Model 1

The model we chose to implement for this part is KNN with K=1.

We chose to embed the words using the following pre-trained embedding model: “word2vec-google-news-300” - Word2Vec embedding model of 300 dimensions.

If a word has no representation in this embedding model, we chose to embed it as a zero vector (with the same dimension as our embedding model).

After training the model on the train set (‘train.tagged’), we evaluated it both on the train set and on the test set (‘dev.tagged’), and received the following F1 score:

```
F1 Score for Model 1 on 'train.tagged': 0.811
```

```
F1 Score for Model 1 on 'dev.tagged': 0.511
```

(*) As we can see, we met the requirement of F1_score > 0.5.

Model 2

The model we chose to implement for this part is Fully Connected NN with the following architecture:

- Input layer of size 300 (the dimension of the embedded word vectors).
- One hidden layer with 10 units.
- Output layer of size 1 (since it is a binary classification task, and we chose to use the Sigmoid function to get only the probability of the positive label).

In addition, we used the following properties in the training process:

- Binary Cross Entropy loss function.
- 10 epochs of training.
- Learning rate of 0.01.

We chose to embed the words using the same word embedding protocol as in Model 1.

After training the model on the train set (‘train.tagged’), we evaluated it both on the train set and on the test set (‘dev.tagged’), and received the following F1 score:

```
F1 Score for Model 2 on 'train.tagged': 0.75
```

```
F1 Score for Model 2 on 'dev.tagged': 0.549
```

(*) As we can see, we met the requirement of F1_score > 0.5.

Model 3

The model we chose to implement for this part is LSTM with the following architecture:

- Input layer of size 300 (the dimension of the embedded word vectors).
- Hidden state of size 100.
- A single fully connected layer to convert the hidden state from size 100 to 1 (in order to calculate the positive label probability using the Sigmoid function).
- A single regular (left to right) LSTM layer.

In addition, we used the following properties in the training process:

- Binary Cross Entropy loss function.
- 10 epochs of training.
- Learning rate of 0.01.

We chose to embed the words using the same word embedding protocol as in Model 1.

After training the model on the train set ('train.tagged'), we evaluated it both on the train set and on the test set ('dev.tagged'), and received the following F1 score:

```
F1 Score for Model 3 on 'train.tagged': 0.88
```

```
F1 Score for Model 3 on 'dev.tagged': 0.573
```

(*) As we can see, we met the requirement of F1_score > 0.5.

Competition Model

The model we chose to implement for this part is a combination of several LSTM models:

- LSTM Word2Vec:
    - Input layer of size 300 (the dimension of the Word2Vec embedded vectors).
    - Hidden state of size 100.
    - A single regular (left to right) LSTM layer.

- LSTM GloVe:
    - Input layer of size 200 (the dimension of the GloVe embedded vectors).
    - Hidden state of size 100.
    - A single regular (left to right) LSTM layer.

- LSTM Total:
    - Input layer of size 200 (2 * hidden state size).
    - Hidden state of size 100.
    - A single bidirectional LSTM layer.

- A single fully connected layer to convert the hidden state from size 100 to 1 (in order to calculate the positive label probability using the Sigmoid function).

First, when exploring the data, we noticed that roughly 95% of the labels are negative, resulting in a very imbalanced dataset. In order to try and reduce this imbalance, we decided to remove **whole** sentences which contain **only** negative labels.

For this model we chose to embed the words using two different pre-trained embedding models:

- "word2vec-google-news-300" – Word2Vec embedding model of 300 dimensions.
- "glove-twitter-200" – GloVe embedding model of 200 dimensions.

Moreover, for each embedding model, if a word has no representation in the model, we chose to embed it as a one-hot vector (with the same dimension as the embedding model) based on its structure. After exploring the data, we chose to account for the following words structures:

- Words that start with '@'.
- Words that start with '#'.
- Words that are links (words that start with 'www' or 'http').
- Words that contain letters, punctuations, and numbers.
- Words that contain only letters and punctuations.
- Words that contain only letters and numbers.
- Words that contain only numbers and punctuations.

- Words that contain only letters.
- Words that contain only punctuations.
- Words that contain no letters, punctuations, and numbers.

If a word does not fit any of the above structures, we chose to embed it as a zero vector (with the same dimension as the embedding model).

Thus, in case that a word is not in a certain embedding model, this protocol allows us to utilize its structure, and give our model some degree of information regarding it (unlike the previous embedding protocols which automatically embed the word as a zero vector).

After embedding the words based on the above, for each word, we inserted each of its embeddings as the input to the corresponding LSTM model (LSTM Word2Vec and LSTM GloVe). Then, we concatenated their outputs and inserted it as the input of the LSTM Total model. By following this procedure, we were able to utilize each of the embedding models both simultaneously and effectively.

In addition, we used the following properties in the training process:

- Binary Cross Entropy loss function.
- 10 epochs of training.
- Learning rate of 0.001.

(*) These hyper-parameters were chosen after experimenting with several ones.

