Name - Rishi Agarwal Department - EE Kaggle ID - Legendarygeneral268
Kaggle Rank - 330 (As of 10/07/2023) Team Name - ProteinXplorers

CAFA 5 Protein Function Prediction

1. Introduction to the problem

CAFA (Critical Assessment of protein Function Annotation) is an experiment, an ongoing, global, community-driven effort to evaluate and improve the computational annotation of protein function. The goal of the competition is to train ML models that can correctly predict the protein functionality from the given protein sequences and some data related to them.

Gene ontology is a database housing the most extensive knowledge base of the known genes' function in a machine and human-readable format that has been used in various large-scale biomedical and bioinformatics experiments. It describes genes concerning three aspects:

- **Molecular functions (MFO)**: According to the specific functions these genes carry out, like transport, catalysis, etc. These range from broad to specific, like protein kinase. They also are followed by the word ‘activity.’
- **Cellular Components (CCO)**: According to the location where the gene performs its functions
- **Biological Process (BPO)**: The big picture process that the gene’s function contributes to

The data is represented as an acyclic graph, with the parent ontologies being the more general description of the children. The nodes are connected to each other with edges, which represent certain ‘relations’ of the form of “is a”, “regulates”, or “is part of”, to name a few. Each term of the GO can be traced from the parents to the roots, which relate to the three different classifications described above. These three descriptions are disjoint, meaning there is no “is a” relation between any two nodes that trace back to them, but relations other than equivalence exist.

Using this data and the relationships between the different relations that relate to the functions of these proteins, we have to train a model to do such classification for us, i.e., classify the protein given to its molecular function by observing the amino acid sequence. This concludes the basic introduction of the GO and the problem, i.e., to use this GO and amino acid sequences of proteins to predict the labels (functions) of unlabelled proteins.

2. Literature Review

BioPython: Biopython is a Python library designed explicitly for bioinformatics, which involves the analysis and manipulation of biological data. It provides various functionalities for tasks such as sequence analysis, protein structure analysis,

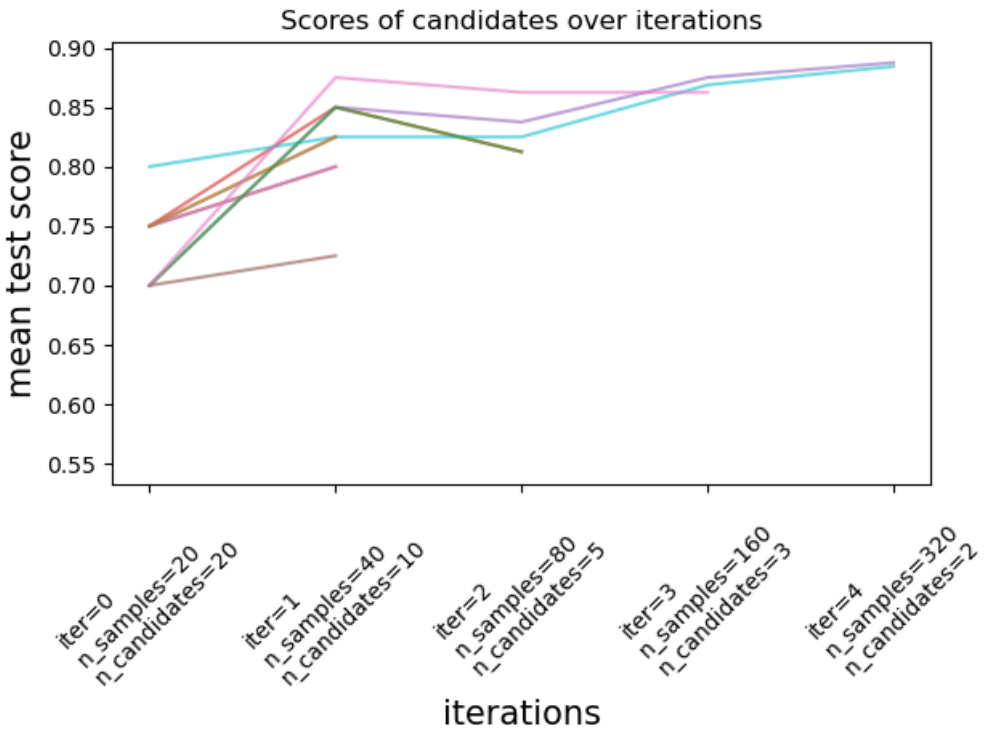
genomic data processing, and more. In the CAFA challenge, the protein sequence data is provided in the FASTA format, which is a text-based format for representing either nucleotide sequences or amino acid (protein) sequences, in which nucleotides or amino acids are represented using single-letter codes. BioPython provides parsers that help in efficiently extracting info on sequences out of these files. It also provides sequence alignment and matching algorithm implementation, such as BLAST and Clustalw alignment program. It also provides various other tools for performing common operations on sequences, such as translation, transcription, and weight calculations.
The documentation for Bypython can be found at:
https://biopython.org/wiki/Documentation

