Project 2

# Data

For the data selection problem, I tried to choose with sufficient number of features, observations and including NA values. So I used Mckinsey’s ProHack Hackathon data which is published on the [web page of the competition](https://prohack.org/page/challenge). I also attended the competition with this solution. The task consists of two steps, data science & optimization. Since optimization is beyond the scope of this project, I will be dealing with the first task which is about predicting the future composite index to measure well-being performance of a newly discovered star. Therefore this is a regression problem.

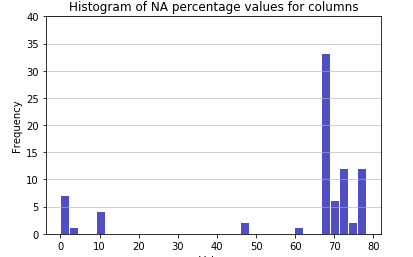
Data consists of 3865 number of points with 79 features and a prediction class.

## Pipeline

Pipeline will be as followed:

* Data preparation
* Data Analysis
* Model Selection
* Try Models with:
  + All columns
  + Top 10 of the best features (extra)
  + Top x of the best features (extra)
  + Select From Model (SFM) method
  + RFE feature selection
* Best Model with cross validation
* Best Model with hypertuning
* Feature Importance with Shap Package

## Data Preparation

Data includes lots of unknown values for the many of the columns. As can be seen below, around 75% of respective columns are including NA with a high percentage rates. In order not to lose most part of the data, I tried an imputing method for NA handling. I also tried *fill.na()* method but it performed worse. Moreover, since there are some negative values in some of the features such as 'Gross income per capita' or 'Private galaxy capital flows (% of GGP)' and their amouts are low, I replaced these values with absolutes.

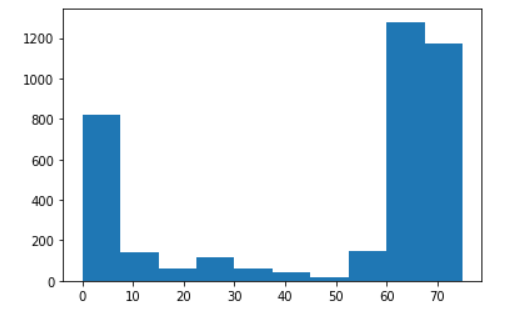
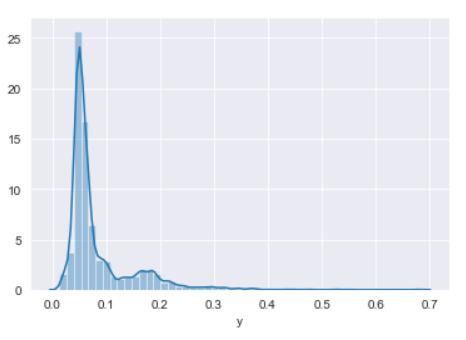
Chart 1: Histogram of NA percentage values for columns

Chart 2: Number of NA valued columns for respective rows

According to Chart 2, most of the rows are also empty. I eliminated rows with columns of 70 and more NA. 250 rows are eliminated with this filter. The rest of the data is imputed based on clustering galaxies. Galaxy is the unique identifier for the data. The process is calculating median of the columns for each galaxy and fill it. Since for some glaxies, respective columns can be full of NA. In order to handle this problem, I divide empty columns into 3 with kmeans and predict other NA values based on these cluster medians.

Chart 3: Distribution of y values

I examined the prediction class, y, in order to understand what I’m dealing with. Most of the y values are spreaded between 0 and 0.1.

