EE485 - Statistical Learning and Data Analytics

Predicting Result of a Soccer Match

Mehmet Berk Şahin 21703190

Turgut Can Mutlu 21702965

December 28, 2021

Abstract

Nowadays, European soccer games attract a lot of attention from the public. Premier League, Bundesliga, La Liga, Seria A and League 1 are the ones which football fans follow the most. To earn money on this love of football, some companies established their betting sites on the internet. In this websites, people try to predict the result of the game and put their money in favor of their predictions. If their predictions are wrong, they lose their money, otherwise they win money. In this project, we are trying to help these fans to earn money by designing a machine learning algorithm to estimate the results of the game correctly. Moreover, our project will help football coaches because looking at the past results, they will have a probabilistic idea about the outcome of the game. Thus, they may prepare their team in such a way that they will have the highest probability of winning the game. Our models predict the result as "win", "lose" or "draw".

1 Dataset

This project uses Machine Learning models to estimate the results of the game when given some statistics from the half time. Our dataset can be found here https://datahub.io/collections/football, which contains various datasets for different football leagues or tournaments ranging from national leagues to world cups. In this project, we used the top 5 European Leagues (Premier League, La Liga, Seria-A, Bundesliga, Ligue 1) that consists of the matches from 9 years (2009- 2019). Link to datasets can be accessed through *here*. Following are the specific links to the data that we have used in our project:

- https://datahub.io/sports-data/english-premier-league
- https://datahub.io/sports-data/spanish-la-liga
- https://datahub.io/sports-data/italian-serie-a
- https://datahub.io/sports-data/german-bundesliga
- https://datahub.io/sports-data/french-ligue-1

In addition to these odds, there are also closing odds, which are the last odds before match stars. They are the same as above but with an additional "C" character following the bookmarker abbreviation/Max/Avg. A sample from out raw dataset can be seen below:

	index	Div	Date	HomeTeam	AwayTeam	FTHG	FTAG	FTR	HTHG	HTAG	...	BbMxAHH	BbAvAHH	BbMxAHA	BbAvAHA	PSH	PSD	PSA
0	0	E0	2009-08-15	Aston Villa	Wigan	0	2	A	0.0	1.0	...	1.28	1.22	4.40	3.99	NaN	NaN	NaN
1	1	E0	2009-08-15	Blackburn	Man City	0	2	A	0.0	1.0	...	2.58	2.38	1.60	1.54	NaN	NaN	NaN
2	2	E0	2009-08-15	Bolton	Sunderland	0	1	A	0.0	1.0	...	1.68	1.61	2.33	2.23	NaN	NaN	NaN
3	3	E0	2009-08-15	Chelsea	Hull	2	1	H	1.0	1.0	...	1.03	1.02	17.05	12.96	NaN	NaN	NaN
4	4	E0	2009-08-15	Everton	Arsenal	1	6	A	0.0	3.0	...	2.27	2.20	1.73	1.63	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
17875	375	I1	26/05/2019	Inter	Empoli	2	1	H	0.0	0.0	...	2.11	2.05	1.85	1.81	1.39	5.35	7.81
17876	376	I1	26/05/2019	Roma	Parma	2	1	H	1.0	0.0	...	1.89	1.85	2.10	2.01	1.20	7.50	14.07
17877	377	I1	26/05/2019	Sampdoria	Juventus	2	0	H	0.0	0.0	...	2.03	1.96	1.95	1.90	3.92	3.98	1.93
17878	378	I1	26/05/2019	Spal	Milan	2	3	A	1.0	2.0	...	2.10	2.02	1.89	1.84	6.25	4.51	1.54
17879	379	I1	26/05/2019	Torino	Lazio	3	1	H	0.0	0.0	...	2.08	2.03	1.88	1.84	2.34	3.76	3.01

17880 rows × 78 columns

Figure 1: Sample Raw Dataset

1.1 Data Preprocessing

Fortunately, data we obtained was already structured so we did not need to put it into 2D array format manually, but there were lot of work to do. The first thing we did was to check whether there were any missing data in the dataset. As it can be seen in figure 1, it seems some features include lot of NaN values. Furthermore, to see the distribution of NaN values, we plotted the matrix in figure 2. Each column represents feature vectors. White rows represent no data entry, grey rows represent the data entry. In figure 2, some features have lot of NaN values, which occurs regularly, and they capture the majority of or half of the feature vectors. Therefore, we decided to remove these vectors as making imputation for these vectors would decrease our models' performance.

After removing the feature vectors with lot of missing entries, we did mean imputation for the remaining features. For example, as it can be seen in figure 2, BbAH has small number of missing values. If we removed this column, we would lose all the information regarding that feature, so instead we filled the missing values with the mean of the corresponding feature vectors. To build a model with good performance, the most important factor is data, so here we wanted to keep the

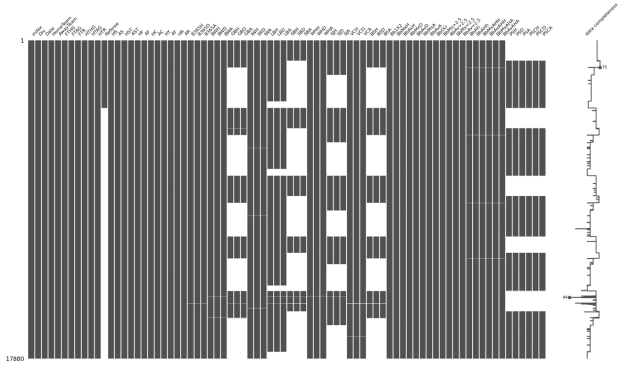


Figure 2: Matrix of Data Entries

information as much as possible. After the mean imputation, we centralized and standardized our dataset.

1.2 Data Analysis

After dropping the features with lots of missing values and doing standard scaling, still there were string type vectors, which are not useful for the model such as date and name of the league. We removed them and there were 46 features remaining. Next, we plotted the correlation matrix, which is given below in figure 3, to see the correlation between our feature vectors. This is important logistic regression classifiers, which we used, assume that feature vectors are linearly independent. Thus, linearly dependent feature vectors would give a poor performance. In figure 3, there are highly correlated features. For example, the correlation between "BbMx¿2.5" and "BbMx¡2.5" is very close to -1. This is expected because these are opposite of each other. If the odd assigned to one is low, the odd assigned to the other is high because it is more probable. Since that kind of feature pairs can be represented in a single feature vector, we discarded one of the pairs in the manual feature selection part. However, one should note that although they are correlated features, discarding one of them may not be the best choice. Maybe we need to combine them in some way. To do so, in PCA part, we implement PCA to do this transformation in a smarter way, which is capturing the most variance. Also, we removed some features among which there are arithmetic relationship. For instance, Free Kicks Conceded is the summation of fouls, offsides and any other offense committed and will always be equal to or greater than the number of fouls. Fouls make up most of the Free Kicks, so they are highly correlated. Instead of removing all the parameters and keeping the summation, we kept the parameters and remove the summation/free kicks conceded feature vector to not lose too much information.

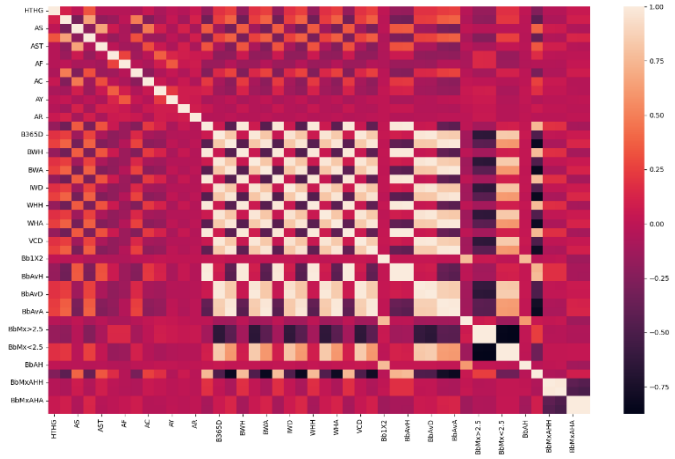


Figure 3: Correlation Matrix Heatmap of Feature Vectors

Another important thing about the dataset is its distribution. To see this to know the data better, we plotted the histogram of the feature vectors which can be seen below:

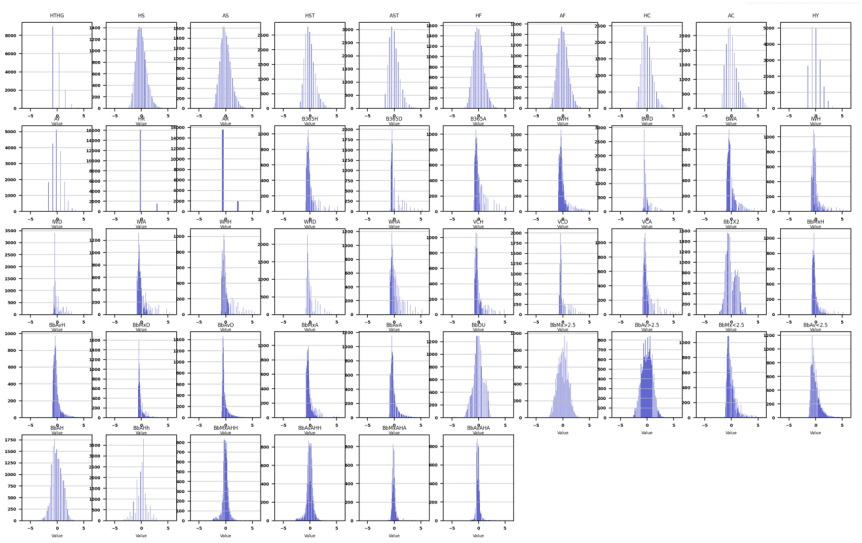


Figure 4: Distribution of Feature Vectors (Standardized and Centralized)

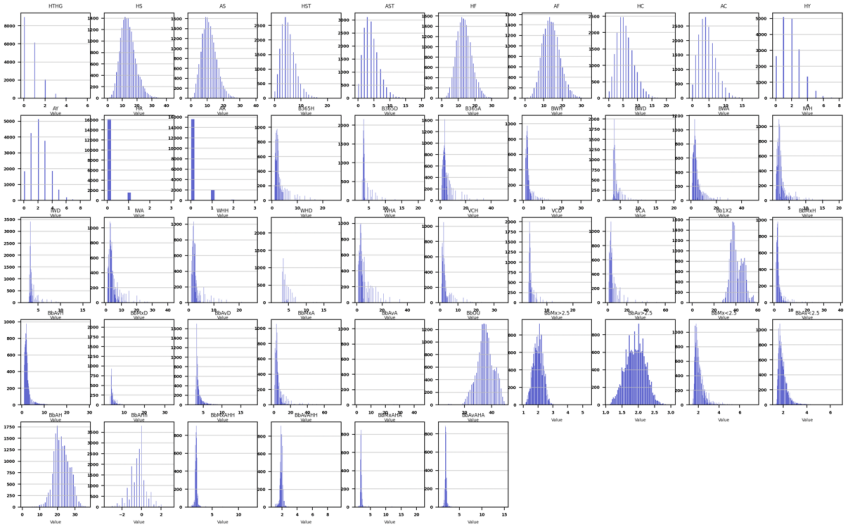
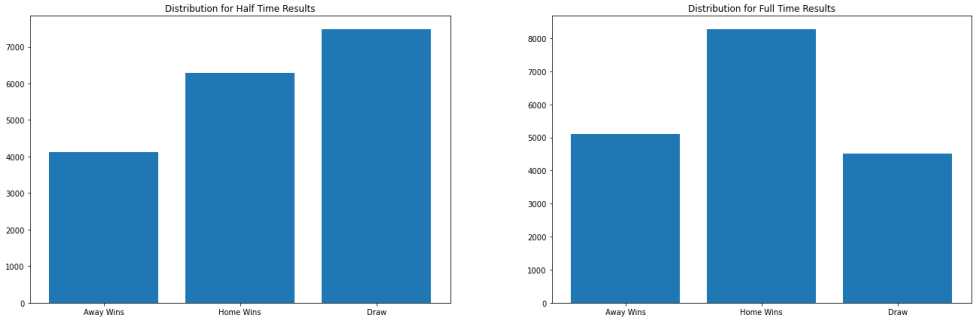


Figure 5: Distribution of Feature Vectors (No Standardization or Centralization)

In Figure 4, the distributions of some of the features are like Gaussian distribution but some of them are not. For example, AF, BbAv¿2.5, HS are very similar to a Gaussian distribution. However, features like BbMx¡2.5 AND BbAv¡2.5 are less likely to Gaussian distribution . Also, comparing figure 5 and figure 4, we see that standardization worked well.

