NBA GAME OUTCOME PREDICTION

EEE-485

PROJECT FINAL REPORT

Group - 3

Tugay Balatlı (21702822)

Ata Berk Çakır (21703127)

1. Introduction and Problem Definition

Basketball is one of the most popular sports around the world, with the use of statistics, following it is getting more and more exciting. Due to the statistical and algebraic roots of the machine learning, combining a game which creates statistically considerable data continuously with such a science can give entertaining and accurate results.

In this project, we focused on predicting NBA regular season games' outcomes. Due to the binary outcome of the NBA games, we worked on the classification problem instances and possible solutions. Then, we decided to work on Logistic Regression, Decision Tree Classifier, Support Vector Machine Classifier. Our solutions are focusing on binary classification; hence, the results will be interpreted as win (1) or lose (0).

2. Dataset Description and Preprocessing

To support our models with useful data, we did some research on the topic. Initially, we tried to comprehend which statistical NBA game data is useful for us. To obtain the expert opinion and basic knowledge about the problem, we examined several articles. As Vaidya (2019) stated, "Offensive rating, defensive rating, field goal percentage, field goal attempts, 3-point percentage, 3-point attempts, assist/turnover ratio, rebound differential and pace" are important variables in our problem [1].

Our first intention was scraping our data from the basketball-reference website [2] to get the played matches and their results individually and NBA website's stats section [3] to get the statistics of the teams for the given time interval. We collected the data manually from the given websites. Then, the collected data instances from 2016-2017, 2017-2018, 2018-2019 and 2020-2021 regular seasons. The reason why we excluded 2019-2020 regular season is because of the pandemic. With the light of the common knowledge and the article written by Botkin and Blomain, we observed that there are a few data instances in this regular season [4]. Additionally, since the league is suspended for a long time, the real statistics were not totally reflected in the dataset regarding 2019-2020. After the collection process, we merged our data with respect to the index of the games as all the games are scheduled in an order.

In our data, we have 3370 instances and 32 columns. Half of the columns include statistics and name of the home team. Other half contains same statistics for the away team for the same game. The statistics are numeric values about the points made in the given game ("PTS") , offence, defense and net ratings ("Team1OFFRTG", "Team1DEFRTG", "Team1NETRTG"), assist, rebound (offensive and defensive), assist to turnover ratio ("Team1AST%", "Team1AST/TO", "Team1OREB%", "Team1DREB%", "Team1AST RATIO"), turnover ratio ("Team1TOV%"), true shooting percentage ("Team1TS%"). In addition to these columns, we have two additional columns that are calculated by the NBA statisticians with these methods: Pace: 48 * ((Team Possessions + Opponent Possessions) / (2 * (Team Minutes Played / 5))) [5], PIE: % of game events did that player or team achieve [6] (Team1PACE, Team1PIE). These columns are included in the opponent's statistics.

After completed the data inspection and collection part, we did some preprocessing on our dataset. First of all, we created a to assign response variable which is called "Team1Win". It has binary values in it and refers to the home team's win situation.

In the preprocessing part, we created a function to apply standardization by min-max scaler using the formula given below:

$$x^{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Hence, we obtained scaled data to decrease the effect of the outliers in numeric columns. Then, we dropped points made in the game, team names from the columns to avoid memorization and hence overfitting. We used the scaled data in two of our methods which are logistic regression and support vector machine classifier. The decision tree classifier searches a split point on each feature to select the best split. That is essentially an if clause that groups target values regarding the occurrences where the test value is less than the split value on the left, and values more than equal on the right. By normalizing

the dataset, we obtain the same order in the feature, hence, we should get the same tree based on our minimum sample and maximum depth preferences. Therefore, using normalization in decision tree classifier would be waste of computation time without any advantage.

In the last step of the data preprocessing, predictor and response variables are separated into two sets named as X for predictors and y for response. After creating these two data frames, we split our dataset into train and test set by coding a function specifically for this purpose. This function takes split size as an input. As a result of our literature review, we found out that using 0.8 would be efficient for the model training and testing parts. After the feedback we received in the first progress, we generated a validation split by using %20 of our training data. We used this set to tune our hyperparameters. The hyperparameters tuned in logistic regression are learning rate and number of iterations. In SVM, again learning rate and number of iterations are tuned, additionally lambda parameter is tuned by using validation set. Lastly, in decision tree classifier are minimum number of samples in a split and maximum depth of the tree.

The multicollinearity is considerable problem for the performance of the models. Therefore, we controlled the correlations between the features (Can be seen in the figure below). The correlation is calculated by using this formula:

$$r_{X,Y} = \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})^2\sum_{i=1}^{n}(y_i-\bar{y})^2}}$$

It shows the correlation between two variables. The multicollinearity means the high intercorrelation between two or more independent variables. Due to the problematic and less reliable results of multicollinearity, we tried to avoid using the variables together which have high correlation in our models.

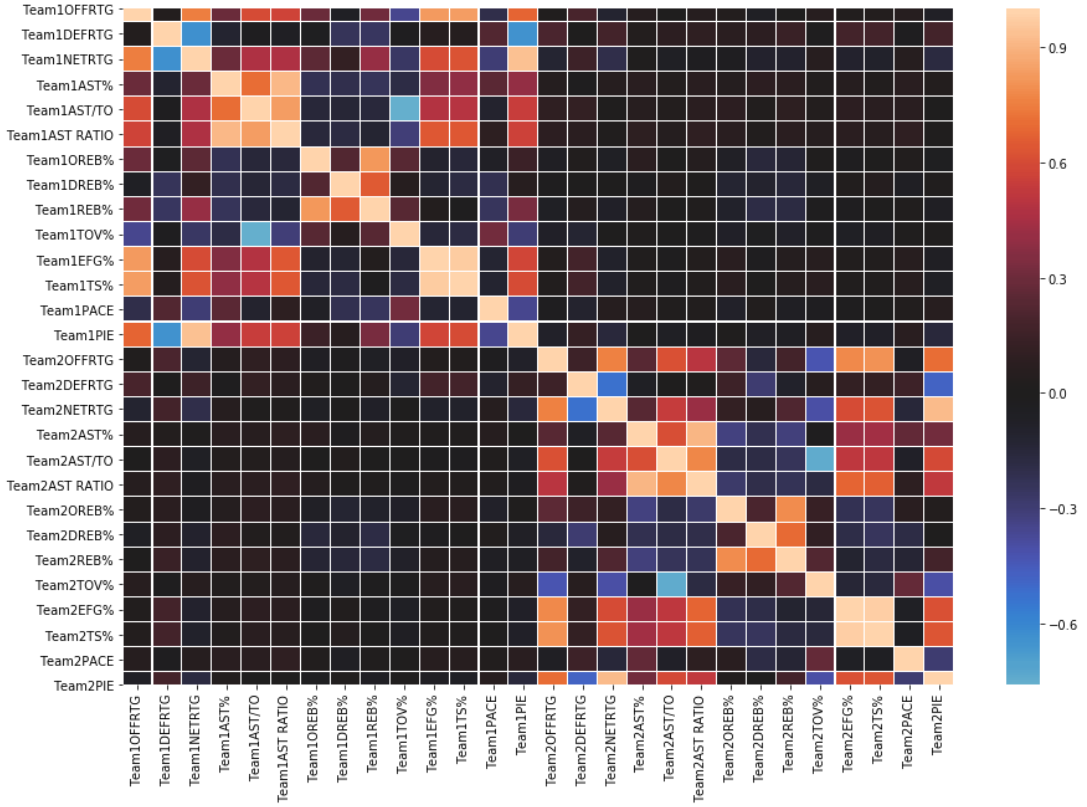

Figure 1 – Correlation Matrix as a Heat Map

After inspection, we observed that there is a high correlation between columns “Team1AST RATIO” and “Team1AST%”. The same observation is also acceptable for the Team 2. Hence, we decided to drop “Team1AST RATIO”. Similar observations are done for the “Team1NETRTG” and “Team1PIE”, “Team2EFG%” and “Team2TS%” duos. Thus, we also decided to drop “Team1PIE” and “Team2TS%” for both teams. We used Seaborn and Matplotlib libraries to visualize our correlation results.

