IPL Score Prediction

Table of Contents

Introduction

*The **Indian Premier League (IPL)** is a professional Twenty20 cricket league in India contested during March or April and May of every year by eight teams representing eight different cities in India*[11]. It is a very popular sporting event with a huge fan following across the globe. This growing popularity of the sport has influenced research on different aspects of the game.

Machine Learning has found entry into the field of sports analytics since long time. Different aspects of the game have been constantly studied and efforts have been made to make sense out of data to improve performance of teams as well as individuals. We aim at creating a Machine Learning model to predict the final score of an innings given a set of input parameters. This information can be crucial for team management to make decisions during a match. For e.g. this prediction can help decide the playing XI members for a match. It can also help one make decision regarding the choice of batsman or bowler to in the match.

In the current scenario, analysts make use of metrics such as Projected Run rate to predict the final score. They plug-in different run-rates to current score try to predict outcome. These models though sometimes effective are very naïve and do not take into consideration the influence of certain external factors. Machine learning models can identify features in the dataset which have more variance and observe patters to generate better outputs. The model we suggest would take into consideration the influence of additional features such as the current batsman, bowler, runs scored so far, venue of the match to predict the final score of an innings.

Literature Review

The project's work is closely related to the idea presented in the paper *Live Cricket Score and Winning Prediction* by Rameshwari Lokhande and P.M. Chawan[3] and has been a starting point for the project. In this paper, authors have presented the idea of live prediction of match in progress. The paper aims at predicting the final score as well as the winning probabilities of the teams. They have considered the importance of factors such as number of wickets fallen, venue of the match, ranking of the teams, pitch report, home team advantage, etc. The paper further proceeds with describing the importance of the features mentioned earlier. These factors have been considered during feature set selection in our project. Authors then proceed with the implementation of Linear Regression, Naïve Bayes Classifier and Reinforced Learning Algorithm models.

The problem of live prediction of final score depending on previous outcomes is analogous to the problem of predicting Stock Prices. Author of the paper *Predicting Stock Prices using LSTM*[9]. RNNs are powerful tools for processing sequential data. LSTM introduces memory cell to capture the dynamic change in data for a period. The paper presents different LSTM models with different configuration of topology and training methods. The paper starts with pre-processing of the data, then performs feature extraction and then describing different methodologies used to train models and selecting the optimum model. The models are used to predict closing values of stock using NIFTY's trading dataset. Author's use an RNN model with two LSTM layers and two dense layer's and then evaluate the performance of the models based on the metric Root Mean Squared Error(RMSE). We plan to adopt similar methodology while implementing the LSTM model to predict final score.

Dataset

The dataset has been obtained from the repository *Indian Premier League (Cricket)*[1] hosted on Kaggle. The repository has no usage constraints. The dataset comprised of two csv files "matches.csv" and "deliveries.csv". The characteristics of the individual files are as below:

1. **Matches.csv:**
   This data file comprises of records of all matches played in IPL from season 2008 to 2017. The data file comprises of 18 features. It contains data corresponding to the name of the teams, venue of the match, outcome, umpires and details pertaining to the matches played. There are 636 entries in the data file.

	id	season	city	date	team1	team2	toss_winner	toss_decision	result	dl_applied	winner	win_by_runs	win_by_wickets	player_of_match	venue	umpire1	umpire2	umpire3
0	1	2017	Hyderabad	4/5/2017	1	5	5	field	normal	0	1	35	0	Yuvraj Singh	Rajiv Gandhi International Stadium, Uppal	AY Dandekar	NJ Llong	
1	2	2017	Pune	4/6/2017	2	4	4	field	normal	0	4	0	7	SPD Smith	Maharashtra Cricket Association Stadium	A Nand Kishore	S Ravi	
2	3	2017	Rajkot	4/7/2017	3	6	6	field	normal	0	6	0	10	CA Lynn	Saurashtra Cricket Association Stadium	Nitin Menon	CK Nandan	
3	4	2017	Indore	4/8/2017	4	8	8	field	normal	0	8	0	6	GJ Maxwell	Holkar Cricket Stadium	AK Chaudhary	C Shamshuddin	
4	5	2017	Bangalore	4/8/2017	5	7	5	bat	normal	0	5	15	0	KM Jadhav	M Chinnaswamy Stadium			

