Google Play Store Data Analysis

Zhang Yun, XingYu Fu

2019.05.07

Contents

1 Introduction

Google play [1] serves as the official app store for the Android, allowing customers to install apps developed with the Android software development kit and published through google. As Figure 1 and Figure 2 suggest, google play is gaining increaing popularity in recent years and has the most diverse available apps among the world leading app stores.

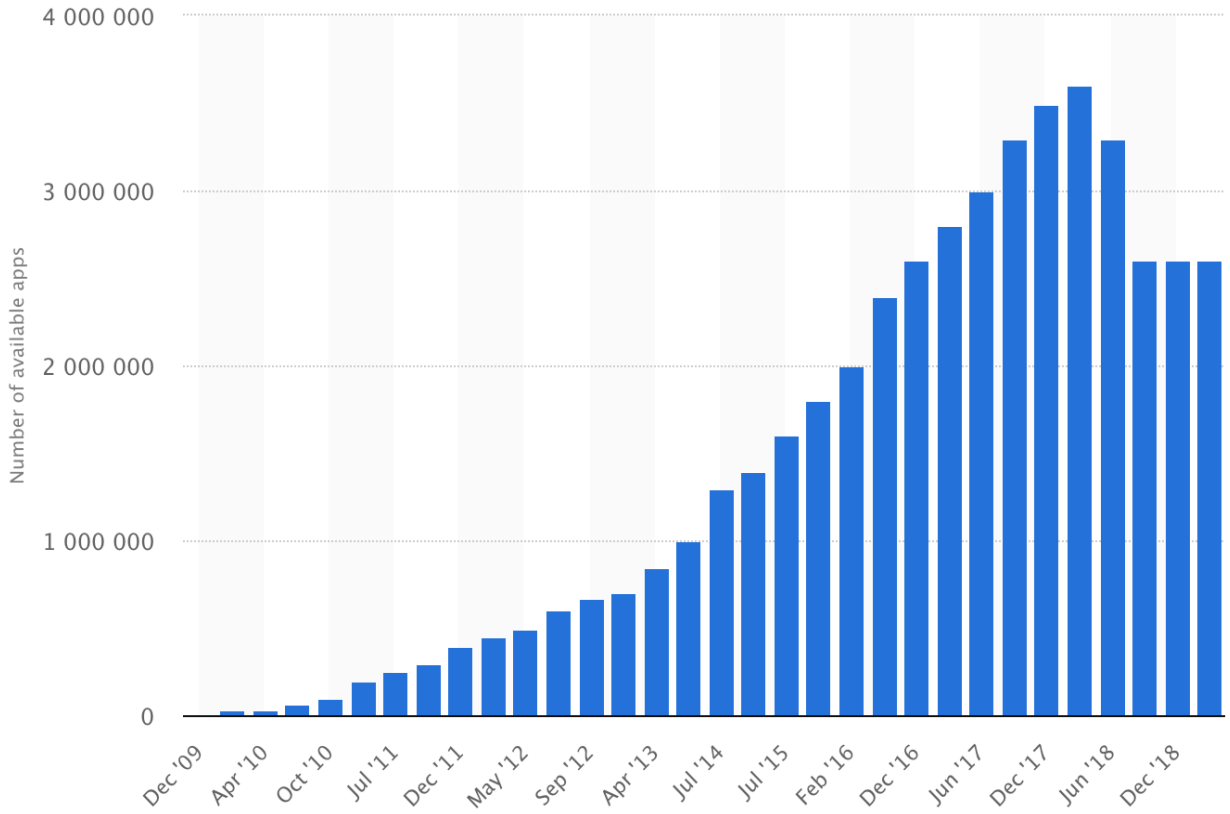

Figure 1: Number of available applications in the google play store from December 2009 to March 2019.

Despite its obvious importance, there are only limited number of researches carried out about this giant platform[1][2] and in this report, we will investigate google play store in depth by analyzing its apps' data through machine learning techniques. The report will organize as follow:

- Section 2 will explore the statistics of google play store to provide insights of its business and customers.
- Section 3 will model the rating of apps by its multiple features through machine learning techniques, where stacking ensemble and Bayesian optimization are applied jointly to make robust predictions.