BLAST (Basic Local Alignment Search Tool): BLAST is a widely used algorithm that compares a query sequence against a database of known protein sequences to find similar sequences. It generates a similarity score based on sequence alignment and statistical significance. It uses some heuristic algorithm to give scores to rapidly search for local alignment matching rather than doing an exhaustive search.
Blast gives a non-ML naive approach for protein function prediction, which is the current baseline in CAFA.
BLAST tool - https://blast.ncbi.nlm.nih.gov/Blast.cgi
Insight to Blast - Link

Diamond is a sequence aligner implementation through which we can apply the BLAST algorithm. It is an optimized implementation that greatly speeds up the pairwise alignment of proteins and translates DNA at 100x-10,000x speed of BLAST.
Github - https://github.com/bbuchfink/diamond
A Kaggle competition implementation for BLAST

Multilabel Classification using XGBoost: The CAFA problem is a multilabel classification problem where every protein sequence is assigned multiple GO terms acting as the labels. We trained decision trees using the XGBoost algorithm for this classification problem. The labels which are GO terms are interrelated through a hierarchical tree. When a GO term is associated with a protein, then so are its ancestors. This might be captured by decision trees where the higher splits divide the protein into more general classes while going down the tree, they become more specific. The XGBoost model is a gradient-based optimized algorithm for forming an ensemble of decision trees. It improves the splits made by the current tree at different levels by calculating the gradient of the loss function.
A good intuition on this is provided by this link
We used the following implementation of XGBoost: XGBoost stable
The input to the XGBoost model cannot be a protein sequence obtained from the FASTA file. We need to obtain feature vectors of the sequences. T5 embeddings, Probert embeddings, and ESM embeddings are some of the popular embeddings in the CAFA competition that were better than the SOTA embeddings. We used the T5 embeddings for the XGBoost model. Grandmaster Sergei Fironov has the credit for generating them. Dataset for obtaining them
https://www.kaggle.com/datasets/sergeifironov/t5embeds

HyperBandSearchCV: A brute force approach for hyperparameter tuning is by listing a set of values for all hyperparameters and then training and evaluating the model on all the combinations. This can be done using scikit learn's GridSearchCV**.** But when the search space is large then this will take a lot of time, and a model with a lot of weights makes it worse. The hyperBandSearchCV approach speeds this process up with almost no loss in the precision of picking out the best parameters. This approach is similar to another approach called the halvingGridSearchCV. These methods first train all the possible combinations(called candidates) for a small subset of the dataset and a small number of epochs. Then they discard half of the candidates based on performance, and in each successive iteration, they double the dataset and epochs till finally, we are left with only one candidate. The following paper introduced hyperBandSearchCV https://www.jmlr.org/papers/volume18/16-558/16-558.pdf
And this paper has been implemented and open-sourced here
A good explanation of hyperparameter tuning is given at
https://scikit-learn.org/stable/modules/grid_search.html#searching-for-optimal-parameters-with-successive-halving
The following is also a good blog for hyperparameter tuning of XGBoost.
https://towardsdatascience.com/binary-classification-xgboost-hyperparameter-tuning-scenarios-by-non-exhaustive-grid-search-and-c261f4ce098d
The following gives good intuition about different hyperparameters of XGBoost.
https://www.kaggle.com/code/prashant111/a-guide-on-xgboost-hyperparameters-tuning



Feature Extraction and Dimensionality Reduction: Feature Extraction aims to reduce the number of features in a dataset by creating new features from the existing ones (and then discarding the original features). These new reduced sets of features

should then be able to summarise most of the information contained in the original set of features. This not only helps in better visualising and understanding large dimensional data but also helps in speeding up our model and sometimes prevents overfitting. Some of the methods used for feature extraction are:

a) **PCA(Principal Component Analysis):** Principal Component Analysis is an unsupervised linear transformation technique used for feature extraction and dimensionality reduction. It identifies patterns in data based on the correlations between different data points. More technically, it aims to find the directions of maximum variance in high-dimensional data and projects it onto a new subspace with equal or fewer dimensions than the original one. Basically we choose the top k eigen vectors of the covariance matrix and then project our data point to reduced space formed by these k vectors. In the reduced space, as is expected in a feature space, the same kind of data will be clustered together.

