BIA 678 Final Report
Professor Guntupalli
Team 8
Braedon Brenna, Saadeldin Mostafa, Samrudh Nagesh, & Aiwu Song
5/11/2023

Football Player Valuation Using FIFA23 Dataset

Introduction

The purpose of this project is to create a model using Spark to use the EA Sports FIFA 23 dataset to predict the value of football players. The project utilizes 4 different regression models, including Linear Regression, Random Forest Regression, Generalized Linear Regression and Gradient Boosting. Afterwards, the performance of these models is evaluated using the Root Mean Square Error (RMSE) metric and time for different fractions of the dataset. The performance of each test is evaluated on local, 3 nodes, 4 nodes and 6 nodes by using Google Dataproc Clusters. The results are compared, finding the best algorithm in terms of accuracy and time to predict the value of football players.

The Data

EA Sports FIFA series produces the best selling football simulation video game every year. The developers analyze every football player playing for any top tier professional team and create the players who look and play exactly like their real life counterparts. The original mens players dataset consist of 109 columns and 10 million rows; most of the columns are for the purposes of gameplay, so they are dropped as they are not useful in our analysis. The dataset we use consists of 1,000,000 rows and 20 columns. Columns include data relevant to FIFA game mechanics like player_id, fifa_update, and fifa_version, which are not beneficial for our analysis, as well as variables like player_positions, overall, and potential which are pertinent to our analysis of football player value.

player_id	player_url	fifa_version	fifa_update	fifa_update_date	short_name	long_name	player_positions	overall	potential	...	cdm	rdm	rwb
158023	/player/158023/lionel-messi/230009	23	9	2023-01-13	L. Messi	Lionel Andrés Messi Cuccittini	RW	91	91	...	63+3	63+3	64+3
165153	/player/165153/karim-benzema/230009	23	9	2023-01-13	K. Benzema	Karim Benzema	CF, ST	91	91	...	64+3	64+3	64+3
188545	/player/188545/robert-lewandowski/230009	23	9	2023-01-13	R. Lewandowski	Robert Lewandowski	ST	91	91	...	66+3	66+3	64+3
192985	/player/192985/kevin-de-bruyne/230009	23	9	2023-01-13	K. De Bruyne	Kevin De Bruyne	CM, CAM	91	91	...	79+3	79+3	78+3
231747	/player/231747/kylian-mbappe/230009	23	9	2023-01-13	K. Mbappé	Kylian Mbappé Lottin	ST, LW	91	95	...	63+3	63+3	67+3

cdm	rdm	rwb	lb	lcb	cb	rcb	rb	gk	player_face_url
63+3	63+3	64+3	59+3	50+3	50+3	50+3	59+3	19+3	https://cdn.sofifa.net/players/158/023/23_120.png
64+3	64+3	64+3	60+3	55+3	55+3	55+3	60+3	18+3	https://cdn.sofifa.net/players/165/153/23_120.png
66+3	66+3	64+3	61+3	60+3	60+3	60+3	61+3	19+3	https://cdn.sofifa.net/players/188/545/23_120.png
79+3	79+3	78+3	74+3	68+3	68+3	68+3	74+3	21+3	https://cdn.sofifa.net/players/192/985/23_120.png
63+3	63+3	67+3	63+3	54+3	54+3	54+3	63+3	18+3	https://cdn.sofifa.net/players/231/747/23_120.png

Figure 1: *Extracts of the FIFA dataset*

Data Preprocessing

As previously mentioned, the original dataset downloaded from the EA sports website has 109 columns, most of which are for the gameplay mechanics and are irrelevant to a player's statistics and value. We drop the unwanted columns and consider only the columns which clearly explain the ability of a football player to any person who does not know about football. We only keep the player's height, weight, work rate, overall stat, potential stat, passing, shooting, defending, dribbling, pace and their playing positions. The numerical stats are between 0-99 where the higher number represents better ability.

short_name	player_pos	overall	potential	value_eur	wage_eur	age	height_cm	weight_kg	preferred_
L. Messi	RW	91	91	54000000	195000	35	169	67	Left
K. Benzema	CF, ST	91	91	64000000	450000	34	185	81	Right
R. Lewandowski	ST	91	91	84000000	420000	33	185	81	Right
K. De Bruyne	CM, CAM	91	91	107500000	350000	31	181	75	Right
K. MbappÃ©	ST, LW	91	95	190500000	230000	23	182	73	Right
T. Courtois	GK	90	91	90000000	250000	30	199	96	Left
M. Salah	RW	90	90	115500000	270000	30	175	71	Left
M. Neuer	GK	89	89	11500000	69000	36	193	93	Right
Neymar Jr	LW	89	89	99500000	200000	30	175	68	Right

weak_foo	skill_move	work_rate	pace	shooting	passing	dribbling	defending	physic
4	4	Low/Low	81	89	90	94	34	64
4	4	Medium/Medium	80	88	83	87	39	78
4	4	High/Medium	75	91	79	86	44	83
5	4	High/Medium	74	88	93	87	63	77
4	5	High/Low	97	89	80	92	36	76
3	1	Medium/Medium						
3	4	High/Medium	90	89	82	90	45	75
4	1	Medium/Medium						
5	5	High/Medium	87	83	85	93	37	61

Figure 2: *An extract of the reduced FIFA dataset, taking into account relevant metrics*

Categorical columns such as player_positions and work_rate are converted to numeric values. Work_rate is converted in a way that low work rate is valued lower than high work rate. Player positions are first converted from individual positions such as LW, ST, RW to a more genetic position FWD. We end up with three generic positions FWD, MID, DEF which mean forward, midfield, and defenders, respectively.