Figure 3.1

2. **Deliveries.csv:**
   This data file comprises of records of every delivery bowled in each of the matches. The records are chronologically arranged. The data includes 23 features including the outcome of every delivery and the number of runs scores and the way runs were scored. There are 150460 entries in the data file.

	match_id	inning	batting_team	bowling_team	over	ball	batsman	non_striker	bowler	is_super_over	...	noball_runs	penalty_runs	batsman_runs	extra_runs	total_runs	score	final_score	player_dismissed	dismissal_kind	fielder
0	1	1	1	5	1	1	DA Warner	S Dhawan	TS Mills	0	...	0	0	0	0	0	0	171			
1	1	1	1	5	1	2	DA Warner	S Dhawan	TS Mills	0	...	0	0	0	0	0	0	171			
2	1	1	1	5	1	3	DA Warner	S Dhawan	TS Mills	0	...	0	0	4	0	4	4	171			
3	1	1	1	5	1	4	DA Warner	S Dhawan	TS Mills	0	...	0	0	0	0	0	4	171			
4	1	1	1	5	1	5	DA Warner	S Dhawan	TS Mills	0	...	0	0	0	2	2	6	171			

Figure 3.1

In addition to the above-mentioned files, we create a new file with records of the teams and numerically encode names of the teams in the matches and deliveries files.

Data Preprocessing

The dataset files are initially loaded using Python Pandas library. After loading the dataset files, we check for 'nan' values in the data and replace it by blank space. If not given a default data type, pandas expect the values to be in the float data type. There are multiple cells in the dataset with blank values for columns like fielder since every delivery would not result in a catch.

```
matches = matches.replace(np.nan, '', regex=True)
deliveries = deliveries.replace(np.nan, '', regex=True)
```

Figure 4.1

The data comprises of multiple features with categorical data like name of the batsmen, bowler, etc. For our models to function effectively we first perform label encoding and convert categorical data into numerical data.

After encoding these features, we perform PCA to assess the variance of different features through the data. After performing PCA we reckon that the *features 'is_super_over', 'wide_runs', 'bye_runs', 'legbye_runs', 'noball_runs', 'penalty_runs', 'batsman_runs', 'extra_runs', 'player_dismissed', 'dismissal_kind', 'fielder'* do not contribute much towards our algorithm. Hence, we drop these fields in the data to reduce the dimensionality of our data and to make our models less complex.

In addition to the above-mentioned processes, we need to perform feature extraction to extract two additional features that we require to perform prediction of the final score – *score and final_score. Score* denotes the total runs scored by the team at the end of the current frame in the data and *final_score* corresponds to the final batting score of the inning.

```
#Populate score field
for id in range(1,637):
    df = total_and_balls.loc[total_and_balls['match_id'] == id]
    total_score = df.loc[df.index[-1], "score"]
    total_and_balls.loc[total_and_balls['match_id'] == id, 'final_score']= total_score
```

Figure 4.2

	match_id	inning	batting_team	bowling_team	over	ball	batsman	non_striker	bowler	total_runs	score	final_score
0	1	1	1	5	1	1	103	347	331	0	0	171
1	1	1	1	5	1	2	103	347	331	0	0	171
2	1	1	1	5	1	3	103	347	331	4	4	171
3	1	1	1	5	1	4	103	347	331	0	4	171
4	1	1	1	5	1	5	103	347	331	2	6	171

Figure 4.3

After pre-processing the data, we set aside records of single match aside to evaluate all the optimum models simultaneously at the end. In our case we have set aside records corresponding to Match Id = 7 which was match between Mumbai Indians and Kolkata Knight Riders played in 2017 season.

