`

Movie Box Office Revenue Prediction

Group – 16

Participants list

- **Rishik Sharma**
- **Atul Choudhari**
- **Umeshkrishna U**

Problem statement

The movie industry has grown massively over the past few years producing billions of dollars of revenue .A prediction model to evaluate the box office success of new movies can help the movie producers and directors make informed decisions .The goal of this project is to develop a model for predicting box office revenues based on public data for movies extracted from popular online movie websites. We plan to use the features like production budget, film genres, runtime, release date etc. from previously released movies to arrive at the conclusion

Overall Summary

1) Data collection: - Scrapped website (https://www.the-numbers.com) to acquire movie data using Beautiful Soup/selenium which are Python libraries
2) Data cleaning: **-** The data collected from the different pages were integrated and cleaned to remove incomplete and missing data.
3) Feature Selection and Modification: **-** Identified important features. Converted Categorical data, transformed data into appropriate data so that it can be used in regression.
4) Built Different Machine Learning models like
    (a) Multiple Linear regression
    (b) Regularization methods: - Lasso & Ridge Regression
    (c) Tree-Based Methods: - Decision tree regression, Bagging and Boosting
5) Optimized Hyperparameter of above models using randomized search and grid search wherever necessary
6) For each model calculated regression metrics like
    i) Root-Mean-Squared-Error (RMSE)
    ii) R^2 or Coefficient of Determination.
7) Finally, compared different methods to select the best model

`

Detailed Description And Analysis

- Dataset Acquisition

The data was collected primarily from different webpages of “The Numbers” https://www.the-numbers.com/movie/budgets website. We decided to select 20 features from the website. We first extracted 5 features (Release Date, Movie, Production Budget, Domestic Gross, Worldwide Gross) from each of the 100 movies listed on the site budgets page using selenium library of python. We then extracted the remaining features form movie details summary page of each movie Our final scrapped dataset consisted of 500 movies with 20 features each as follows:

Feature	Description
Movie	Name of the movie
Release Date	date in which the movie was released
Production Budget	Represents how much money was spent to produce a movie in US dollar (USD)
Domestic Gross	Revenue generated in the country (USD)
Worldwide Gross	Revenue generated globally (USD)
Running time	Total length of movie in minutes
Creative type	8 unique types on which movie is based on like Kids Fiction, Super Hero etc.
Opening	Earning in the opening weekend
legs	(Domestic Gross /biggest weekend)
Production method	4 unique types on which movie is produced like animation, Live Action etc.
Genre	10 different categories of movie like drama, adventure, horror etc.
Domestic Share	Domestic Gross / Worldwide Gross
Theater counts	No of theater movie was screened in
Infl. Adj. Dom. BO	inflation adjusted value
No of Lead Actors	Total lead actors in movie
Lead Actor	Name of lead actor in movie
Cast	Cast of the movie
Support Actors	Support actor name
No of Cast	Total cast in movie
No of Support Actors	Total Support Actors in movie

`

Data Visualization & Preparation

We import the data (.xlsx file) which was scrapped and check its shape

```
df=pd.read_excel("Selenium_df_Movie_Master_500.xlsx")
```

```
df.shape
```

```
(500, 20)
```

Fig 2. Importing the Data

We subsequently dropped the features 'Release Date', 'Movie' which we have not included in the model

```
# dropping the features not required in the model
df=df.drop(['Release Date', 'Movie'],axis=1)
```

```
df.shape
```

```
(500, 18)
```

Fig 3. Dropped features

We rename some feature names into more meaningful names for clear interpretability

```
df.rename(columns = {'Type':'Creative Type:'}, inplace = True)
df.rename(columns = {'Method':'Production Method'}, inplace = True)
df.rename(columns = {'Artist_Group_1':'Lead Actor'}, inplace = True)
df.rename(columns = {'Artist_Group_1_Count':'No of Lead Actors'}, inplace = True)
df.rename(columns = {'Artist_Group_2':'Cast'}, inplace = True)
df.rename(columns = {'Artist_Group_2_Count':'No of support  cast'}, inplace = True)
df.rename(columns = {'Artist_Group_3':'Support Actors'}, inplace = True)
df.rename(columns = {'Artist_Group_3_Count':'No of support  Actors'}, inplace = True)
```

Fig 4. Rename Columns

`

Handling Missing Values

We checked for Null values in the dataset using df.isnull().sum() and heatmap

```
1 df.isnull().sum()
```

```
Production Budget         0
Domestic Gross            0
Worldwide Gross           0
Opening                   9
Legs                      8
Domestic Share            8
Theater counts            9
Infl. Adj. Dom. BO        7
Running Time             16
Genre                     2
Production Method         1
Creative Type:            2
Lead Actor                0
No of Lead Actors         0
Cast                      3
No of support  cast       3
Support Actors           20
No of support  Actors     20
dtype: int64
```

Fig 5 Checking Null values

```
1 sns.heatmap(df.isnull(), cbar = False)
2 plt.show()
```

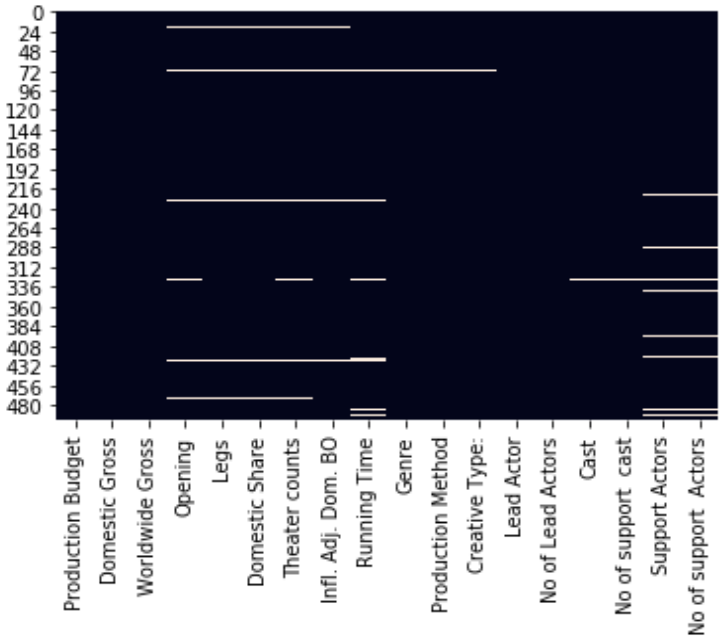


Fig 6. Heatmap to check missing values

We see that there are missing values in features such as Opening Legs, Domestic Share, Theater counts, Infl. Adj. Dom. BO, Running Time, Genre, Production Method, Creative Type, Cast, No of support cast, No

`

of support cast and No of support Actors. As the total rows with missing values were less, we decided to drop the missing values

```
: 1 df.isnull().sum()

: Production Budget         0
  Domestic Gross            0
  Worldwide Gross           0
  Opening                   0
  Legs                      0
  Domestic Share            0
  Theater counts            0
  Infl. Adj. Dom. BO        0
  Running Time              0
  Genre                     0
  Production Method         0
  Creative Type:            0
  Lead Actor                0
  No of Lead Actors         0
  Cast                      0
  No of support  cast       0
  Support Actors            0
  No of support  Actors     0
  dtype: int64

: 1 df.shape

: (464, 18)
```

Fig 7. Dropped missing values

`

Checking correlation

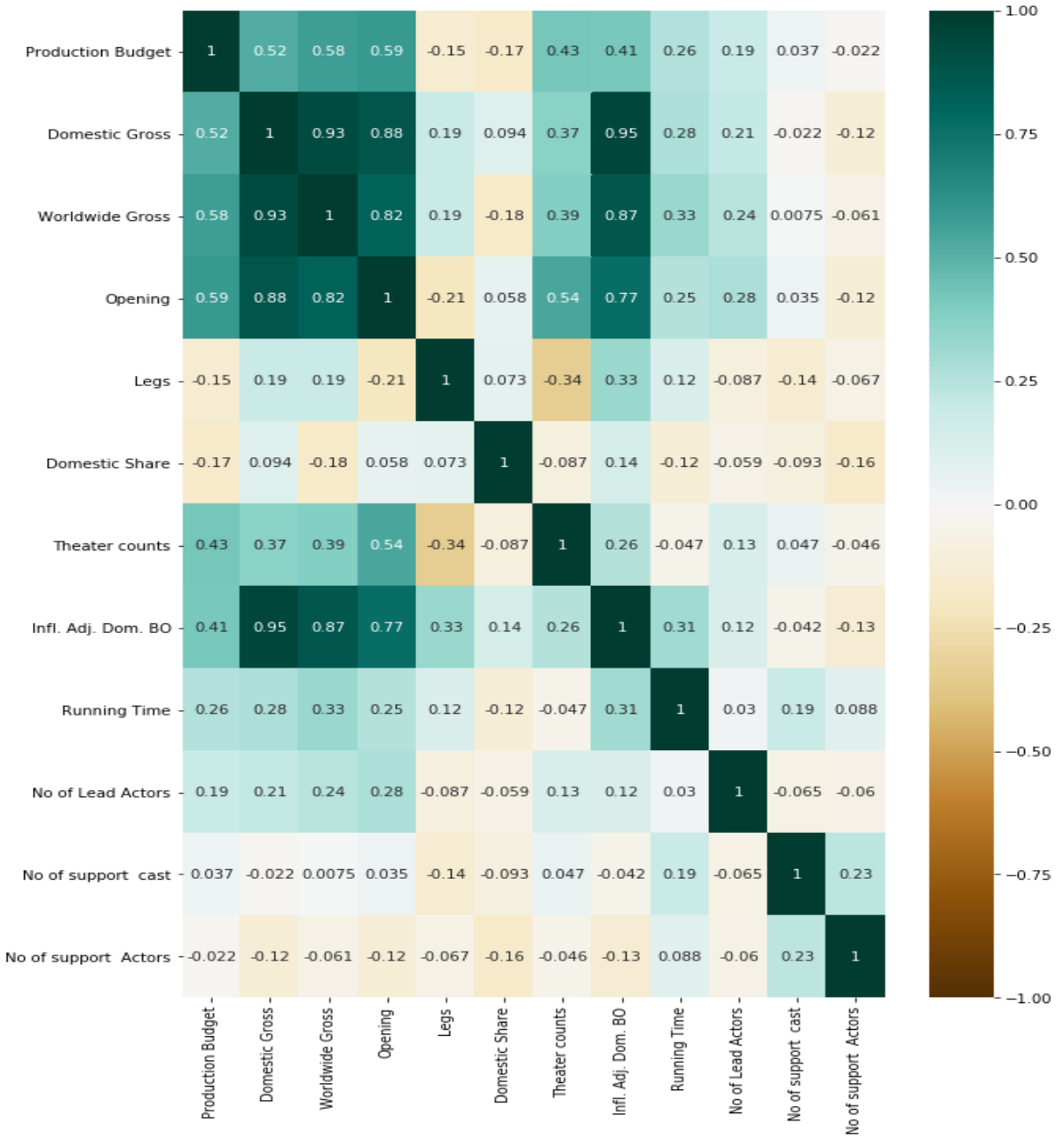


Fig 8 Heatmap for correlation

We clearly see

- Strong positive correlation of our target variable Worldwide gross with Opening, Domestic gross &. infl. Adj Dom BO
- Moderate positive correlation of our target variable Worldwide gross with Production Budget
- There may be problems of multi-collinearity as features like Opening, Domestic gross &. infl. Adj Dom BO have Strong positive correlation amongst themselves

