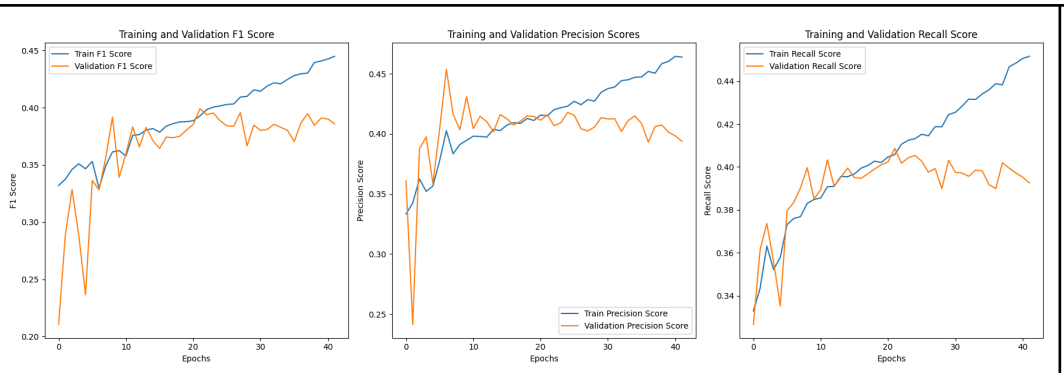
- **Value Clipping**: This involves clipping the gradients when their absolute value exceeds a predefined threshold. The result is that the gradient vector’s direction may be changed.
- **Norm Clipping**: This involves scaling the whole gradient if the L2 norm of the gradient vector exceeds a certain threshold. This method preserves the direction of the gradient and is generally the preferred method for gradient clipping.

Stability across different clipping values:
Our experiments demonstrated a relatively stable performance across a range of gradient clipping values, as indicated by the mean validation f1 scores. This stability suggests that our model is somewhat robust to the specific choice of gradient clipping threshold within the tested range.

Optimal clipping value:
We observed that using **None** is the best option, which indicates that for our specific model architecture, dataset, and training setup, the issue of exploding gradients might not be as critical. This could mean either the problem doesn't occur under our current configurations or the optimizer and the inherent model architecture (such as LSTM's gating mechanisms) are sufficient to mitigate the issue.

F1 - Precision - Recall	**Gradient clipping**
	0.5

 Training and Validation F1 Score Train F1 Score Validation F1 Score F1 Score Epochs Training and Validation Precision Scores Train Precision Score Validation Precision Score Precision Score Epochs Training and Validation Recall Score Train Recall Score Validation Recall Score Recall Score Epochs	2.4
 Training and Validation F1 Score Train F1 Score Validation F1 Score F1 Score Epochs Training and Validation Precision Scores Train Precision Score Validation Precision Score Precision Score Epochs Training and Validation Recall Score Train Recall Score Validation Recall Score Recall Score Epochs	4.3
 Training and Validation F1 Score Train F1 Score Validation F1 Score F1 Score Epochs Training and Validation Precision Scores Train Precision Score Validation Precision Score Precision Score Epochs Training and Validation Recall Score Train Recall Score Validation Recall Score Recall Score Epochs	6.2
 Training and Validation F1 Score Train F1 Score Validation F1 Score F1 Score Epochs Training and Validation Precision Scores Train Precision Score Validation Precision Score Precision Score Epochs Training and Validation Recall Score Train Recall Score Validation Recall Score Recall Score Epochs	8.1

	None

Mean values of training-validation f1, precision, recall scores

Mean training f1	Mean validation f1	Mean training precision	Mean validation precision	Mean training recall	Mean validation recall	Gradient clipping
0.39	0.31	0.40	0.36	0.40	0.37	0.5
0.38	0.36	0.40	0.40	0.39	0.39	2.4
0.37	0.36	0.41	0.41	0.40	0.39	4.3
0.38	0.35	0.41	0.39	0.40	0.38	6.2
0.40	0.36	0.42	0.39	0.41	0.39	8.1
0.39	0.36	0.41	0.40	0.40	0.39	None

Experiments with dropout

Dropout Regularization for Neural Networks
Dropout is a technique where randomly selected neurons are ignored during training. They are “dropped out” randomly. This means that their contribution to the activation of downstream neurons is temporarily removed on the forward pass, and any weight updates are not applied to the neuron on the backward pass.

As a neural network learns, neuron weights settle into their context within the network. Weights of neurons are tuned for specific features, providing some specialization. Neighboring neurons come to rely on this specialization, which, if taken too far, can result in a fragile model too specialized for the training data. This reliance on context for a neuron during training is referred to as complex co-adaptations.