3. Methodology

The algorithms that are selected for the purpose of the project are Logistic Regression, Support Vector Machine Classifier and Decision Tree Classifier. We aimed to get binary results as it is stated above, therefore, using these algorithms can give desired outcome. The methods are selected by considering several purposes which are computation time, efficiency, interpretability, compatibility with our dataset and reliability in the predictions and results.

Below, in each subsection you will find the reasons why these algorithms are selected and what are the advantages and disadvantages of these algorithms. To check the simulation results, computation time results and performance analysis, you can see the simulation results section.

3.1. Logistic Regression

Firstly, we assumed that our dataset is linearly separable, by considering this and the theoretical background of the method stated below, using logistic regression is useful. When the predicted outcome of the model is categorical, logistic regression can be used. Since we are trying to predict the game results which 1 or 0 the logistic regression model fits best to our problem therefore, we decided to use this method.

As it is faster than many other methods in classifying new instances and since it is easier to interpret, we focused on logistic regression. Mainly, it does not make any assumptions about distributions of the classes, instead it fits a function. Hence, we are able to get high efficiency in training and prediction parts. One of the biggest advantages of this method is its probability outputs. We are able to check if the prediction results are confident and sufficiently reliable to assign a class. The flexibility of choosing a threshold for assigning the obtained probabilities to a binary class is another reason why we selected this method. We compared the probability of the instance belongs to class 0 and probability of the instance belongs to class 1 instead of using a threshold directly.

In the disadvantages of this algorithm, it requires independent variables which are cleansed from multicollinearity. Therefore, it needs preprocessing beforehand. However, we forecasted such issues before we begin to implement the algorithms. Thus, we got rid of this issue. Additionally, the algorithm is much more sensitive to the outliers if we compare it with the other methods that we are using. Hence, we used min-max normalization to decrease the effect of the outliers.

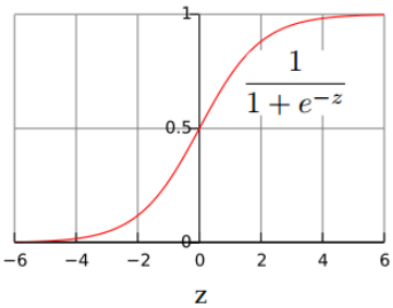

Figure 2 – Logistic Function (Sigmoid Function)

We are trying to learn directly $P(Y|X)$ with assuming $P(Y|X)$ takes functional form. In other words, sigmoid function applied to a linear function of the data. The sigmoid funtion is shown in the following figure.

The calculated probability is the hypothesis's output. When given an input X, this is utilized to determine how certain a predicted value can be the actual value. The probabilities are calculated by the following formulas [8].

$$P(Y=1|X) = \frac{1}{1+e^{(w_0 \sum_{i=1}^{n} w_i X_i)}} \quad (1)$$

$$P(Y=0|X) = \frac{e^{(w_0 \sum_{i=1}^{n} w_i X_i)}}{1+e^{(w_0 \sum_{i=1}^{n} w_i X_i)}} \quad (2)$$

The predicted value for Y will be equal to the output of the class with highest $P(Y|X)$.

$$1 < \frac{P(Y=0|X)}{P(Y=1|X)} \tag{3}$$

$$1 < e^{(w_0 \sum_{i=1}^{n} w_i X_i)} \tag{4}$$

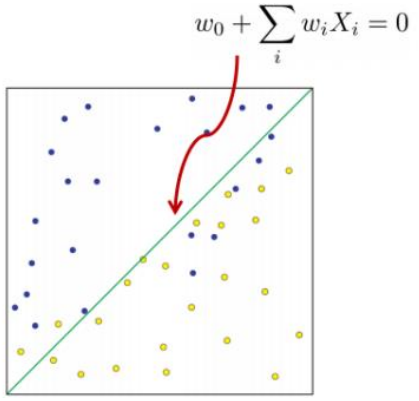

Figure 3 – Linear Decision Boundary

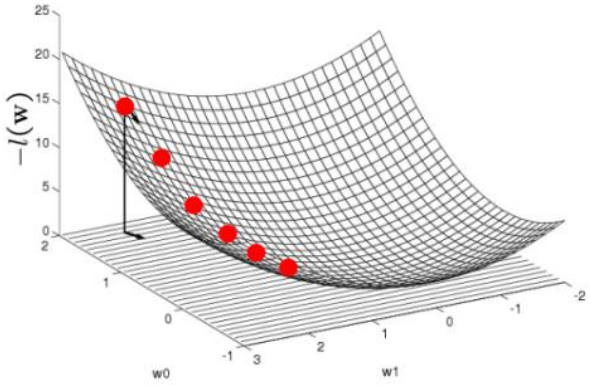

Figure 4 – Gradient Ascent

In order to learn parameters $w_0, w_1, \ldots\ldots, w_n$ we are going to use maximum conditional likelihood estimator which is the following where training data is $\{X^{(j)}, Y^{(j)}\}_{j=1}^{n}$

$$\widehat{w}_{MCLE} = arg\max_{w} \prod_{j=1}^{n} P\left(Y^{(j)} \middle| X^{(j)}, w\right) \tag{5}$$

Since conditional likelihood for Logistic Regression is concave, we are going to use Gradient Ascent to optimize parameters.

Gradient:

$$\nabla_w l(w) = \left[\frac{\partial l(w)}{\partial \mathrm{w}_0}, \ldots\ldots, \frac{\partial l(w)}{\partial \mathrm{w}_n}\right] \tag{6}$$

Update rule (where n is learning rate):

$$\Delta w = n \nabla_w l(w) \tag{7}$$

As we expected, the logistic regression worked well on our dataset as the dataset is linearly separable. We used validation split to update the iteration number and learning rate in gradient descent to tune our parameters.

Performance of the logistic regression with respect to the validation and the test prediction results are in simulation results part. Besides, you can check the obtained sigmoid function with the updated beta values in simulation results.

3.2. Support Vector Machine Classifier

We are going to use support vector machine classifier for linearly separable classes. Despite mentioning linearly separable classes, SVM classifier has also been used efficiently in non-linear classification too. It can be implemented with kernel trick which maps the inputs to the higher dimensional spaces and does classification. We considered this flexibility while we are choosing SVM as a solution method. However, we did not need to use it as we are satisfied with the simulation results (See the simulation results section).

We have around 20 features after the preprocessing. Hence, due to the high number of features, using SVM can help us obtain good results according to the literature review we have done. In addition to these, since we are optimizing the margin in the SVM, having a dataset which is suitable to separate classes by a margin distance was helped considerably while choosing this method. On the other hand, there are also downsides of SVM. As it is non-probabilistic method, it does not provide probability estimates of each instances. Hence, we are not able to interpret the results by using probabilistic approaches instead we use linear algebra.

In the theoretical part, SVM algorithm creates a model that assigns new examples to one of two categories, making it a non-probabilistic binary linear classifier, given a series of training examples, each marked as belonging to one of two categories [7]. The key idea behind SVM is hyperplane that maximizes the margin will have better generalization. For a linearly separable data SVM can better be understand from the following figure.

