An Integrated Development Environment for Faster Feature Engineering

Michael R. Anderson Michael Cafarella Yixing Jiang Guan Wang Bochun Zhang
University of Michigan
Ann Arbor, MI 48109
{mrander, michjc, ethanjyx, wangguan, bochun}@umich.edu

ABSTRACT

The application of machine learning to large datasets has become a core component of many important and exciting software systems being built today. The extreme value in these *trained systems* is tempered, however, by the difficulty of constructing them. As shown by the experience of Google, Netflix, IBM, and many others, a critical problem in building trained systems is that of *feature engineering*. High-quality machine learning features are crucial for the system's performance but are difficult and time-consuming for engineers to develop. Data-centric developer tools that improve the productivity of feature engineers will thus likely have a large impact on an important area of work.

We have built a demonstration integrated development environment for feature engineers. It accelerates one particular step in the feature engineering development cycle: evaluating the effectiveness of novel feature code. In particular, it uses an index and runtime execution planner to process raw data objects (*e.g.*, Web pages) in order of descending likelihood that the data object will be relevant to the user's feature code. This demonstration IDE allows the user to write arbitrary feature code, evaluate its impact on learner quality, and observe exactly how much faster our technique performs compared to a baseline system.

1. INTRODUCTION

Many of today's most compelling software systems, such as Google's core search engine, Netflix's recommendation system, and IBM's Watson question answering system, are *trained systems* that use machine learning techniques to extract value from very large datasets. They are quite difficult to construct, requiring many years of work, even for the most sophisticated technical organizations. One reason for this difficulty appears to be *feature engineering*, which we have discussed previously [1].

A feature is a set of values distilled from raw data objects and used to train a supervised machine learning system. For example, consider a Web search engine that uses a trained regressor to estimate the relevance of a given Web page to a user's query; suitable features might include whether the user's query appears in italicized type, whether the user's query is in the document's title, and so on. Human-written features are generated by applying user-defined feature code to a raw data object. Good feature code is not only correct from a software engineering point of view, but it also produces a useful feature for the machine learning task at hand.

Features have two important characteristics. First, good features are crucial for the overall performance of trained systems [1]. Public accounts from the engineering teams of successful trained systems indicate that their success comes from slow development of many good features, rather than statistical breakthroughs [5, 7].

Second, features are difficult for engineers to write. Because the raw input data is large and diverse (as in a Web crawl), the "specification" of the feature is always unclear. It is often difficult for the engineer to predict whether a particular feature will in fact improve the learned artifact's accuracy; the programmer may struggle to implement a feature function that will ultimately serve no useful purpose. As a result, feature development involves many small code-and-evaluate iterations by the engineer.

Unfortunately, each such iteration can be time-consuming. Every evaluation entails applying the novel feature function to a massive input dataset, using the resulting feature data to train a machine learning artifact, and then testing the artifact's accuracy. Our demonstration integrated development environment (IDE) aims to speed up this code-and-evaluate loop. Its intended impact on engineer productivity is designed to be akin to that of giving the engineer a faster compiler: she will not necessarily write better code, but she should be able to evaluate the effectiveness of her code more quickly. Our system should thus improve the productivity of feature engineers, thereby enabling trained systems that are either more accurate or less expensive to construct.

Modern Feature Engineering — Today there is no explicit support for feature engineering in most software stacks. When developing novel feature code, a feature engineer uses the following evaluation sequence:

1. The engineer uses a bulk data processing system, such as MapReduce [4], to apply the feature functions to a large raw dataset. This job is very time-consuming due to the large input, but it produces the desired set of feature vectors.

2. The feature vectors are used to train a machine learning artifact, generally using a standard software toolkit,

Proceedings of the VLDB Endowment, Vol. 7, No. 13

such as Weka [6]. The precise runtime cost of this step is dependent on the chosen machine learning technique.

3. The resulting artifact—say, a text classifier that determines whether a news story is related to *politics*, *sports*, or *entertainment*—is evaluated for accuracy using a holdout set of labeled data. If the artifact's accuracy is not sufficient, the engineer modifies her feature code and returns to the first step. If the artifact's accuracy is acceptable, the engineer's job is complete.

The runtime costs of machine learning, reflected in the second step here, are well-known and are an active area of research. We focus instead on the first step: the time spent in bulk data processing. Current systems simply process the entire raw input dataset in an arbitrary order, and always process the data in full. If instead the bulk data processor chose to process the most promising raw inputs first, the engineer could terminate the processing task early and still be able to evaluate the current iteration of the feature code.