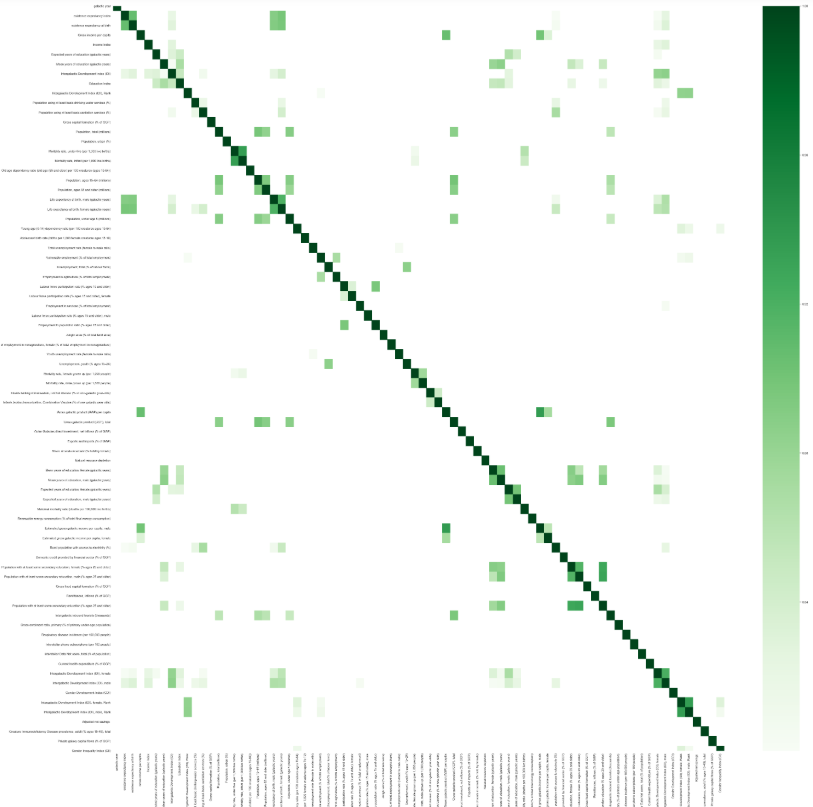
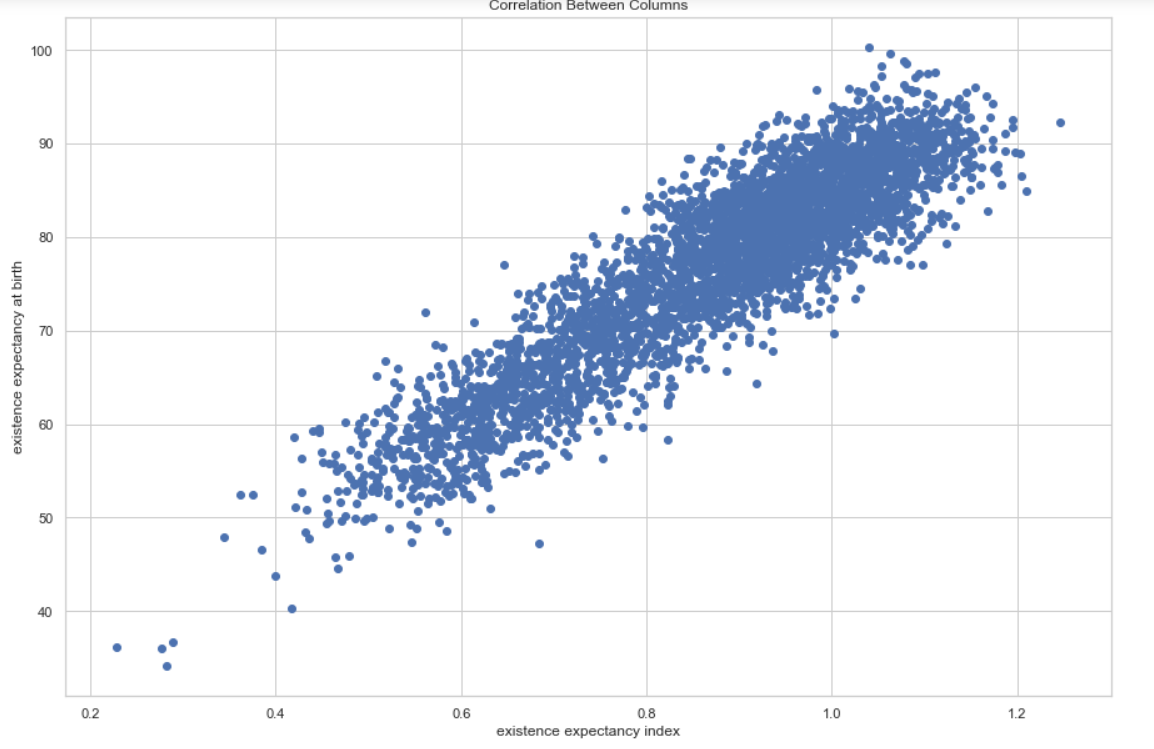


Chart 4: Correlation Matrix of the features

Since there are lots of features, some of the features are too correlated that we can eliminate. Removing correlated features can decrease complexity without any or a bit loss in accuracy. In order to understand it clearly; high correlations, only 80% or higher, are colored as green. I checked the first two high correlated columns which are *existence expectancy index* & *existence expectancy at birth.* Below graph proves the high correlation:

Chart 5: Correlation between one of high correlated column pair

In further steps of the project, this analysis will be helpful since using both these features will not increase the accuracy, however it will increase complexity and run time.

