Mechanism of Action (MoA) Prediction

Problem Description

The goal of this project is to classify drugs on the basis of biological activity using machine learning techniques. The objective of the development of pharmaceutical drugs is to classify certain proteins associated with a particular disease and then to create molecules that can target those proteins.

The term mechanism of action (MOA) in pharmacology refers to the biochemical interaction by which a drug substance generates its pharmacological impact. A mechanism of action typically involves mention of molecular targets, such as an enzyme or receptor, to which the drug binds. Receptor sites have drug affinities depending on the drug's chemical structure, as well as on the specific action that takes place there. Drugs bind to receptors found on the cell surface or in the cytoplasm (a jelly-like substance inside a cell). The drug will take on one of two roles: agonist or antagonist after the receptors bind to a cell.

Motivation

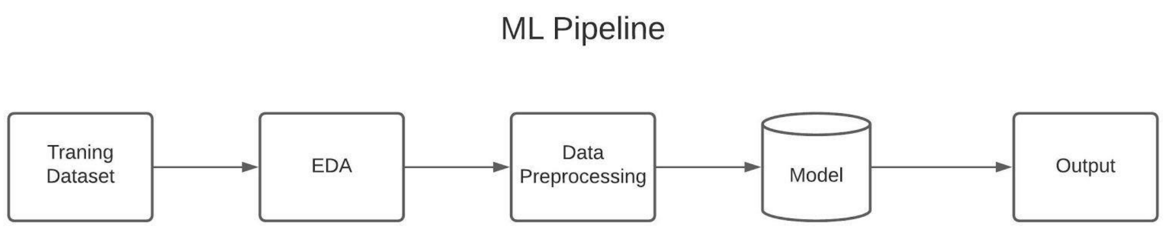
The biological domain is vast and something that influences each one of us in our everyday lives. Gaining a deeper understanding would help us in solving the mysteries of the biotic forms around us. However, the challenge here is that the data produced while experimenting in this area is enormous. Manually processing the data can be a herculean task and here comes the role of machine learning and the advances in computational efficiency.

We chose this dataset as it pertains to the biological field and true to its nature, is a highly complex dataset. By running multiple models, we seek out to find models which can live up to the task of successfully understanding this complex data and successfully generating the required predictions. We plan on testing relatively simple models and along with them, a few complex models to see how well they perform against each other.

Technical Problem Statement

This problem turns out to be a Multi-Label Classification problem. Multi-label classification involves predicting many mutually non-exclusive classes or labels, unlike usual classification tasks where class labels are mutually exclusive.

During the implementation of these models, we compare them on the basis of their log loss errors. Log loss is a value that computes the error with which we classify an instance of the particular class. Here we take into consideration the probabilities that the model outputs. We penalize severely the model which misclassifies an attribute. Ideally, if we predict the correct class with probability 1 we have zero log loss.



The general pipeline which we follow for the models is shown in the figure.

We start with the initial dataset and do an Exploratory Data Analysis step to get more understanding of the data. After gaining insight into the dataset we do the preprocessing steps to get the data ready for the models to train on. After training the models we finally output the predictions and save it onto a csv file.

The various models we will be using for this dataset includes:

- Logistic Regression
- SVM
- Neural Network
- ResNet
- XGBoost

Dataset

This dataset comes from the Mechanism of Action competition in Kaggle. It consists of the following files :

- train_features.csv: Features for the training set.
- train_targets_scored.csv: The binary MoA targets that are scored.
- test_features.csv: Features for the test data.
- submission csv: Predicted values by the trained model.

We shall dive deeper into the details of these files.

Train_features.csv

This is a rather wide data with 876 columns. The breakdown of these columns are as follows:

Column Name	Number of columns	Column Type
sid_id	1	Categorical
g-XXX	772	Numerical
c-XXX	100	Numerical
cp_type	1	Categorical
cp_time	1	Categorical

cp_dose	1	Categorical

Each row in the file is a particular sample and Sig_id is the unique primary key of the sample.

The 772 columns starting with “g-” indicate the gene expression column. In the synthesis of a functioning gene product, gene expression is the mechanism by which information from a gene is used. Mostly these things are proteins. The XXX denoted the nos. from 0 to 771.

The 100 columns which start with “c-" indicate the cell viability column. Cell viability determines the ability of organs, cells or tissues to maintain or recover a state of survival. Viability can be distinguished from the all-or-nothing states of life and death using a quantifiable index that ranges between the integers of 0 and 1 or if more easily understood, the range of 0% and 100%. Similar to the gene columns the XXX refers to the cell column number.

cp_type is a binary categorical feature which indicates the samples are treated with a compound or with a control perturbation. Cp_dose refers to the dosage that these compounds were exposed to. There are three discrete levels of dosage - low, medium and high, making this a categorical variable. Cp_time refers to the amount of time the samples were exposed to the doses. This is measured in terms of hours.

