Transcription Factor Binding Prediction

Chanikya Mohan
Department of Computer Science
University of Florida
konyala@ufl.edu

Abstract - The following is report of implementation of a LSTM model for predicting the Transcription Factor for binding and non-binding site of DNA Sequence. The sections of the report include brief introduction to the problem followed by dataset description, data preprocessing. Later, the implementation of model using Keras with Tensorflow backend. The model's parameters, network structure, optimizers etc are all experimented and the results are compared. The final conclusion includes details on the best approach to solve the problem followed by its results.

Keywords- LSTMs, DNA sequences, Embedding Matrix, Keras, TensorFlow, Jupyter, Anaconda

Kaggle Team Name: ChanikyaMohan
GitHub repo:
https://github.com/ChanikyaMohan/DNA_TranscriptionFactorBindingPrediction

1. INTRODUCTION

To make proteins from the DNA, a process called Transcription plays an important role. With the help of transcription the DNA information is used and transcribed into an RNA molecule.

The enzyme transcribing must attach to a promoter region on the DNA sequence. The binding sites in most cases are very near to the promoter regions. In addition to that, they can be also far from the promoter regions. The following problem is about predicting the transcription binding factor give a DNA sequence classifying it into binding or non-binding sites using deep learning.

2. DATA SET & DATA PREPROCESSING

The dataset contains SP1 transcription factor binding and non-binding sites on human chromosome1. There are a total of 2000 training tuples containing equal sets of binding and non-binding regions i.e., 1000 each. The attributes of the dataset are

- id - an integer identifier to uniquely identify a sequence
- Sequence - a DNA sequence containing 14 characters
- Each DNA sequence is made of only fours characters A,C,G,T indicating the respective nucleotides.
- Label - it indicates whether the sequence is a binding site (TFBS) or non-binding site (non-TFBS) indicating with a boolean value 1 and 0 respectively.

The data set is in csv format which has been read using Pandas dataframe.The tuples are grouped together based on their class label i.e., all label 1 tuples together and all tuples with label 0 are together. To get tuples in random order, used sampling method in pandas with a fraction of 1 to get all the tuples. The values of column id are not used in the network so dropped those values.

3. ARCHITECTURE OVERVIEW

Coming the architecture implementation of the network, LSTM is good choice as the prediction is based on sequence data and LSTMs are good in retating long term dependencies and

classifying based on that. The whole architecture is developed, trained and run using Anaconda. The implementation of LSTM is done using Keras with TensorFlow backend. Initially the DNA sequence is split by character level and a vector is generated for each input sequence. By converting the sequence into vector makes it capable of handling by the LSTM. The vectors are generated by using Tokenizer method in Keras. After the vectors has been generated there are embedded into a matrix, using a predefined matrix. The embedded matrix is fed as input to the LSTM which after processing the sequence give a vector as output. This output is fed into a dense layer containing sigmoid function to get a probability value ranging from 0 to 1. The probability value indicates the class which it belongs to (0 or 1) with a cut off of 0.5.