After setting aside the evaluation data, we extract the target values from the data and then we split the data into Train and Test data in the ration 4:1. At the end we have train data *X_train* and *y_train* with 120100 records and test data *X_test* and *y_test* with 30025 records.

Models and Implementation

For this project, we have implemented four models – Linear Regression, K Nearest Neighbors Regressor, Random Forest Regressor and Recurrent Neural Networks using LSTM. We have selected four regression models with increasing complexity and will later compare these models to identify the tradeoff between performance and complexity of the models. In order to evaluate the performance of our regression models, we consider two metrics – R2 score and Mean Square Error (MSE)[10].

1. Linear Regression

Linear Regression fits a linear model with coefficients w = (w1, …, wp) to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation[5]. Since, Linear Regression does not have any tuning Hyper parameters, we simply create a model and fit our training data. We then evaluate the model by predicting values on test data. The evaluation results are as below:

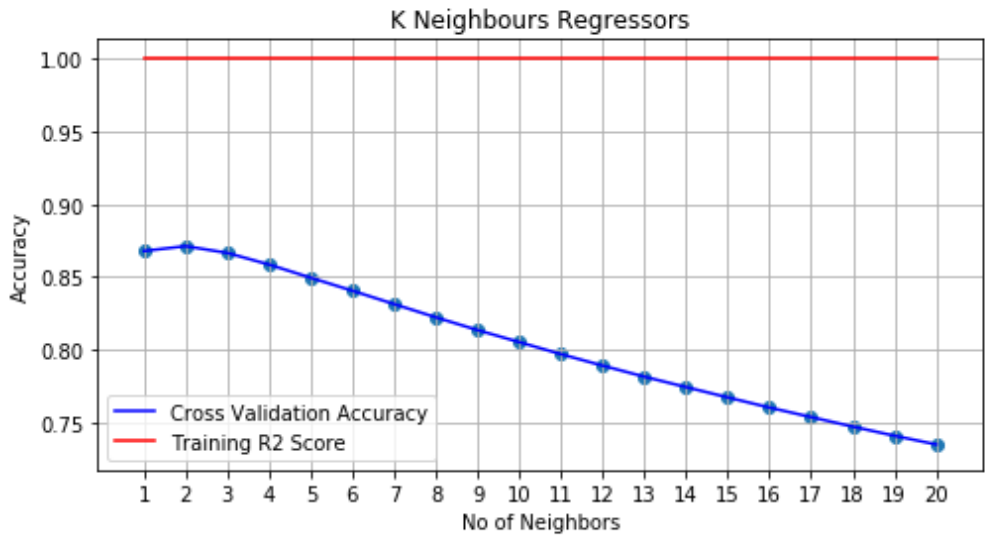
Train/Test	R2 Score	Mean Squared Error
Train Data	0.210787129937	678.9248990081413
Cross Validation	0.2105561586301486	
Test Data	0.21666095930040563	674.876222704071

Figure 5.1

Looking at the performance of the model on both train and test data we can see that the model is not able to identify a linear relationship between the input features and the prediction. Thus, we need a more complex model that can capture non-linear relationship between them.

2. K Nearest Neighbors Regressor

K Nearest Neighbors Regressor, predict the value of the target based on the value of k nearby neighbors based on distance. KNN performs better when in our n-dimensional space the data points that contribute to the same final score are nearby. In order to identify the most optimum model, we evaluate training error and cross validation error for the models for value of k from 1-20.




```
Maximum Cross Validation accuracy - Neighbors: 2, Accuracy: 0.8711764340407484
```

Figure 5.2

KNN models will always perform best on train data and thus, all the models have accuracy 1 on train data. We can see that the Cross-Validation Accuracy increase till k=2 and then decreases. We can infer that the optimum model is for the value of k=2 and after that point the model becomes over-fitting as the Cross-Validation Accuracy decreases further. In addition to the value of k, we set the value of parameter weights=”distance”. This gives weight points by the inverse of their distance.

