Data Mining Project: Determining Mid Build in StarCraft

Group:
Noah Blumenfeld
Thomas Eliassen
Bridger Fisher
Ian Sime

Date:
12/13/17

Abstract

StarCraft is a very popular video game with a big following around the globe. Taking the first step in to playing StarCraft can seem overwhelming. We wished to improve our ability to play the game. One way to get better is to increase our ability to predict our opponent's strategy. We took a data set that contained 39 features to create a model that would predict what strategy our opponent was using. After trying out 5 different algorithms, we finally made a model using gradient boosting that gave us a 88% accuracy at predicting what the strategy was. By improving our ability to recognize a build order we can more successfully choose a strategy that will lead to our victory.

Table of Contents

Problem Description

We wanted to improve our ability to play StarCraft. StarCraft is real time strategy game where the player builds their base and maneuver their troops. Maneuvering your troops are known as the micro game, and building your base is known as the macro game. Depending on your strategy, the order of which you build buildings will change, this is known as build order. A good player needs to know what strategies work best against the enemy's strategy. If we want to improve our game, knowing what strategy the enemy is using will give us a big advantage. We decided to tackle this problem by looking specifically at the Protoss race. There are three races in the game: Protoss, Terran, and Zerg. Protoss are technologically advanced and physically strong. Terrans are basically humans, but they are split into factions because of this they fight themselves as well as other species. The Zerg are not technologically advanced and rely on genetic traits. StarCraft is a complicated game, but by improving our knowledge about enemies build orders we can improve our game significantly.

Data Collection & Preprocessing

The data we used was published by Google's Deepmind. We took a very small portion of that dataset for our project. StarCraft is a huge game with many different decisions, starting with which race you play with. The race our dataset is focused on is the Protoss race. Protoss are physically strong and extremely intelligent. Our dataset is focused on the build order, which is a big part of the overall strategy. This means it is focused on the buildings, units produced, and upgrades that you can build during a game. Our data contains 7 different classes, known as midBuild, that represents 6 different build strategies the player can choose to follow during the middle part of the game.

To preprocess this data we replaced outliers with mean values. We used a quantile of 97%, defining outliers as data points falling in the outside 3%. However, we did leave the 0 values as 0's. We did this because the 0's represent a decision made in gameplay. If there is a 0, it shows that a player decided to upgrade, or build a different unit or building. If we were to drop those 0's we would lose important information about the decisions being made in-game. This would affect our model's accuracy, because the 0's help determine the midDuild. We also cut the number of features from 57 to 39. We found that there were many redundant features, such as ProtossAirArmor2 and ProtossGroundArmor2. These are upgrades in the same path and the only way to upgrade ProtossGroundArmor2 is to upgrade ProtossGroundArmor1 first, due to this we dropped ProtossGroundArmor2. There are multiple features like this, that we from our knowledge of the game seems redundant.

Data Mining Problem

To improve our game we decided to use this dataset as a classifier where we can learn what buildorder the enemy Protoss player are most likely to choose. As our big dataset has information about Protoss buildings and upgrades with labels that names different Protoss build orders. We can by treating it as a classification problem learn to recognize these build, such that we can react appropriately. Since the data was gathered to be used in an Artificial Intelligence development project it is perfectly built to be a classification problem and we only had to adapt and specify the data.

Algorithms

We started with 5 different algorithms we thought might give us the best results: Decision Tree, Random Forest, Gradient Boosting, KNN and Logistic Regression. Testing all the different parameters possible would take up hundreds of lines of code if we were to hard code it, so we decided to use the grid_search method in order to make this process much easier. We then chose out a few parameters from each classifier that we thought would be interesting and ran each through grid_search so we could obtain what the most optimal setup was for each algorithm (see our step 2 code for the specifics about what parameters we chose for each classifier). After running our code, we found that Decision Tree, Random Forest, and Gradient boosting were by far the best algorithms to run, while KNN and Logistic Regression were significantly worse than the other 3, even when they were optimized to use the best parameters. Looking at this, it is very logical that those 2 were perform much worse.

The data we have does not "cluster" in a way that would allow for KNN to give us high accuracy. Our data is also split into 7 different classes that we are trying to predict, which also explains why logistic regression could only get up to about a 60% accuracy. Since we didn't run multinomial logistic regression, we could never hope for anything higher than that since a significant amount of the rows were always going to be labeled the wrong class. This left us with Decision Tree, Random Forest and Gradient Boosting. While they were all very close in accuracy, Gradient Boosting was about 3% better than the others in all cases.