To train the model, the data set is split into train and validation data sets with a split ratio of 0.2. The encoded sequences i.e., embedded matrices are fed into model and trained. The training was done in a batch size of 10 for a total of 15 epochs. The corresponding values of loss and accuracy for both train data and validation data are plotted at the end of each epoch so that the trend of learning can be analyzed. At the end of the 15 epoch the **best achieved accuracy** was **90.5%.** Shown in Image 1.

```
Train on 1600 samples, validate on 400 samples
Epoch 1/15
1600/1600 [==============================] - 17s 10ms/step - loss: 0.7333 - acc: 0.4950 - val_loss: 0.6959 - val_acc: 0.5050
Epoch 2/15
1600/1600 [==============================] - 14s 9ms/step - loss: 0.5925 - acc: 0.6731 - val_loss: 0.3484 - val_acc: 0.8625
Epoch 3/15
1600/1600 [==============================] - 14s 9ms/step - loss: 0.4011 - acc: 0.8206 - val_loss: 0.2916 - val_acc: 0.8900
Epoch 4/15
1600/1600 [==============================] - 14s 9ms/step - loss: 0.3669 - acc: 0.8431 - val_loss: 0.3109 - val_acc: 0.8675
Epoch 5/15
1600/1600 [==============================] - 14s 9ms/step - loss: 0.3579 - acc: 0.8369 - val_loss: 0.2842 - val_acc: 0.8875
Epoch 6/15
1600/1600 [==============================] - 14s 9ms/step - loss: 0.3554 - acc: 0.8437 - val_loss: 0.2832 - val_acc: 0.9025
Epoch 7/15
1600/1600 [==============================] - 14s 9ms/step - loss: 0.3565 - acc: 0.8456 - val_loss: 0.2809 - val_acc: 0.8875
Epoch 8/15
1600/1600 [==============================] - 14s 9ms/step - loss: 0.3446 - acc: 0.8494 - val_loss: 0.2966 - val_acc: 0.8725
Epoch 9/15
1600/1600 [==============================] - 14s 9ms/step - loss: 0.3513 - acc: 0.8400 - val_loss: 0.2685 - val_acc: 0.8925
Epoch 10/15
1600/1600 [==============================] - 14s 9ms/step - loss: 0.3476 - acc: 0.8575 - val_loss: 0.2727 - val_acc: 0.8950
Epoch 11/15
1600/1600 [==============================] - 15s 9ms/step - loss: 0.3444 - acc: 0.8481 - val_loss: 0.3381 - val_acc: 0.8375
Epoch 12/15
1600/1600 [==============================] - 15s 9ms/step - loss: 0.3466 - acc: 0.8531 - val_loss: 0.2672 - val_acc: 0.9025
Epoch 13/15
1600/1600 [==============================] - 15s 9ms/step - loss: 0.3498 - acc: 0.8444 - val_loss: 0.2708 - val_acc: 0.8950
Epoch 14/15
1600/1600 [==============================] - 15s 9ms/step - loss: 0.3434 - acc: 0.8519 - val_loss: 0.2953 - val_acc: 0.8800
Epoch 15/15
1600/1600 [==============================] - 15s 9ms/step - loss: 0.3429 - acc: 0.8531 - val_loss: 0.2652 - val_acc: 0.8950
```

Image 1: Training results at the end of each epoch

4. EVALUATION METRICS

The various performance metrics used to evaluate the model while training and to predict the results are:

- **Accuracy:** Given by total number of correct predictions by total number of predictions. Shown in Formula 1.

$$Accuracy = \frac{true\ positives + true\ negatives}{total\ predictions}$$

Formula 1: Equation to calculate accuracy

- **Loss:** Made use of the inbuilt **keras 'binary_crossentropy'** loss evaluation while training the model.
- **F1-Score:** F1 score can be calculated based on precision and recall values with the equation shown in Formula 3.

$$p = \frac{tp}{tp + fp}, \quad r = \frac{tp}{tp + fn}$$

Formula 2: p represents precision calculated from number of true positives (tp) and false positives (fp). Similarly, r represents Recall calculated from number of true positives (tp) and false negatives (fn)

$$F1 = 2\frac{p \cdot r}{p + r}$$

Formula 3: F1 score calculated from precision and recall values.

5. PERFORMANCE EVALUATION

After the model has been trained the trained model is saved to JSON format. Additionally the trained model weights are also saved so that the model can be loaded any time

and predict results on new input data. The loss and accuracy of both the train set and validation set are available after each epoch. These are plotted on a graph from the Figure 1 and Figure 2, we can see the trends of training. The respective values closely overlap each other following the trend of validation which shows that the training is reliable and is improving for each epoch.