## Model Selection

For this dataset, we tried multiple models such as: ***Decision Tree Regressor, Gradient Boosting Regressor, Random Forest Classifier, XGBoost Regressor, Support Vector Regressor, k-Nearest Neighbors*** and ***Multi-Layer Percepton.***

### Scenario 1:

This scenario is solving models using all available features. So we define the models and predict. Since one of the performance criteria of the regression problems is *RMSE* score, below we can see the rmse scores of each model, individually. Random Forest, KNN and Xgboost performed pretty good when comparing to others.



Table 1: Model scores for scenario 1

### Scenario 2: (Extra)

We tried to choose the best features by using a **sklearn.feature\_selection** module. Since there are lots of features and some of the features does not affect the outcome, analyzed the best features by using *f\_regression* as a scoring function inside of the **SelectKBest** module. We selected top 10 features into the pipeline.

Using this features, we tried the pre-specified models. Results can be seen below, which is not satisfying since almost all the models’ scores gets worse, except multi-layer perceptron.



Table 2: Model scores for scenario 2

This shows us that we need to include more features than top 10.

### Scenario 3 (Extra):

We need to increase the number of features, so by using the method in previous scenario, we can select top *n* features. According to the Table 2, including more features for most of the models increased performance. In fact, KNN, Random Forest, XGBoost and Gradient Boosting performed pretty good when compared to others and increasing number of features also increased the model performance.

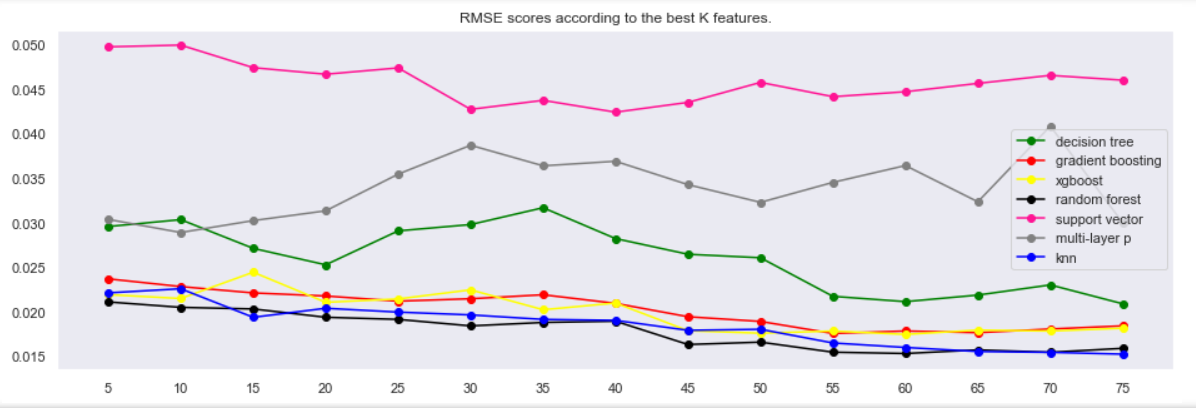


Chart 6: Model Scores for different number of features scenario

We performed better than the first scenario (using all available features) for most of the models. Below Table shows the minimum scores for the respective model with the optimal number of features. Now, we can try other methods to improve these results.



Table 3: Models scores for scenario 3

### Scenario 4:

This scenario is about using a subset of features based on the feature importance values acquired by the method used. In this scenario, we used the **sklearn.feature\_selection.SelectFromModel** module for three models, which are *Random Forest*, *Gradient Boosting, XGBoost*, *Decision Tree & Suppor Vector Regressor.* MLP and KNN does not support this module. The threshold value to use for feature selection is kept default, which is *mean*. This means that features whose importance is greater or equal are kept while the others are discarded.



