STAT 844/CM 764:
Statistical Learning – Function Estimation

Final Project Report:
Data Analysis on Google Play Store Apps

Presented by:
Si Ao Chen

Introduction

Nowadays, Google play store is gaining more popularity with an increasing demand on the smartphones in the market. As a result, consumers spend a significant amount of time on using different kinds of applications on their electronic gadgets, and application creators are interested in knowing the driven factors that make an application successful. Hence, using the dataset containing detailed information on thousands of applications, it would be very useful to analyze and predict the application rating in order to find successful products.

This report will be divided in several sections. First, we will examine the details of the Google Play Store data obtained from Kaggle. Then, we will discuss some of the preprocessing methods applied to this dataset, after which we will look into data analysis and data prediction using the three main methods learned in this course, namely the smoothing method, the random forest and different boosting methods. Furthermore, I will present the performance comparison between the models presented in this project. Moreover, we will interpret the findings in the context of this project and draw conclusions for potential real business to understand. Finally, I will present some aspects that could be improved and introduce some other machine learning models that could be used if more time is allocated. The programming portion of this project is attached to Appendix.

Data

The dataset is obtained on Kaggle, where the author scraped it from Google Play Store. The source is as follows:
https://www.kaggle.com/lava18/google-play-store-apps

The dataset has size 10841 entries and 13 variables, where each row represents one application with 13 features, namely:

Categorical variables:

- Category
- Type
- Content.Rating
- Genres

Continuous variables:

- App
- Rating (response)
- Reviews
- Size
- Installs
- Price
- Last.Updated
- Current.Ver
- Android.Ver

Our goal is to predict the Rating of each application ranged from 0 to 5 using the rest of the features. Hence, it is important to perform some feature visualization to grasp a general idea of the data. Hence, we will examine the distribution of the variables to discover some interesting facts.

The variables plotted below are distributions of three continuous variables (i.e. *Installs*, *Reviews*, *Size*) and the response variable *Rating*, where *Installs* and *Reviews* indicate respectively the total amount of installs and reviews for the applications.

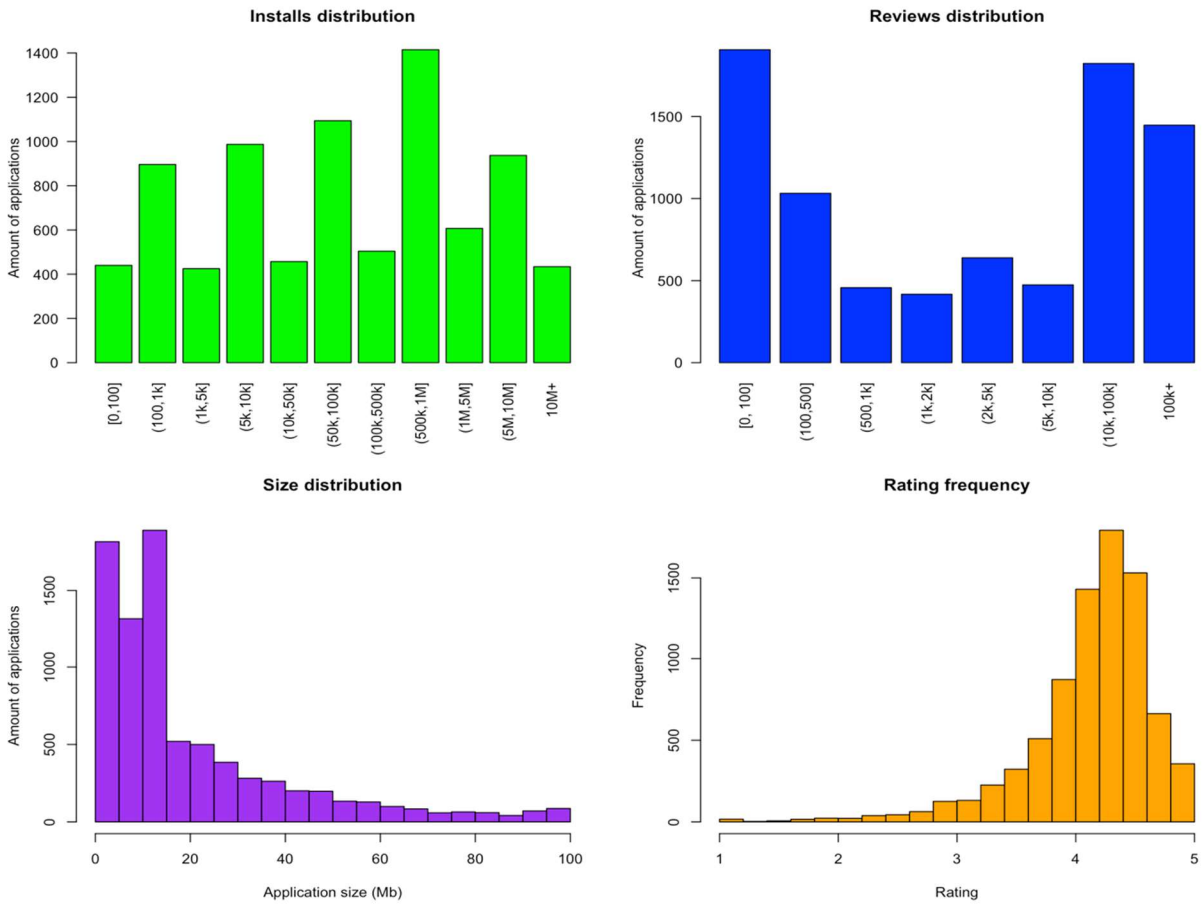

Figure 1 Visualizations of Installs, Reviews, Size and Rating distributions

As we can observed, most ratings given to applications are quite positive, i.e. between rating 4 and 5. However, the *Rating* distribution is heavily left-skewed, hence it is needed to transform it during the modeling phase, and we will use log-transformation to normalize it. Due to rounding effect from the data, most applications have number of downloads rounded to the highest place value, and we can see that a large portion of applications have number of downloads close to a million. On the other hand, we can see in the *Review* distribution that a large portion of applications have either very few numbers of reviews, i.e. between 0 and 100,

or very high volume of reviews of more than 10 thousand reviews, which means that the opinions to applications are polarized. In addition, we can examine the application size distribution heavily right-skewed, which indicates that most of the applications have small sizes. Since *Installs* and *Reviews* have large values and they may impact greatly the response variable, we wish to log-transform it as well. *Size* feature also needs to be log-transformed as it is right-skewed.

Another three features (*Last.Updated*, *Content.Rating* and *Curr.ver)* are visualized in Figure 2 below:

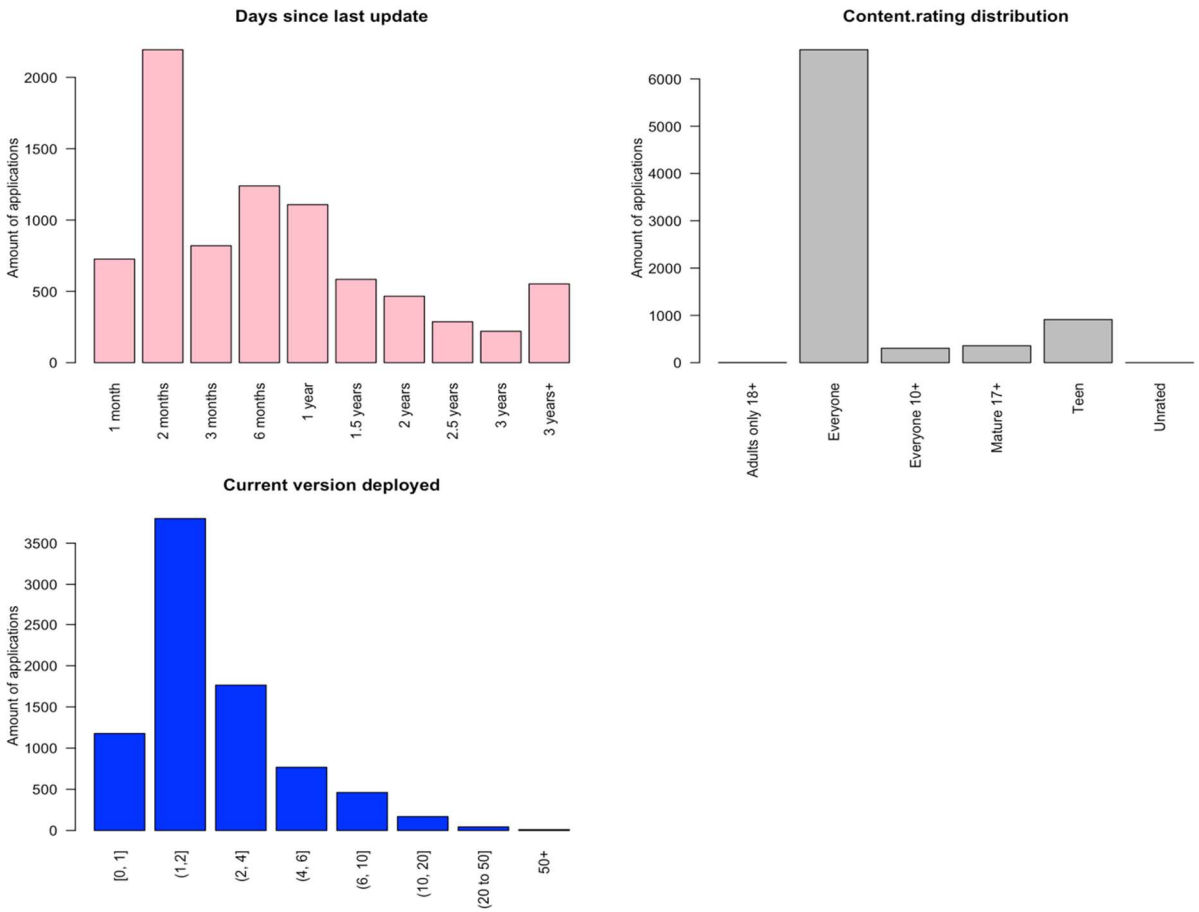

Figure 2 Visualizations of days since last update, Content.rating and Curr.ver distributions

We can observe in the first plot that most applications updates within half of a year, and a large proportion of them updates within two months, which is quite frequent. The second plot shows that most of applications are suited for everyone, and very few applications in the dataset are developed in a way that only mature users are appropriate to use it. In the last graph, we can see that most applications in the dataset are of version from 1 to 4, which means that most applications are still in their early development.

The distributions of applications type and Android minimum required versions are shown in pie chart in Figure 3 below:

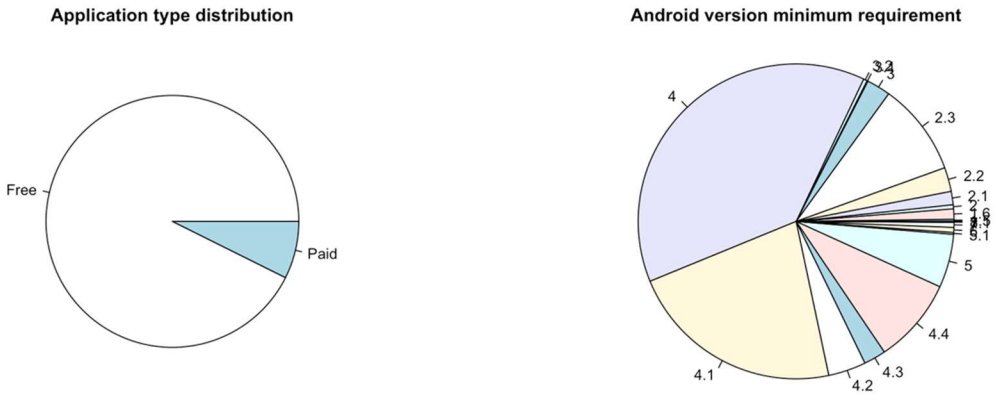

Figure 3 Distributions of application type and minimum Android version required

In the applications type pie chart, we can observe that most applications are free, where main source of profits could be in-app purchase and implemented ads in the applications. In the second graph, even though many categories are of small proportions, and some of them have their labels tangled together, but the main takeaway is that most applications requires at least Android version 4 or higher. It means that with more advanced technologies programmed in the applications, most of them cannot run properly with older phones that are on the edge of elimination.

Moreover, we can visualize the distribution of the applications based on their categories, as shown in the figure below:

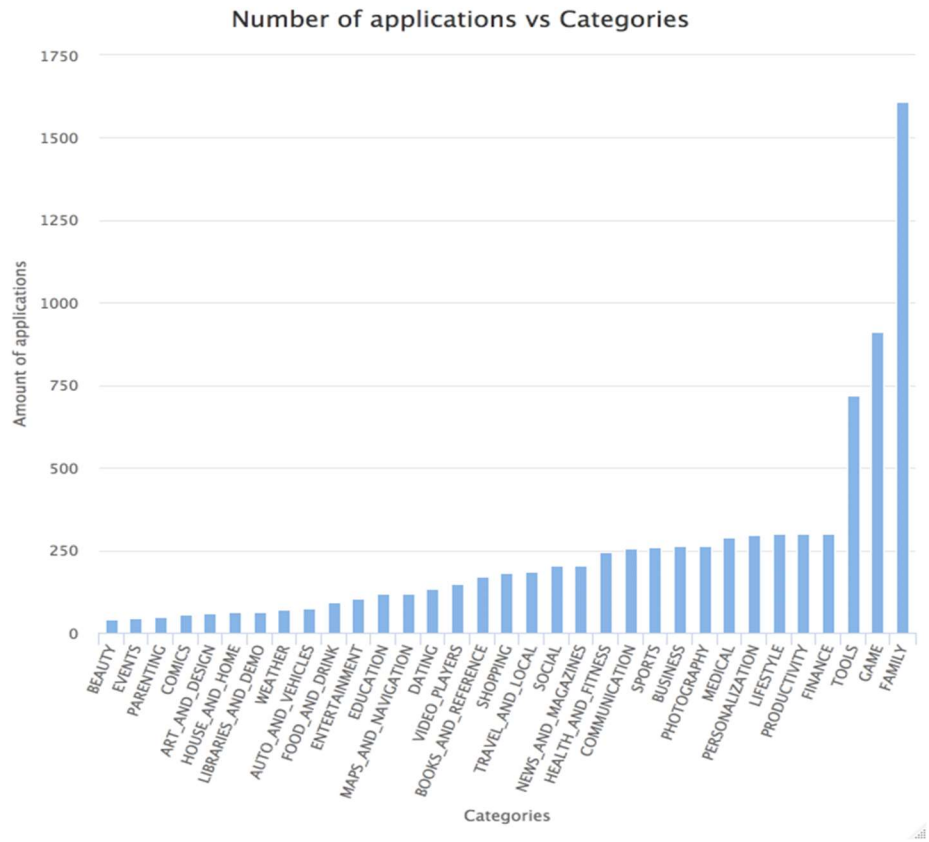

Figure 4 Categories distribution

Figure 4 indicates that software developers tend to create applications in the Family, Game and Tools categories. A possible explanation why Family category is the most popular one is that it is suited for the whole family, i.e. children, teenagers and parents. Hence it is more likely to make a profit. Tools category is popular for the similar reasoning as it is appropriate to all ages. Game is another popular category because players can easily spend some money to have a better in-game experience.

Let us see the relationship between some explanatory variables and the response variable. Based on the plots above, features such as Rating, Installs and Reviews are skewed, hence we will normalize it by applying log-transformation to each variable in the plots shown below:

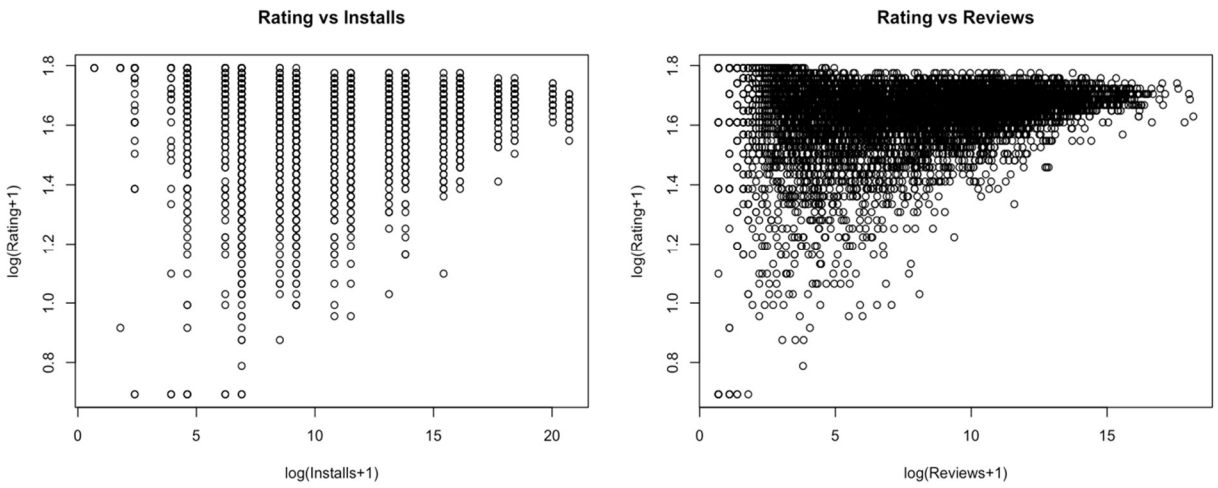

Figure 5 Relationship between Installs and Reviews with Rating after log-transformation

We can observe that a general trend is that the more Installs and Reviews an application has, the more likely that its rating is high. We can conduct a feature engineering to create a new feature called RVI, which stands for Rating value indicator:

$$RVI = \log(Reviews + 1) * \log(Installs + 1)$$

The importance of this new feature will be discussed later in the report.