After training the model on the train set (‘train.tagged’), we evaluated it both on the train set and on the test set (‘dev.tagged’), and received the following F1 score:

```
F1 Score for the Competition Model on 'train.tagged': 0.703
```

```
F1 Score for the Competition Model on 'dev.tagged': 0.685
```

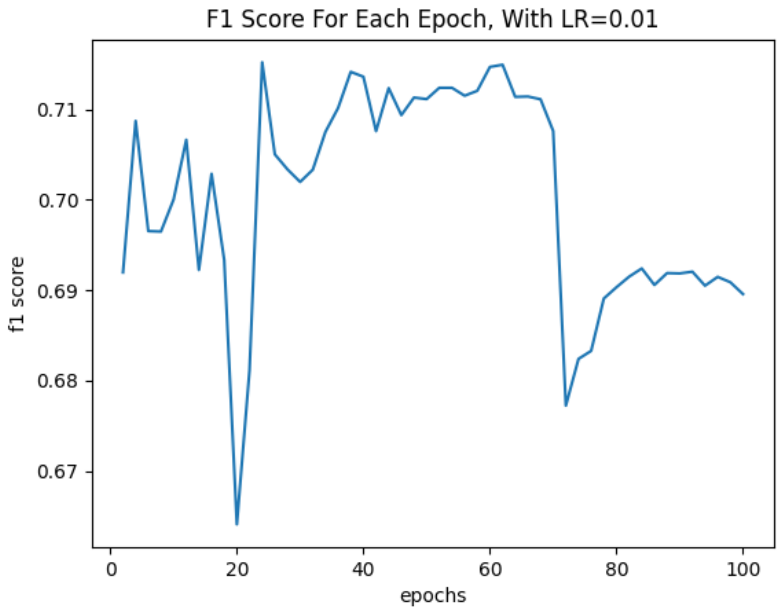
(*) As we can see, we met the requirement of F1_score > 0.5.

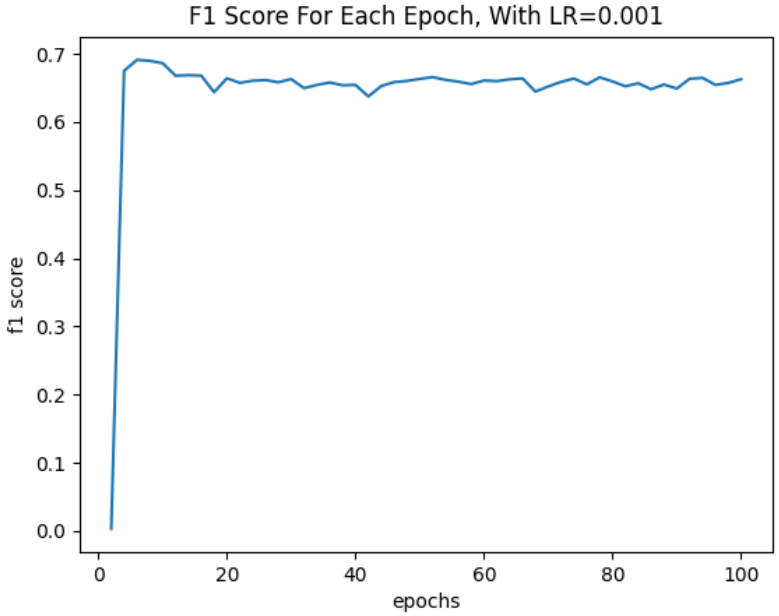
Our final model was trained on **both** ‘train.tagged’ and ‘dev.tagged’.

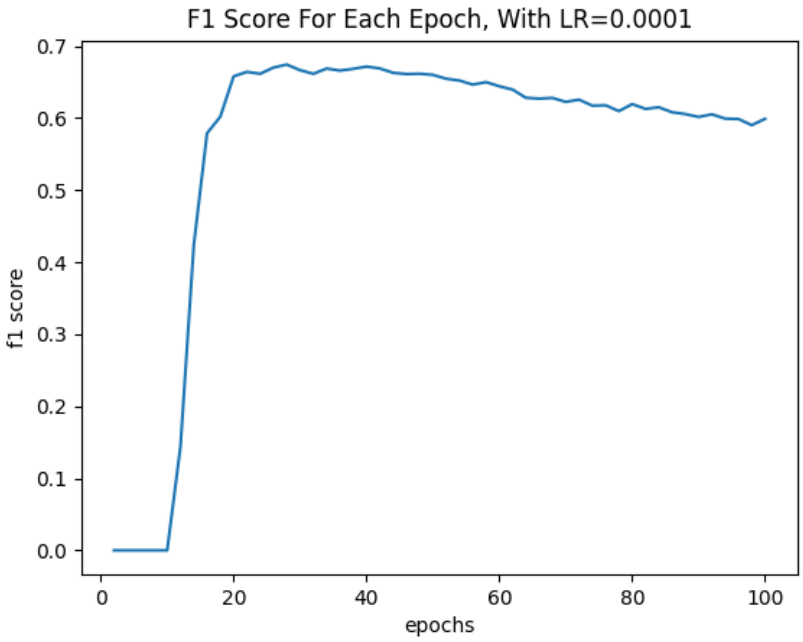
(*) We made sure that training on ‘dev.tagged’ as well indeed improves our results, by using Cross Validation.

Learning Rate Tuning Index

In order to find a good learning rate, we conducted several experiments with the following values: 0.01, 0.001, 0.0001.







As we can see, 0.01 is an extremely unstable value compared to the other values. Moreover, among the remaining two, it appears that 0.001 gains overall better results.

Thus, we will set the learning rate to 0.001.