Table 4: Number of features by model

We can see from the above that every model requires different number of features, this means that we can not select one number for all regressors. Also, except xgboost, all models performed worse than the previous scenario, which was the best for all.

### Scenario 5:

This scenario is about using best subset of features based on the recursive feature elimination process. Since feature elimination process in this scenario is computationally expensive, as previously mentioned, we will drop the high correlated features. Dropping correlated feautures will not be affect the model score. So, we removed features which have greater than 75% of correlation. Dropped high correlated features can be seen in the notebook. There are 31 columns left for this process.

After that we used the Recursive Feature Elimination process. It is a process that assigning weights to features and select them recursively by considering smaller and smaller sets of features. We have chosen the optimal number of features to be 15, which is half of the all eliminated columns. MLP and KNN do not support this module. We used **sklearn.feature\_selection.RFE** module and the found the best 15 features for the models.

When we trained the model with these optimal features, our rmse scores are as follows. Note that this scenario is not improved the best solutions we have found.



Table 5: model scores for scenario 5

Therefore, we will be using top 3 models to do cross validation and hyper-parameter tuning.

## Cross Validation

##### XGBoost

I used cross validation methods in order not to split train test data biased. For XGBoost, tried two methods such as stratified & Kfold. Both of the method shows consistent results.



Table 6: RMSE scores for both cross-val methods with different Ks.

K = 10 is the minimum average rmse scores for both of the methods. For xgboost, I will be using K = 10 for hyper-tuning.

##### Random Forest

For Random Forest, I tried only Stratified since both methods resulted pretty much the same. Again, K = 10 is the minimum average rmse score.



Table 7: RMSE scores for stratified cross-val method with different Ks

##### KNN



Table 8: RMSE scores for stratified cross-val method with different Ks

For KNN, different from other algorithms, K = 9 scored the minimum rmse. Also RMSE deviation is higher for KNN when comparing to others.

## Hyper-Tuning

Main aim is to beat the scores at pre-step with best features. I used RandomSearch to find the best parameter list, and use these best parameters in GridSearch to achive the optimal.

##### XGBoost

Initial parameters for XGBoost:

1. params = {
2. 'learning\_rate': [0.01, 0.04, 0.1, 0.3, 0.5],
3. 'min\_child\_weight': [1, 5, 10],
4. 'gamma': [0, 0.5, 1, 1.5, 2, 5],
5. 'subsample': [0.6, 0.8, 0.9, 1.0],
6. 'colsample\_bytree': [0.6, 0.8, 1.0],
7. 'max\_depth': [3, 4, 5, 6],
8. 'sampling\_method': ['uniform', 'gradient\_based'] ,
9. 'n\_estimators': [50, 100, 150, 200],
10. 'min\_child\_weight': [1, 3, 5, 7]
11. }

Initial model test score is 0.0179, with Random Search, RMSE train is 0.0069, RMSE test is 0.0186.

Best parameters for Random Search is:

1. {
2. 'subsample': 1.0,
3. 'sampling\_method': 'gradient\_based',
4. 'n\_estimators': 200,
5. 'min\_child\_weight': 7,
6. 'max\_depth': 5,
7. 'learning\_rate': 0.1,
8. 'gamma': 0,
9. 'colsample\_bytree': 0.6
10. }

Lastly, I used following parameters for grid search:

1. grid\_search = {
2. 'subsample': [model.best\_params\_['subsample'] - 0.01,
3. model.best\_params\_['subsample'],
4. model.best\_params\_['subsample'] + 0.01],
5. 'learning\_rate': [model.best\_params\_['learning\_rate'] - 0.01,
6. model.best\_params\_['learning\_rate'],
7. model.best\_params\_['learning\_rate'] + 0.01],
8. 'min\_child\_weight': [model.best\_params\_['min\_child\_weight'] - 1,
9. model.best\_params\_['min\_child\_weight'],
10. model.best\_params\_['min\_child\_weight'] + 1],
11. 'max\_depth': [model.best\_params\_['max\_depth'] - 1,
12. model.best\_params\_['max\_depth'],
13. model.best\_params\_['max\_depth'] + 1]
14. }

With grid search, RMSE train is 0.009, RMSE test is 0.019 which is worse than random search. Although, I tried hyper-tuning methods, Both Random Search and Grid search did not perform good for XGBoost which was unexpected.