Technical Challenge — The technical core to our work is a bulk data processing system that performs *input selection optimization*. Instead of blindly processing every raw input, our processor chooses inputs that will best exploit the user's feature code. For example, if the user's feature code is most useful when applied to news-oriented Web pages, the processor should locate news pages for processing first. In contrast, if the user's feature code is designed to predict e-commerce prices, the processor should instead focus on pages from Amazon.com. Because the feature code changes with every invocation of the system, the "usefulness" of an input can change with each iteration; preprocessing the input data to determine input utility is impossible, so the system must locate and choose useful inputs on the fly.

This problem is somewhat similar to that of *active learning*, in which a learning system is given a budget of supervised data labels [8]. The system must choose which training feature vectors to label that will maximize overall improvement to the artifact's accuracy. Unlike active learning, however, we wish in our setting to avoid the time costs associated with obtaining the feature vectors themselves. This difference makes most active learning techniques irrelevant.

Our technical solution combines a runtime planner with domain-independent indexing and is called ZOMBIE. Its technical details are described in a separate paper currently under preparation. This paper describes the working IDE that is built on top of the ZOMBIE technique.

Demonstration — Our IDE allows the user to enter arbitrary feature function code and then apply that function to a large input set of Web pages. The user can process the features using either a standard BASELINE data processing system or using our ZOMBIE system. All tasks run on a configurable number of machines on Amazon's EC2 service.

In the rest of this paper we will provide an overview of our system's approach to evaluating feature code and its architecture (Section 2). We also describe its user interface (Section 3) and details of our live demonstration (Section 4).

2. SYSTEM FRAMEWORK

We now examine our feature development system's inputs and outputs in more detail.

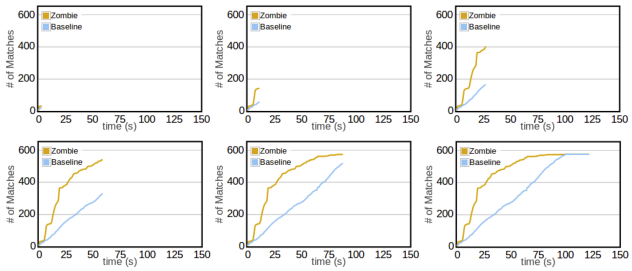

Figure 1: Snapshots over time of ZOMBIE's performance compared with BASELINE.

2.1 Inputs and Outputs

In addition to novel feature function code, an engineer who uses the feature IDE must supply a handful of parameters to configure the overall learning task:

- A large raw dataset, such as a Web crawl. The IDE will obtain feature vectors by applying the user's functions to each element in this dataset.
- A method for providing a supervised label for the generated feature vectors. This could be a database of human-provided labels or an algorithmic "distant supervision" technique [9].
- A machine learning training procedure that consumes the labeled feature vectors and produces a learned artifact, such as a naïve Bayes classifier. This procedure is likely drawn from a machine learning library.
- A method for scoring the learned artifact's quality. This could simply measure the accuracy of the learned artifact over holdout set of labeled training vectors.

The engineer's feature code will change repeatedly. In contrast, these four parameters will remain static for the life of an engineering task. They define the environment in which the user's feature code must be successful. For our demonstration, we will preconfigure these values to describe a four-way classification task that consumes a dump of Wikipedia, uses synthetic labels derived from semantic tags on those pages, and employs multiclass naïve Bayes classification. While this is a simple task designed to allow demo users to quickly complete a full evaluation sequence, it takes full advantage of the behavior of the ZOMBIE system.

When the user has completed a draft of the feature code and is ready to begin Step 1 of the above evaluation sequence, she clicks "Go" on the IDE's main interface. The feature development system then applies the feature function code to each element in the raw input set. Since the raw input set can be extremely large, this *function application* step is often very time-consuming. A standard feature development system might use MapReduce; our IDE uses the ZOMBIE system for choosing which raw inputs to process first.

Runtime Output — One way the engineer can save time is to terminate function application early and simply train the artifact using whatever feature vectors were in fact generated. Doing so with the BASELINE system will likely cause many high-value vectors to be dropped from the output. ZOMBIE, however, locates and selects the most useful items early in

the function application process, so that early termination will lose comparatively fewer high-value vectors. For a given amount of processing time, our ZOMBIE-powered IDE can produce a set of training vectors that is more useful than that produced by MapReduce's blind scans of the input set. Of course, there is no secret raw data available to our our system: if it is run to completion, BASELINE and ZOMBIE will produce identical training sets.