[1]https://play.google.com/store

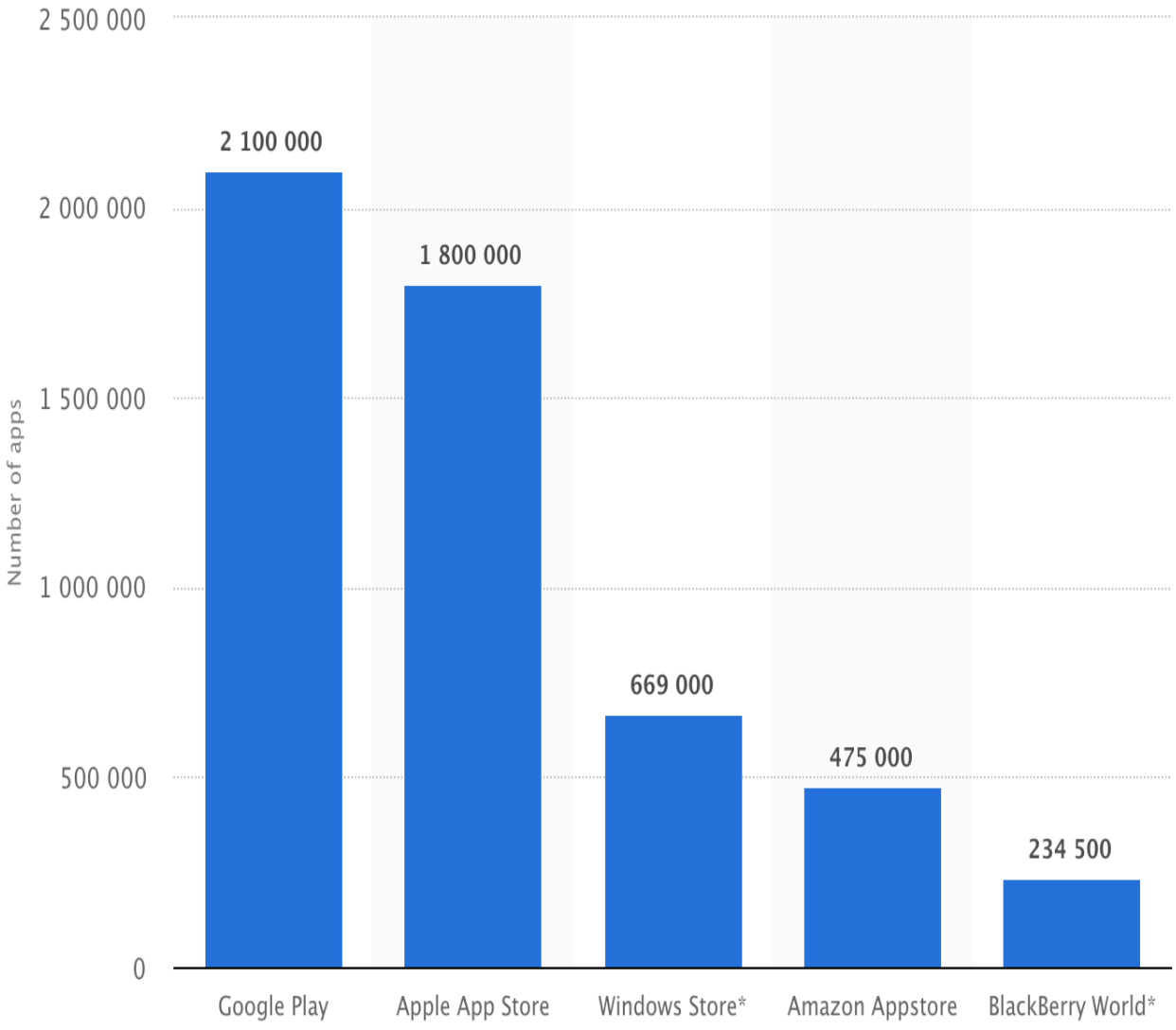

Figure 2: Number of apps available in leading app stores as of 1^{st} *quarter 2019.*

The data sources for this project are kaggle[2] and statista[3]. And we appreciate these two platforms for their generous sharings.

[2]https://www.kaggle.com/lava18/google-play-store-apps
[3]https://www.statista.com/

2 Exploratory Data Analysis

The first but also the most important step of all data science projects is to understand your data. To name a few aspects, say:

- What data do we have and what do they mean to the real-world problem?
- How to make the data manipulable by computer and how to generate new meaningful data from the original one?
- What's the distribution of data?
- What's the relationship within data?
- What features are more important to our task?

These are all insightful questions that should be addressed to truly understand data. In this section, we will conduct an in-depth research to answer the mentioned questions in google play store data setting. All visualizations are finished by seaborn python package[4].

2.1 Data Description, Pre-processing and Feature Engineering

There are 10841 app samples available at the google play store data and each sample from the original data set has the following 10 variables:

- App: Application name in the format of string (e.g. HD Mickey Minnie Wallpapers, TurboScan: scan documents and receipts in PDF).
- Category: Category this app belongs to in the format of string (e.g. ART_AND_DESIGN, BUSINESS).
- Rating: Users' numerical rating of the app in the format of float.
- Reviews: The number of reviews users make about the app in the format of integer.
- Size: Size of the app in the format of string (e.g. 19M, 201K).
- Installs: Number of user downloads for the app in the format of string (e.g. $1,000,000+$, $50,000+$).
- Price: Price of the app in the format of string (e.g. 0, $3.99).
- Content-Rating: Age group the app is targeted at in the format of string (e.g. Everyone, Teen).
- Last-Updated: Date when the app was last updated on play store (e.g. July 26, 2018).
- Android-Ver: Minimal required Android version for the app in the format of string (e.g. 4.0.3 and up).

[4]http://seaborn.pydata.org

We need pre-process the data to make it computable by algorithms. Specifically, we conduct the following transformations:

- Size: Convert the original string into float using megabyte as unit (e.g. change 19M into 19.0; change 201K into 0.19).
- Installs: Remove the plus sign " + " and comma "," from the string and convert it to float.
- Price: Remove the dollar sign "$" from the string and convert it to float.
- Last-Updated-Till-Now: Convert the original feature into the number of days from the given date to 2019.01.01.
- Android-Ver: Keep the first decimal point and discard the rest of them (e.g. change "4.0.3 and up" into 4.03).

In addition to these, we note that there are missing values in some samples' features and we fill them with the average feature values of the categories them belong to.
After pre-processing, we can construct new features from the original ones using domain knowledges, in this case, i.e. marketing, e-commerce, and costumer behaviour & psychology. This construction process is termed as feature engineering in applied data science and well-crafted features can sometimes improve the performance of machine learning algorithm. We will back to this topic at subsection 3.4.
Specifically, for each sample, we construct the following 4 features:

- Data-Flow: Size times Installs, i.e. the total megabytes flowed from the app.
- Sale-Volume: Price times Installs, i.e. the gross income earned by selling the app.
- Review-Install-Ratio: Reviews divided by Installs, i.e. the ratio of active users against total users.
- Name-Length: The number of characters in app's name. This engineered feature seems to be trifling at the first glance. However, as the analysis going, we will find the magical power of Name-Lenght in predicting app's rating.

Figure 3 visualizes the distributions of Category and Content-Rating, and we can see that:

- There are 33 different categories. FAMILY, GAME, and TOOLS are the top 3 categories with the most diverse apps.
- There are 6 different Content-Ratings, and most apps are rated as EVERYONE, i.e. no age constraint.

Figure 4 visualizes the distributions of Rating, Last-Update-Till-Now, Android-Ver and Name-Length, and we can see that Rating's distribution is approximately a normal distribution with mean $\mu = 4.2$ and standard deviation $\sigma = 0.48$.