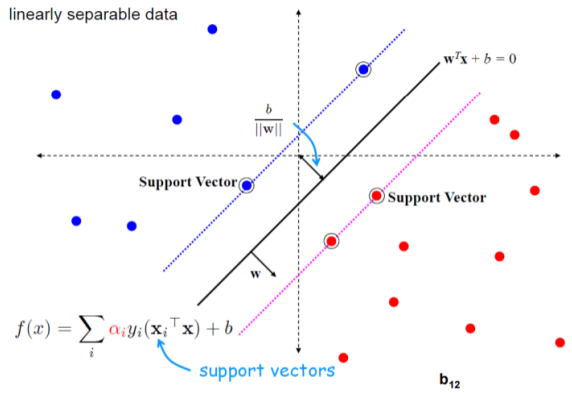

Figure 5- Support Vector Machine

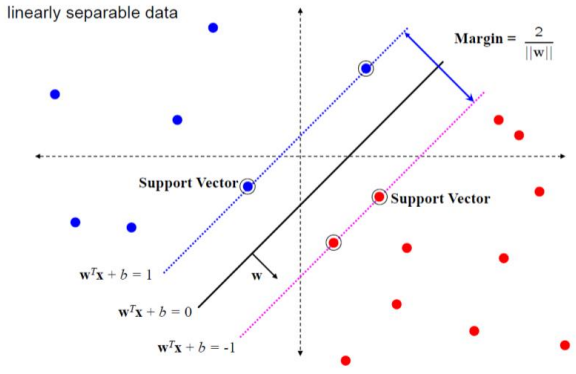

Figure 6 – Geometric representation of SVM

Since $w^T x + b = 0$ and $c(w^T x + b) = 0$ define the same plane we can decide the normnalization of w such that $w^T x_+ + b = +1$ and $w^T x_- + b = -1$ for the positive and negative support vectors respectively. The margin can be found by

$$\frac{w}{\|w\|} * (x_+ - x_-) = \frac{w^T(x_+ - x_-)}{\|w\|} = \frac{2}{\|w\|}$$

The geometric representation of SVM is the following

Then learning SVM can be formulated as optimization problem where it is a quadratic optimization problem subject to linear constraints and there exist a unique minimum [8].

$$\min_w \|w\|^2 \; subject\ to\ y_i(w^T x_i + b\ ) \geq 1\ for\ i = 1 \ldots \ldots N$$

Before obtaining the results of this algorithm, we used regularization parameter lambda in our loss function. The hinge loss is used to maximize the margin and to update the weights, we used gradient descent.

In the validation part of the SVM, we changed learning rate, number of iterations and regularization parameter lambda.

3.3. Decision Tree Classifier

In this method, we aimed to use decision tree classifier. It is based on a tree structure and decisions made on the internal nodes and branches.

Since explaining the methods to other business units is significant in the industry, we decided to use decision tree because of its simple interpretability and understandability. Our first intuition was to use it to have easily explainable algorithm within our methods.

Additionally, we are able to implement the algorithm without any normalization and outliers have less significance in the results. The reasons of this are explained in the data preparation part.

The disadvantages of this algorithm are also considered in the literature review before choosing the algorithm. The resulting tree can change easily. Even though the bagging and boosting can reduce this unstable nature of decision trees, the implementations of the bagging and boosting expansions are harder as we observed during our research. Hence, we decided to continue with the decision tree classifier merely.

As a theoretical result, the leaves show the respective class for the selected specific part of the tree. The separation in the branches to classes is works with the if statements in general. The classification and node structures are based on several concepts in decision tree classifier. Those are entropy, information gain and Gini index.

As a stopping criterion for splitting, the algorithm uses the purity of the child node. It is done by checking entropy and information gain regarding the entropy.

Information is reduction in the uncertainty in outcome. Its formula is:

$$I(X) = -log_2 p(x)$$

Entropy is expected amount of the information while observing the output of random variable X. Formula for the entropy is:

$$H(X) = E(I(X)) = \Sigma_i p(x_i) I(x_i) = -\Sigma p(x_i) log_2 p(x_i)$$

Information gain is reduction in uncertainty by knowing Y. The formula for information gain is:

$$IG(X,Y) = H(Y) - H(Y|X)$$

Information gain also helps us to indicate which attributes in the data set's feature column are helpful for the discrimination of the given classes. Information gain value shows us the importance as a metric.

As it is stated above, the purity is the decision mechanism for the stopping conditions. Gini index is calculated to understand the purity. Where $p(i|t)$ is the relative frequency of class i at node t. Here is the formula of Gini index:

$$GINI(t) = 1 - \Sigma_i p(i|t)^2$$

Lower Gini index values are considered for the further investigation which is done by adding new branches. [8]

The changeable parameters of decision tree classifier are minimum number of samples in a split and maximum depth of the tree. We focused on choosing these in the validation part. You can check the simulation results.

4. Simulation Setup

Our initial focus was creating a neat and functional dataset. To do it, we used two different data sources and collected our data manually. Before merging the datasets, we created a Python code to assign binary values based on comparison of points made by each team, and home team's win or lose situation (See the Appendix for the code implementations). It has simple logic behind it similar to binary encoders, such as One Hot Encoder. As we do not have any other categorical value, we did not face with a challenge about encoding.

After determining the problem statement and collecting our data, we started with a literature review to grasp which methods can be useful in our dataset. While we were examining the classification methods, we also started to preprocess and cleanse the dataset.

We expected to see some challenges, hence we started to solve the problematic parts before. First of all, outlier values were our first concern, however, after applying normalization by using Min-Max Scaler algorithm, we fixed the possible problems in general. As we are focusing on the binary results, we will not need to rescale the results. The new data that can be added should be revised before prediction or train part because of the scaling process. Additionally, the class imbalance was not a big problem for us as there is always one winner and one loser.

As we planned, we created train and test sets by homogeneously splitting our data set. Data cleaning and preprocessing is completed. We focused on avoiding multicollinearity, which indicates that there is strong inter-correlation between two or more independent variable, to increase our models' performance. Besides, after the feedback we received, we separated train set to validation and train sets. We trained the selected models by using train set and then, we used validation set to tune our parameters. Our split sets are used in different formats like data frame or Numpy array to comply with our 'from scratch' implemented methods.

We started to implement the algorithms after the data preprocessing part. We have currently implemented all methods that we have proposed in the beginning of the semester.

We tested our models with different metrics. These are recall, precision and accuracy values, but the accuracy is not known as strong metric to indicate model's performance. Hence, we tried to widen our performance by checking other metrics. We controlled F1 and F2 scores which are the combinations of recall and precision. After that, we validated our models with the validation set and tuned the hyperparameters that are mentioned above in each method's subsection. We used confusion matrix to assess these metrics in numerical values and did necessary calculations by using True Positive, True Negative, False Positive and False Negative values.

5. Simulation Results

Simulation results are obtained after the simulation setup is configured. We started with visualizing and analyzing our data as we started with correlation analysis and scaling before. In the data distributions, there was no significant skewness, correlation according to the pair plot we drew. Hence, we did not apply any transformation on our raw data columns except min-max scaling.

5.1. Training the Models

After preparing our implementation, validation and testing codes, we used the split data which we have covered in simulation setup part. Firstly, we trained our models by using train set. Some of our challenges regarding the implementation were the computation times and computer memory usage. We measured our methods' time performances by adding timer to our codes. The training part and testing parts are lasted as following (Validation is changing with respect to searched parameters):

	Elapsed Time
Logistic Regression	27.59 seconds
Decision Tree	19.74 seconds
SVM	22.97 seconds

Figure 8 – Computation Time

5.2. Validation of the Models

Then, we validated our models' performance and hyperparameters by using validation set. You can see the validation results and selected hyperparameters for each method in below. Our parameter tuning method is like grid search method. We iterate over pre-determined values and their combinations.

5.2.1. Logistic Regression

We changed two different hyperparameters in logistic regression which are iteration number and learning rate for the gradient descent. We obtained best accuracy result which is 95.12% by using 2000 iterations and 1e-06 as learning rate in validation set.