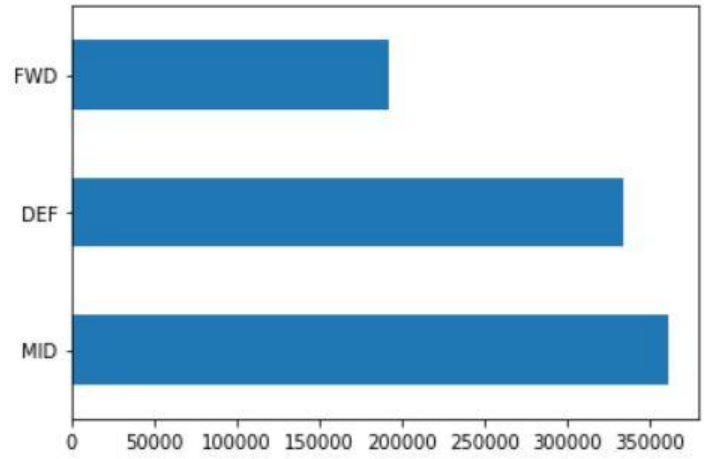


Figure 3: *Graph depicting the amount of players in FWD, DEF, and MID positions*

From **Figure 3** we can see that there are more midfielders in the dataset than defenders and the forwards are the least and hence they are more valuable. The player positions are then converted into numeric using dummy variables. After applying dummy variables, we get a dataset ready for training using machine learning algorithms.

overall	potential	value_eur	wage_eur	age	height_cm	weight_kg	weak_foot	skill_moves	work_rate	pace	shooting	passing	dribbling	defending	physic	fixed_pos_dummies
91	91	5.4E7	195000.0	35	169	67	4	4	1	81.0	89.0	90.0	94.0	34.0	64.0	2.0
91	91	6.4E7	450000.0	34	185	81	4	4	3	80.0	88.0	83.0	87.0	39.0	78.0	2.0
91	91	8.4E7	420000.0	33	185	81	4	4	4	75.0	91.0	79.0	86.0	44.0	83.0	2.0
91	91	1.075E8	350000.0	31	181	75	5	4	4	74.0	88.0	93.0	87.0	63.0	77.0	0.0
91	95	1.905E8	230000.0	23	182	73	4	5	3	97.0	89.0	80.0	92.0	36.0	76.0	2.0
90	90	1.155E8	270000.0	30	175	71	3	4	4	90.0	89.0	82.0	90.0	45.0	75.0	2.0

Figure 4: *Cleaned dataset ready for ML algorithm testing*

The table above shows the dataset which we can use for training machine learning algorithms. We start by using VectorAssembler to create a vector column for our features and our target variable is value_eur which is the player value in Euros.

The final_data consists of the features vector “features” and our target variable “value_eur”. We then split our dataset randomly into a training set and a testing set, using a 70:30 ratio.

```
assembler = VectorAssembler(inputCols=['overall',
 'potential',
 'wage_eur',
 'age',
 'height_cm',
 'weight_kg',
 'weak_foot',
 'skill_moves',
 'work_rate',
 'pace',
 'shooting',
 'passing',
 'dribbling',
 'defending',
 'physic'], outputCol="features")
output = assembler.transform(df4)
final_data = output.select("features", "value_eur")
```