1.3 Class Imbalance Problem

Considering the machine learning algorithms, most of them tend to result in a better accuracy when the dataset is distributed equally among the classes. One of our problem is that the dataset turned out to be less consistent than we thought. Analyzing the data, what we faced is that "class imbalance problem". For a soccer game, possible outcome probability for 3 different outcomes should be roughly 33,3%. However, this probability is independent from the number of samples



((a)) Half Time ((b)) Full Time

Figure 6: Distribution of Results for Half Time and Full Time Results

because the sample space consists of 3 different outcomes. When it comes to our situation, we are highly dependent on the distribution of the outcomes of each soccer match, and how evenly distributed they are in the dataset. For our project, especially Logistic Regression process has been affected by this problem because it is a probabilistic approach, which is dependent on the distribution of the dataset. Therefore, for this algorithm to give the best result, the given input data should be evenly distributed.

Above figure consist of information for the distributon of the half time results and full time results for each three possible outcomes. As in the figure, for half time, number of "draw" in the half time is the highest number of data. For the half time results, "Away Wins" results are conderably less than the "Draw" data. In here, the draw class is called **majority class** and away class is called **minority class**. Therefore, one can observe that "Draw" has higher impact on our algorithm since we are trying to guess the full time results from half time statistics. On the other hand, considering the full time results, number of "Home Team Wins" data is the one with the highest number of samples. The minority class is more difficult to anticipate because, by definition, there are few instances of this group. This indicates that learning the properties of instances from this class and distinguishing them from the majority class is more difficult for a model.

1.4 Manual Feature Selection

What is more is that, considering the information about the data as above, one should also take into account that domain knowledge about concept of a soccer match is important, we believe. There exists 46 features, but not all of them are useful for us. Consider the betting odds data; main purpose of our project is actually to beat their statistics, so we have omitted those features completely. For our purpose, we also have omitted the full time results since that are the ones that our algorithm tries to guess. Also, date of the match and referee of the game are inconclusive for statistical information since they are independent of effective features, we think. Here, we tried to choose the most informative features among the 46 features manually. To achieve this, we used correlation between the features, distribution of the features and our knowledge about football leagues. Considering the above results for data, we choosed HS, HST, AST, HS, AS, HC, AC, HTHG and HTAG (check Appendix for abbreviations) as our manual features for our purpose. We believe that our choice of features makes sense in a way. Let us take the AC (Away Corners) as example. Increasing number of away team corners means that away team has more chance to be exist in the penalty area of home team, which increase the chance of goal position, we think. So, taking these into account, we believe that it is worth to try and compare the results of the

algorithms with the choice of our features.

1.5 Principal Component Analysis

As opposed to the supervised learning models such as logistic regression, PCA is an unsupervised learning method. That is, it does not require any labelling. It is large dimensionality-reduction method, and it is used to avoid curse-of-dimensionality. In other words, it helps to eliminate number of features/dimensions in the dataset. The main purpose of PCA is transforming a large set of variables into a smaller one that still contains most of the information in the previous dataset. What do we mean by information? It is variance. PCA's goal is to extract new features which captures most of the variance in the existing dataset. The algorithm is the following:

1. Standardization of the original dataset. For this, mean and variance for each feature should be calculated, and dataset should be standardized by the following

$$X_{new} = \frac{x - \mu}{\sigma} \tag{1}$$

2. Covariance matrix computation. For a given dataset, covariance matrix can be reached by the following

$$Cov(x, y) = \frac{\sum(X_i - \bar{X})(Y_i - \bar{Y})}{N} \tag{2}$$

   where N is the number of data. For a given sample point, one can use the following

$$Cov(x, y) = \frac{\sum(X_i - \bar{X})(Y_i - \bar{Y})}{N - 1} \tag{3}$$

3. Compute the eigenvectors and eigenvalues of the covariance matrix to identify the principal components. In our case, the square matrix for the principal component is our covariance matrix. Let v be a vector, λ be the scalar and A be the square matrix, eigenvalue λ which satisfies the equation $Av = \lambda v$ is the eigenvalue that associated with vector in A. Then manipulating above equation, we can see that

$$(A - \lambda I)v = 0 \tag{4}$$

   We know that we are dealing with non-zero vectors, therefore only way to satisfy the equation is that

$$det(A - \lambda I) = 0 \tag{5}$$

   Solving the above equation yields to give principal components of our dataset with associated eigenvalues and eigenvectors.

4. Choose the highest n principal component which captures most of the variance in the old dataset.

Note that number of principal components, which is n, is chosen by the user, therefore it is a hyperparameter. How should we choose it? There are various ways to choose the number of principal components. There is no single way. Our way is to check the ratio of the variance that principal components capture, and we think capturing 95% of the variance of the previous dataset is sufficient.

2 Model Selection

To decide which model to use, one needs to choose the task s/he is doing. Mostly, it can be either regression task or classification task. In this project, we are doing classification of the match results so using models like linear regression would be meaningless. We can use Logistic Regression, Support Vector Machines, Naïve Bayes Classifier, Neural Networks, Random Forest, etc. Among these, we decided to choose Logistic Regression and Neural Networks, reasons for these choices will be explained in subsequent sections.

2.1 Logistic Regression

We believe that there can be linear relationship between the features and the outputs because for example, if the home team scores more goal, it is more likely to win or if the home team receives more red cards or yellow cards, it is more likely to lose. Thus, to represent these linear relations between the variables and the outputs in a probabilistic way, we decided to use logistic regression. Furthermore, logistic regression is much simpler version of neural networks. It can be considered as one output layer with activation function Sigmoid or SoftMax (single class classification or multiclass classification). Hence, before implementing one of the most powerful and complex algorithms, which is neural network, one should first try logistic regression since if it gives the desired performance, it is preferred to neural networks thanks to its simplicity and small computational cost (except for big datasets). Logistic regression can be considered as two-step process: forward pass and backward pass. Before getting into these, first, we need to define the matrices we will deal with.

$$X = \begin{bmatrix} x_{11} & \dots & x_{1m} & 1 \\ \vdots & \ddots & \vdots & \vdots \\ x_{m1} & \dots & x_{nm} & 1 \end{bmatrix} \text{ and } W = \begin{bmatrix} w_{11} & \dots & w_{1p} \\ \vdots & \ddots & \vdots \\ w_{m1} & \dots & w_{mp} \\ b_1 & \dots & b_p \end{bmatrix} \tag{6}$$

Row and column vectors of X are observation and feature vectors respectively. Row and column vectors correspond to feature vectors and feature weights for each class respectively. Thus, we our data consists of n samples, m features and p classes. Later we will just put the numbers into these unknowns to implement the model. In addition to that, there are additional terms in X and W matrices, these are 1 extension and $b_i \in \{1, 2, ..., p\}$ They are bias terms, which adds some nonlinearity in our model. First, we need to calculate the error and using this error we need to update the weights. This update will be done by gradient descent algorithm, which is simply given below

$$W(n+1) \leftarrow W(n) - \eta \nabla_W L(W(n)) \tag{7}$$

The idea of gradient descent is simply updating the weights in a such way that error decreases the most. To achieve this, one needs to find the gradient or partial derivative of E with respect to W. The constant parameter η in equation (2) is learning rate. It determines how big your updates are. If they are too small, then it will take longer to converge to the optimum point. If updates are too large, then gradient descent may not converge on the local minima and produces oscillatory behavior. However, with appropriate learning rate, gradient descent always converges in local minima. To calculate the gradient, one needs to do forward pass as the following:

1. First, we calculate the linear combination of feature vector as we did in linear regression case but now we need to do this for each class so we need to do matrix multiplication.

$$Z = -XW \tag{8}$$

2. Since we are doing multiclass logistic regression, we need to represent the results for different classes as a probability of observation belonging to each class. To achieve that, we need to take the SoftMax for each row

$$P_i = softmax(Z_i) = \frac{exp(Z_i)}{\sum_k^p Z_{ik}} \tag{9}$$

   Note that each row of P matrix should add up to 1 due to probability axiom 2.

3. Finally, we take the argmax for each row and choose the class with highest the probability. Note that doing this, we model the likelihood probability as the following

$$P(Y_i|X_i, W) = P_{i,k} = \frac{exp(Z_{i,Y_{i=k}})}{\sum_{k=0}^C Z_{ik}} \tag{10}$$

Whole training set can be represented as the following

$$P(Y|X, W) = \prod_{i=1}^{n} \frac{exp(-X_i W_{k=Y_i)})}{\sum_{k=0}^C -X_i W_k} \tag{11}$$

Thus, logistic regression tries to find the W that maximizes the above likelihood expression. Instead of doing that, we can take the logarithm and put a minus sign on it and use that expression as our loss function, which is given below

$$L(W) = \frac{1}{n}(\sum_{i=1}^{n}(X_i W_{k=Y_i}) + \sum_{i=1}^{n} log \sum_{k=0}^{p} exp(-X_i W_k)) \tag{12}$$

In case of our model is complex compared to the dataset or number of iterations we are doing is too much, which can cause overfitting, we add a small regularization term to the loss function. Final loss can be seen below

$$L(W) = \frac{1}{n}(\sum_{i=1}^{n}(X_i W_{k=Y_i}) + \sum_{i=1}^{n} log \sum_{k=0}^{p} exp(-X_i W_k)) + \mu||W||^2 \tag{13}$$

From that loss function, gradient can be calculated as follows

$$\begin{aligned} \nabla_{W_k} L(W) = \frac{1}{n}(\sum_{i=1}^{n} X_i^T I_{Y_{i=k}} - \sum_{i=1}^{n} X_i^T P_i) + 2\mu W &= \frac{1}{n}(X^T Y_{one-hot-encoded} - X^T P) + 2\mu W \\ &= \frac{1}{n}(X^T(Y_{one-hot-encoded} - P)) + 2\mu W \end{aligned} \tag{14}$$

After finding that, one can put this into the update in equation (2) term and the only thing remained is to implement this into a Python code, setting a proper learning rate, which was found by trial-and-error, and wait for algorithm to converge local minimum. Also note that, we put -1 term in equation (3), but it does not matter if there is any minus or not because we are using softmax function.

