Promoter classifiers

Deep Shah, 20211004

Introduction

Understanding gene regulation requires first identifying and analysing promoter regions in DNA sequences. Upstream DNA sequences known as promoters control the expression of genes and are crucial for cellular functions like growth, differentiation, and reaction to external stimuli. Accurately predicting these promoter areas is crucial for the advancement of genetics, biotechnology, and medical research. In computational biology, promoter classifier models—which use statistical and machine learning techniques to find promoter regions in genomic sequences—have become essential tools. These methods identify putative promoters in previously undiscovered or poorly understood genomes by using vast datasets of known promoter sequences to understand patterns and attributes suggestive of promoter regions. As sequencing technologies continue to generate vast amounts of genomic data, the task of promoter classification becomes one of paramount importance.

The recent surge in AI techniques have contributed to the task substantially. The field of Natural Language Processing (NLP) has seen recent strides with the development of LSTMs, Transformers and LLM-SLM architectures which are extremely efficient at feature extraction and learn underlying patterns with ease. This report provides a brief overview of neural networks and the dataset in use, and then delves into the three models made to address the question of classification of promoters.

Dataset used

The dataset used was a portion of the DREAM challenge, where the regression problem was transformed into a classification one, by having 100,000 sequences from the higher range of the promoter score, labelled as positive, and 100,000 sequences as negative from the lower range of the promoter score, labelled as negative. The image below visualizes the data.

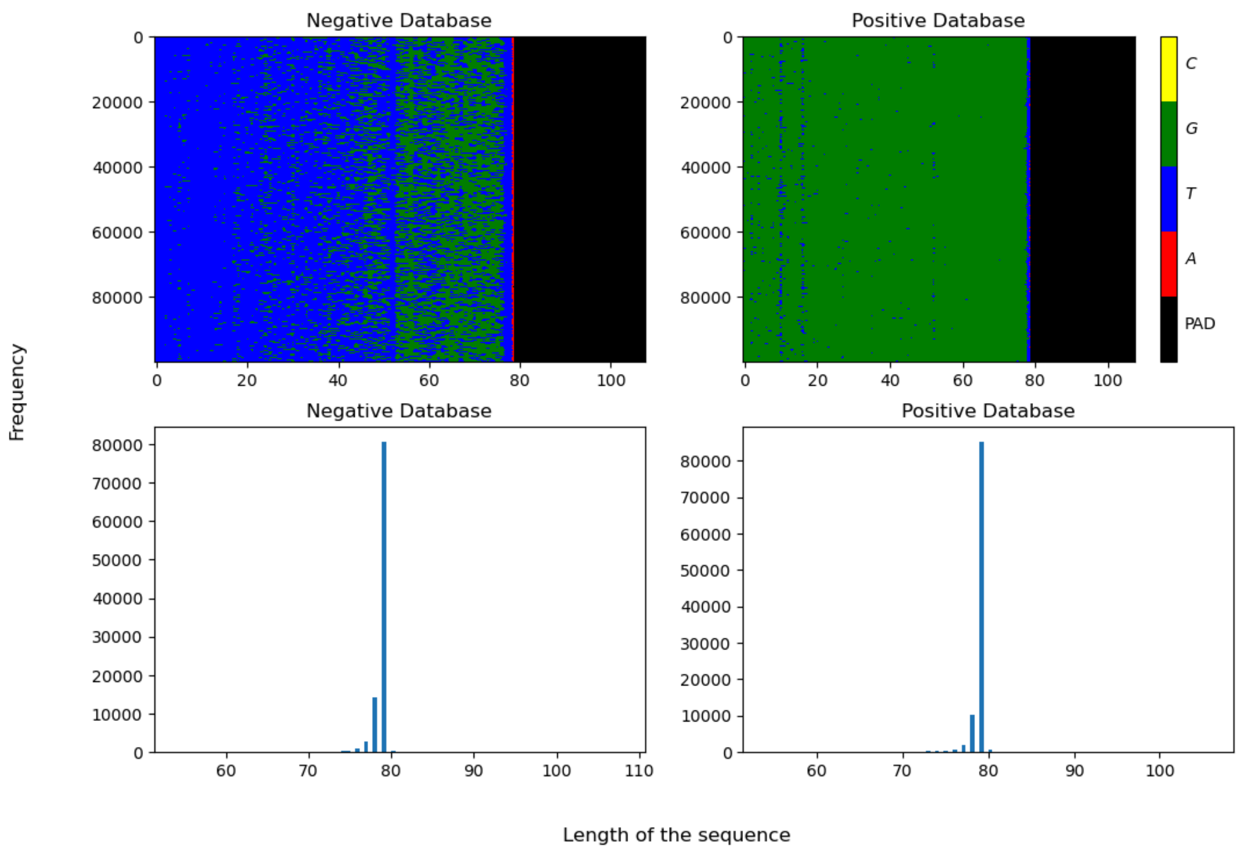


Figure-1. The sequences of each database were characterized as colours and one can observe that the negative sequences are T rich while the positive sequences are G rich. The same can be seen in the image below where position wise frequencies of A, T, G and C are plotted and visualized.

For the Markov models, all the sequences were considered, while for the LSTM and Conv models, sequences more than 80 in length were ignored. Additionally, during prediction, any sequence more than 80 bases in length was pruned to the first 80 bases and the predictions reported. For the sequences smaller than 80 bases, a padding was added which pre-padded the sequences with a character ‘X’.

In Markov models, for an n^{th} order markov model, n ‘X’ characters were added in the beginning of every sequence to signify a start site, and one ‘X’ character was added at the end to signify the end. In the LSTM and Conv models, ‘X’ was added until the sequence length was 80, and then one ‘X’ was added at the start and the end, making the total sequence length to be 82, for all sequences. This was done to allow for multiprocessing and efficient storage of the data in the form of numpy arrays.