Logistic Regression			
	Accuracy (%100)		Accuracy (%100)
Itertation = 200 Learning Rate = 1e-07	55.04	Itertation = 1000 Learning Rate = 1e-05	66.02
Itertation = 200 Learning Rate = 1e-06	67.95	Itertation = 1000 Learning Rate = 1e-07	68.54
Itertation = 200 Learning Rate = 1e-05	95.10	Itertation = 2000 Learning Rate = 1e-06	95.12
Itertation = 1000 Learning Rate = 1e-07	55.63	Itertation = 2000 Learning Rate = 1e-05	61.87
Itertation = 1000 Learning Rate = 1e-06	69.48		

Figure 9 – Logistic Regression Validation

5.2.2. SVM Classifier

In SVM Classifier, we changed three parameters, again number of iterations are two of them. Differently, since we added the regularization parameter in SVM, we changed the lambda value also. We obtained 71.22% accuracy by using 1000 iterations with 1e-4 learning rate and 1e-2 lambda penalty value.

SVM		SVM	
	Accuracy (%100)		Accuracy (%100)
Iteration = 200 Learning Rates = 1e-3 Lamda = 1e-2	71.21	Iteration = 1000 Learning Rates = 1e-4 Lamda = 1e-2	71.22
Iteration = 200 Learning Rates = 1e-3 Lamda = 1e-3	69.28	Iteration = 1000 Learning Rates = 1e-4 Lamda = 1e-3	67.50
Iteration = 200 Learning Rates = 1e-4 Lamda = 1e-2	70.62	Iteration = 1000 Learning Rates = 1e-5 Lamda = 1e-2	71.06
Iteration = 200 Learning Rates = 1e-4 Lamda = 1e-3	71.21	Iteration = 1000 Learning Rates = 1e-5 Lamda = 1e-3	70.62
Iteration = 200 Learning Rates = 1e-5 Lamda = 1e-2	55.04	Iteration = 2000 Learning Rates = 1e-3 Lamda = 1e-2	71.22
Iteration = 200 Learning Rates = 1e-5 Lamda = 1e-3	55.04	Iteration = 2000 Learning Rates = 1e-3 Lamda = 1e-3	69.28
Iteration = 1000 Learning Rates = 1e-3 Lamda = 1e-2	71.21	Iteration = 2000 Learning Rates = 1e-4 Lamda = 1e-2	71.22
Iteration = 1000 Learning Rates = 1e-3 Lamda = 1e-3	69.28	Iteration = 2000 Learning Rates = 1e-4 Lamda = 1e-3	69.28

Figure 10 – SVM Validation

Decision Tree		Decision Tree	
	Accuracy (%100)		Accuracy (%100)
Max Depth = 2 Min Sample Split = 10	74.07	Max Depth = 4 Min Sample Split = 10	82.17
Max Depth = 2 Min Sample Split = 100	74.08	Max Depth = 4 Min Sample Split = 100	82.40
Max Depth = 2 Min Sample Split = 1000	70.37	Max Depth = 4 Min Sample Split = 1000	70.37
Max Depth = 2 Min Sample Split = 10000	54.39	Max Depth = 4 Min Sample Split = 10000	54.39
Max Depth = 3 Min Sample Split = 10	78.93	Max Depth = 5 Min Sample Split = 10	90.74
Max Depth = 3 Min Sample Split = 100	78.94	Max Depth = 5 Min Sample Split = 100	87.03
Max Depth = 3 Min Sample Split = 1000	70.37	Max Depth = 5 Min Sample Split = 1000	70.37
Max Depth = 3 Min Sample Split = 10000	54.40	Max Depth = 5 Min Sample Split = 10000	54.39

Figure 11 – Decision Tree Validation

5.2.3. Decision Tree Classifier

Since we do not have any optimization like gradient descent in decision tree classifier, we changed different parameters here. First one is minimum number of samples in a split and maximum depth (See Figure 12). The best values are 10 for min. sample split and 5 for max depth with accuracy 90.74%.

5.3. Test Results

After completing the validation and parameter tuning, we tested our models with allocated test set and obtained several metrics as we discussed before. We obtained several results and tables. You can see the results in figures and confusion matrices that we have obtained.

Logistic Regression (learning rate=1e-6/iteration=2000)

		Predicted	
		0	1
Actual	0	338	33
	1	0	303

SVM (iterations=1000/learning rate=0.0001/lambda=0.01)

		Predicted	
		0	1
Actual	0	288	83
	1	111	192

Decision Tree (max depth=5/minimum sample split=10)

		Predicted	
		0	1
Actual	0	338	33
	1	34	269

Figure 12 – Confusion Matrices for All Methods

The relevant statistics that can be gauged and collected by using confusion matrices are in below:

5.3.1. Logistic Regression

We obtained 95.1% accuracy with 94.8% F1 score in test results.

```
Performance metrics for full gradient ascent when learning rate is equal to 1e-6 and iteration is equal to 2000
Accuracy: %95.10385756676558
tp: 303.0
tn: 338.0
fp: 33.0
fn: 0.0
Precision:  0.9017857140173257
Recall:  0.9999999996699669
NPV:  0.999999999704142
FPR:  0.08894878703801919
FDR:  0.09821428568505526
F1 score:  0.9483567573482652
F2_score:  0.9786821489838105
Time elapsed:  27.591750000000502
```

Figure 13 – Test Results of Logistic Regression

The final logistic regression model's β values are obtained as follows:

Weights for Logistic Reg	
b0 = -0.0114917	b13 = 0.0524057
b1 = -0.0708439	b14 = -0.119327
b2 =0.116843	b15 = -0.0296091
b3 = 0.0445741	b16 = -0.0776155
b4 = 0.0568117	b17 = -0.0382752
b5 = 0.00690385	b18 = -0.03986
b6 = -0.0110064	b19 = -0.0554183
b7 = 0.0235698	b20 = 0.0524232
b8 = -0.0334375	b21 = -0.0700611
b9 = 0.0721776	b22 = -0.069171
b10 = 0.069998	b23 = 0.0234691
b11 = -0.0444395	b24 = 1,05727
b12 = -0.0863345	

Figure 15 – Logistic Regression β Values

5.3.2. SVM Classifier

SVM classifier gave worse result scores than the other two methods. We classified 480 instances correctly by using this method.

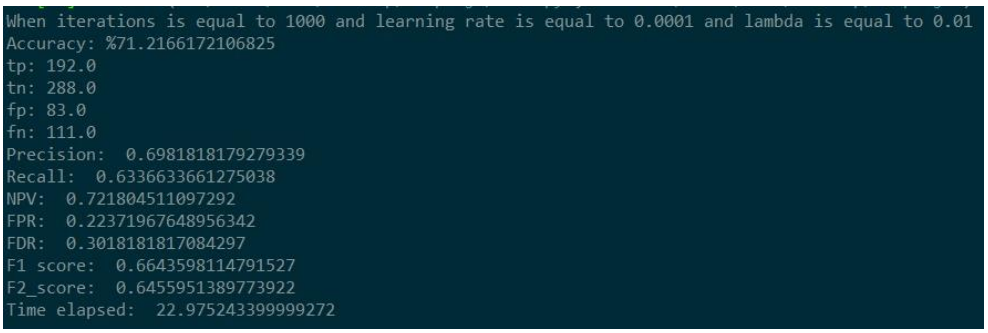

Figure 16 – Test Results of SVM

5.3.3. Decision Tree Classifier

Decision tree classifier's test results with updated parameters are given in the below figure. We have 88.9% F1 Score which is good and 90.05% accuracy.