2.2 Feed-Forward Neural Network Classifier

In this project, our main purpose is doing multi-class classification. Earlier, we implemented logistic regression classifier and we thought that it was not powerful enough to capture the information in the data because we run logistic regression for 100,000 epochs and training accuracy did not reach to 100%. It remained at 49% and we concluded that logistic regression is not powerful enough to learn the data. Thus, we decided to implement a feed-forward neural network to increase the performance. Since our task is classification, we chose cross-entropy loss function given below:

$$L(W) = -\sum_{i=1}^{N} y_i ln(o_i) \tag{15}$$

where y_i's are ground truths and o_i's is predicted values. Obviously, the loss function depends on the network weights because predicted values are calculated by forward pass. Because of its stochasticity and its fast calculations, we chose mini-batch stochastic gradient descent as a learning algorithm. That is, at each epoch we shuffled the train set, divided it into the groups/mini batches, which are in the size of given batch size. Then, we trained our model for each batch and update the weights accordingly then continued with the other batches. This procedure continues until the validation loss converges (early stopping). To update the weights, we used the usual gradient descent rule

$$\Delta W(n) \leftarrow \Delta W(n-1) - \eta \frac{\partial L(W(n-1))}{\partial W} \tag{16}$$

Thus, we need to calculate the partial derivatives of the loss function with respect to all network weights. To achieve this, first, we need to calculate the loss and the update the weights accordingly, so we should first do forward pass.

2.3 Forward Pass

In our architecture, we chose sigmoid activation function as it is between 0 and 1 so it can represent a probabilistic value and the last layer has SoftMax activation function because our task is multi-class classification. Sigmoid and SoftMax are given below:

$$\phi(x) = \frac{1}{1+e^{-x}} \quad \text{and} \quad softmax(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{3} e^{x_j}} \tag{17}$$

Note that softmax function represents the probability of the i^{th} class for a sample. The general form of the matrices consisting of weights and inputs for a neuron are the following:

$$W_E = \begin{bmatrix} w_{11} & \dots & w_{1p} & \theta_1 \\ \vdots & \ddots & \vdots & \vdots \\ w_{n1} & \dots & w_{np} & \theta_n \end{bmatrix} \text{ and } X_E = \begin{bmatrix} w_{11} & \dots & w_{1n} \\ \vdots & \ddots & \vdots \\ x_{p1} & \dots & x_{pn} \\ -1 & \dots & -1 \end{bmatrix} \tag{18}$$

The additional column and row vectors are bias terms for each layer. Now, given the train data or mini batch, we can calculate the final output as follows:

$$Y = softmax(W_E^{(L)} \phi(W_E^{(L-1))} \phi(\dots \phi(W_E^1 X_E))) \tag{19}$$

L represents the layer's order in the neural network. Of course, at the output of each layer, one should add the -1 extensions to the output matrix of each layer except the last one. After those recursive calculations, we get the probabilistic values for each class thanks to SoftMax, and we can calculate the loss using cross-entropy. Now, we are ready to do backward propagation or backward pass to find the gradients of each weight and update them.

2.4 Backward Propagation

The algorithm for backward propagation in feed-forward neural networks with stochastic gradient descent is the following:

1. Pick a random sample set from the training data.
2. Using the network weights and the sample set, do the forward pass and calculate the outputs of each layer and the loss.
3. Find out the local gradients/delta values as the following:

$$\begin{aligned} \delta_1(n) &= \Gamma^{'}(V_1)e(n) \\ \delta_2(n) &= \Gamma^{'}(V_2)W_1^T\delta_1(n) \\ &\vdots \\ \delta_L(n) &= \Gamma^{'}(V_2)W_1^T\delta_1(n) \end{aligned} \tag{20}$$

   Where $V_L = W_E^{(L)}V_{L-1}$ and V_L denotes the activation function of the L^{th} layer (before getting into the nonlinear function). Thus, we need to define $V_0 = X_E$, which is our dataset. Additionally, $\Gamma^{'}(V_L)$ is a matrix whose elements are the derivative of the activation functions in L^{th} layer.

4. Update the weights as follows:

$$\begin{aligned} W_E^{(1)} &\leftarrow W_E^{(1)} + \eta\delta_1(n)x_E^{(1)^T} \\ W_E^{(2)} &\leftarrow W_E^{(2)} + \eta\delta_2(n)x_E^{(2)^T} \\ &\vdots \\ W_E^{(L)} &\leftarrow W_E^{(L)} + \eta\delta_L(n)x_E^{(L)^T} \end{aligned} \tag{21}$$

   where x_E^L is the L_{th} layer's extended input.

5. Compute the new loss.
6. If the algorithm converges or all epochs completed, quit. Else, go to step 1 and continue.

To calculate the first delta value, we need to take the derivative of the SoftMax function, but this will lead us to lot of calculations, which will increase the computational cost of our algorithm. Instead of doing this, we simply equated the first gradient to the error function e(n). This is always the case when the loss is cross-entropy, and the last activation function is SoftMax. We are going to prove this now. To begin with, we can write the cross-entropy loss function in an element-wise form given below

$$E = -\sum_{i=1}^{R} y_i ln(o_i) \tag{22}$$

where y_i's are ground truths and o_i's are predictions. And let Z_k be the activation potential of k^{th} output neuron. Thus, to calculate the local gradients, we need to find the partial derivative of E with respect to z_k, so we can write

$$\frac{\partial E}{\partial z_k} = -\sum_{i=1}^{R} y_i \frac{\partial ln(o_i)}{\partial z_k} \tag{23}$$

Using the chain rule, we get the following

$$\frac{\partial E}{\partial z_k} = -\sum_{i=1}^{R} \frac{y_i}{o_i} \frac{\partial o_i}{\partial z_k} \tag{24}$$

Note that $\frac{\partial o_i}{\partial z_k}$ is the derivative of the softmax function. Using equation (24), we can write it as follows

$$\frac{\partial o_i}{\partial z_k} = \begin{cases} o_i(1-o_i), & if\ i = k \\ -o_i o_k, & otherwise \end{cases} \tag{25}$$

Using this, we can divide the equation (19) as follows,

$$\frac{\partial E}{\partial z_k} = -\sum_{i \neq k}^{R} [\ \frac{y_i}{o_i}(-o_i o_k)] \ + \frac{y_k}{o_k} o_k(1-o_k) \tag{26}$$

and then we get

$$\frac{\partial E}{\partial z_k} = o_k \sum_{i \neq k}^{R} y_i - y_k(1-o_k) \tag{27}$$

Since we are doing multi-class classification, our labels or ground truths are in one-hot-encoded representation, so that implies

$$\sum_{i \neq k}^{R} y_i = 1 - y_k \tag{28}$$

Therefore, by using the equation (27) and (28), we get

$$\frac{\partial E}{\partial z_k} = o_k(1-y_k) - y_k(1-o_k) = o_k - y_k \tag{29}$$

Proof is completed. Using this aspect of cross-entropy and softmax, in software, we equated the delta values of the k^{th} output neurons to $o_k - y_k$. Other delta values are built on top of the first delta value thanks to the recursive nature of the algorithm. For the delta values of logistic hidden units, one needs to calculate the derivative of the sigmoid given below

$$\phi^{'}(x) = \phi(x)(1-\phi(x)) \tag{30}$$

After calculating the gradients, we updated the weights as in the backpropagation algorithm given above.

3 Performance Metrics

To evaluate the classification models, there are various performance metrics. Perhaps the simplest metric to use is classification accuracy, which is the ratio between the number of correctly classified samples and the total sample size. Sometimes a metric may not be appropriate for a specific task so, it may be misleading. For example, in anomaly detection, there can be 10 anomalies in 1000 samples. In this case, if one assumes there is no anomaly in the dataset, the accuracy of this prediction would be 99%. Although this performance seems almost perfect with respect to a classification accuracy metric, it is the worst performance as it did not make any correct prediction. To evaluate that kind of cases where the accuracy metric becomes insufficient and misleading, one should use different metrics such as precision, recall and F1 score, which will be explained next.

3.1 Confusion Matrix

Confusion Matrix is a tabular visualization of the ground-truth labels versus model predictions. Each row represents the instances in a predicted class and each columns represents the instances in an actual class. Confusion Matrix shows the model's performance for different classifications. A general confusion matrix for multi-class classification can be seen below:

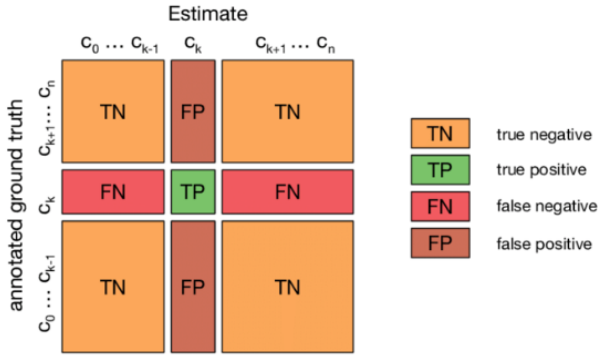


Figure 7: Confusion Matrix for Multi-Class Classification for class k [1]

In Figure 7, we see that the confusion matrix of a classification with n classes when considering the class k. To explain recall, precision and F1 score, we need to define TP, TN, FN, FP. Let's define the k^{th} class as the positive class, so we can make new definitions as the following: In the Testing Models section, we put our model's final performance on the test/unseen data with respect to different classification metrics.

4 Optimization

Hyperparameters are parameters whose value is used to control the learning process. As opposed to the other parameters such as weights which are determined by training, hyperparameters are determined by a data scientist. Some examples of hyperparameters are batch size, regularization coefficient, learning rate, number of layers, number of neurons, etc. Since they may affect the performance significantly, we spent lot of time to do this task. And the most reliable way to achieve this is via k-fold cross-validation.

4.1 K-Fold Cross Validation

If we evaluate the performance of our model by splitting our data into train and validation sets, our deductions will be probably incorrect due to the high variance and high bias of our data. If we try to optimize our model's performance on a single split, then we will overfit, that is, our hyperparameters will be the best ones for the given split, but they may be the worst one for another split. However, we do not want that. Instead, we want to optimize our model in such a way that the hyperparameters we found are the best ones for any dataset we found, namely, we want to generalize as much as we can with the dataset we have. To achieve this, we need to decrease the variance and the bias generated from a single split. The best way of doing this is Leave-One-Out Cross-Validation (LOOCV). LOOCV states that we need to train our models for every sample in the data and test them on this sample. That is, if we have n samples, we need to train a model on n-1 samples, test it on the remaining sample, do this n times for every sample in the data and take

the average of the performances. Due to its computational cost, LOOCV is very hard to do so to approximate this, we did 7-Fold Cross-Validation. Using this section we tuned our hyperparameters as explained in the next section.