Markov Models

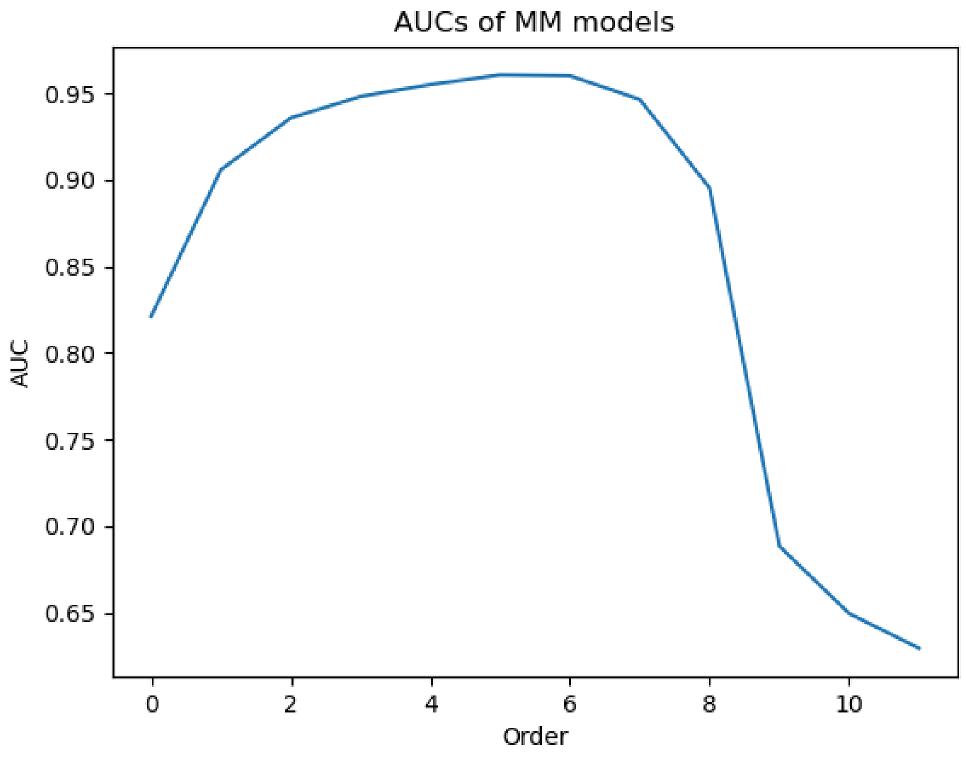
Markov models were created using numpy and were made so that they were parametrized by a variable N which decided the order of the markov model. The section below describes the results obtained from the markov models trained and optimized on the above data.

About the training

The frequencies of N-mers and their next base were counted using a sliding window method, where the starting window would always be N-Xs and the window would stop when it hits another X as predicted next base. These frequencies were stored in a matrix where a pseudo count of 1 was added to every matrix cell which remained as 0, including the ones which are biologically impossible, such as XXAX for a 4 order markov model. These were then normalized to obtain the probabilities or the transitions. Two models were made, one which modelled the positive sequences, and another which modelled the negative sequences. These models were then used to predict the log odds score of the sequence and thresholds for bifurcating positive from negative sequences was decided by the ROC scores and the value closest to (1,0).

Prediction

The below plot shows the AU-ROC of each of the models and we see that 6th order markov models (6OMM) perform better than all the other models. K-fold cross validation was performed and it was observed that the curves overlapped heavily for all t=of them, thus a train-test split of 0.8 was set for cross validation, and K-folds were not performed every time.



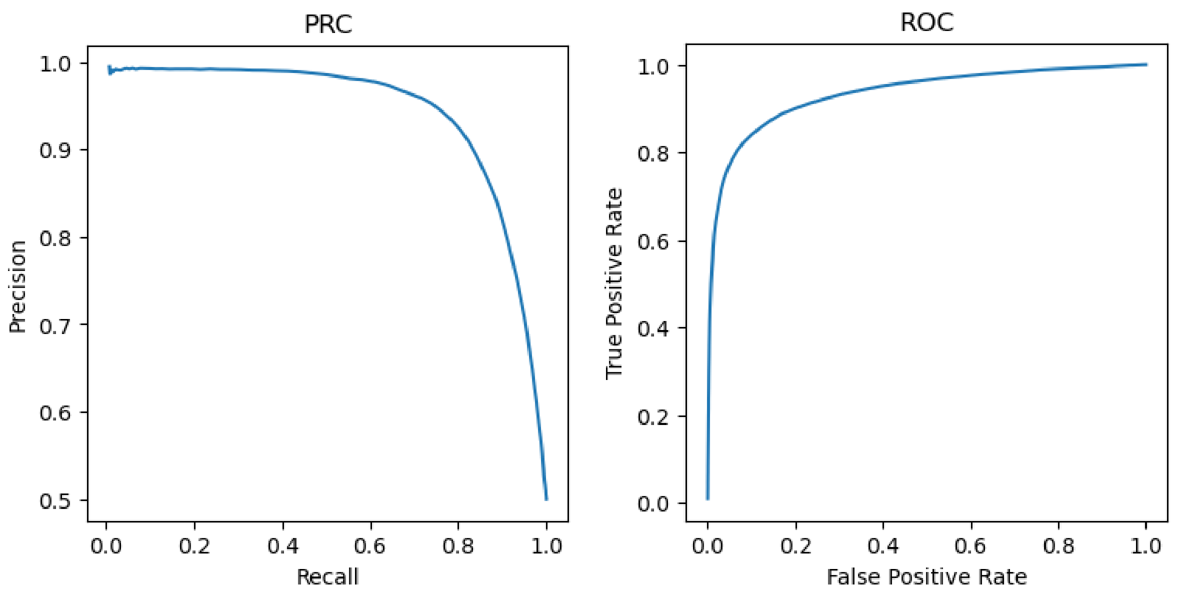


Figure-2. The above plot shows the performance of the markov models across different orders and the plots below show the PRC and ROC curves calculated for 6OMM as it performed the best.

The model

The model probabilities were visualized and plotted for 2OMM and are shown below. This gave us a general idea as to what the models look like and thus when a similar exercise was done for 6OMM, we see that the transition probability to a G is generally higher in the positive model (indicated by a brighter band at G) than in the negative model. This shows that the model really captured the GC percentage as well in its transition probabilities.