We then evaluate the performance of this optimum model on test data.

Configuration: k=2, weights=distance

```
# Predict the best model on test data
model = KNeighborsRegressor(n_neighbors=2, weights='distance', p=2, n_jobs=-1)
knr = model.fit(X_train,y_train)
pred = knr.predict(X_test)
test_acc = r2_score(y_test, pred)
test_acc_mean = mean_squared_error(y_test, pred)
print(f'R square score of the model on test data: {test_acc}')
print(f'Mean squared error of the model on test data: {test_acc_mean}')
```

```
R square score of the model on test data: 0.9044300211769578
Mean squared error of the model on test data: 82.33715283027406
```

Figure 5.3

Test / Train	R2 Score	MSE
Train	1	0
Cross Validation	0.8711764340407484	
Test	0.9044300211769578	82.33715283027406

Figure 5.4

Looking at the performance of the model, we infer that in the dataset, the combination of features that contribute to a final score are closer in the n-dimensional space.

3. Random Forest Regressor

A random forest is a meta estimator that fits a number of classifying decision trees on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting[7]. Random forest uses several base decision tree estimators that train on random samples of data using random subset of features. Random forests are great at reducing bias and variance using basic non-complex models.

In order to identify the optimum model, we have tried multiple configurations using combinations of the parameters *max_depth* and *n_estimators*. The performance of the different configurations is as given below.

```
model = RandomForestRegressor(n_estimators = estimator, max_depth=depth, random_state=0, n_jobs=-1)
```

Number of Estimators	Maximum Depth	Training Accuracy	Cross Validation Accuracy
10	10	0.4840049028425052	0.47693872639379864
10	100	0.9864901860876021	0.9286533825286192
50	10	0.491970066893198	0.4813928338003521
50	100	0.9927607632575627	0.9443133126256662
100	10	0.4901200917987477	0.4844173565056498
100	100	0.9933213438845026	0.9458115557919676
1000	10	0.4915126219793772	0.485660999465179
1000	100	0.9938383645011852	0.9468169475827226

```
Maximum Cross Validation Accuracy: 0.9468169475827226, No of estimators: 1000, Max Depth: 100
```

Figure 5.5

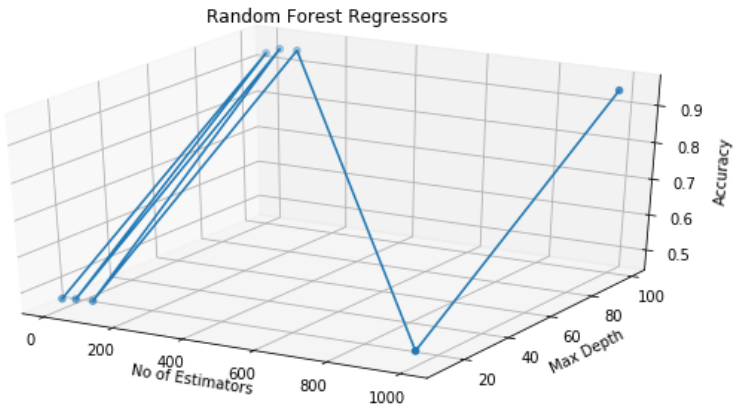


Figure 5.6

Looking at the figures, we identify the model with Highest Cross Validation Accuracy i.e. model with 1000 estimators and 100 depth to be the optimum model. Evaluating the performance of this model on test data, we get the results as below:

```
# Predict the best model on test data
model = RandomForestRegressor(n_estimators =1000, max_depth=100, random_state=0)
rfr = model.fit(X_train,y_train)
pred = rfr.predict(X_test)
test_acc = r2_score(y_test, pred)
test_acc_mean = mean_squared_error(y_test, pred)
print(f'R square score of the model on test data: {test_acc}')
print(f'Mean squared error of the model on test data: {test_acc_mean}')
```

```
R square score of the model on test data: 0.9567427701315365
Mean squared error of the model on test data: 37.267740252289755
```