If neurons are randomly dropped out of the network during training, other neurons will have to step in and handle the representation required to make predictions for the missing neurons. This is believed to result in multiple independent internal representations being learned by the network.

The effect is that the network becomes less sensitive to the specific weights of neurons. This, in turn, results in a network capable of better generalization and less likely to overfit the training data.

How to Dropout
Dropout is implemented per-layer in a neural network. It can be used with most types of layers, such as dense fully connected layers, convolutional layers, and recurrent layers such as the long short-term memory network layer.

Dropout may be implemented on any or all hidden layers in the network as well as the visible or input layer. It is not used on the output layer.

A new hyperparameter is introduced that specifies the probability at which outputs of the layer are dropped out, or inversely, the probability at which outputs of the layer are retained. The interpretation is an implementation detail that can differ from paper to code library.

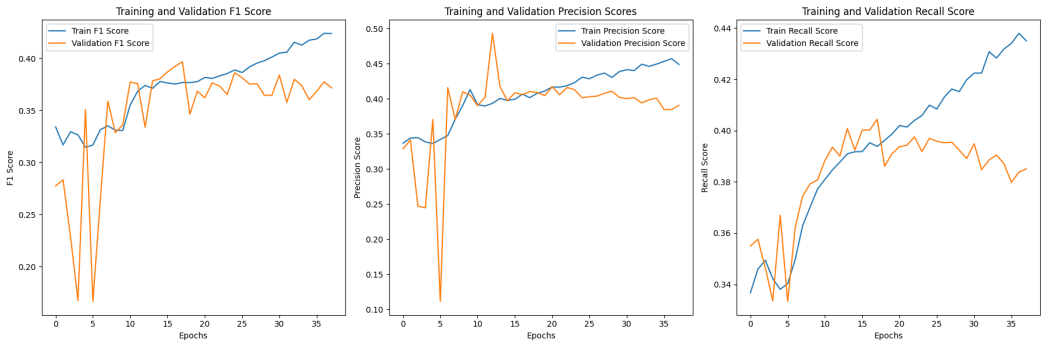
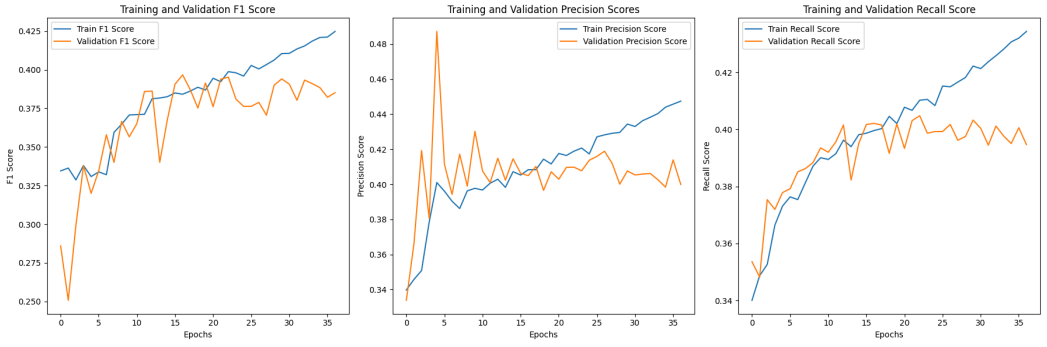
A common value is a probability of 0.5 for retaining the output of each node in a hidden layer and a value close to 1.0, such as 0.8, for retaining inputs from the visible layer.

In the simplest case, each unit is retained with a fixed probability p independent of other units, where p can be chosen using a validation set or can simply be set at 0.5, which seems to be close to optimal for a wide range of networks and tasks. For the input units, however, the optimal probability of retention is usually closer to 1 than to 0.5.

Dropout is not used after training when making a prediction with the fit network. The weights of the network will be larger than normal because of dropout. Therefore, before finalizing the network, the weights are first scaled by the chosen dropout rate. The network can then be used as per normal to make predictions.