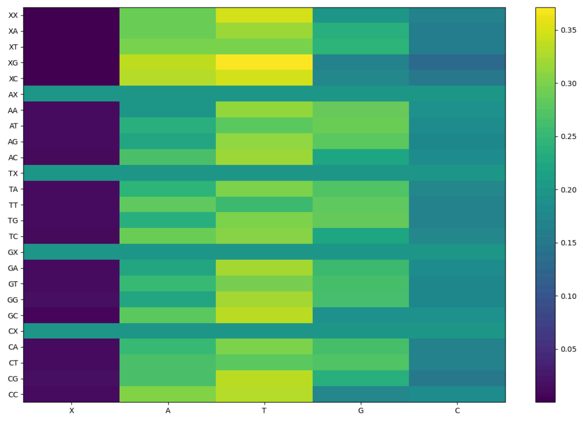
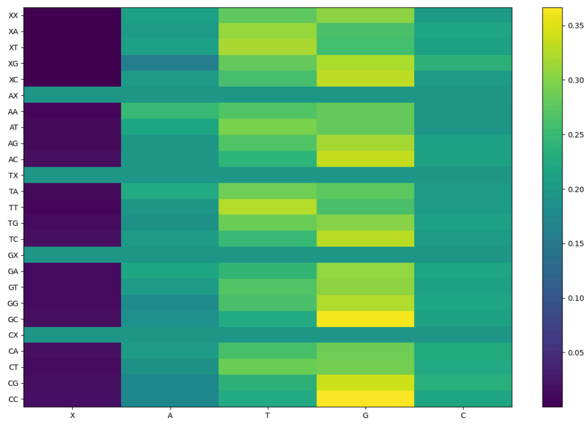


Figure-3 (A) the blocks above (Negative model) and below (Positive model) represent the transition probabilities of a 2OMM and we can see the brighter band around the G column in the Positive model.



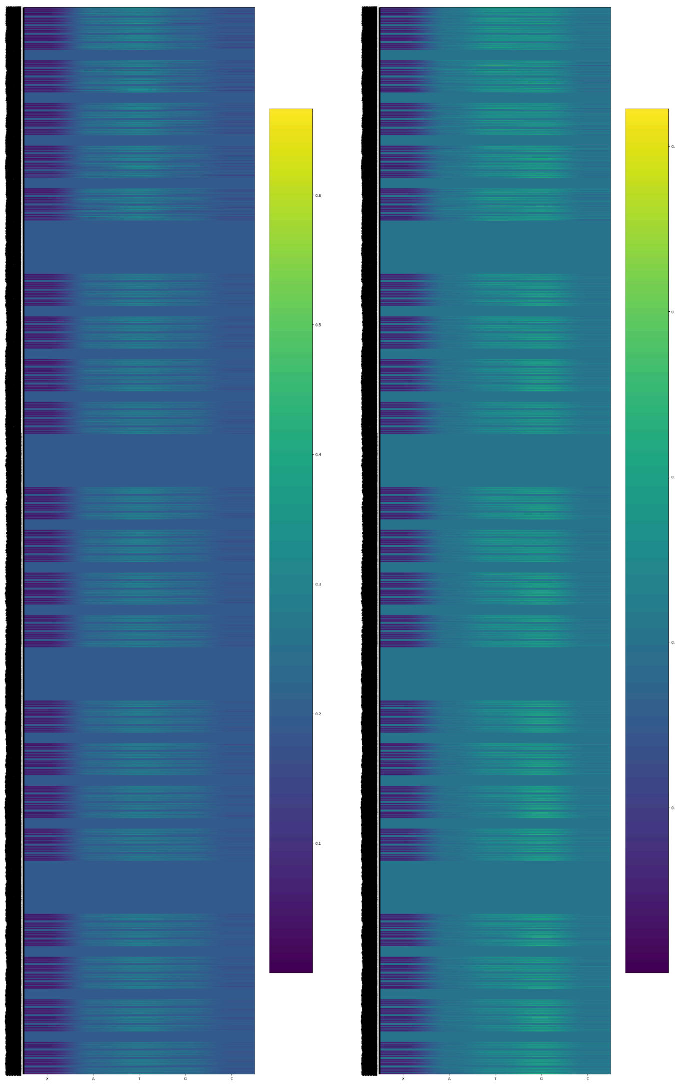


Figure-3 (B) Both the columnar images on the right (left columnar image: Negative model, right columnar image: Positive model) represent the transition probabilities of a 6OMM and a similar bright band can be seen.

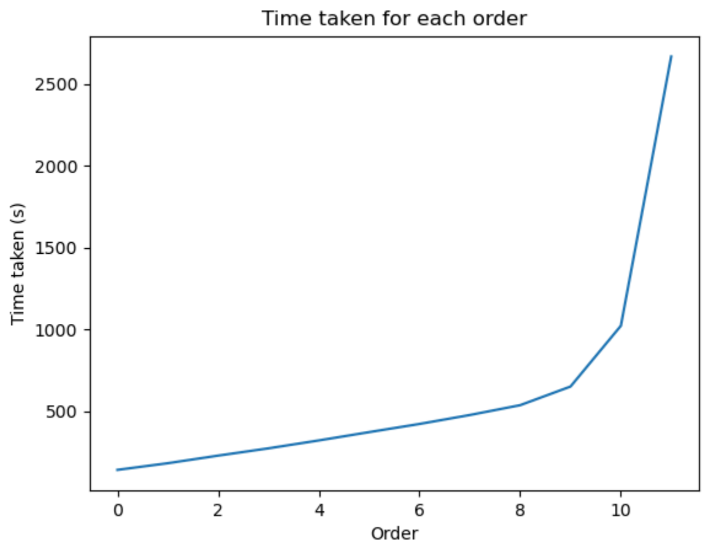


Figure-4 The time taken by each order for training can be seen above where the time initially increases linearly because the bulk of the time is taken by the overhead of numpy and python to generate and store the matrices, bu then the time grows exponentially as to now index the matrix and retrieve values take much more time.

