Features That Predict App Updates

Mark Zimmerman
Mentor: Dr. Wenjun Zhou

Executive Summary

In this project, Dr. Zhou and I study the different features of mobile apps such as app description, update description, and many others to find which features best predict rating changes. We use a dataset of a large number of apps at time 1 (T1) and time 2 (T2). The apps at T1 are from June of 2015, and the apps at T2 are from September of 2015. The time difference in these two datasets of the same apps allows us to analyze what changed and what may have caused those changes. The first step was to clean the data because a dataset of this many apps was very messy. There was missing data, non-English words, and symbols that cluttered text analysis. Once the data were cleaned, we then had to make subsets of the apps that were updated and not updated between T1 and T2. After this separation of data, we could start our analysis in R. Analysis was performed using the Latent Dirichlet Allocation (LDA) model, regression analysis, and other abstract predictive models. The performance metrics on our models to calculate propensity scores that we used include accuracy, precision, recall, and f1 with accuracy being the most influential in our decision to choose the best model. After testing many models, we found that our best model was the Support-Vector Machine (SVM) model. The features in our best model used to predict update include: (1) number of ratings for past versions, (2) average rating for past versions, (3) number of rating for all app versions, (4) is the app free, (5) weight of the current version, (6) number of stars for all app versions, (7) number of stars for

current app version, (8) number of days since last update, (9) how many ratings per day since last update, (10) category, and (11) subcategory. We will continue to test more models with different features to predict the update status of the apps. Also, a next step includes looking deeper into why some apps are being updated and some are not.

Introduction

Mobile apps are becoming more and more popular every day. With the increased number of apps, there are always reminders to constantly update these apps. It is interesting to see if these updates can be used to improve user engagement. The engagement can be reflected in many different ways. For example, having more app updates could reflect app higher ratings or the opposite. In this project, we dive deeper into app updates to see if there is a story they tell about the user engagement of the app.

It is challenging to assess the causal effect of an update, since app developers self-selected to update or not. We cannot run a controlled experiment. Therefore, the plan is to use propensity score matching: each app that is updated is to be matched with an app that is not but had the same propensity of being updated. To do so, we need to train a model that can tell us a quantitative score that corresponds to likelihood of an update.

In this project, our goal is to find a good predictive model at a baseline to see if an app would be updated.

Data Description

Our data is comprised of several categories (lifestyle, weather, social networking) of apps pulled directly from Apple's App Store recorded at two separate times. 212,465 apps were found at time 1 (T1), June 2015. Checking the apps in the same categories, we found 222,550 apps at time 2 (T2), September 2015, with a small number of new apps added or delisted. Our analysis will focus on apps that are actively listed at T1 and T2.

Each app has 33 features including identifier features like App ID and App URL, categorical features like category and language, and numerical features like *Days Since Last Update* and *Rating of the Current Version*. Features such as *App Description* and *Update Description* allow us to perform text analysis to learn more about each specific app and to see if there are any overlapping patterns.

Data Cleaning

Firstly, many of the apps had missing information, so these apps were omitted in our analysis. Next, since this project focuses on App Updates, we merged the two datasets into one data set with a binary variable, Update. After the two datasets were merged, there were 193,383 apps that existed at both T1 and T2. Apps that changed their Update Date between T1 and T2 were marked as Updated. Apps that did not change their Update Date between T1 and T2 were marked as Not Updated. This merge allows us to have a Treatment Group and a Control Group. Figure 1 shows a visual description of our data setup that Dr. Zhou provided for me to understand the setup of the data.