##### Random Forest

Initial parameters for Random Forest:

1. params = {
2. 'max\_depth': list(np.linspace(3, 15, 3, dtype = int)) + [None],
3. 'max\_features': ['auto', 'sqrt','log2', None],
4. 'min\_samples\_leaf': [1, 4, 6, 8, 12],
5. 'min\_samples\_split': [2, 5, 7, 10, 14],
6. 'n\_estimators': list(np.linspace(10, 150, 5, dtype = int))
7. }

Initial model test score is 0.01538, With Random Search, RMSE train is 0.011, RMSE test is 0.016.

Best parameters for Random Search is:

1. best\_params = {
2. 'n\_estimators': 150,
3. 'min\_samples\_split': 7,
4. 'min\_samples\_leaf': 4,
5. 'max\_features': None,
6. 'max\_depth': 15
7. }

Lastly, I used following parameters for grid search:

1. grid\_search = {
2. 'max\_features': [rf.best\_params\_['max\_features']],
3. 'min\_samples\_leaf': [rf.best\_params\_['min\_samples\_leaf'] - 1,
4. rf.best\_params\_['min\_samples\_leaf'],
5. rf.best\_params\_['min\_samples\_leaf'] + 1],
6. 'min\_samples\_split': [rf.best\_params\_['min\_samples\_split'] - 1,
7. rf.best\_params\_['min\_samples\_split'],
8. rf.best\_params\_['min\_samples\_split'] + 1],
9. 'n\_estimators': [rf.best\_params\_['n\_estimators'] - 10,
10. rf.best\_params\_['n\_estimators'],
11. rf.best\_params\_['n\_estimators'] + 10]
12. }

With grid search, RMSE train is 0.0008, RMSE test is 0.0171 which is worse than random search. Both Random Search and Grid search did not perform again good for Random Forest which was unexpected. Next, I will try with KNN, which performed best among others.

##### KNN

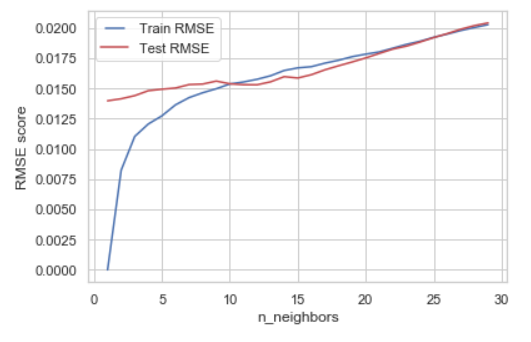
Since KNN is best among others, with all features scenario, I analyzed more to increase performance.

Chart 7: Train & Test RMSE for different number of neighbors

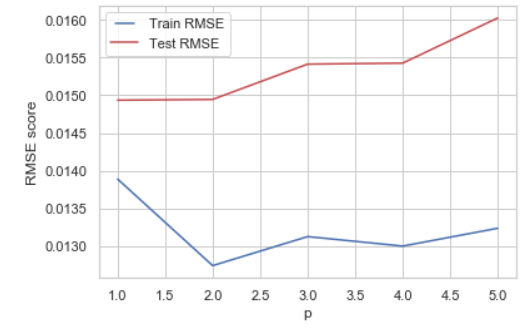
Chart 7 shows that number of test & train scores are consistent but increase when number of neighbors increases. Lowest RMSE score is resulted from n is around 2.

Chart 8: Train & Test RMSE for different p values.

Above chart shows that lowest test score is 1 & 2 for this case.

However, I relaxed the parameters to let model find its best parameters. Initial parameters for KNN:

1. params = {
2. 'algorithm': ['ball\_tree', 'kd\_tree', 'brute', 'auto'],
3. 'leaf\_size': [15, 20, 25, 30, 35, 40],
4. 'metric': ['minkowski', 'neg\_root\_mean\_squared\_error'],
5. 'n\_neighbors': list(np.linspace(3, 20, 1, dtype = int)),
6. 'p': [1, 2, 3, 4, 5]
7. }

Initial model test score is 0.0152, With Random Search, RMSE train is too small, 2.13E-09, RMSE test is 0.01432. Random search performed better than initial results with all available features.