Preprocessing

In this dataset, there are several issues that we must fix before doing any exploratory data analysis. The first step is to remove any entries that need to be deleted. For instance, there is one entry where one feature is missing, and all other features were left-shifted by one column. Since it occurred only in one entry, it was decided to remove this entry completely. Moreover, there are roughly 10% of entries have duplicated application name, however based on the nature of the dataset, it is assumed that each entry represents a unique application. Thus, it is necessary to remove these duplicated entries using the *dplyr* library.

Then, it is time to perform some data manipulations. The table below shows some values of the column we need to modify:

Table 1 Examples of column values in the original dataset

Size	Installs	Price	Last.Updated	Current.Ver	Android.Ver
19M	10,000+	0	July 31, 2018	1.0.0	4.0.3 and up
2.6M	500,000+	0	May 4, 2016	3.6.0.115_FN	2.3 and up

6.8M	100,000+	$4.99	March 25, 2018	1.5.2	4.0 and up
Varies with device	100,000,000+	0	July 31, 2018	47.1.2249.129326	Varies with device
556k	10,000+	0	July 11, 2014	1.1	2.2 and up

As we can see, the Last.Updated feature is originally a date type, but we want it to indicate the number of days since the most recent update of the application so that we can convert it into continuous variables. In order to do so, we set the current date to September 1, 2018, which is around the date when the data was scraped, and we used the library *lubridate* to calculate the day difference between each date in the Last.Updated column with the current date.

Moreover, we can observe that features such as Size, Installs, Price, Current.Ver and Android.Ver contain non-numeric characters, however we want to interpret them as numeric columns. It is worth noting that some of the entries in these features are purely strings because they have values as "Varies with device", hence we identify these entries and transform all of them to *NA* because the information is not available. Then, for size column, we want to remove the unit, i.e. M for Megabyte and k for Kilobyte, and scale properly entries with unit as Kilobyte to Megabyte. The Price and Installs columns have also similar issues with Size, except that the non-numeric character is dollar sign for Price, comma and plus sign for Installs. We will remove them as well to make these features purely numeric. For Current.Ver and Android.Ver features, we wish to treat them as decimal numbers for modeling. In other words, we will trim all the non-numeric characters and everything after the second dot of a version. For instance, given an Android.Ver value to be "4.1.2 and up", we wish to transform it into 4.1.

After conducting the data preprocessing above, we are left with a certain portion of missing values in our dataset. In fact, four columns contain NA values: Rating, Size, Current.Ver and Android.Ver. It is important to realize that we cannot use entries with missing response variables to do prediction nor validation, hence we will drop all the entries with NA Rating values. Regarding the other three columns, we will replace the missing values with median of the corresponding columns. Thus, after the imputation, the example above will become as follows:

Table 2 Processed result for the same data as in the previous table

Size	Installs	Price	Last.Updated	Current.Ver	Android.Ver
19	10000	0	32	1	4
2.6	500000	0	850	3.6	2.3

6.8	100000	4.99	160	1.5	4
13	100000000	0	32	47.1	4
0.556	10000	0	1513	1.1	2.2

Lastly, as discussed in the Data section, *Rating, Installs, Reviews* and *Size* features will be log-transformed for future analysis.

Now, all data preprocessing is completed, we will start with data modeling using the smoothing method, random forests and boosting methods.

Smoothing methods

In this dataset, we have more than one explanatory variate, so the first model we can try is K nearest neighbor fitting. The plots below show the different fitted function with different parameter value k used in the nearest neighbor fitting with RVI as the sole predictor:

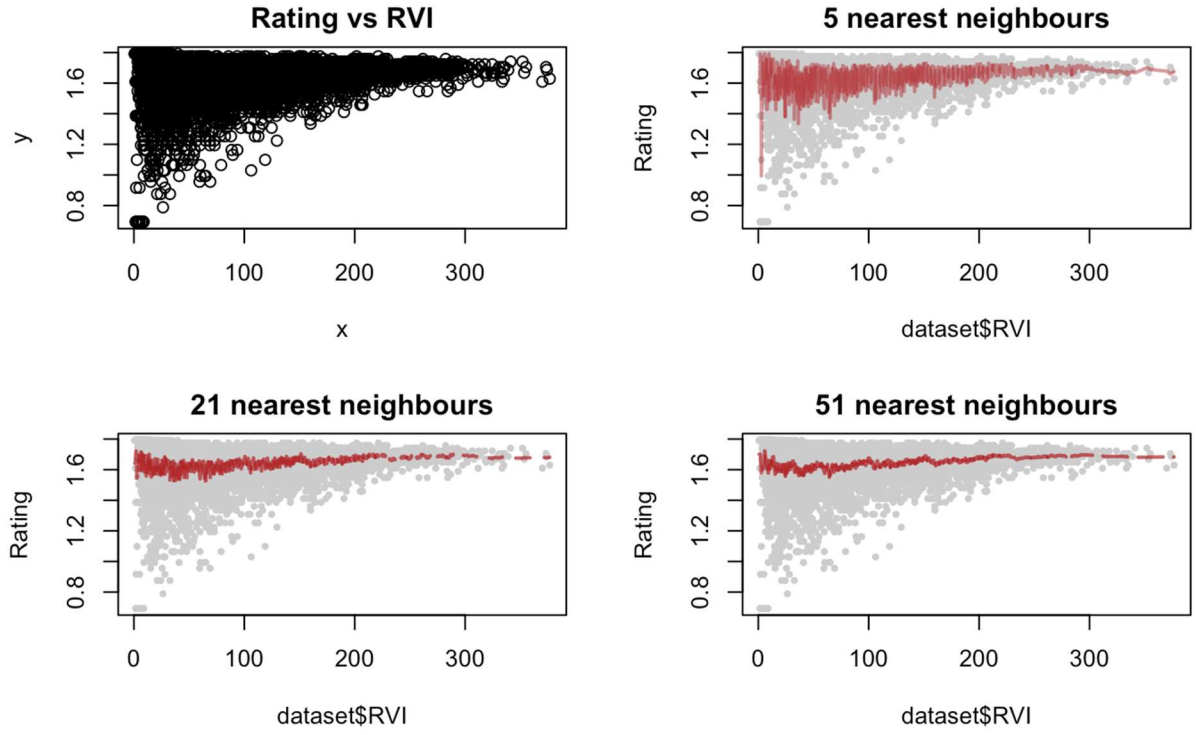

Figure 6 K-nearest neighbor fitting with various k

We can observe that the larger the size of neighbors becomes, the smoother becomes the fitted function.