Figure 5: *Vector assembler code*

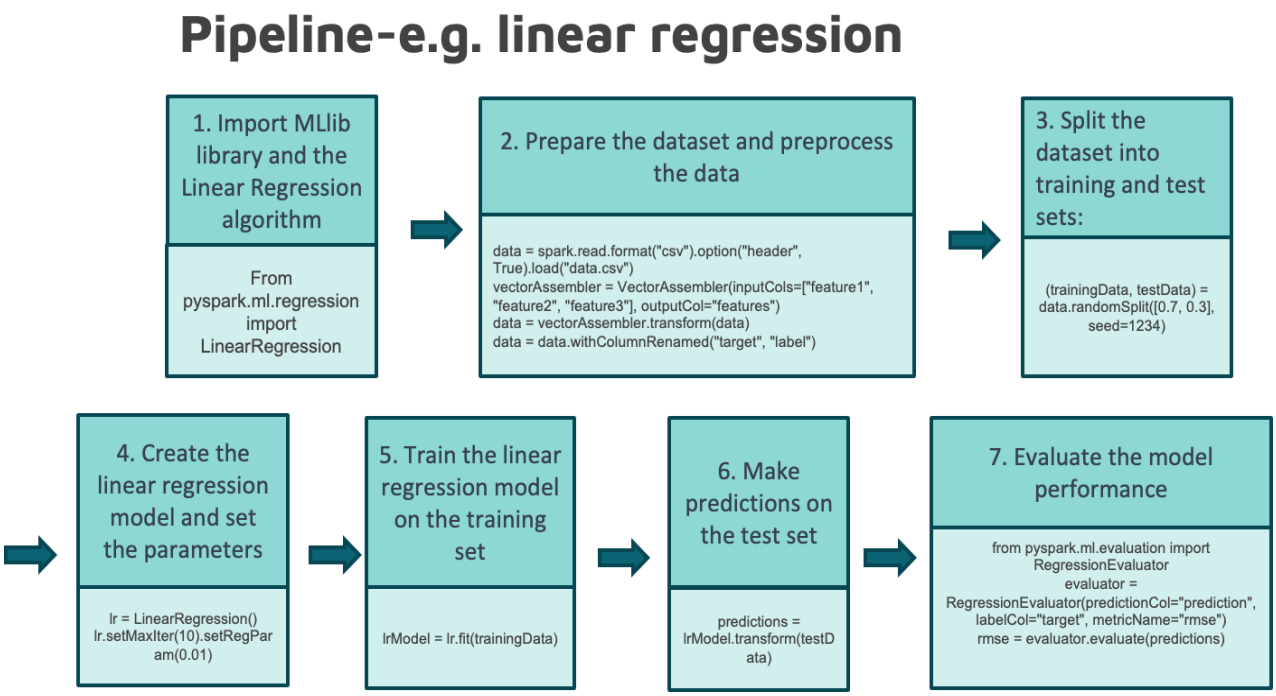


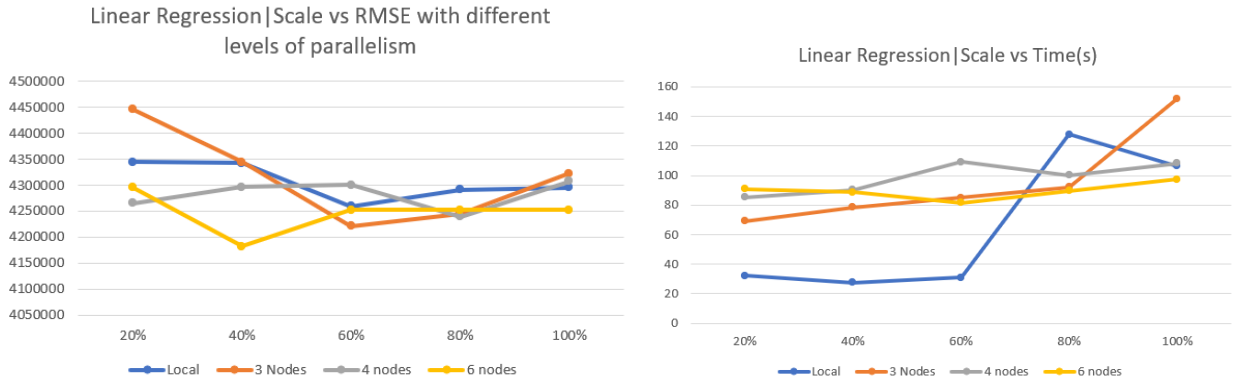
Figure 6: *Graphical representation the pipeline utilized in the ML testing*

Google Cloud Platform

We use GCP for parallel processing. Since we have a large dataset, we use Google Dataproc to create a dataproc cluster with 3 nodes, 4 nodes and 6 nodes to scale out our dataset. For each model, we partition the dataset into 20%, 40%, 60%, 80%, and 100% to test the overall speed and accuracy of the model on GCP. These RMSE values are plotted against the amount of time taken for GCP to complete the model.

Linear Regression

The first regression algorithm we used is linear regression. We use the default parameters for the linear regression algorithm and use the RMSE metric for evaluation of the algorithm. As seen in **Figures 7 & 8**, we observe that linear regression is not best suited for our dataset because we get a very high RMSE. The RMSE is over 4 million, although we are dealing with a large value target variable, the RMSE is very high regardless. We also observe that there is higher variance in RSME when we use 20% dataset across multiple nodes than when we use 100%, where it is more converged. We also observe that the time taken is high, 140 seconds for 100% of the dataset.