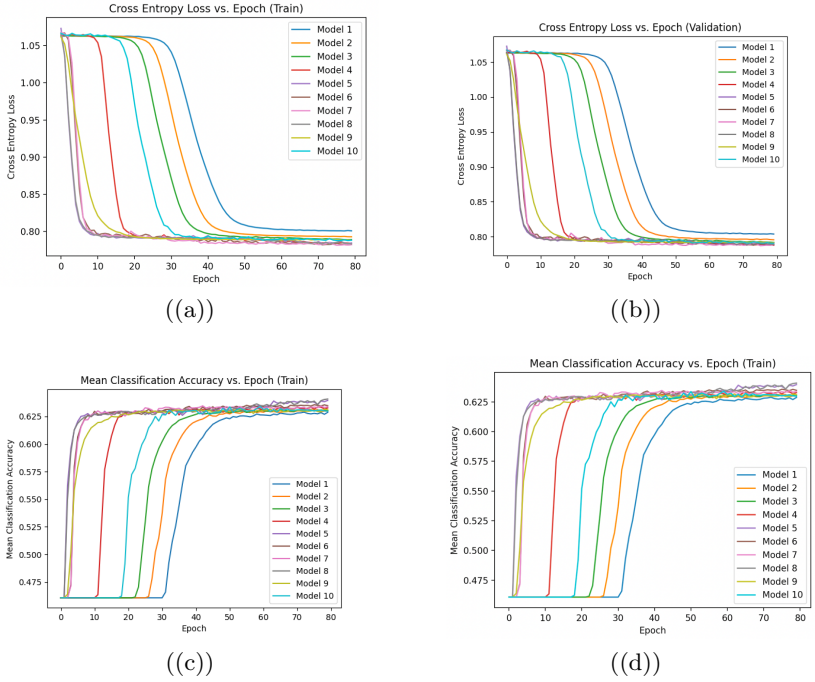
4.2 Hyperparameter Tuning

The hyperparameters for a neural network are number of layers, number of neurons at each layer, batch size, and learning rate. By trial and error, we observed that the optimum number of layers is 3 because if we increase or decrease the layer number, the accuracy decreased significantly so we decided to keep it constant in the tuning stage. Since we do not have any GPU or TPU to accelerate the training process, we did not do Grid Search, which is a brute force method. Instead, we set the hyperparameters that we thought that might be useful and run them with 5-fold cross-validation. Table of models and the results can be seen below:

Model No	1^{st} Layer Neuron No	2^{nd} Layer Neuron No	Batch Size	Learning Rate	Mean Classification Accuracy (Validation)	Cross-Entropy Loss (Validation)
1	32	16	500	0.2	0.625	0.8
2	64	16	500	0.2	0.628	0.795
3	128	16	500	0.2	0.63	0.792
4	128	32	250	0.2	0.63	0.788
5	128	64	250	0.2	0.637	0.789
6	128	32	250	0.5	0.634	0.788
7	128	32	250	0.5	0.635	0.789
8	128	64	250	0.5	0.637	0.788
9	256	128	500	0.5	0.63	0.792
10	256	64	500	0.2	0.628	0.79

Table 1: Average Results of Neural Network for 5-Fold Cross-Validation

To observe the learning procedure of the models, we took the average of each model's 5 learning curves and display them in a single plot given by following:



((a)) ((b))

((c)) ((d))

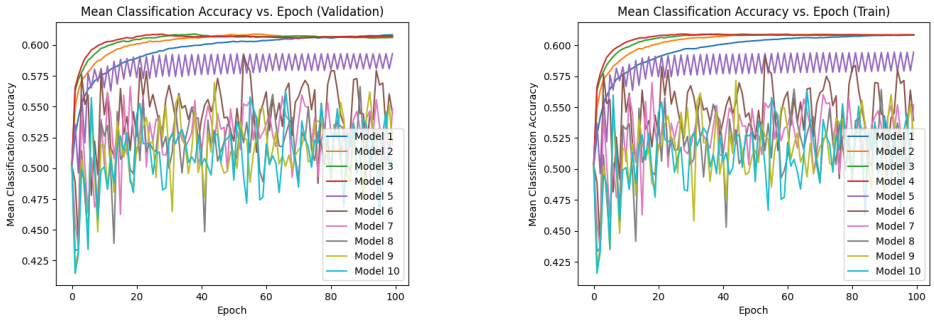
Figure 8: Cross Entropy Losses and Mean Classification Accuracies

As it can be seen from the above learning curves, model 8 is the fastest one in convergence and its average performance for 5-fold on the validation sets is the largest so we choose that model. For logistic regression, we did not use regularization coefficient because it was not able to learn the train data with 100% accuracy. Furthermore, we implemented full-batch learning in logistic regression, so we only tune the learning rate. The table and plots are given below:

Model No	Learning Rate	Mean Classification Accuracy (Validation)	Cross-Entropy Loss (Validation)
1	0.0025	0.6026	0.8709
2	0.005	0.6084	0.8561
3	0.0075	0.6071	0.8513
4	0.01	0.6072	0.8493
5	0.02	0.5925	0.8836
6	0.03	0.5304	1.1727
7	0.04	0.5481	1.479
8	0.05	0.5323	1.631
9	0.06	0.5131	2.7369
10	0.07	0.5273	2.2846

Table 2: Average Results of Logistic Regression for 5-Fold Cross-Validation

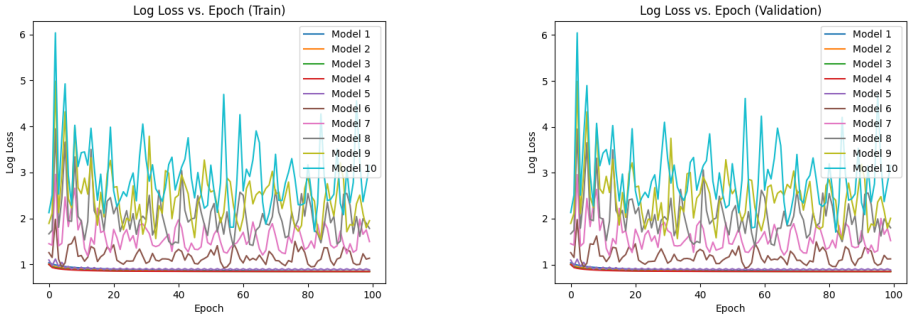
To observe the learning of the models, we took the average of each model's 5 learning curves taken from folds in CV and display them in a single plot given below:



((a)) ((b))

Figure 9: Mean Classification Accuracy for Validation (a) and Train (b)

In Figure 9, we clearly see that model with learning rates lower than 0.02 converges to a local minimum whereas models with learning rates greater than 0.02 do not converge to a local minimum due to the large step size. And model with learning rate 0.02 exhibits oscillatory behavior. Almost all converging models ended up at the same mean classification accuracy, but model 1's cross entropy loss is significantly larger compared to the others, so we removed it. Lastly, we choose the model 4 because it converged faster than the others. In Figure 10 reverse of this situation can be observed.



((a)) ((b))

Figure 10: Average Logarithmic Loss for Train (a) and Validation (b) for 5-fold CV

5 Testing Results

	Precision	Recall	F1 Score
A	0.3235	0.7165	0.4457
D	0.0911	0.1147	0.1015
H	0.4517	0.5478	0.4952

Table 3: Logistic Regression Without PCA

	Precision	Recall	F1 Score
A	0.4267	0.7625	0.5472
D	0.2045	0.1514	0.174
H	0.5725	0.8248	0.6759

Table 4: Feed Forward Neural Network

	Precision	Recall	F1 Score
A	0.3235	0.7165	0.4457
D	0.0911	0.1147	0.1015
H	0.4517	0.5478	0.4952

Table 5: Ensemble (Logistic Regression + Neural Network)

	Precision	Recall	F1 Score
A	0.3827	0.7928	0.5162
D	0.2364	0.3107	0.2685
H	0.5316	0.5778	0.5537

Table 6: Logistic Regression with Manual Feature Selection

	Precision	Recall	F1 Score
A	0.4432	0.72	0.5544
D	0.3384	0.3719	0.3544
H	0.6	0.7629	0.6717

Table 7: Feed-Forward Neural Network (with Manuel Feature Selection)

	Precision	Recall	F1 Score
A	0.3994	0.7918	0.531
D	0.2588	0.3185	0.2856
H	0.5543	0.6287	0.5891

Table 8: Ensemble with Manuel Feature Selection

Considering the test results, we saw that manual feature selection resulted in better accuracy and better results than algorithm's choice. It is not so surprising because we were able to see the distribution of each features in plots clearly. PCA may extract the noise in the dataset and this will cause our model to learn this random noise. We believe that difference in the results with the manual feature selection is caused by this. By this fact and by the power of our domain knowledge about a casual soccer game, we were able to feed our algorithms with effective features better than PCA.

6 Conclusion

All in all the purpose of this project was estimating the match results given the statistics of the teams and half time results. Considering the class imbalance in our dataset and random guess of a soccer match result being approximately 33,3%, we believe that our results are satisfying.

By this project we learned how to approach a real life machine learning problem and deal with it and how to do data preprocessing, hyperparameter tuning, optimization and internal mechanisms of machine learning algorithms.

The reason that we haven't put any confusion matrix to our report is that precision, recall and F1 score summarizes the confusion matrix. Also we wanted to save some space.

7 Appendix

7.1 Tables of Features

Feature Names	Meaning
Date	Match Date (dd/mm/yy)
Time	Time of match kick off
HomeTeam	Home Team
Div	League Divison
AwayTeam	Away Team
FTHG and HG	Full Time Home Team Goals
FTAG and AG	Full Time Away Team Goals
FTR and Res	Full Time Results (H=Home Win, D=Draw, A=Away Win)
HTHG	Half Time Home Team Goals
HTAG	Half Time Away Team Goals
HTR	Half Time Results (H=Home Win, D=Draw, A=Away Win)

Table 9: General Knowledge about the Match

Feature Names	Meaning
Attendance	Crowd Attendance
Referee	Match Referee
HS	Home Team Shots
AS	Away Team Shots
HST	Home Team Shots on Target
AST	Away Team Shots on Target
HHW	Home Team Hit Woodwork
AHW	Away Team Hit Woodwork
HC	Home Team Corners
AC	Away Team Corners
HF	Home Team Fouls Committed
AF	Away Team Fouls Committed
HFKC	Home Team Free Kicks Conceded
AFKC	Away Team Free Kicks Conceded
H0	Home Team Offsides
A0	Away Team Offsides
HY	Home Team Yellow Cards
AY	Away Team Yellow Cards
HR	Home Team Red Cards
AR	Away Team Red Cards
HBP	Home Team Booking Points (10 = yellow, 25 = red)
ABP	Away Team Booking Points (10 = yellow, 25 = red)

Table 10: Half Time Match Statistics

Feature Names	Meaning
B365H	Bet365 Home Win odds
B365D	Bet365 Draw odds
B365A	Bet365 Away Win odds
BSH	Blue Square Home Win odds
BSD	Blue Square Draw odds
BSA	Blue Square Away Win odds
BWH	Bet& Win Home Win odds
BWD	Bet & Win Draw odds
BWA	Bet & Win Away Win odds
GBH	Gamebookers Home Win odds
GBD	Gamebookers Draw odds
GBA	Gamebookers Away Win odds
IWH	Interwetten Home Win odds
IWD	Interwetten Draw odds
IWA	Interwetten Away Win odds
LBH	Ladbrokes Home Win odds
LBD	Ladbrokes Draw odds
LBA	Ladbrokes Away Win odds
PSH and PH	Pinnacle Home Win odds
PSD and PD	Pinnacle Draw odds
PSA and PA	Pinnacle Away Win odds
S0H	Sporting Odds Home Win odds
S0D	Sporting Odds Draw odds
S0A	Sporting Odds Away Win odds
SBH	Sportingbet Home Win odds
SBD	Sportingbet Draw odds
SBA	Sportingbet Away Win odds
SJH	Stan James Home Win odds
SJD	Stan James Draw odds
SJA	Stan James Away Win odds
SYH	Stanleybet Home Win odds
SYD	Stanleybet Draw odds
SYA	Stanleybet Away Win odds
VCH	VC Bet Home Win odds
VCD	VC Bet Draw odds
VCA	VC Bet Away Win odds
WHH	William Hill Home Win odds
WHD	William Hill Draw odds
WHA	William Hill Away Win odds