Lastly, I used following parameters for grid search:

1. grid\_search = {
2. 'algorithm': [kn.best\_params\_['algorithm']],
3. 'leaf\_size': [kn.best\_params\_['leaf\_size'] - 1,
4. kn.best\_params\_['leaf\_size'],
5. kn.best\_params\_['leaf\_size'] + 1],
6. 'metric': [kn.best\_params\_['metric']],
7. 'n\_neighbors': [kn.best\_params\_['n\_neighbors'] - 1,
8. kn.best\_params\_['n\_neighbors'],
9. kn.best\_params\_['n\_neighbors'] + 1],
10. 'p': [kn.best\_params\_['p'] - 1,
11. kn.best\_params\_['p'],
12. kn.best\_params\_['p'] + 1]
13. }

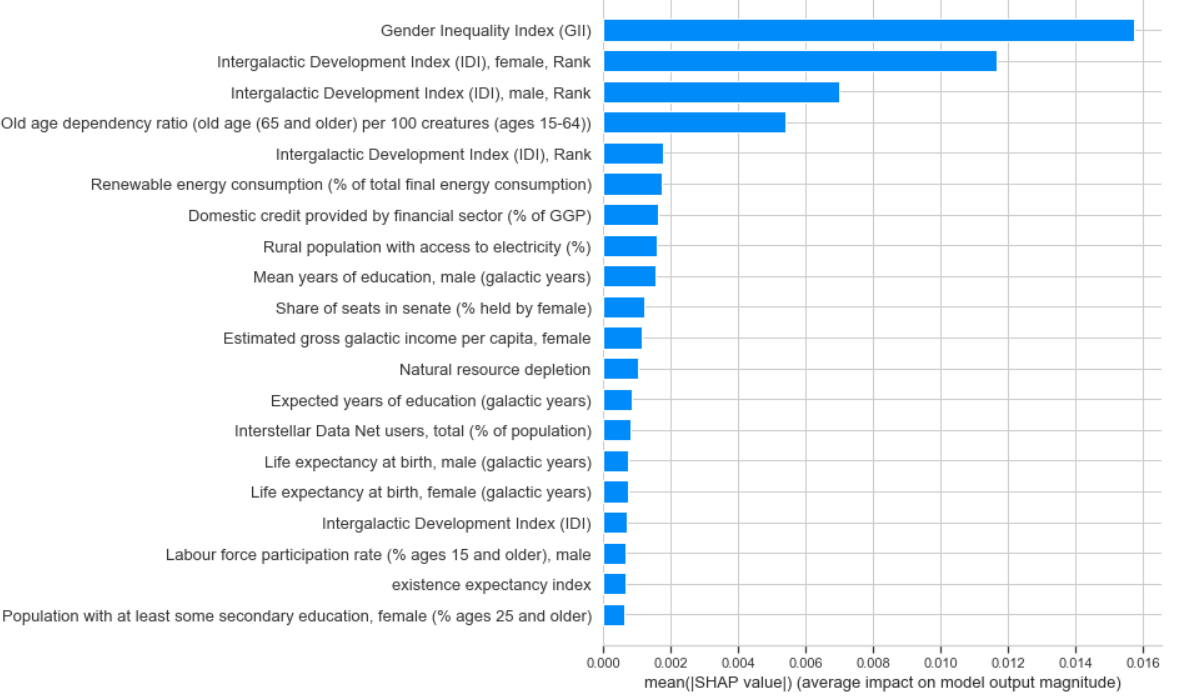
With grid search, RMSE train is 0.001, RMSE test is 0.0171 which is worse than random search. However, we have beaten the regular KNN with all features by cross-validation and hypertuning via random search. This dataset requires greater part of the features to be involved in prediction. Although train scores are so low that seems to be a overfit, test scores of random search for KNN beats other scenarios. Now, we will discuss the next option, feature importance by Shapley value.

## Shapley Value

This method is calculated feature importance based on local and global allocations using Shapley Value from game theory. Also we can use this package in order to explore which feature affects the model output and how they affect it. I will be using XGBoost for this option since it performed quite well in previous scenarios after KNN. Also, the package implementation for XGBoost is effective and fast as they mentioned in their github page.

The test rmse score for initial XGBoost implementation to the whole dataset is approximately 0.017.

First, I tried the affect of feature importance on the dataset. Although previous implementations shows that we should include more features to decrease rmse score, will try the feature importance with shapley values.