The locally weighted sums of squares (loess) method is also used to assess this dataset. In particular, the built-in loess function in R and the function presented in this course that minimizes loess are both assessed:

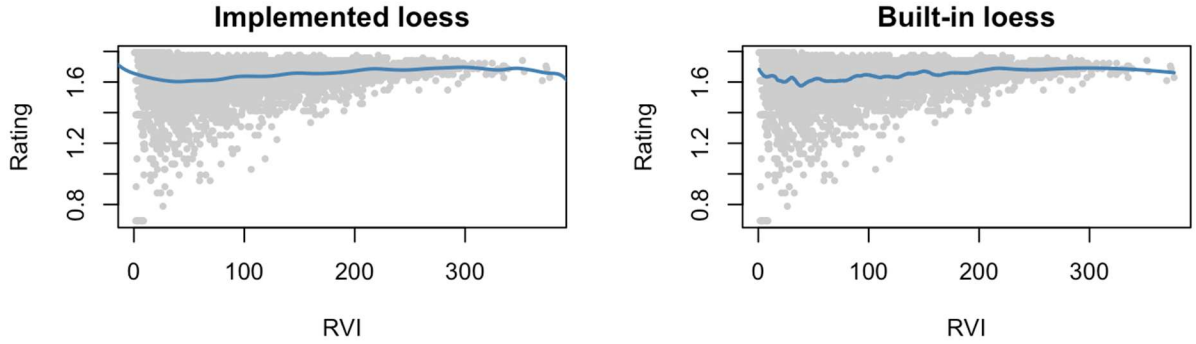

Figure 7 loess methods comparison

Under the same span value 0.1, we can see that the left loess method implemented in this course is smoother while the built-in loess method on the right fluctuates more, so the right one is preferred. In general, lower span value would capture more the trend of the data, while higher span can potentially make the fitted line smoother. To visualize this effect the figure below is the loess method with span value 0.2:

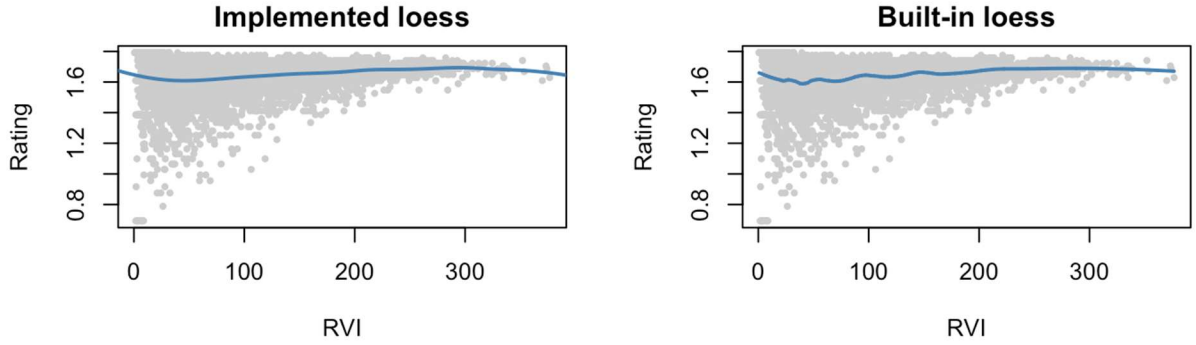

Figure 8 loess method comparison with span = 0.2

Since the dataset discussed in this report is quite large, a lower span value is preferred to have a better explanation of the trend, hence span = 0.1 is used to fit the data.

Cross validation on loess

5-fold cross validation is used to assess the prediction error of the loess method with span = 0.1. Please refer to Appendix to see the implementation of the 5-fold cross validation. The table below shows the average prediction error:

Table 3 loess method prediction error

MSE	RMSE	RMSLE
0.01355987	0.1132439	0.04618312

We can see that the prediction is not as bad as we expected, which means that the predictor is very useful in predicting the response variable.

Generalized additive model

The generalized additive model (GAM) is an effective method to fit the response variable with more than one explanatory variable. In particular, 3D scatter plot is also possible using two of the most important predictors. In this dataset, *Installs* and *Reviews* will be used as predictors to plot the 3D scatter plot. Moreover, it is possible to add some interaction terms, and the differences are shown on the right below:

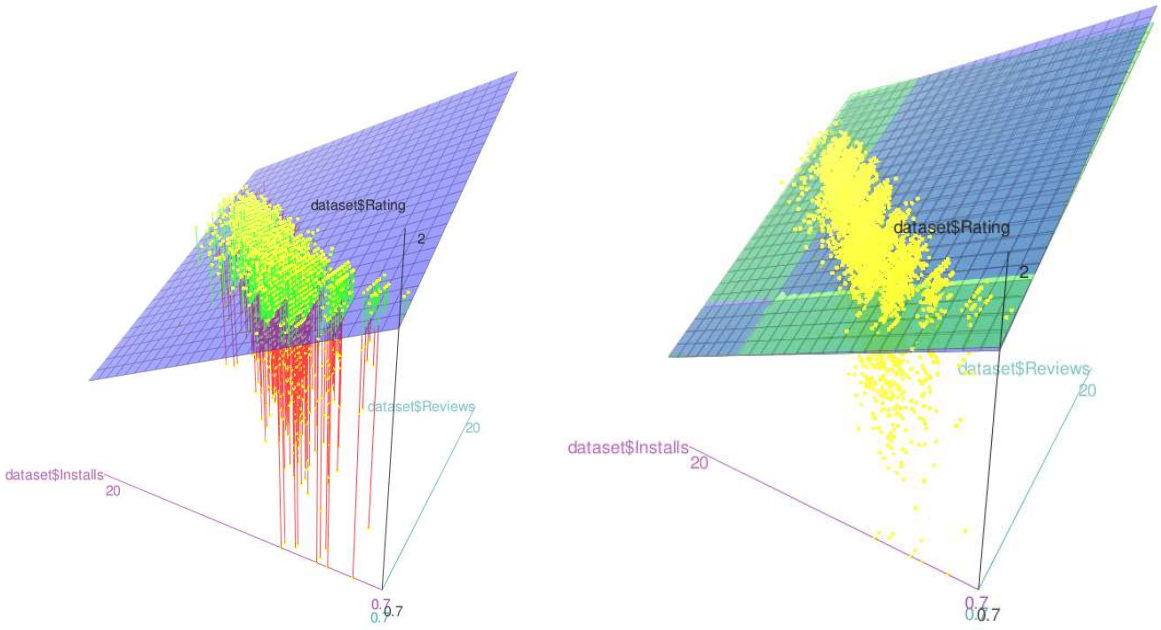

Figure 9 3D scatter plot with RVI and Last.Updated predictors

The left plot above is the normal linear 3D plot. On the right, the blue one is linear, and the green one is with an interaction. The difference in the plot shows the additivity effect in linear terms.

It is also possible to add quadratic terms to the plot, Figure 10 presented below shows the different between additive and non-additive effects in quadratic terms:

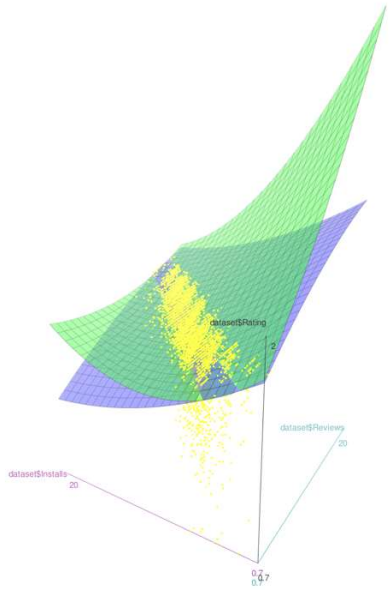

Figure 10 Quadratic additivity effect

where the blue one is the additive quadratic plot, and the green one is the non-additive quadratic plot. The differences in the two methods are mainly places where no points exist.

Using the function scatter 3D, many other methods can be visualized as well. For example, the following plots are the comparison between the thin-plate and the loess method with degree of freedom = 30, as well as the comparison between the additive model and the thin-plate with degree of freedom 15 and 30, respectively:

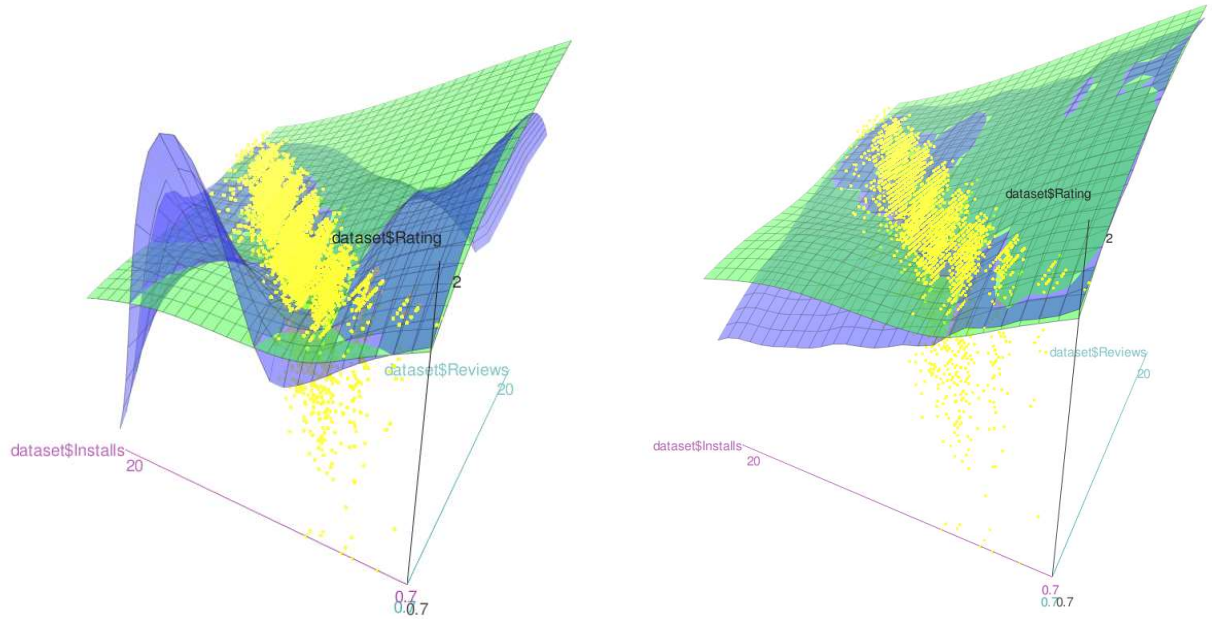

Figure 11 On the left: Comparison of loess(blue) and thin-plate(green) spline with df=30; On the right: Comparison of additive model(blue) and thin-plate(green) spline with df=30

We can see that the major differences between in these two plots are at the corners where there exist fewer points or no point at all. Moreover, loess method expresses more fluctuation at the corners. Also, it is worth noting that the high flexibility of the fit is due to a high value of the degree of freedom.

