

Assignment 1

Natural Language Processing - NLP Course

Michael Toker tok@campus.technion.ac.il
Mor Ventura mor.ventura@campus.technion.ac.il

Data Pre-processing	● ● ● ● ● ● ● ● ● ● ●	1
First Model	● ● ● ● ● ● ● ● ● ● ● ● ● ● ●	2
Second Model	● ● ● ● ● ● ● ● ● ● ● ● ● ● ●	3
Competitive Model	● ● ● ● ● ● ● ● ● ● ● ● ● ● ●	5

Data Pre-processing

As a start, we extracted the tagged data from the files in order to arrange them in a data structure of dictionary, where the keys are the embedding representation vectors and the values are the tags.

Embedding:

- Trained model - We used the trained model over Glove - Twitter - 25 dataset as our embedding for model 1 (because of running limitations) and Glove - Twitter - 100 dataset as our embedding for model 2. We wanted to choose the most similar dataset to our corpus.
- Creating list of lists - splitting the data to list of sentences (separated by blank line), each is a list of words (separated by line).
- Padding with start words and end words - we padded each sentence with start word ('morVentura') and end word ('michaelToker') that have unique embedding. We used it for the concatenated embedding form (next bullet).
- Concatenated embedding - We represented each word by its embedding, the previous words embedding and the next ones (with the size of window as a hyperparameter). By that action - we created a unique embedding that relies on a local window (neighborhood).
- Familiar symbols - We were giving any word which contains one of the following symbols: ['@', '#', 'http'] the embedding of those symbols since they are already familiar by the trained model.

- Capital Letter - We concatenated to each embedding additional bit that indicates whether the word started with a capital letter or not. We used our given data as its own tagging as additional feature (self-supervised style).
- Word index in sentence - we add to each word embedding it's index (location) in the sentence. It has only been used in the second model.

Tags:

- Splitting - the data from the tags (by 'tab').
- Binary tagging - 'O' equals 'False' and any other tag equal 'True'

Data Imbalance:

We Noticed that our given corpus is imbalanced - most of the words were given 'False' tag. Our models were having hard time learning with this data - They learned to always predict 'False' tag and by that getting low loss. So we duplicated our minor 'True' tagged data to achieve a uniform distributed data (and shuffled it). Eventually we got better results without it (empirically).

First Model

- SVM - We chose to work with SVM.
- Hyper-parameters - In order to find the best hyper parameters we were using 'Grid-SearchCV'. our best classifier we found is with the following hyper-parameters [kernel = 'rbf', C = 1.25].
- Test and Evaluation - using the built in commands of scikit-learn - 'predict' and 'predict_proba'. f1 metric is calculated, and also the classification report and confusion matrix.

First model - Results:

image - f1

```

done testing first model
start evaluating first model
eval score: 0.9525244449910829
      precision    recall  f1-score   support

   False       0.96       0.99       0.97       15133
    True       0.81       0.41       0.55        1128

 accuracy              0.95       16261
 macro avg       0.88       0.70       0.76       16261
weighted avg       0.95       0.95       0.95       16261

f1: 0.5474794841735053
Sensitivity is 0.41
Specificity is 0.99
PPV is 0.81
NPV is 0.96
Accuracy is 0.95
F1 is 0.55
AUROC is 0.847