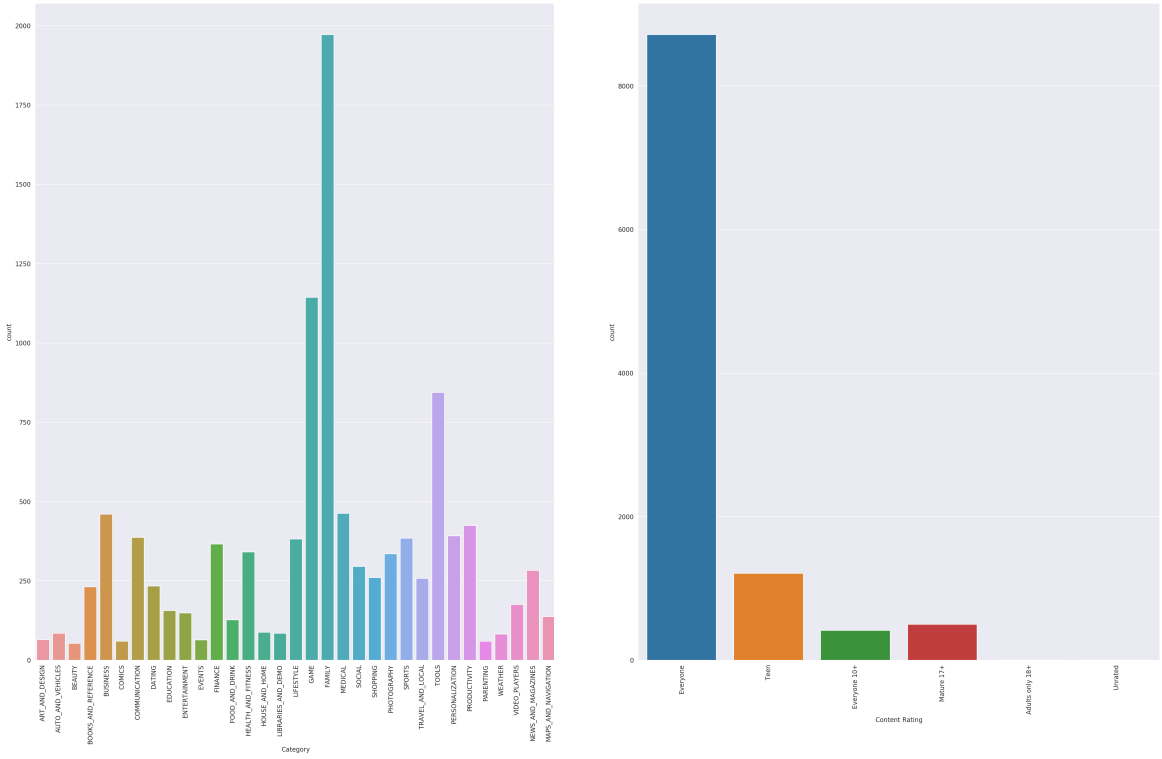

Figure 3: Distributions of Category and Content-Rating.

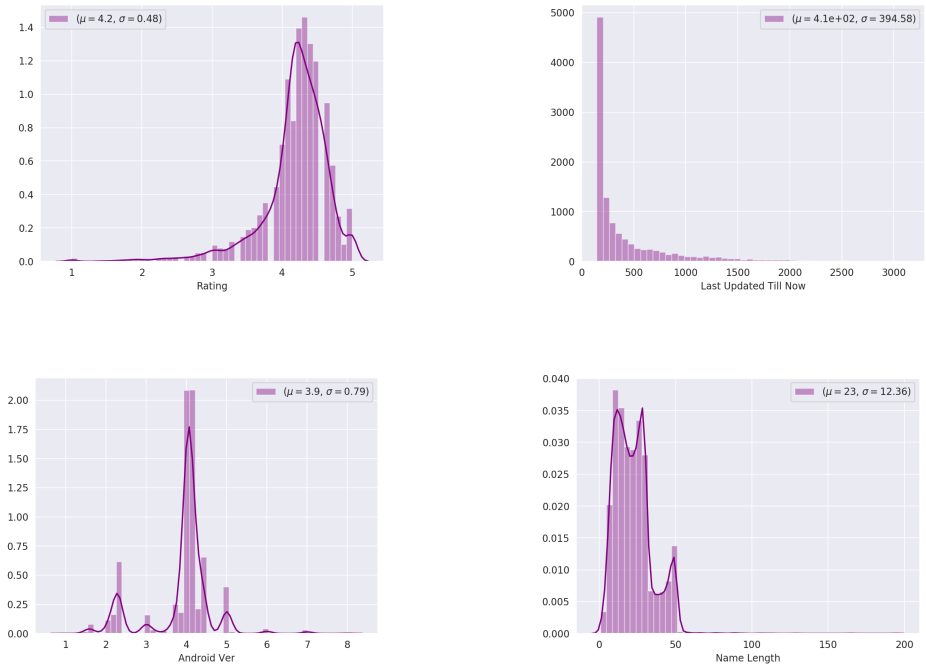

Figure 4: Distributions of Rating, Last-Update-Till-Now, Android-Ver and Name-Lenght.