Train_targets_scored.csv

This file contains the labels for our dataset. There are 207 columns consisting of sig_id for each column and 206 classes.
These 206 classes are the various MoAs for each of the rows in our training dataset. Each row in our training file is mapped to one of the rows in this file and is a one to one mapping. Each sample can be part of one or more of these classes at a time making this a Multi-label classification problem.

Test_features.csv

In this file, we have the non-labelled set of samples which will act as the test set for our models. The trained algorithms are run on this to get the predicted MoA which is then used to evaluate the accuracy of the model over unseen data.

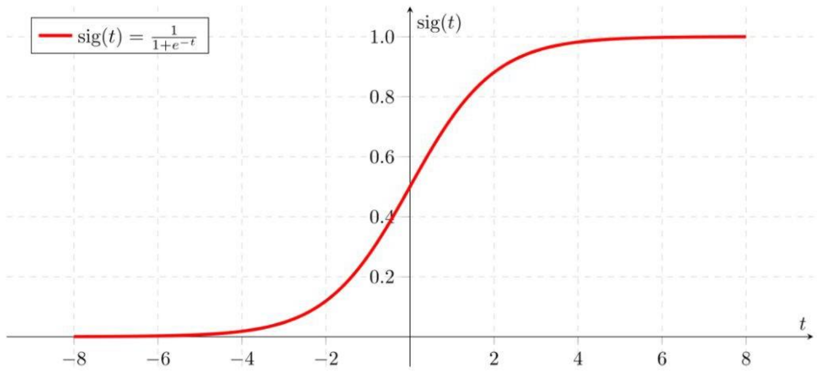
Submission.csv

This file contains the values that our model finally outputs after training. This file is then required to be uploaded on Kaggle to get the final test score.

Models

Logistic Regression

Logistic regression is very similar in regards to linear regression. They both take the inputs, scale them according to their weights and outputs a *y* value. However, linear regression cannot be used for classification tasks as it outputs a numerical value which is too high for a probabilistic classification task. Logistic regression overcomes this by squeezing the outcome of the linear regression between 0 and 1 using the sigmoid function.



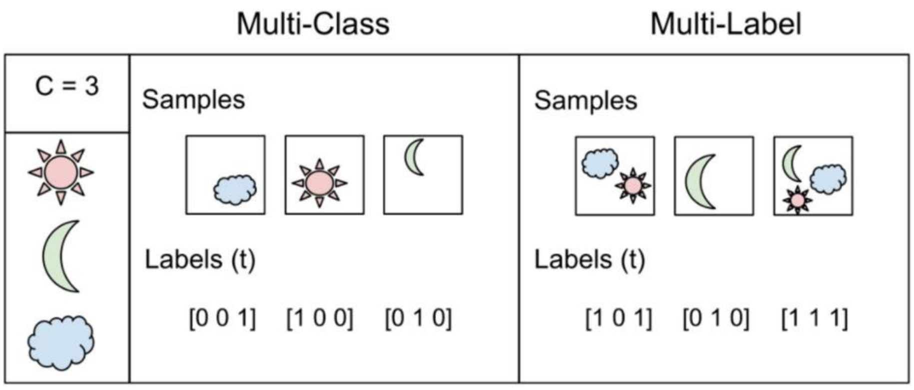
Plot of the sigmoid Function

$$f(x) = \frac{1}{1+e^{-x}}$$

In the above equation, x represents the output from a linear regression equation. The function is called as the Sigmoid function or the Logistic function

SVM

In its simple form, SVMs are applied on binary classification, dividing data points either in 1 or 0. For multilabel classification, the same principle is utilized. The multilabel problem is broken down to multiple binary classification cases. It basically divides the data points in class x and rest. Consecutively a certain class is distinguished from all other classes.



The number of classifiers necessary for one-vs-one multiclass classification can be retrieved with the following formula (with n being the number of classes):

$$\frac{n * (n-1)}{2}$$

In the one-vs-one approach, each classifier separates points of two different classes and comprising all one-vs-one classifiers leads to a multiclass classifier. The kernel takes two data points and calculates a distance score of those. This score is higher for closer data points and vice versa. Using this score helps to transform the data points to a higher-dimensional mapping, which reduces the computational effort and time and is especially useful for huge amounts of data. It prevents the need for a more complex transformation. We use a strategy that consists of fitting one model per target to convert the SVM to a multilabel classifier. This is an easy approach for extending models that do not support multi-target classification in a native way.