`

Checking Distribution

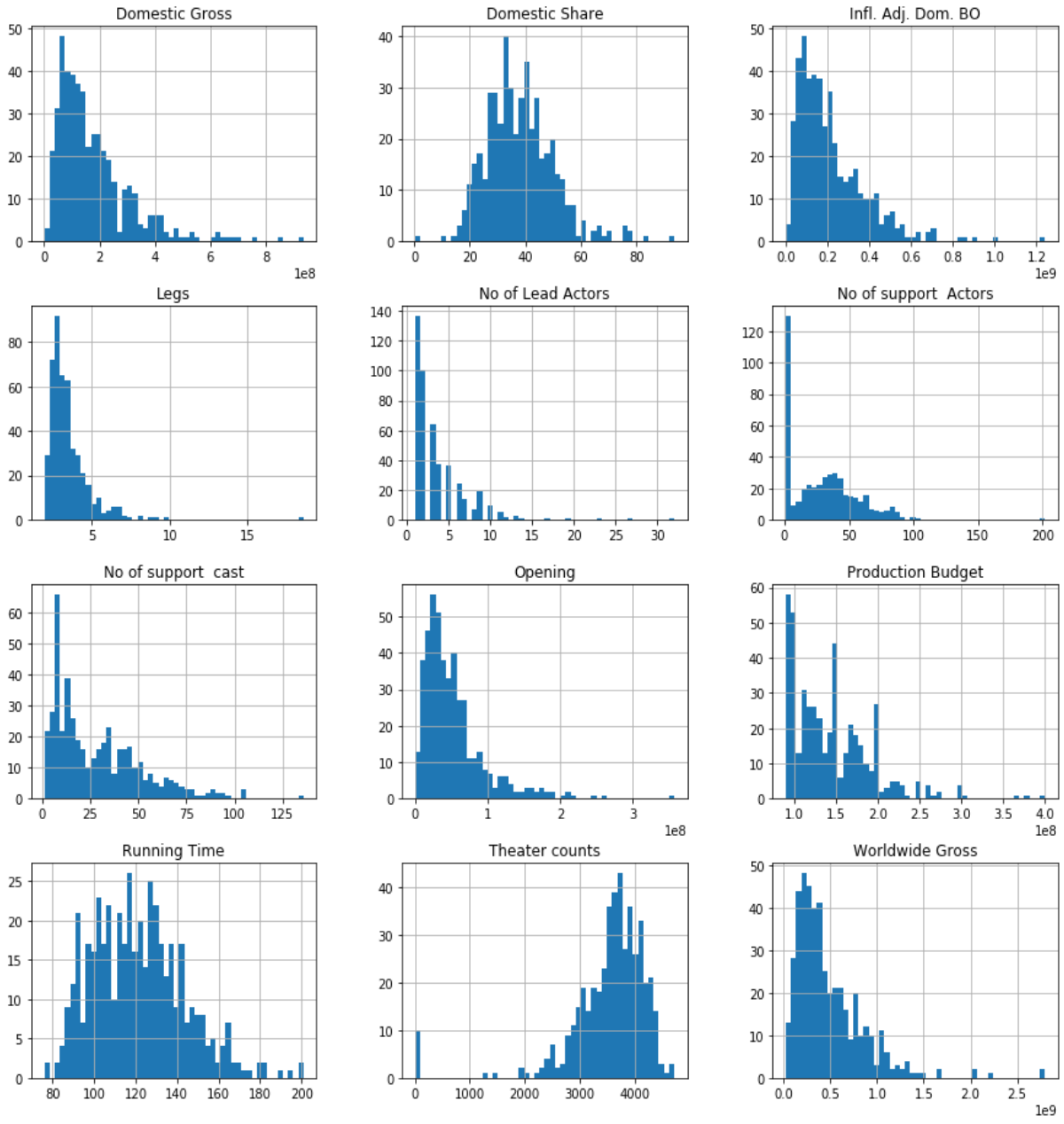


Fig 9. Histogram & Density plot

We can observe

1. Positive skewness from the histogram plot for Our target variable worldwide gross
2. Theater counts is negatively skewed.
3. Running time is approximately normal skewed
4. We may have to apply Transformation to the data

`

Outlier Analysis

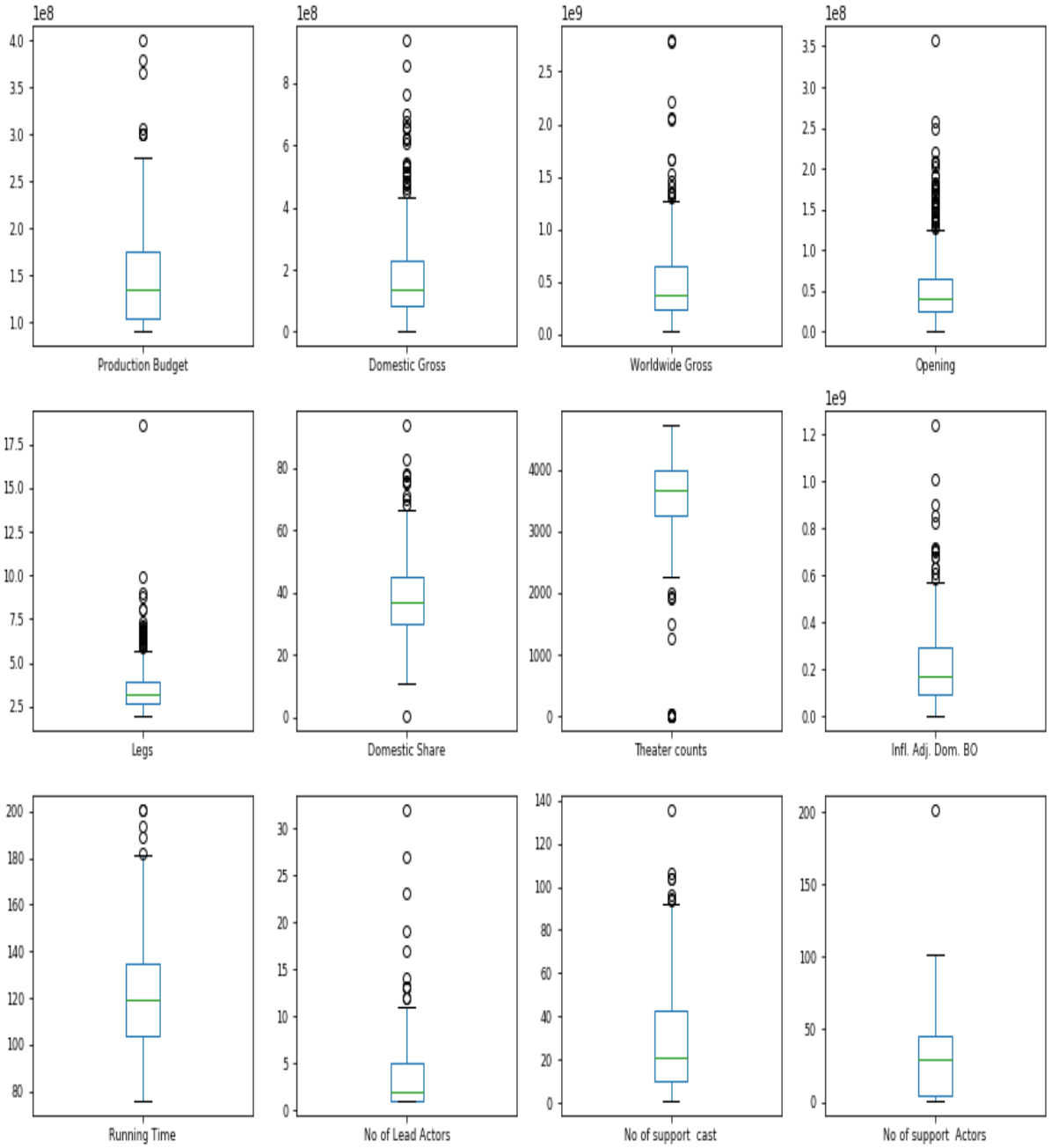
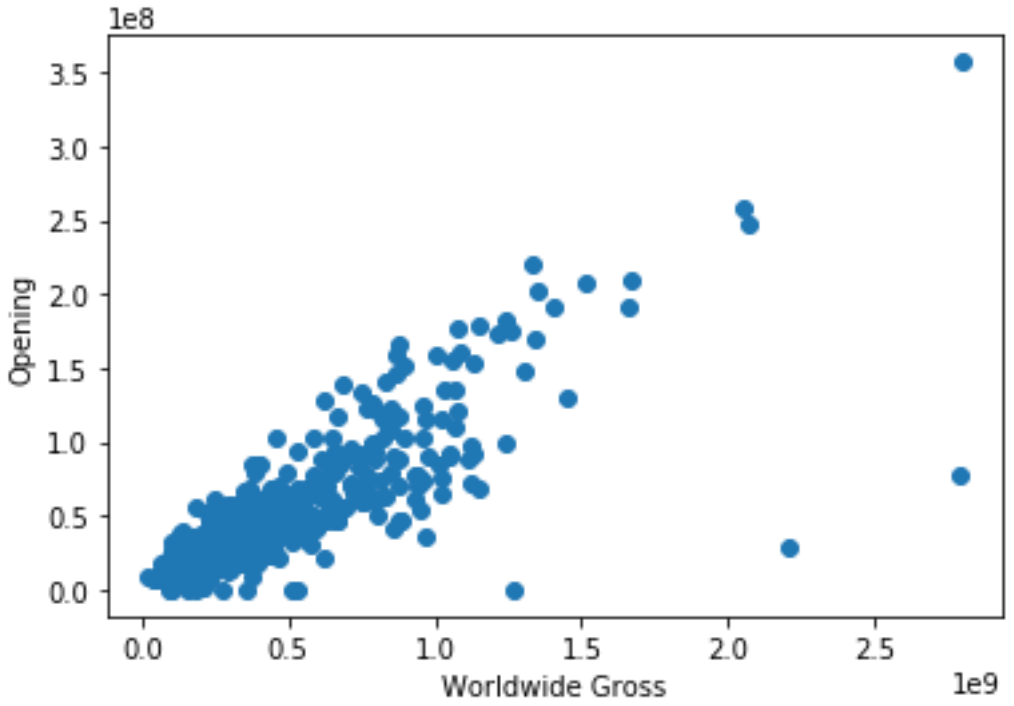