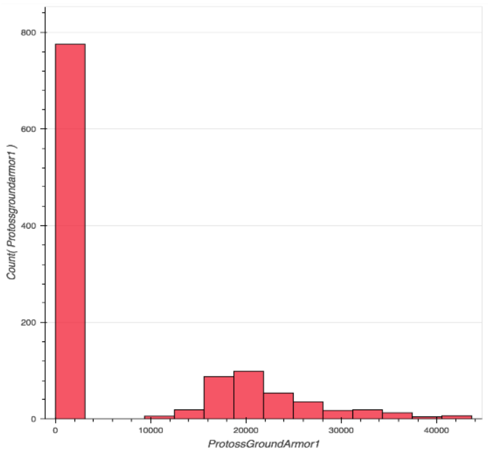
Best Model

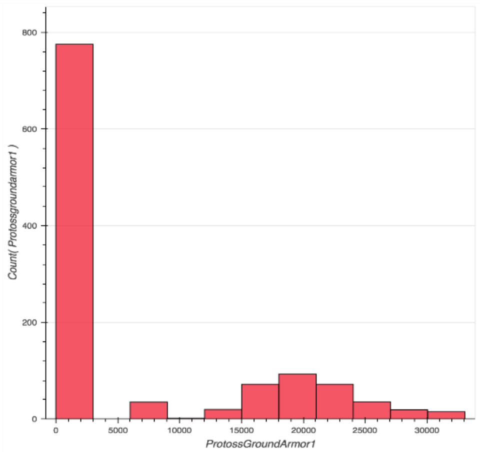
Gradient boosting essentially takes one tree and improves the parts of it that aren't accurate in order to make it as optimized as possible. Which is why it make sense that it performs better than Decision Tree and Random Forest. However, both Decision Tree and Random Forest would both be viable options to use for our data set, but as we were looking to use only 1 algorithm we decided that gradient boosting would be the best one for us to use. The gradient booster gave us a high accuracy and from our experience with the game, the attributes Gradient Boosting picked for most important feature makes sense. This because all the features featured describes important parts of different builds.

Results

Distribution Graphs:

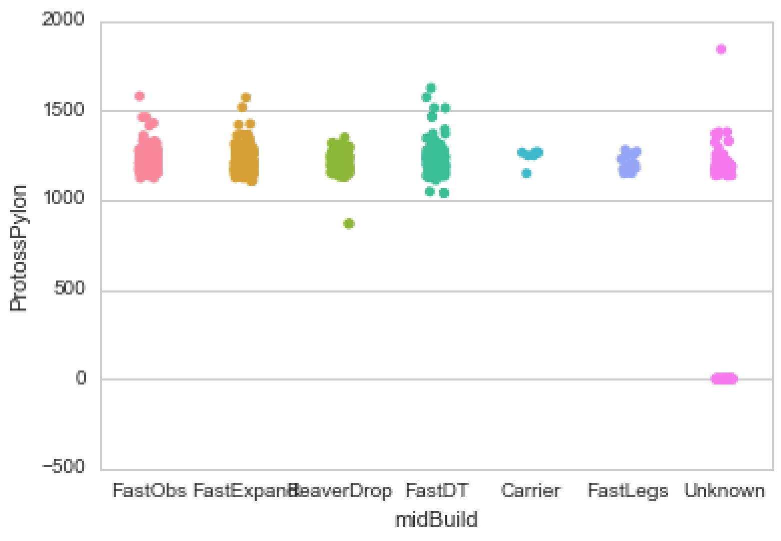
Before processing data: After processing data:

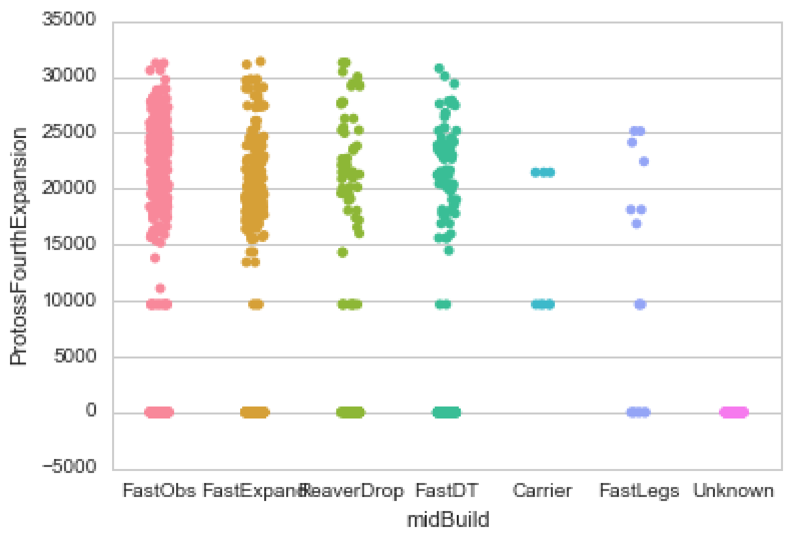




The dynamic histograms we plotted show the distribution of the data for a given feature. Every graph has a large column for the 0 value, and a good distribution other than that. The reason that the 0s are still there after preprocessing the data is because they are extremely meaningful 0s. The 0s represent decisions that are made in gameplay. Choosing a different upgrade or unit creates a 0 in certain features. For example if the player chooses to upgrade ProtossGroundArmor1 the ProtossArmor1 feature would receive a 0. If we were to replace these 0s with the mean for the data valuable information would be lost. As stated earlier, for step 3 we filled in the outliers with the median of the column, therefore preserving the rest of the information the row had to tell us and this is why there is a new column that appears between the massive 0 column and the start of the actual data.

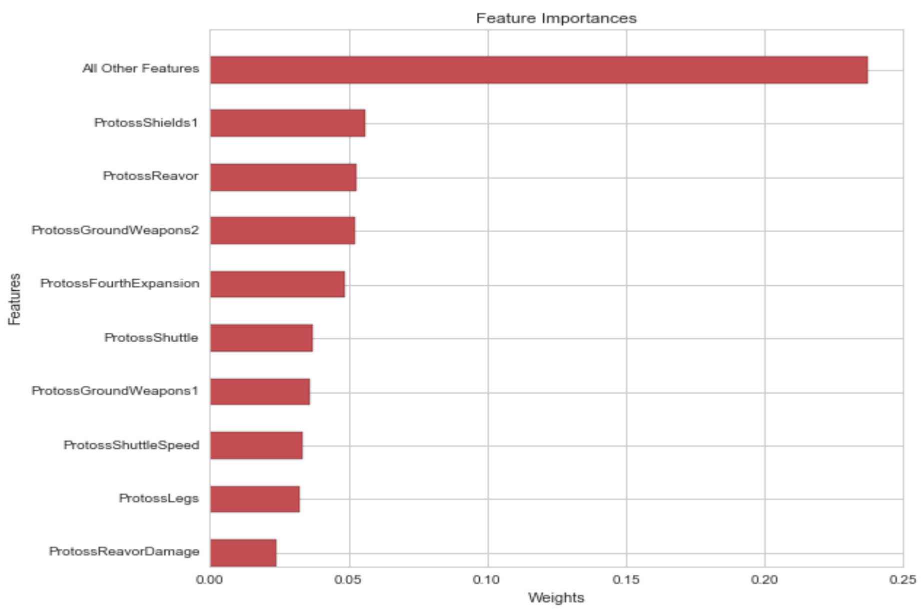
Categorical Scatter Plots:



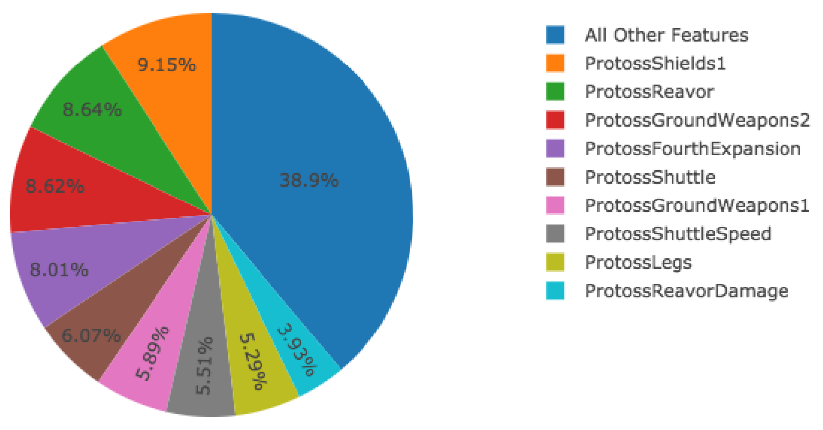


The Seaborn Categorical Scatterplot shows the relationship of the occurrences of one attribute and its classifications at the midBuild, or halftime, of the game. The x-axis is the classification of strategy taken by the player at the midBuild, or halftime, of the game. The y-axis is the number of occurrences of that attribute. Then overall the scatter plot will show the relationship between the counts of each attribute and its classification. Therefore we can take each attribute and show the occurrences of each classification and look for patterns. The pattern we are looking for is if there are attributes that are most often categorized as a certain strategy or alternatively not very likely to be classified as a certain strategy.

Feature Importances:



This graph shows the importance of the features from the gradient boosting algorithm. One of the most important features is ProtossShields1. ProtossShields1 is an upgrade to protoss units in the game. It increases the amount of damage they can take before they are killed. This is important because the stronger your units are the easier it becomes to eliminate the enemy units without losing your own. Another important feature is is ProtossReavor. ProtossReavor describes a ground unit that is used mostly as a siege unit. The reavor has a slightly longer range than most defense units. To win in StarCraft you need to kill all of your enemy's units and buildings. This makes the reavor an important unit because it allows you to destroy your enemy's bases and defense units while their defense units can't hit them.



This chart, like the bar graph above, shows the importance of certain features. It shows the importance of the top 9 features individually and shows the importance of the other 30 features combined. The most import feature appears as the second largest piece of the pie chart due to the grouping of the other 30 features. The grouping of the 30 other features has an importance of ~ 39%. The importance of ProtossShields1 is 9.15% and the importance of ProtossReavor is 8.64%. It makes sense that these features are the most important, the grouping of 30 makes sense because it is the sum of the importance for 30 features. The ProtossReavor is an important unit in the game due to its range and ProtossShields in an important upgrade in the game.

Improved Model

For step 2 we preprocessed our data by removing outliers. We first used a quantile of 97% to define our outliers and completely removed the non-normal outliers from our data. The outliers were the values that were outside the normal distribution, the leftover 3% which is about 400 rows of data. We then split up the data into 80% for training data and 20% for testing data. We continued with each of our models and used grid search to find the most accurate on all 57 attributes. Our highest accuracy was 93.2% with gradient booster. We then decided that we needed to take another look at the preprocessing in step 3. After discussing the preprocessing, we found that many of the values we had deleted yielded important information. We decided in step 3 to still use a quantile of 97% to separate out the outliers and simply replace these values with the mean value of the data, instead of completely removing it. This way we keep almost 400 rows of data and we can still learn valuable information. We also decided to cut down on some of the attributes. Our rationale was that many of the attributes are simply upgrades of the same attribute and that focusing on choosing that attribute would simplify the data immensely. Therefore, we removed the non-essential upgrade attributes and continued testing the models with 39 attributes, randomly dividing our data into 80% for training and 20% for testing. The best final accuracy was 87% on the test data for gradient boosting. This accuracy was a little less but still very high, not as overfit, and makes our model much better.

Summarize

We chose a dataset based on stats from the mid-game in StarCraft and are trying to use these stats to predict what build the player is choosing at this point. This is a very important part of the game, as choosing a certain build can either set you up for failure or success depending on what your opponent’s build is. After going through multiple different types of algorithms, we chose Gradient Boosting as the algorithm to model our data. This gave us an accuracy of about 88%, which means that if we were to have access to all the needed attributes during a game we would have an 88% chance that the build our model predicted would be correct. This would be huge for a player during a game, and while this is something that is more likely than not to be unrealistic to use in real time it is something that could help with the analysis of games after they were played, such as a pro team seeing what choices were good/bad to do with the information that they had.

Going forward it would be interesting to apply this to the early as well as end of the game to see if any valuable data could be pulled from those areas. Using algorithms to search for this type of information within data can be invaluable to players trying to analyze their choices in games and optimize their strategy to become the best player in the world.