Feature Names	Meaning
Bb1X2	Betbrain bookmakers used to calculate match odds averages and maximums
BbMxH	Betbrain maximum home win odds
BbAxH	Betbrain average home win odds
BbMxD	Betbrain maximum draw odds
BbAxD	Betbrain average draw odds
BbMxA	Betbrain maximum away win odds
BbAxA	Betbrain average away win odds
MaxH	Market maximum home win odds
MaxD	Market maximum draw odds
MaxA	Market maximum away win odds
AvgH	Market average home win odds
AvgD	Market average draw odds
AvgA	Market average away win odds
BbOU	Betbrain bookmakers used to calculate over/under 2.5 goals averages and maximums
BbMx¿2.5	Betbrain maximum over 2.5 goals
BbAv¿2.5	Betbrain average over 2.5 goals
BbMx¡2.5	Betbrain maximum under 2.5 goals
BbAv¡2.5	Betbrain average under 2.5 goals
GB¿2.5	Gamebookers over 2.5 goals
GB¡2.5	Gamebookers under 2.5 goals
B365¿2.5	Bet365 over 2.5 goals
B365¡2.5	Bet365 under 2.5 goals
P¿2.5	Pinnacle over 2.5 goals
P¡2.5	Pinnacle under 2.5 goals
Max¿2.5	Market maximum over 2.5 goals
Max¡2.5	Market maximum under 2.5 goals
Avg¿2.5	Market average over 2.5 goals
Avg¡2.5	Market average under 2.5 goals
BbAH	BetBrain bookmakers used to Asian handicap averages and maximums
BbAHh	BetBrain size of handicap (home team)
AHh	Market size of handicap (home team) (since (2019/2020)
BbMxAHH	Betbrain maximum Asian handicap home team odds
BbAxAHH	Betbrain average Asian handicap home team odds
BbMxAHA	Betbrain maximum Asian handicap away team odds
BbAxAHA	Betbrain average Asian handicap away team odds
GBAHH	Gamebookers Asian handicap home team odds
GBAHA	Gamebookers Asian handicap away team odds
LBAHH	Ladbrokes Asian handicap home team odds
LBAHA	Ladbrokes Asian handicap away team odds
LBAH	Ladbrokes size of handicap (home team)
B365AHH	Bet365 Asian handicap home team odds
B365AHA	Bet365 Asian handicap away team odds
B365AH	Bet365 size of handicap (home team)
PAHH	Pinnacle Asian handicap home team odds
PAHA	Pinnacle Asian handicap away team odds

Feature Names	Meaning
MaxAHH	Market maximum Asian handicap home team odds
MaxAHA	Market maximum Asian handicap away team odds
AvgAHH	Market average Asian handicap home team odds
AvgAHA	Market average Asian handicap away team odds