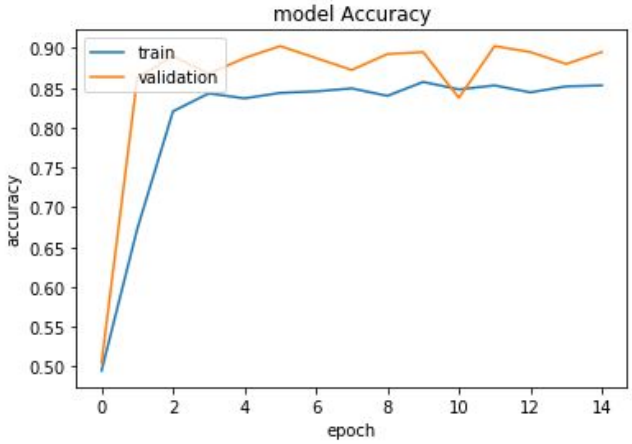

Figure 1: Accuracy of train and validation set after each epoch

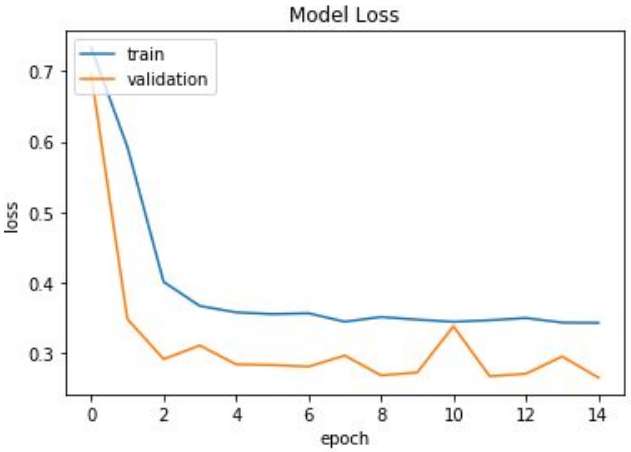

Figure 2: Loss of train and validation set after each epoch

After the 15th epoch the accuracy is not being improved any further and staying around 88% to 90% with approximately constant loss value. So, the further training of the model has been stopped.

To double check the accuracy a random sample of 400 tuples is extracted from the dataset and predicted class labels on them. The accuracy for the predictions was approximately equal to **86.5%** to verify error free functioning of the trained model.

The highest achieved accuracy and loss rate for the corresponding training are mentioned in Table 1.

	Accuracy	Log Loss
Train data set	88.31%	0.3228
Validation set	90.25%	0.2652

Table 1: Highest Accuracy and log loss values on train and validation data sets in all the epochs

The values are subject to change as the model is trained a random sample subset of fraction 1 from the main data set. This random sample gives random order of tuples everytime we train the model.

6. IMPROVING ACCURACY

Initially, the implementation was very simple with just the encoded DNA sequence fed as input to the model. Encoded DNA sequence contains Character to Integer sequence encoding and is generated with the help of tokenizer. With this simple implementation the accuracy achieved was **79.25%** on test data. Later, the experiments were done by tuning the hyper parameters such as units, output_dim etc. The batch size, number of epochs are also changed to observe the change in performance. Different optimizers like AdaDelta, SVM, Adagrad are used. The best accuracy was achieved with adagrad optimizer. The learning rate is also tuned to improve the accuracy. The cut off value

in the final prediction values of the probabilities has also been changed to 0.4 ,0.6 from 0.5 and the accuracy is observed. It only made a little change to overall accuracy increasing/decreasing it by ~1%. To process the input data more the sequence vector is additionally embedded into a matrix by feeding into Embedding layer on the network stack. This matrix embedding influenced the performance with a huge factor achieving a final best score of **90.5%** on test data.

7.SUBMISSION ON KAGGLE

After the model has been trained, the test data is grabbed from Kaggle competition and predictions are made on them. The obtained predictions are later appended with identifiers of respective tuples and written to a csv file using Pandas. The initial accuracy in the competition of 78.947% and improved it with the modifications to 79.487% standing in top 60%.

8.REFERENCES

[1] Generate Embedding matrices in Keras "https://machinelearningmastery.com/use-word-embedding-layers-deep-learning-keras//"

[2] API of tokenizer from Keras "https://keras.io/preprocessing/text/"

[3] Sepp Hochreiter and Jurgen Schmidhuber. (1997). "Long Short Term Memory". *Neural Computation* 9(8): 1735-1780