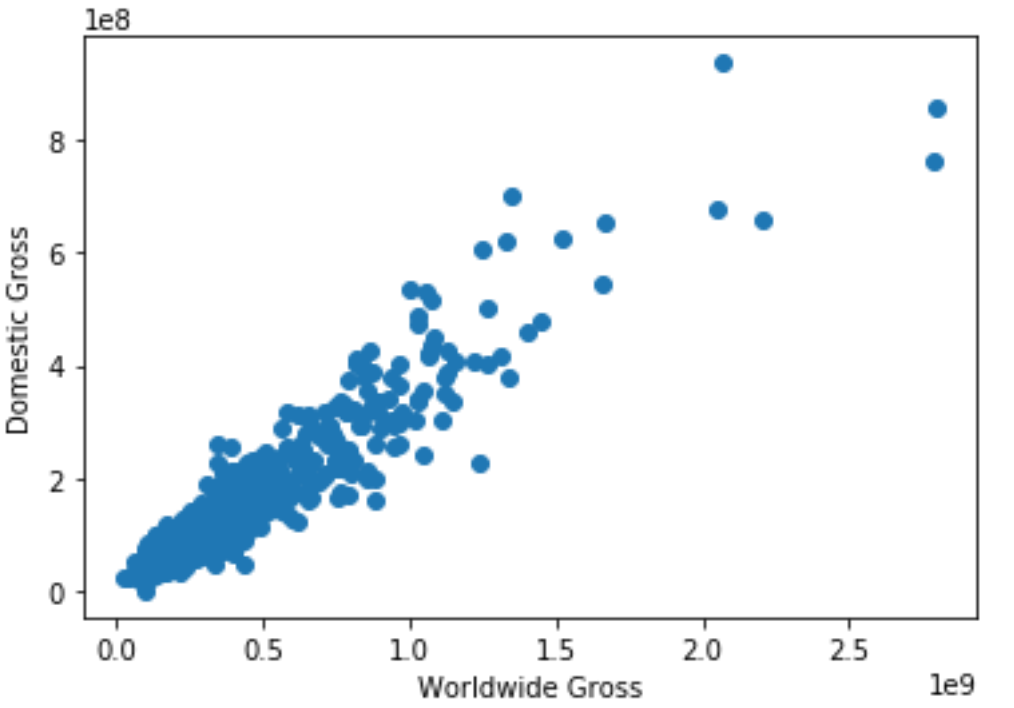
Fig 10. Box Plot

We see outlier in all features. They are data-points that are extreme to normal observations. Many machine learning models, like linear & logistic regression, are easily impacted by the outliers in the training data. Instead of linear models, we can use tree-based methods which are less impacted by outliers. Tree based methods are robust to outliers.
We have to treat the outliers during the Linear Regression model

`



Worldwide Gross vs Opening



Worldwide Gross vs Domestic Gross

Fig 11.Scatter Plots

We see that Opening Value greater than 300000000,Domestic Gross greater than 800000000 are extreme values with respective to target variable

`

Analyzing Categorical Values

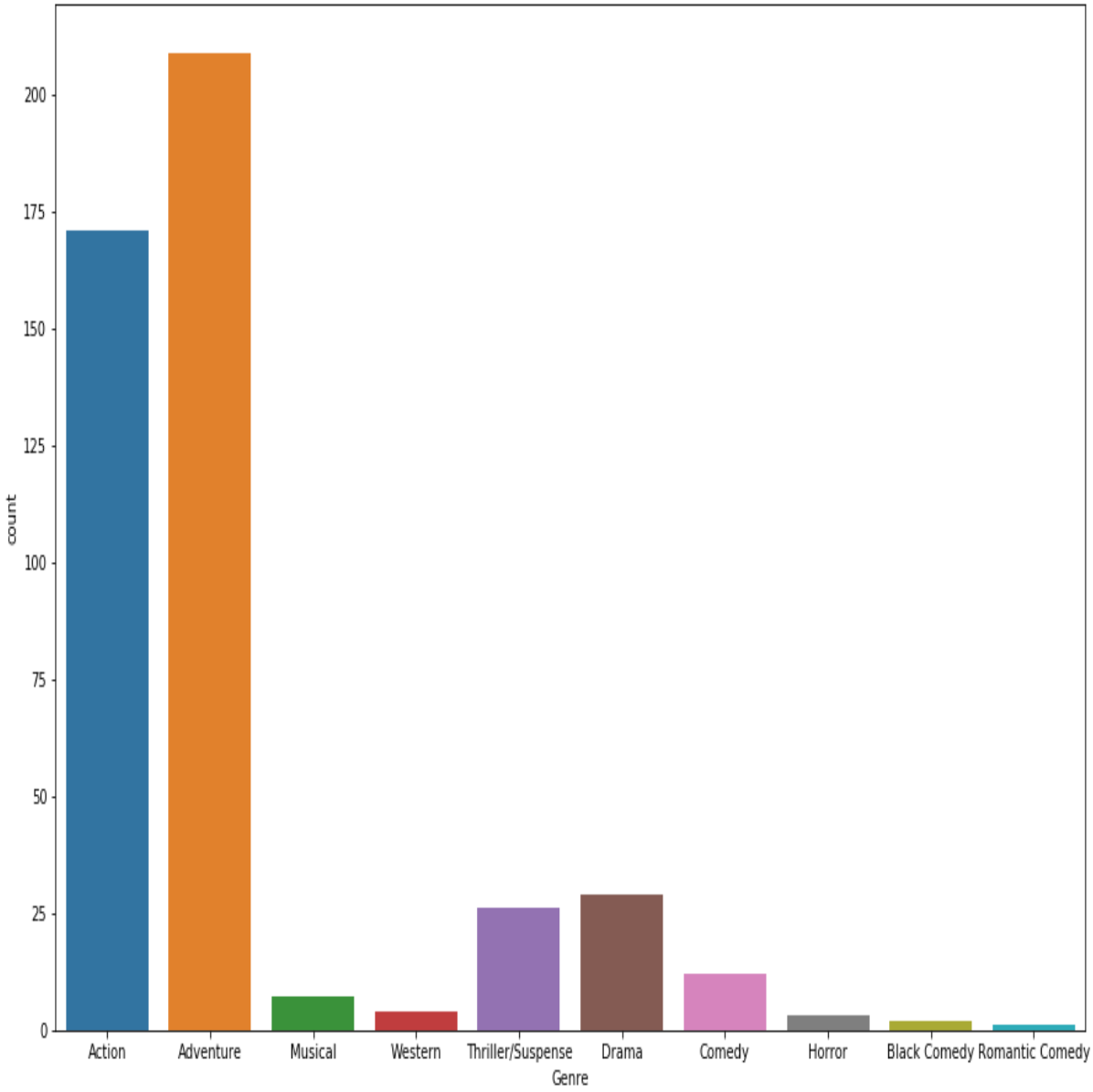


Fig 12 Countplot for genre

we see that

- There are 10 unique genres
- Most of the movies (more than 50% of data) have genre as Action (more than 160) or Adventure (more than 200)

`

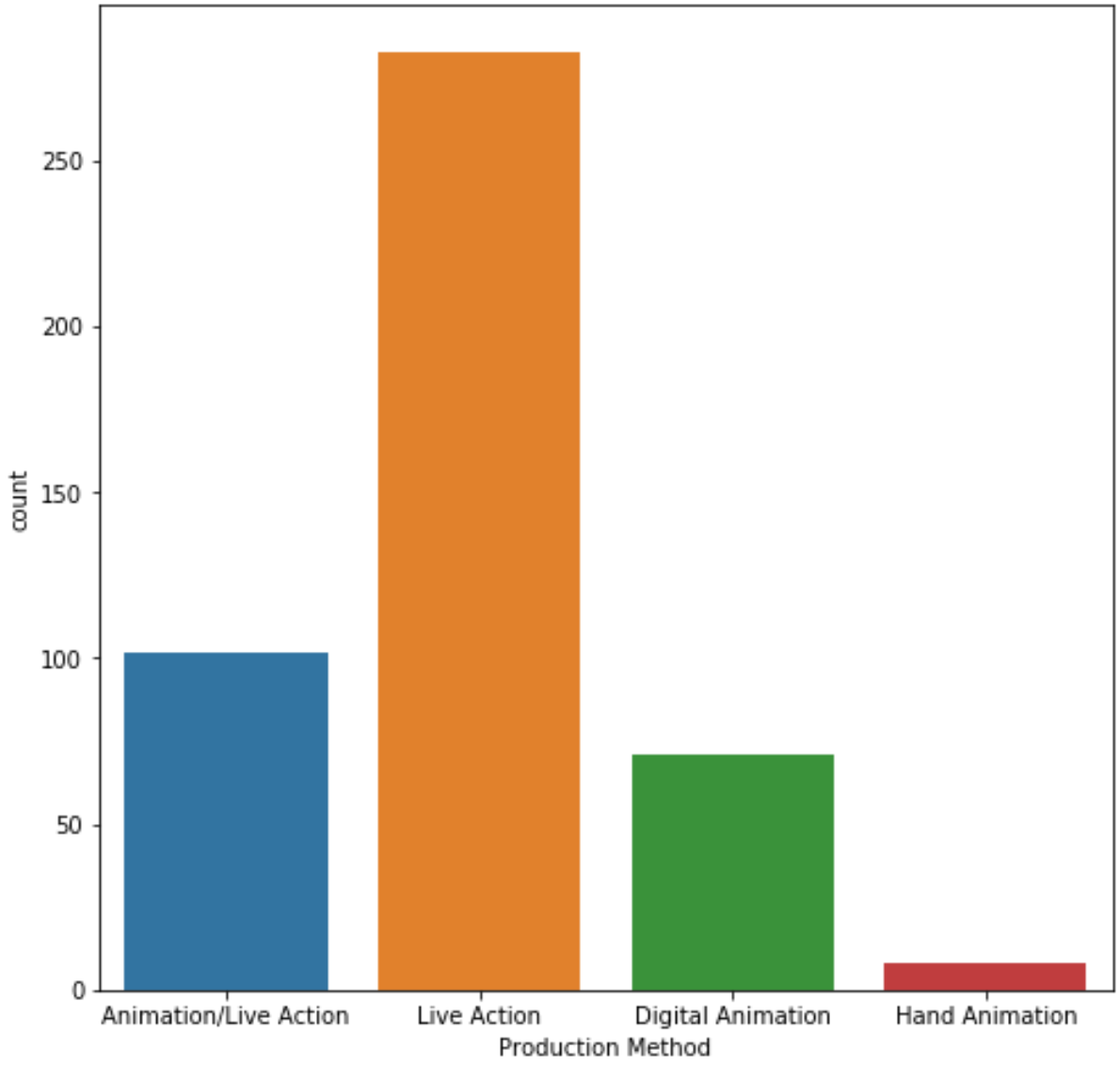


Fig 13 Countplot for Production Method

We see that

- There are 4 unique Production Methods
- Most of the movies (approximately 300 more than 50% of data) have production method as Live Action