The model can also now be used to generate promoter sequences and non promoter sequences by sampling from the distribution decided by the markov model transition matrix. Below are two samples of the same:

Negative:
AATAGTCCAACCCCGCAGCACGGAGCTAACAGTATGTGAGGTGCAATACGGTCGTGTCATCG
AAA

GCcontent:0.5076
Positive:
CATCGTAGTACTAGCCTCAGGGCGATATAGCAGTTTCGTACGTACTTTAGCCTGAGTGTATAA
TTGTAAAGGCCCGCGGGACAGGAGTGACACGCAGGGACCTCGGCTTGTCTGGTGTGCGTT
CAGTTGTACC

GCcontent:0.5263

LSTM Models

Given the successful implementation of MMs we sought to complicate the models to achieve better results at the expense of interpretability. We first used LSTM (Long Short Term Memory) models made using PyTorch to tackle the task. The next few subsections talk about the theory behind LSTM architecture and the details and results of our implementation of the same for the task.

What is LSTM?

A subclass of recurrent neural networks (RNNs) known as long short-term memory networks (LSTMs) is designed to effectively describe sequential data while addressing the vanishing gradient problem that troubles traditional RNNs. Memory cells and three gating mechanisms—the input gate, forget gate, and output gate—as well as a specialized design are used to do this. The output gate chooses which data should be sent to the next time step, the forget gate decides which data should be deleted, and the input gate controls the amount of data that is added to the memory cell. The network can selectively keep or remove information over long periods of time thanks to these gates, which are influenced by sigmoid activation functions and learnable weights. Because of this architectural layout, LSTMs are especially effective for applications such as natural language processing, time-series forecasting, and speech recognition, where the comprehension of long-term dependencies within data is essential. In genomics too, similar utilities can be found for LSTMs where entire sequences can be analysed by an LSTM layer, the representation of which can be used as the encoding for downstream processing tasks using DNNs.

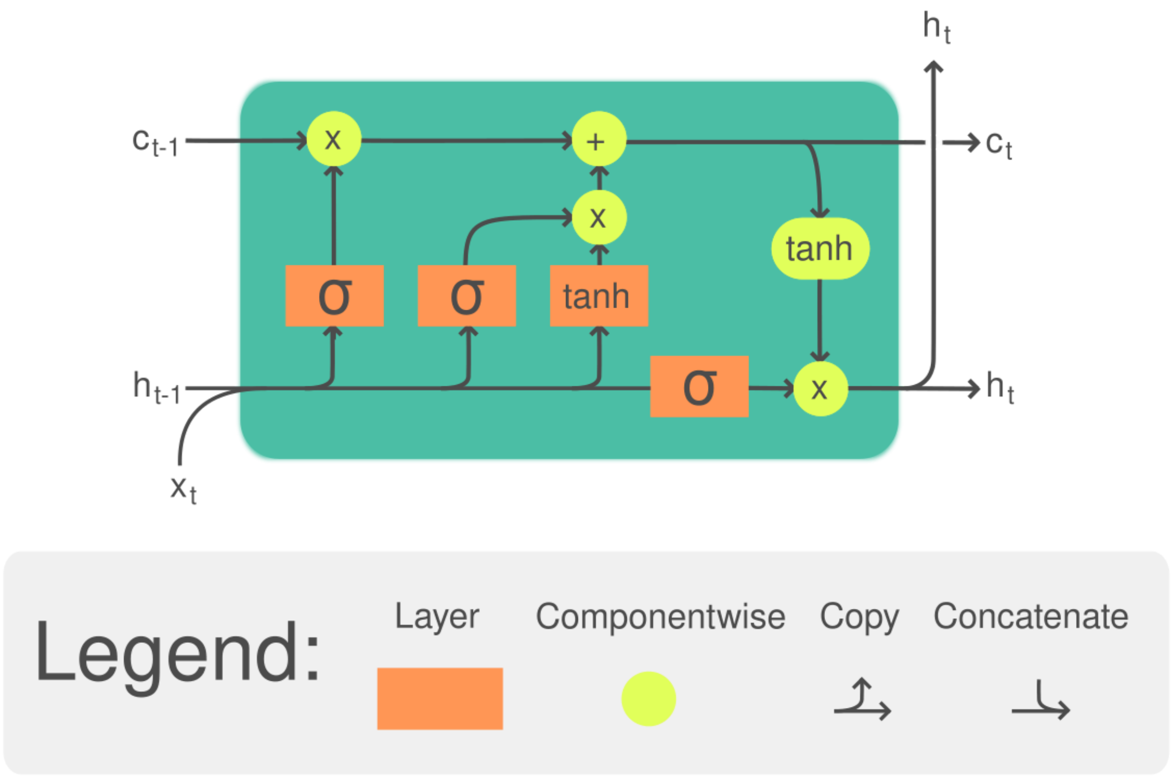


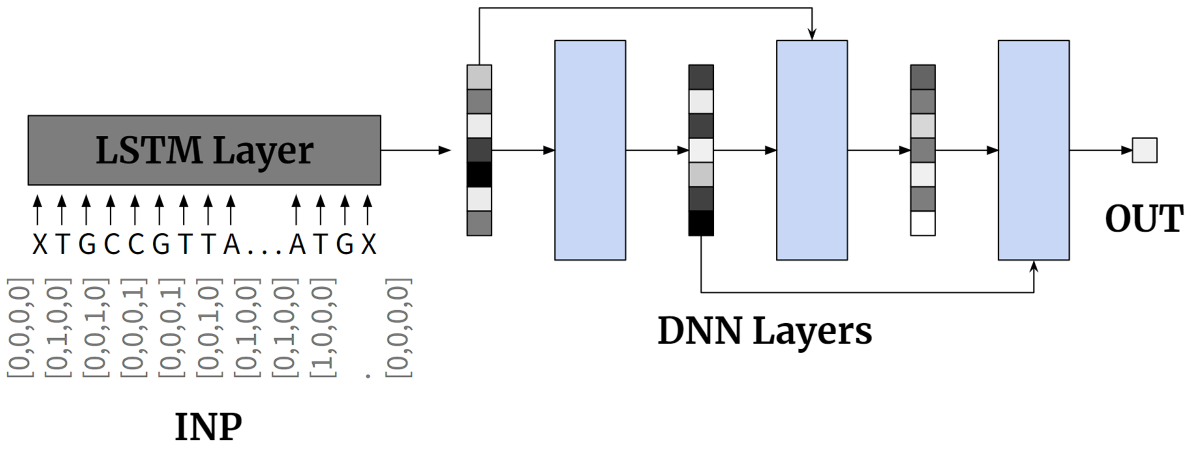
Figure-5. The above shows the general architecture of an LSTM layer. C_t (cell state) shows the continuous flow of information while H_t (hidden state) represents the information about the current state in context of the previous states. Both C_t and H_t in a conjoint manner decide which information needs to be ignored, remembered and forgotten. ***σ*** layer is the sigmoid layer where a layer of neurons process the information and apply the sigmoid activation. 'Tanh' represents the same with a tanh activation function instead.