Table 11: Betting Odds Data

7.2 Gantt Chart

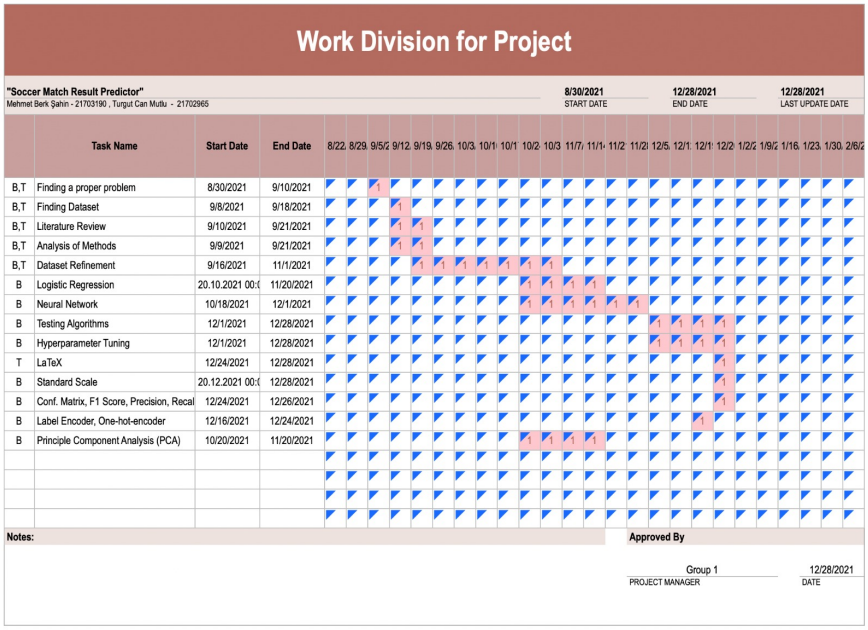


Figure 11: Gantt Chart

7.3 Code

```
import pandas as pd
import numpy as np
from os import path, makedirs, walk
from time import time
import matplotlib.pyplot as plt
import seaborn as sn

np.random.seed(4)
######################################### CLASSES ##########################################


class Layer(object):
```

```
    def __init__(self, neuron_num, activation, input_dim, std=0.01):

        self.neuron_num = neuron_num
        self.input_dim = input_dim
        self.weights = np.random.normal(size=(neuron_num, input_dim), loc=0, scale=std)
        self.bias = np.random.normal(size=(neuron_num, 1), loc=0, scale=std)
        self.weight_ext = np.concatenate((self.weights, self.bias), axis=1)

        self.activation = activation
        self.prev_weight = 0

        self.out = None
        self.delta = None
        self.prev_update = 0

    def output(self, X, activation):

        X_ext = np.concatenate((X, (-1 * np.ones(X.shape[0])).reshape((X.shape[0], 1))), axis=1
        act_pot = X_ext @ self.weight_ext.T
        if activation == "softmax":
            self.out = np.exp(act_pot - self.logsumexp(act_pot, axis=1, keepdims=True))
        elif activation == "sigmoid":
            self.out = np.exp(act_pot) / ( 1 + np.exp(act_pot) )
        else:
            self.out = np.tanh(act_pot)
        return self.out

    def logsumexp(self, X, axis=None, keepdims=False):

        X_max = np.amax(X, axis=axis, keepdims=keepdims)
        if X_max.ndim > 0:
            X_max[~np.isfinite(X_max)] = 0
        elif not np.isfinite(X_max):
            X_max = 0
        temp = np.exp(X - X_max)
        with np.errstate(divide='ignore'):
            z = np.sum(temp, axis=axis, keepdims=keepdims)
            output = np.log(z)
        if not keepdims:
            X_max = np.squeeze(X_max, axis=axis)
        output+=X_max
        return output


class NeuralNetwork(object):

    def __init__(self, early_stopping=None):
```

```
    self.layers = []

def add_layer(self, layer):
    self.layers.append(layer)

def forward_pass(self, X):
    temp = X
    for layer in self.layers:
        temp = layer.output(temp, activation=layer.activation)
    return temp

def backward_pass(self, X, Y, learning_rate, batch_size=0, moment=0):

    Y_pred = self.forward_pass(X)
    out_error = Y - Y_pred

    #Find the delta values
    for j in range(len(self.layers)-1, -1, -1):
        #Input layer
        if j == 0:
            second_layer, first_layer = self.layers[1], self.layers[0]
            weights = second_layer.weight_ext[:, :-1].reshape((second_layer.weight_ext.shap
            first_layer.delta = (weights.T @ second_layer.delta)
        else:
            second_layer, first_layer = self.layers[j], self.layers[j - 1]
            first_ext = np.concatenate(
                (first_layer.out, (-1 * np.ones(first_layer.out.shape[0])).reshape(first_la
                axis=1)
            act_pot = first_ext @ second_layer.weight_ext.T
            if second_layer.activation == 'sigmoid':
                sig = np.exp(act_pot) / ( 1 + np.exp(act_pot) )
                gamma = sig * ( 1 - sig )
            else: # softmax and cross entropy case
                gamma = 1
            # Output Layer
            if j == (len(self.layers)-1):
                second_layer.delta = np.ones((second_layer.neuron_num, out_error.shape[0]))
            #Hidden Layers
            else:
                third_layer = self.layers[j+1]
                weights = third_layer.weight_ext
                weights = weights[:,:-1].reshape((weights.shape[0], weights.shape[1]-1))
                second_layer.delta = gamma.T * (weights.T @ third_layer.delta)

    #Updating weights
    for j in range(len(self.layers)-1, -1, -1):
        if j != 0:
            second_layer, first_layer = self.layers[j], self.layers[j-1]
```

```
            X_ext = np.concatenate((first_layer.out, (-1 * np.ones(first_layer.out.shape[0]
            update = learning_rate * (second_layer.delta @ X_ext) / batch_size
            second_layer.weight_ext += update + moment * second_layer.prev_update
            second_layer.prev_update = update + moment * second_layer.prev_update # Keep th
        else:
            layer = self.layers[0]
            X_ext = np.concatenate((X, (-1 * np.ones(X.shape[0])).reshape((X.shape[0], 1))),
            if moment == 0:
                layer.weight_ext += learning_rate * ( layer.delta @ X_ext ) / batch_size
            else:
                change = learning_rate * ( layer.delta @ X_ext ) / batch_size
                layer.weight_ext += change + moment * layer.prev_update
                layer.prev_update = change + moment * layer.prev_update

def predict(self, X):
    Y_pred = self.forward_pass(X)
    Y_pred_raw = np.copy(Y_pred)
    Y_pred = np.argmax(Y_pred_raw, axis=1)
    return Y_pred_raw, Y_pred

def fit(self, X, Y, X_test, Y_test, batch_size, learning_rate, epoch, moment=0):

    cross_ent_hist_tst = []
    cross_ent_hist = []
    X_train = X
    Y_train = self.oneHotEncoder(Y=Y)
    X_vall = X_test
    Y_vall = self.oneHotEncoder(Y=Y_test)

    stop_train = 0

    mceT_history = []
    mce_history = []
    prev_cross_val = 0

    batch_num = X_train.shape[0] // batch_size

    for epc in range(epoch):

        #print(f"Epoch {epc+1}")

        indices = np.random.permutation(X.shape[0])
        #Y_pred_raw_tst, Y_pred_tst = self.predict(X_vall)
        #mce = np.mean(Y_pred_tst == Y_test, axis=0)

        for j in range(batch_num):

            index = indices[j * batch_size: (j + 1) * batch_size]
```

```
                X_batch, Y_batch = X_train[index], Y_train[index]
                self.backward_pass(X_batch, Y_batch, learning_rate, batch_size, moment)

            Y_pred_raw_tst, Y_pred_tst = self.predict(X_vall)
            Y_pred_raw, Y_pred = self.predict(X_train)



            mce = np.mean(Y_pred_tst == Y_test, axis=0)
            #print("MCE Validation Accuracy:", mce)
            mce_history.append(mce)
            mce2 = np.mean(Y_pred == np.argmax(Y_train, axis=1), axis=0)
            #print("MCE Train Accuracy:", mce2)
            mceT_history.append(mce2)

            cross_entropy_val = self.cross_entropy(Y_vall, Y_pred_raw_tst)
            #print("Cross Entropy Validation Loss:", cross_entropy_val)
            cross_ent_hist_tst.append(cross_entropy_val)

            cross_entropy = self.cross_entropy(Y_train, Y_pred_raw)
            #print("Cross Entropy Train Loss:", cross_entropy)
            cross_ent_hist.append(cross_entropy)


        return {"train_cross_entropy_history": cross_ent_hist,
                "val_cross_entropy_history": cross_ent_hist_tst,
                "train_mce_history": mceT_history,
                "val_mce_history": mce_history}


    def cross_entropy(self, Y, Y_pred):
        return -np.mean(np.sum(Y * np.log(Y_pred), axis=1), axis=0)

    def oneHotEncoder(self, Y=None):
        encoded = np.zeros((Y.shape[0], 3))
        for obs in range(Y.shape[0]):
            vector = np.zeros(3)
            vector[Y[obs]] = 1
            encoded[obs,:] = vector

        return encoded

class PCA (object):

    """
    A class to represent one PCA transformation.

    Attributes
```

```
    ----------
    explained_variances : list
        explained variance by new transformed features
    num_components : int
        number of principle components

    Methods
    -------
    fit(X):
        Outputs the transformed dataset
    transform(X):
        By using eigen vectors, calculates the transformed features

    """

    def __init__(self, num_components=1):
        self.explained_variances = []
        self.num_components = num_components

    def fit(self, X):
        X_scaled = StandardScale(X)
        #Find Covariance Matrix
        cov_matrix = np.cov(X_scaled.T)
        #Eigenvalue Decomposition
        self.eig_vals, self.eig_vec = np.linalg.eig(cov_matrix)
        for i in range(0, len(self.eig_vals)):
            self.explained_variances.append(self.eig_vals[i] / np.sum(self.eig_vals))
        #print(np.sum(self.explained_variances))
        #print(self.explained_variances)
        indices = sorted(range(len(self.eig_vals)), key=lambda k: self.eig_vals[k])
        self.eig_vals = sorted(self.eig_vals, reverse=True)
        self.eig_vec = [self.eig_vec[i] for i in indices]

        return self.transform(X_scaled)

    def transform (self, X):
        projections = []
        for component in range(self.num_components):
            projections.append(X @ self.eig_vec[component])
        return pd.DataFrame(data=projections)


class MultiClassLogisticRegression(object):
    """
    A class to represent one multiclass logistic regression operation.

    X is (N,M) matrix
    W is (M,C) matrix
```

```
Y is (N) matrix

N: Number of observations
M: Number of features
C: Number of classes

Attributes
----------
ler_rate : float
    learning rate of gradient descent algorithm
max_iter : float
    maximum number of iteration
reg : float
    regularization term in logistic regression log loss function
"""

def __init__(self, ler_rate=0.02, max_iter=1000, reg=0, scale=0.01):

    self.ler_rate = ler_rate * scale
    self.max_iter = max_iter
    self.reg = reg
    self.W = np.zeros((19, 3))

def log_loss(self, X, Y, W):

    H = - X @ W
    loss = (np.trace(X @ W @ Y.T) + np.sum(np.log(np.sum(np.exp(H), axis=1)))) / X.shape[0]
    return loss

def get_gradient(self, X, W, Y):

    H = - X @ W
    #Compute the softmax in logspace for numerical stability
    y_prob =  self.softmax(H, axis=1)#softmax(H, axis=1)
    #softmax(H, axis=1)#np.exp(H) / (np.sum(np.exp(H), axis=1, keepdims=True) * np.ones(H.s
    grad = (X.T @ (Y - y_prob)) + 2 * self.reg * W
    #print(y_prob)
    return grad

def oneHotEncoder(self, Y):

    classes = np.unique(Y)
    self.classes = classes
    encoded_y = np.zeros((len(Y), len(classes)))
    for i, category in enumerate(classes):
        indices = np.where(Y == category)
        encoded = np.zeros(len(classes))
        encoded[i] = 1
```

```
        encoded_y[indices] = encoded

    return encoded_y

def fit(self, X, Y, X_val, Y_val):
    y_train, Y_vall = self.oneHotEncoder(Y), self.oneHotEncoder(Y_val)
    X_train, X_vall = np.concatenate((X, np.ones((X.shape[0], 1))), axis=1), np.concatenat
    W = np.zeros((X_train.shape[1], y_train.shape[1]))
    loss_hist, loss_val = [], []
    acc_train, acc_test = [], []
    for iter in range(self.max_iter):

        W -= self.ler_rate * self.get_gradient(X_train, W, y_train)
        loss_hist.append(self.log_loss(X_train, y_train, W))
        loss_val.append(self.log_loss(X_vall, Y_vall, W))
        self.W = W
        y_pred = self.predict(X_val)[1]
        #print(y_pred)
        y_predt = self.predict(X)[1]
        acc = np.sum(y_pred == np.argmax(Y_vall, axis=1), axis=0) / y_pred.shape[0]
        acct = np.sum(y_predt == np.argmax(y_train, axis=1), axis=0) / y_train.shape[0]

        acc_train.append(acct)
        acc_test.append(acc)

    return np.array(loss_hist), np.array(loss_val), np.array(acc_train), np.array(acc_test

def predict(self, X):

    X = np.concatenate((X, np.ones((X.shape[0], 1))), axis=1)
    H = - X @ self.W
    y_prob = np.exp(H) / (np.sum(np.exp(H), axis=1, keepdims=True) * np.ones(H.shape))
    return y_prob, self.classes[np.argmax(y_prob, axis=1)]

def logsumexp(self, X, axis=None, keepdims=False):

    X_max = np.amax(X, axis=axis, keepdims=keepdims)
    if X_max.ndim > 0:
        X_max[~np.isfinite(X_max)] = 0
    elif not np.isfinite(X_max):
        X_max = 0
    temp = np.exp(X - X_max)
    with np.errstate(divide='ignore'):
        z = np.sum(temp, axis=axis, keepdims=keepdims)
        output = np.log(z)
    if not keepdims:
        X_max = np.squeeze(X_max, axis=axis)
    output+=X_max
```

```
        return output

    def softmax(self, X, axis=None):
        #For numerical stability, calculate the softmax in logspace
        return np.exp(X-self.logsumexp(X, axis=axis, keepdims=True))

##########################################################################################

######################################## FUNCTIONS ######################################

def oneHotEncoder(Y):

    classes = np.unique(Y)
    encoded_y = np.zeros((len(Y), len(classes)))
    for i, category in enumerate(classes):
        indices = np.where(Y == category)
        encoded = np.zeros(len(classes))
        encoded[i] = 1
        encoded_y[indices] = encoded

    return encoded_y