```

image - confusion matrix

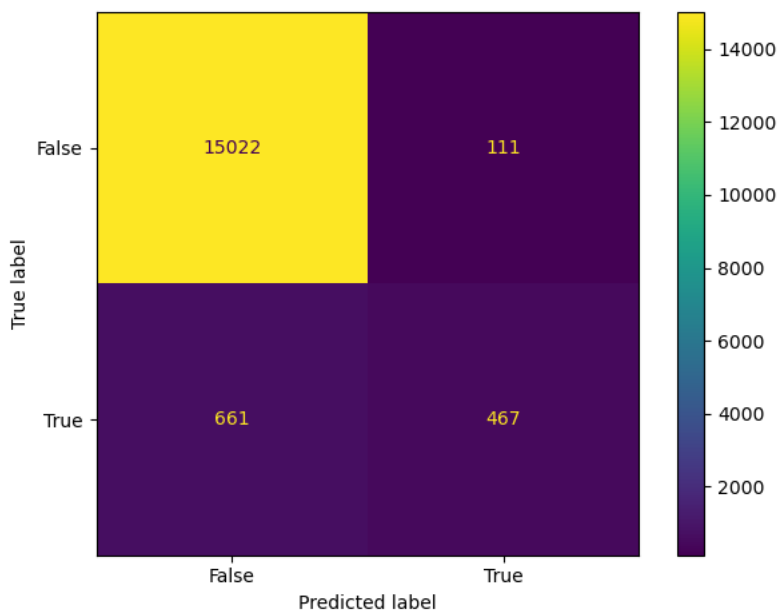


image - classification report

Second Model

Training:

- Dense layers - We used a fully connected neural network with 5 fully connected layers.
- Additional layers - max-Pooling, batch normalization, dropout and Relu.

- Explanation:
 - batch normalization - to keep the training stable.
 - activation ReLU - to let the model learn non linear relations of the input.
 - dropout - to introduce a regularization, that will help the model to learn features that are meaning full and not "memorizing the data".
 - nll loss - The negative log likelihood loss. This loss function get values between zero and one (zero is the best). This loss is high when the prediction is wrong.
- Train - We trained the model and tried various of hyper-parameters (while testing them on the dev dataset). We have tried various combinations of learning rate, batch size, number of epochs and number of hidden layers in the FC layers. Finally we use the a FC NN with 5 layers, max-Pooling, batch normalization and Relu.

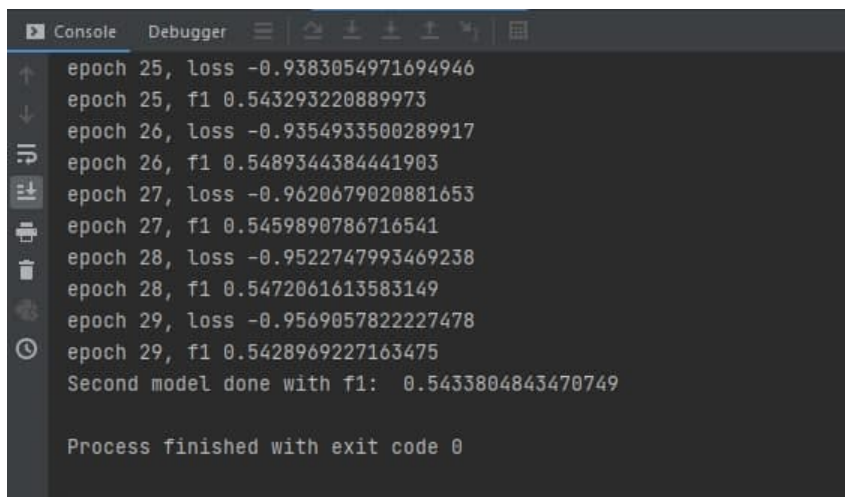
Test:

We used the dev data to evaluate the model after each step in the training presses in order to tune the hyper-parameters. Finally we evaluate the model on the dev data (we never train the model on this data).

We got the best f1 score with:

- learning rate = 0.000048
- number of epochs = 26
- batch size = 1024

We get a f1 score of 0.5433



```

Console  Debugger
epoch 25, loss -0.9383054971694946
epoch 25, f1 0.543293220889973
epoch 26, loss -0.9354933500289917
epoch 26, f1 0.5489344384441903
epoch 27, loss -0.9620679020881653
epoch 27, f1 0.5459890786716541
epoch 28, loss -0.9522747993469238
epoch 28, f1 0.5472061613583149
epoch 29, loss -0.9569057822227478
epoch 29, f1 0.5428969227163475
Second model done with f1: 0.5433804843470749

Process finished with exit code 0

```

Competitive Model

- We chose as our competitive model, our second model - we trained it also on the dev dataset and saved its weights ("comp m2" = "comp m3")