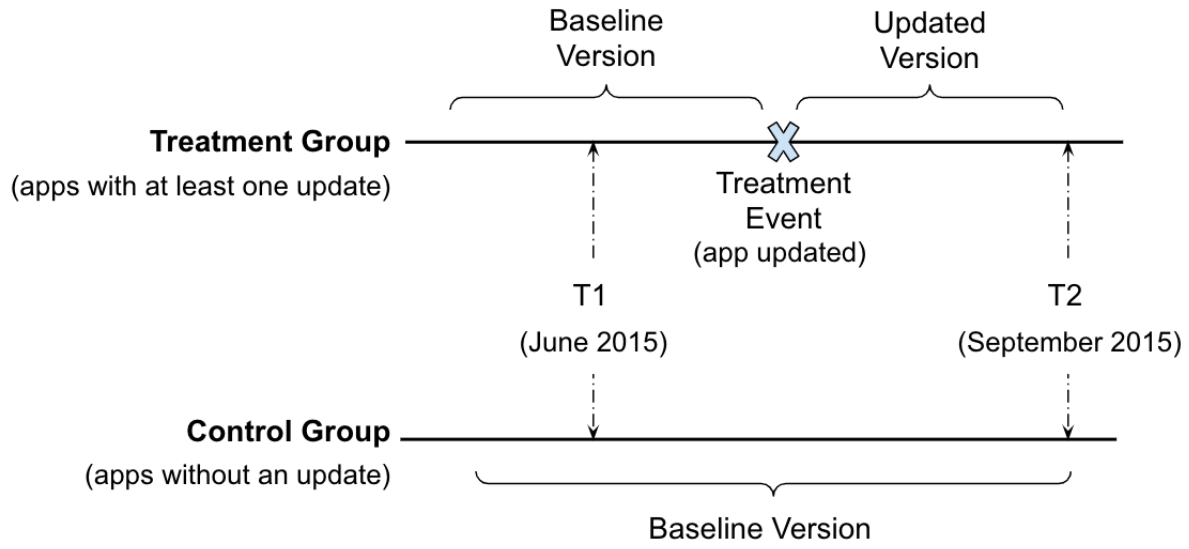

Figure 1

Once the dataset had both a control and treatment group, the next step was to find the English apps. We are not interested in doing any analysis in different languages or apps that have symbols or non-English words in their description. Because of this, we created a language detection function that specifies the language of the app description and the update description and removes any apps that do not have full English words. The final step in the data cleaning process was to find apps that have at least one rating. This portion of the data cleaning was performed because we are putting an emphasis on user engagement, so we are not as interested in apps that do not have ratings. In summary, the data cleaning process included omitting the apps that had large amounts of missing features, merging the apps from T1 and T2 into one dataset of apps, only including apps with English descriptions, and finding apps that had at least one rating. Figure 2 shows a visual, that Dr. Zhou provided, of the cleaned dataset we perform our analysis on.