Variables interactions and model selection

The function *gam* will be used to assess the variables interaction as well as selecting the best gam model for the dataset. Please see Appendix for the details.

The follow plots showcase the interaction of different continuous predictors:

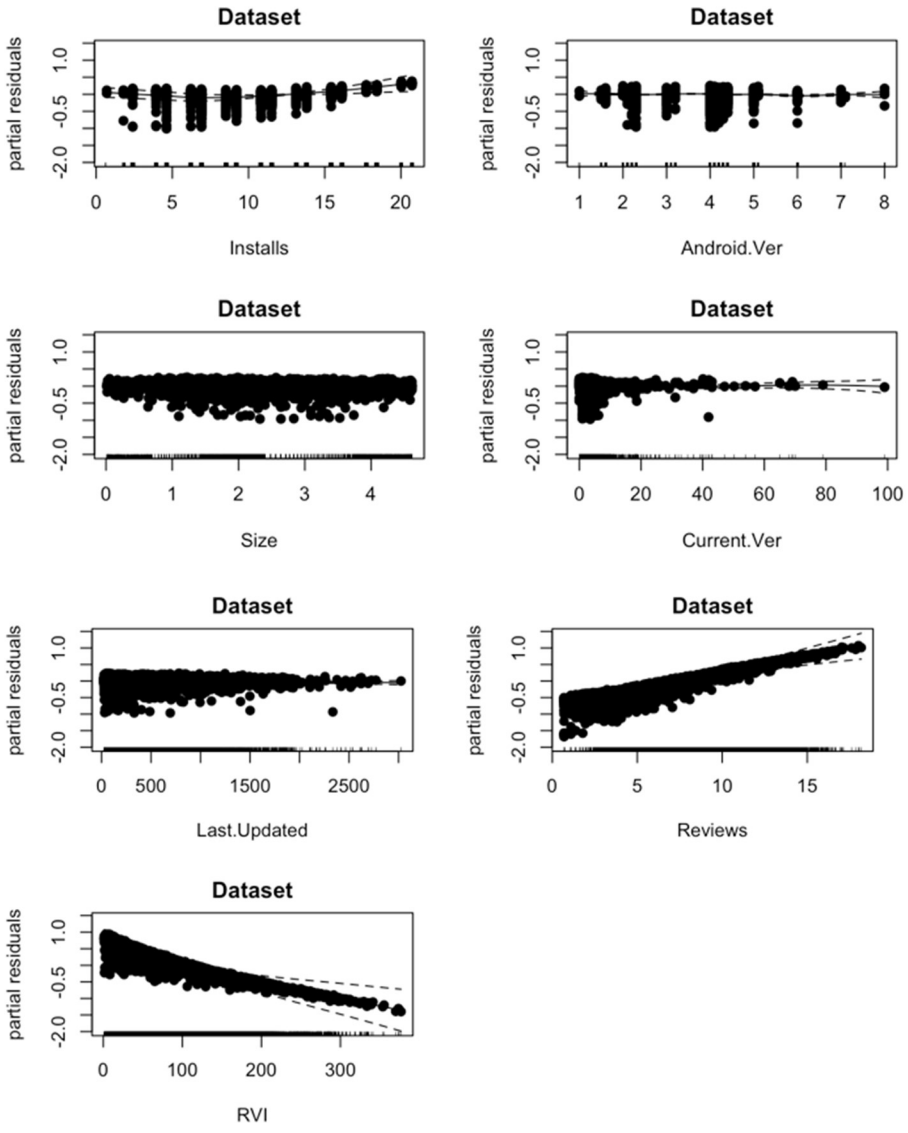

Figure 12 Variables interactions

We can see that the smoothed *RVI* and *Reviews* have an excellent trend.

Using 5-fold cross-validation, the summary of the best GAM model is shown below:

MSE	RMSE	RMSLE	GCV	Time
0.01587419	0.1259928	0.05219048	0.012196	6.31 sec

Random forests

Random forest is a powerful machine learning method for regression. When using it for regression purposes, the general idea of a random forest model is that the model uses a group of decision trees during the training phase, after which it gathers the result of each weak learners and output a final mean prediction. If we kept efficiency in mind, it is important to choose an optimal number of decision trees used in the model. The default setting is to use 500 trees, which will be extremely slow to run, and it is very likely to overfit the model. Hence, we will plot our random forest model to visualize the best number of trees to use, as shown in the figure 13 below.

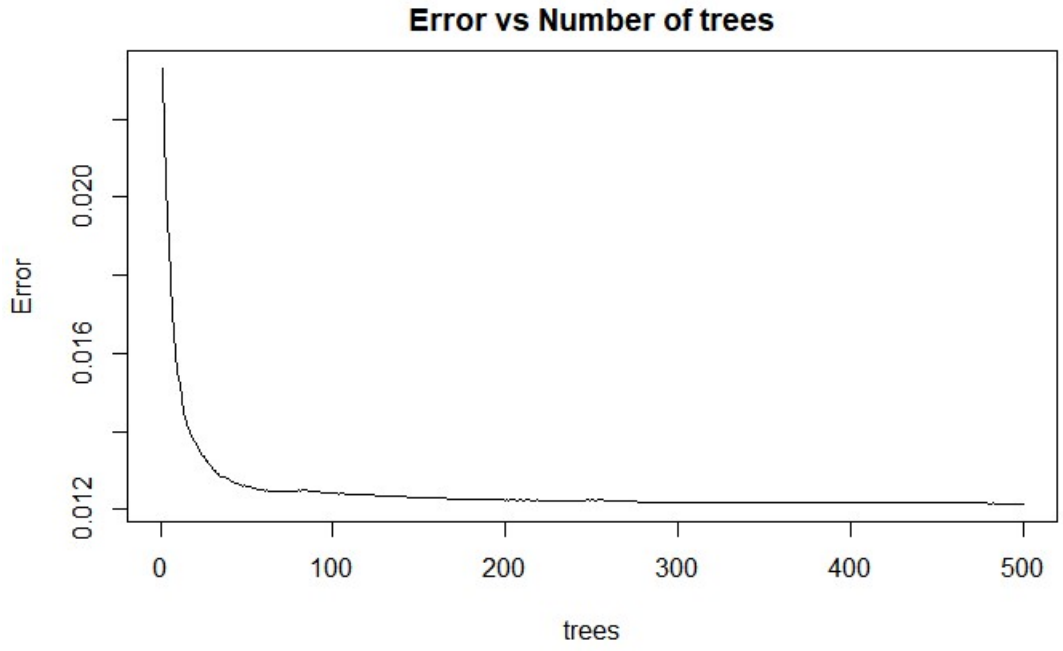

Figure 13 Performance of using different number of trees in random forest

As we can observe, even though the line is decreasing when trees = 500, the line hits a plateau after around 100. Since the performance does not increase drastically after trees = 100, and we wish to decrease the training time, then we decide that trees = 100 would be the optimal choice.

The measure on the importance of variates can be done with two methods. The first method calculates the drop of residual sum of squares (RSS) when the corresponding variate is dropped (type 2 measure), whereas the second method is a native random forest method that calculate the average decrease in model prediction accuracy when the corresponding variate is dropped (type 1 measure). The plot on the variate importance shown in Figure 15 is helpful to visualize these two types of measures of importance, where the left one is the type 1 measure and the right one is the type 2 measure.