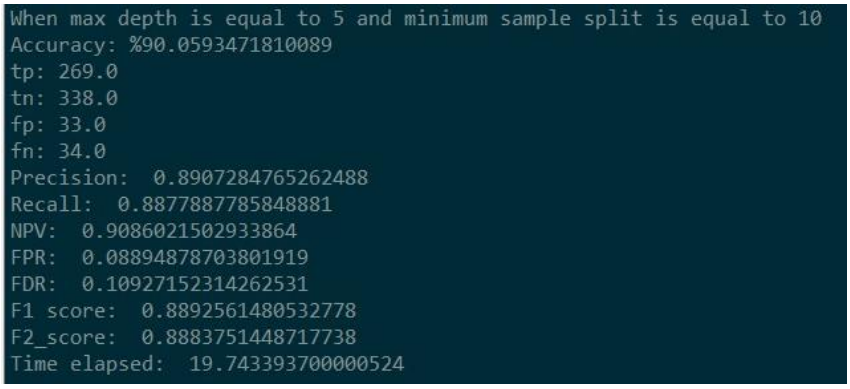

Figure 17 – Test Results of Decision Tree

6. Discussion

We encountered some challenges during implementing project and the solutions we obtained to those challenges are mentioned below this topic. Even though we had partly clean data, we spent a lot of time on the preprocessing as we wanted to get good results in every method that is used for classification in this dataset.

Besides, we struggled with the optimum value of hyperparameters while effectively finding and tuning them because these parameters can be changed a lot depending on whether the model is underfitting or overfitting. The most difficult aspect of the project was determining suitable hyperparameter values. After determining the hyperparameters using our validation sets we made predictions with the optimized parameters using the test sets. The performance results of the different models are already stated in the results part. Finally, in order to choose the best model we decided to plot Receiving Operations Characteristic (ROC) curves for each model. In the ROC curves right upmost corner stands for the best model where TPR equals to 1 and FPR equals to 0 and left down most corner stands for the worst model where TPR equals to 0 and FPR equals to 1. The ROC curves of all three models can be seen below. As shown in the figure 8 Logistic Regression Classifier is the closest one to the right upmost corner therefore we can state that it gives the best performance than followed by Decision Tree Classifier and SVM Classifier.

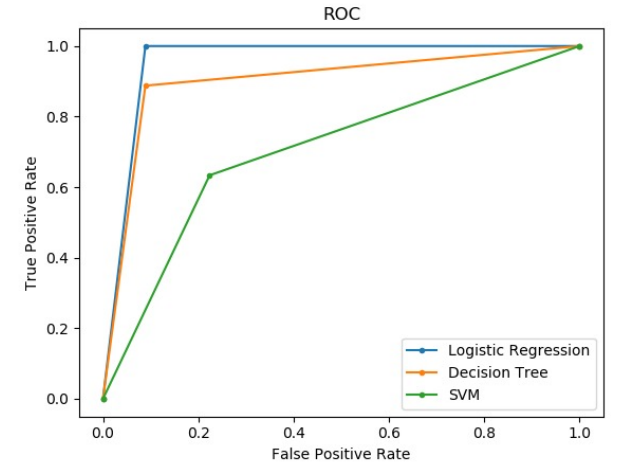

Figure 8 – ROC Curves of All Models

7. Timeline

You can see the list of the tasks, division of tasks among group members and allocated dates to each task below.

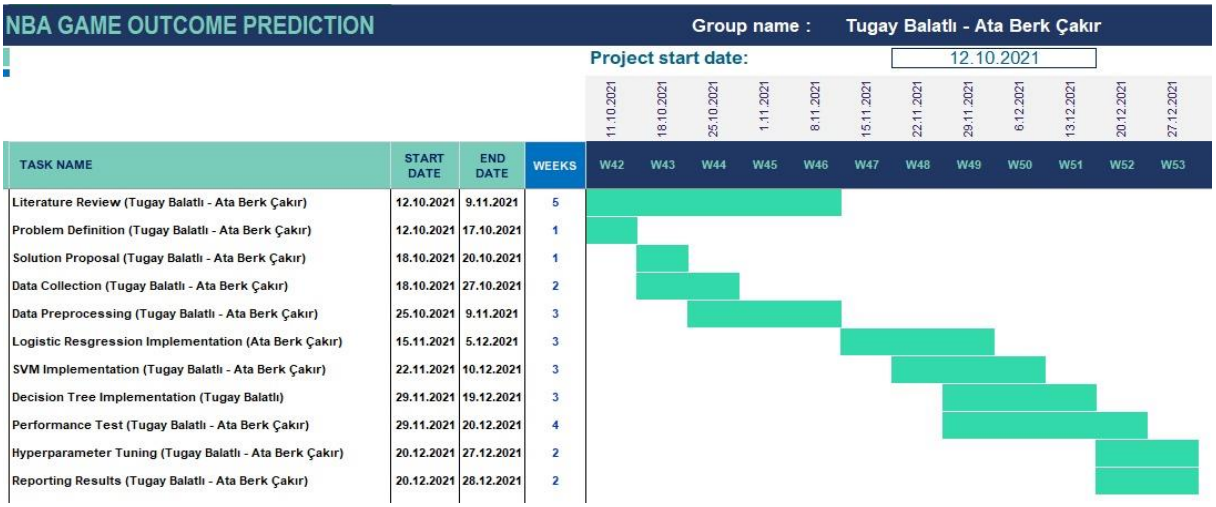

Figure 9 – Gantt Chart of The Project Plan

8. Conclusion

To conclude, in this project we are not only learn how implement three different classification algorithm from scratch without using any prebuild library but also experienced how to handle a complete machine learning project with all the aspects such as data scouting, data preprocessing and performance optimization. After implementing all three model we run performance algorithms on the validation sets in order to optimize the hyper parameters of the models. Finally, we run the models with optimized parameters on the test set to get predicted NBA game results. According to our test results we can state that all three algorithms are suitable for chosen dataset; however, the Logistic Regression Classifier gives the best performance which is kind of an expected result. It utilizes a sigmoid function and works best on the binary classification problem. That is why it is one of the most common and useful classification algorithms in machine learning. To sum up, during this term project we had a chance to enhance our theoretical knowledge that we gain in the lessons with a real-life machine learning coding skills. We believe that the experience that we gain while implementing this project will be beneficial for our professional working life in the future.

References

[1] C. Vaidya. (2019). Which NBA Statistics Actually Translate to Wins? [Online]. Available: https://watchstadium.com/which-nba-statistics-actually-translate-to-wins-07-13-2019/ [Accessed: 17-Nov-2021].

[2] Basketball-Reference. (2021). NBA Schedule and Results [Online]. Available: https://www.basketball-reference.com/leagues/NBA_2021_games.html [Accessed: 17-Nov-2021].

[3] NBA. (2021). NBA Advanced Stats [Online]. Available: www.nba.com/stats/teams [Accessed: 17-Nov-2021].

[4] B. Bodkin & M. Kaskey-Blomain. (2020). NBA Suspends Season Due to Coronavirus Outbreak; Owners Preparing For No Games Until June, Per Report [Online]. Available: https://www.cbssports.com/nba/news/nba-suspends-season-due-to-coronavirus-outbreak-owners-preparing-for-no-games-until-june-per-report/ [Accessed: 17-Nov-2021].

[5] Basketball-Reference. (2021). Glossary [Online]. Available: https://www.basketball-reference.com/about/glossary.html#mp [Accessed: 18-Nov-2021].

[6] NBA (2021). Frequently Asked Questions – What is PIE [Online]. Available: https://www.nba.com/stats/help/faq/#:~:text=In%20its%20simplest%20terms%2C%20PIE,to%20be%20a%20winning%20team [Accessed: 18-Nov-2021].

[7] C. Hsu & C. Chang & C. Lin (2016). A Practical Guide to Support Vector Classification [Online]. Available: https://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf [Accessed: 21-Nov-2021].

[8] G. James, D. Witten, T. Hastie, R. Tibshirani (2013). An Introduction to Statistical Learning. Springer. p. 6.