b) **Autoencoders:** Auto encoders provide a popular approach for reducing the dimensions of the data. An autoencoder is an unsupervised multilayered ANN consisting of three parts: encoder, code, decoder. This ANN is trained on the input data and has to give the same input data as the output. The encoder layer is expected to map the input data on a smaller space (of the code) called the latent space and the decoder learns to reconstruct the data from this latent space. After the training we can remove the decoder part and the encoder part performs dimensionality reduction for us.
The following are some good blogs for learning about PCA and auto encoders:
https://towardsdatascience.com/principal-component-analysis-for-dimensionality-reduction-115a3d157bad
https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

RNN, LSTM and BiLSTM: Since the problem at hand involves sequential data, RNNs and LSTM seem to be the obvious choice for this task.

a) **RNNs** keep a hidden state which basically captures the data from the sequence it has seen so far. They keep on updating this hidden state and use it to predict output from the next neuron in the sequence. All of the neurons share the same weights. Parameter sharing enables the network to generalise to different sequence lengths. But this leads to the problem of exploding and vanishing gradients because the gradients are also the same and on multiplication to calculate the net gradient they increase or decrease exponentially.

b) **LSTM** proposes a solution to these problems. They have a special cell state called Ct, where information flows and three special gates: the forget gate, the input gate, and the output gate. The forget gate decides what information is not important and can be forgotten, input gate decides what info should be stored and the output gate decides what information should be shown as the output at any point of the sequence processing. The LSTMs are unidirectional, that is info flows only in one direction. They can only have information of the past sequence and not the future.

c) **BiLSTM** are nothing but two LSTMs working in parallel and storing information from both directions. So their state is updated not only from past data but also from the future sequence. In protein sequence alignment the sequence lying to the left and to the right are equivalent and should be equally considered when updating any neuron, so BiLSTMs are a better choice than LSTMs.

The following is a good blog for learning intuition and implementation of RNNs and LSTMs
https://ai.plainenglish.io/recurrent-neural-networks-for-multilabel-text-classification-tasks-d04c4edd50ae#8e5d
The following repo has some good tutorial on RNNs for sentiment analysis
https://github.com/bentrevett/pytorch-sentiment-analysis

3. Experimentation

A) **Exploratory Data Analysis (EDA)** - There are 5 essential files in the competition dataset upon which we performed an EDA to understand the data better and to get insights into how to decide which methods to use. Following is a brief description of the same:
In the problem statement, we have been given a dataset of various proteins that have been annotated based on their functions as understood by experimental verification. As part of this data, we are given the Gene Ontology table of these proteins in the **GO-basic.obo** file and the amino acid sequences of these proteins in the **Train_sequences.fasta** file. Each protein is also classified into the identified species of the protein group that they belong to in the **train_taxonomy.tsv** file. The entries in the GO table follow the GO-ID nomenclature, where each node in the table has a specific ID, and each term of the FASAT file has been named according to the UniProt ID nomenclature. The corresponding nomenclatures are mapped to each other in the **train_terms.tsv** file.

To perform the EDA, we took reference from this notebook from the discussions page in the Kaggle competition. This is the link to our notebook. We first defined the paths to all the files and then wrote a function to read the data. Then we stored it in a graph format from the obo file and displayed. We further performed statistical checks and counting operations on the data to get a sens eof what the files are and what they entail. These can be found in our notebook here