Chart 9: Absolute SHAP values and the feature importance

Above chart shows that Gender Inequality Index contributes to well-being index most. Therefore, we will discuss the number of features on model performance.

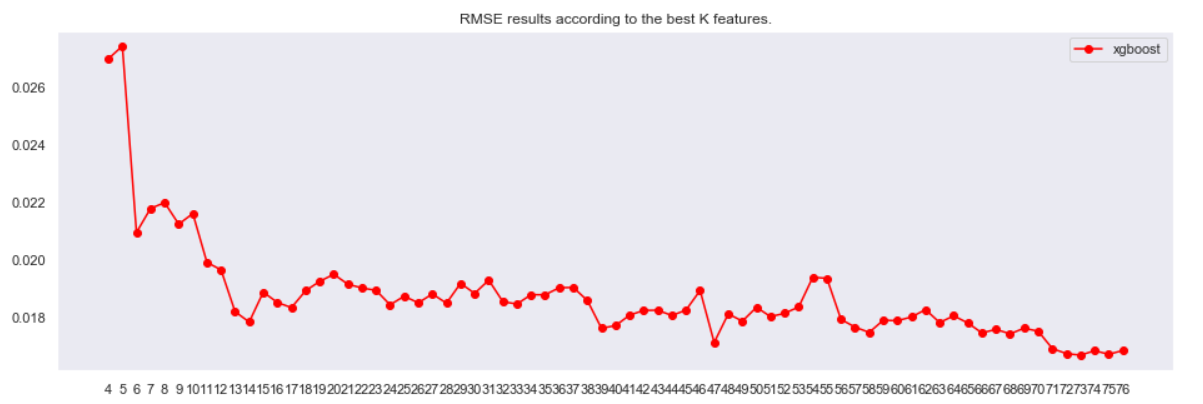
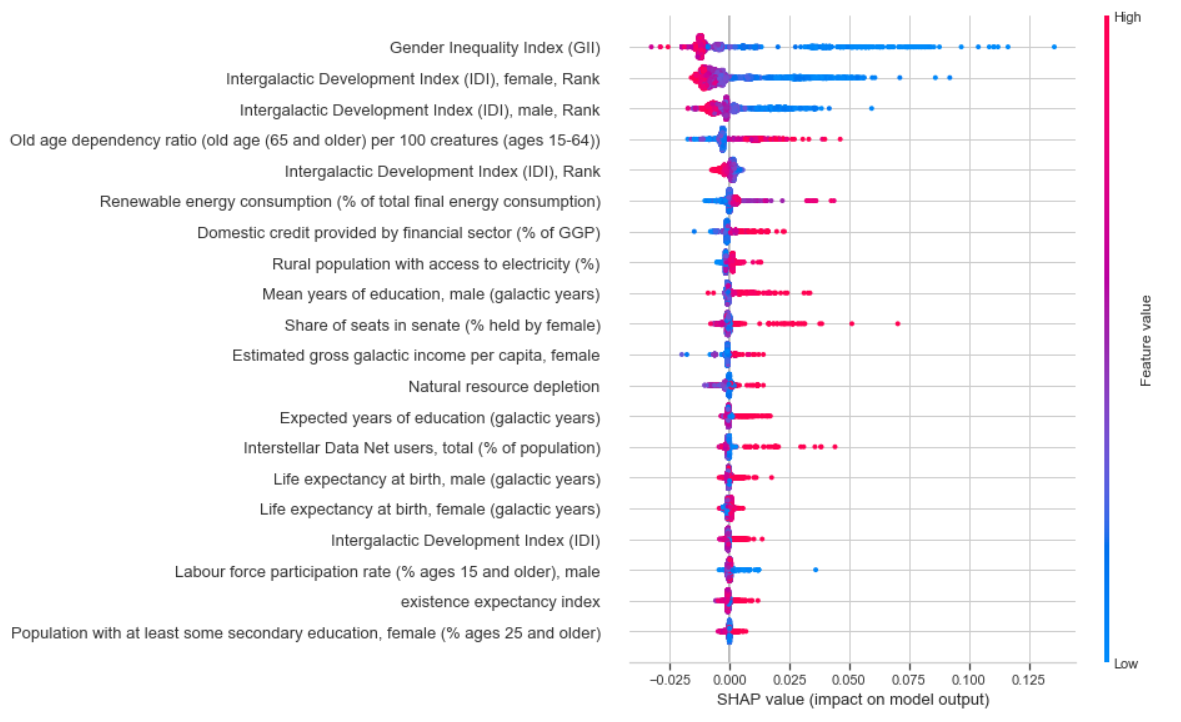
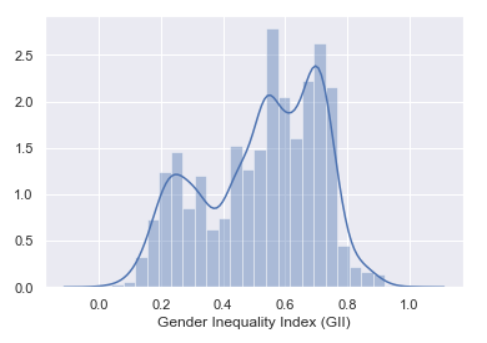