The dataset and hyperparameters

The data is modified to fit the LSTM architecture. Each sequence is encoded as a stack of 4 bit vectors where X is represented by all zeros, while other bases are one hot encoded. Thus each sequence after padding becomes a 82x4 sized vector. Each dataset is thus a stack of such stacks which can be batch processed by the model. The batch size for all the models are 512, learning rate is 0.02, optimizer is Adam. Training is done for 15 epochs and losses at each training iteration (after every batch) is recorded.

LSTMskip

The first approach we took was to have skip connections, where the input to the layer n+1 is the input of layer n concatenated with the output of layer n. This makes it so that the model can decide when to use a layer to perform further feature extractions and thus keeps weights close to 0 whenever further feature extractions are not possible or feasible. Below is a schematic which represents the architecture of the LSTMskip model. The hidden state of the last 'X' is passed into the deep neural network such that the model is motivated to retain absolutely crucial information.



The exact architecture is mentioned below.

Note: Each DNN layer is also followed by a ReLU activation and a dropout layer with p=0.2

```
LSTMModelWithSkip(
  (RNN): LSTM(4, 64, batch_first=True)
  (DNN1): Linear(in_features=64, out_features=64)
  (bn1): BatchNorm1d(64)
  (DNN2): Linear(in_features=128, out_features=64)
  (bn2): BatchNorm1d(64)
  (DNN3): Linear(in_features=128, out_features=64)
  (bn3): BatchNorm1d(64)
  (DNN4): Linear(in_features=128, out_features=64)
  (bn4): BatchNorm1d(64)
  (DNN5): Linear(in_features=128, out_features=64)
  (bn5): BatchNorm1d(64)
  (DNN6): Linear(in_features=128, out_features=64)
  (bn6): BatchNorm1d(64)
  (DNN7): Linear(in_features=128, out_features=64)
  (bn7): BatchNorm1d(64)
  (DNN8): Linear(in_features=128, out_features=64)
  (bn8): BatchNorm1d(64)
  (DNN9): Linear(in_features=128, out_features=1)
```

)
The performance of the model is shown below along with the training and testing losses and accuracies plotted. We have also analysed the weights of the model, which are shown as well. The weights of the first layer show interesting patterns where some neurons look at some specific information in the input vector as noted by the distinct lines in the weight matrix. The subsequent layers however process the input to the previous layer and almost ignore the output of the previous layer, as noted by a complicated left side of the weight matrices, and a silent right side, as the weights are more or less completely 0.

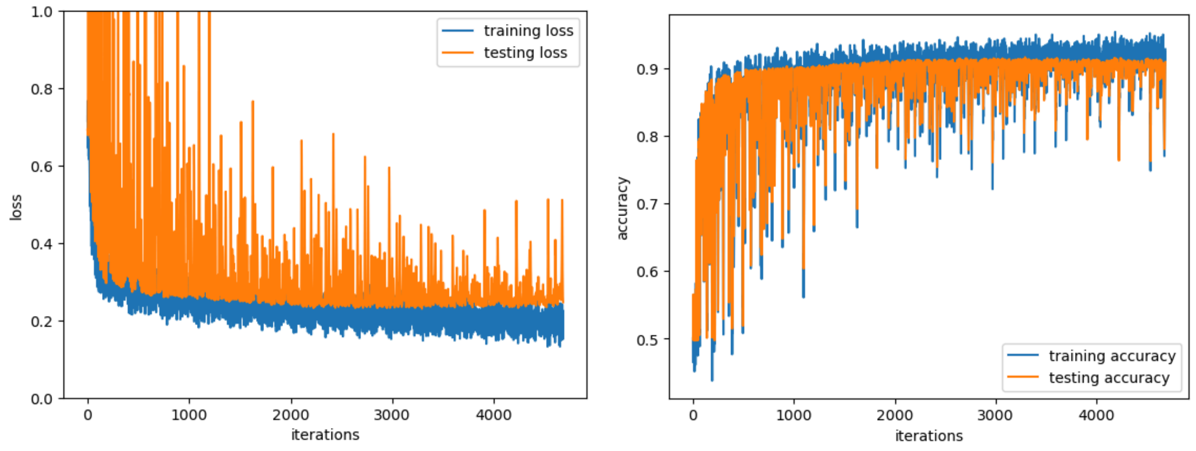


Figure 6: We see that there is high fluctuation in the testing losses, which indicate a very shallow minima. The final testing loss is 0.25 and the final testing accuracy is 0.91