`

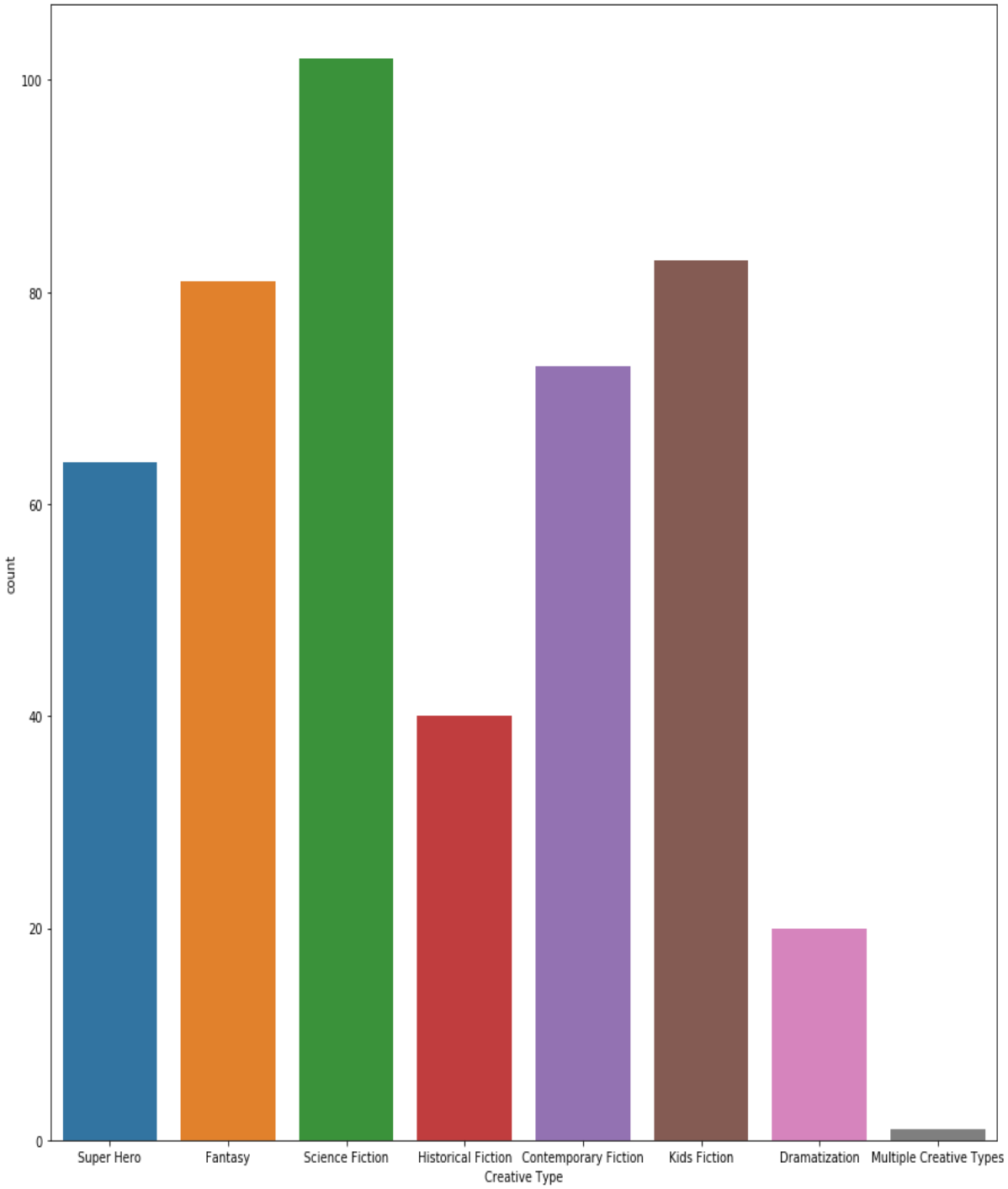


Fig 14 Countplot for Production Method

We see that

- There are 8 unique Creative types
- Creative Type Science Fiction has the highest movies

`

We see that there are 231 unique Lead actors

```
1 df["Lead Actor"].value_counts()
```

```
Johnny Depp           15
Tom Cruise            12
Will Smith            11
Leonardo DiCaprio     10
Hugh Jackman           9
                      ..
Pete Ploszek           1
Ed Asner*              1
Jim Sturgess           1
Jodie Foster           1
Ty Burrell             1
Name: Lead Actor, Length: 231, dtype: int64
```

Fig.15 Unique lead Actors

We see that there are 379 unique casts for the movies

```
1 df["Cast"].value_counts()
```

```
Ian McKellen          5
John Turturro         4
Halle Berry           4
Joan Cusack           3
Stellan Skarsgård     3
                     ..
Ricky Gervais         1
Lynn Collins          1
Kumail Nanjiani       1
Cobie Smulders        1
C.J. Adams            1
Name: Cast, Length: 379, dtype: int64
```

Fig.16 Unique Casts

We see that there are 304 unique Support Actors for the movies

```
1 df["Support Actors"].value_counts()
```

```
Stan Lee           25
Ridley Scott        8
Peter Jackson       7
David Yates         6
Michael Bay         5
                   ..
Chris Ellis         1
Byron Howard        1
Hayden McFarland    1
Cameron Diaz        1
Brad Pitt           1
Name: Support Actors, Length: 304, dtype: int64
```

Fig.16 Unique Support Actors

`

The get_dummies() function is used to convert categorical variable into dummy/indicator variables.

```
]: 1 df1=pd.get_dummies(df,drop_first=True)

]: 1 df1.tail()
```

]:

	Production Budget	Domestic Gross	Worldwide Gross	Opening	Legs	Domestic Share	Theater counts	Infl. Adj. Dom. BO	Running Time	No of Lead Actors	...	Support Actors_Walt Becker	Support Actors_Walt Dohrn	Support Actors_Walter F. Parkes	Act
492	90000000	62362560	143049560	21689125.0	2.88	43.6	2770.0	100337294.0	127.0	2	...	0	0	0	
495	90000000	58250803	273477501	23633317.0	2.46	21.3	3854.0	58250807.0	118.0	1	...	0	0	0	
496	90000000	58220776	87420776	8234926.0	7.03	66.6	2518.0	102488283.0	131.0	1	...	0	0	0	
497	90000000	51877963	203653524	15193907.0	3.41	25.5	3086.0	76104384.0	126.0	3	...	0	0	0	
499	90000000	42885593	140900000	14274503.0	3.00	30.4	2504.0	85117153.0	107.0	2	...	0	0	0	

5 rows × 942 columns

Fig 17 Converting Categorical Variables

As are data had lot of outliers and also the values are quite high, we decide to apply RobustScaler on the above data. Then we use the train_test_split function on the above converted data to split the dataset for two different purposes: training and testing. The testing subset is for building the model. The testing subset is for using the model on unknown data to evaluate the performance of the model**.**

```
from sklearn.preprocessing import RobustScaler
scaler = RobustScaler()
robust_scaled_df = scaler.fit_transform(df1)
robust_scaled_df = pd.DataFrame(robust_scaled_df, columns=df1.columns)
robust_scaled_df
```

```
X=robust_scaled_df.drop(["Worldwide Gross"],axis=1)
y=robust_scaled_df["Worldwide Gross"]
from sklearn.model_selection import train_test_split
seed = 10
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state = seed)
```

Fig 18 Splitting data into test-train sets

`

- **Model Building & optimization**

In data exploration we have observed that our target variable depends on multiple factors. We have outliers in the data. As we know tree-based methods are robust to outliers we have built and compared following Machine Learning models to evaluate the best models for our dataset.

1. Decision-Tree Regression
2. Bagged Decision Trees
3. Bagged KNN regression
4. Gradient Boosting regression
5. Ridge Regression
6. Lasso Regression
7. Multiple Linear Regression

1) Decision-Tree Regression

Regression trees are used when the dependent variable is continuous. For regression trees, the value of terminal nodes is the mean of the observations falling in that region. Therefore, if an unseen data point falls in that region, we predict using the mean value. Decision tree algorithms can be applied to both regression and classification tasks; in this project we have done a simple regression implementation using Python and scikit-learn.

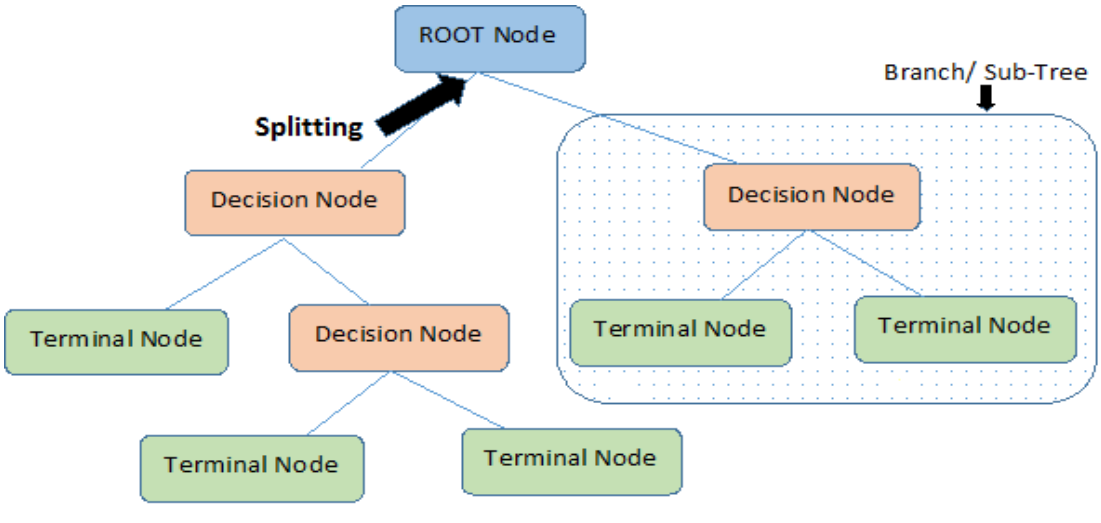


Fig 19 Basic Decision Tree structure
(source:- https://www.kdnuggets.com/)

`

We import the required libraries and use GridSearchCV function of python to tune the various hyperparameters of the model. Hyperparameter tuning is searching the hyperparameter space for a set of values that will optimize your model architecture. We set parameter CV=10 which does 10-fold cross validation.

```
# Applying GridSearch-cross validation to get best model paramaters
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import GridSearchCV
param_grid = {"criterion": ["mse", "mae"],
              "max_depth": [8,15,20,50],
              "max_leaf_nodes": [40,50,60],
              "min_samples_leaf": [10,15,20],
              "min_samples_split": [10,15,20],
              }
Regressor = GridSearchCV(DecisionTreeRegressor(),param_grid, cv=10,n_jobs=-1)
Regressor.fit(X_train,y_train)
```

Decision Tree Parameters

criterion: This parameter determines how the impurity of a split will be measured.

max_depth: This determines the maximum depth of the tree. The default value is set to none. This will often result in over-fitted decision trees. The depth parameter is one of the ways in which we can regularize the tree, or limit the way it grows to prevent **over-fitting**.

min_samples_split: The minimum number of samples a node must contain in order to consider splitting. The default value is two.

min_samples_leaf: The minimum number of samples needed to be considered a leaf node. The default value is set to one. Use this parameter to limit the growth of the tree

We fit the tree with the following best parameters as obtained by GridSearchCV.

```
Regressor.best_params_
```

```
{'criterion': 'mse',
 'max_depth': 8,
 'max_leaf_nodes': 40,
 'min_samples_leaf': 10,
 'min_samples_split': 10}
```

```
DT_model=Regressor.best_estimator_
DT_model.fit(X_train, y_train)
```

```
DecisionTreeRegressor(ccp_alpha=0.0, criterion='mse', max_depth=8,
                      max_features=None, max_leaf_nodes=40,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=10, min_samples_split=10,
                      min_weight_fraction_leaf=0.0, presort='deprecated',
                      random_state=None, splitter='best')
```

`

We predict on train and test set as follows

```
y_train_pred = DT_model.predict(X_train)
y_test_pred = DT_model.predict(X_test)
from sklearn.metrics import r2_score, mean_squared_error
#print(LR_model1.score(X_train, y_train))
print ("R-squared Training",r2_score(y_train, y_train_pred))
print ("R-squared Testing",r2_score(y_test, y_test_pred))
print ("Train RMSE : ",np.sqrt(mean_squared_error(y_train, y_train_pred)))
print ("Test RMSE : ",np.sqrt(mean_squared_error(y_test, y_test_pred)))
```

Decision Tree Regression Results

R^2 Training	R^2 Testing	Train RMSE	Test RMSE
0.9302	0.8669	0.2436	0.2703

R-squared

R-squared is a statistical measure of how close the data are to the fitted regression line. It is also known as the coefficient of determination

R-squared = Explained variation / Total variation

R-squared is always between 0 and 1

- 0 indicates that the model explains none of the variability of the response data around its mean.
- 1 indicates that the model explains all the variability of the response data around its mean.