Chart 10: RMSE scores for every number of features

As we previously discovered that the test rmse scores decrease as the number of features increase. We can conclude that feature importance is not a good option for this dataset. However, by using shap package, we can investigate how much the distribution of features affects the outcome.

Chart 11: The distribution of the impacts each feature has on the model output

As we can understand from the above chart, the Gender Inequality Index has more impact on the well being index value. However, red values of GII impact the outcome negatively, as the corresponding SHAP values are negative. On the other hand, blue values impact positively. Red values of GII means that high values of the feature, blue values represents low valeus of the feature. Therefore we expect the predicted well being indexes to be high for low GII values, otherwise for high GII values.

Chart 12: Distribution of GII feature

In order the test the affect of different values of GII on the outcome easily, we divide the data into two parts. Above Distribution of GII shows that we can divide data into two parts from approximately 0.45, which seems to be the breaking point. With the top 20 features, the rmse test score is 0.0276, which is pretty high. However, below we can see the distribution of well being indexes for low GII data, on average 0.14.

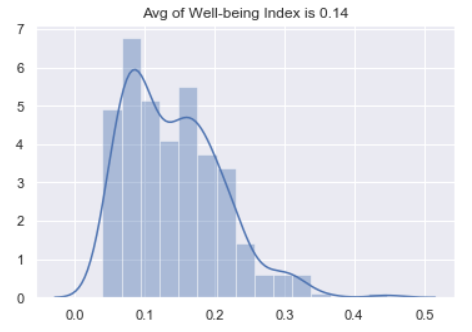


Chart 13: Distribution of outcome values for Low GII values

On the other hand, the distribution of well being indexes for High GII values can be seen below.

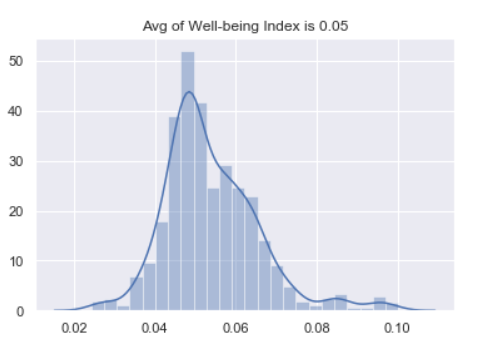


Chart 14: Distribution of outcome values for High GII values

For High GII values, on average, the well being index is 0.05, which is lower than the previous implementation. Actually, this proves our hypothesis which was the predicted well being indexes to be high for low GII values, low for high GII values. In addition to that, the rmse test score for High GII values is 0.00655, which is the lowest score for the entire project.

## References

<https://towardsdatascience.com/the-5-classification-evaluation-metrics-you-must-know-aa97784ff226>

<https://bookdown.org/max/FES/recursive-feature-elimination.html>

<https://math.usu.edu/~adele/forests/cc_home.htm>

<https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html>

<https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectFromModel.html#sklearn.feature_selection.SelectFromModel>

<https://www.kaggle.com/uciml/internet-advertisements-data-set>

<https://www.kaggle.com/dansbecker/shap-values>

<https://www.kaggle.com/dansbecker/advanced-uses-of-shap-values>

<https://prohack.org/>

<https://medium.com/@mohtedibf/in-depth-parameter-tuning-for-knn-4c0de485baf6>

<https://www.nature.com/articles/s42256-019-0138-9>

<https://christophm.github.io/interpretable-ml-book/shap.html#examples-4>