def StandardScale(X):
    X -= np.mean(X, axis=0)
    return X / np.std(X, axis=0)

#Gaussian Naive Bayes Classifier

def train_classifier(clf, X_train, y_train):
    start = time()
    clf.fit(X_train, y_train)
    end = time()
    print("Model trained in {:2f} seconds".format(end - start))


def predict_labels(clf, features, target):
    start = time()
    y_pred = clf.predict(features)
    end = time()
    print("Made Predictions in {:2f} seconds".format(end - start))

    acc = sum(target == y_pred) / float(len(y_pred))

    return acc


def model(clf, X_train, y_train, X_test, y_test):
    train_classifier(clf, X_train, y_train)
```

```
    acc = predict_labels(clf, X_train, y_train)
    print("Training Info:")
    print("-" * 20)
    print("Accuracy:{}".format(acc))

    f1, acc = predict_labels(clf, X_test, y_test)
    print("Test Metrics:")
    print("-" * 20)
    print("Accuracy:{}".format(acc))


def derive_clean_sheet(src):
    arr = []
    n_rows = src.shape[0]

    for data in range(n_rows):

        # [HTHG, HTAG]
        values = src.iloc[data].values
        cs = [0, 0]

        if values[0] == 0:
            cs[1] = 1

        if values[1] == 0:
            cs[0] = 1

        arr.append(cs)

    return arr

def dataClassifier(X_train):
    dfA = pd.DataFrame()
    dfH = pd.DataFrame()
    dfD = pd.DataFrame()

    X_train = pd.DataFrame(X_train)
    for i in range(len(X_train)):

        if X_train.iloc[i]['HTR'] == "A":
            newRow = X_train.iloc[i]
            dfA = dfA.append(newRow)
        elif X_train.iloc[i]['HTR'] == "H":
            newRow = X_train.iloc[i]
            dfH = dfH.append(newRow)
        elif X_train.iloc[i]['HTR'] == "D":
            newRow = X_train.iloc[i]
```

```
            dfD = dfD.append(newRow)

    # dfA = dfA.drop("HTR", axis=1)

    # dfH = dfH.drop("HTR", axis=1)

    # dfD = dfD.drop("HTR", axis=1)

    return dfA, dfH, dfD

def meanAndVar(x, y, z):
    '''FOR HOME WINS'''
    mValHome = x.mean()
    vValHome = x.var()

    '''FOR AWAY WINS'''
    mValAway = y.mean()
    vValAway = y.var()

    '''FOR DRAW'''
    mValDraw = z.mean()
    vValDraw = z.var()

    return mValHome, mValDraw, mValAway, vValHome, vValDraw, vValAway

def get_prior_probs(h, a, d, trainData):
    # Prob of Home wins / whole data
    priorHome = len(h) / len(trainData)

    # Prob of Away wins / whole data
    priorAway = len(a) / len(trainData)

    # Prob of Draw / whole data
    priorDraw = len(d) / len(trainData)

    return priorHome, priorAway, priorDraw

def calculate_pdf(newInputValue, trainMean, trainVar):
    probDensity = (1 / np.sqrt(2 * np.pi * trainVar)) / np.exp(-1 / 2 * ((newInputValue - trai
    return probDensity


def prior_prob_frame(mValHome, mValDraw, mValAway, vValHome, vValDraw, vValAway, X_train):
    checkIndex = ["HTHG", "HTAG", "HST", "AST", "HS", "AS"]

    priorFrameforHome = pd.DataFrame()
    priorFrameforAway = pd.DataFrame()
    priorFrameforDraw = pd.DataFrame()
```

```
    for i in checkIndex:
        x = calculate_pdf(X_train[i], mValHome[i], vValHome[i])
        priorFrameforHome[i] = x

    for i in checkIndex:
        x = calculate_pdf(X_train[i], mValAway[i], vValAway[i])
        priorFrameforAway[i] = x

    for i in checkIndex:
        x = calculate_pdf(X_train[i], mValDraw[i], vValDraw[i])
        priorFrameforDraw[i] = x

    return priorFrameforHome, priorFrameforAway, priorFrameforDraw

def final_result_probabilities(probFrameForHome, probFrameForAway, probFrameForDraw):
    productAway = probFrameForAway.product(axis=1)

    productHome = probFrameForHome.product(axis=1)

    productDraw = probFrameForDraw.product(axis=1)

    draw = productDraw * priDraw
    away = productAway * priAway
    home = productHome * priHome

    drawResult = np.log(draw)

    awayResult = np.log(away)

    homeResult = np.log(home)

    finalProbFrame = pd.DataFrame()

    finalProbFrame['Home Wins'] = homeResult

    finalProbFrame['Away Wins'] = awayResult

    finalProbFrame['Draw'] = drawResult

    return finalProbFrame

def accuracy_calculator(finalProbFrame, Y_train):
    finalArr = finalProbFrame.to_numpy()

    finalLoc = []
    for i in range(len(finalArr)):
        finalLoc.append(np.argmax(finalArr[i]))
```

```
    finalLocName = []
    for i in finalLoc:
        if i == 0:
            finalLocName.append('H')
        elif i == 1:
            finalLocName.append('A')
        elif i == 2:
            finalLocName.append('D')

    arrPred = np.array(finalLocName)
    arrTrain = np.array(Y_train)

    accuracy = sum(arrPred == arrTrain) / arrPred.shape[0]
    return accuracy, finalLoc

def train_test_split(X, y, split=0.2, shuffle=False):
    np.random.seed(10) # Put a random seed so that there will be no data leakage
    indices = np.arange(0, len(y))
    if shuffle:
        np.random.shuffle(indices)
    test_limit = round(len(y) * split)
    X_test, y_test = X[indices[-test_limit:]], y[indices[-test_limit:]]
    X_train, y_train = X[indices[:-test_limit]], y[indices[:-test_limit]]
    return X_train, y_train, X_test, y_test

def label_encoder(y):
    temp = y
    unique = np.unique(temp)
    for i in range(len(unique)):
        temp = np.where(temp == unique[i], i, temp)
    return temp

def confusion_matrix(y_pred, y):
    class_num = len(np.unique(y))
    matrix = np.zeros((class_num, class_num))
    for pred in range(len(y)):
        matrix[y[pred]][y_pred[pred]] += 1
    return matrix

def train_logreg_cv(params, X, y, k):

    log_t_losses = []
    log_v_losses = []
    mce_acc_train = []
    mce_acc_test = []

    kfolds = np.random.choice(np.arange(0, len(X)), size=(k, len(X) // k), replace=False)
```

```
for i in range(len(params)):
    learning_rate = params[i]
    logv_loss, logt_loss, mce_train, mce_test = 0, 0, 0, 0

    for fold in range(k):
        ##########################################   LOGISTIC REGRESSION    ############
        x_val, y_val = X[kfolds[fold]], y[kfolds[fold]]
        x_train, y_train = X[(np.delete(kfolds, fold, axis=0)).reshape(-1)], y[(np.delete(
        model = MultiClassLogisticRegression(ler_rate=learning_rate, max_iter=50)
        logt, logv, acctr, acc = model.fit(x_train, y_train, x_val, y_val)


        logv_loss += logv / k
        logt_loss += logt / k
        mce_train += acctr / k
        mce_test += acc / k

    print("#" * 20, f"Model {i + 1}", "#" * 20)
    print("Mean Logarithmic Loss (Train):", logv_loss[-1])
    print("Mean Logarithmic Loss (Validation):", logt_loss[-1])
    print("Mean Classification Accuracy (Train):", mce_train[-1])
    print("Mean Classification Accuracy (Validation):", mce_test[-1])

    log_t_losses.append(logt_loss)
    log_v_losses.append(logv_loss)
    mce_acc_train.append(mce_train)
    mce_acc_test.append(mce_test)

plt.figure()
plt.title("Log Loss vs. Epoch (Train)")
plt.xlabel("Epoch")
plt.ylabel("Log Loss")
for i in range(len(params)):
    plt.plot(log_t_losses[i])
plt.legend([f"Model {i + 1}" for i in range(len(params))])
plt.show()

plt.figure()
plt.title("Log Loss vs. Epoch (Validation)")
plt.xlabel("Epoch")
plt.ylabel("Log Loss")
for i in range(len(params)):
    plt.plot(log_v_losses[i])
plt.legend([f"Model {i + 1}" for i in range(len(params))])
plt.show()

plt.figure()
```

```
    plt.title("Mean Classification Accuracy vs. Epoch (Train)")
    plt.xlabel("Epoch")
    plt.ylabel("Mean Classification Accuracy")
    for i in range(len(params)):
        plt.plot(mce_acc_train[i])
    plt.legend([f"Model {i + 1}" for i in range(len(params))])
    plt.show()

    plt.figure()
    plt.title("Mean Classification Accuracy vs. Epoch (Validation)")
    plt.xlabel("Epoch")
    plt.ylabel("Mean Classification Accuracy")
    for i in range(len(params)):
        plt.plot(mce_acc_test[i])
    plt.legend([f"Model {i + 1}" for i in range(len(params))])
    plt.show()

def train_neural_cv(params, X, y, k):

    kfolds = np.random.choice(np.arange(0, len(X)), size=(k, len(X) // k), replace=False)

    models = []

    t_cross, v_cross, t_mce, v_mce = [], [], [], []


    for model in range(len(params)):

        n1, n2, batch_size, learning_rate = params[model]
        train_cross, val_cross, train_mce, val_mce = [0,0,0,0]

        for fold in range(k):

            #print("*"*20, f"Fold {fold+1}", "*"*20)

            x_val, y_val = X[kfolds[fold]], y[kfolds[fold]]
            x_train, y_train = X[(np.delete(kfolds, fold, axis=0)).reshape(-1)], y[(np.delete(

            net = NeuralNetwork(early_stopping=30)
            net.add_layer(Layer(neuron_num=n1, input_dim=X_scaled.shape[1], activation="sigmoi
            net.add_layer(Layer(neuron_num=n2, input_dim=n1, activation="sigmoid"))
            net.add_layer(Layer(neuron_num=3, input_dim=n2, activation="softmax"))
            history = net.fit(X=x_train, Y=y_train, X_test=x_val, Y_test=y_val,
                              batch_size=batch_size, learning_rate=learning_rate, epoch=80)

            train_c, val_c = np.array(history["train_cross_entropy_history"]), np.array(histor
            train_m, val_m = np.array(history["train_mce_history"]), np.array(history["val_mce
```

```
            train_cross += train_c/k
            val_cross += val_c/k
            train_mce += train_m/k
            val_mce += val_m/k

        print("#"*20, f"Model {model+1}", "#"*20)
        print("Mean Classification Accuracy (Train):", train_mce[-1])
        print("Mean Classification Accuracy (Validation):", val_mce[-1])
        print("Cross-Entropy Loss (Train):", train_cross[-1])
        print("Cross-Entropy Loss (Validation):", val_cross[-1])

        t_cross.append(train_cross)
        v_cross.append(val_cross)
        t_mce.append(train_mce)
        v_mce.append(val_mce)

        models.append(val_cross[-1])

models = np.array(models)
best_model = np.argmax(models)
print(f"Best model is the model {best_model}")

plt.figure()
plt.title("Cross Entropy Loss vs. Epoch (Train)")
plt.xlabel("Epoch")
plt.ylabel("Cross Entropy Loss")
for i in range(len(models)):
    plt.plot(t_cross[i])
plt.legend([f"Model {i+1}" for i in range(len(params))])
plt.show()

plt.figure()
plt.title("Cross Entropy Loss vs. Epoch (Validation)")
plt.xlabel("Epoch")
plt.ylabel("Cross Entropy Loss")
for i in range(len(models)):
    plt.plot(v_cross[i])
plt.legend([f"Model {i+1}" for i in range(len(params))])
plt.show()

plt.figure()
plt.title("Mean Classification Accuracy vs. Epoch (Train)")
plt.xlabel("Epoch")
plt.ylabel("Mean Classification Accuracy")
for i in range(len(models)):
    plt.plot(t_mce[i])
plt.legend([f"Model {i+1}" for i in range(len(params))])
plt.show()
```

```
    plt.figure()
    plt.title("Mean Classification Accuracy vs. Epoch (Validation)")
    plt.xlabel("Epoch")
    plt.ylabel("Mean Classification Accuracy")
    for i in range(len(models)):
        plt.plot(v_mce[i])
    plt.legend([f"Model {i+1}" for i in range(len(params))])
    plt.show()

    return best_model

def display_performance(y_pred, y):

    classes = np.sort(np.unique(y))
    scores = []
    for i in range(len(classes)):
        TP, TN, FP, FN = 0, 0, 0, 0
        recall, precision, f1 = 0, 0, 0
        cl = classes[i] # choose the class
        for j in range(len(y)):
            # Make other classes 0 except the positive class

            # Check whether the sample is positive value
            if y[j] == cl:
                # Sample is positive
                if y_pred[j] == y[j]:
                    TP += 1
                else:
                    FN += 1
            else:
                # Sample is negative
                if y_pred[j] == y[j]:
                    TN += 1
                else:
                    FP += 1

        recall, precision = TP / (TP + FN), TP / (TP + FP)
        f1 = (2 * recall * precision) / (recall + precision)

        scores.append(np.array([recall, precision, f1]))

    return scores
```

```
        #y_pred = model.predict(x_val)
        #acc = sum(y_pred == y_val) / y_pred.shape[0]
        #print("Logistic Regression")
        #print("-" * 20)
        #print(f"Accuracy for fold {fold + 1} is {acc}")
        #print()
        #accuracy.append(acc)


##############################################################################################


# Data gathering
en_data_folder = 'english-premier-league_zip'
es_data_folder = 'spanish-la-liga_zip'
fr_data_folder = 'french-ligue-1_zip'
ge_data_folder = 'german-bundesliga_zip'
it_data_folder = 'italian-serie-a_zip'

# data_folders = [es_data_folder]
data_folders = [en_data_folder, es_data_folder,
                fr_data_folder, ge_data_folder, it_data_folder]