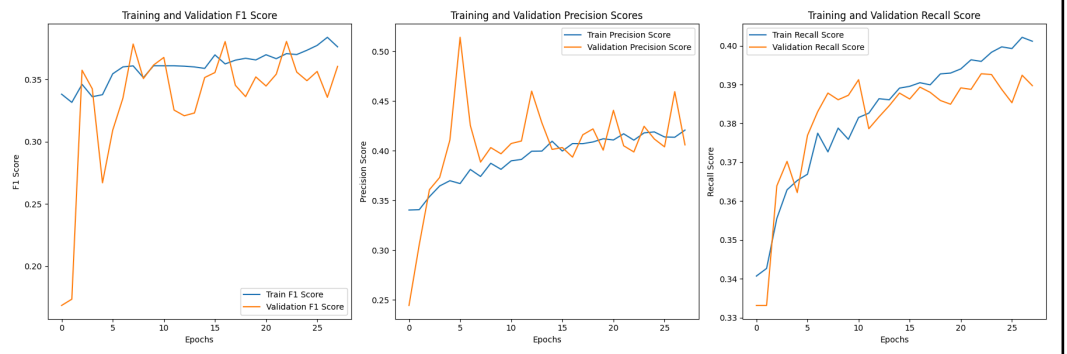
If a unit is retained with probability p during training, the outgoing weights of that unit are multiplied by p at test time. The rescaling of the weights can be performed at training time instead, after each weight update at the end of the mini-batch. This is sometimes called “inverse dropout” and does not require any modification of weights during training. Both the Keras and PyTorch deep learning libraries implement dropout in this way.

From the following results we can see that our model **does not achieve** much higher scores. From our experiments we observed that we get the best results **using dropout** equal to 0.1.

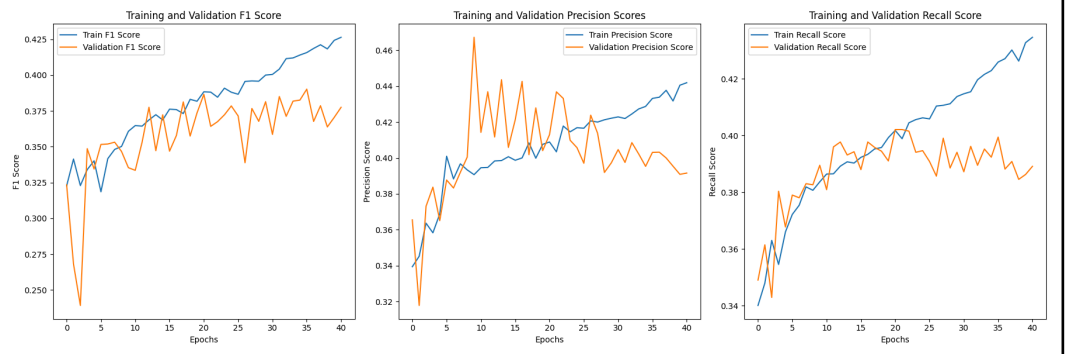
F1 - Precision - Recall	**Gradient clipping**
	0.0
	0.1


Training and Validation F1 Score
Train F1 Score
Validation F1 Score
F1 Score
Epochs
Training and Validation Precision Scores
Train Precision Score
Validation Precision Score
Precision Score
Epochs
Training and Validation Recall Score
Train Recall Score
Validation Recall Score
Recall Score
Epochs

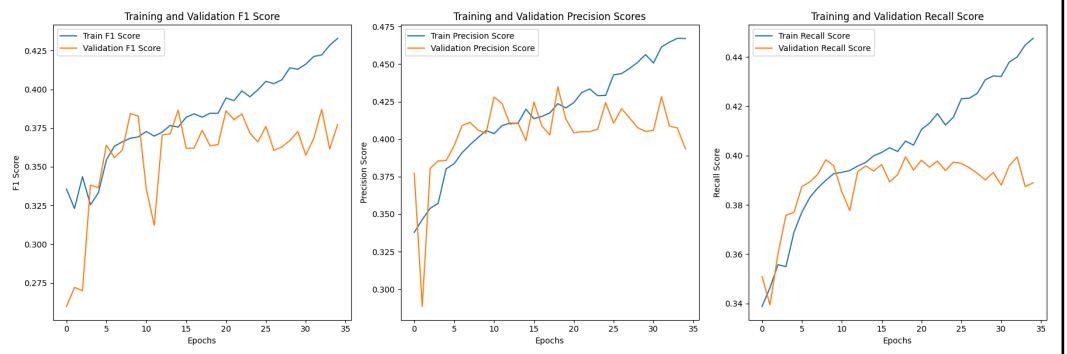
0.2


Training and Validation F1 Score
Train F1 Score
Validation F1 Score
F1 Score
Epochs
Training and Validation Precision Scores
Train Precision Score
Validation Precision Score
Precision Score
Epochs
Training and Validation Recall Score
Train Recall Score
Validation Recall Score
Recall Score
Epochs