Test / Train	R2 Score	MSE
Train	0.9938383645011852	
Cross Validation	0.9468169475827226	
Test	0.9567427701315365	37.267740252289755

Figure 5.7

Random Forest Regressor performs better than the models, discussed previously and has great performance on the test data as well. Random Forest Regressor also helps us get an insight on the importance of the features within the data. It is strikingly similar to what we have obtained using PCA. Features such as venue of the match, batsman on strike and bowler has higher importance as compared to other factors.

```
model.feature_importances_
```

```
array([0.5104596 , 0.0173503 , 0.07689576, 0.07390588, 0.06671664,
       0.00576856, 0.0351335 , 0.03760185, 0.04856545, 0.00410225,
       0.12350022])
```

Figure 5.8

4. Recurrent Neural Networks (RNNs) using LSTM layers

RNNs are a class of artificial neural networks with capability to use their internal state to process variable length input sequences. These models are great to work on temporal data. As discussed in the Literature Review, our problem is analogous of stock market prediction and thus, RNNs using Long Short-Term Memory layers can be used for prediction.

The input data to the Keras models are reshaped to include “timestamp” for the LSTM layers.

Performance before Normalization of data: The models used so far were trained on un-normalized data and performed well. But, the LSTM model initially performed poorly on the un-normalized data. For multiple configurations, with combinations of number of nodes and layers, the models always converged to an MSE value of about 660 with R2 score of about 0.26. The models were tested for different types of optimizers: 'adam', 'SGD' with different learning rates.

Performance after normalization of data: The performance of the models changed drastically when we normalized the data. Also, after running multiple trials we observe that the model converges very quickly within 50 epochs. This helped in identifying the ideal value of the hyperparameter *epochs*. This inference is supported by the Training Loss vs Epochs graph. Also, after trying multiple trials for the activation functions, it was observed that the models function better when we use 'relu' activation for the dense layers.

Code Excerpt of the optimum model during training:

```
# Model 8
# A RNN with one LSTM Layers with 200 neurons and three dense layers each with 100 nodes
regressor8 = Sequential()
regressor8.add(LSTM(200, input_shape=(X_trains.shape[1], X_trains.shape[2])))
regressor8.add(Dense(100, activation='relu'))
regressor8.add(Dense(100, activation='relu'))
regressor8.add(Dense(100, activation='relu'))
regressor8.add(Dense(1))
regressor8.compile(loss='mean_squared_error', optimizer='adam')
history8 = regressor8.fit(X_trains, y_train_nn, epochs=50, batch_size=32, verbose=1)
```

```
Epoch 1/50
120100/120100 [==============================] - 22s 180us/step - loss: 0.0153
Epoch 2/50
120100/120100 [==============================] - 21s 176us/step - loss: 0.0135
Epoch 3/50
120100/120100 [==============================] - 21s 177us/step - loss: 0.0129
Epoch 4/50
120100/120100 [==============================] - 21s 175us/step - loss: 0.0123
Epoch 5/50
```

The different configurations tested for identifying the optimum model is as below:

Number of LSTM Layers	Nodes in LSTM Layers	Number of Dense Layers	Nodes in Dense Layers	MSE
1	50	2	50	0.0057
1	100	2	50	0.0048
1	50	0	50	0.0124
1	100	0	100	0.0123
2	50	0	50	0.0120
2	100	0	100	0.0121
1	100	3	100	0.0022
1	200	3	100	0.0020
1	200	3	200	0.0018
1	300	3	200	0.0019