2.2 Relationships between Variables

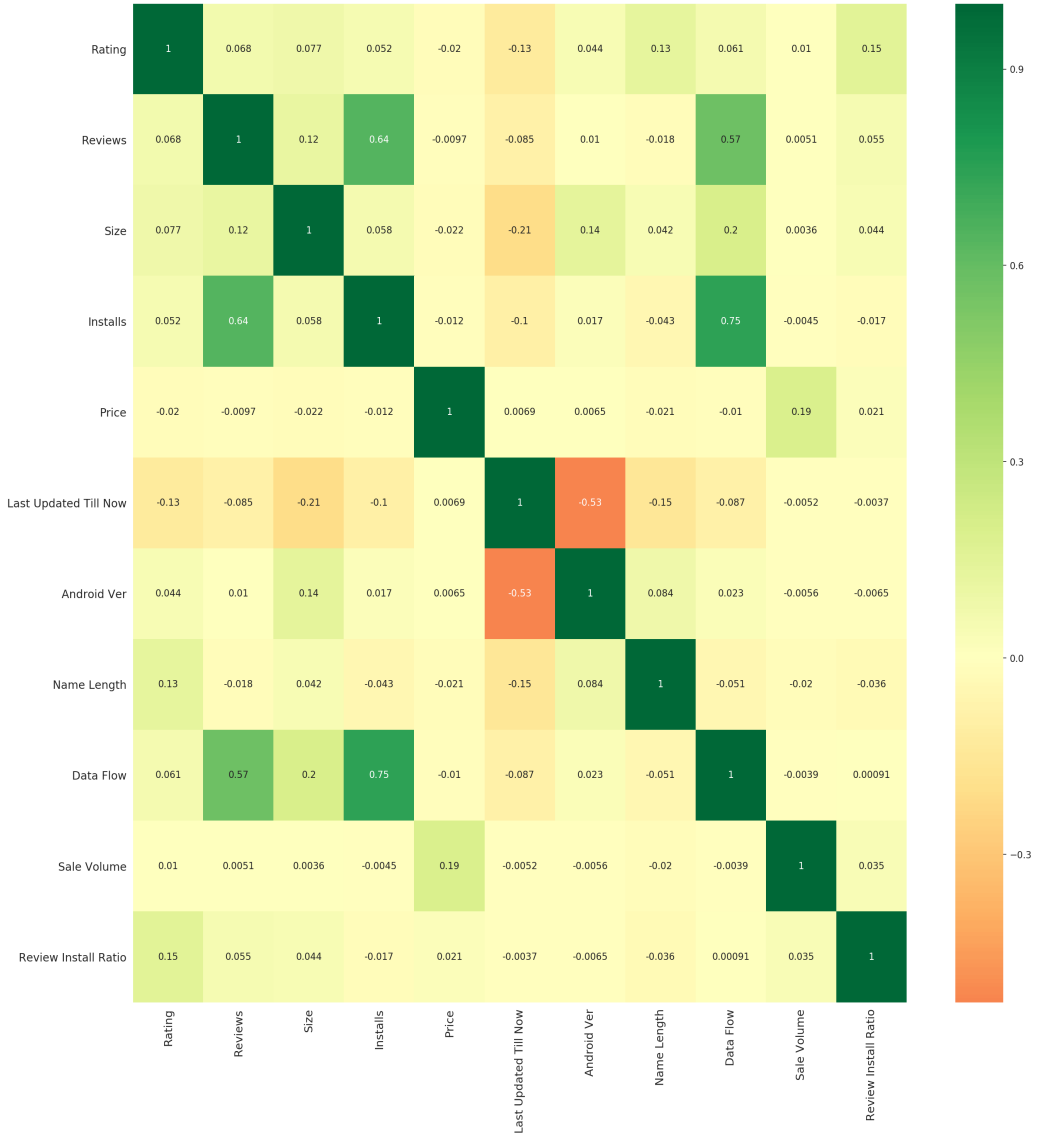

Figure 5: Correlation matrix of all numerical features, each entry represents the correlation coefficient, which is visualized by color, between a pair of features.

Figure 5 visualizes feature correlations, and we can observe some strong correlations:

- Last-Update-Till-Now and Android-Ver are strongly negative correlated, which is intuitive since newly-updated apps are often supported by more recent operation system.

- Reviews and Installs are strongly positive correlated since both of these features directly reflect app popularity.
- Data-Flow are strongly positive correlated to Installs due to its construction, and Reviews for the strong positive relationship between Installs and Reviews.

We further examine the first two relationships in Figure 6 (the third relationship is discarded since this correlation is somehow trivial). As we can see, the linear regression lines that fit these two relationships both have steep slopes.

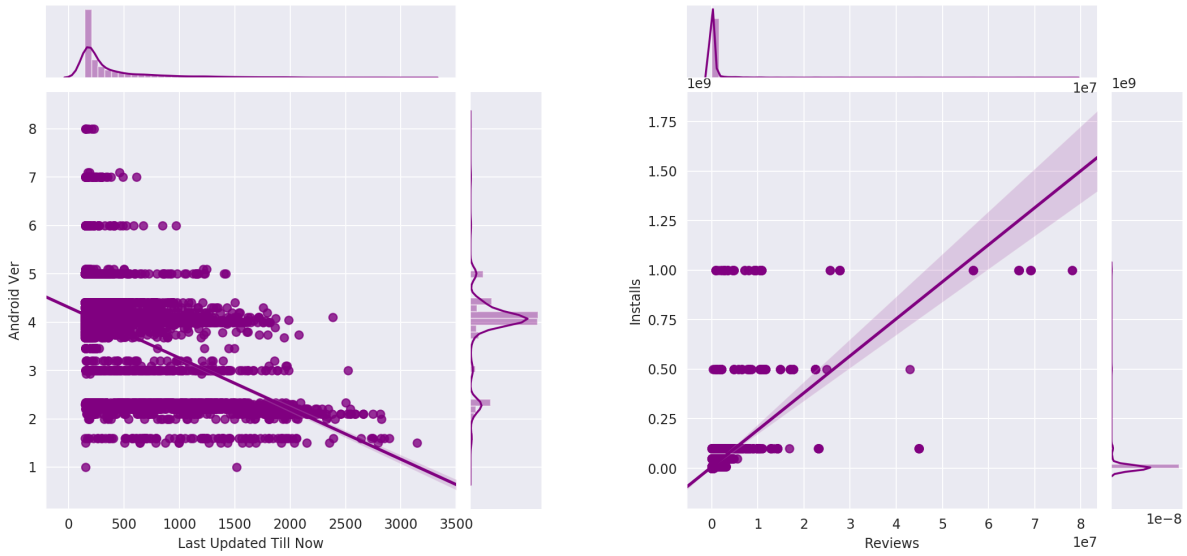

Figure 6: Scatters and regression lines of two strongly correlated feature pairs, i.e. Last-Update-Till-Now & Android-Ver and Reviews & Installs.

Another major problem we care is the correlations between features and rating, which will be the predictive target (label) in section 3. As can be seen from Figure 5:

- Rating is strongly positive correlated to Review-Install-Ratio.
- Rating is strongly positive correlated to Name Length, which is against intuition.
- Rating is strongly negative correlated to Last-Update-Till-Now.

We further visualizes the linear relationships between numerical features versus rating in Figure 7 and this topic will continue in subsection 2.3.