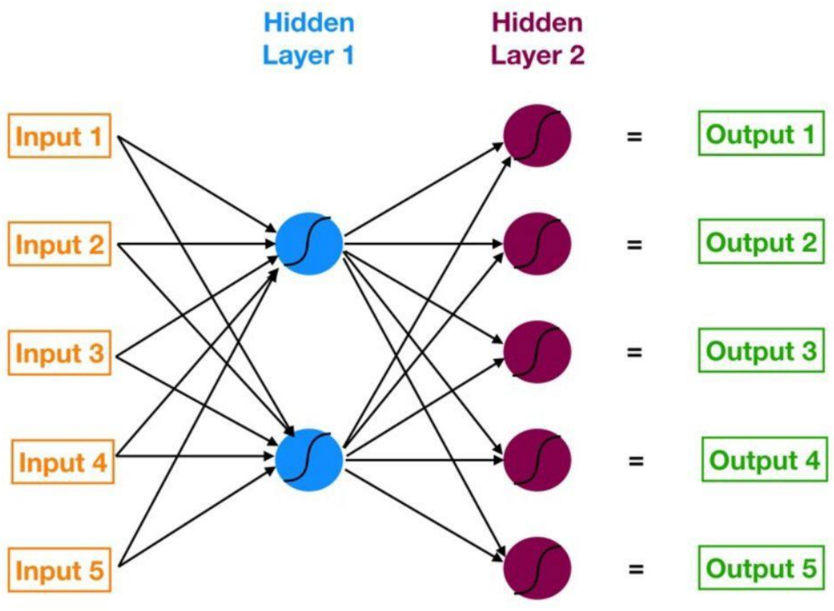
Radial Basis Function (RBF) -- Kernel function

$$k(x_i, x_j) = exp(-\gamma \|x_i - x_j\|^2)$$

The Regularization parameter tells the SVM optimization how much you want to avoid misclassifying each training example. For large values of C, the optimization will choose a smaller-margin hyperplane if that hyperplane does a better job of getting all the training points classified correctly.

Neural Network

A neural network is a model which is inspired from the way in which neurons in a biological brain work. Just like how a neuron receives input and propagates the signal throughout its network. We have the artificial neuron receiving and sending activations to its subsequent layers. In the brain, if a neuron is fired by appropriate synaptic inputs, the neuron will also shoot. In a similar fashion, there are input and output neurons on an artificial neural network, which are connected by weighted synapses. The weights influence how much of the neural network's forward propagation goes through. During backpropagation, the weights can then be adjusted, this happens through the learning phase.



Neural Network Pictorial Representation

We have the following components in a NN:

1. Neurons. Input neurons and output neurons represent the inputs and outputs of a neural network. There are no predecessor neurons for input neurons, but they have an output. Likewise, an output neuron does not have a successor neuron, it has inputs.
2. Connections and Weights. A neural network consists of links, with each link transmitting a neuron's output to another neuron's input. A weight is allocated to each link.

3. Propagation Function. This determines a neuron's input from the outputs of predecessor neurons. During the forward propagation stage of training, the propagation function is leveraged.
4. Learning Rule. A feature that modifies the weights of the ties is the learning rule. This helps to generate a favored output for the neural network for a given input.

The output coming from a neuron can be put into an equation as follows

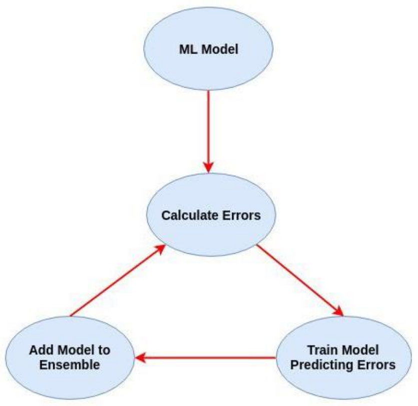
$$y = f(x_1 w_1 + x_2 w_2 + b)$$

Here y is the output from a neuron, *f(.)* is the activation function, *w* refers to the weights associated with an input x_i and *b* is the bias term which has no weights (usually considered as x_0)

XGBoost

XGBoost is a decision-tree-based ensemble Machine Learning algorithm that uses a gradient boosting framework. It can perform the three main forms of gradient boosting (Gradient Boosting (GB), Stochastic GB and Regularised GB) and it is robust enough to support fine-tuning and addition of regularisation parameters. This ensemble method seeks to create a strong classifier based on previous 'weaker' classifiers. By adding models on top of each other iteratively, the errors of the previous model are corrected by the next predictor, until the training data is accurately predicted or reproduced by the model.

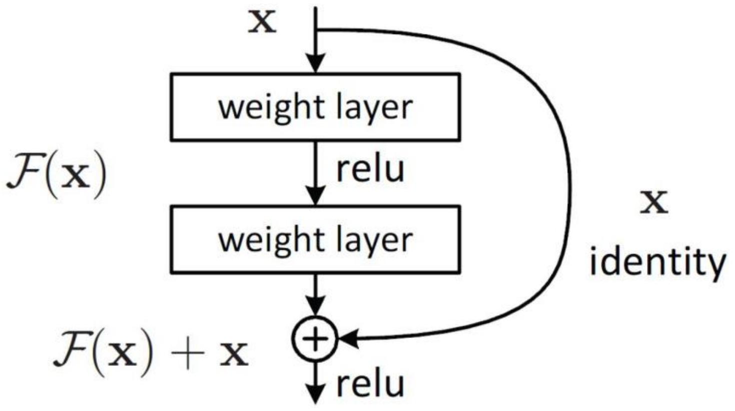
When using gradient boosting for regression, the weak learners are regression trees, and each regression tree maps an input data point to one of its leaves that contains a continuous score. XGBoost minimizes a regularized (L1 and L2) objective function that combines a convex loss function (based on the difference between the predicted and target outputs) and a penalty term for model complexity (in other words, the regression tree functions). The training proceeds iteratively, adding new trees that predict the residuals or errors of prior trees that are then combined with previous trees to make the final prediction. It's called gradient boosting because it uses a gradient descent algorithm to minimize the loss when adding new models.