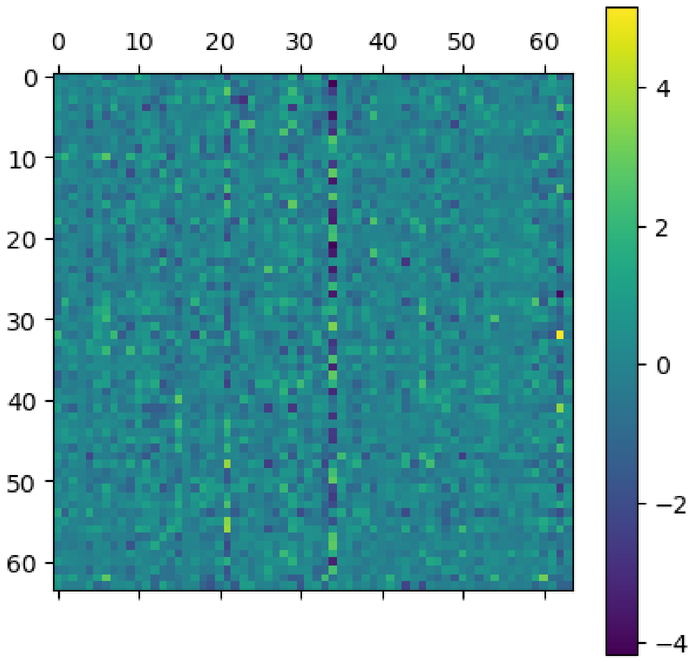


Figure 7: The first DNN layer after the LSTM layer. The dimensions are 64x64 as only one input is processed and a same sized output is generated. The subsequent layers will then receive both this input and output pair in a concatenated manner. The presence of lines may indicate that specific patterns are captured in the neurons which may correlate to presence or absence of promoter regions in the sequence.

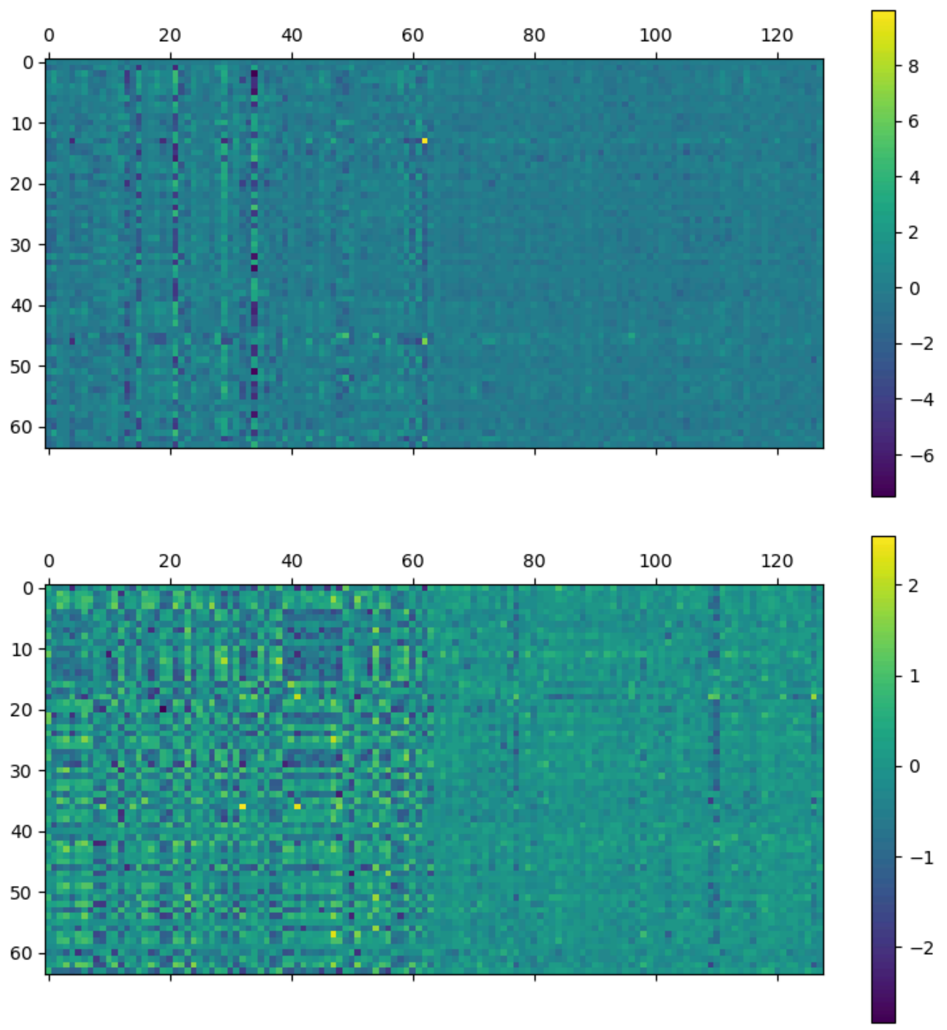


Figure 8: The above image shows the weights of the first layer receiving skip connection, and the one below is the second layer. As we can see, the model processed the initial input (left side) in a very similar manner as that of layer 1, and the right side (output of the last layer) is more or less uniformly distributed. While similar things cannot be said for the second layer, we draw the conclusion that one layer is sufficient to extract all the features as the left side has much more variability as compared to the right side. Further layers also follow similar trends, however as we go to the last few layers, both the sides become very similar indicating that processing of the input is stopped and both the input and output are deemed similar. The last layer is then used to make the precision which has a very similar distribution as that of the third layer left side (the second image), further boosting our hypothesis that one layer should be sufficient to capture all the information provided by the LSTM.

LSTMsingle

Under similar hyperparameters we trained a LSTM model with a single DNN layer which processes the output of the LSTM layer to output probability. Below are the training and testing losses and accuracies. We can see from the test losses that the model is much more stable and the losses do not highly fluctuate indicating a strong deep minima.

