Machine Learning Engineer Nanodegree

Capstone Project

June Yim
May 15th, 2019

I. Definition

Project Overview

When graduate schools decide to get new entrants, they consider many records of the applicants including undergraduate GPA, GRE, TOEFL, letters of recommendation and so on. From the Data Science and Machine Learning point of view, by analyzing the admission data and training that, we can make a prediction model to help applicants to concentrate on more important factors which lead them to get the acceptance.

Problem Statement

The problem is : “How much I can have confidence that I can get the acceptance from graduate school with my record?” The record includes GPA, GRE score, TOEFL score, and so on.
The confidence is expressed as a number in range [0, 1]. That is, we can think the confidence as the probability of getting the acceptance.
The solution to the problem is the confidence level that the applicants can expect for the graduate admission. The confidence level is a number between 0 and 1 (inclusive) so the applicants can think of this as the probability that he or she can get the admission with his or her records(GRE score, TOEFL score,...). I will attempt to solve this problem with various regression models including linear regression, decistion tree, random forest and so on. Furthurmore, I will also try to apply classification algorithms like Gaussian Naive Bayes.

Metrics

There are many metrics to measure performance of regression problem including RMSE, MAE, R squared and so on:

- RMSE(Root Mean Square Error): This measures the standard deviation of the errors the system makes in its predictions. Smaller is better.
- MAE(Mean Absolote Error): This measures the average magnitude of the errors in a set of predictions, without considering their direction. It’s the average over the test sample of the absolute differences between prediction and actual observation where all individual differences have equal weight.
- R squared(Coefficient of Determination): This is the proportion of the variance in the dependent variable that is predictable from the independent variables. Larger is better.

There are many discussions about which metric is better for many circumstances. Actually, in my project there will not be big difference on which metric I choose so I can use any of them at this time, but I chose RMSE because I want to take precision first, penalize large prediction errors more and computationally simple metric. Although MAE is also computationally simple and easy to understand, RMSE penalize large errors more than MAE.(I've referred to this article: https://medium.com/human-in-a-machine-world/mae-and-rmse-which-metric-is-better-e60ac3bde13d) Compared to R squared, RMSE is better option for a metric with the 'precision-first' objective. (My reference about this: https://www.kdnuggets.com/2018/04/right-metric-evaluating-machine-learning-models-1.html)

Secondly, I benchmarked a few classification algorithms so I need another metric suitable for comparing classification models. As for regression models, there are many metrics for classification models including AUC-ROC Curve, Log loss, F-Beta score, F1 score, an so on. Among them I will use F1 score as a metric because it is commonly used for many classification problems and easy to understand. (Actually, I have more knowledge and experience in F1 score than another metrics.)
F1 Score is harmonic mean of precision and recall. That is, F1 = 2 * precision * recall / (precision + recall)

'Precision' is the number of correct positive results divided by the number of all positive results returned by the classifier.
'Recall' is the number of correct positive results divided by the number of all relevant samples (all samples that should have been identified as positive).
(For more detail, please refer to these links:https://medium.com/thalus-ai/performance-metrics-for-classification-problems-in-machine-learning-part-i-b085d432082bhttps://en.wikipedia.org/wiki/F1_score)

II. Analysis

Data Exploration

I used Graduate Admission dataset from kaggle.
(https://www.kaggle.com/mohansacharya/graduate-admissions)
This dataset contains 9 parameters:

- Serial No. (1 to 500)
- GRE Scores (290 to 340)
- TOEFL Scores (92 to 120)
- University Rating (1 to 5)
- Statement of Purpose (1 to 5)
- Letter of Recommendation Strength (1 to 5)
- Undergraduate CGPA (6.8 to 9.92)
- Research Experience (0 or 1)
- Chance of Admit (0.34 to 0.97)

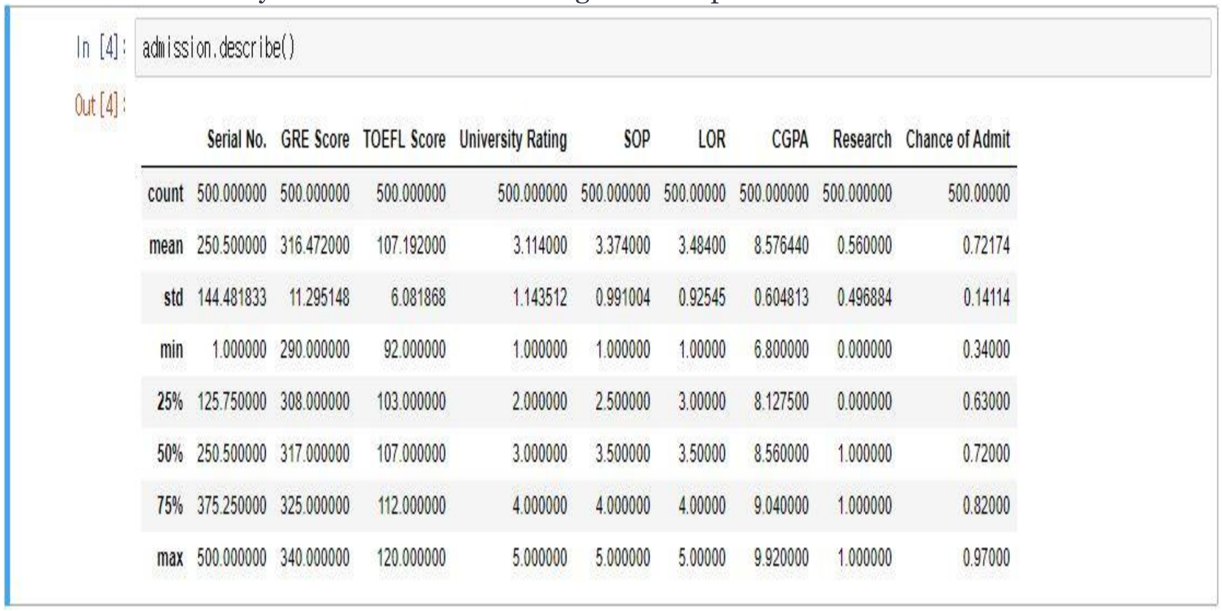
Let's see some real data in this dataset to have a sense for how the raw data look:

> *admission.head()*

Serial No.	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research	Chance of Admit
1	337	118	4	4.5	4.5	9.65	1	0.92

Serial No.	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research	Chance of Admit
2	324	107	4	4.0	4.5	8.87	1	0.76
3	316	104	3	3.0	3.5	8.00	1	0.72
4	322	110	3	3.5	2.5	8.67	1	0.80
5	314	103	2	2.0	3.0	8.21	0	0.65

I'll set 'Chance of Admit' parameter as the label and the remaining parameters as inputs to make a prediction model. Serial No. will be dropped in making the model because it is simply used for indexing purpose.
As I showed in the jupyter notebook ("capstone.ipynb") this set has 500 entries, all variables are numeric (there is no categorical variable) and missing values or abnormalities about data were not found.
Let's look at summary statistics of variables to get overall picture of data:



```
In [4]: admission.describe()

Out[4]:
       Serial No.  GRE Score  TOEFL Score  University Rating         SOP       LOR        CGPA    Research  Chance of Admit
count  500.000000 500.000000   500.000000         500.000000  500.000000 500.00000  500.000000  500.000000        500.00000
mean   250.500000 316.472000   107.192000           3.114000    3.374000   3.48400    8.576440    0.560000          0.72174
std    144.481833  11.295148     6.081868           1.143512    0.991004   0.92545    0.604813    0.496884          0.14114
min      1.000000 290.000000    92.000000           1.000000    1.000000   1.00000    6.800000    0.000000          0.34000
25%    125.750000 308.000000   103.000000           2.000000    2.500000   3.00000    8.127500    0.000000          0.63000
50%    250.500000 317.000000   107.000000           3.000000    3.500000   3.50000    8.560000    1.000000          0.72000
75%    375.250000 325.000000   112.000000           4.000000    4.000000   4.00000    9.040000    1.000000          0.82000
max    500.000000 340.000000   120.000000           5.000000    5.000000   5.00000    9.920000    1.000000          0.97000
```