season_range = (9, 18)

data_files = []
for data_folder in data_folders:
    for season in range(season_range[0], season_range[1] + 1):
        data_files.append(
            'data/{}/data/season-{:02d}{:02d}_csv.csv'.format(data_folder, season, season + 1))

data_frames = []

for data_file in data_files:
    if path.exists(data_file):
        data_frames.append(pd.read_csv(data_file))

data = pd.concat(data_frames).reset_index()
temp = data
#Drop the features having more than 100 NaN values.
input_filter = []
for feature in data.columns:
    if np.sum(data[feature].isna(), axis=0) < 100:
        input_filter.append(feature)

new_data = data[input_filter].dropna(axis=0)

#Drop Irrelevant Features
```

```
y, X = new_data["FTR"].values, new_data.drop(labels=["Div", "Date", "HomeTeam", "AwayTeam",
                                        "FTR", "HTR", "index"], axis=1)

y = label_encoder(y)
X.drop(columns=['FTHG', 'FTAG', 'HTAG'], axis=1, inplace=True)
#STANDARD SCALING
X_scaled = StandardScale(X)

#X_scaled, y = np.asarray(X_scaled).astype(np.int), np.asarray(y).astype(np.int)

x_train, y_train, x_test, y_test = train_test_split(X=X_scaled.values, y=y, split=0.2, shuffle=

# Parameters for hyperparameter tuning are as follows: neuron 1, neuron 2, batch_size, learning
params = [[32, 16, 500, 0.2],
          [64, 16, 500, 0.2],
          [128, 16, 500, 0.2],
          [128, 32, 250, 0.2],
          [128, 64, 250, 0.5],
          [128, 32, 250, 0.5],
          [128, 32, 250, 0.5],
          [128, 64, 250, 0.5],
          [256, 128, 500, 0.5],
          [256, 64, 500, 0.2]]
#train_neural_cv(params, x_train, y_train, 5)


print("#"*30, "FINAL TESTS WITH AND WITHOUT PCA", "#"*30)
##############################################   LOGISTIC REGRESSION    ########################

#PRINCIPLE COMPONENT ANALYSIS
X_transformed = PCA(num_components=18).fit(X_scaled).values.T

x_train, y_train, x_test, y_test = train_test_split(X=X_transformed, y=y, split=0.2, shuffle=T

params = [0.0025, 0.005, 0.0075, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07]
#train_logreg_cv(params, x_train, y_train, k=5)


# learning rate of 0.01 is the best one
print("Logistic Regression is being trained...")
# Test the Logistic Regression on Unseen Data
model = MultiClassLogisticRegression(ler_rate=0.1, max_iter=100)
log_train, log_test, acc_train, acc_test = model.fit(x_train, y_train, x_test, y_test)
y_pred1, labels = model.predict(x_test)

A, D, H = display_performance(labels, y_test)
A, D, H = np.around(A, 4), np.around(D, 4), np.around(H, 4)
```

```
print("*"*20, "Logistic Regression", "*"*20)
print("Positive Class is chosen as A = Away Team Wins")
print("Precision:", A[1])
print("Recall:", A[0])
print("F1 Score", A[2])
print("-"*50)
print("Positive Class is chosen as D = Draw")
print("Precision:", D[1])
print("Recall:", D[0])
print("F1 Score", D[2])
print("-"*50)
print("Positive Class is chosen as H = Home Team Wins")
print("Precision:", H[1])
print("Recall:", H[0])
print("F1 Score", H[2])
print("-"*50)
print("Mean Classification Accuracy:", acc_test[-1])
print("Log Loss:", log_test[-1])

##############################################   FEED-FORWARD NEURAL NETWORK    ################

x_train, y_train, x_test, y_test = train_test_split(X=X_scaled.values, y=y, split=0.2, shuffle=

n1, n2, batch_size, learning_rate = 128, 64, 250, 0.5
net = NeuralNetwork(early_stopping=30)
net.add_layer(Layer(neuron_num=n1, input_dim=X_scaled.shape[1], activation="sigmoid"))
net.add_layer(Layer(neuron_num=n2, input_dim=n1, activation="sigmoid"))
net.add_layer(Layer(neuron_num=3, input_dim=n2, activation="softmax"))
history = net.fit(X=x_train, Y=y_train, X_test=x_test, Y_test=y_test,
                                                    batch_size=batch_size, learning_rate=learni
y_pred2, labels = net.predict(x_test)

A, D, H = display_performance(labels, y_test)
A, D, H = np.around(A, 4), np.around(D, 4), np.around(H, 4)

print("*"*20, "Feed-forward Neural Network", "*"*20)
print("Positive Class is chosen as A = Away Team Wins")
print("Precision:", A[1])
print("Recall:", A[0])
print("F1 Score", A[2])
print("-"*50)
print("Positive Class is chosen as D = Draw")
print("Precision:", D[1])
print("Recall:", D[0])
print("F1 Score", D[2])
print("-"*50)
print("Positive Class is chosen as H = Home Team Wins")
print("Precision:", H[1])
```

```
print("Recall:", H[0])
print("F1 Score", H[2])
print("-"*50)
print("Mean Classification Accuracy:", history["val_mce_history"][-1])
print("Cross-Entropy Loss:", history["val_cross_entropy_history"][-1])

y_pred_ens = (y_pred1 + y_pred2)/2
A, D, H = display_performance(np.argmax(y_pred_ens, axis=1), y_test)
A, D, H = np.around(A, 4), np.around(D, 4), np.around(H, 4)

print("*"*20, "Ensemble (Logistic Regression+Neural Network)", "*"*20)
print("Positive Class is chosen as A = Away Team Wins")
print("Precision:", A[1])
print("Recall:", A[0])
print("F1 Score", A[2])
print("-"*50)
print("Positive Class is chosen as D = Draw")
print("Precision:", D[1])
print("Recall:", D[0])
print("F1 Score", D[2])
print("-"*50)
print("Positive Class is chosen as H = Home Team Wins")
print("Precision:", H[1])
print("Recall:", H[0])
print("F1 Score", H[2])
print("-"*50)
print("Mean Classification Accuracy:", (history["val_mce_history"][-1]+acc_test[-1])/2)



print("#"*30, "FINAL TESTS WITH MANUAL FEATURE SELECTION", "#"*30)

input_filter = ["HTHG","HST","AST","HS", "AS","HC", "AC", "HTAG"]
objects = ["FTR", "HTR"]
data[input_filter] = StandardScale(data[input_filter])

#Drop Irrelevant Features
y, X = new_data["FTR"].values, new_data[input_filter]

y = label_encoder(y)
#STANDARD SCALING
X_scaled = StandardScale(X)

#X_scaled, y = np.asarray(X_scaled).astype(np.int), np.asarray(y).astype(np.int)

x_train, y_train, x_test, y_test = train_test_split(X=X_scaled.values, y=y, split=0.2, shuffle=

# learning rate of 0.01 is the best one
```

```
print("Logistic Regression is being trained...")
# Test the Logistic Regression on Unseen Data
model = MultiClassLogisticRegression(ler_rate=0.1, max_iter=100)
log_train, log_test, acc_train, acc_test = model.fit(x_train, y_train, x_test, y_test)
y_pred1, labels = model.predict(x_test)

A, D, H = display_performance(labels, y_test)
A, D, H = np.around(A, 4), np.around(D, 4), np.around(H, 4)
print("#"*20, "FINAL TEST RESULTS", "#"*20)
print("*"*20, "Logistic Regression", "*"*20)
print("Positive Class is chosen as A = Away Team Wins")
print("Precision:", A[1])
print("Recall:", A[0])
print("F1 Score", A[2])
print("-"*50)
print("Positive Class is chosen as D = Draw")
print("Precision:", D[1])
print("Recall:", D[0])
print("F1 Score", D[2])
print("-"*50)
print("Positive Class is chosen as H = Home Team Wins")
print("Precision:", H[1])
print("Recall:", H[0])
print("F1 Score", H[2])
print("-"*50)
print("Mean Classification Accuracy:", acc_test[-1])
print("Log Loss:", log_test[-1])

##############################################   FEED-FORWARD NEURAL NETWORK    ################

x_train, y_train, x_test, y_test = train_test_split(X=X_scaled.values, y=y, split=0.2, shuffle=

n1, n2, batch_size, learning_rate = 128, 64, 250, 0.5
net = NeuralNetwork(early_stopping=30)
net.add_layer(Layer(neuron_num=n1, input_dim=X_scaled.shape[1], activation="sigmoid"))
net.add_layer(Layer(neuron_num=n2, input_dim=n1, activation="sigmoid"))
net.add_layer(Layer(neuron_num=3, input_dim=n2, activation="softmax"))
history = net.fit(X=x_train, Y=y_train, X_test=x_test, Y_test=y_test,
                                                    batch_size=batch_size, learning_rate=learni
y_pred2, labels = net.predict(x_test)

A, D, H = display_performance(labels, y_test)
A, D, H = np.around(A, 4), np.around(D, 4), np.around(H, 4)

print("*"*20, "Feed-forward Neural Network", "*"*20)
print("Positive Class is chosen as A = Away Team Wins")
print("Precision:", A[1])
print("Recall:", A[0])
```

```
print("F1 Score", A[2])
print("-"*50)
print("Positive Class is chosen as D = Draw")
print("Precision:", D[1])
print("Recall:", D[0])
print("F1 Score", D[2])
print("-"*50)
print("Positive Class is chosen as H = Home Team Wins")
print("Precision:", H[1])
print("Recall:", H[0])
print("F1 Score", H[2])
print("-"*50)
print("Mean Classification Accuracy:", history["val_mce_history"][-1])
print("Cross-Entropy Loss:", history["val_cross_entropy_history"][-1])

y_pred_ens = (y_pred1 + y_pred2)/2
A, D, H = display_performance(np.argmax(y_pred_ens, axis=1), y_test)
A, D, H = np.around(A, 4), np.around(D, 4), np.around(H, 4)

print("*"*20, "Ensemble (Logistic Regression+Neural Network)", "*"*20)
print("Positive Class is chosen as A = Away Team Wins")
print("Precision:", A[1])
print("Recall:", A[0])
print("F1 Score", A[2])
print("-"*50)
print("Positive Class is chosen as D = Draw")
print("Precision:", D[1])
print("Recall:", D[0])
print("F1 Score", D[2])
print("-"*50)
print("Positive Class is chosen as H = Home Team Wins")
print("Precision:", H[1])
print("Recall:", H[0])
print("F1 Score", H[2])
print("-"*50)
print("Mean Classification Accuracy:", (history["val_mce_history"][-1]+acc_test[-1])/2)



'''
##########################################   NAIVE BAYES CLASSIFIER    ####################

input_filter = ["HTHG","HST","AST","HS", "AS","HC", "AC", "HTAG"]
objects = ["FTR", "HTR"]
data[input_filter] = StandardScale(data[input_filter])
cols_to_consider = input_filter + objects
data = data[cols_to_consider]
```

```
X = data[input_filter]
Y = data['HTR']
Z = data["FTR"]

x_train, y_train, x_test, y_test = train_test_split(X=X.values, y=Y, split=0.2, shuffle=True)
x_train, x_test = pd.DataFrame(x_train, columns=input_filter), pd.DataFrame(x_test, columns=inp
x_train["HTR"] = y_train
x_test["HTR"] = y_test



#Drop Irrelevant Features


#y, X = new_data["FTR"].values, new_data.drop(labels=["Div", "Date", "HomeTeam", "AwayTeam", "
#X = X[input_filter]

#x_train, y_train, x_test, y_test = pd.DataFrame(x_train, columns=input_filter), y_train, \
 #                                  pd.DataFrame(x_test, columns=input_filter), y_test



dfA, dfH, dfD = dataClassifier(x_test)

mValHome, mValDraw, mValAway, vValHome, vValDraw, vValAway = meanAndVar(dfH[input_filter], dfA
                                                                        dfD[input_filter])
priHome, priAway, priDraw = get_prior_probs(dfH, dfA, dfD, x_test)

mValHome_test, mValDraw_test, mValAway_test, vValHome_test, vValDraw_test, vValAway_test = mean
    dfH[input_filter], dfA[input_filter], dfD[input_filter])

priHome_test, priAway_test, priDraw_test = get_prior_probs(dfH, dfA, dfD, x_test)

probFrameForHome_test, probFrameForAway_test, probFrameForDraw_test = prior_prob_frame(mValHome
                                                                                       mValAway
                                                                                       vValDraw
                                                                                       x_test)


y_pred3 = final_result_probabilities(probFrameForHome_test, probFrameForAway_test, probFrameFor
results, loc = accuracy_calculator(y_pred3, y_train)
print(np.unique(loc))
print(loc)
A, D, H = display_performance(loc, y_test)
A, D, H = np.around(A, 4), np.around(D, 4), np.around(H, 4)

print("*"*20, "Feed-forward Neural Network", "*"*20)
print("Positive Class is chosen as A = Away Team Wins")
print("Precision:", A[1])
```

```
print("Recall:", A[0])
print("F1 Score", A[2])
print("-"*50)
print("Positive Class is chosen as D = Draw")
print("Precision:", D[1])
print("Recall:", D[0])
print("F1 Score", D[2])
print("-"*50)
print("Positive Class is chosen as H = Home Team Wins")
print("Precision:", H[1])
print("Recall:", H[0])
print("F1 Score", H[2])
print("-"*50)
#print("Mean Classification Accuracy:", history["val_mce_history"][-1])
#print("Cross-Entropy Loss:", history["val_cross_entropy_history"][-1])


y_pred_ens = (y_pred1 + y_pred2)/2
A, D, H = display_performance(y_pred_ens, y_test)
A, D, H = np.around(A, 4), np.around(D, 4), np.around(H, 4)

print("*"*20, "Ensemble (Logistic Regression+Neural Network)", "*"*20)
print("Positive Class is chosen as A = Away Team Wins")
print("Precision:", A[1])
print("Recall:", A[0])
print("F1 Score", A[2])
print("-"*50)
print("Positive Class is chosen as D = Draw")
print("Precision:", D[1])
print("Recall:", D[0])
print("F1 Score", D[2])
print("-"*50)
print("Positive Class is chosen as H = Home Team Wins")
print("Precision:", H[1])
print("Recall:", H[0])
print("F1 Score", H[2])
print("-"*50)
print("Mean Classification Accuracy:", (history["val_mce_history"][-1]+acc_test[-1])/2)
'''
```