Predicting Bazel Build Times Using Machine Learning

Muhammad Umar Nadeem
Stanford University
munadeem@stanford.edu

Sarah Raza
Stanford University
sraza007@stanford.edu

1. Abstract

Similar to other tools like Make, Ant, Gradle, Buck, Pants and Maven, Bazel is a tool that automates software builds and tests. To be able to support developer's daily activities, the build service system runs millions of builds per day. It is a known issue in Bazel that a build with many targets could exceed its execution deadline. This is problematic because it reduces the developer's productivity and maintains high energy and resource consumption. In this paper, we outline a linear regression algorithm with feature crossing that predicts the CPU-time of a Bazel build. This model will allow developers to design code for efficient use of resources in advance of executing the build.

2. Introduction

Bazel, initially designed to fit the way software is developed at Google, is an open source tool that automates software builds and tests. It has the following key features: multi-language support, high-level build language, multi-platform support, reproducibility, and scalability. These key features directly and positively correlate with the tool's growing customer base.

In light of expanding use of Bazel, it has become a known issue that a build with many targets could exceed its execution deadline. This translates to the fact that the build system often times out without fully executing on deeply structured codebases. This is problematic because it reduces a developer's productivity by requiring recurring redevelopment after time wasted waiting for the build to execute. This is also problematic because it unnecessarily uses a large amount of computing resources. Through a lens of environmentalism, the scale of the technology industry means that wasted energy on tools as fundamental to all code as build-systems, especially one with a growing user base, has lasting, large-scale negative impact.

In this paper, we discuss the development of an algorithm that, when trained on the commits made to the codebase of a particular company or project using Bazel as build-system, can predict the amount of CPU time required for the codebase to build after a new commit. More specifically, the inputs to our algorithm are the following: the most common prefix of the file paths of the files changed in a specific commit to a codebase, and the most common file type changed in that same commit to a codebase. We then use a linear regression model with feature crossing to output the predicted CPU-time required to build the code after that commit. This model will allow developers to predict Bazel build times and thus account for inefficiencies in advance of executing builds.

3. Related Work

The most relevant paper in this field is "Smart Build Targets Batching Service at Google" by Wang et al. This paper is similar to our work in that it also focuses on using a Bazel build system and linear regression to predict metrics that will ultimately help optimize builds. It differs in its focus on creating a batching algorithm that alters the way Bazel itself works rather than predicting the CPU-time of the build. Another paper that discusses Bazel build-systems is "Using Remote Cache Service for Bazel" by Lam et al. While this paper also focuses on providing a solution to reduce lengthy builds, it proposes the development of a remote cache to boost performance rather than predicting CPU time. Unfortunately, this solution is not universal

because it can be expensive and time consuming to create a remote execution farm at the scale necessary to cause a tangible impact.

The paper "Fast distributed compilation and testing of large C++ projects" by Matev et al. also focuses on optimizing build time, but more specifically in the context of C++ projects. Therefore, this paper was limited in its ability to be useful for companies with codebases in any other language.

The paper "Scalable build service system with smart scheduling service" by Wang et al. also focuses on optimizing build time. This paper offered interesting insights into how a company can be more efficient, but only for large scale firms that could afford the time and resources required to implement it.

Finally, it is important to highlight an additional paper by Harmon et al. titled "A retargetable technique for predicting execution time of code segments." This paper focused specifically on predicting build metrics, and it was similar to the first two papers because it presented a technique that predicts point-to-point execution times on code segments. It is similar to this paper because it is focused on predicting execution time, however it differs in the fact that it only predicts execution time for segments of code. Also, the paper differs because it does not use linear regression as its machine learning algorithm of choice. Instead, the researchers develop their own method called micro-analysis, a method that utilizes machine-description rules.

4. Dataset and Features

Our dataset consisted of information on 500 different builds in the open source bazelbuild/bazel project hosted on Github. Because each new commit to the codebase requires the project to rebuild, we considered one commit equivalent to one build. In order to save the relevant input and output information, we had to locally build each commit and thus computing resources limited us to have 300 examples of training data, 100 examples of validation data, and 100 examples of testing data.