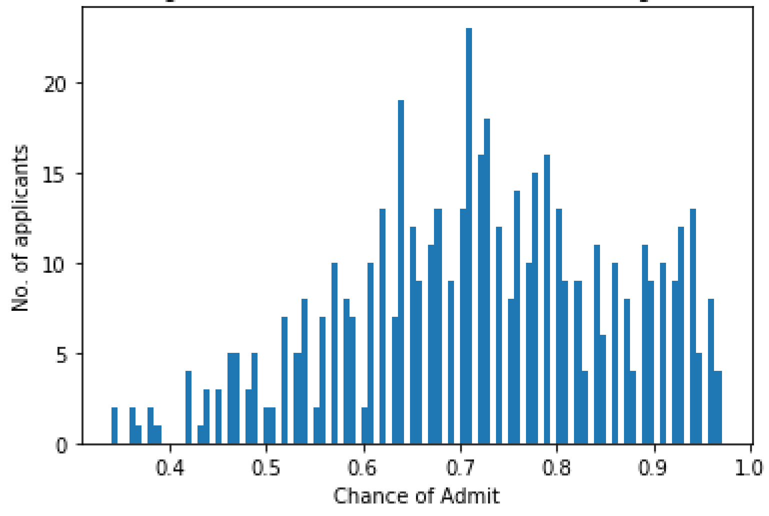
We can see every variable has 500 entries so there is no missing value in the dataset. And from the mean, standard deviation and the other statistics we can expect there are not abnomalities we should take care of before building up the prediction model. Lastly, when we look at GRE Score and TOEFL Score, they are mostly three-digit numbers. But the other variables are one-digit numbers. This means we need some data preprocessing like scaling to get better model.

Exploratory Visualization

By plotting some variables we can have more in-depth understanding of each variable as well as the relationship among the variables.
[Fig 1] shows the distribution of our target variable, 'Chance of Admit'.

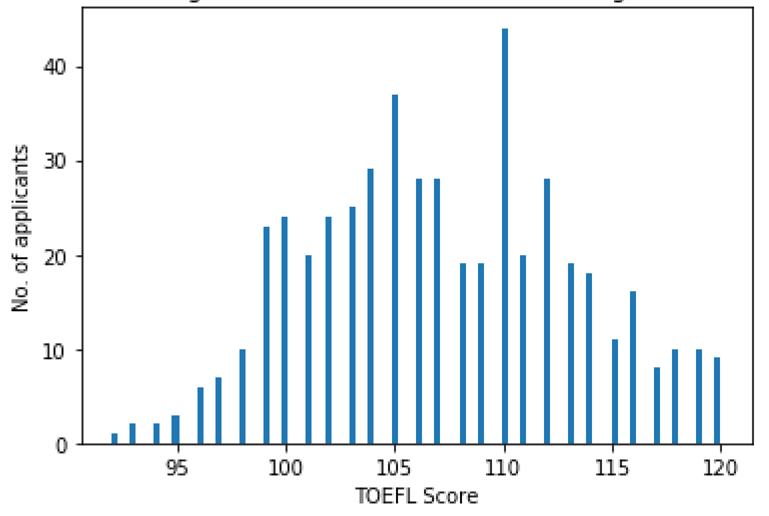
We can see that most applicants think that their chance of admit is over 60% and the largest portion is in 60~80% range.

[Fig 1] Chance of Admit Distribution Histogram

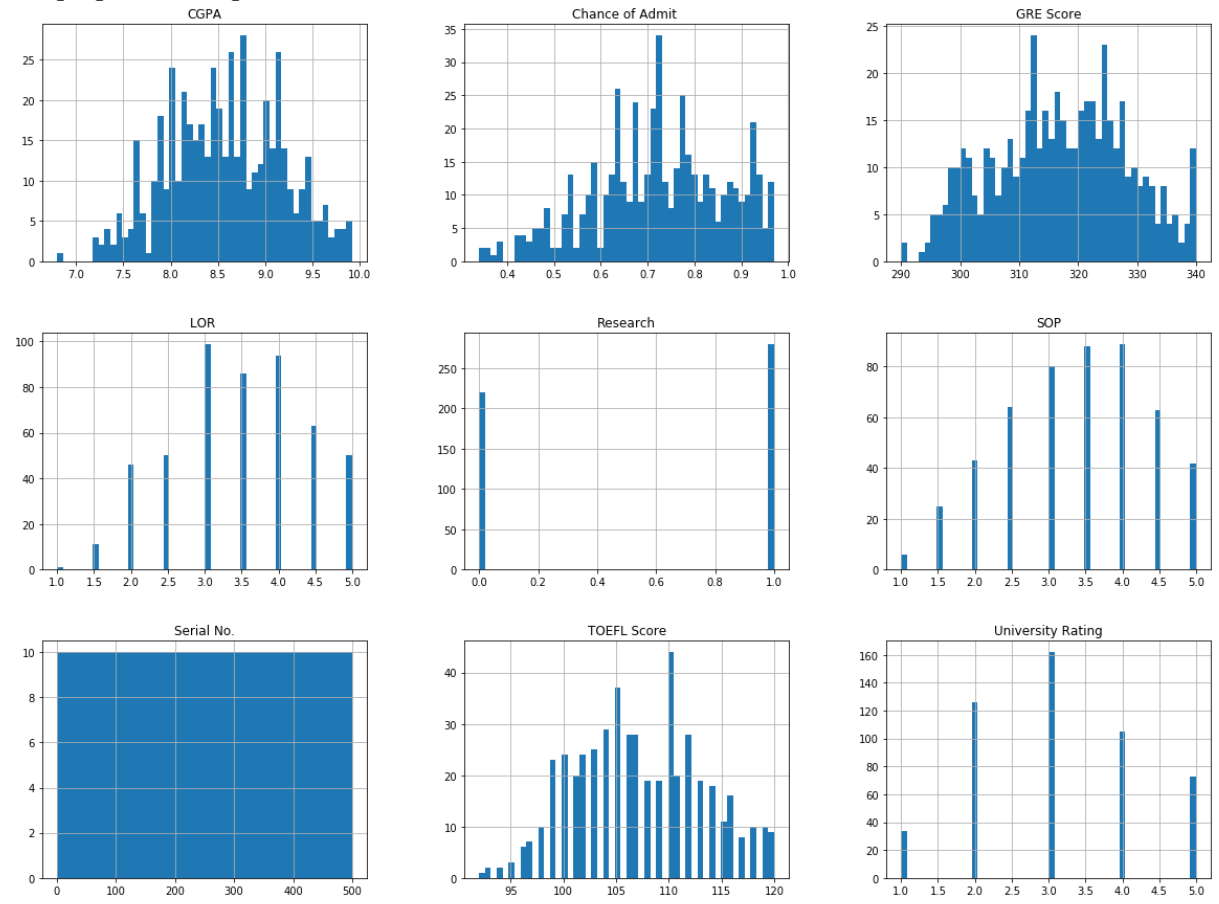


Let’s look at one more distribution.
[Fig 2] shows the distribution of TOEFL Score. We can see that most applicants have scores over 100. (pretty high-level applicants)