We have R^2 Training of 0.93 and R^2 Testing of 0.8669 which is signifies we have a good regression model

Root Mean Squared Error (RMSE)

RMSE = $\sqrt{}$mean(pred-y)^2 where pred – y is the error or residuals

As both the RMSE values test & test are close we say that there is no problem of overfitting.

`

Graph of Actual vs Predicated

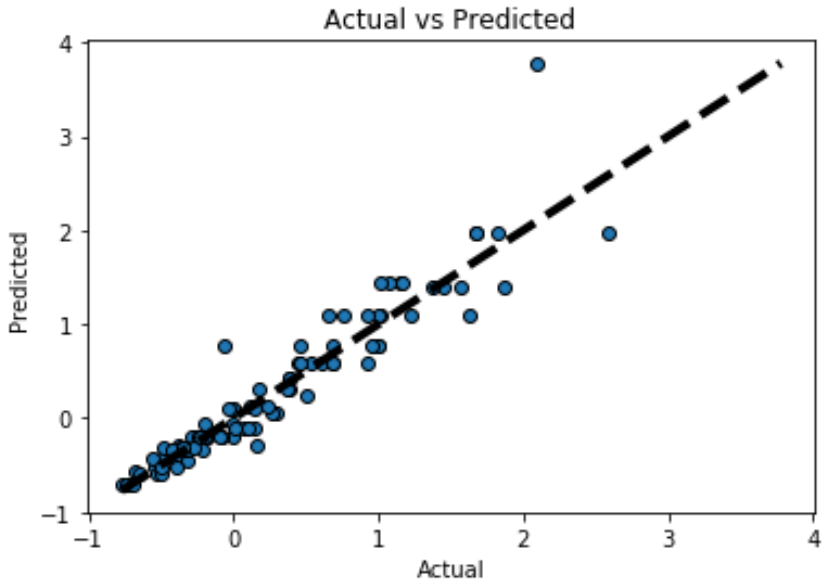


Fig 20. Actual vs Predicated

We see that most of the predicted values are closer to the actual values. Model is a good fit to our data.

Using residual plots, we assess whether the observed error (residuals) is consistent with error.

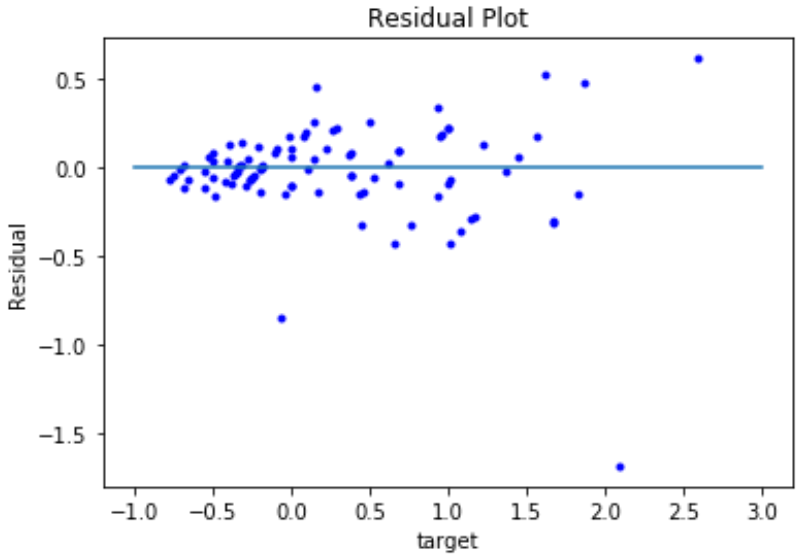


Fig 21. Residual Plot

The residuals are centered on zero throughout the range of fitted values.

`

Bagged Decision Trees

Many times, the single model does not perform well alone. In Ensemble learning multiple models (“weak learners”) are trained and combined to get improved results. Bagging, that often considers weak learners, learns them independently from each other in parallel and combines them following some kind of averaging process

We use binning for better sampling before applying bagging.

```
#bin target variable for better sampling
bins = 50*np.arange(8)
binned_y = np.digitize(y, bins)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,stratify=binned_y)
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,stratify=binned_y)
```

We import the required libraries and define weak learner as DecisionTreeRegressor.

```
from sklearn.ensemble import BaggingRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import GridSearchCV
```

```
single_estimator = DecisionTreeRegressor()
ensemble_estimator = BaggingRegressor(base_estimator = single_estimator)
```

We use GridSearchCV with 10 FoldCross Validation to tune parameters

```
param_grid = param_grid = {'oob_score' : [True, False],'n_estimators':[100,150],'base_estimator__max_depth':[3,5],
                          "base_estimator__min_samples_leaf": [2,5],"base_estimator__min_samples_split":[2,4],}
random_bag = GridSearchCV(ensemble_estimator,param_grid, cv=10,n_jobs=-1)
random_bag.fit(X_train,y_train)
```

We use the best parameters to fit the model

```
random_bag.best_params_
```

```
{'base_estimator__max_depth': 5,
 'base_estimator__min_samples_leaf': 2,
 'base_estimator__min_samples_split': 2,
 'n_estimators': 100,
 'oob_score': False}
```

```
DTbagging_model_1 = random_bag.best_estimator_
DTbagging_model_1.fit(X_train, y_train)
```

`

We fit the model on the train and test data

```
y_train_pred = DTbagging_model_1.predict(X_train)
y_pred = DTbagging_model_1.predict(X_test)
from sklearn.metrics import r2_score, mean_squared_error
print ("R-squared Training",r2_score(y_train, y_train_pred))
print ("R-squared Testing",r2_score(y_test, y_pred))
print ("Train RMSE : ",np.sqrt(mean_squared_error(y_train, y_train_pred)))
print ("Test RMSE  : ",np.sqrt(mean_squared_error(y_test, y_pred)))
```

Bagging Decision Tree Regression Results

R^2 Training	R^2 Testing	Train RMSE	Test RMSE
0.9787	0.8969	0.1303	0.2773

We see that bagging decision Tree have given better results than the standalone Tree regressor. We also see smaller & closer RMSE values and hence no overfitting

Graph of Actual vs Predicated

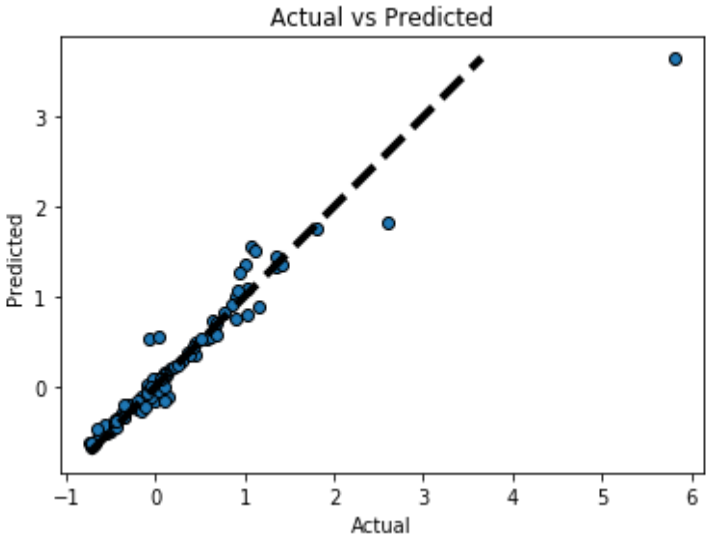


Fig 22. Actual vs Predicated

We see that most of the predicted values are closer to the actual values. Model is a good fit to our data. Using residual plots, we assess whether the observed error (residuals) is consistent with error.

`

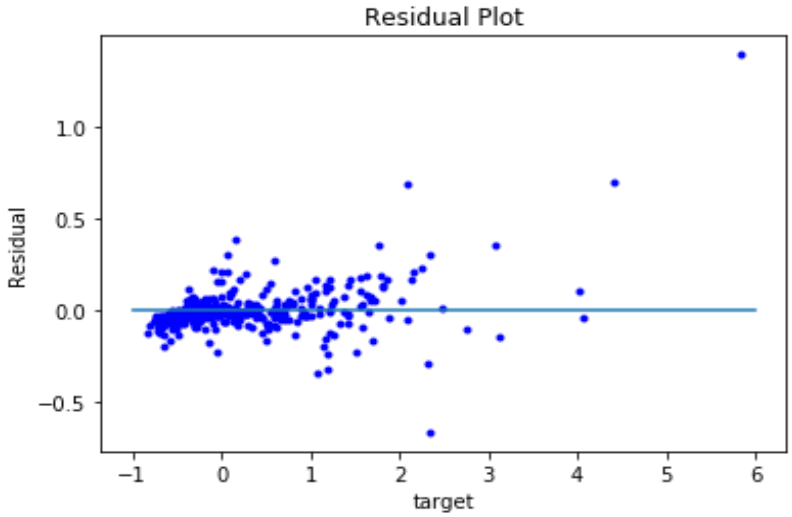


Fig 23. Residual Plot

The residuals are centered on zero throughout the range of fitted values.

Bagging KNeighborsRegressor

It is Similar to the above model we have created but we use KNeighborsRegressor as weak model in this case

```
from sklearn.ensemble import BaggingRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import GridSearchCV
```

```
single_estimator = KNeighborsRegressor()
ensemble_estimator = BaggingRegressor(base_estimator = single_estimator)
```

```
param_grid = {'oob_score' : [True, False],
              'n_estimators': [100,150],
              'base_estimator__n_neighbors':[3,7,9]
              }
random_bag = GridSearchCV(ensemble_estimator,param_grid, cv=10,n_jobs=-1)
random_bag.fit(X_train,y_train)
```

We train the model on the best parameter

```
random_bag.best_params_
```

```
{'base_estimator__n_neighbors': 3, 'n_estimators': 100, 'oob_score': True}
```

```
bagging_model_1 = random_bag.best_estimator_
bagging_model_1.fit(X_train, y_train)
```

`

Testing the model

```
:
1 y_train_pred = bagging_model_1.predict(X_train)
2 y_pred = bagging_model_1.predict(X_test)
3 from sklearn.metrics import r2_score, mean_squared_error
4 print ("R-squared Training",r2_score(y_train, y_train_pred))
5 print ("R-squared Testing",r2_score(y_test, y_pred))
6 print ("Train RMSE : ",np.sqrt(mean_squared_error(y_train, y_train_pred)))
7 print ("Test RMSE  : ",np.sqrt(mean_squared_error(y_test, y_pred)))
```

Bagging KNeighborsRegressor Results

R^2 Training	R^2 Testing	Train RMSE	Test RMSE
0.8934	0.9191	0.2925	0.2443

We get R^2 of 0.9191 which is more than the bagged Decision tree. The RMSE scores are closer and we see to overfitting.