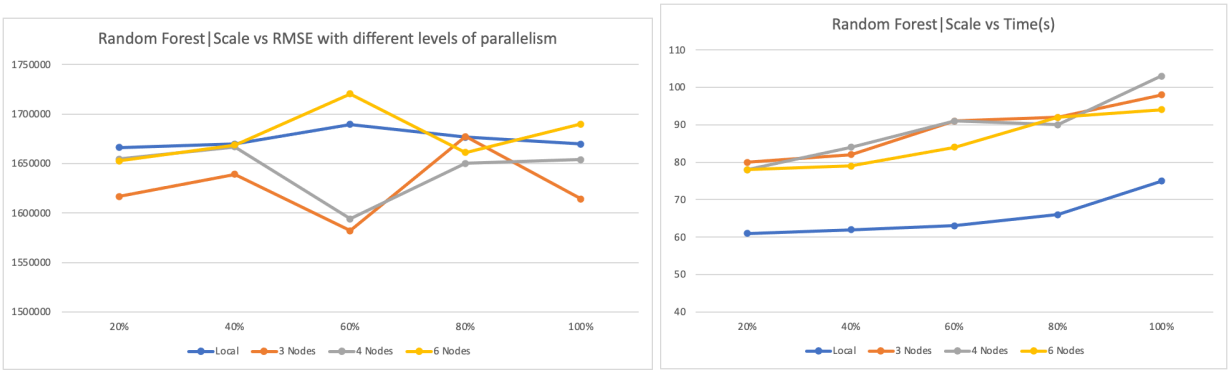
Figures 7 & 8: *Graphs depicting LR model performance, measuring RMSE and time*

```
# from pyspark.ml.regression import LinearRegression

# lr = LinearRegression(featuresCol="features", labelCol="value_eur")
# lr_model = lr.fit(train_data)


# # In[32]:


# print("Coefficients: " + str(lr_model.coefficients))
# print("Intercept: " + str(lr_model.intercept))


# # In[33]:


# #Summarize the model over the training set and print out some metrics:
# trainingSummary = lr_model.summary
# print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
# print("R squre: %f" % trainingSummary.r2)


# # In[34]:


# lr_predictions = lr_model.transform(test_data)
# lr_predictions.select("prediction","value_eur","features").show(5)
# from pyspark.ml.evaluation import RegressionEvaluator
# lr_evaluator = RegressionEvaluator(predictionCol="prediction",                 labelCol="value_eur",metricName="r2")
# print("R Squared (R2) on test data = %g" % lr_evaluator.evaluate(lr_predictions))


# # In[35]:


# test_result = lr_model.evaluate(test_data)
# print("Root Mean Squared Error (RMSE) on test data = %f" % test_result.rootMeanSquaredError)
```

Figure 9: *PySpark code for the LR model*

Random Forest

The second regression model used is the random forest model. The parameters used for each model included a number of Trees equal to 10 and the maximum Depth equal to 5. The model's effectiveness was evaluated based on the RMSE metric and the amount of time the model took to complete in GCP. As shown in **Figures 10 & 11**, the random forest model is more effective at predicting the value of football players than models like linear regression and generalized linear regression, as the RF model has a lower RMSE value. Interestingly, the RMSE values for 60% of the data had a wider range than other intervals of the model. As for the time taken to complete, the RF model was quickest on local, and relatively similar on each of the different node clusters. As expected, the overall time taken to complete increased as the amount of data present within the model increased.



Figures 10 & 11: *Graphs depicting RF model performance, measuring RMSE and time*

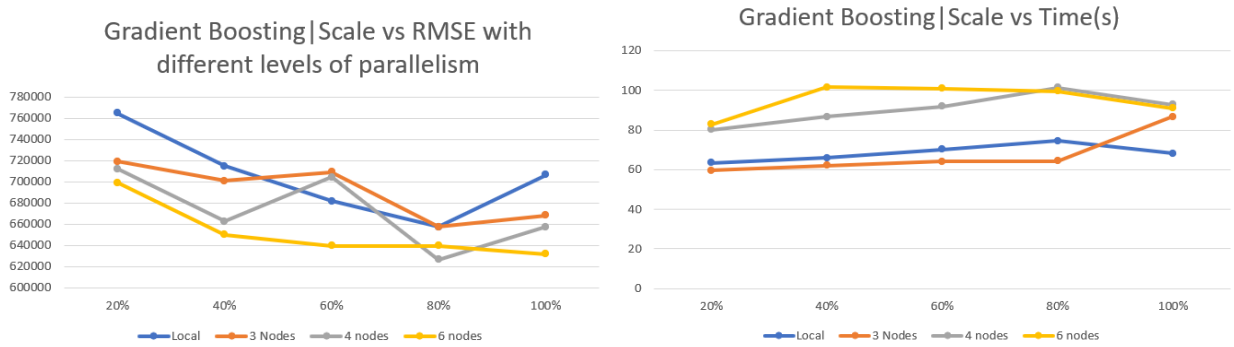
```
# In[51]: RANDOM FOREST


from pyspark.ml.regression import RandomForestRegressor

rf = RandomForestRegressor(featuresCol="features", labelCol="value_eur", numTrees=10, maxDepth=5, seed=42)
model = rf.fit(train_data)


# In[52]:


predictions = model.transform(test_data)


# In[54]:


evaluator = RegressionEvaluator(metricName="rmse", predictionCol="prediction", labelCol="value_eur")
rmse = evaluator.evaluate(predictions)
print("RMSE on test data = %g" % rmse)
```

Figure 12: *PySpark code for the RF model*

Gradient Boosting

The third regression model used is the Gradient Boosting model. The parameters used for this model are the default parameters. As seen in **Figures 13 and 14**, the gradient boosting model

performed best utilizing the 6 node cluster, in terms of RMSE. The time taken for the model to complete on the GCP Dataproc was relatively similar for each node cluster and percentage of data utilized. The 3 node cluster performs best in terms of time, but by a very slim margin.