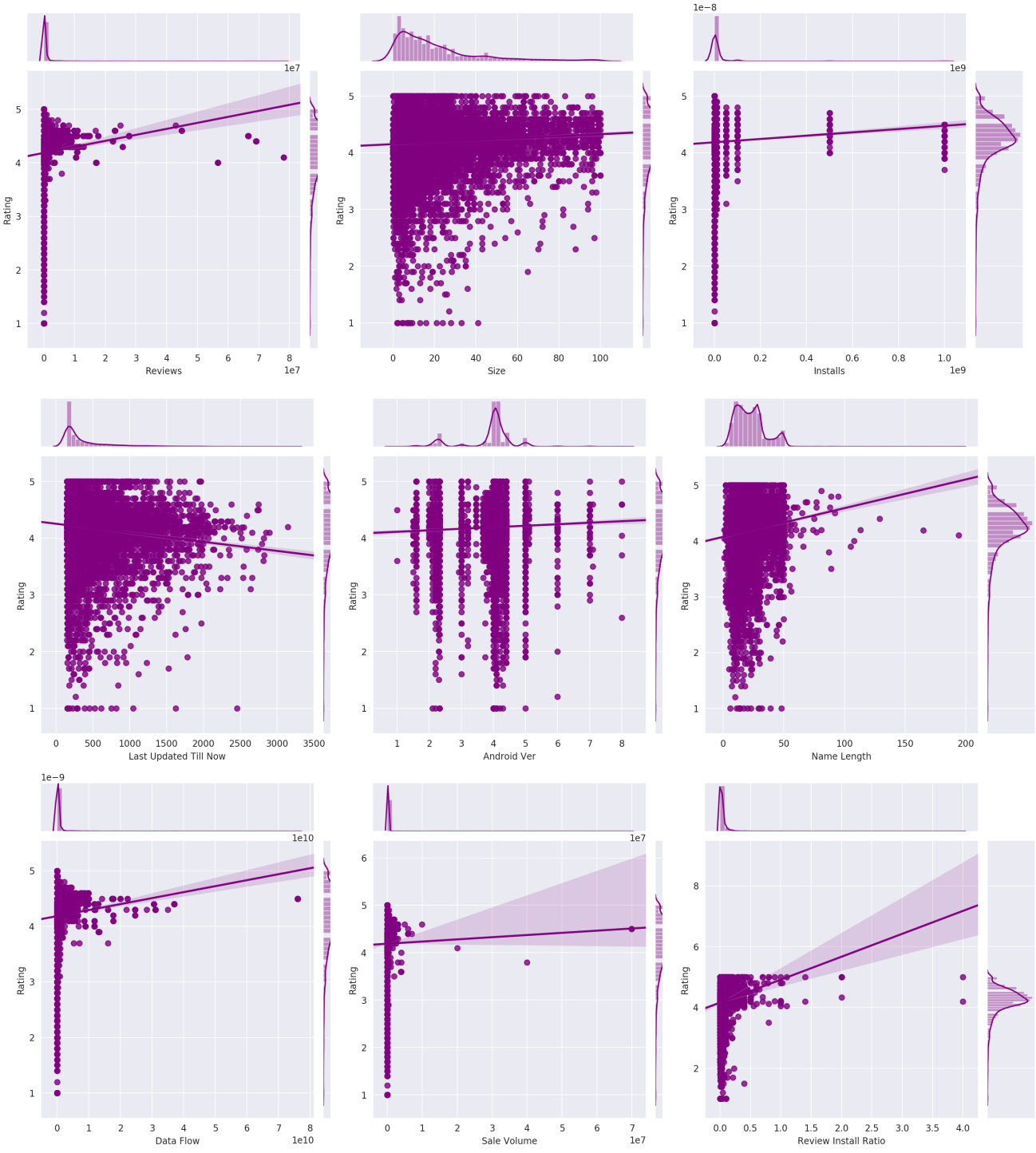

Figure 7: Scatters and regression lines of numerical features versus rating.

2.3 Machine Learning based Feature Importances

Understanding what features are important for our prediction task is crucial since it can provide insights about the underlying real-world problem and help data scientists remove negligible features (i.e. noise) from machine learning model. Here we adopt two algorithms to measure feature importances with respect to rating prediction:

- Mean Decrease Impurity (MDI): This is an in-sample importance measure based on random forest algorithm. For each decision tree in the trained forest, we can measure each feature's importance inside the tree using its total impurity reduction on the nodes of the tree. We then average the importances over all the trees in forest to obtain MDI importance measure.

- Mean Decrease Accuracy (MDA): This is an out-of-sample importance measure. To measure feature f's importance, we first train a machine learning model on the training data set and then test it on a testing data of which feature f is randomly shuffled. We compare the prediction accuracy before and after the random shuffle and use the accuracy reduction ratio as f's MDA. Algorithm 1 demonstrates the calculation pipeline of MDA.

Algorithm 1: Mean Decrease Accuracy (MDA)

Input: Training Data Set $D = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$; Testing Data Set $D^{'} = \{(x_1^{'}, y_1^{'}), (x_2^{'}, y_2^{'}), ..., (x_M^{'}, y_M^{'})\}$; Machine Learning Algorithm L; Feature List $F = \{f_1, f_2, ..., f_n\}$;

Train L on training data set D ;

Test L on testing data set $D^{'}$ and calculate its $R_2 = 1 - \frac{\sum_{i=1}^{M}(L(x_i^{'}) - y_i^{'})^2}{\sum_{i=1}^{M}(\overline{y}^{'} - y_i^{'})^2}$;

for $f \in F$ **do**

  Shuffle the values of f for $\forall x_i^{'} \in D^{'}$. We denote the resulted test set as $D_f^{'}$;

  Test L on $D_f^{'}$ and calculate its $R_2^f = 1 - \frac{\sum_{i=1}^{M}(L(x_i^{'f}) - y_i^{'})^2}{\sum_{i=1}^{M}(\overline{y}^{'} - y_i^{'})^2}$;

  Define $MDA_f = \frac{R_2 - R_2^f}{R_2}$;

end

In general, MDI and MDA screen the data in two different aspects and their computation results are not necessarily the same. And therefore, we need to check both of them to get a balanced look about our data.

Back to our google play store setting, Figure 8 visualizes feature importances with respect to rating prediction, showing that MDI and MDA both agree the engineered feature Review-Install-Ratio is most important, which coincides with result from subsection 2.2. Besides, the original feature Reviews, which is the component of Review-Install-Ratio, is also assigned with high importance by both MDI and MDA. Another interesting observation that worth further investigation is both MDI and MDA assign the feature Name-Length, which intuitively seems to be irrelevant to rating, with high scores and this founding also coincides with result from subsection 2.2.