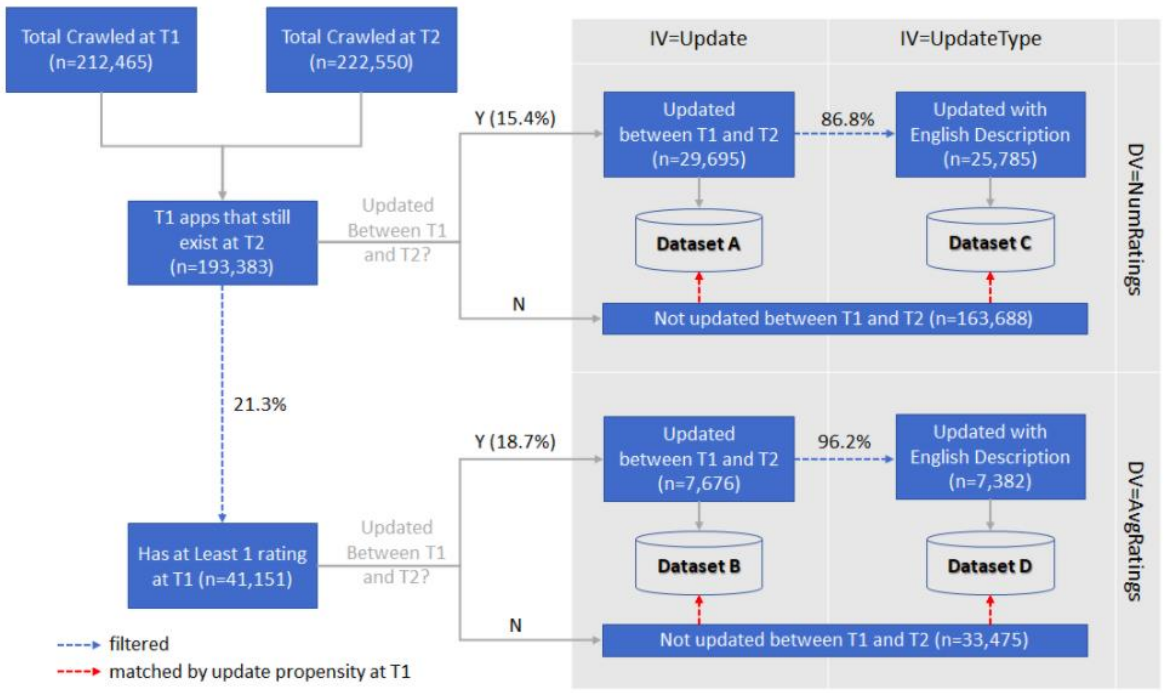

Figure 2

Analysis

Rpart

To start our analysis, we performed a simple rpart model. The first step in this process was to create a balanced sample by stratifying the data. The non-updated apps largely outnumbered the updated apps, so we made our data balanced by under sampling the non-updated apps. The next step was to split the balanced data into the training and holdout samples. Using 8 handpicked variables that we thought would best predict if the app was updated, we ran the rpart model. The following formula is the equation we used in this predictive model to predict update.

$$Update = \beta_1 RateP\_n + \beta_2 RateP\_avg + \beta_3 RateA\_n + \beta_4 RateA\_avg + \beta_5 RateC\_avg + \beta_6 RateC\_n + \beta_7 DaysSinceUpdate + \beta_8 RateC\_nPerDay$$

We then test the performance of the holdout sample to validate if our model will work for new data. After running the model, the next step is to make a confusion matrix of the predicted updated apps versus the actual updated apps. The confusion matrix allows us to find the accuracy, precision, recall, and f1 of the model. We decided that accuracy would be the metric we are going to use to best describe the performance of the model. Using the default parameters of the rpart model and the chosen features in the formula above, we found an accuracy of 65.21%. With the default complexity parameter of 0.05, rpart plot is not very complex as seen below in Figure 3.

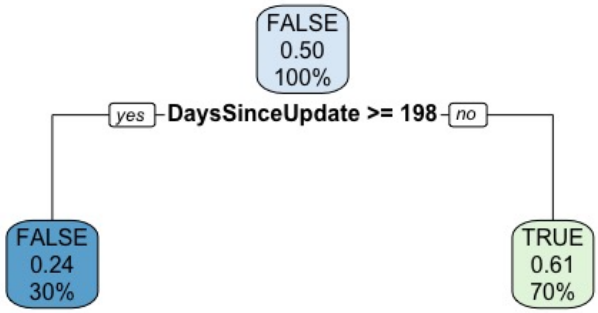

Figure 3

Now that we have a baseline accuracy, we fine tune the complexity parameters. We created a repeatable function to calculate the four performance metrics, while allowing for the complexity parameter to be set manually. To find the complexity parameter that gave us the most accurate model, we created a for loop that went through a range of complexity parameters and saved them into a data frame. We were then able to plot this data frame to visualize where the accuracy peaked. Using the plot, we found that the rpart model with these set features was more

accurate with a complexity parameter of 0.001. This model has an accuracy of 73.46%. With a much higher complexity parameter, the rpart is much more complex as seen below in Figure 4.