[Fig 2] TOEFL Score Distribution Histogram

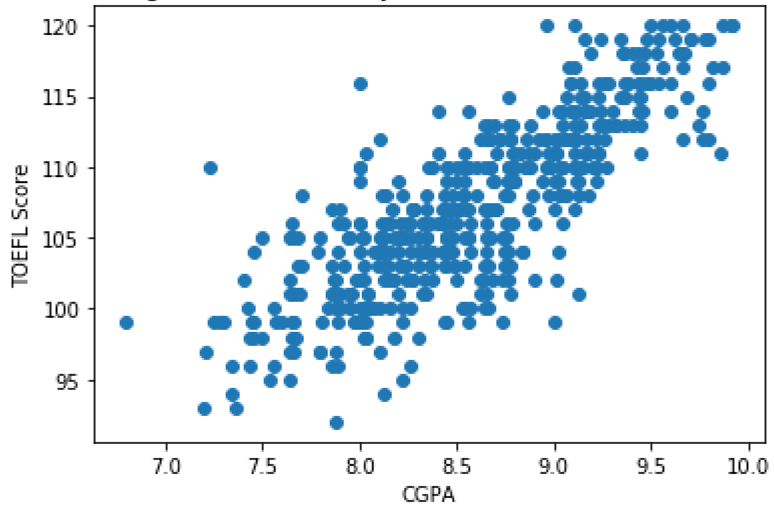


Congregated histogram of all variables is here:



Now, let’s see the relations between some variables.
We can see the relationship between TOEFL Score and CGPA in [Fig 3].

[Fig 3] Bi-variate Analysis of CGPA and TOEFL Score



This scatter plot shows that CGPA is positively related to TOEFL Score. In other words, the applicants who have high CGPA also tend to have high TOEFL Score and vise versa.
We can find the most important features which are higly correlated to the target with following code:

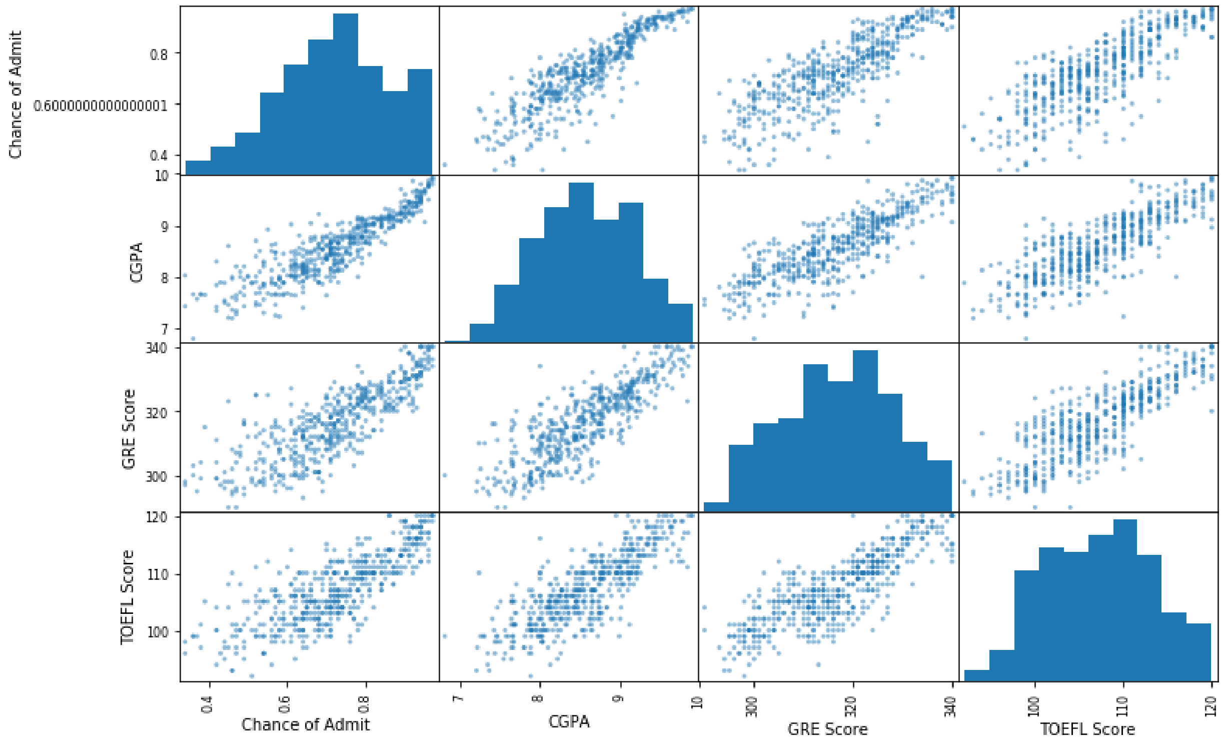
```
corr_matrix = admission.corr()
corr_matrix["Chance of Admit "].sort_values(ascending=False)
```

Feature	Corr
Chance of Admit	1.000000
CGPA	0.882413
GRE Score	0.810351
TOEFL Score	0.792228
University Rating	0.690132
SOP	0.684137
LOR	0.645365
Research	0.545871
Serial No.	0.008505

In this table, 'Corr' represents the strength of relationship between each feature and the target ('Chance of Admit'). Corr can take a number (-1, 1), if Corr is close to -1 it means they are negatively related, or if Corr is close to 0, it means they are not related, or if Corr is close to 1 it means they are positively related.

Now we know that top three important variables which are highly related to the target variable ('Chance of Admit') are CGPA, GRE Score, and TOEFL Score.

We can see the correlations among them by plotting the correlation matrix:



Algorithms and Techniques

For the regression problem like this, there are many algorithms which show good performance, Linear Regression, Decision Tree, Random Forest and so on.
Let's look at each algorithm briefly:

[Linear Regression]

- Linear Regression establishes a relationship between dependent variable (Y) and one or more independent variables (X) using a best fit straight line (also known as regression line).
- It is represented by an equation Y=a+b*X + e, where a is intercept, b is slope of the line and e is error term. This equation can be used to predict the value of target variable based on given predictor variable(s).
- Pros: easy to undertand, simple to implement and very fast
- Cons: There must be linear relationship between independent and dependent variables. It is very sensitive to Outliers. It can terribly affect the regression line and eventually the forecasted values.

Though choosing anyone would be good I chose Linear Regression because it is simple, easy to implement and understand. For the other algorithms I used them as benchmarks.
When I used Linear Regression algorithm, I used "Chance of Admit" as a dependent(target) variable, and the others as independent(input) variables. I implemented all these in the jupyter notebook. ("capstone.ipynb")

Benchmark

I benchmarked several algrithms not only in regression category but also in classification category:

- Decision Tree
- Random Forest
- SVM(Support Vector Machine)
- Gaussian Naive Bayes
- Decision Tree Classifier
- Random Forest Classifier
- Logistic Regression

Let’s take a look at briefly each benchmark algorithm:

Algorithm	**Explanation**
Decision Tree	Decision tree can be used to classification and regression both having a tree like structure. In a decision tree building algorithm first the best attribute of dataset is placed at the root, then training dataset is split into subsets. Splitting of data depends on the features of datasets. This process is done until the whole data is classified and we find leaf node at each branch. Information gain can be calculated to find which feature is giving us the highest information gain. Decision trees are built for making a training model which can be used to predict class or the value of target variable.
Random Forest	This also can be used for classification as well as regression. Multiple number of decision trees taken together forms a random forest algorithm. Each decision tree includes some rule-based system. For the given training dataset with targets and features, the decision tree algorithm will have set of rules. In random forest unlike decision trees there is no need to calculate information gain to find root node. It use the rules of each randomly created decision tree to predict the outcome and stores the predicted outcome. Further it calculates the vote for each predicted target. Thus high voted prediction is considered as the final prediction from the random forest algorithm.
Support Vector Machine	Support vector machine is a binary classifier. Raw data is drawn on the n-dimensional plane. In this a separating hyperplane is drawn to differentiate the datasets. The line drawn from centre of the line separating the two closest data-points of different categories is taken as an optimal hyperplane. This optimised separating hyperplane maximizes the margin of training data. Through this hyperplane, new data can be categorised.
Gaussian Naive Bayes	It is a technique for constructing classifiers which is based on Bayes theorem used even for highly sophisticated classification methods. It learns the probability of an object with certain features belonging to a particular group or class. In short, it is a probabilistic classifier. In this method occurrence of each feature is independent of occurrence another feature. It only needs small

Algorithm	Explanation
	amount of training data for classification, and all terms can be precomputed thus classifying becomes easy, quick and efficient.
Logistic Regression	In logistic regression we have lot of data whose classification is done by building an equation. This method is used to find the discrete dependent variable from the set of independent variables. Its goal is to find the best fit set of parameters. In this classifier, each feature is multiplied by a weight and then all are added. Then the result is passed to sigmoid function which produces the binary output. Logistic regression generates the coefficients to predict a logit transformation of the probability.