Figures 13 & 14: *Graphs depicting GB model performance, measuring RMSE and time*

```
lr = GBTRegressor(featuresCol="features", labelCol="value_eur")
lr_model = lr.fit(train_data)
#lr_model.save('gs://bia-678-team/my-model')


# In[32]:


# print("Coefficients: " + str(lr_model.coefficients))
# print("Intercept: " + str(lr_model.intercept))


# In[33]:


#Summarize the model over the training set and print out some metrics:
# trainingSummary = lr_model.summary
# print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
# print("R squre: %f" % trainingSummary.r2)

lr_predictions = lr_model.transform(train_data)
lr_predictions.select("prediction","value_eur","features").show(5)
from pyspark.ml.evaluation import RegressionEvaluator
lr_evaluator = RegressionEvaluator(predictionCol="prediction",                 labelCol="value_eur",metricName="r2")
print("R Squared (R2) on train data = %g" % lr_evaluator.evaluate(lr_predictions))

lr_evaluator = RegressionEvaluator(predictionCol="prediction",                 labelCol="value_eur",metricName="rmse")

print("RMSE on train data = %g" % lr_evaluator.evaluate(lr_predictions))

# In[34]:


lr_predictions = lr_model.transform(test_data)
lr_predictions.select("prediction","value_eur","features").show(5)
lr_evaluator = RegressionEvaluator(predictionCol="prediction",                 labelCol="value_eur",metricName="r2")
print("R Squared (R2) on test data = %g" % lr_evaluator.evaluate(lr_predictions))
```

Figure 15: *PySpark code for the GB model*

Generalized Linear Regression

Generalized Linear Regression (GLR) is a statistical framework that extends linear regression to handle non-normally distributed response variables. We chose GLR model as an experiment because it has the following advantages over the traditional linear regression model:

1. Greater applicability: GLR can handle more types of response variables, while LR can only handle normally distributed continuous response variables. GLR can be applied to non-normally distributed response variables such as binary, count, or skewed continuous data.
2. Better predictive power: Because GLR can more accurately capture the distributional characteristics of the response variable, it can sometimes have better predictive power than LR.
3. Better parameter estimation: GLR allows for modeling the response variable using different error distributions, which can result in more accurate parameter estimation. In contrast, assuming a normal distribution of the response variable in LR can lead to inaccurate parameter estimation.
4. Stronger robustness: GLR's error distribution is typically assumed to be strict, which can reduce reliance on assumptions about the data distribution and make it more robust.

Then, we used python to make exploratory data analysis on the distribution of the dependent variable (value_eur), and the results are shown in **Figure 16** below:

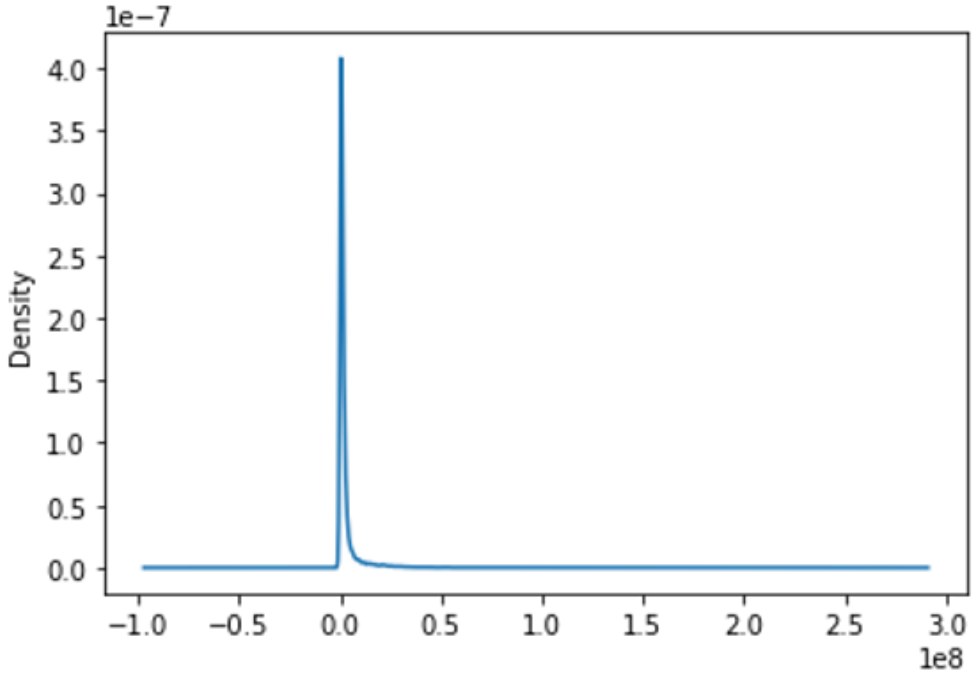


Figure 16: *Exploratory data analysis of the value_eur variable*

From the above **Figure 16**, we can see that since the data distribution of the dependent variable does not present a normal distribution feature, based on the above image and text analysis, we believe that the GLR model may achieve better results than the LR model.