Figure 5.9

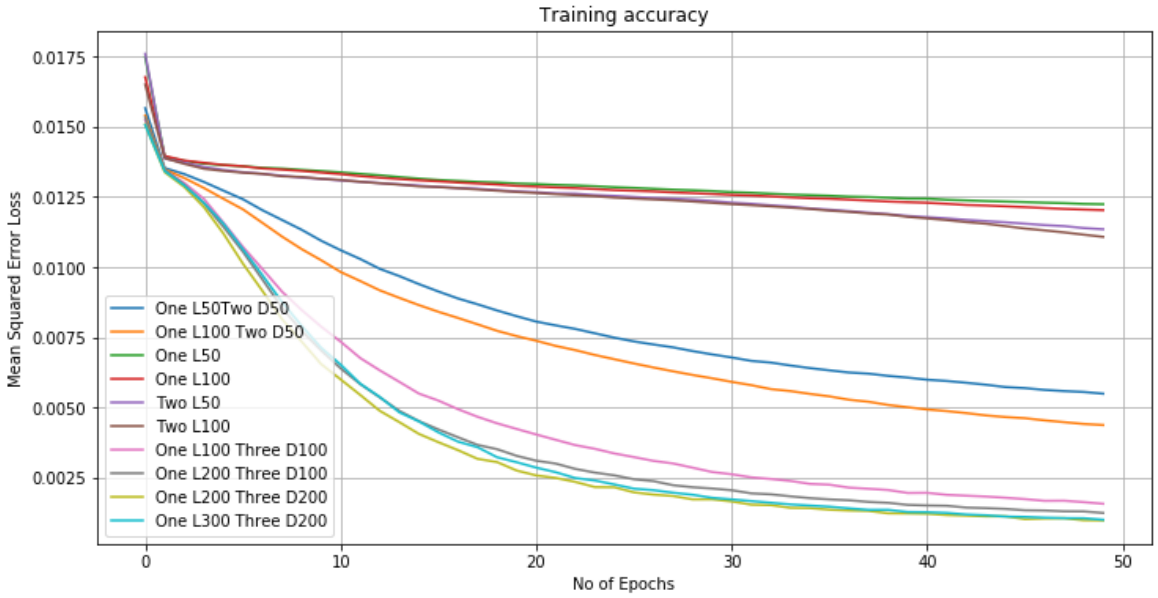


Figure 5.11

Looking at the figures, we reckon that the optimum model is with the configuration:
No of LSTM Layers: 1, No of Nodes in LSTM: 200, No of Dense Layers: 3, Nodes in Dense Layers: 100

Models with a greater number of layers and nodes seem to be over-fitting and hence, we consider this as the optimum model. Performance of the models on test data can be evaluated as below:

```
# Prediction of best model
#Optimum model is with the configuration One LSTM layer with 200 nodes and Three Dense Layers with 200 ndoes each
pred = regressor8.predict(X_tests)
test_acc = r2_score(y_test_nn, pred)
test_acc_mean = mean_squared_error(y_test_nn, pred)
print(f'R square score of the model on test data: {test_acc}')
print(f'Mean squared error of the model on test data: {test_acc_mean}')
```

```
R square score of the model on test data: 0.9241242817444608
Mean squared error of the model on test data: 0.001338420633915955
```

Test / Train	R2 Score	MSE
Train		0.020
Test	0.9241242817444608	0.001338420633915955

Figure 5.12

The RNN as expected being a complex model is capable of identifying the non-linear complex relationship between the features and the prediction values.

Comparison of models

Consolidating the data of the performance of different models, we can evaluate the optimum model for the given problem.

Test Data Performance:

Model	R2 Score – Test	MSE – Test
Linear Regression	0.21666095930040563	674.876222704071
K Nearest Neighbors Regressor	0.9044300211769578	82.33715283027406
Random Forest Regressor	0.9567427701315365	37.267740252289755
Recurrent Neural Networks using LSTM	0.9241242817444608	0.001338 (Normalized)

Figure 6.1

Evaluation on Separated Data:

We also, evaluate the performance of the models on the evaluation data of a match that we have set aside. We compare it against the projected run rate of the match.