4.1 Raw Input Data

Our first step to obtaining the data was saving it from the commits. As our input data, we recorded the prefix of the file paths for the files changed in each commit and the types of the files changed. To obtain this data, we created a python script that ran a "git diff" command 500 times - once for each commit - and recorded the information into a .csv file. The different options for prefix and file type are shown to the right:

While we considered using different data as our input, such as the number of files changed, we ultimately concluded file prefixes and type were the best predictors. To better understand why we chose file prefixes, consider a C++ project in which you change one header that is included in hundreds of files in the code base. In contrast, consider that instead you change ten files that are not included anywhere else in the code base. The former will take a lot longer to build than the latter. In general, which folder a file is located in tends to be an indicator of how important it is to the code base: a file in third_party is likely to be a dependency used in one other file where as a file in src/main is likely to be essential to the functionality of the codebase and therefore used much more frequently. Thus, we settled on using files changed as our input. We additionally use file type (shown to the right) because lower level languages such as C++ tend to compile faster than

languages such as Python. This intuition for what input features was backed up by the paper titled "Smart Build Targets Batching Service at Google" by Wang et al.

4.2 Raw Output Data

Due to the implementation of a supervised learning algorithm to predict CPU-time, it was important to record the CPU-time for each commit. To do this, the following command was used to save the Build Event Protocol (BEP) for one commit: "bazel build //src:bazel-dev --build_event_json_file=path_to_save_file". The BEP is a .json file containing a variety of information about the build. Within the buildMetric field of the BEP, there is a field called timingMetric that includes the CPUTime in milliseconds. A python script was used to save the BEP for each commit and then traversed each of the .json files to fill a .csv file with the CPUTime for each commit. An example of some of the output data is shown to the right:

2440
7844
10158
12167
1459
1629
7754
23810
5638
4021

4.2 Preprocessing Data

The next step in preparing the data was converting it into an information type compatible with linear regression. This was accomplished through Bucketization.

To train the linear regression model, the string input features were converted into discrete, integer features. This was done using tensor flow CategoricalColumns with in-memory vocabulary. The vocabulary list for the prefixes was: "bazelci/", "examples/", "scripts/", "site/", "src/conditions", "src/java_tools", "src/main", "src/test", "src/tools", "third_party/", "tools/". The vocabulary list for the types was: "JAVA", "C/C++", "Starlark", "python", "HTML/CSS/JS."

5. Methods

5.1 Linear Regression

In machine learning, regression analysis is a set of statistical processes for estimating the relationships between a dependent variable (denoted as y) and one or more independent variables (denoted as x). It tries to find the line that most closely fits the data according to a specific mathematical criterion. For example, the method of ordinary least squares computes the unique line (or hyperplane) that minimizes the sum of squared distances between the true data and that line (or hyperplane).

Given a set of data set $\{y_i, x_{i1}, ..., x_{ip}\}^n_{i=1}$ of n statistical units. A simple linear regression model has the following form: $y_i = \beta_0 + \beta_1 x_{i1} + ... + \beta_p x_{ip} + \varepsilon_i = x_i^T \beta + \varepsilon_i$, i = 1,...,n where T denotes the transpose, so that $x_i^T \beta$ is the inner product between vectors x_i and β.

Fitting a linear model to a given data set usually requires estimating the regression coefficients β such that the error term $\varepsilon_i = y_i - x_i^T \beta$ is minimized across all n samples. For example, it is common to use the sum of squared errors $\sum^n_{i=1} \varepsilon^2_i$ as the quality of the fit.

It is important to note that linear models can be efficiently trained using stochastic gradient descent.

5.2 Feature Crossing

A feature cross is a synthetic feature formed by multiplying (crossing) two or more features. Crossing combinations of features can provide predictive abilities beyond what those features can provide individually. As a consequence, feature crosses can help us learn non-linear functions using linear regression. A well-known example is that the XOR function f(x,y) where x,y,f(x,y) ∈ {0,1} is not linearly separable and it cannot be written as f (x, y) = αx + βy + γ where

α, β and γ are real numbers. However, the XOR function can be written as f (x, y) = x + y − 2xy, where the xy term is a feature cross for x and y.