Graph of Actual vs Predicated

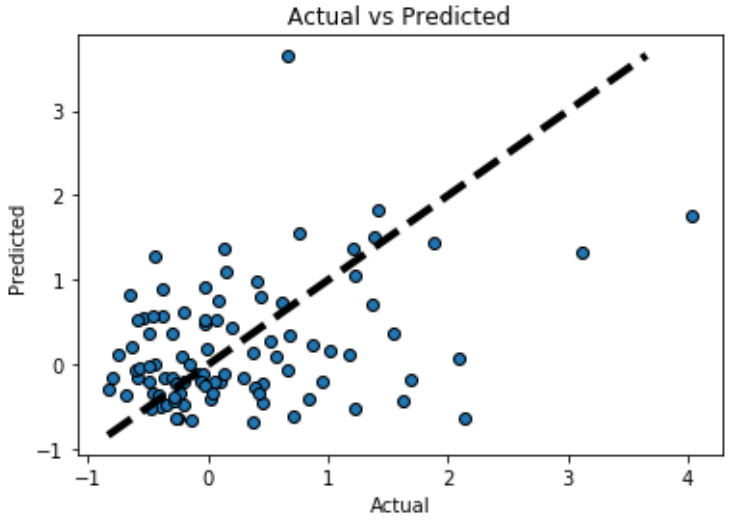


Fig 24. Actual vs Predicated

We see that most of the predicted values are closer to the actual values. Model is a good fit to our data.

Using residual plots, we assess whether the observed error (residuals) is consistent with error.

`

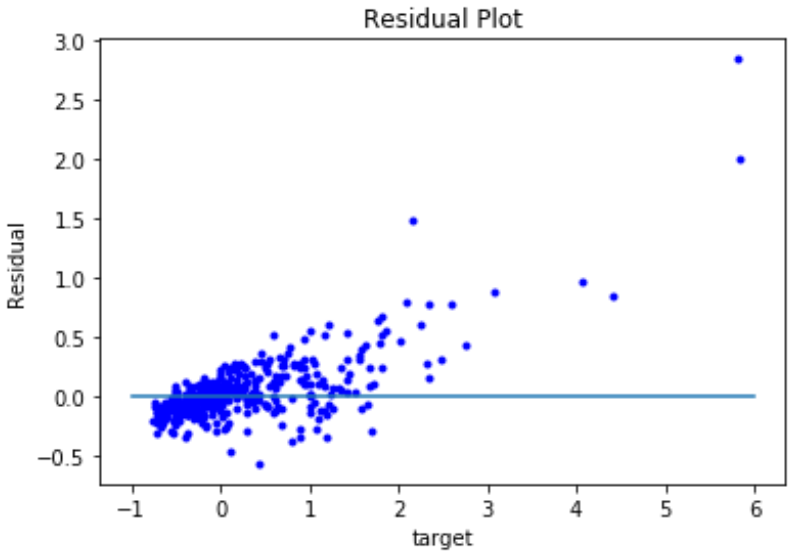


Fig 25. Residual Plot

The residuals are centered on zero throughout the range of fitted values. We see some extreme residuals for this model

Gradient Boosting regression

Boosting ensemble technique which considers weak learners, learns them sequentially in a very adaptative way (base model depends on the previous ones) and combines them following a deterministic strategy. gradient boosting combines weak "learners" into a single strong learner in an iterative fashion. Decision trees are used as the weak learner in gradient boosting

The parameters that can be tuned are similar to the decision tree regressor explained above but in addition we need to specify

- n_estimators:- Number of trees to be added
- 'learning rate' is a tuning parameter in an optimization algorithm that determines the step size at each iteration while moving toward a minimum of a loss function
- There is a trade-off between learning rate and n_estimators.

We import the required libraries and use GridSearchCV as used in above models to find the best parameters

`

```
bins = 50*np.arange(10)
binned_y = np.digitize(y,bins,right=False)
```

```
from sklearn.model_selection import train_test_split
seed=10
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=seed,stratify=binned_y)
```

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import GradientBoostingRegressor
param_grid =  {
              'max_depth':[3,5],
              'min_samples_split':[21,25],
              'min_samples_leaf':[21,25],
              'n_estimators':[100],
              'learning_rate':[0.1,0.3,1],
               'loss':['ls','huber']
              }
Boosted_Regressor = GridSearchCV(GradientBoostingRegressor(warm_start=True),
                                 param_grid,cv=10,n_jobs=-1)
Boosted_Regressor.fit(X_train,y_train)
```

We fit the model based on the best parameters obtained and predict for the test data

```
Boosted_Regressor.best_params_
```

```
{'learning_rate': 0.3,
 'loss': 'huber',
 'max_depth': 3,
 'min_samples_leaf': 21,
 'min_samples_split': 21,
 'n_estimators': 100}
```

```
GradientBoosting_model_1=Boosted_Regressor.best_estimator_.fit(X_train,y_train)
```

```
y_train_pred = GradientBoosting_model_1.predict(X_train)
y_test_pred = GradientBoosting_model_1.predict(X_test)
from sklearn.metrics import r2_score, mean_squared_error
print ("R-squared Training",r2_score(y_train, y_train_pred))
print ("R-squared Testing",r2_score(y_test, y_test_pred))
print ("Train RMSE : ",np.sqrt(mean_squared_error(y_train, y_train_pred)))
print ("Test RMSE  : ",np.sqrt(mean_squared_error(y_test, y_test_pred)))
```

Gradient Boosting regression Results

R^2 Training	R^2 Testing	Train RMSE	Test RMSE
0.9327	0.7732	0.2136	0.5250

We get R^2 of 0.7732 on testing data which is less than the bagging with Decision tree Regressor and Bagged KNN regressor. The R^2 Training of the model can be increased but it overfits the data.

`

Graph of Actual vs Predicated

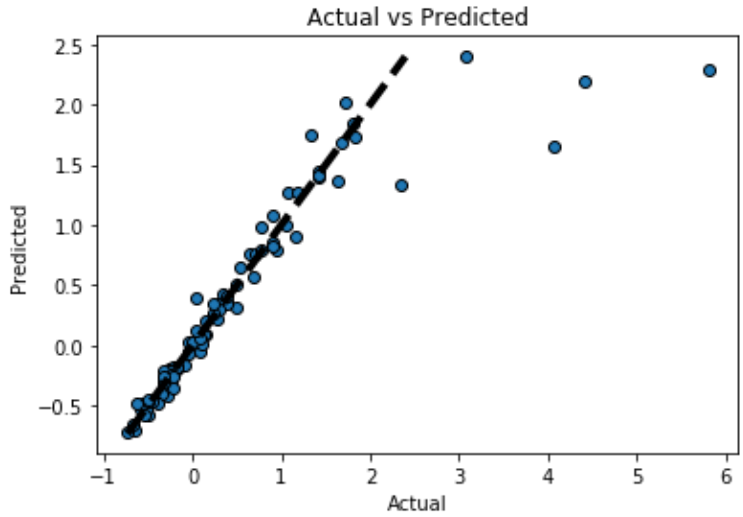


Fig 26. Actual vs Predicated

We see that most of the predicted values are closer to the actual values. Model is a good fit to our data.

Using residual plots, we assess whether the observed error (residuals) is consistent with error.

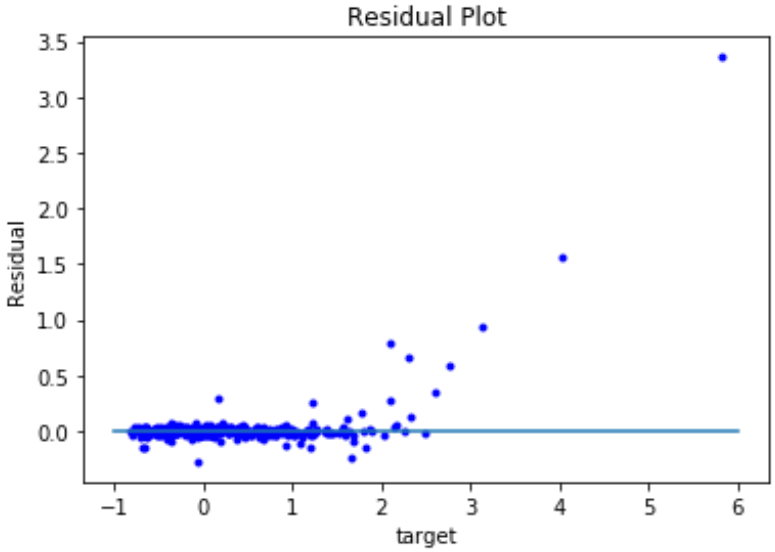


Fig 27. Residual Plot

The residuals are centered on zero throughout the range of fitted values. Though we see some large errors

`

Ridge & Lasso Regression

Ridge and Lasso regression are some of the simple techniques to reduce model complexity and prevent over-fitting which may result from simple linear regression.

Ridge Regression*:* In ridge regression, the cost function is altered by adding a penalty equivalent to square of the magnitude of the coefficients

$$\sum_{i=1}^{M}(y_i-\hat{y}_i)^2=\sum_{i=1}^{M}\left(y_i-\sum_{j=0}^{p}w_j\times x_{ij}\right)^2+\lambda\sum_{j=0}^{p}w_j^2$$

Lasso Regression: The cost function for Lasso (least absolute shrinkage and selection operator) regression can be written as

$$\sum_{i=1}^{M}(y_i-\hat{y}_i)^2=\sum_{i=1}^{M}\left(y_i-\sum_{j=0}^{p}w_j\times x_{ij}\right)^2+\lambda\sum_{j=0}^{p}|w_j|$$

We import the required libraries and use GridSearchCV to get the best alpha value for optimizing coefficients in ridge regression. Higher the alpha value, more restriction on the coefficients; when alpha value is 0 linear and ridge regression resembles.