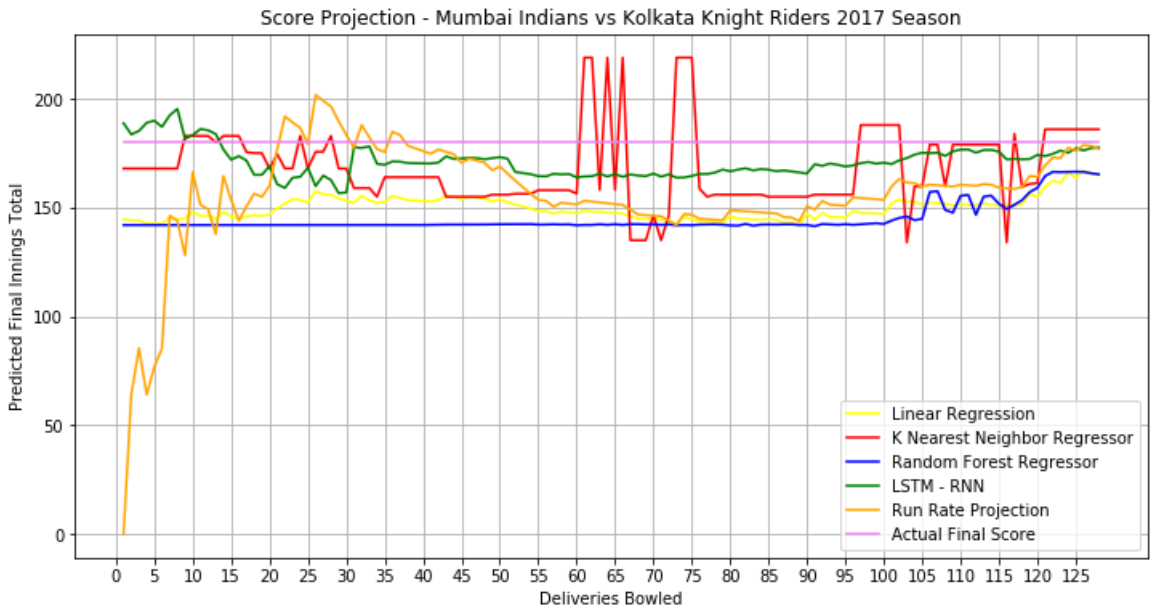


Figure 6.2

The figure represents the ball to ball prediction of the final score of the innings using the optimum models for each of the models considered. The purple line represents the actual final score. We can see that RNN with LSTM (green) performs performed better than other models and the projected run-rate. Right from the beginning the predictions are lose to the result and is very stable. Random Forest Regressor model even with better accuracy is stable yet produces a sub-optimal solution.

Conclusion

We have compared the performance of the models on test data and the evaluation data of a match.

Looking at figures 6.1 and 6.2 we can see that the models Random Forest Regressors and LSTM RNNs perform better than the rest of the models. Both the models have good accuracy on test data. Random Forest Regressors(RFRs) utilizing ensemble methods can produce good models with low values of bias and variance . Neural networks on the other are very efficient in identifying non-linear mapping of features and target values and demonstrate high accuracy. Both the models take considerable amount of time for training, but the RFRs tend to take more time while predicting. Looking at the performance of the models on the evaluation set of the match, we can see that the prediction by the RNN has been stable and very close to the actual value right from the beginning. RFRs on the other hand, despite demonstrating high accuracy on test data does not seem to perform well for the instance.

Keeping these factors in mind, we would like to propose the **Recurrent Neural Networks Model using LSTM** for predicting the final score for a team in a match.

References

1. Indian Premier League Kaggle Dataset
2. Predicting Stock Prices Using LSTM
3. Live Cricket Score and Winning Prediction
4. Sklearn Data Processing
5. Sklearn Linear Regression
6. Sklearn K Nearest Neighbor Regressor
7. Sklearn Random Forest Regressor
8. Keras LSTM
9. Predicting Stock Price with LSTM
10. Metrics and scoring: quantifying the quality of predictions
11. Indian Premier League