In practice, machine learning models seldom cross continuous features. However, machine learning models do frequently cross one-hot feature vectors for better results. For example, suppose we bin latitude and longitude, producing separate one-hot five-element feature vectors, e.g. [0, 0, 0, 1, 0]. Creating a feature cross of these two features results in a 25-element one-hot vector (24 zeroes and 1 one). The single 1 in the cross identifies a particular conjunction of latitude and longitude. By feature crossing these two one-hot vectors, the model can form different conjunctions, which will ultimately produce far better results compared to a linear combination of individual features.

5.2 Linear Regression and Feature Crossing in Context

After preprocessing our data and bucketing the file fragments so that they represent numerical data, we ran the linear regression model as described above, using stochastic gradient descent as our optimizer. To further increase accuracy, we crossed the two inputs (prefix and type) and retrained the model.

6. Experiments, Results, & Discussion

6.1 Experiments

In our experiments, we focused on tuning two hyperparameters: learning rate and batch size. We did this using grid search cross-validation with 3 folds. In the end, we found that the optimal learning rate was .01 (the default value given by SGD) and the associated batch size was 16. Our conclusion was backed up by existing research looking at the relationship between batch size and learning rate. Specifically, in the paper "Cyclical Learning Rates for Training Neural Networks," Smith et al. found that according to their research the minimum amount of loss for a learning rate of .01 occurred when the batch size was 16.

The primary metric for accuracy and precision was root mean squared error (RMSE). This is a measure of the standard deviation of the residuals, where residuals are a measure of how far from the regression line the data points are. The lower the RMSE, the better fit the linear model of regression has. The following is the RMSE equation, where f = forecasts (expected values or unknown results), o = observed values (known results), and the bar above the squared differences is the mean.

$$RMSE = \sqrt{\overline{(f-o)^2}}$$

We used RMSE to help us understand not only how effective our model is, but also whether our linear regression was more successful with or without feature crosses.

6.1 Results

As shown below, the root means squared error was lower when the model was run with feature crosses. Quantitatively, we observed this because RMSE was numerically lower with feature crosses. Qualitatively, we observed this because the training and validation lines looked more similar when feature crosses were used.

RMSE for linear regression without featurization for the training data (table to the right). Average: 5735.84 milliseconds.

RMSE without Featurization

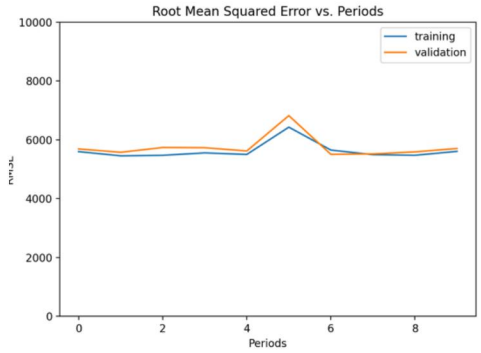
period 01:	6890.14
period 02:	5997.34
period 03:	5482.53
period 04:	5442.71
period 05:	5588.51
period 06:	5506.91
period 07:	5842.11
period 08:	5442.39
period 09:	5429.90

RMSE with Featurization

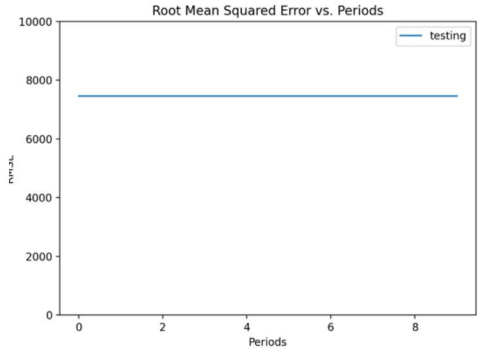
period 01:	5456.73
period 02:	5473.42
period 03:	5556.61
period 04:	5503.59
period 05:	6430.74
period 06:	5652.71
period 07:	5495.58
period 08:	5474.57
period 09:	5608.88

RMSE for linear regression with featurization for the training data (table to the left). Average: 5628.09 milliseconds.