XGBoost Flow

ResNet

We introduce a theorem in the realm of neural networks, called the Universal Approximation Theorem. This theorem states that a single layer of neural network is enough to model any multi-layer neural network, provided enough power. The layer could be huge, though and the network is vulnerable to overfitting the data. Therefore in the research community, there is a general theme that our network architecture needs to go deeper. Due to the infamous disappearing gradient problem, deep networks are difficult to learn, because the gradient is back-propagated to previous layers, repeated multiplication will make the gradient infinitely low. As a result, the output becomes saturated as the network goes further, or even continues to decline rapidly. ResNet's central principle is to add a so-called "identity shortcut connection" that skips one or more layers.


x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

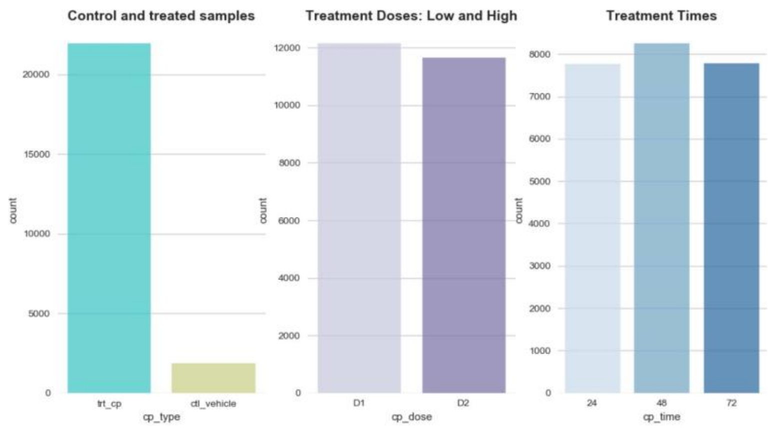
The "skip" connection in ResNet

It is easy for our network to learn the identity with the residual network. One explanation for skipping over layers is to prevent the issue of disappearing gradients by reusing activations from a previous layer before its weights are learned by the neighboring layer. The weights adapt to mute the upstream layer during training and amplify the previously skipped layer.