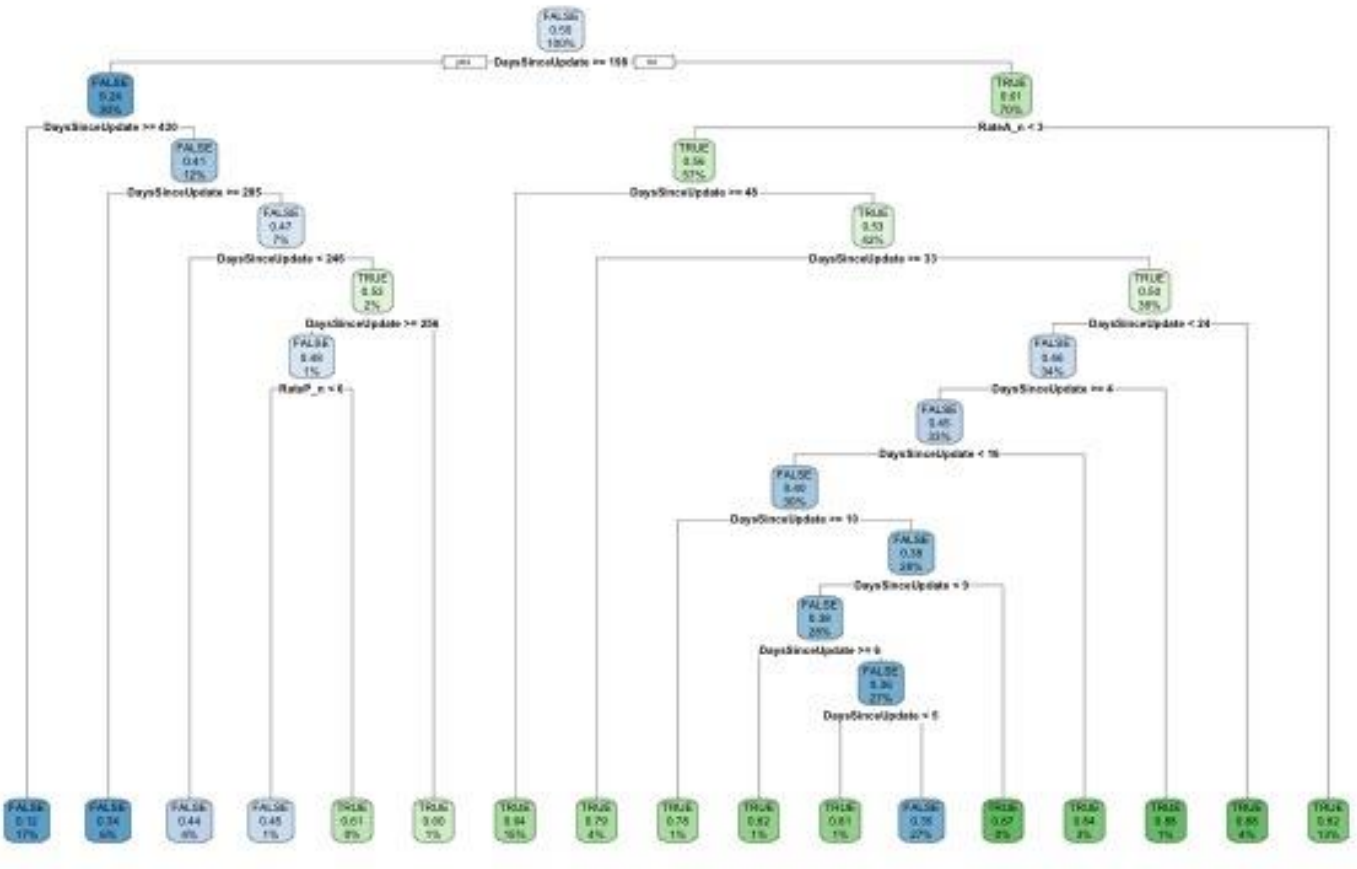

Figure 4

Weighted Rpart

Our next method was to use a weighted rpart model. After the dataset had been cleaned and merged, there are 193,383 apps. Out of those apps, only 29,695 have been updated. To create a balanced sample, we used a weighted vector. Because we made a function to find the four

performance metrics, it was easy to implement this model into that function and quickly find how well our model performed. Using a weighted vector and searching for the best fit complexity parameter of 0.020, we were only able to produce a model with 60.08% accuracy.

Support-Vector Machine (SVM)

Now that we have built up a repeatable process, we decided to dive into more complex models starting with the SVM model. In this process of prepping for more complex models, we decided to add a few more features to predict update. Here is the updated formula we use moving forward.

$$Update = \beta_1 RateP\_n + \beta_2 RateP\_avg + \beta_3 RateA\_n + \beta_4 IsFree + \beta_5 CurrentVersion\_wt + \beta_6 RateA\_avg + \beta_7 RateC\_avg + \beta_8 RateC\_n + \beta_9 DaysSinceUpdate + \beta_{10} RateC\_nPerDay + \beta_{11} Category\_new + \beta_{12} Subcategory\_new$$

All the complex models, including the SVM model, I had little to know experience with, so I got to learn them and add them to my predictive modelling toolbox. The coding to train and fit this model was different than the rpart, but the process was the same. Using the SVM model, we got an accuracy of 84.39%. This is our most accurate model found this semester.