10. Appendix

Appendix 1 – Preprocessing Codes

```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

raw_data = pd.read_excel("raw_data.xlsx")
raw_data = raw_data.iloc[:,5:]
correlation = raw_data.corr(method='pearson')
f, ax = plt.subplots(figsize=(15, 10))
ax = sns.heatmap(correlation, center = 0,  linewidths=.1)
#t = sns.pairplot(correlation)
```

Appendix 2 – Logistic Regression Codes

```
#import libraries
import pandas as pd
import numpy as np
from timeit import default_timer as timer

start = timer()

def TrainTestSplit(X, y, raw_data, split = 0.8):
    X_train = X[:int(raw_data.shape[0]*0.8)]
    X_test = X[int(raw_data.shape[0]*0.8):]
    y_train = y[:int(raw_data.shape[0]*0.8)]
    y_test = y[int(raw_data.shape[0]*0.8):]
    return X_train, X_test, y_train, y_test

def ValidationSplit(X_train, y_train, split = 0.8):
    X_train = X_train[:int(X_train.shape[0]*0.8)]
    X_val = X_train[int(X_train.shape[0]*0.8):]
    y_train = y_train[:int(y_train.shape[0]*0.8)]
```

```
    y_val = y_train[int(y_train.shape[0]*0.8):]
    return X_train, X_val, y_train, y_val

raw_data = pd.read_excel("raw_data.xlsx")

#Splitting data to predictors and response
y = raw_data["Team1Win"]
X = raw_data.loc[:, raw_data.columns != "Team1Win"]

X_train, X_test, y_train, y_test = TrainTestSplit(X,y,raw_data)
X_train, X_val, y_train, y_val = ValidationSplit(X_train,y_train)

y_test = y_test.to_frame()
y_train = y_train.to_frame()
y_val = y_val.to_frame()

y_val['index_col'] = y_val.index
y_test['index_col'] = y_test.index
y_train['index_col'] = y_train.index

X_train['index_col'] = X_train.index
X_test['index_col'] = X_test.index
X_val['index_col'] = X_val.index

df_train = pd.merge(X_train, y_train,how="left")
df_val = pd.merge(X_val, y_val,how="left")
df_test = pd.merge(X_test, y_test,how="left")

df_train = df_train.iloc[:,5:]
df_val = df_val.iloc[:,5:]
df_test = df_test.iloc[:,5:]
```

```
df_train = df_train.drop(["Team1AST RATIO", "Team2AST RATIO", "Team1PIE",
        "Team2PIE"], axis = 1)
df_val = df_val.drop(["Team1AST RATIO", "Team2AST RATIO", "Team1PIE",
        "Team2PIE"], axis = 1)
df_test = df_test.drop(["Team1AST RATIO", "Team2AST RATIO", "Team1PIE",
        "Team2PIE"], axis = 1)


df_test = df_test.drop('index_col',1).values
df_train = df_train.drop('index_col',1).values
df_val = df_val.drop('index_col',1).values


y_test = y_test.drop('index_col',1).values
y_train = y_train.drop('index_col',1).values
y_val = y_val.drop('index_col',1).values


def find_min_max(dataset):
    minmax = list()
    for i in range(len(dataset[0])):
        col_values = []
        for row in dataset:
            col_values.append(row[i])
        value_min = min(col_values)
        value_max = max(col_values)
        minmax.append([value_min, value_max])
    return minmax
#normalize dataset
def normalize_dataset(dataset, minmax):
    for row in dataset:
        for i in range(len(row)):
            row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])


# dataset normalization
normalize_dataset(df_train,find_min_max(df_train))
```

```
normalize_dataset(df_test,find_min_max(df_test))
normalize_dataset(df_val,find_min_max(df_val))

def sigmoid(z):
    sig = (1/(1+np.exp(-z)))
    return sig

def model_fit_full(features, labels, rate,iterations):
    weights = np.zeros(len(features[0])+1)
    beta0 = weights[0]
    betai = weights[1:len(weights)]
    for iteration in range(iterations):
        z = beta0 + np.dot(features, betai)
        predicted_labels = sigmoid(z)
        error = np.subtract(labels,predicted_labels)[:,0]
        gradient = np.dot(features.T, error)
        beta0 += rate * np.sum(error)
        betai += rate * gradient
    predicted_weights = np.append(beta0, betai)
    return predicted_weights

def predict(test_features,weights):
    beta0 = weights[0]
    betai = weights[1:len(weights)]
    predicted_labels = []
    for c in range(len(test_features)):
        score = beta0 + np.sum(np.dot(test_features[c], betai))
        prob_true = sigmoid(score)
        prob_false = 1-sigmoid(score)
        if prob_false >= prob_true:
            prediction = 0
        else:
```

```
            prediction = 1
        predicted_labels.append(prediction)
    return predicted_labels

def calculate_accuracy(predicted_labels,test_labels):
    tp = 0.0 #true positive count
    tn = 0.0 #true negative count
    fp = 0.0 #false positive count
    fn = 0.0 #false negative count
    for i in range(len(predicted_labels)):
        prediction = predicted_labels[i]
        actual = test_labels[i]
        if ((prediction == 1) & (actual == 1)):
            tp+=1
        elif ((prediction == 1) & (actual == 0)):
            fp+=1
        elif ((prediction == 0) & (actual == 0)):
            tn+=1
        elif ((prediction == 0) & (actual == 1)):
            fn+=1
    accuracy = ((tp+tn)/(tp+tn+fp+fn))*100
    return accuracy,tp,tn,fp,fn

def performance_metrics(tp,tn,fp,fn):
    precision = tp/(tp+fp+0.0000001)
    recall = tp/(tp+fn+0.0000001)
    NPV = tn/(tn+fn+0.0000001)
    FPR = fp/(fp+tn+0.0000001)
    FDR = fp/(fp+tp+0.0000001)
    F1_score = 2*precision*recall/(precision+recall+0.0000001)
    F2_score = 5*precision*recall/(4*precision+recall+0.0000001)
    print("Precision: ",precision)
```

```
    print("Recall: ",recall)
    print("NPV: ",NPV)
    print("FPR: ",FPR)
    print("FDR: ",FDR)
    print("F1 score: ",F1_score)
    print("F2_score: ",F2_score)


################## Validation ##################################################
#iteration = [200, 1000, 2000]
#learning_rates = [1e-7,1e-6,1e-5]
#accuracy_max = 0
#perf_metrics = []
#for iterations in iteration:
#    for rate in learning_rates:
#        weights = model_fit_full(df_train, y_train,rate,iterations)
#        predicted_labels = predict(df_test,weights)
#        accuracy,tp,tn,fp,fn = calculate_accuracy(predicted_labels,y_test)
#      print("When learning rate is equal to {} and iteration is equal to {}".format(rate,iterations))
#        print("Accuracy: %"+str(accuracy))
#        print("tp: "+str(tp))
#        print("tn: "+str(tn))
#        print("fp: "+str(fp))
#        print("fn: "+str(fn))
#        performance_metrics(tp,tn,fp,fn)
#        if accuracy_max < accuracy:
#            accuracy_max = accuracy
#            params = [rate, iterations]
#        perf_metrics.append([iterations,rate,accuracy])
################## Validation ##################################################
```

```
print("Performance metrics for full gradient ascent when learning rate is equal to 1e-6 and
      iteration is equal to 2000")
weights = model_fit_full(df_train, y_train,1e-6,2000)
predicted_labels = predict(df_test,weights)
accuracy,tp,tn,fp,fn = calculate_accuracy(predicted_labels,y_test)
print("Accuracy: %"+str(accuracy))
print("tp: "+str(tp))
print("tn: "+str(tn))
print("fp: "+str(fp))
print("fn: "+str(fn))
performance_metrics(tp,tn,fp,fn)

end = timer()
print("Time elapsed: ",end - start)

############################ Plot ROC Curve ########################
from sklearn.metrics import roc_curve
from matplotlib import pyplot as plt
lr_fpr, lr_tpr, _ = roc_curve(y_test, predicted_labels)
plt.plot(lr_fpr, lr_tpr, marker='.', label='Logistic Regression')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title("ROC")
plt.legend()
plt.show()
```