Each benchmark was performed on train data set at first, and then on test set.
Thinking of overfitting due to small data size, I used Scikit Learn's k-fold cross-validation to diminish overfitting and get better results. It randomly splits the training set into k distinct subsets called folds, then it trains and evaluates the model k times, picking a different fold for evaluation every time and training on the other k-1 folds. (I can decide and give the number k to the cross-validation function as a parameter)
The results are in the following Implementation section.

III. Methodology

Data Preprocessing

As I mentioned above, any abnormalities in data set did not found. Furthermore, as there is no categorical variable, we can focus our consideration on preprocessing for numerical variables. When we look at the data statistics (see Data Exploration section above) the value scale ranges are different among variables, we need to adjust scale to get more accurate results.
Scikit learn provides some scaling methods including Standard scaler, MinMax Scaler, and so on. I used MinMaxScaler which scales every numerical data into (0, 1).

Implementation

1. Define Input & Output : As a first step, I defined output(target) variable and input variables for a prediction model.
   Output is "Chance of Admit" and the remaining variables are inputs to be used for the prediction. Among the variables, "Serial No." was dropped.
2. Split Train & Test set : Next, I splitted the data set into train and test set. Train set will be used for model implementation and test set will be used for measuring the implemented model's prediction performance for unseen data.
   For splitting, I used Scikit learn's train_test_split function with proper argument settings (test set size to be 20% of data set and random state to be 42 which will be used as a seed by random number generator)
3. Feature Scaling : To adjust input data scale, I used MinMaxScaler of Scikit Learn.
   This scaler converts every input data (all numeric) into (0, 1) scale. To do this, I applied fit_transform method to train data and transform method to test data.
4. Now, It's time to fit our train data. I chose Linear Regression algorithm and fit the train data which I splitted in step 2.
   Scikit learn provides many machine learning algorithms. To use linear regression

algorithm, I imported LinearRegression from sklearn.linear_model. There are not much thing to do after importing. Just calling 'fit' with our train input and label variable is enough to fit our train data. Lastly, I performed prediction with train data input using predict method of LinearRegression.
I used default hyperparameter setting like this:

> *LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)*

5. Measure Performance : To measure our prediction model's performance, some metrics are needed. I chose RMSE for measuring my regression model's performance as I mentioned before.
   To use that, I imported mean_squared_error from sckit learn and gave train data label and the prediction result from step 4 as function arguments. And then applied numpy's square root function to get RMSE.
6. Compare with test data set : After prediction and measurement with train data, to see the performance of my prediction model on unseen data, I performed prediction and measurement (wth RMSE) again on test data set. If the RMSE of train set is far lower than that of test set, this means overfitting happens. My result was train RMSE: 0.0593 and test RMSE: 0.0608, nearly same two numbers, which means overfitting did not happen in my linear regression model.
7. Benchmark : To benchmark, I tried some other regression algorithms including Decision Tree, Random Forest. Also, for wider benchmarking I tried some classification algorithms including SVM(Support Vector Machine), Gaussian Naive Bayes, Decision Tree Classifier, Random Forest Classifier and Logistic Regression. To use these classification algorithms I needed to define binary classification label. I divided the target variable - Chance of Admit - into two classes, 1 and 0. When the Chance of Admit is greater than 80%, the data instance is labelled with '1' or else, it is labelled with '0'. For metric I used Sckit learn's score method(the mean accuracy on the given test data and labels) as well as f1 Score(weighted average of the precision and recall)

   The hyperparameters for each benchmark algorithm I used are as follows:

- Decision Tree (random_state=42, and default settings for the others)
  ('random_state' is the seed used by the random number generator.)

> *DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, presort=False, random_state=42, splitter='best')*

- Random Forest (n_estimators=100, random_state=42, and defaults for the others)
  ('n_estimators' is the number of trees in the forest.)

> *RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=None, oob_score=False, random_state=42, verbose=0, warm_start=False)*

- Support Vector Machine (random_state=42, and defaults for the others)

> *SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, decision_function_shape='ovr', degree=3, gamma='auto_deprecated', kernel='rbf', max_iter=-1, probability=False, random_state=42, shrinking=True, tol=0.001, verbose=False)*

- Gaussian Naive Bayes (all default setting)

> *GaussianNB(priors=None, var_smoothing=1e-09)*

- Decision Tree Classifier (all default setting)

> *DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None, max_features=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, presort=False, random_state=None, splitter='best')*

- Random Forest Classifier (n_estimators=100, random_state=42, and defaults for the others)

> *RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini', max_depth=None, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=None, oob_score=False, random_state=42, verbose=0, warm_start=False)*

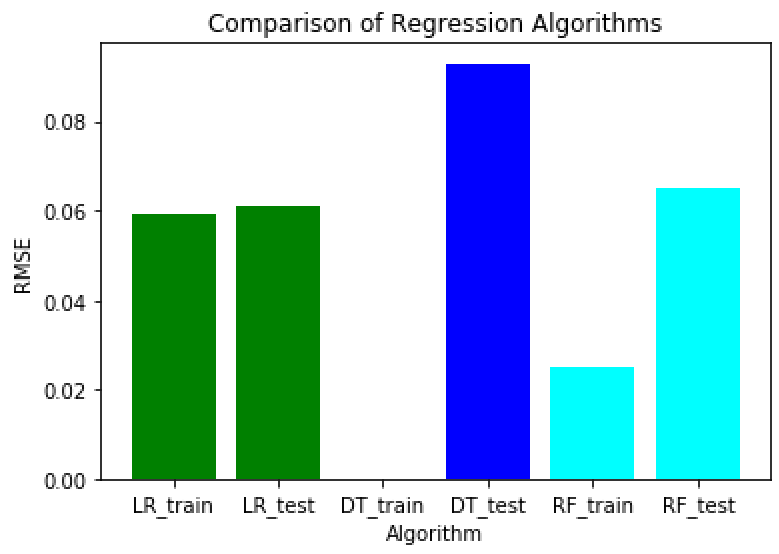
- Logistic Regression (all default setting)

> *LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True, intercept_scaling=1, max_iter=100, multi_class='warn', n_jobs=None, penalty='l2', random_state=None, solver='warn', tol=0.0001, verbose=0, warm_start=False)*