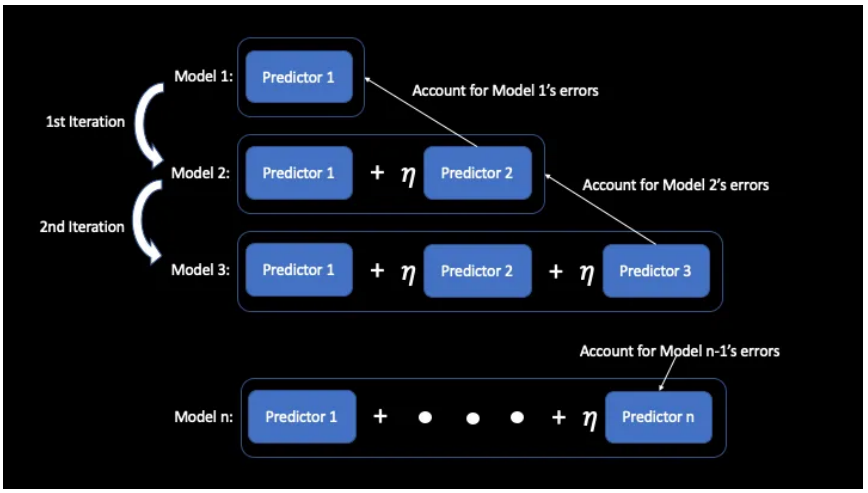
B) **Sequence similarity matching using BLAST algorithm** - This was a naive approach of similarity matching and sequence alignment of protein sequences on the argument that proteins with similar amino acid sequences have similar functions and structures. This similarity matching was implemented using the diamond sequence aligner. The protein fasta file which is provided in the competition was used for forming the database of the sequence aligner on which it will perform BLAST alignment for requested sequence. Then the unseen sequences are matched and top k matched sequences are obtained. We say that the associated GO terms with these sequences also belong to the unseen sequence and the number of times each term is repeated in the matched sequence gives the strength of this term being associated

with the original sequence. So for every GO term(GO_i) we say its probability of being associated with original sequence = (# occurrence of GO_i on all matched sequences) / (total number of matched terms).

C) **XGBoost Implementation** - This was a series of tasks where we had to select one basic ML method from a pool of methods provided by the mentor and implement the same on the data. We first learned and implemented the basic algorithm, then modified it, then applied dimension reduction and did hyperparameter tuning. Clearly, it was a lengthy task. This proved to be an excellent exercise to handle the data and get used to different techniques to improve results while revealing that a simple algorithm is not well suited for this task. I say this because the results (discussed in the Results section in detail) could have been better using this.

Technical Details - We used the XGBClassifier from the Scikit library for our model and split the training dataset into an 80:20 split. This was done for preliminary testing before submissions to save them. We used the roc_auc accuracy parameter. We are using the T5 embeddings as input.

About XGBoost - XGBoost is short for Extreme Gradient Boosting, a type of implementation of Gradient Boosting Machines on Decision trees exploiting various optimizations for faster training. It creates an ensemble called Gradient *Boosting Trees,* which work on the basic principle of Gradient Boosters, but by creating splits and constructing a tree. Gradient Boosting Machines work by making smaller error prediction models on the labels and these error predictor models are further assisted by models that predict *their* error. This forms a chain of models, and since the gradient of the loss is the error, they are called Gradient boosters.



(Image source and understanding developed from this blog)

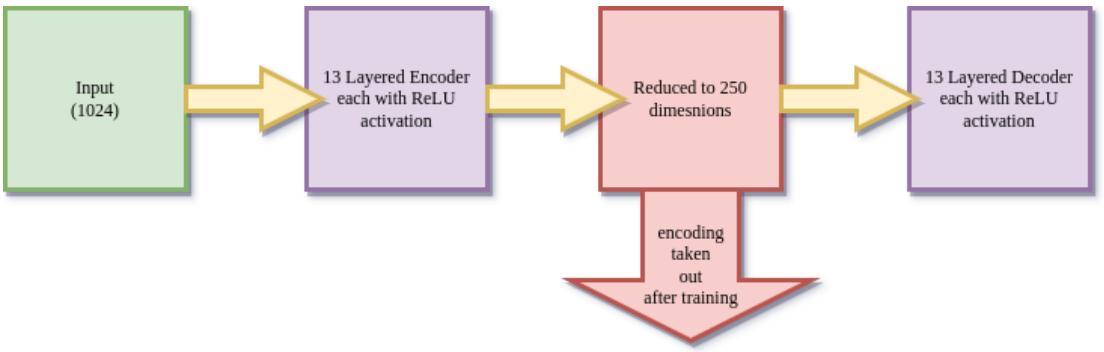
XGBoost then simply applies this technique by first starting with a very simplistic tree that predicts the same answer every time. We define a parameter o in the loss of the model and find this parameter o by setting the gradient to zero. Then, we find the splits by first finding these optimal parameters o for each split and then calculating the total loss. If it is less than before the split, we consider the split and then repeat.