Then we built a GLR model to analyze the data, here we used the following parameters and explained the meaning of the parameters in detail: family = "gaussian", link = "identity", maxIter = 10, regParam = 0.3.

family: This parameter specifies the probability distribution of the response variable. In this case, "gaussian" indicates that the response variable is normally distributed.
link: This parameter specifies the link function that relates the linear predictor to the mean of the response variable. In this case, "identity" indicates that the linear predictor is equal to the mean of the response variable.
maxIter: This parameter specifies the maximum number of iterations to run the optimization algorithm. In this case, the algorithm will run for a maximum of 10 iterations.
regParam: This parameter specifies the regularization parameter to use in the optimization algorithm. A higher value of regParam will result in stronger regularization. In this case, the value is set to 0.3.
After using the Pyspark code to complete the model establishment and run successfully, we randomly intercepted the data size on the GCP platform to 100%, 80%, 60%, 40%, 20%, and then submitted job according to the type of different nodes in the distributed computing system, so as to calculate the model RMSE and running time under different data volumes and different distributed computing systems, and drawed the following two charts:

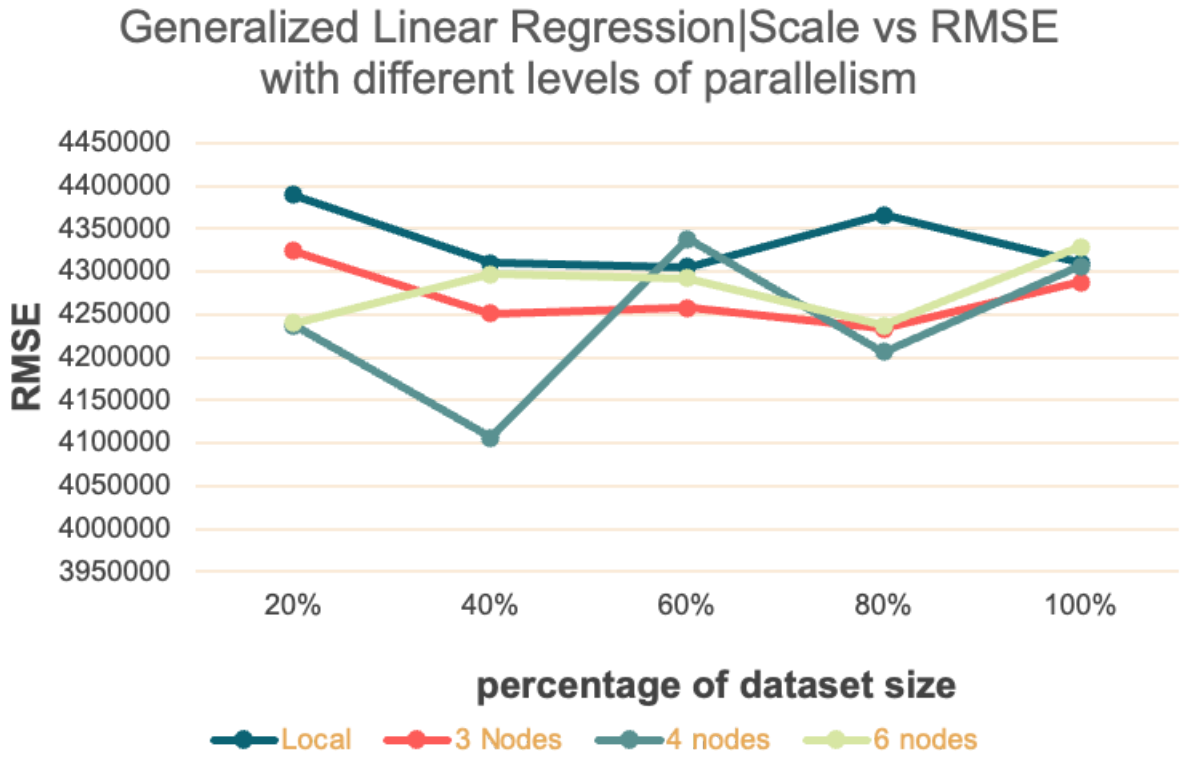


Figure 17: *Graph depicting GLR model performance, measuring RMSE*

From **Figure 17**, we can conclude 3 points:

1. The performance of the GLR model is not significantly improved compared to the linear regression model, and the range of RMSE is similar to that of LR.

2. Little RMSE difference between local running and distributed computing with different nodes when using 100% of the data.
3. The RMSE of distributed computing using 4 nodes has the best comprehensive effect.