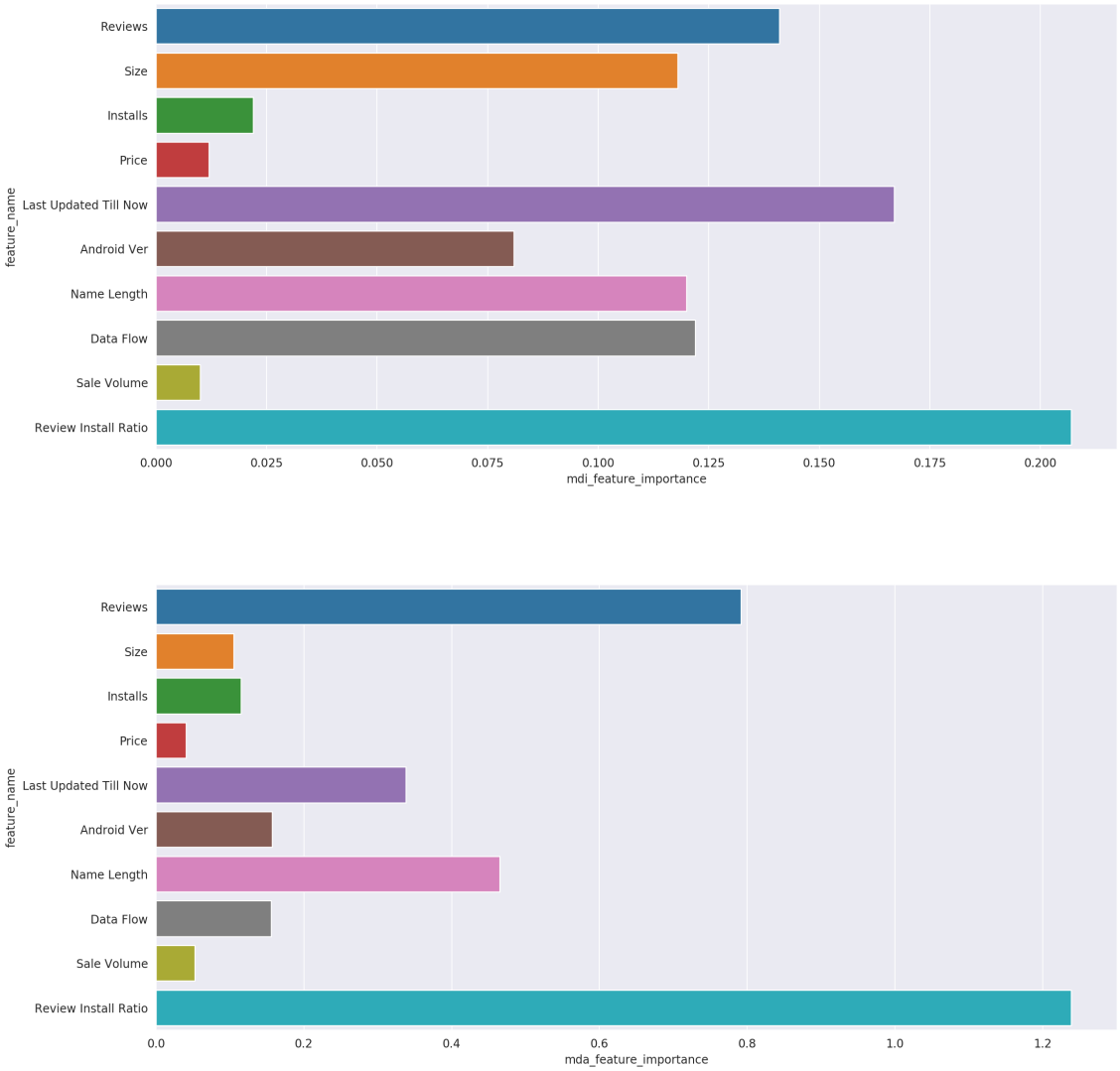

Figure 8: MDI and MDA Feature importances with respect to rating prediction.

3 App Rating Prediction

Can we predict an app's rating based on its features? This is an important question in many ways:

- For marketing researchers, they can further understand the relationship between customer's unconcious behaviours (i.e. apps' features) and concious responses (i.e. apps' ratings).
- For google play store, they can develop some sorts of recommendation algorithm based on the rating prediction so that they can promote their app sales.

For the reasons above, we will build a systematic framework to conduct app rating prediction in this section.

3.1 Problem Formulation

Let $D = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$ be the google play store data, where $x_i \in R^{p\times 1}$ is the feature vector and $y_i \in R$ is the rating of the i^{th} app. We attemp to train a machine learning model $L : R^{p\times 1} \to R$ that can predict an app's rating value given its feature vector, i.e. for any feature vector $x \in R^{p\times 1}$, we predict its rating:

$$\hat{y} = L(x)$$

The formulation above lies in the regression problem, where many sophisticated statistical methods have been developed. To name a few, linear models (e.g. Lasso[3], Ridge[4]), non-linear models (e.g. Support Vector Regression[5]), and tree-based models (e.g. Random Forest[6], Gradient-Boosting Tree[7]). All the mentioned models will be used in this section to serve as bricks of our ultimate rating prediction algorithm.

3.2 Stacking Ensemble Prediction

Ensemble learning[8] is a technique that combines multiple outputs from different machine learning algorithms to make a better prediction. Generally, ensemble learning can lower the prediction variance and increase accuracy compared to any of its components. There are mainly three types of ensemble method: Bagging[9], Boosting[10], and Stacking[11].
In this work, we will adopt stacking ensemble to improve predictive performance. Specifically, we will train several base learners on the training data set (in our case, i.e. Lasso, Ridge, Support Vector Regression, Random Forest, and Gradient-Boosting Tree) and then combine them by a meta learner (in our case, i.e. Ridge). The meta learner will assign differet weights to base learners and generate ultimate prediction based on their weighted predictions.

Algorithm 2 is the training algorithm of stacking ensemble.

Algorithm 2: Training of Stacking Ensemble Model

Input: Training Data Set $D = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$; Base Learners $\{L_1, L_2, ..., L_T\}$; Meta Learner L;

 # Step1: Train Base Leaners

for $t = 1, 2, ..., T$ **do**

  Train L_t on D;

end

 # Step2: Construct Training Data D^* for Meta Learner

$D^* = \varnothing$;

for $n = 1, 2, ..., N$ **do**

  for $t = 1, 2, ..., T$ **do**

    $z_{nt} = L_t(x_n)$;

  end

  $z_n = (z_{n1}, z_{n2}, ..., z_{nT})^T$;

  $D^* = D^* \cup \{(z_n, y_n)\}$;

end

 # Step3: Train Meta Learner

Train L on D^* ;

After training, the inference rule of stacking ensemble is a composition of meta leanrner and base learners, i.e. for any feature vector $x \in R^{p \times 1}$, we predict its rating:

$$\hat{y} = L(z)$$

where $z = (L_1(x), L_2(x), ..., L_T(x))^T$ is the predictions of base learners.

3.3 Hyper-Parameters Tuning through Bayesian Optimization

There are eight hyper-parameters to be tuned in the above learning system and their values will impact the predictive power of our system:

- Base learner Lasso's α: ranging from 0.1 to 5.0 and taking float value, determines the level of penalty of the L_1 norm of base Lasso's regression weights.
- Base learner Ridge's α: ranging from 0.1 to 5.0 and taking float value, determines the level of penalty of the L_2 norm of base Ridge's regression weights.
- Base learner Support Vector Regression's ϵ: ranging form 0.01 to 0.50 and taking float value, determines predictions' maximal allowed deviation from the training targets.
- Base learner Support Vector Regression's C: ranging from 0.1 to 5.0 and taking float value, determines the level of penalty of the amount up to which deviations larger than ϵ are tolerated.
- Base learner Random Forest's maximal tree depth: ranging from 3 to 20 and taking integer value, determines the maximal depth of each tree in the forest.

- Base learner Gradient-Boosting Tree's maximal tree depth: ranging from 3 to 20 and taking integer value, determines the maximal depth of each tree in the boosting.
- Base learner Gradient-Boosting Tree's learning rate: ranging from 0.01 to 0.50 and taking float value, determines the update speed of base Gradient-Boosting.
- Meta learner Ridge's α: ranging from 0.1 to 5.0 and taking float value, determines the level of penalty of the L_2 norm of meta Ridge's regression weights.