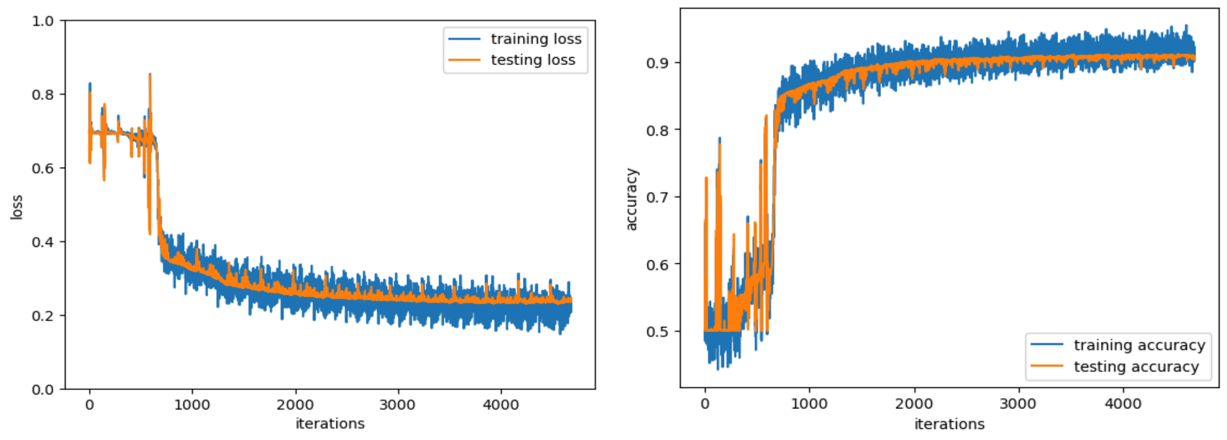


Figure 9: The losses and accuracies are plotted simultaneously for training and testing sets. The model achieves 0.23 testing loss and 0.91 testing accuracy.

These data confirm our hypothesis that only one layer is sufficient to draw equally reliable conclusions as compared to multiple layers of DNN.

ConvBreathing

The following subsection talk about a novel architecture that performs well on the task which uses convolution layers. For the data is transposed so that each row indicates presence of respective bases which columns indicate the positions and thus a 1 in position i, j describes base i being present in position j in the padded sequence.

Architecture

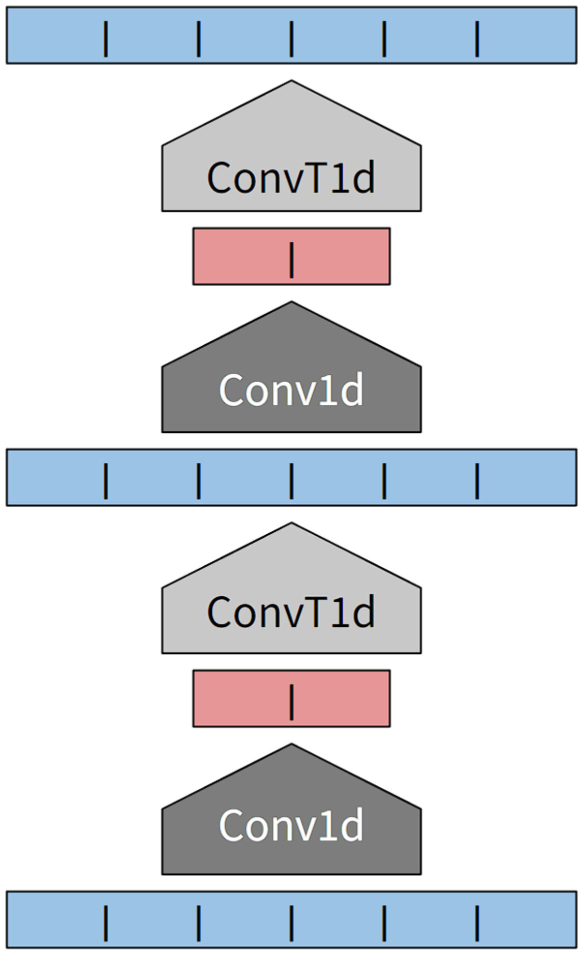


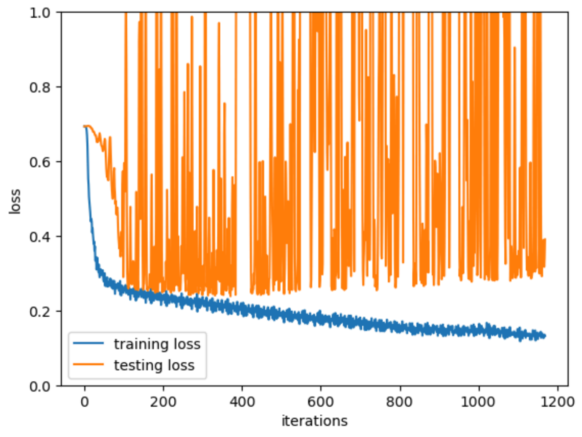
Figure 10: ConvBreathing architecture.

The above image demonstrates how a block of a ConvBreather operates wherein data from the surrounding bases is assimilated into a position using a 1d Convolution filter, and then projected back to the same length using a ConvTranspose 1d filter. Several of these blocks are then combined sequentially to perhaps obtain necessary information from far away bases. Two DNN layers are then applied at the final output which predicts the probability.

The exact architecture is shown below

```
NormalCNNClassifier(
  (PyCNN): CNNBreathing(
    (Conv1d_0): Conv1d(4, 32, kernel_size=(5,), stride=(1,))
    (ConvTr_1): ConvTranspose1d(32, 32, kernel_size=(5,), stride=(1,))
    (Conv1d_2): Conv1d(32, 32, kernel_size=(5,), stride=(1,))
    (ConvTr_3): ConvTranspose1d(32, 32, kernel_size=(5,), stride=(1,))
    (Conv1d_4): Conv1d(32, 32, kernel_size=(5,), stride=(1,))
    (ConvTr_5): ConvTranspose1d(32, 32, kernel_size=(5,), stride=(1,))
    (Conv1d_6): Conv1d(32, 4, kernel_size=(5,), stride=(1,))
    (ConvTr_7): ConvTranspose1d(4, 4, kernel_size=(5,), stride=(1,))
  )
  (DNN): Sequential(
    (0): Linear(in_features=328, out_features=512, bias=True)
    (1): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Dropout(p=0.1, inplace=False)
    (4): Linear(in_features=512, out_features=1, bias=True)
  )
)
```