8. Benchmark Results :

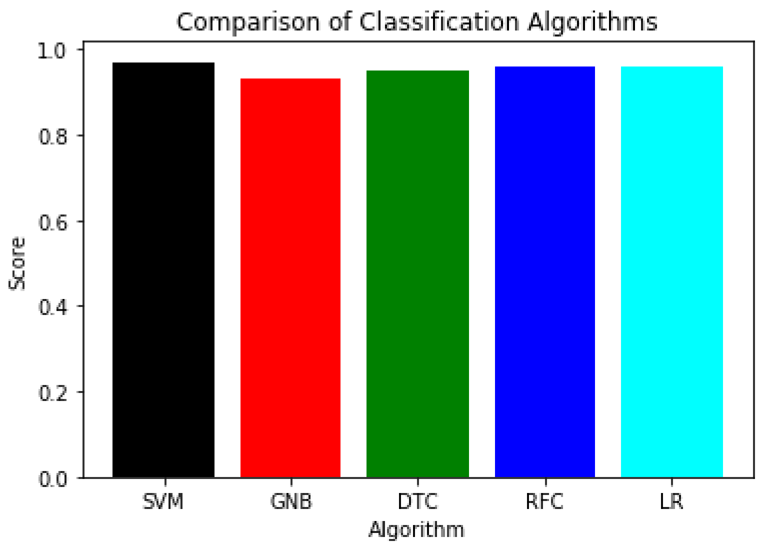
- Regression Algorithms Benchmark Results

Algorithm	RMSE(train)	RMSE(test)
Linear Regression	0.0593	0.0608
Decision Tree	1.6653e-17	0.0929
Random Forest	0.0250	0.0651



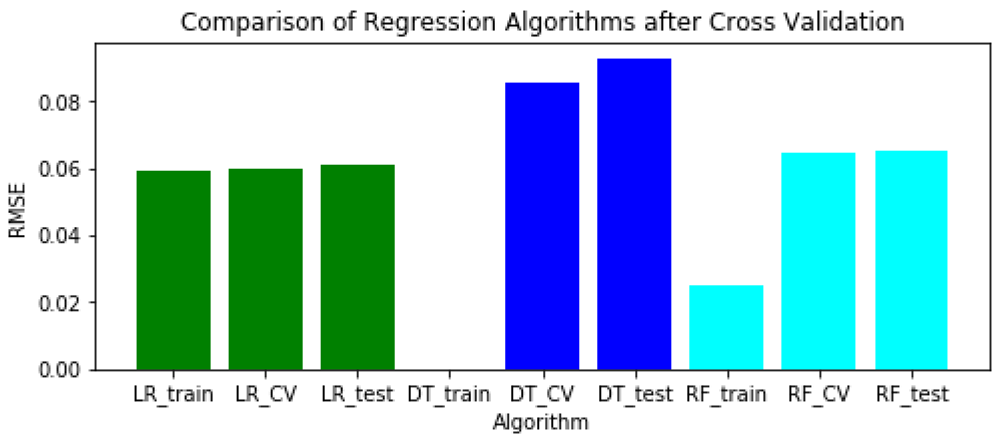
- Classification Algorithms Benchmark Results

Algorithm	Score	F1 Score
SVM	0.97	0.94
Gaussian Naive Bayes	0.93	0.88
Decision Tree Classifier	0.95	0.91
Random Forest Classifier	0.96	0.92
Logistic Regression	0.96	0.92



9. Try cross validation with different validation sets : I divided the original data set into two groups - train, test set - to train(fit)the model with train set and test the model for unseen data. But overfit problem occurred for some models especially it was serious on decision tree algorithm (RMSE for train set was nearly zero!)
   There can be many reasons that this overfitting problem occurs but the small size of data set is major reason, I think.
   To address this, I tried cross validation technique which divides the data set into smaller groups and uses one by one as the validation data set. Scikit learn privides cross_val_score function so I imported it and made 10 subgroups (by setting 'cv' parameter to be 10). I used 'negative mean squared error' as a 'scoring' parameter for cross_val_score function. Scikit learn cross-validation features expect a utility function (greater is beter) rather than a cost function (lower is better), so the scoring function is actually the opposite of the MSE(i.e., a negative value), which is why I used 'negatieve mean squared error' and computed '-scores' before calculating the square root in the code file('capstone.ipynb').
   As cross validation made 10 RMSE values, I calculated mean and standard deviation of them and I repeated this process for three regression algorithms: Linear Regression, Decision Tree, Random Forest.
   The cross validation results are as follows: (RMSE Mean for test data set)

- [Linear Regression] 0.059
- [Decision Tree] 0.085
- [Random Forest] 0.064

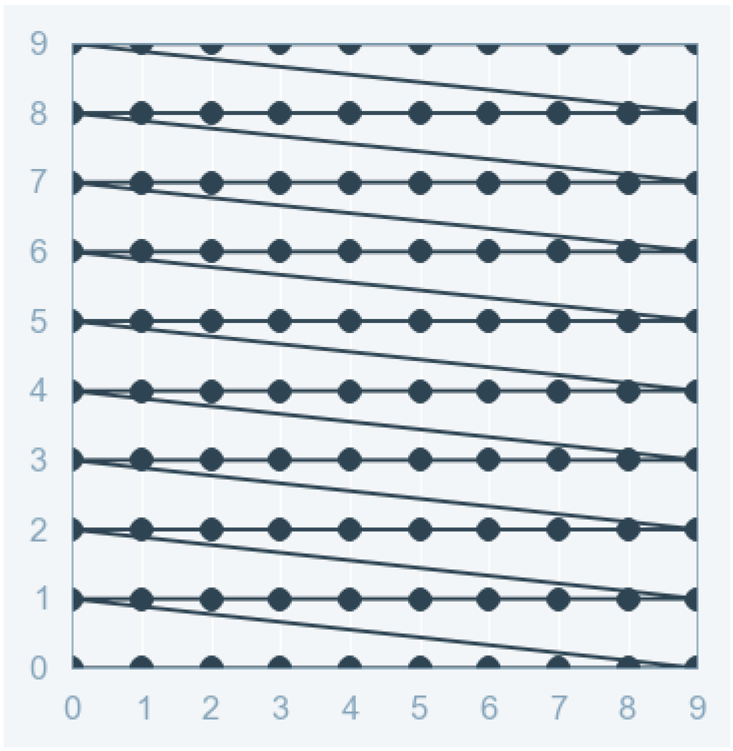


From the above plot we can see that they are considerably improved after cross validation.

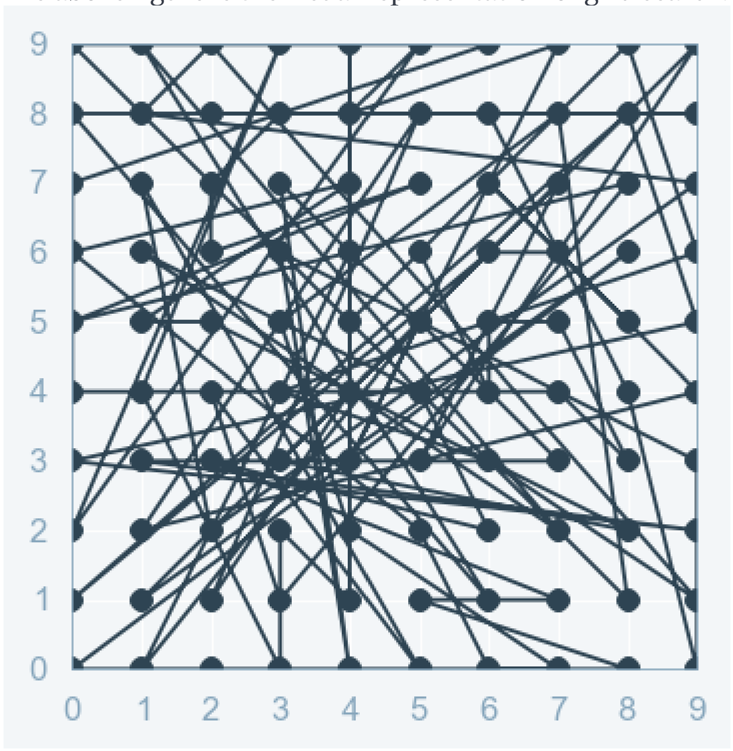
Refinement

I tried to refine the prediction model with hyperparameter tuning. As there is no room for hyperparameter tuning in Linear Regression, I tried with Random Forest algorithm.
To find out the best hyperparameter combination, I used Grid Search of Scikit Learn. And the next, I also tried Randomized Search to compare the result.
At first, let’s take a look at briefly these two methods:
(https://medium.com/@senapati.dipak97/grid-search-vs-random-search-d34c92946318)