Neural Network

The neural network model uses the same training process as the rpart model. However, this model does not handle missing values in the data well, so the data had to have a more fine-tuned cleaning than our original cleaning of our whole dataset. This made our remaining training and holdout samples even smaller than the previous models. Using the neural network model, we found an accuracy of 83.76%.

Naïve Bayes

The next model we used to predict update was the Naïve Bayes model. This model used the same package as the SMV model, so the coding was similar. It also used the same training method as the rpart model, so again, our training and holdout samples were small. The more complex models we tested on our data, the more comfortable I became with coding them. The Naïve Bayes model gave us our worst prediction of any model, with an accuracy of 58.65%.

Automated Machine Learning (Auto ML)

The final model we used this semester, was the Auto ML model. This was the most complex model we tried. The code requires the user to start an "h2o cluster", where only then, can the user start the process of training and fitting the data. I even had to update my Java version on my computer to run this code. The code to train the data was extremely different than the other models because this model requires the user to use only the h2o package. The first step in training the data was to import the data to the h2o cluster. With Dr. Zhou's help, we were able to get the code working. Using the Auto ML model, we got an accuracy of 78.63%.

Conclusion

After testing several predictive models to predict if an app was updated between T1 and T2, we found that the SVM model was the most accurate model. We learned that the more complex model is not always the most accurate, as the SVM model had a mix of complexity and simplicity. The number of features included in predicting update also has a large effect on the accuracy of each model. After playing around with the number of features to include in the predictions, we landed on the 12 features we mentioned in the second formula.

This semester, the goal was mostly understanding and cleaning the data. Looking ahead, there are many different directions we can take this project. We will investigate how large an effect each feature has on the predictions and why each feature has that effect. The three research questions we have for next semester include: (1) Will updates matter to an app's user evaluation? (2) Will update types make a difference in user evaluation? (3) Will the answer to these questions depend on additional factors (e.g., type of app, way of communication, etc.)?

Appendix 1

Feature Explanation

"AppID" - numerical ID to make organized
"AppURL" - url of the app
"Category" - text descriptors of how the app is categorized
"Subcategory" - text descriptors of how the app is categorized under the category
"AppName" - text that the app is called
"AppAuthor" - user ID of owner of the app; “By [user ID]”
"AppSeller" - user ID, sometimes same as App Author; “[user ID]
"AppDescription" - text description of app
"CurrentVersion" - version ID
"UpdateDate" - date when current version was published
"UpdateDescription" - textual description of update; Ex: “Fixed bugs”
"AppDevices" - text of what devices are compatible with app; “This app is designed for…”
"Compatible" - software required to run app; “Requires iOS … or later”
"Price" - amount app costs; “Free”, “$0.99”...
"AppSize" - size in MB of app
"Language" - choice language app is in
"RateCurrentVersion" - how many stars the current app version is rated and how many ratings; 0-5 stars; “4 and a half stars, 14 ratings”
"RateAllVersions" - how many starts the app is rated through all versions and how many ratings; 0-5 stars
"RateC_avg" - number of stars of current app version
"RateC_n" - number of ratings for current app version
"RateA_avg" - number of stars for app, all versions
"RateA_n" - number of ratings for all app versions
"CurrentVersion_wt" - weight of current version, how many ratings come from the current version versus all the other versions; "RateC_n" / "RateA_n"
"RateP_n" number of ratings for past app versions
"RateP_avg" - average rating for past app versions
"DaysSinceUpdate" - number of days since last update
"RateC_nPerDay" - how many ratings per day since app was updated
"DaysSinceUpdate_log" - log of days since app was updated
"DaysSinceUpdate_sqr" - square root of days since app was updated
"DaysSinceUpdate_ord" - range of days since last update; “[90, 120]” or “[360, 720]”
"Category_new" - cleaned version of categories to avoid confusing with subcategories
"Subcategory_new" - cleaned version of subcategories to avoid confusion with categories
"IsFree" - TRUE or FALSE if app is free
“Update” – TRUE or FALSE if app was updated between T1 and T2