Appendix 3 – SVM Codes

```
import numpy as np
import pandas as pd
from timeit import default_timer as timer

start = timer()
```

```
def TrainTestSplit(X,y,raw_data,split = 0.8):
    X_train = X[:int(raw_data.shape[0]*0.8)]
    X_test = X[int(raw_data.shape[0]*0.8):]
    y_train= y[:int(raw_data.shape[0]*0.8)]
    y_test = y[int(raw_data.shape[0]*0.8):]
    return X_train, X_test, y_train, y_test

def ValidationSplit(X_train, y_train, split = 0.8):
    X_train = X_train[:int(X_train.shape[0]*0.8)]
    X_val = X_train[int(X_train.shape[0]*0.8):]
    y_train = y_train[:int(y_train.shape[0]*0.8)]
    y_val = y_train[int(y_train.shape[0]*0.8):]
    return X_train, X_val , y_train, y_val

raw_data = pd.read_excel("raw_data.xlsx")

#Splitting data to predictors and response
y = raw_data["Team1Win"]
X = raw_data.drop(["Visitor/Neutral","PTS","Home/Neutral",
                   "PTS.1","Team1Win","Team1AST RATIO", "Team2AST RATIO", "Team1PIE",
        "Team2PIE"], axis = 1)

X_train, X_test, y_train, y_test = TrainTestSplit(X,y,raw_data)
X_train, X_val, y_train, y_val = ValidationSplit(X_train,y_train)
X_train = X_train.values
X_test = X_test.values
y_train = y_train.values
y_test = y_test.values
X_val = X_val.values
y_val = y_val.values

def find_min_max(dataset):
```

```
    minmax = list()
    for i in range(len(dataset[0])):
        col_values = []
        for row in dataset:
            col_values.append(row[i])
        value_min = min(col_values)
        value_max = max(col_values)
        minmax.append([value_min, value_max])
    return minmax
#normalize dataset
def normalize_dataset(dataset, minmax):
    for row in dataset:
        for i in range(len(row)):
            row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])

# dataset normalization
normalize_dataset(X_train,find_min_max(X_train))
normalize_dataset(X_test,find_min_max(X_test))
normalize_dataset(X_val,find_min_max(X_val))

def fit(X,y,iteration,learning_rate,lambda_param):
    n_samples, n_features = X.shape
    y_hat = np.where(y <= 0, -1, 1)
    w = np.zeros(n_features)
    b = 0
    for _ in range(iteration):
        i = 0
        for x in X:
            condition = y_hat[i]*(np.dot(x,w)-b) >= 1
            if condition:
                w -= learning_rate*(2*lambda_param*w)
            else:
```

```
            w -= learning_rate*(2*lambda_param*w - np.dot(x,y_hat[i]))
            b -= learning_rate * y_hat[i]
        i+=1
    return w,b

def predict (X,w,b):
    predict = np.dot(X,w) - b
    return np.sign(predict)

def calculate_accuracy(predicted_labels,test_labels):
    tp = 0.0 #true positive count
    tn = 0.0 #true negative count
    fp = 0.0 #false positive count
    fn = 0.0 #false negative count
    for i in range(len(predicted_labels)):
        prediction = predicted_labels[i]
        actual = test_labels[i]
        if ((prediction == 1) & (actual == 1)):
            tp+=1
        elif ((prediction == 1) & (actual == 0)):
            fp+=1
        elif ((prediction == 0) & (actual == 0)):
            tn+=1
        elif ((prediction == 0) & (actual == 1)):
            fn+=1
    accuracy = ((tp+tn)/(tp+tn+fp+fn))*100
    return accuracy,tp,tn,fp,fn

def performance_metrics(tp,tn,fp,fn):
    precision = tp/(tp+fp+0.0000001)
    recall = tp/(tp+fn+0.0000001)
    NPV = tn/(tn+fn+0.0000001)
```

```
    FPR = fp/(fp+tn+0.0000001)
    FDR = fp/(fp+tp+0.0000001)
    F1_score = 2*precision*recall/(precision+recall+0.0000001)
    F2_score = 5*precision*recall/(4*precision+recall+0.0000001)
    print("Precision: ",precision)
    print("Recall: ",recall)
    print("NPV: ",NPV)
    print("FPR: ",FPR)
    print("FDR: ",FDR)
    print("F1 score: ",F1_score)
    print("F2_score: ",F2_score)

################## Validation ##################################################
#iteration = [200, 1000, 2000]
#learning_rate = [1e-3,1e-4,1e-5]
#lambda_par = [1e-2,1e-3]
#accuracy_max = 0
#perf_metrics = []
#for iterations in iteration:
#    for learning_rates in learning_rate:
#        for lambda_pars in lambda_par:
#
#            w,b = fit(X_train,y_train,iterations,learning_rates,lambda_pars)
#            predictions = predict(X_test,w,b)
#
#            true_predictions = []
#            for prediction in predictions:
#                # print(prediction)
#                if prediction == -1:
#                    true_predictions.append(0)
#                else:
```

```
#                 true_predictions.append(1)
#             print("When iterations is equal to {} and learning rate is equal to {} and lambda is equal to {}".format(iterations,learning_rates,lambda_pars))
#             accuracy,tp,tn,fp,fn = calculate_accuracy(true_predictions,y_test)
#             print("Accuracy: %"+str(accuracy))
#             print("tp: "+str(tp))
#             print("tn: "+str(tn))
#             print("fp: "+str(fp))
#             print("fn: "+str(fn))
#             performance_metrics(tp,tn,fp,fn)
#             if accuracy_max < accuracy:
#                 accuracy_max = accuracy
#                 params = [iterations, learning_rates, lambda_pars]
#             perf_metrics.append([iterations,learning_rates,lambda_pars,accuracy])
################## Validation ##################################################

w,b = fit(X_train,y_train,1000,1e-4,1e-2)
predictions = predict(X_test,w,b)

true_predictions = []
for prediction in predictions:
    # print(prediction)
    if prediction == -1:
        true_predictions.append(0)
    else:
        true_predictions.append(1)

accuracy,tp,tn,fp,fn = calculate_accuracy(true_predictions,y_test)
print("When iterations is equal to 1000 and learning rate is equal to 0.0001 and lambda is equal to 0.01")
print("Accuracy: %"+str(accuracy))
print("tp: "+str(tp))
```

```
print("tn: "+str(tn))
print("fp: "+str(fp))
print("fn: "+str(fn))
performance_metrics(tp,tn,fp,fn)

end = timer()
print("Time elapsed: ",end - start)

############################ Plot ROC Curve ######################
from sklearn.metrics import roc_curve
from matplotlib import pyplot as plt
lr_fpr, lr_tpr, _ = roc_curve(y_test, true_predictions)
plt.plot(lr_fpr, lr_tpr, marker='.', label='SVM')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title("ROC")
plt.legend()
plt.show()
```