To tune hyper-parameters rigorously, we give several notations:

- $\theta \in R^{8\times 1}$ is the hyper-parameters vector whose entries correspond to the above mentioned parameters.
- Θ is the search space of θ and is constructed by taking Cartesian product of each hyper-parameters' mentioned ranges.
- $F : \Theta \to R$ is the function that measures the performance of given hyper-parameters vector. Specifically, for each $\theta \in \Theta$, we will train our learning system on the training data set using hyper-parameters θ and define $F(\theta)$ as the trained system's Negative Mean Absolute Error (NMAE) on the validation data set.

With notations above, we can define the hyper-parameters tuning problem as an optimization problem, i.e. find the optimal $\theta^* \in \Theta$ such that:

$$\theta^* \in \arg\max_{\theta\in\Theta} F(\theta)$$

However, this optimization problem has three obstacles:

- the search space Θ is too large due to its high dimensionality.
- $\forall\theta \in \Theta$, the computaion of $F(\theta)$ is time-consuming since we need to train six models, including five base learners and one meta learner, on the training data set, and then test them on the validation data set to get the value of $F(\theta)$.
- F doesn't have explict analytical expression and is not differentiable.

To overcome the mentioned obstacles, we use Bayesian Optimization (BO)[12] to conduct hyper-parameters tuning. We choose it for following reasons:

- BO can handle high dimensional search space.
- BO only searches promising candidates so that low-quality queries are avoided. By "promising", we mean some sorts of optimality of the candidates.
- BO has no prior assumption on the optimization objective F.

Algorithm 3 demonstrates the process of Bayesian optimization:

Algorithm 3: Bayesian Optimization (BO)

Input: Objective function F; Fitting Stochastic Process P;
 # Initialize the observation data of objective function F
$O_0 = \varnothing$;
 # Initialize the acquisition function α
$\alpha_0 : \Theta \to R. \quad \alpha_0(\theta) = 0, \forall \theta \in \Theta$;
 # Find query points sequentially
for *n = 1, 2, ..., I* **do**
  # Maximize acquisition function α **to obtain the next query point**
  $\theta_n \in \arg\max_{\theta \in \Theta} \alpha_{n-1}(\theta)$;
  # Compute the value of F **at the query point**
  $F_n = F(\theta_n)$;
  # Update the observation data of F
  $O_n = O_{n-1} \cup \{(\theta_n, F_n)\}$;
  # Update the posterior distribution of the fitting stochastic process P
  $P_n = P_0 | O_n$;
  # Update the acquisition function
  $\alpha_n = \alpha_0 | P_n$;
end

3.4 Experiments

We have 4 experiments in this subsection to test our modelling. Our stacking learning system is built upon scikit-learn python package[5] and we use open-sourced Bayesian optimization python package[6] to conduct hyper-parameter tuning.
As we can see from Figure 9, Bayesian optimization can efficiently search for high-quality hyper-parameters and improve model's NMAE on validation set by 10% compared to the initial hyper-parameters setting.
As we can see from Figure 10, stacking outperforms any of its component models, which validates that ensemble can diliver more accurate prediction.
From Figure 11, we see that the meta learner Ridge assigns different weights to its sub-models, indicating that stacking can take advantage of well-performed sub-models (in our case, i.e. Gradient-Boosting Tree) and punish under-performed sub-models to achieve better result.
From Figure 12, we can see that our constructed features improve model's predictive power by 48.36% in R_2 performance measure compared to the non-engineered one.

[5]https://scikit-learn.org/stable/index.html
[6]https://github.com/fmfn/BayesianOptimization

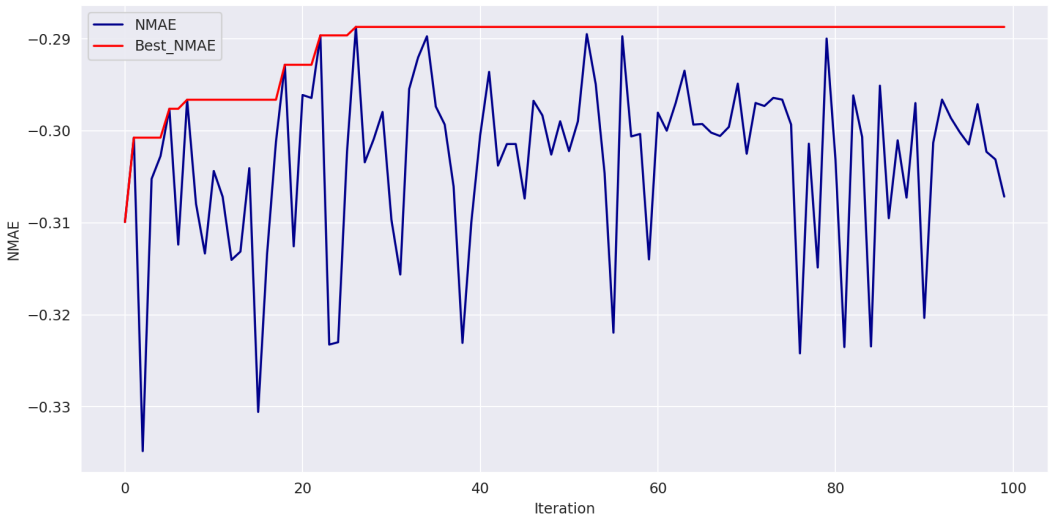

Figure 9: The process of Bayesian optimization, where the blue line is NMAE on validation set as hyper-parameters change across iteration and the red line is the optimal NMAE until the current searching.

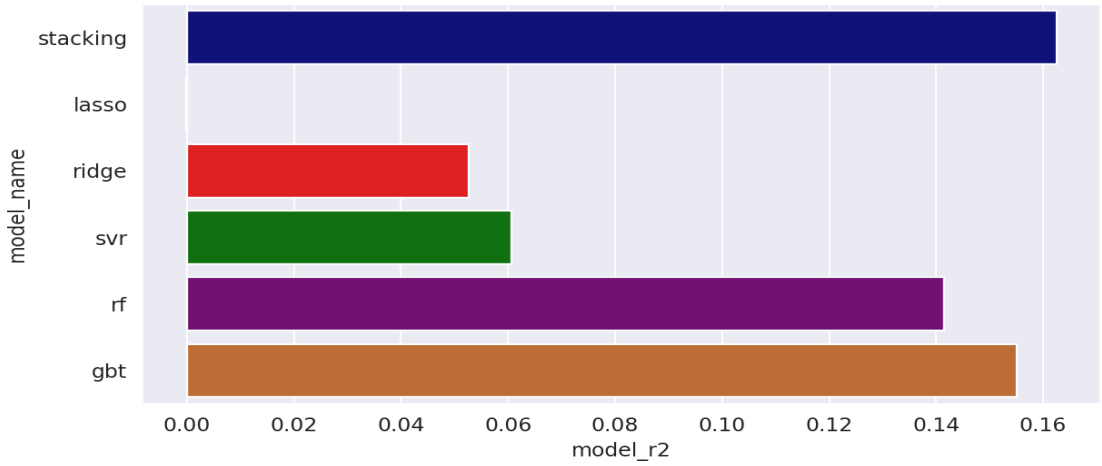

Figure 10: Performance comparison among different machnie learning algorithms, where the horizontal axis represents the R_2 scores of models in the test set.