Appendix 2

Confusion Matrices for Each Model

These confusion matrices show how each model predicted the apps to be updated (True or not updated (False) vs. the actual number of updated and not updated apps. The accuracy metric measures from all the classes (True and False), how many of them are predicted correctly. The precision metric measures how correct the model is out of those predicted true. The recall metric calculates how many actual True's the model captures by labelling it correctly. Finally, the f1 score is a balance between the precision and recall. F1 score is better to use if there is an uneven class distribution.

Rpart with default complexity parameters

Accuracy: 65.21%
Precision: 60.09%
Recall: 85.91%
F1: 70.71%

	Predicted	
Actual	False	True
False	2,616	831
True	3,365	5,066

Rpart with best fit complexity parameter

Accuracy: 73.46%
Precision: 75.85%
Recall: 68.27%
F1: 71.86%

	Predicted	
Actual	False	True
False	4,699	1,871
True	1,282	4,026

Weighted Rpart

Accuracy: 61.08%
Precision: 56.28%
Recall: 96.81%
F1: 71.18%

	Predicted	
Actual	False	True
False	1,546	188
True	4,435	5,709

SVM

Accuracy: 84.39%
Precision: 74.72%
Recall: 18.97%
F1: 30.26%

	Predicted	
Actual	False	True
False	3,180	45
True	568	133

Neural Network

Accuracy: 83.76%
Precision: 81.73%
Recall: 86.97%
F1: 84.27%

	Predicted	
Actual	False	True
False	377	91
True	61	407

Naïve Bayes

Accuracy: 58.65%
Precision: 92.63%
Recall: 18.80%
F1: 31.26%

	Predicted	
Actual	False	True
False	461	7
True	380	88

Auto ML

Accuracy: 78.63%
Precision: 84.43%
Recall: 75.72%
F1: 79.84%

	Predicted	
Actual	False	True
False	340	73
True	127	396

Appendix 3

Using the Latent Dirichlet Allocation (LDA) model in text analysis of app descriptions and update descriptions, we tested for app classification and copycat identification. With the LDA model, we wanted to categorize the app updates based on the words they used in their update description. We could separate the updates into categories and examine each category of update with a different perspective. For example, we could dive deeper into the bug fixes category of updates to see when and why apps are having to fix issues with bugs. The LDA model allowed us to create a document term matrix of all the words used in the update description for each app. We then cleaned this matrix of filler words (the, a, an, etc.), numbers, punctuation, and symbols. Also in the cleaning process, we removed sparse words that only showed up in the matrix a few times to only include the main ideas of each update. Finally, our goal was to narrow down the updates into three main topics. Sadly, there was a lot of overlap

between topics. We had already spent a good amount of time working on this, so we decided to move onto another part of the project.

Our second goal way to examine the data to look for copycats. When looking at the raw data, we noticed some similar patterns between apps. We noticed that some apps with the same app author or app seller had similar app descriptions even though they were not the same apps. This sparked an idea to see if some apps were copying parts of other apps' app descriptions instead of creating their own. After finding copycats, we would look into how they evolve and if they are more or less likely to be updated. We tried to identify the copycats by using a for loop that looked through pairs of app descriptions and formulated a score using the cosine function. If a pair of apps had a cosine function over a certain barrier, they would be labelled as copycats. Unfortunately, this code took far too long to run (over 1 day), so we decided to move on from this idea since it was not vital to our project goal.

Appendix 4

The final part of the project we wanted to complete this semester was making our data and functions generalizable using GitHub. Dr. Zhou and I connected our R Studios with GitHub, so we could collaborate more easily next semester. Additionally, GitHub allows other users to test their apps to see if the models we use would accurately predict if they were updated between two points in time. We created the "testDID" package in GitHub, which is currently a private depository, but the goal is to make it public one day.