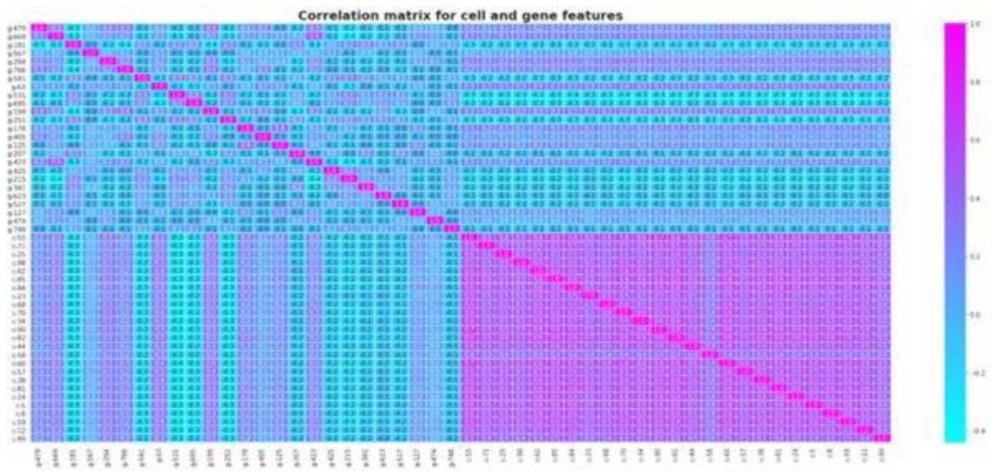
Implementation

EDA

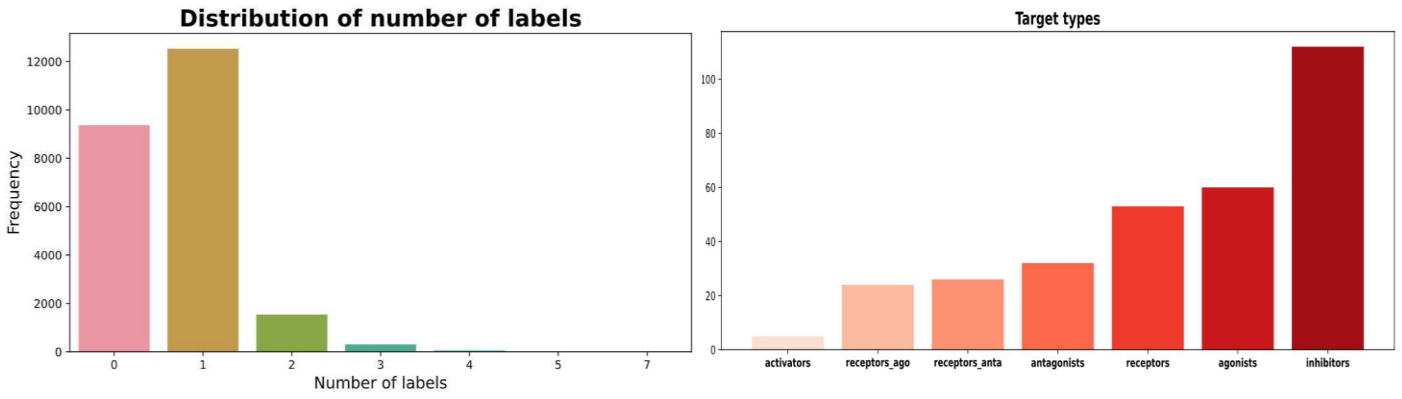
During EDA, we find out that the dataset is pretty clean with no empty cells or null values.The samples are also equally distributed amongst the various categorical variables present in our dataset, as seen in the figure below.



We find that there are correlations amongst the genes and also amongst the cell viability columns. We also find that some of the genes may be completely correlated to all the cells, forming bands

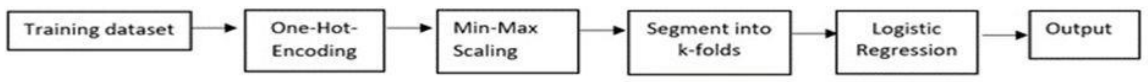


A large no of samples have either 0 or 1 MoA present and some as large as 7, as seen in the class frequency plot below.

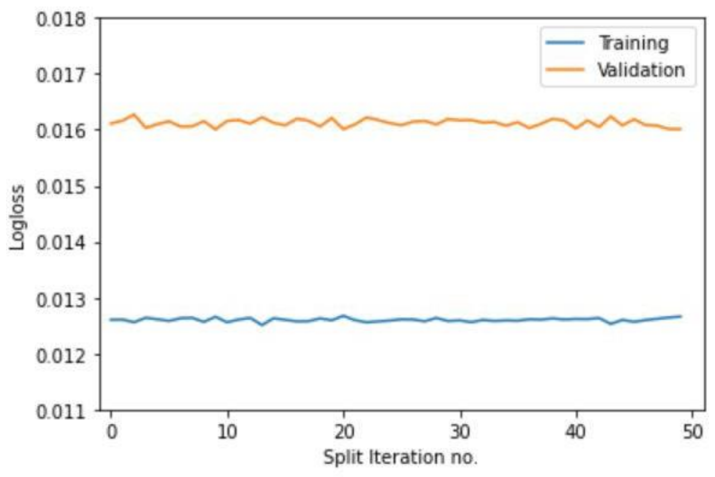


Due to sheer size and types of features we had. There were several steps that were common across the models. Since most of the models cannot handle categorical variables, we used one-hot encoding to convert these categorical variables into numerical data. This adds to the already high number of features. To reduce the number of features we used the Principal Component Analysis to reduce it to a fewer number of dimensions by making use of the correlations within the dataset.
Also, since it is better for the values of the features to remain within a particular range, to make the training more effective and not have certain features dominate due to its scale, we use normalization to rescale the features.

Logistic Regression

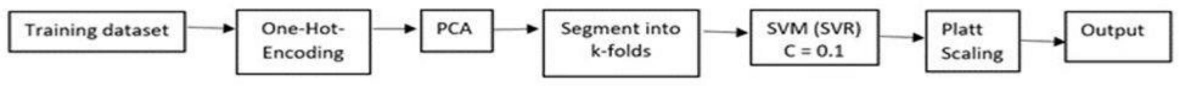


After the preprocessing steps, we run logistic regression with L2 regularization and C = 0.008 where C is the inverse of the regularization strength. We run with 50 folds and plot the cross validation and training score for each of these folds.



SVM

For SVM we take the training set and perform one-hot encoding to convert categorical data into numeric values for an efficient implementation. We have performed PCA for dimensionality reduction, in part to reduce the number of parameters and therefore the variance of the predictor. We segment it into 5-folds to get a more accurate cross validation score and less overfitting on the model. Then we implemented the SVM library using SVR class.



Tuning Parameters:

1) Kernel - After performing exploratory data analysis, and learning and testing with kernels like polynomial and RBF kernel, RBF(Radial Basis Function)/Gaussian kernel was opted as it performed better mapping of data points and gave a lower error score.
2) C - It is the regularization parameter and controls the tradeoff between smooth decision boundary and classifying training points correctly. We started training with C = 0.01, 0.03 and eventually increased it to set at 0.1 as for the larger value, a smaller margin became acceptable with the decision function classifying training points better.