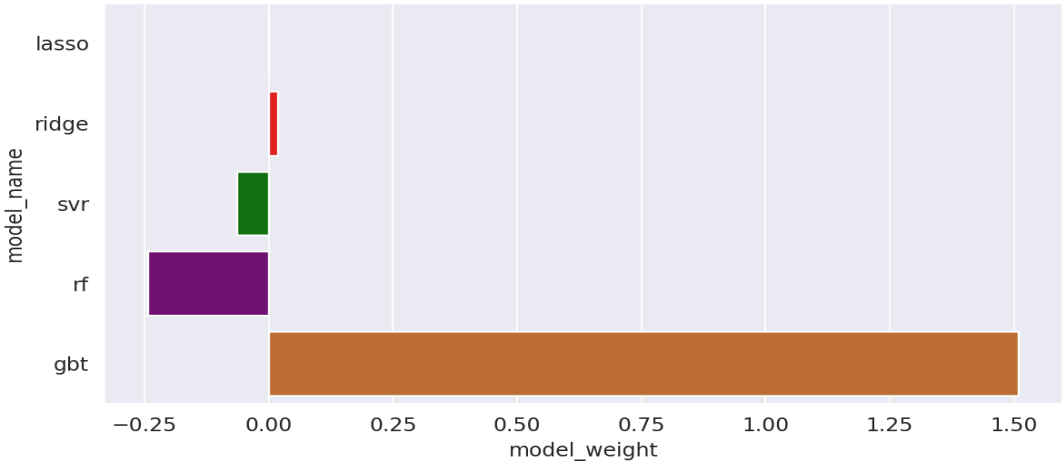

Figure 11: Model importances inside the stacking ensemble model, where the horizontal axis represents the meta-ridge regression coefficients of different sub-models in stacking.

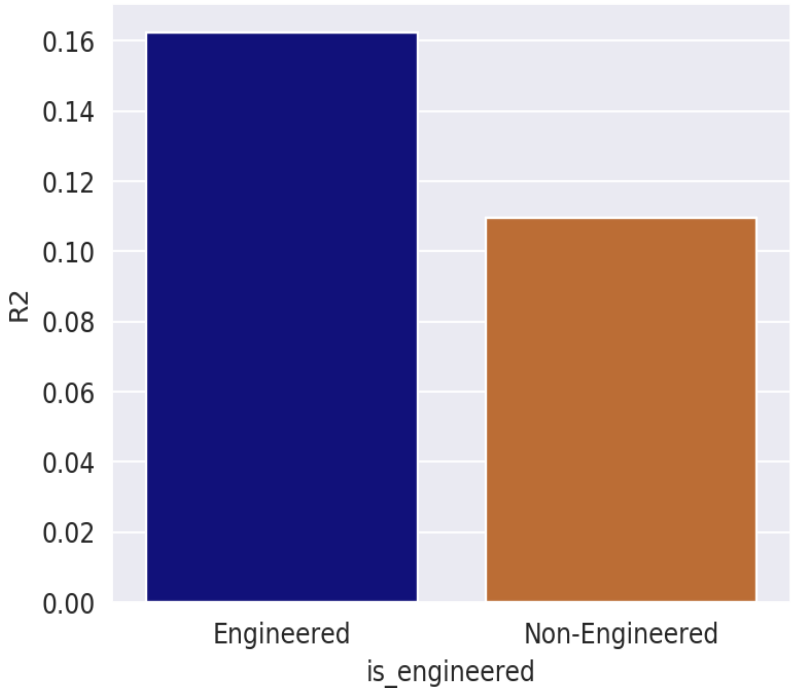

Figure 12: Comparison between the R_2 scores before and after feature engineering, where the blue bar represents stacking performance with engineered features and the orange bar represents stacking performance without engineered features.

References

[1] McIlroy S, Ali N, Hassan A E. Fresh apps: an empirical study of frequently-updated mobile apps in the google play store[J]. Empirical Software Engineering, 2016, 21(3): 1346-1370.

[2] Viennot N, Garcia E, Nieh J. A measurement study of google play[C]//ACM SIGMETRICS Performance Evaluation Review. ACM, 2014, 42(1): 221-233.

[3] Tibshirani R. Regression shrinkage and selection via the lasso[J]. Journal of the Royal Statistical Society: Series B (Methodological), 1996, 58(1): 267-288.

[4] Hoerl A E, Kennard R W. Ridge regression: Biased estimation for nonorthogonal problems[J]. Technometrics, 1970, 12(1): 55-67.

[5] Smola A J, Schölkopf B. A tutorial on support vector regression[J]. Statistics and computing, 2004, 14(3): 199-222.

[6] Liaw A, Wiener M. Classification and regression by randomForest[J]. R news, 2002, 2(3): 18-22.

[7] Friedman J H. Greedy function approximation: a gradient boosting machine[J]. Annals of statistics, 2001: 1189-1232.

[8] Zhou Z H. Ensemble learning[J]. Encyclopedia of biometrics, 2015: 411-416.

[9] Breiman L. Bagging predictors[J]. Machine learning, 1996, 24(2): 123-140.

[10] Freund Y, Schapire R E. Experiments with a new boosting algorithm[C]//icml. 1996, 96: 148-156.

[11] Wolpert D H. Stacked generalization[J]. Neural networks, 1992, 5(2): 241-259.

[12] Snoek J, Larochelle H, Adams R P. Practical bayesian optimization of machine learning algorithms[C]//Advances in neural information processing systems. 2012: 2951-2959.

[13] Friedman J, Hastie T, Tibshirani R. The elements of statistical learning[M]. New York: Springer series in statistics, 2001.

[14] Julian I A. Modern Multivariate Statistical Techniques: Regression, Classification, and Manifold Learning[J]. 2008.