XGBoost Implementation - This[1] is a link to our implementation notebook, and this[2] is the link to our evaluation and submission of the same. This was done as a

team by myself and Krishnansh Agarwal. We first ran into the problem that implementations of the XGBoost Algorithm exist only for binary classification problems. Since ours is a multilabel classification problem, we used a softmax output layer to get the probabilities. In this instance, we trained the model on all the train data, and predictions were taken using the Eval notebook on a reduced size of 35000 data points. This is a recurring problem for submissions to the competition as the predictions for all data points for 1500 labels when converted to the TSV format, take up a lot of RAM, so much so that Kaggle sessions regularly run out of memory.

On the first run without any tuning, we achieved a Fmax score of 0.046. Even considering that the submission was made on reduced labels, still, such a result goes on to show that there is little to expect from simplistic algorithms. We implemented parameter tuning to confirm this fact.

Hyperparameter Tuning[3] - We tried hyperparameter tuning using the Grid search method. We referred to this blog for an insight into how it works. It is a brute-force method where we first define an n-dimensional grid, n being the number of hyperparameters, by selecting a set number of values for each parameter. Next, we travel to each grid of this cell and try out those sets of values and evaluate the accuracy. Usually, this is done on a reduced dataset for faster results. We tested over a range of hyperparams for the parameters but soon realized that a normal grid search would not cut it, as the time to check for all the hyperparameters was immense. We applied two methods to overcome this
i) Dimension Reduction[4] - We further reduced the size of the T5 embeddings that were the input to the model and this time the search time for tuning reduced. This is the notebook where we generate encodings of reduced dimensions and this is the notebook where we hypertune parameters using these reduced dimensions. We use 13 layered encoder and 13 layered decoder for this purpose and finally generated encodings of dimension 200. This helped in improving time but the scores remained almost the same with a non-significant increase. The autoencoder looks like:



```
# Encoder Layers
encoded1 = Dense(3000, activation = 'relu')(input_dim)
encoded2 = Dense(2750, activation = 'relu')(encoded1)
encoded3 = Dense(2500, activation = 'relu')(encoded2)
encoded4 = Dense(2250, activation = 'relu')(encoded3)
encoded5 = Dense(2000, activation = 'relu')(encoded4)
encoded6 = Dense(1750, activation = 'relu')(encoded5)
encoded7 = Dense(1500, activation = 'relu')(encoded6)
encoded8 = Dense(1250, activation = 'relu')(encoded7)
encoded9 = Dense(1000, activation = 'relu')(encoded8)
encoded10 = Dense(750, activation = 'relu')(encoded9)
encoded11 = Dense(500, activation = 'relu')(encoded10)
encoded12 = Dense(250, activation = 'relu')(encoded11)
encoded13 = Dense(encoding_dim, activation = 'relu')(encoded12

decoded1 = Dense(250, activation = 'relu')(encoded13)
decoded2 = Dense(500, activation = 'relu')(decoded1)
decoded3 = Dense(750, activation = 'relu')(decoded2)
decoded4 = Dense(1000, activation = 'relu')(decoded3)
decoded5 = Dense(1250, activation = 'relu')(decoded4)
decoded6 = Dense(1500, activation = 'relu')(decoded5)
decoded7 = Dense(1750, activation = 'relu')(decoded6)
decoded8 = Dense(2000, activation = 'relu')(decoded7)
decoded9 = Dense(2250, activation = 'relu')(decoded8)
decoded10 = Dense(2500, activation = 'relu')(decoded9)
decoded11 = Dense(2750, activation = 'relu')(decoded10)
decoded12 = Dense(3000, activation = 'relu')(decoded11)
decoded13 = Dense(X.shape[1], activation = 'sigmoid')(decoded1
# Combine Encoder and Deocder layers
autoencoder = Model(inputs = input_dim, outputs = decoded13)
```

ii) Halving Grid Search CV[5] - Halving Grid Search is a modification of normal Grid search, which cleverly divides the search space into intervals and performs the model training in increasing dataset sizes. First, a very small dataset is used on two halves of the search space, and the better of them is chosen, therefore halving the search space. Then, the dataset size is doubled for a more rigorous evaluation, and the process is repeated. This greatly reduces the time for searching without great loss in finding optimal parameters. This[6] was the reference where we learned this technique. We implemented this in this notebook on the reduced dimensionality data and managed to improve our results, although not by much, as discussed in the concluding section of this task.