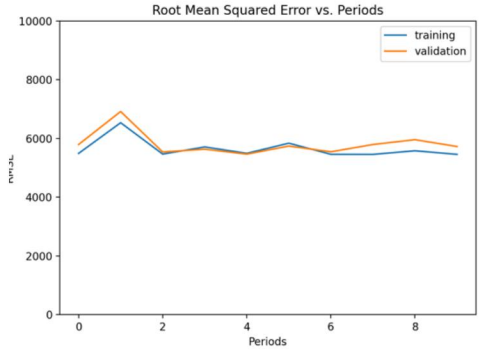
RMSE for linear regression without featurization for the training and validation data:



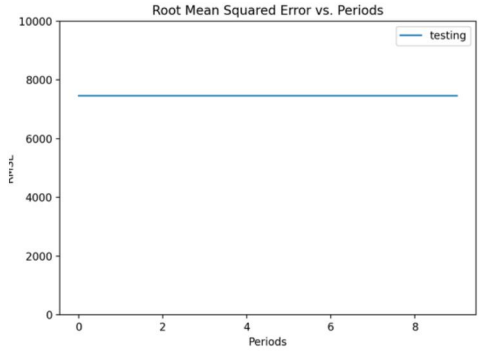
RMSE for linear regression without featurization for the testing data:



RMSE for linear regression with featurization for the training and validation data:



RMSE for linear regression with featurization for the testing data:



6.2 Discussion

There are two major signs of overfitting. The first is if the performance of the model on the training dataset is significantly better than the performance on the validation dataset. The second is if the model's performance on the validation dataset improves to a point and then begins to get worse. In our case, neither of these occur so we do not have overfitting.

Additionally, linear regression was more successful (there was a lower average RMSE), albeit slightly more successful, with feature crosses. This makes sense because feature crosses can represent a non-linear relationship with linear regression and the relationship between file prefix + type and CPU time is not necessarily linear.

7. Conclusion

This paper outlined a linear regression algorithm with feature crossing that is able to predict the CPU-time of a Bazel build. This model will allow developers to design code for efficient use of resources in advance of executing the build.

That being said, there is room for improvement. Given that most of our CPU times were in the tens of thousands of milliseconds, our root mean squared error being in the 5000-6000 millisecond range is acceptable but less than ideal. To improve our results, we would have to collect more data. As of right now, we are using 500 commits worth of information to represent the build metrics for almost 30,000 commits. Since we simply selected the 500 most recent commits, it is unlikely that we have representative data of the entire build system. Similarly, the granularity of our collected data was limited to *most common* prefixes and file types by computing power. Thus, given more time and resources in the future, we have means to train a more successful model.

8. Acknowledgements

We would like to thank Professor Moses Charikar, Professor Chris Ré, Head TA Ian Tullis, and the rest of the CS 229 staff for their support and guidance.

9. Contributions

Sarah Raza established rapport with our project mentors at EngFlow, the startup we partnered with, and developed a high-level understanding of the project motivations. She wrote the python scripts for collecting data from open source Bazel commits. She also contributed heavily to the development of the machine learning model through implementation of feature bucketization and crossing.

Muhammad Umar Nadeem worked to parse the dense information regarding build systems and their build metrics into the target dataset for the machine learning algorithm. He also contributed heavily to the development of the machine learning model through implementation of data sorting, the training model, and the testing model.

Both contributed equally to the authorship of this paper.

10. References

D. C. Montgomery, E. A. Peck, and G. G. Vining. “Introduction to linear regression analysis.” 2012.

Harmon, Marion G., Theodore P. Baker, and David B. Whalley. "A retargetable technique for predicting execution time of code segments." *Real-Time Systems* 7.2 (1994): 159-182.

Lam, Alpha. "Using Remote Cache Service for Bazel: Save time by sharing and reusing build and test output." *Queue* 16.4 (2018): 31-43.

Matev, Rosen. "Fast distributed compilation and testing of large C++ projects." *EPJ Web of Conferences*. Vol. 245. EDP Sciences, 2020.

Smith, Leslie N. “Cyclical Learning Rates for Training Neural Networks.” *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)* (2017): 464-472.

Wang, Kaiyuan, et al. "Scalable build service system with smart scheduling service." *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2020.

Wang, Kaiyuan, et al. "Smart Build Targets Batching Service at Google." *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021.

11. Code

The code for this project can be found at: https://github.com/umar-nadeem/CS229-Final-Project