```
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
```

```
Ridge()
params =  {'alpha':[0.001,0.05,0.1,0.5,1,10,100],"normalize":[True,False]}
model = GridSearchCV(Ridge(),param_grid=params,n_jobs=-1)
model.fit(X_train, y_train)
```

```
model.best_estimator_
```

```
Ridge(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=None,
      normalize=False, random_state=None, solver='auto', tol=0.001)
```

```
Ridgr_model_1=model.best_estimator_.fit(X_train,y_train)
y_train_pred = Ridgr_model_1.predict(X_train)
y_pred = Ridgr_model_1.predict(X_test)
from sklearn.metrics import r2_score, mean_squared_error
print ("R-squared Training",r2_score(y_train, y_train_pred))
print ("R-squared Testing",r2_score(y_test, y_pred))
print ("Train RMSE : ",np.sqrt(mean_squared_error(y_train, y_train_pred)))
print ("Test RMSE  : ",np.sqrt(mean_squared_error(y_test, y_pred)))
```

`

Ridge regression Results

R^2 Training	R^2 Testing	Train RMSE	Test RMSE
0.9953	0.9060	0.0560	0.3379

We get R^2 of 0.9060 on testing data which is less than the bagging with KNN but better than boosting and Decision Tree algorithms. We see lower values of RMSE score for training. There is significant difference between RMSE values.

Graph of Actual vs Predicated

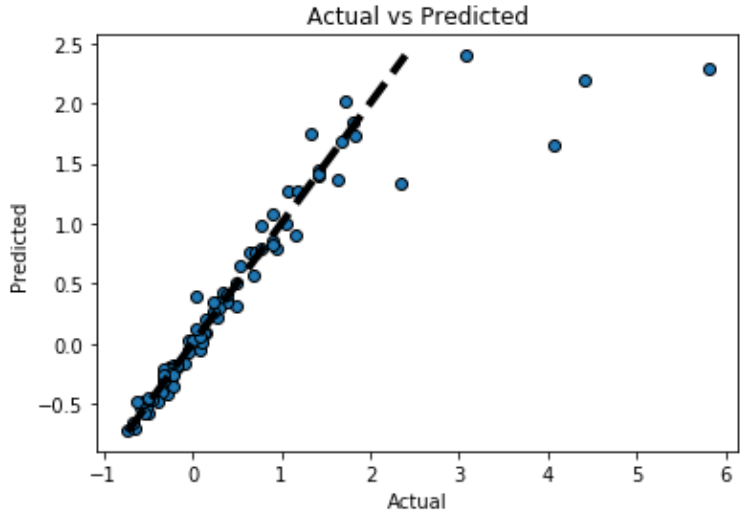


Fig 28. Actual vs Predicated

We see that most of the predicted values are closer to the actual values. Model is a good fit to our data.

Using residual plots, we assess whether the observed error (residuals) is consistent with error.

`

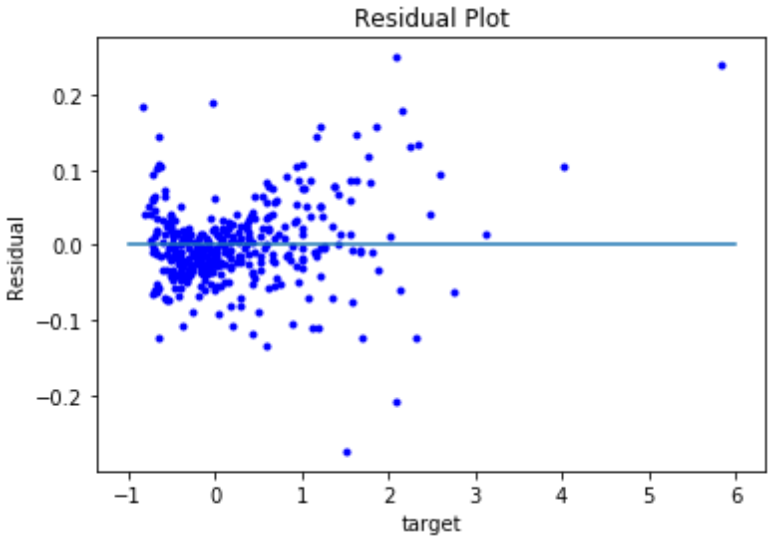


Fig 29. Residual Plot

The residuals are centered on zero throughout the range of fitted values. Though we see some large errors

Lasso regression can lead to zero coefficients i.e. some of the features are completely neglected for the evaluation of output. So, Lasso regression not only helps in reducing over-fitting but it can help us in feature selection

```
from sklearn.linear_model import Lasso
```

```
params =  {'alpha':[0.001,0.01,1]}
lasso_grid = GridSearchCV(Lasso(),param_grid=params,n_jobs=-1)
lasso_grid.fit(X_train, y_train)
```

```
lasso_grid.best_estimator_
```

```
Lasso(alpha=0.001, copy_X=True, fit_intercept=True, max_iter=1000,
      normalize=False, positive=False, precompute=False, random_state=None,
      selection='cyclic', tol=0.0001, warm_start=False)
```

```
Lasso_model_1=lasso_grid.best_estimator_.fit(X_train,y_train)
```

```
y_train_pred = Lasso_model_1.predict(X_train)
y_pred = Lasso_model_1.predict(X_test)
from sklearn.metrics import r2_score, mean_squared_error
print ("R-squared Training",r2_score(y_train, y_train_pred))
print ("R-squared Testing",r2_score(y_test, y_pred))
print ("Train RMSE : ",np.sqrt(mean_squared_error(y_train, y_train_pred)))
print ("Test RMSE  : ",np.sqrt(mean_squared_error(y_test, y_pred)))
```

`

Lasso regression Results

R^2 Training	R^2 Testing	Train RMSE	Test RMSE
0.9717	0.9151	0.1384	0.3212

We get R^2 of 0.9151 and lower values of Train and Test RMSE scores. Also, we have reduced the number of features required for the training of the model.

```
1 coeff_used = np.sum(Lasso_model_1.coef_!=0)
2 print("number of features used: ", coeff_used)
```

```
number of features used:  65
```

Graph of Actual vs Predicated

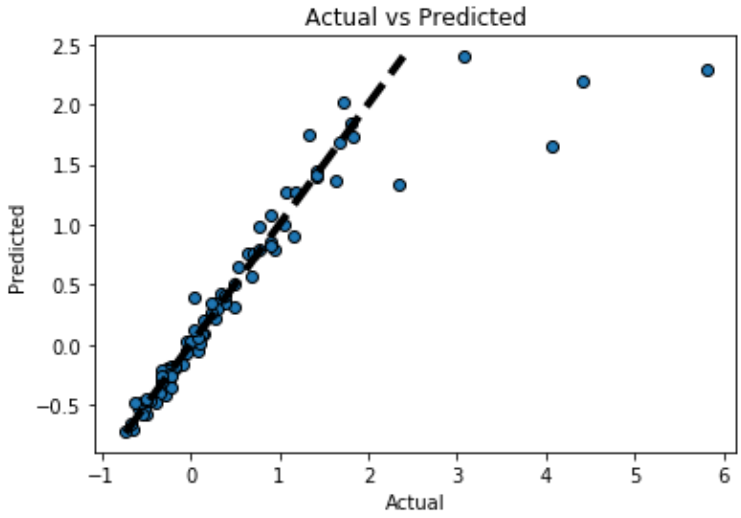


Fig 30. Actual vs Predicated

We see that most of the predicted values are closer to the actual values. Model is a good fit to our data.

Using residual plots, we assess whether the observed error (residuals) is consistent with error.

`

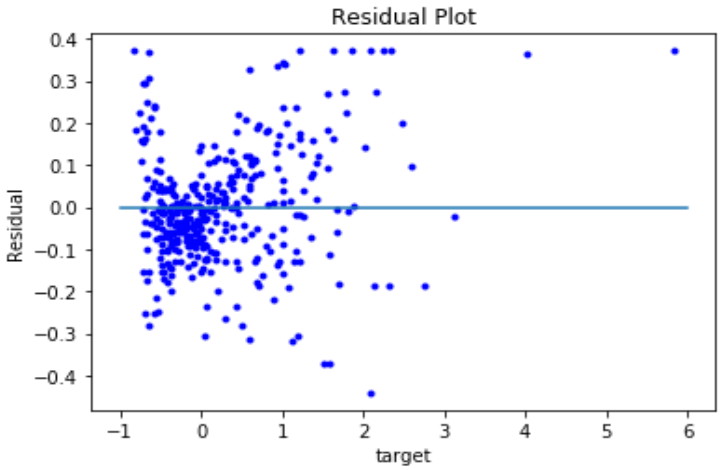


Fig 31. Residual Plot

The residuals are centered on zero throughout the range of fitted values. Though we see some large errors

Multiple-Linear-Regression

multiple regression is the extension of Simple linear regression that involves more than one explanatory variable. The Formula for Multiple Linear Regression Is:

$yi=\beta 0+\beta 1xi1+\beta 2xi2+...+\beta pxip+\epsilon$

where, for $i=n$ observations:

yi=dependent variable

xi=explanatory variables

$\beta 0$=y-intercept (constant term)

βp=slope coefficients for each explanatory variable

ϵ =the model's error term (also known as the residuals)

The multiple regression model is based on the following assumptions:

- linear relationship between the dependent variables and the independent variables.
- No or little multicollinearity.
- Residuals should be normally distributed with a mean of 0 and variance σ.
- No auto-correlation
- Data is homoscedastic

`

We first calculate the Variance Inflation Factors (VIFs) for all our numerical features in the dataset. It is used for Checking multicollinearity. Higher values signify that it is difficult to impossible to assess accurately the contribution of predictors to a model. Values of 10 or more are generally regarded as very high

```
#Using user defined function to calculate VIF Score
import statsmodels.formula.api as smf
def vif_cal(input_data, dependent_col):
    x_vars=input_data.drop([dependent_col], axis=1)
    xvar_names=x_vars.columns
    for i in range(0, xvar_names.shape[0]):
        y=x_vars[xvar_names[i]]
        x=x_vars[xvar_names.drop(xvar_names[i])]
        rsq=smf.ols(formula='y~x', data=x_vars).fit().rsquared
        vif=round(1/(1-rsq),2)
        print( " VIF score for", xvar_names[i], "is:" , vif)
vif_cal(df.select_dtypes(np.number),'Worldwide Gross')
```

```
VIF score for Production Budget is: 1.75
VIF score for Domestic Gross is: 27.94
VIF score for Opening is: 13.57
VIF score for Legs is: 3.07
VIF score for Domestic Share is: 1.16
VIF score for Theater counts is: 1.7
VIF score for Infl. Adj. Dom. BO is: 15.51
VIF score for Running Time is: 1.32
VIF score for No of Lead Actors is: 1.16
VIF score for No of support  cast is: 1.13
VIF score for No of support  Actors is: 1.13
```

- We see that Vif score for Domestic Gross, Opening, Infl. Adj. Dom. BO is higher than 10 and hence they are highly correlated. Multi-collinearity may be a problem
- VIFs for dummy variables representing nominal variables with three or more categories, usually not a problem. So, we do not calculate VIFs for categorical values

We drop the High VIF features and recheck the VIFs

```
df_vf=df.drop(["Domestic Gross","Opening"],axis=1)
```

```
#Using user defined function to calculate VIF Score
import statsmodels.formula.api as smf
def vif_cal(input_data, dependent_col):
    x_vars=input_data.drop([dependent_col], axis=1)
    xvar_names=x_vars.columns
    for i in range(0, xvar_names.shape[0]):
        y=x_vars[xvar_names[i]]
        x=x_vars[xvar_names.drop(xvar_names[i])]
        rsq=smf.ols(formula='y~x', data=x_vars).fit().rsquared
        vif=round(1/(1-rsq),2)
        print( " VIF score for", xvar_names[i], "is:" , vif)
vif_cal(df_vf.select_dtypes(np.number),'Worldwide Gross')
```