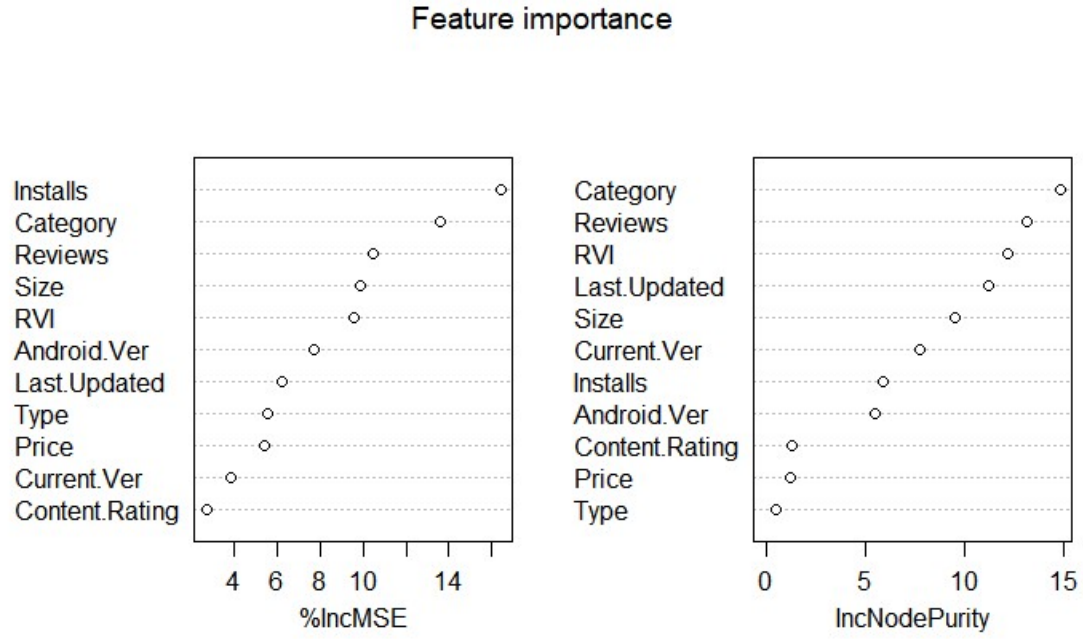

Figure 14 Feature importance based on two measures

We can see that the newly created feature *RVI* is indeed very useful and important in the prediction. Hence, it is an interesting question of how many variables we should choose to do the modeling. 5-fold validation will be used to determine the answer:

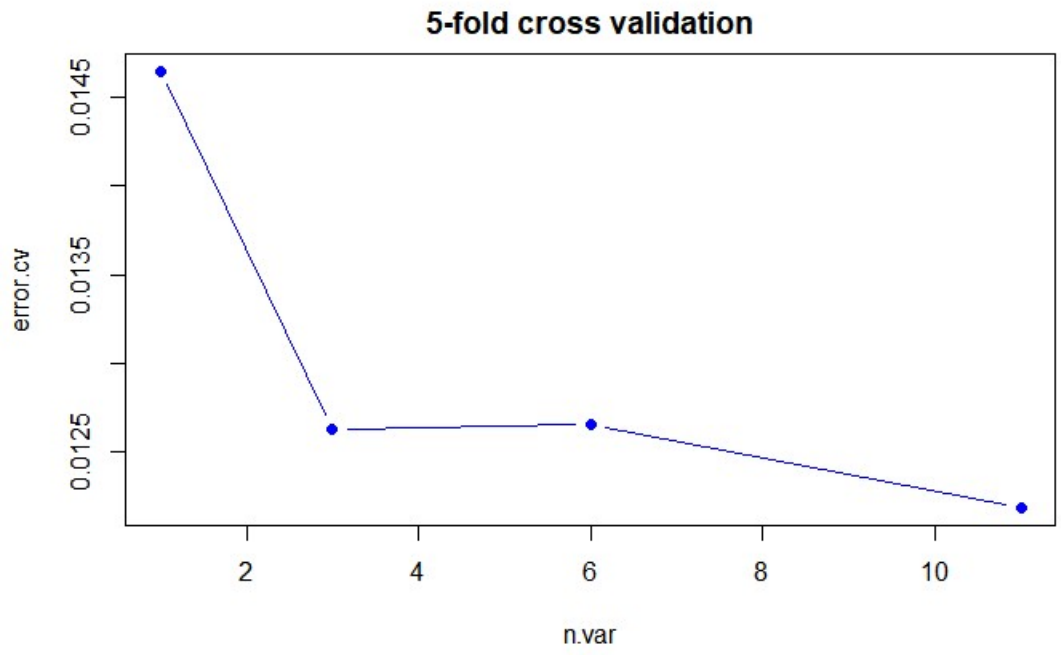

Figure 15 5-fold cross validation

The plot indicates that we should use all the 11 variables to achieve the lowest prediction error.

When training our data, we will perform random forest with both mtry = p (i.e. we are using all the variates as candidates in each split) and mtry = p/3. It is important to note that Genres feature have more than 100 categories, which is a problem to the random forest model. Also, it is very similar to Category feature, so it is decided to drop this feature for future trainings. Then, we have in total 11 predictors in this dataset due to reduction of Genres feature, App ID column and the continuous response variable.
Table below shows a few predicted outputs and their corresponding actual results:

Table 4 Comparison of some predicted outputs and the actual results

Predicted outputs	1.675768	1.652469	1.638166	1.670574	1.675251
Actual values	1.629241	1.686399	1.648659	1.686399	1.686399

Table below summarizes the difference in performance of using bagging (i.e. mtry=p) and uncorrelated random forest (i.e. mtry = $11/3 \approx 4$):

Table 5 Performance difference between bagging and uncorrelated random forest

Type	**MSE**	**RMSE**	**RMSLE**	**Time**
Bagging	0.01283464	0.1105801	0.0450789	104.44 sec

Uncorrelated random forest	0.01277273	0.1102212	0.04496482	46.89 sec

We can observe that uncorrelated have slightly lower error on all of the three terms, but the difference is not too significant. However, the computation time between the methods is large, hence the uncorrelated method performs better with a much shorter amount of time compared to the bagging method. Thus, we will use mtry = 4 to continue the analysis. The figure below shows the plot of the actual output against the predicted output using random forest with mtry = 4 using the test data entries:

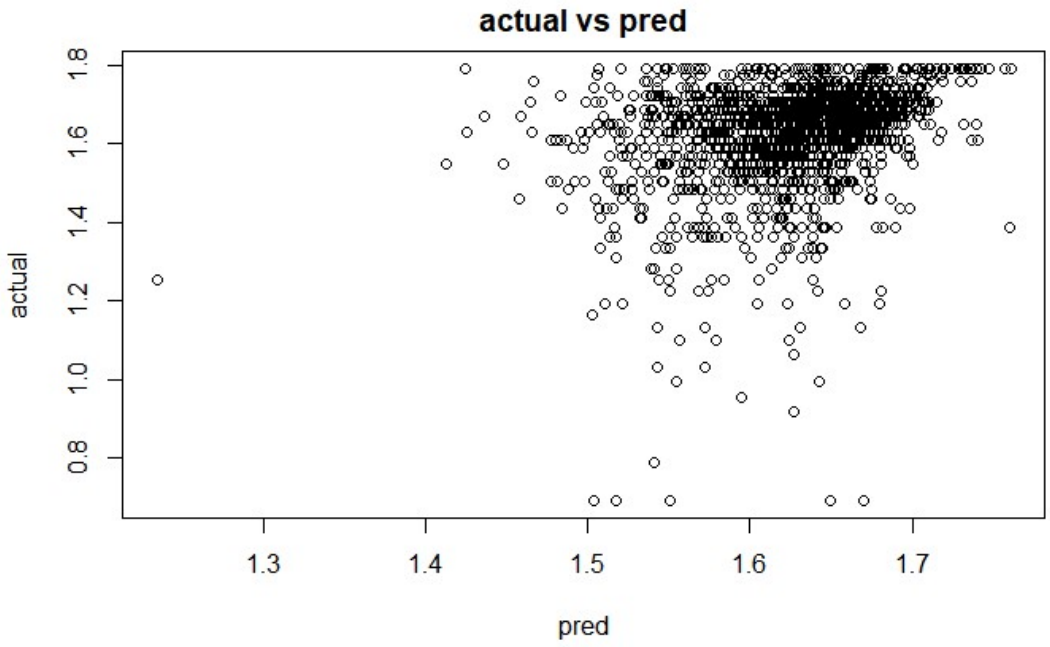

Figure 16 Actual output vs predicted output

We can see that despite of the fact that a proportion of the predicted values have a difference compared to the actual value, most predictions are very close to the real solution.