0.3


Training and Validation F1 Score
Train F1 Score
Validation F1 Score
F1 Score
Epochs
Training and Validation Precision Scores
Train Precision Score
Validation Precision Score
Precision Score
Epochs
Training and Validation Recall Score
Train Recall Score
Validation Recall Score
Recall Score
Epochs

0.4


Training and Validation F1 Score
Train F1 Score
Validation F1 Score
F1 Score
Epochs
Training and Validation Precision Scores
Train Precision Score
Validation Precision Score
Precision Score
Epochs
Training and Validation Recall Score
Train Recall Score
Validation Recall Score
Recall Score
Epochs

0.5

Mean values of training-validation f1, precision, recall scores

Mean training f1	Mean validation f1	Mean training precision	Mean validation precision	Mean training recall	Mean validation recall	Dropout
0.37	0.35	0.41	0.38	0.39	0.38	0.0
0.38	0.37	0.41	0.41	0.40	0.39	0.1
0.39	0.34	0.41	0.38	0.40	0.38	0.2
0.36	0.33	0.39	0.40	0.38	0.38	0.3
0.38	0.36	0.41	0.40	0.40	0.39	0.4
0.38	0.36	0.42	0.40	0.40	0.39	0.5

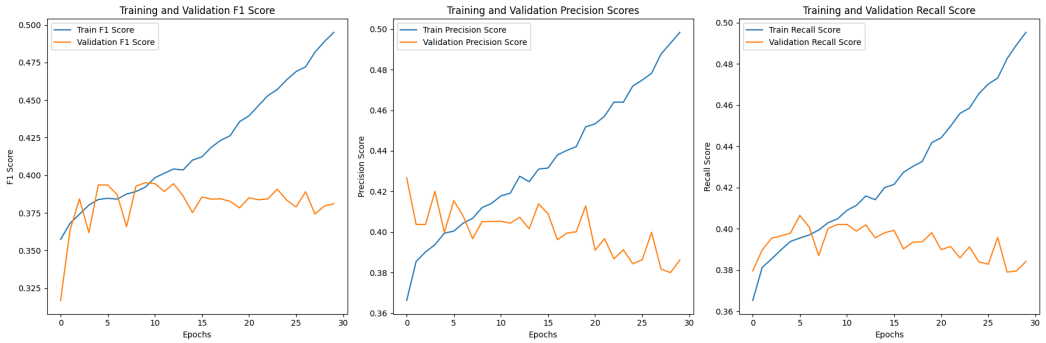
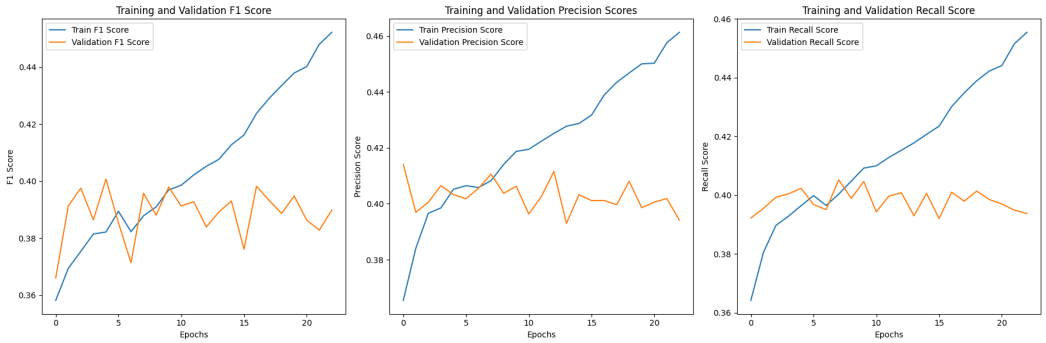
Use of attention

Machine learning-based attention is a mechanism which intuitively mimicks cognitive attention. It calculates "soft" weights for each word, more precisely for its embedding, in the context window. These weights can be computed either in parallel (such as in transformers) or sequentially (such as recurrent neural networks). "Soft" weights can change during each runtime, in contrast to "hard" weights, which are (pre-)trained and fine-tuned and remain frozen afterwards.

Attention was developed to address the weaknesses of leveraging information from the hidden outputs of recurrent neural networks. Recurrent neural networks favor more recent information contained in words at the end of a sentence, while information earlier in the sentence is expected to be attenuated. Attention allows the calculation of the hidden representation of a token equal access to any part of a sentence directly, rather than only through the previous hidden state.

Earlier uses attached this mechanism to a serial recurrent neural network's language translation system, but later uses in Transformers large language models removed the recurrent neural network and relied heavily on the faster parallel attention scheme.