Results and Conclusion - The conclusion we can draw is that although we need more accuracy, this task armed us with the necessary knowledge to implement new solutions to the problem which we are currently working on.

We did this task in a series of extended sittings together, so it is difficult to point out exactly who did what, but broadly speaking, I handled the submission "Eval Notebook" and tried various submission sizes to get the first submission up which happened for 35k labels, and also helped in debugging the model notebook and implemented the parameter tuning, while Krishnansh did the model notebook and dimension reduction implementation.

The following table summarises the significant milestones in the task along with the accuracies we got on them.

Milestone	Accuracy(FMax by Submission)
Simple XGBoost with binary output	0.0026
XGBoost with probabilities by Softmax	0.04308
XGBoost after dimension reduction	0.04672
XGBoost with hyper-param tuning	0.05493

Finally, we remark that this was a valiant attempt, and for a simple algorithm like XGBoost, the results are considerable (scaling up to the actual 142,000 data points, a theoretical accuracy of ~0.2 could have been achieved)

References -

[1] XGBoost training Notebook
[2] XGBoost Evaluation Notebook
[3]Grid Search Notebook
[4] Dimension Reduction Notebook
[5] HyperaBand Search on Dimensioned
[6] Halving Grid Search reference

D) **LSTM and BiLSTM :** Due to the sequential nature of the dataset, LSTM and BiLSTM seemed a good choice for the problem. So we took references from different blogs and github repo to do our implementation. First we used a simple LSTM layer on which we fed the pretrained T5 and ProBert embeddings. This baseline model did not give good results at all. Then we tried adding some Dense layers in front of the LSTM layer. This was because we learned that LSTM is usually used to capture sequential information and then pass it to a deep network that then uses this info to make predictions. So we took reference from this[1] notebook to add a similar MLP model in front of the LSTM. This worked better after we made some more changes to that implementation. This[2] is our implementation for the LSTM model.
We took reference from this[3] repository.

Next instead of using the pretrained embedding we decided to pass the sequences directly to the network and put an embedding layer before the LSTM. This embedding layer would learn the representation of the sequences in a real vector space. Initially we kept the embeddings to 21 with the argument that there were 21 different amino acids in the dataset and so our embeddings layer would learn 21 dimensional vectors, each dimension corresponding to each amino acid. Also we changed the LSTM layer with a BiLSTM layer. For this implementation we took reference from this[4] blog which also tries solving a similar problem of protein function prediction. We again added some Dense layers in front of the BiLSTM layer. The final output had a sigmoid activation and the binary_crossentropy was used as the loss function. We also added regularisation to the BiLSTM layer and dropout of the neurons as well. This was done so as to prevent overfitting. The model was not doing so good this time. Next we tried to fine tune the model using a manually written code then using the keras tuner[5] which is a keras library for hyper parameter tuning

of the deep networks. This also has an implementation of advanced search algorithms such as halving grid search and Bayseian search. After searching for the best hyperparameters for a long time, the model scoring improved but it was not so good and it seemed like it had reached its saturation. This[6] is the notebook where we implemented the above model. We obtained the following results:

Model	FMax Score
LSTM + ProBert	0.129
BiDirectional LSTM with Embedding Layer	0.116

This is how our BiLSTM Model looks:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 2000)]            0

 embedding_1 (Embedding)     (None, 2000, 384)         80640

 bidirectional_1 (Bidirectio  (None, 512)              1314816
 nal)

 dropout_1 (Dropout)         (None, 512)               0

 dense_2 (Dense)             (None, 512)               262656

 dense_3 (Dense)             (None, 500)               256500

=================================================================
Total params: 1,914,612
Trainable params: 1,914,612
Non-trainable params: 0
_________________________________________________________________
```

The score obtained by the LSTM was underwhelming but next we are trying to implement an attention mechanism and GRU which usually promise to prove much better than the LSTM approach.

References -

[1] Reference for MLP+LSTM
[2] Our LSTM Notebook
[3] Reference Repo for LSTM
[4] Reference blog for BiLSTM
[5] Keras Tuner Documentation
[6] Our BiLSTM Implementation

4. Further Resources

Finally this is the sheet where we have done all the tasks assigned to us
https://docs.google.com/spreadsheets/d/1k1PeMh89COWt8ksnJBRCNRSYzluN9Zhn6BjAfW9iXJA/edit#gid=55360585

And this is the github repository where the code from all the teams is compiled
https://github.com/BSBE-IITK/CAFA5-PFP