Method	**Explanation**
Grid Search	In Grid Search, we try every combination of a preset list of values of the hyper-parameters and evaluate the model for each combination. The pattern followed here is similar to the grid, where all the values are placed in the form of a matrix. Each set of parameters is taken into consideration and the accuracy is noted. Once all the combinations are evaluated, the model with the set of parameters which give the top accuracy is considered to be the best.
Randomized Search	Randomized search is a technique where random combinations of the hyperparameters are used to find the best solution for the built-in model. It tries random combinations of a range of values. To optimize with randomized search, the function is evaluated at some number of random configurations in the parameter space.



The above figure is the visual representation of grid search.



The above figure is the visual representation of randomized search.

To implement grid search, I imported GridSearchCV from Scikit Learn and defined parameter grid first:

```
param_grid = [
# try 9 (3×3) combinations of hyperparameters
{'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6]},
# then try 6 (2×3) combinations with bootstrap set as False
{'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
```

Next, set the hyperparameters(estimator=RandomForestRegressor, cv=5, scoring='neg_mean_squared_error', return_train_score=True, and the others as defaults), and execute grid search.

- cv : This determines the cross-validation splitting strategy.
- return_train_score : If 'False', the cv_results_ attribute will not include training scores.

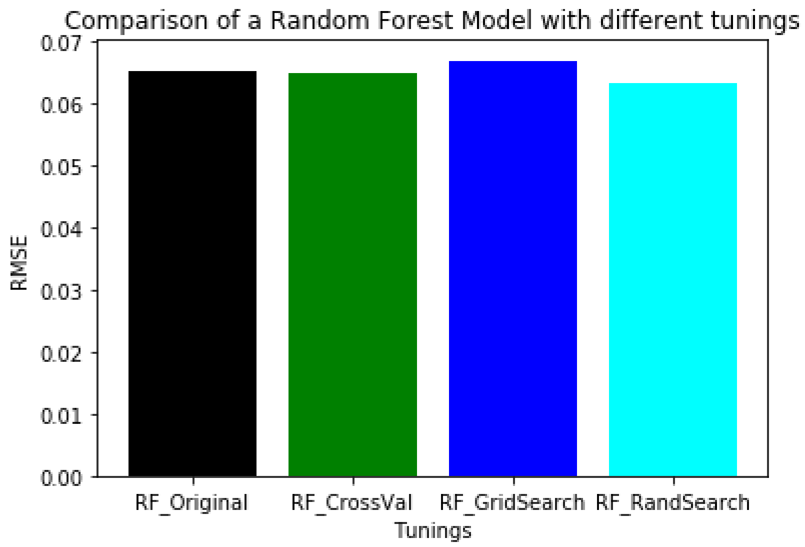
```
forest_reg = RandomForestRegressor(random_state=42)
# train across 5 folds, that's a total of (9+6)*5=75 rounds of training
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
scoring='neg_mean_squared_error', return_train_score=True)
grid_search.fit(x_train, y_train)
```

After finding the best estimator with grid search and applying that to the model, I got 0.06682 RMSE. (little bit worse than random forest model's RMSE after cross validation)

Next, I imported RandomizedSearchCV from Scikit Learn and followed the similar procedure:

```
param_distribs = {
'n_estimators': randint(low=1, high=200),
'max_features': randint(low=1, high=6),
}
forest_reg = RandomForestRegressor(random_state=42)
rnd_search = RandomizedSearchCV(forest_reg, param_distributions=param_distribs,
n_iter=10, cv=5, scoring='neg_mean_squared_error', random_state=42)
rnd_search.fit(x_train, y_train)
```

After finding the best estimator with randomized search and applying that to the model, I got 0.06309 RMSE. (this is the best RMSE so far)
Here is the comparison plotting:



My random forest model after applying randomized search shows best performance.

IV. Results

Model Evaluation and Validation

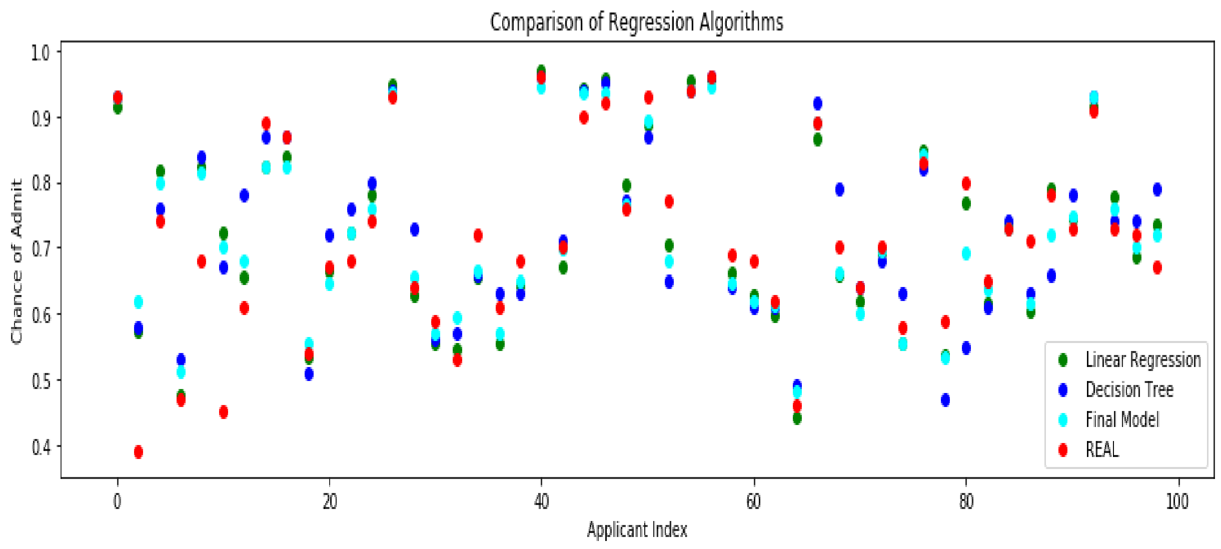
To test my final model, I tried some test data and compared the predicted outputs with labels.(see the source code in ‘capstone.ipynb’)
The part of results are as follows:

Index	Prediction	Real
0	0.93	0.93
10	0.70	0.45
20	0.65	0.67
30	0.57	0.59
40	0.95	0.96
50	0.89	0.93
60	0.62	0.68
70	0.60	0.64
80	0.69	0.80
90	0.75	0.73

My final model seems to work well and is making reasonable outputs for test(unseen) data.