We used two models to utilize attention **RNNModelAttention** and **RNNModelWithEnhancedAttention**. Both models achieved similar results with each other and slightly **better** in comparison with our previous experiments.

F1 - Precision - Recall	Attention model
	RNNModelAttention
	RNNModelWithEnhancedAttention

Mean values of training-validation f1, precision, recall scores

Mean training f1	Mean validation f1	Mean training precision	Mean validation precision	Mean training recall	Mean validation recall	Attention model
0.42	0.38	0.43	0.40	0.43	0.39	RNNModel Attention
0.41	0.39	0.42	0.40	0.41	0.40	RNNModel WithEnhancedAttention

Table of trials

Below is the table of trials where you can observe the values which led our model to perform slightly better, even though the change was insignificant.

Hyperparameter	Value	F1 train	F1 validation
layers	2	0.38	0.36
Hidden size	88	0.37	0.35
cell type	LSTM	0.38	0.36
skip connections	False	0.40	0.36
Gradient clipping	None	0.39	0.36
Attention model	RNNModelWithEnhancedAttention	0.41	0.39

Hyper-parameter tuning

For the hyper-parameter tuning we conducted experiments with the layers, the hidden size, the cell type, the skip connections and the gradient clipping,. We found that the values for the hyper-parameters that led the model to perform slightly better were 2 layers, 88 for the hidden size, LSTM for the cell type, False for the skip connections technique, None for the gradient clipping and for the attention models we experimented with, the RNNModelWithEnhancedAttention yielded better results .
In all our trials, the evident **overfitting**, since almost every time the train score is greater than the validation score, of our model suggests that the unique characteristics of tweet data present a unique challenge. This overfitting may derive from the inherently noisy and sparse nature of text data derived from social media, where slang, abbreviations, and diverse linguistic expressions prevail. Such characteristics can lead the model to learn patterns specific to the training set that do not generalize well to unseen data. Additionally, the limitation of a relatively small dataset could exacerbate the issue, as the model might not be exposed to a sufficiently varied representation of the problem space.

Optimization

For the optimization we used the Adam optimizer. It was evident for every experiment that our model is very unstable from the fluctuation of our metrics.

Evaluation

For evaluation of every hyperparameter and for every experiment we were plotting the learning curves that was showing the f1, recall and precision training and validation scores. We focused mostly on increasing the f1 score since we are building a classifier.

f1: F1 Score is a measure that combines recall and precision. As we have seen there is a trade-off between precision and recall, F1 can therefore be used to measure how effectively our models make that trade-off. One important feature of the F1 score is that the result is zero if any of the components (precision or recall) fall to zero. Thereby it penalizes extreme negative values of either component.

Precision: Precision is a metric that gives you the proportion of true positives to the amount of total positives that the model predicts. It answers the question “Out of all the positive predictions we made, how many were true?”

Recall: Recall focuses on how good the model is at finding all the positives. Recall is also called true positive rate and answers the question “Out of all the data points that should be predicted as true, how many did we correctly predict as true?”

Results and overall analysis

Best trial

The best trial was for the number of **layers** equal to **2**, **hidden size** equal to **88**, **cell type** equal to **LSTM** and without using **skip connections** and **gradient clipping**. Also we observed that with using attention we can score higher results.

Comparison with the first project

In comparison with the first project where we used LogisticRegression of scikit-learn, we achieved a slightly higher f1 training and validation score.

Comparison with the second project

In comparison with the second project where we used PyTorch neural networks, we achieved similar results regarding the f1 training and validation score using RNN.

Bibliography

- https://www.kaggle.com/code/mlwhiz/multiclass-text-classification-pytorch
- https://turbolab.in/text-classification-with-keras-and-glove-word-embeddings/
- https://coderzcolumn.com/tutorials/artificial-intelligence/how-to-use-glove-embeddings-with-pytorch
- https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Text-Classification/blob/master/utils.py
- https://www.analyticsvidhya.com/blog/2020/01/first-text-classification-in-pytorch/
- https://en.wikipedia.org/wiki/Early_stopping
- https://www.sabrepc.com/blog/Deep-Learning-and-AI/Epochs-Batch-Size-Iterations
- https://en.wikipedia.org/wiki/Activation_function
- https://machinelearningmastery.com/using-optimizers-from-pytorch/
- https://stackoverflow.com/questions/42711144/how-can-i-install-torchtext
- https://medium.com/@nerdjock/deep-learning-course-lesson-10-6-gradient-clipping-694dbb1cca4c
- https://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/
- https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/
- https://en.wikipedia.org/wiki/Attention_(machine_learning)