Training



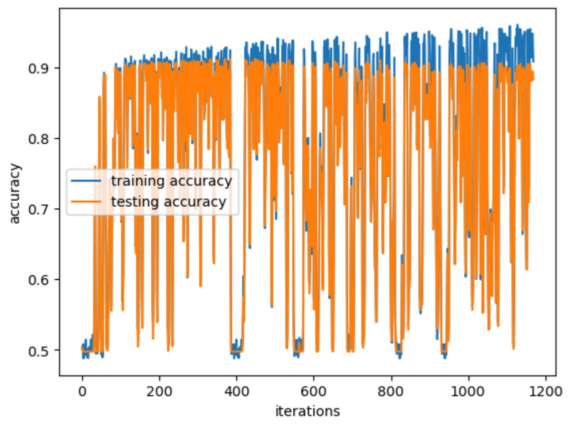


Figure 11: We can see in the above training steps that the losses are highly fluctuating. We can also see a lot of overfitting in the model as indicated by high testing losses and lower training losses. The final test loss was 0.4 and the final test accuracy was 0.87

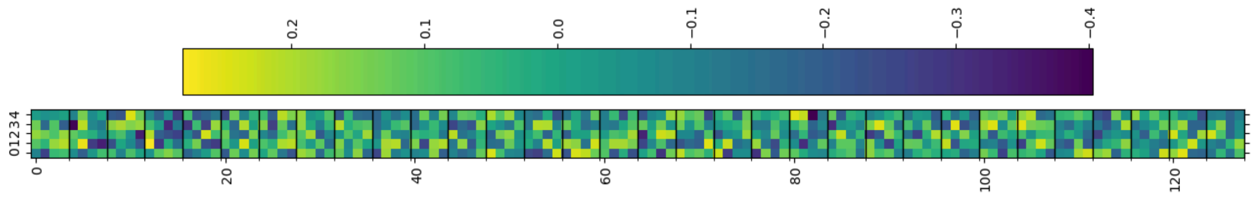


Figure 12: These show the Conv filters of the first conv layer. These are not interpretable.

The highly fluctuating losses are indicative of a very poor model. These would be much less pronounced with a lower learning rate, however it took much longer to converge so it is not reported.

Other Models

PyramidalCNN

Here the model architecture was such that at each step a convolution layer decreases the sequence size by 1. Thus 81 such layers make the final output to be a vector of length 4 after which a DNN layer is introduced that generates a probability. This model did not converge and gave random results as accuracy was 0.49..

Voting

Here all the three models, the LSTMskip, LSTM and ConvBreathing were given the sequence and by majority voting the outcome was decided. Each model is given equal weight during the aggregation step.

Results

The models perform with the following ROCs and PRCs

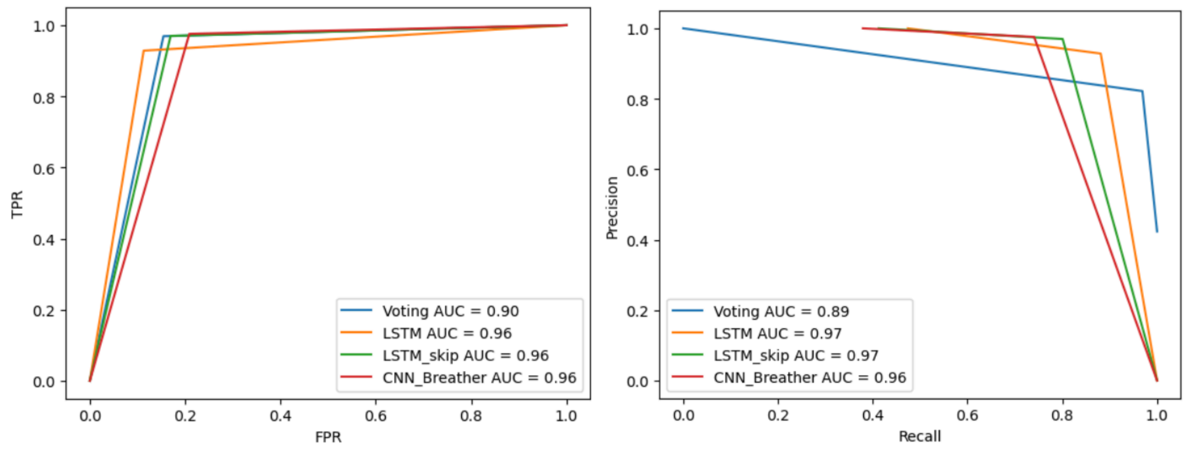


Figure 12: The graph on the left shows the ROC while that on the right shows the PRC. The AUCs of each are indicated in the legend.

Each model performed the following way on the external test set

20211004	Deep Viren Shah	deep.viren_shah	6	CNN_Breather_OUT.txt
				auroc= 0.952311236979786
				auprc= 0.961258553738756
				---- 	
				LSTM_OUT.txt
				auroc= 0.958271332341311
				auprc= 0.965297467063905
				---- 	
				LSTM_SKIP_OUT.txt
				auroc= 0.960987375798009
				auprc= 0.967727398897583

Discussion

Overall the LSTMskip model outperforms the other models in the testing and cross-validation step. We can see that the markov models not far behind with an AU-ROC of 0.95 are much more interpretable and can also be used for generating promoter sequences which the other models cannot do. We also see that the bigger the model, the harder it becomes to train it even with sufficient data, which is not the case with the Markov models where the higher order models were only limited with the data and the training step was relatively fast. The advantage of the neural networks was that they could be trained on a GPU relatively very fast and are parallelized which is not the case with a markov model training step. One must be very careful thus in choosing the model they would use for this purpose. Another observation is that these models perform well standalone, and do not give the same accuracy when voting is considered. This could perhaps be due to them capturing different features from the data and thus different models get different things correct. Thus some sort of Boosting model could perform much better than the current methods used.

Conclusion

We have made four models to address the challenge of classifying promoter sequences, namely a markov model, two LSTM models and a Convolution network. We see that these models capture different features and can be used for further downstream tasks such as target binding localization. More work needs to be done to improve the stability of some of these models so that the convergence occurs at a much more accurate state.

Acknowledgements

We would like to thank Dr. Leelavati Narlikar for providing us with the opportunity to work on this project and present it in class. We would also like to thank Prof. M. S. Madhusudhan for providing us with the access to the 42 cluster where all the computation was carried out.

Data Availability

The code along with the dataset used can be found at
https://github.com/Muzansama511/PromoterClassifier/