3) Gamma - Here, gamma is set to ‘scale’ and computed as 1 / (n_features * xtrain.var()) , where we use gamma value obtained to consider points close to/far away from plausible separation line in calculation for the separation line.

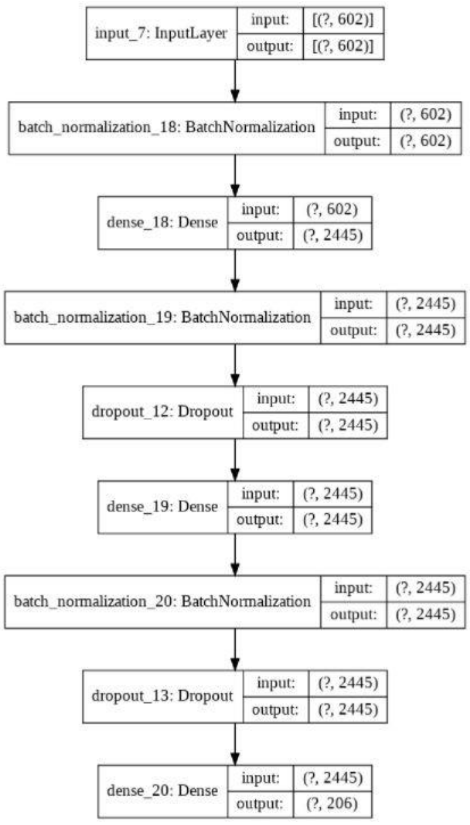
Next, Platt scaling was performed where we are essentially just performing logistic regression on the output of the SVM with respect to the true class labels. Here, we created a new data set that has the same labels, but with one dimension (the output of the SVM). Then we trained on this new data set, and fed the output of the SVM as the input to this calibration method, which returns a probability. The pre and post calibration log loss values were -

raw: 0.10190956138382551

cal: 0.017780453500434706

Neural Network

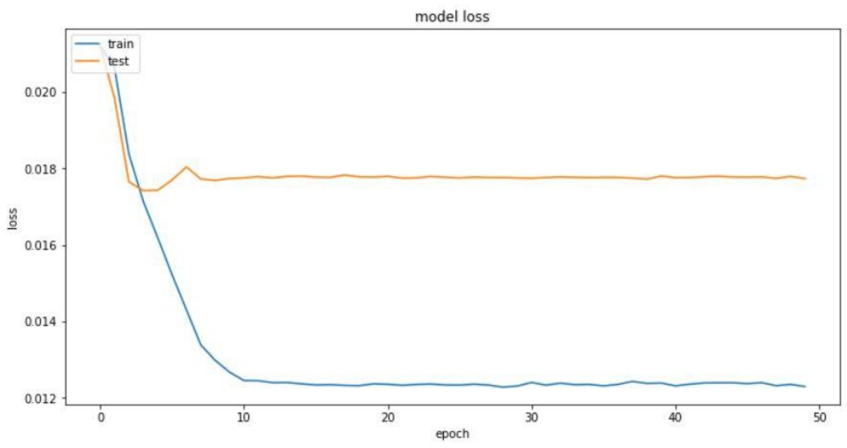
For the neural network we create a 3 layer Neural Network (2 hidden layers and an output layer). Before each of the hidden layers we have a Batch Normalization layer and a Dropout layer. The batch normalization layer speeds up training by normalizing the activations of each layer. This is similar to the normalization we do before inputting the dataset to the model. Batch normalization reduces the amount by what the hidden unit values shift around
Dropout is a regularization method that approximates training a large number of neural networks with different architectures in parallel.A single model can be used to simulate having a large number of different network architectures by randomly dropping out nodes during training. This is what is called a dropout and offers a very computationally cheap and remarkably effective regularization method to reduce overfitting and improve generalization error in deep neural networks of all kinds.



For tuning the hyperparameters of the model we make use of the grid search technique. This takes in an array of hyperparameters and runs the model on all the possible combinations of these hyperparameters and returns the best performing model. From the grid search we find out the best performing hyperparameters are:

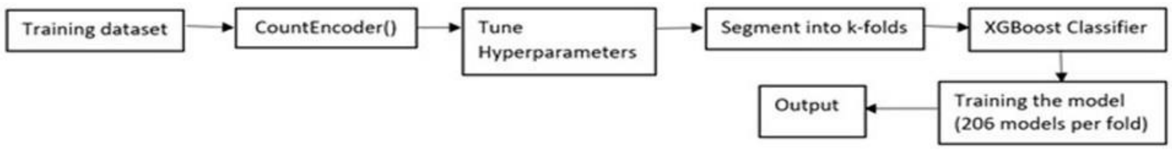
Hyperparameter	Value
Epochs	200
Batch Size	128
Optimizer	Adamax
Layer Value Initialization	uniform
Neurons	2445
Hidden Layer Activation	Softmax

The result obtained below is after a 5 fold cross validation.