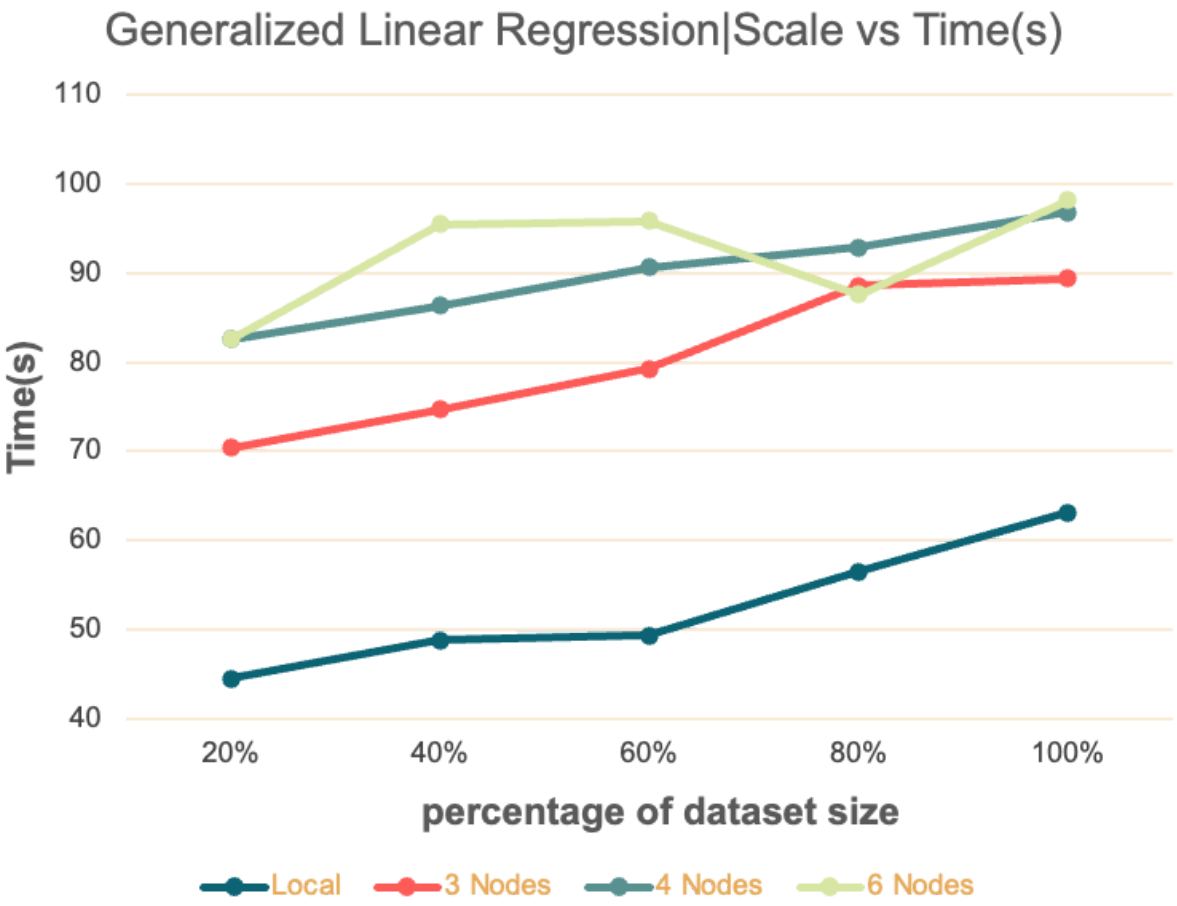


Figure 18: *Graphs depicting GLR model performance, measuring time*

From **Figure 18**, we can get some conclusions as follows:

1. Generally speaking, except for very few cases, the running time tends to increase as the amount of data being processed increases.
2. Overall, the running time is the shortest when using distributed computing with 4 nodes.

From the analysis of the RMSE of the model, it is not difficult to see that the GLR model does not significantly improve the performance. We guess that it may be because the parameters of the model do not perfectly match the data itself. Specifically, in the setting of the family parameter, we used Gaussian, but it is obviously from the previous analysis of the dependent variable that the dependent variable does not present a normal distribution, so the setting of the parameter itself does not have a good effect on improving the model. In order to verify our guess, we performed the data visualization analysis again with the logarithm of the dependent variable, and the results are as follows:

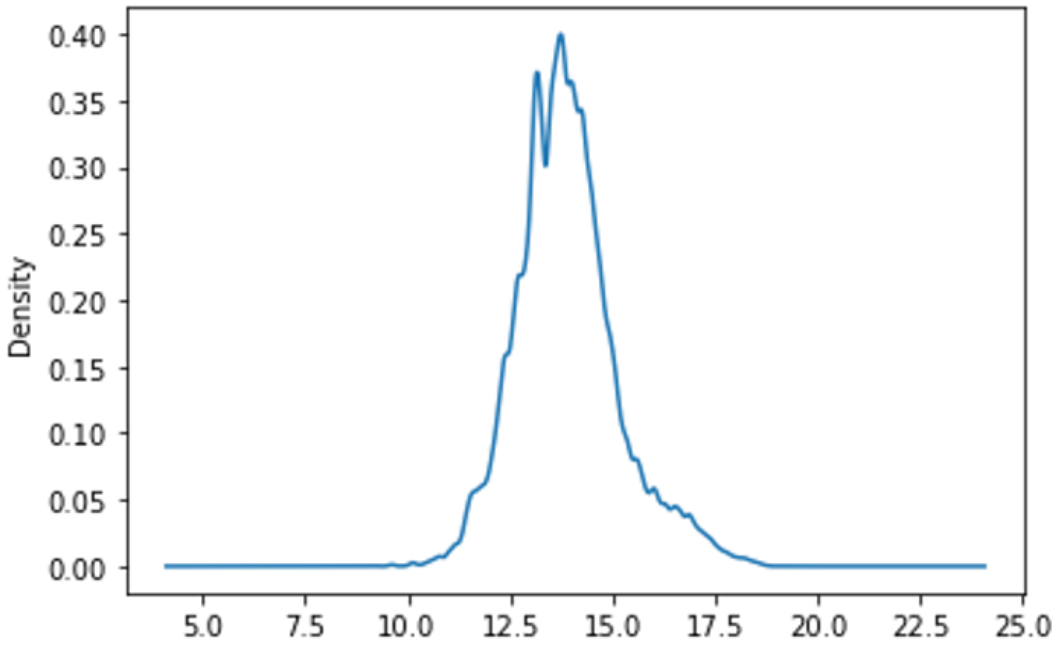


Figure 19: *Data visualization analysis of the logarithmic function of the dependent variable*

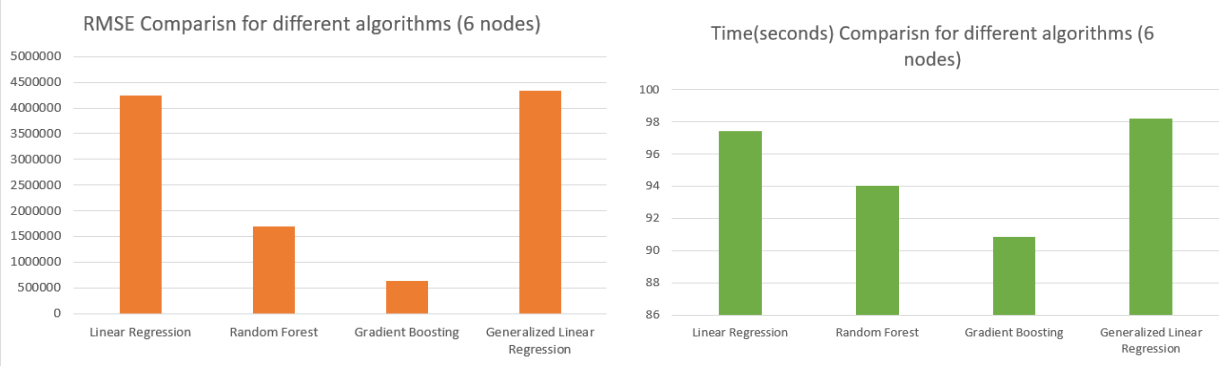
When we take the logarithm of the dependent variable and then analyze its distribution, we can easily see that the distribution of the logarithm of the dependent variable roughly presents a normal distribution. Therefore, if we want to improve the effect of the original GLR model, a necessary step is to set the dependent variable in the model to the logarithm of the original dependent variable.

```
glm = GeneralizedLinearRegression(featuresCol="features", labelCol="value_eur", family="gaussian", link="identity", maxIter=10, regParam=0.3)

# In[ ]:

glm_model = glm.fit(train_data)

# In[ ]:

glm_predictions = glm_model.transform(test_data)
glm_predictions.select("prediction","value_eur","features").show(5)

# In[ ]:

from pyspark.ml.evaluation import RegressionEvaluator
glm_evaluator = RegressionEvaluator(predictionCol="prediction",                 labelCol="value_eur",metricName="r2")
print("R Squared (R2) on test data = %g" % glm_evaluator.evaluate(glm_predictions))

# In[ ]:

from pyspark.ml.evaluation import RegressionEvaluator
evaluator = RegressionEvaluator(labelCol='value_eur', predictionCol='prediction', metricName='rmse')
rmse = evaluator.evaluate(glm_predictions)
print("Root Mean Squared Error (RMSE) on test data = %f" % rmse)
```

Figure 20: *PySpark code for the GLR model*

Comparison

We compare the four regression models in terms of accuracy and time for 100% of the data and at 6 nodes scaling.



Figures 21 & 22: *Bar plots comparing regression models time and RMSE*

We clearly observe that Gradient Boosting is the most accurate regression algorithm in terms of both RMSE and time. The next best algorithm is Random Forest regression, and we also observe that Linear Regression and Generalized Linear Regression have larger time and RMSE values, as well as having overall similar results.

Conclusion

By comparing the RMSE and Time for 4 different machine learning algorithms, we found that **Gradient Boosting** is the most accurate and fastest.

Reasons:

1. Gradient Boosting provides **high accuracy** in prediction as it minimizes the loss function during training.
2. It also provides the ability to **identify important features** in the dataset that are important for prediction.
3. Gradient Boosting also provides **high efficiency** in training large datasets as observed, this can also be because it can be **parallelised easily**.