Justification

To compare my final model with benchmarks, I entered 50 test data into the benchmark models as well as my final model and compared the results. (see the source code in ‘capstone.ipynb’)
I attached the resulting plot here:

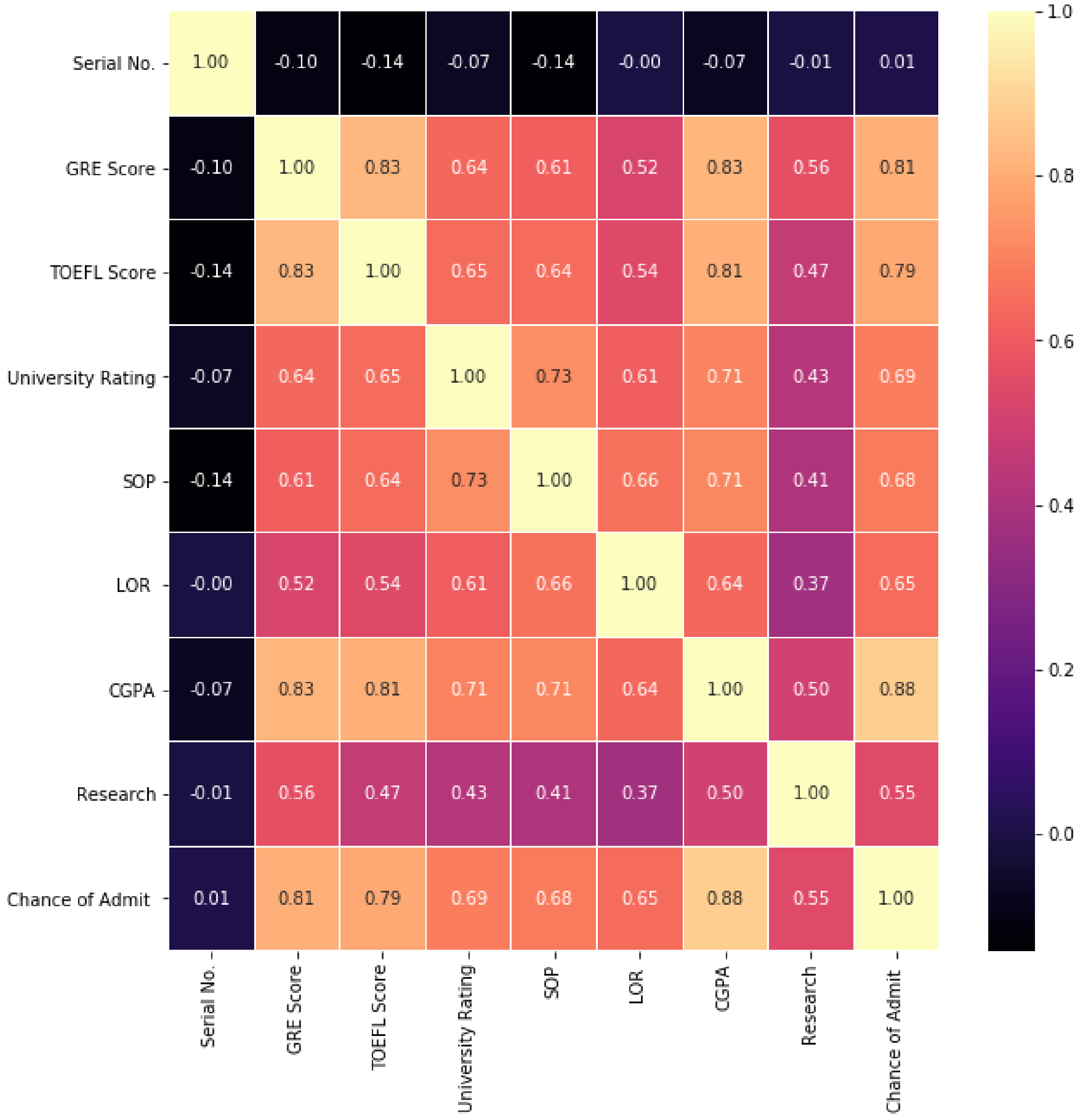


I plotted 3 predictions (Linear Regression, Decision Tree, My Final Model) and real values(labels) for 50 test data.(Index 0, 2, 4, ... , 98)
With some exception, my final model's prediction is mostly closer to the real value or at least nearly same as the other model's predictions.

V. Conclusion

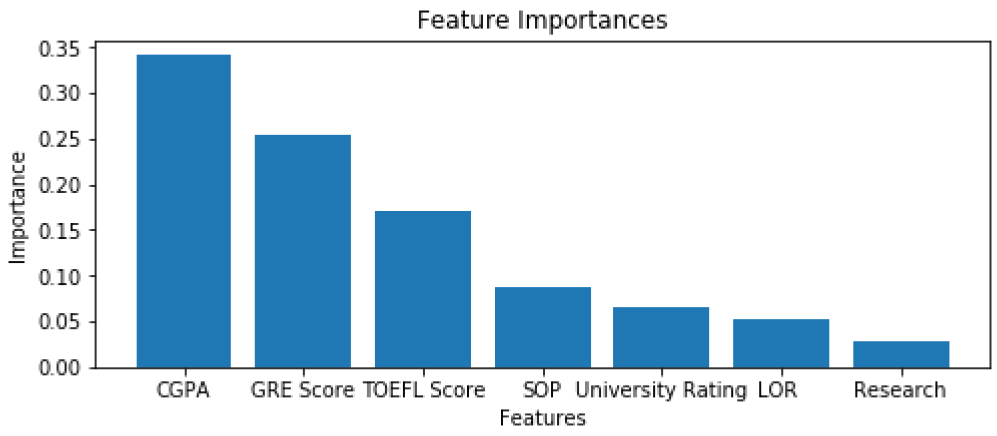
Free-Form Visualization

To see feature importance, plotting heatmap is another good choice. Heatmap shows us the correlations among features, so we can determine the most important features by seeing the correlations between the target("Chance of Admit") and the other features(input variables)



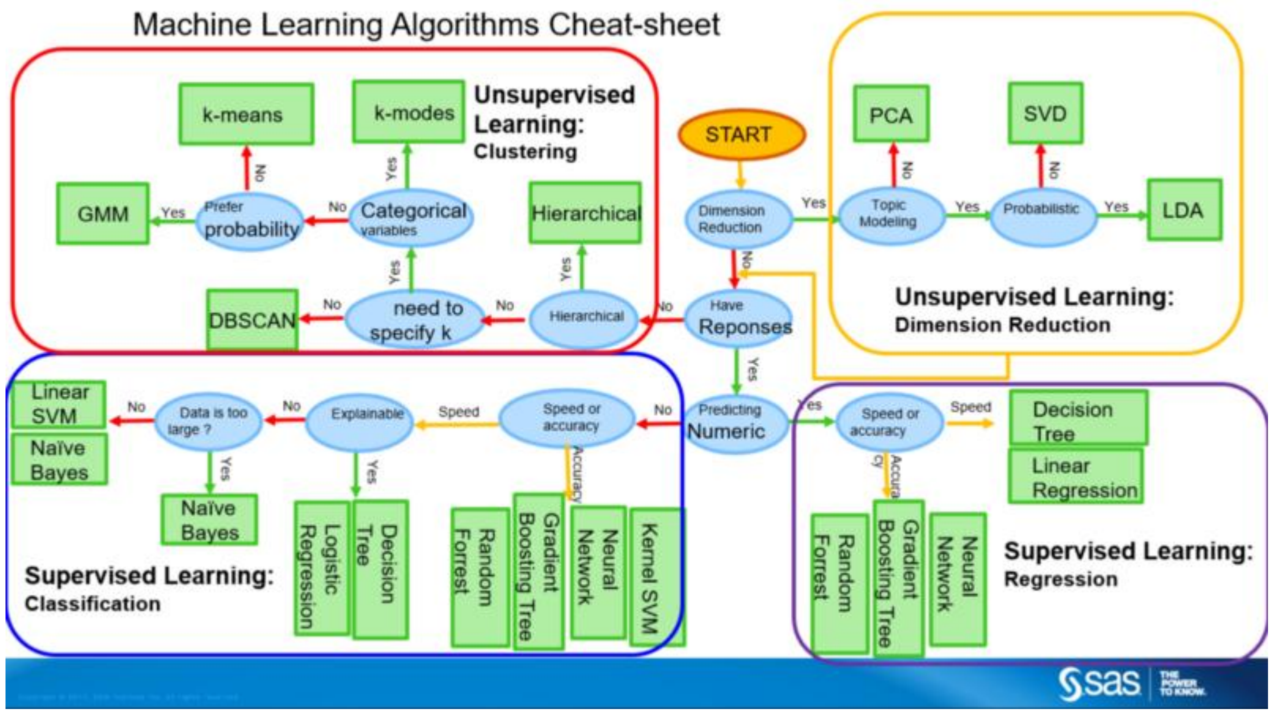
	Serial No.	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research	Chance of Admit
Serial No.	1.00	-0.10	-0.14	-0.07	-0.14	-0.00	-0.07	-0.01	0.01
GRE Score	-0.10	1.00	0.83	0.64	0.61	0.52	0.83	0.56	0.81
TOEFL Score	-0.14	0.83	1.00	0.65	0.64	0.54	0.81	0.47	0.79
University Rating	-0.07	0.64	0.65	1.00	0.73	0.61	0.71	0.43	0.69
SOP	-0.14	0.61	0.64	0.73	1.00	0.66	0.71	0.41	0.68
LOR	-0.00	0.52	0.54	0.61	0.66	1.00	0.64	0.37	0.65
CGPA	-0.07	0.83	0.81	0.71	0.71	0.64	1.00	0.50	0.88
Research	-0.01	0.56	0.47	0.43	0.41	0.37	0.50	1.00	0.55
Chance of Admit	0.01	0.81	0.79	0.69	0.68	0.65	0.88	0.55	1.00