XGBoost

XGBoost model stands for eXtreme Gradient Boosting, here we framed the problem as a n-binary classification problem (where n=206 and is the total number of classes). MultiOutputClassifier wrapper in sklearn is used here. CountEncoder() was used for count encoding of categorical features i.e. for a given categorical feature, it replaces the names of the groups with the group counts. It was used for encoding of categorical data in cp_type, cp_time and cp_dose.



The hyperparameter values were obtained using standard hyperparameter tuning procedures. These parameters were those that yielded the highest cross-validation score on the training set. The hyperopt library was used for this. We tuned hyperparameters like gamma, learning rate, max_depth of the decision trees, number of estimators etc.

params = {'classify_estimator_colsample_bytree': 0.6522,

'classify_estimator_gamma': 3.6975,

'classify_estimator__learning_rate': 0.0503,

'classify_estimator__max_delta_step': 2.0706,

'classify_estimator__max_depth': 10,

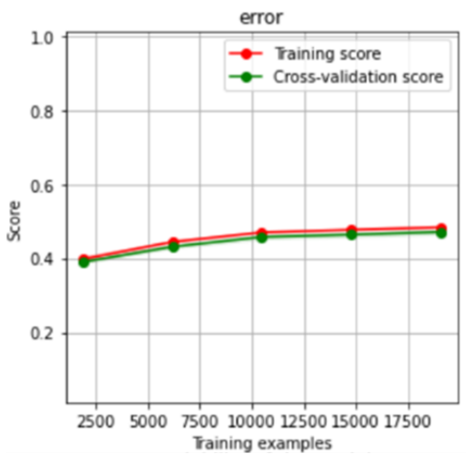
'classify_estimator__min_child_weight': 31.5800,

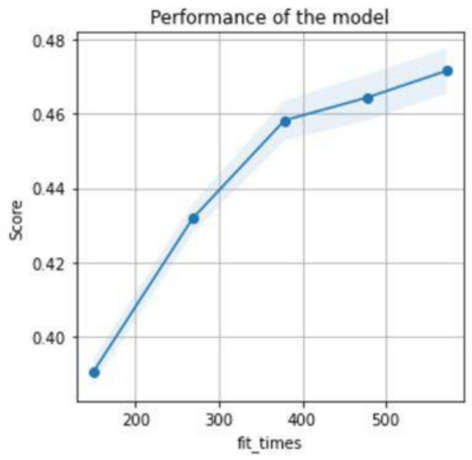
'classify__estimator__n_estimators': 166,

'classify__estimator__subsample': 0.8639

}

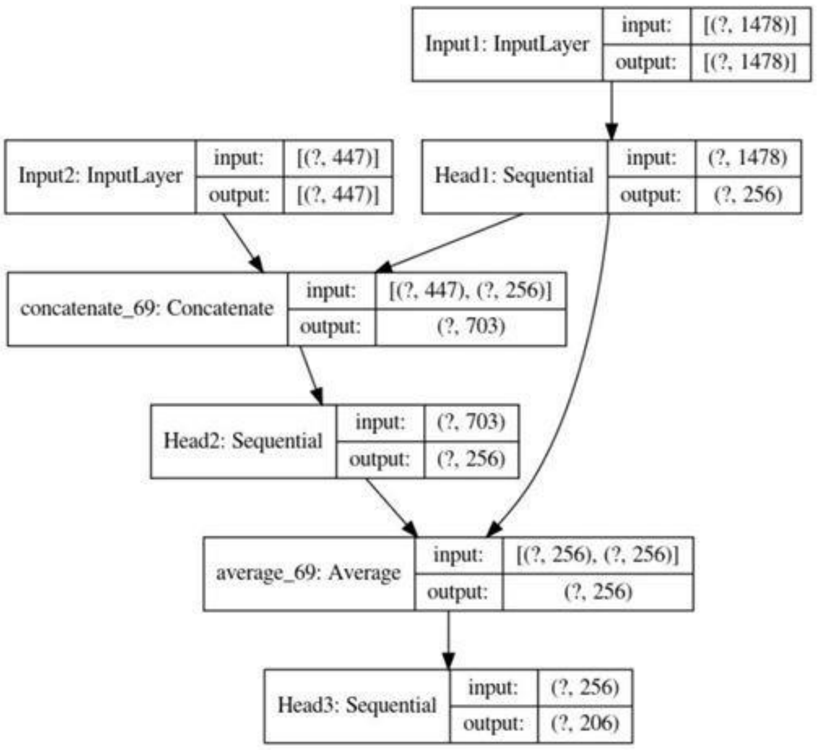
The model was then segmented into 5 folds and passed to the classifier, and then for training to obtain the output. Framing this problem as a binary classification problem had the disadvantage that we needed to train as many models as we have classes. For this problem this means training 206 models per fold, for the large number of features included in this dataset, which took approximately ~10 - 15 minutes.