Boosting

As we know, random forest methods use a group of decision trees and aggregates the results from each tree by taking the average. However, there are other methods of combining the results of these weak learners like linear combination with weight assignment to each tree. Such method is called boosting, where it uses other tree's performance to improve their own prediction power.

In the implementation of boosting method attached in the Appendix, it has an important parameter called M, which determines the number of trees involved in the prediction. It is essential to select the optimal M as too many M will make the model prediction very slow, whereas too small M can result in high prediction errors. The figure below shows the result using M={10,100,500,1000}:

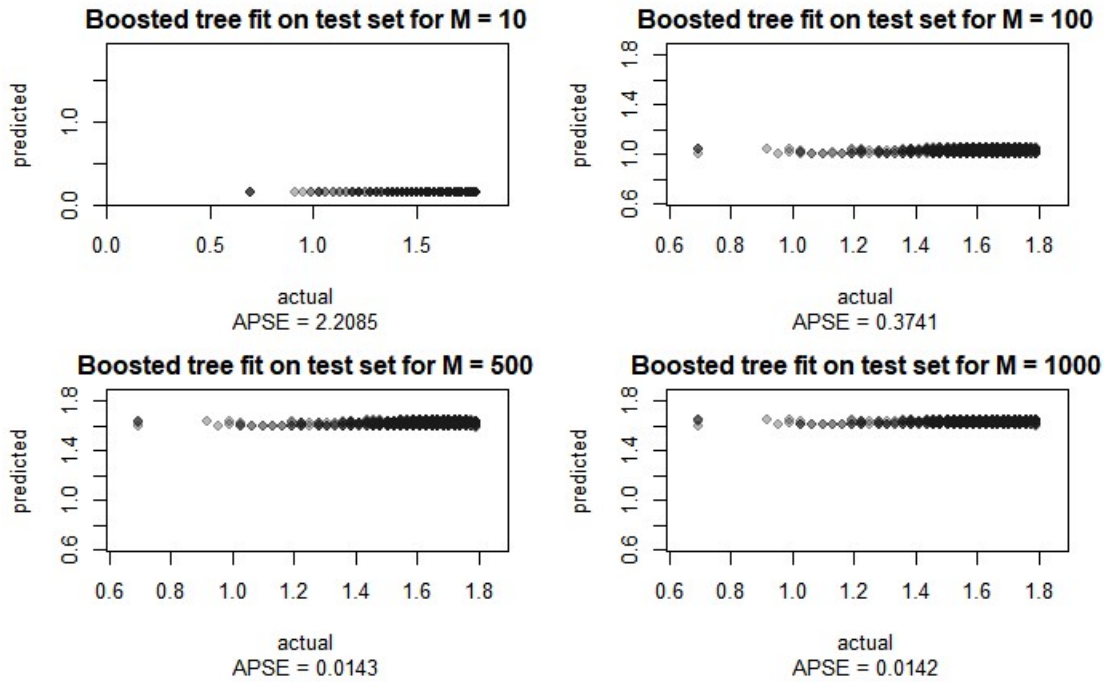

Figure 17 Performance comparison using different M

We can see that the APSE of M=10 and M=100 are quite different than M=500. However, M=500 and M=1000 have very similar APSE. Hence M=500 is the optimal M to perform prediction.

Another important parameter is the maximum depth of the boosting trees. We can see the difference in performance with different depths, as shown below:

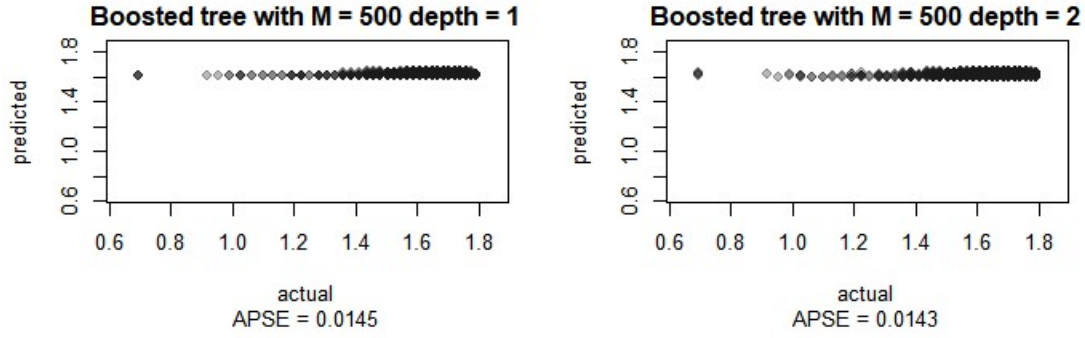

Figure 18 Difference in prediction with different depth values

Figure 18 above shows that the prediction error APSE = 0.0145 is similar using both depth values, and it matches with the best APSE obtained from the previous plots, which means that small trees have excellent prediction power.

Gradient Boosting

We can use the gradient boosting method to build a relative influence summary. More specifically, gradient boosting provides the relative influence of a variate amongst all the predictors, and the relative influence can also be shown in scaled form and in percentage:

Table 6 relative influence of all the predictors in scaled and in percentage forms

	Scaled	**Percentage (%)**
Category	1.000000000	40.24
Reviews	0.726710525	29.25
RVI	0.593756667	23.90
Installs	0.061386335	2.47
Last.Updated	0.048992496	1.97
Curr.Ver	0.021490719	0.86
Size	0.012475166	0.50
Android.Ver	0.011398888	0.46
Price	0.008582323	0.35
Content.Rating	0.000000000	0.00
Type	0.000000000	0.00

We can observe that the result is similar to what we obtained from the random forest method, where RVI, Category and Reviews are the most important features. The figure below is the plot of feature importance calculated using different trials of the gradient boosting method.

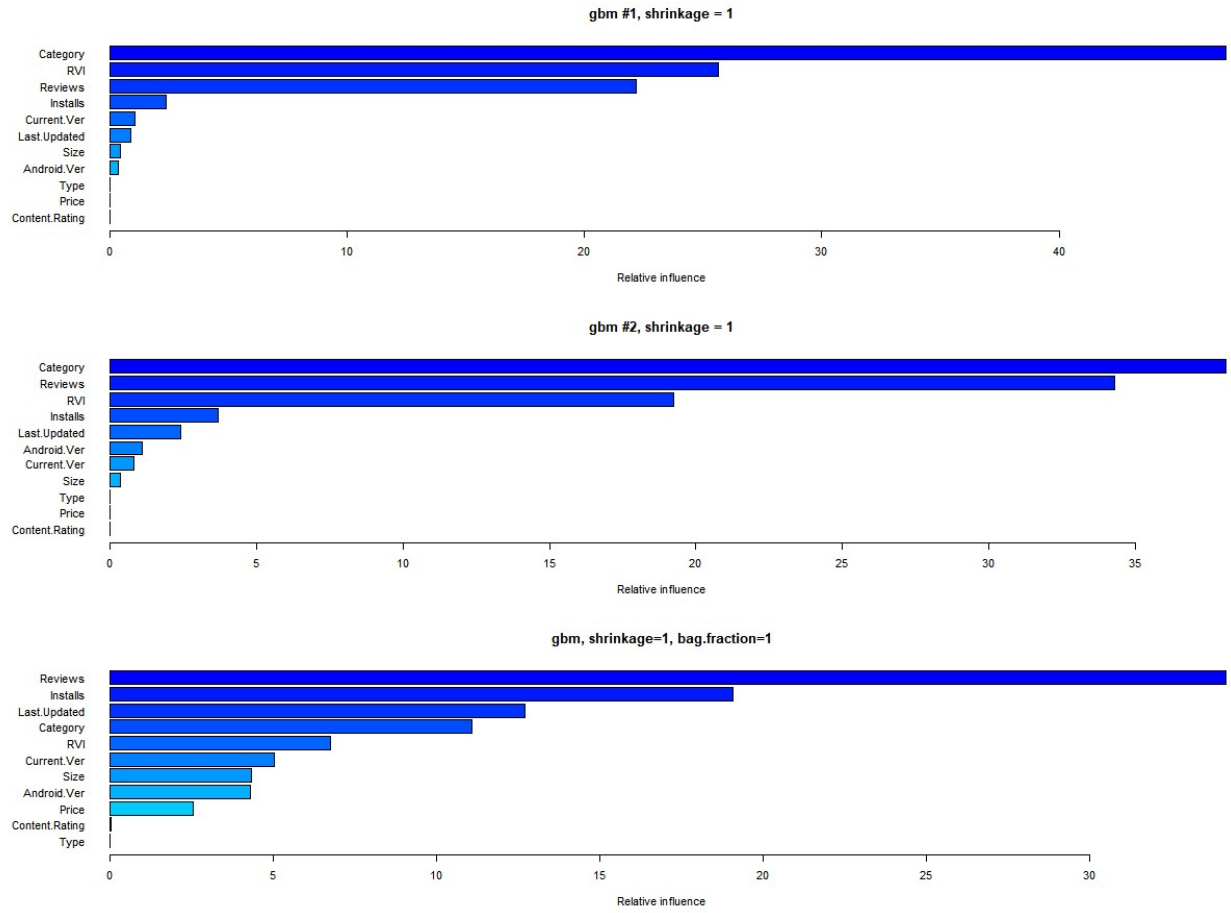

Figure 19 Feature importance with different bag.fraction value

We can see that with bag.fraction = 1, the result is slightly different than the other two plots, where the bag.fraction value is at its default 0.5. It is because when bag.fraction=1, the model uses all the training set for prediction, whereas models with bag.fraction=0.5 only uses half of the training set.