Appendix 4 – Decision Tree Codes

```
import pandas as pd
import numpy as np
from timeit import default_timer as timer

start = timer()

def entropy(data):
    labels = np.unique(data)
    entropy = 0
    for label in labels:
        p = len(data[data == label]) / len(data)
        entropy += -p * np.log2(p)
    return entropy
```

```
def gini_index(data):
    labels = np.unique(data)
    gini = 0
    for label in labels:
        p = len(data[data == label]) / len(data)
        gini += p**2
    gini = 1 - gini
    return gini

def information_gain(parent, l_child, r_child):
    weight_l = len(l_child) / len(parent)
    weight_r = len(r_child) / len(parent)
    gain = gini_index(parent) - (weight_l * gini_index(l_child) + weight_r * gini_index(r_child))
    return gain

def split(dataset, feature_index, threshold):
    dataset_left = np.array([row for row in dataset if row[feature_index] <= threshold])
    dataset_right = np.array([row for row in dataset if row[feature_index] > threshold])
    return dataset_left, dataset_right

def get_best_split(dataset, num_samples, num_features):
    best_split = {}
    max_info_gain = -1000000000
    for feature_index in range(num_features):
        feature_values = dataset[:, feature_index]
        candidate_thresholds = np.unique(feature_values)
        for threshold in candidate_thresholds:
            dataset_left, dataset_right = split(dataset, feature_index, threshold)
            if len(dataset_left) > 0 and len(dataset_right) > 0:
                y, left_y, right_y = dataset[:, -1], dataset_left[:, -1], dataset_right[:, -1]
                curr_info_gain = information_gain(y, left_y, right_y)
                if curr_info_gain > max_info_gain:
                    best_split["feature_index"] = feature_index
```

```
                best_split["threshold"] = threshold
                best_split["dataset_left"] = dataset_left
                best_split["dataset_right"] = dataset_right
                best_split["info_gain"] = curr_info_gain
                max_info_gain = curr_info_gain
    return best_split

def calculate_leaf_value(Y):
    Y = list(Y)
    return max(Y, key=Y.count)

def build_tree(dataset, curr_depth=0, min_samples_split = 2, max_depth= 2):
    X, Y = dataset[:,:-1], dataset[:,-1]
    num_samples, num_features = np.shape(X)
    if num_samples >= min_samples_split and curr_depth <= max_depth:
        best_split = get_best_split(dataset, num_samples, num_features)
        if best_split["info_gain"]>0:
            left_subtree = build_tree(best_split["dataset_left"], curr_depth+1, min_samples_split,
        max_depth)
            right_subtree = build_tree(best_split["dataset_right"], curr_depth+1, min_samples_split,
        max_depth)
            node_obj = Node(best_split["feature_index"], best_split["threshold"],
                    left_subtree, right_subtree, best_split["info_gain"])
            return node_obj
    leaf_value = calculate_leaf_value(Y)
    node_obj = Node(value=leaf_value)
    return node_obj

def fit(X, Y, min_sample_split=2, mx_depth=2):
    dataset = np.concatenate((X, Y), axis=1)
    tree = build_tree(dataset,min_samples_split = min_sample_split, max_depth = mx_depth)
    return tree

def predict(test_set, root):
    prediction_list = [make_prediction(test_instance, root) for test_instance in test_set]
```

```
    return prediction_list


def make_prediction(test_instance, tree):
    if tree.value != None:
        return tree.value
    feature_val = test_instance[tree.feature_index]
    if feature_val <= tree.threshold:
        return make_prediction(test_instance, tree.left)
    else:
        return make_prediction(test_instance, tree.right)


class Node():
    def __init__(self, feature_index=None, threshold=None, left=None, right=None, info_gain=None,
          value=None):
        self.feature_index = feature_index
        self.threshold = threshold
        self.left = left
        self.right = right
        self.info_gain = info_gain
        self.value = value


def calculate_accuracy(predicted_labels,test_labels):
    tp = 0.0 #true positive count
    tn = 0.0 #true negative count
    fp = 0.0 #false positive count
    fn = 0.0 #false negative count
    for i in range(len(predicted_labels)):
        prediction = predicted_labels[i]
        actual = test_labels[i]
        if ((prediction == 1) & (actual == 1)):
            tp+=1
        elif ((prediction == 1) & (actual == 0)):
            fp+=1
        elif ((prediction == 0) & (actual == 0)):
```

```
            tn+=1
        elif ((prediction == 0) & (actual == 1)):
            fn+=1
    accuracy = ((tp+tn)/(tp+tn+fp+fn))*100
    return accuracy,tp,tn,fp,fn

def performance_metrics(tp,tn,fp,fn):
    precision = tp/(tp+fp+0.0000001)
    recall = tp/(tp+fn+0.0000001)
    NPV = tn/(tn+fn+0.0000001)
    FPR = fp/(fp+tn+0.0000001)
    FDR = fp/(fp+tp+0.0000001)
    F1_score = 2*precision*recall/(precision+recall+0.0000001)
    F2_score = 5*precision*recall/(4*precision+recall+0.0000001)
    print("Precision: ",precision)
    print("Recall: ",recall)
    print("NPV: ",NPV)
    print("FPR: ",FPR)
    print("FDR: ",FDR)
    print("F1 score: ",F1_score)
    print("F2_score: ",F2_score)

def TrainTestSplit(X, y, raw_data, split = 0.8):

    X_train = X[:int(raw_data.shape[0]*0.8)]
    X_test = X[int(raw_data.shape[0]*0.8):]
    y_train= y[:int(raw_data.shape[0]*0.8)].values.reshape(-1, 1)
    y_test = y[int(raw_data.shape[0]*0.8):].values.reshape(-1, 1)
    return X_train, X_test, y_train, y_test

def ValidationSplit(X_train, y_train, split = 0.8):
    X_train = X_train[:int(X_train.shape[0]*0.8)]
    X_val = X_train[int(X_train.shape[0]*0.8):]
    y_train = y_train[:int(y_train.shape[0]*0.8)]
```

```
    y_val = y_train[int(y_train.shape[0]*0.8):]
    return X_train, X_val , y_train, y_val


raw_data = pd.read_excel("raw_data.xlsx")

y = raw_data["Team1Win"]
X = raw_data.drop(["Visitor/Neutral","PTS","Home/Neutral",
            "PTS.1","Team1Win","Team1AST RATIO", "Team2AST RATIO", "Team1PIE", "Team2PIE"], axis = 1)

X_train, X_test, y_train, y_test = TrainTestSplit(X,y,raw_data)
X_test = np.array(X_test)
y_test = np.array(y_test)

X_train, X_val, y_train, y_val = ValidationSplit(X_train,y_train)
X_val = np.array(X_val)
y_val = np.array(y_val)



################## Validation ######################################################
#mx_depth = [2,3,4,5]
#min_sample_split = [10,100,1000,10000]
#accuracy_max = 0
#perf_metrics = []
#for depth in mx_depth:
#    for min_split in min_sample_split:
#        decision_tree = fit(X_train, y_train, min_split, depth)
#        prediction = predict(X_val, decision_tree)
#        accuracy,tp,tn,fp,fn = calculate_accuracy(prediction, y_val)
#                print("When max depth is equal to {} and minimum sample split is equal to {} ".format(depth,min_split))
#        print("Accuracy: %"+str(accuracy))
#        print("tp: "+str(tp))
```

```
#       print("tn: "+str(tn))
#       print("fp: "+str(fp))
#       print("fn: "+str(fn))
#       performance_metrics(tp,tn,fp,fn)
#       if accuracy_max < accuracy:
#           accuracy_max = accuracy
#           params = [min_split, depth]
#       perf_metrics.append([depth,min_split,accuracy])
################## Validation ######################################################

decision_tree = fit(X_train, y_train, 10, 5)
prediction = predict(X_test, decision_tree)

accuracy,tp,tn,fp,fn = calculate_accuracy(prediction,y_test)
print("When max depth is equal to 5 and minimum sample split is equal to 10 ")
print("Accuracy: %"+str(accuracy))
print("tp: "+str(tp))
print("tn: "+str(tn))
print("fp: "+str(fp))
print("fn: "+str(fn))
performance_metrics(tp,tn,fp,fn)

end = timer()
print("Time elapsed: ",end - start)
############################ Plot ROC Curve ######################
from sklearn.metrics import roc_curve
from matplotlib import pyplot as plt
lr_fpr, lr_tpr, _ = roc_curve(y_test, prediction)
plt.plot(lr_fpr, lr_tpr, marker='.', label='Decision Tree')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title("ROC")
plt.legend()
plt.show()
```