From the heatmap, we can see the top three important variables which are highly related to the target variable (Chance of Admit) are CGPA, GRE Score and TOEFL Score, which coincides with what we saw in the 'Exploratory Visualization' section above.
For a comparison, let's plot the feature importances which we got from 'randomized search'. (see the source code in 'capstone.ipynb')



We can see the same results with heatmap above.

Next, I want to introduce a pretty useful visualization that classifies machine learning algorithms as well as selection process very well: (from blogs.sas.com)
When we want to solve a real-world problem with machine learning, we can frequently come across ambiguous situations that we can't be sure which algorithm we should apply.
In that case, this visualization can be a useful guidance.



Reflection

In this project I made a prediction model to predict the 'chance of admit' for a graduate school with some input features including CGPA, GRE score, TOEFL score and so on.
To make the model I benchmarked many algorithms in regression as well as classification models.
For a metric I used RMSE (Root Mean Square Error) and used it to meaure the performance of algorithms.
One interesting thing occurred in fitting the train data with Decision Tree algorithm. The metric - RMSE - was nearly zero! At first, I thought there must be some calculation error or parameter setting error, but I came to know it was significant overfitting. To address this overfitting

problem, I used cross validation which divides the original data set into small groups and use them one by one as validation data. After cross validation I could get rid of overfitting as well as get more improved score.
To further refine my model, I tried hyperparameter tuning with built-in functions of scikit learn. Those are grid search and randomized search.
After implementation of those two, as randomized search gave me the better score, I applied the best estimators (which the randomized search found) to my model which led to my final model.

Improvement

The target variable I used in this project was the interviewees' reply (how much possibility(0% ~ 100% range) they expect they got the admission acceptance from the graduate school with there track records (i.e. the features including CGPA, GRE, TOEFL, and so on))
In other words, the label in this supervised learning problem was not the real results but the interviewees' expectations.
If I can collect the sufficiently many real acceptance data (Yes or No), and use them as labels I can make more useful prediction model.
From the implementation algorithm point of view, there are several ways to improve my prediction model, for example, another pre-processing techniques or different models. Among them I tried XGBoost. Bagging and Boosting is one of good options used to get better performance in machine learning. So with XGBoost I can expect to get improved model.
XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework.
XGBoost is being used frequently by Kaggle's top rankers because it has outstanding advantages as follows:

- It implemented regularization, so it helps to reduce overfitting.
- It implements parallel processing so is far faster as compared to GBM.
- It has an built-in routine to handle missing values.
- It makes splits upto the max_depth specified and then start pruning the tree backwards and remove splits beyond which there is no positive gain. Typical GBM would stop splitting a node when it encounters a negative loss in the split.
- It allows user to run a cross-validation at each iteration of the boosting process and thus it is easy to get the exact optimum number of boosting iterations in a single run.
- Users can start training an XGBoost model from its last iteration of previous run.

To implement XGBoost, I installed the package and imported it to my ipynb source code.

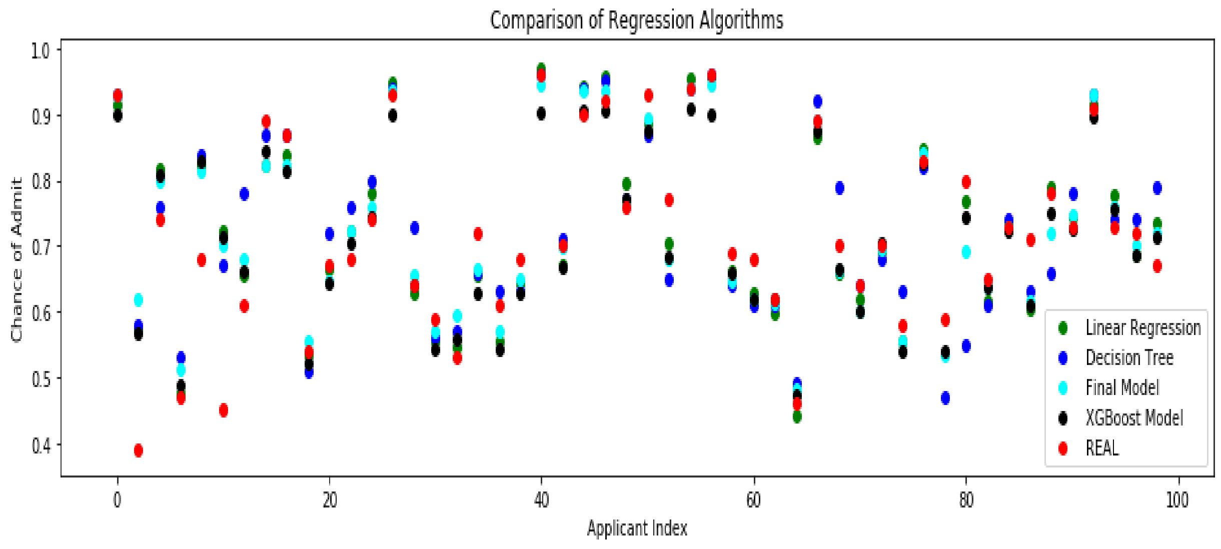
> *import xgboost*
>
> *xgb = xgboost.XGBRegressor(n_estimators=100, learning_rate=0.03, gamma=0.005, subsample=0.5, colsample_bytree=0.9, max_depth=2, random_state=42)*

Let's take a look at the parameters briefly:

- gamma : A node is split only when the resulting split gives a positive reduction in the loss function. Gamma specifies the minimum loss reduction required to make a split. (default = 0)
- subsample : Same as the subsample of GBM. Denotes the fraction of observations to be randomly samples for each tree. (default = 1)
- colsample_bytree : Denotes the subsample ratio of columns for each split, in each level. (default = 1)
- max_depth : The maximum depth of a tree. (default = 6)

After changing these parameters several times on trial-and-error basis to get better result, I came to get the RMSE 0.06258, which is the best score so far.
To use the tuning method like GridSearchCV will be smarter way to tune the parameters.

To check the improvement, I plotted the comparison chart again with the same 50 test data:



From the above plot, we can see the improvement after applying XGBoost.