```
VIF score for Production Budget is: 1.6
VIF score for Legs is: 1.53
VIF score for Domestic Share is: 1.15
VIF score for Theater counts is: 1.53
VIF score for Infl. Adj. Dom. BO is: 1.85
VIF score for Running Time is: 1.28
VIF score for No of Lead Actors is: 1.06
VIF score for No of support  cast is: 1.13
VIF score for No of support  Actors is: 1.1
```

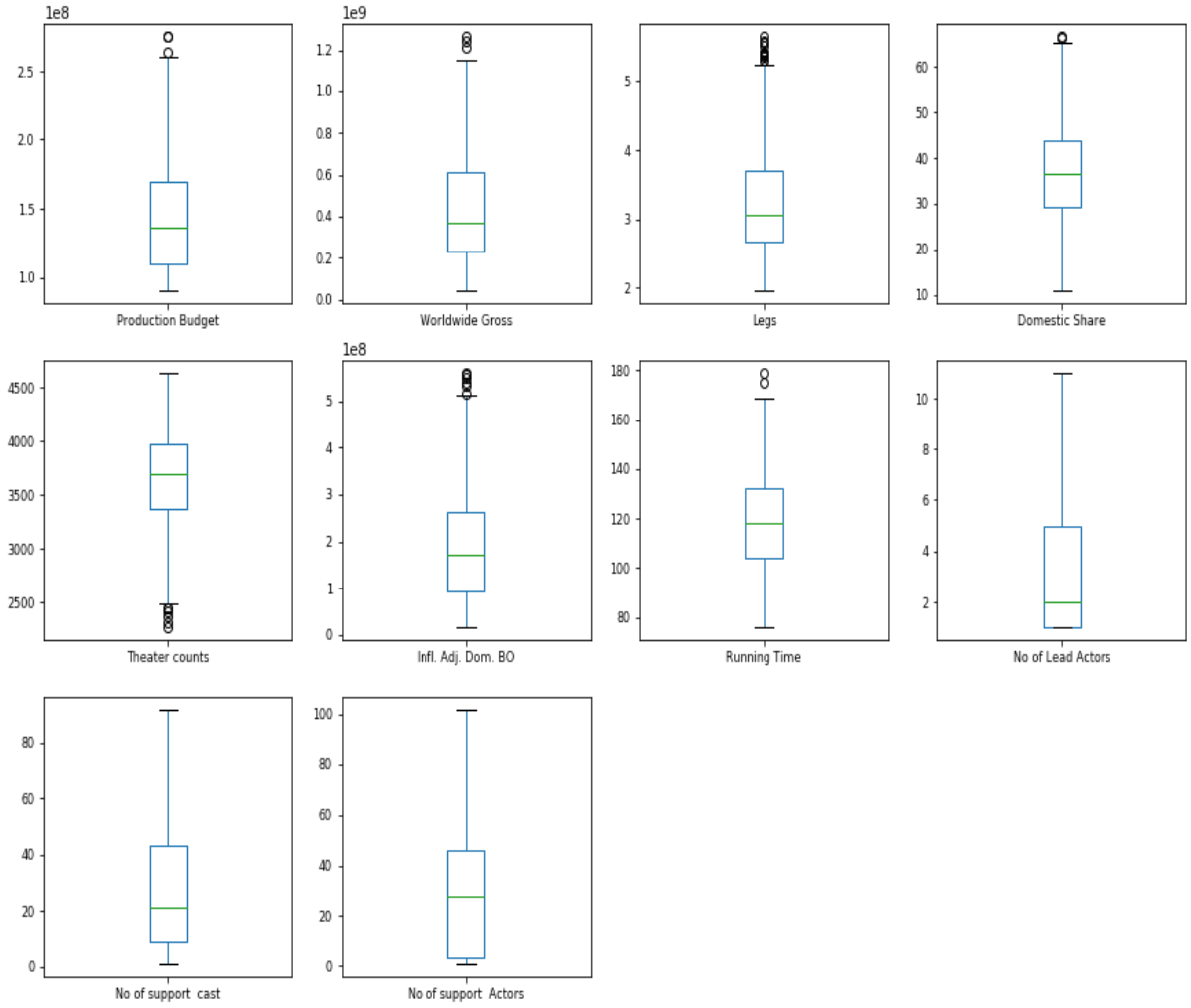
`

We see that now all the features have low VIFs Score (less than 2)

During the data preparation we have seen that we have outliers which we need to handle for MLR regression

```
Q1 = df_vf.quantile(0.25)
Q3 = df_vf.quantile(0.75)
IQR = Q3 - Q1
df_out = df_vf[~((df_vf < (Q1 - 1.5 * IQR)) |(df_vf > (Q3 + 1.5 * IQR))).any(axis=1)]
df_out.shape
```

```
(383, 16)
```



We then perform Robust scaling on the data

```
from sklearn.preprocessing import RobustScaler
scaler = RobustScaler()
robust_scaled_df_vf = scaler.fit_transform(df_vf_without_outlier)
robust_scaled_df_vf = pd.DataFrame(robust_scaled_df_vf, columns=df_vf_without_outlier.columns)
```

We then fit the model and test using 10-Fold cross validation

`

```
#Model training
from sklearn.linear_model import LinearRegression
model = LinearRegression()
MLR_model_no_outlier = model.fit(X_train3, y_train3)
y_train_pred = MLR_model_no_outlier.predict(X_train3)
y_test_pred = MLR_model_no_outlier.predict(X_test3)
```

```
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_squared_error
kfold1=KFold(n_splits=10)
r1=cross_val_score(MLR_model_no_outlier,X_train3,y_train3,cv=kfold1,scoring='r2')
print("R-squared Training",r1.mean())
```

```
R-squared Training 0.8458483554599192
```

```
kfold1=KFold(n_splits=10)
result2=cross_val_score(MLR_model_no_outlier,X_test3,y_test3,cv=kfold1,scoring='r2')
print("R-squared Testing",result2.mean())
```

```
R-squared Testing 0.9015235740181492
```

MLR regression Results

R^2 Training	R^2 Testing	Train RMSE	Test RMSE
1.0	0.8865	5.2494	0.2488

R^2 Training with Cross Validation	R^2 Testing with Cross Validation
0.8458	0.9015

We check the Q-Q plot which shows that residuals have normal distributed

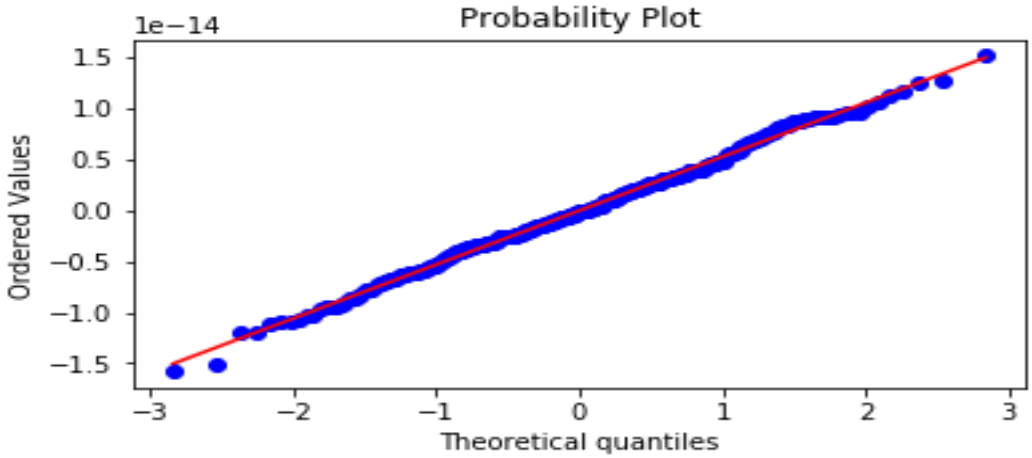


Fig 32. Q-Q plot

`

Graph of Actual vs Predicated

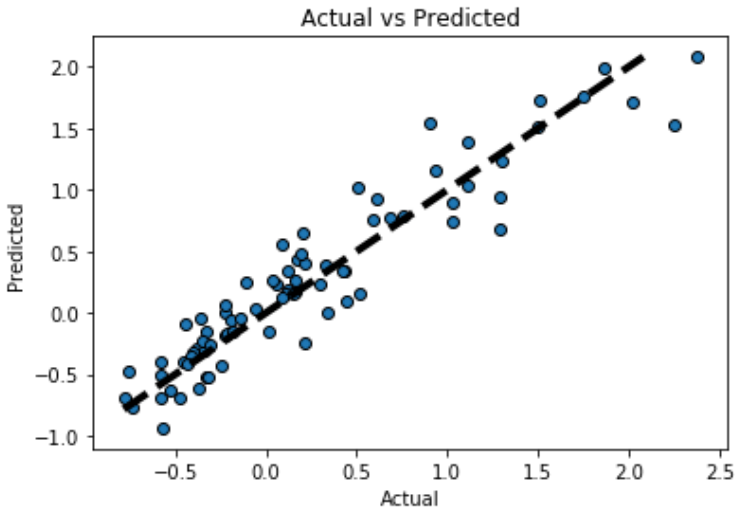


Fig 33. Actual vs Predicated

We see that most of the predicted values are closer to the actual values. Model is a good fit to our data.

Using residual plots, we assess whether the observed error (residuals) is consistent with error.

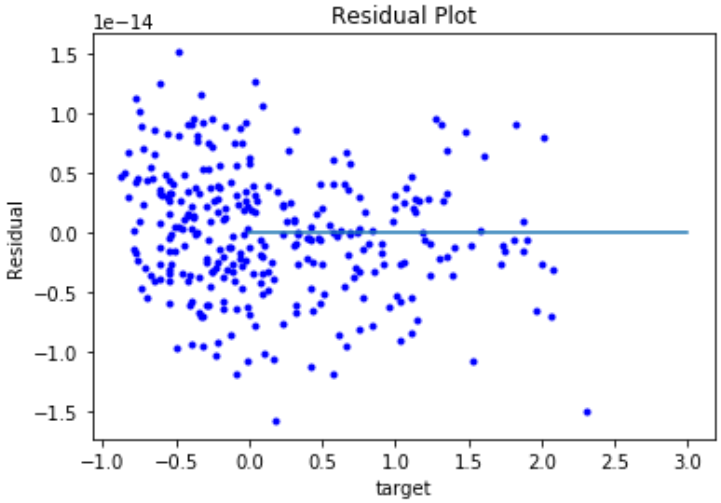


Fig 34. Residual Plot

The residuals are centered on zero throughout the range of fitted values. The residuals are random and do not show any pattern.

`

Identification of The Best Model

Algorithm	R^2 Training	R^2 Testing	Train RMSE	Test RMSE
Decision Tree (DT)	0.9302	0.8669	0.2436	0.2703
Bagging with DT	0.9787	0.8969	0.1303	0.2773
Bagging with KNN	0.8934	0.9191	0.2925	0.2443
Gradient Boosting	0.9327	0.7732	0.2136	0.5250
Ridge Regression	0.9953	0.9060	0.0560	0.3379
Lasso Regression	0.9717	0.9151	0.1384	0.3212
Multiple Linear Regression	1.0	0.8865	5.2494	0.2488

Conclusion

We see that as we have many outliers in the original data Tree based models perform better than MLR though single decision tree is not a very powerful predictor. As we have may features (942 after categorical encoding) interpreting decision tree and its graph are very difficult. We also noticed that bagging boosting and tree model overfits and its parameters have to be adjusted to get the best possible metrices.

We used Lasso regression which gave is the best R squared testing result of 0.9151 with lower RMSE values for both Test and train data. We also checked that it had reduce the number of features and used 65 features of the total 942 feature.