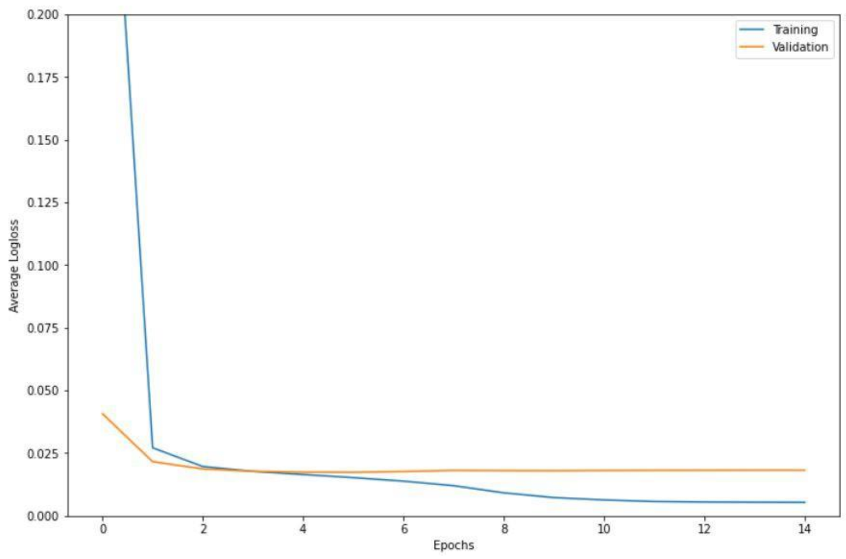
ResNet

The ResNet model consists of an ensemble of 3 Neural Network models. “Head1” in the figure given below is a NN which is trained on the preprocessed original dataset. “Head2” is NN which is trained on a dataframe which has a set of “most important features” concatenated with the original features. The most important feature selection is done by using recursive feature elimination on the original dataset, and are those columns which are believed to contribute more towards getting a better prediction. The output of these two layers, Head1 and Head2, are averaged over and then introduced to another NN, “Head3”, for training. This averaging between the outputs of the first NN and the second NN is the skip step which makes this a ResNet.


Input1: InputLayer | input: [(?, 1478)] | output: [(?, 1478)]
Input2: InputLayer | input: [(?, 447)] | output: [(?, 447)]
Head1: Sequential | input: (?, 1478) | output: (?, 256)
concatenate_69: Concatenate | input: [(?, 447), (?, 256)] | output: (?, 703)
Head2: Sequential | input: (?, 703) | output: (?, 256)
average_69: Average | input: [(?, 256), (?, 256)] | output: (?, 256)
Head3: Sequential | input: (?, 256) | output: (?, 206)

Hyperparameter	Head1	Head2	Head3
Optimizer	Adam		
Layer Value Initialization	glorot_uniform	glorot_uniform	lecun_normal
Hidden Neurons	512,256	512,512,256,256	256,206
Hidden Layer Activation	elu	elu,relu,elu,relu	selu,selu

The above table represents the hyperparameters used for each of the networks. The consequent parameters in any particular cell represent the varying values for the hidden layers in the network.



From the above plot, the result after 10-fold cross validation, we can see that the ResNet does a much better job at not overfitting the data as compared to the NN.

Results and Analysis

We have tabulated the results of the implementation in the table below.

	Cross Validation Score	Test Score
Logistic Regression	0.017125	0.02029
SVM	0.017800	0.02126
Neural Network	0.016399	0.01997
XGBoost	0.016724	0.01969
ResNet	**0.016840**	**0.01860**

We have ResNet as the best performing model with XGBoost being the next best model. These results do not come as a surprise upon analysis. The ensemble models such as XGBoost and ResNet are the best performing models as compared to the single models such as SVM, Logistic Regression, and simple NN. The dataset as such is quite complicated with over 800 features this can lead to a situation called "curse of dimensionality" This basically means that with a higher number of features things become harder to classify as they lose their distinguishability and also the models becomes more prone to overfitting. The ensemble methods do what the single models cannot perform. extract more information from the given dataset without overfitting the model to the dataset.

Discussions

Through this project we have gained a good knowledge on how ML algorithms can be applied to real world datasets and in our case, specific to the field of biology. There were several challenges that we faced during the execution of this project, grasping the pharmacological jargon, dealing with extensive runtimes for training our models, to name a few. Due to the complexity of the data, our computer systems were not enough to handle the computational processes of some of the models. We had to rely on publicly available cloud GPU services like Google Colab and Kaggle notebooks to help us in managing the training of our models. This taught us on how to use these services which will also be useful for future endeavours.

The amount of time spent per model was ~1 week. This was not a sufficient time to completely get the best out of each of the models. However, it was enough time to experiment, develop an understanding of the model, and to tune it’s hyperparameters on our own.

As for the future work on this project, we plan to improve the performance of the simple models. As of now the models perform poorly most probably because the dataset is still too complicated for them to handle. Upon finding a good way to reduce the complexity of the data for the simpler models, it might also help the more complex models to perform better also.