Cross-validation

The library *caret* is very useful to perform cross validation that can set different tree depths, number of trees and shrinkage. Since we determined earlier that the optimal number of trees is 500, we will test for different value of shrinkage = [0.1, 0.5, 1] and different tree depths = [1,3,5]. The result is shown below:

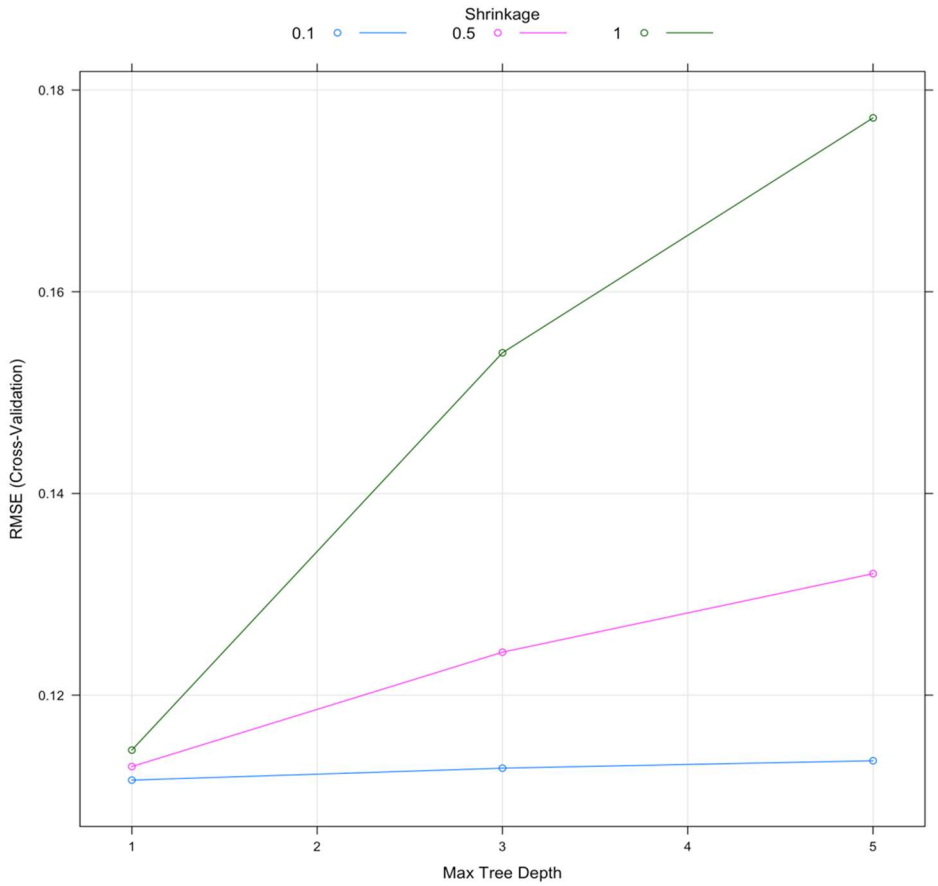

Figure 20 gbm Cross validation

We can conclude that for the gradient boosting method, the optimal parameters are

Table 7 optimal paramter for gradient boosting

Shrinkage	n.trees	Tree.depth
0.1	500	1

which can obtain the following result:

Table 8 Error prediction and computation time for gradient boosting

MSE	RMSE	RMSLE	Time
0.01296621	0.1138693	0.04699032	163.67 sec

Please see Appendix for the detailed summary of the fit for the gradient boosting method.

XGBoost

XGBoost stands for Extreme Gradient Boosting, which is an advanced version of gradient boosting. It has much faster processing time with high prediction power. By using caret to run the XGBoost method, the table below shows the result:

Table 9 Error prediction and computation time for XGBoost

MSE	RMSE	RMSLE	Time
0.01638778	0.1280147	0.0529205	78.54 sec

Statistical Conclusions

The table below summarizes the error prediction and the time consumed for each of the method presented in this report:

Table 10 summary of prediction errors and time consumed

Methods	MSE	RMSE	MSLE	Time
GAM	0.01587419	0.1259928	0.05219048	6.31 sec
Random forest	0.01277273	0.1102212	0.04496482	46.89 sec
Gradient boosting	0.01296621	0.1138693	0.04699032	163.67 sec
XGBoost	0.01638778	0.1280147	0.0529205	78.54 sec

We can see that even though the differences in errors prediction are not too far apart, the random forest model is the best one among the four. Both of its error prediction and computation time are ideal compared to the other models. Even though GAM has a much shorter computation time, its MSLE is not ideal.

XGBoost should have been performed much better than the other models, however due to insufficient parameter tuning, its performance is much inferior than expected, but its computation time is much better compared to gradient boosting, which proves its processing speed.

We know that random forest runs in parallel and it is less prone to overfit, hence it has a much better running time than gradient boosting, which is processed in series and is more likely to overfit the data. More delicate parameter tuning is required so that gradient boosting can outperform random forest. In other words, random forest is easier to apply than gradient boosting because the former does not require too much effort on parameters tuning to obtain a decent result. Thus, the final model for this dataset is the random forest model, and its optimal parameters are set as follows:

Table 11 paramters of the final model

Mtry	n.tree	Number of variates
4	100	11

The final formula is:

$$Rating \sim Category + Reviews + Installs + RVI + Size + Last.Updated + Curr.Ver + Android.Ver + Content.Rating + Type + Price$$

where *Rating, Reviews, Installs* and *Size* are log-transformed from the original dataset.

Conclusions in the context of the problem

Based on the results above, we can observe that RVI, Category, Installs, and Number of reviews have a great influence to the rating of an application. It is also very logical because a high rating application is more likely to have more downloads and more reviews. An application with more reviews but few downloads may indicate that reviews for this application are generally negative, which prevent new users to download the application. Moreover, the category of an application is also important to its rating because some categories (like Family and Tools) is easier to be accepted by users of any age and any background. In addition, these types of applications are generally not too complicated to manipulate, thus if they convey all its simple usages smoothly to the users, it would be very likely to obtain a high rating.

Furthermore, based on the discussed models, we can observe that the price of an application and its content rating have little to do with its rating. In other words, these factors should not have too much influence on the applications development.

Future work

There are a few caveats in this dataset that I wish to fix if I have more time. For instance, the percentage of variance explained using the random forest model is around 14%, which indicates that the predictors in this dataset does not understand well the variance of the response variable due to possibly weak relationship between the response variable and the predictors. Many attempts on creating new features and transforming existing features has been made, but none of them can increase significantly the percentage of variance explained. It would be interesting to develop or transform a feature that can have a closer relationship with the response variable.

In addition, it is worth mentioning that there is another dataset containing 60 thousand user reviews and properties of each review such as the sentiment (positive, negative or neutral) and

the polarity score and the subjectivity score of the corresponding sentiment. However, this additional dataset is only focused on about one thousand unique applications, whereas in the original dataset, there exists more than eight thousand distinct applications. It would be interesting to expand the user reviews dataset to include more applications. As a result, it could possibly introduce some useful features that have tighter relationship with the response variable.

Moreover, it is also recommended to try Neural Network to this dataset. However, by using the neural network method in *caret* library, the computation time takes too long to run such that is it not worth to leverage the prediction power of the neural network model. It requires more time to understand and tune the parameters of deep learning models in order to use it smoothly against this dataset.