Figure 1 shows graphs for a single run of the IDE on a single set of user-written feature code. The x-axis is time spent in feature function execution, and the y-axis shows the number of high-value raw data items processed so far. In this dataset 0.5% of the items are high-value, but a real-life dataset may likely have even fewer: consider a task requiring web pages of computer science faculty. Given a crawl of roughly 5 billion pages and an estimated 50,000 computer science faculty in the US, the high-value percentage falls to 0.001%. Even limited to academic domains, the high-value percentage would be far less than 1%.

Describing raw inputs as "high-value" or "low-value" is a helpful but unnecessary simplification. Our system also works with data items that have fine-grained differences in utility. Further, it works with pages whose utility depends on whether similar items have been seen previously. For example, a web page from a faculty member of a yet-unseen institution may have more utility than one from an institution that has already provided several pages.

The tan upper line of Figure 1 describes the raw data items processed by ZOMBIE, while the lower blue line describes the raw data items processed by BASELINE. As the IDE runs, it continually updates the graph with new results. The engineer can use this information to decide when the generated training vector set is "good enough" and then terminate the function application. Clearly, the ZOMBIE line indicates it is able to find many more high-quality raw inputs early on; a user who terminates after, say, 25 seconds of execution would find that ZOMBIE yields 3-4x as many high-value inputs as the traditional BASELINE system. After roughly 75 more seconds of execution, both mechanisms have found an identical training set. Of course, in non-demonstration settings these runtimes would be tens of minutes or even hours.

2.2 Our Approach

The core novel functionality of our IDE is supplied by the ZOMBIE execution system. It must decide which raw data inputs will yield the biggest payoffs to the learned artifact, without actually spending the computation time necessary to convert the raw inputs to feature vectors. In some ways this task resembles that of a multi-armed bandit problem [3], in which an agent must simultaneously gather information about the payouts of the bandit arms while exploiting the best of those arms to maximize overall utility. There are good optimal policies for agents faced with a multi-armed bandit problem, which we use for ZOMBIE.

A standard bandit has two main components: a set of bandit arms and a reward function that determines the utility given by a "pull" of one of those arms. We obtain the set of arms by performing an initial clustering of the raw inputs, which is independent of any feature code. The ZOMBIE planner can decide to "pull" by choosing a random element from a cluster and processing it using the programmer's feature functions. The pull's reward is given by some measure of the resulting feature vector's impact on the quality of the learned artifact. We use the upper confidence bound algorithm `UCB1` [2] to repeatedly determine which arm (that is, cluster of inputs) to examine next.

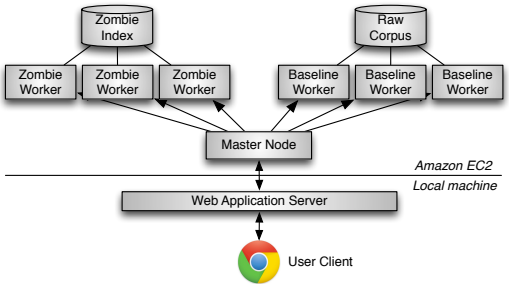

Figure 2: The IDE's basic runtime architecture.

2.3 System Architecture

Figure 2 shows the IDE's basic architecture. A web front-end server presents an interface to the user and spawns any needed jobs on machines at Amazon's EC2 service.

When the engineer is writing a new feature function, all communication is between the web application and the engineer's browser. When the developer decides to evaluate the feature code, then the IDE starts to use the Amazon EC2 service. It launches a series of Worker machines, which apply the user's feature code to the raw data inputs, thereby obtaining usable feature vectors. The BASELINE Workers simply scan through the entire raw input data. The ZOMBIE Workers attempt to choose inputs intelligently through use of the feature-independent index described above. Each Worker runs on a different region of the input set, and reports results back to a Master Node every few seconds. The Master Node collates responses from the Workers and presents ongoing statistics to the user's browser. The BASELINE Workers are present for comparative illustration only and in real-world operation would be omitted, with their computing resources instead used as ZOMBIE Workers.

Although the feature engineer ultimately aims to build a high-quality learned artifact, in this demonstration system we want to illustrate how effectively ZOMBIE chooses useful inputs. Thus, as described below in Section 3, the IDE emphasizes runtime execution statistics like those in Figure 1.

3. USER INTERFACE

The ZOMBIE IDE is a web application with two primary pages. The first page, shown in Figure 3, is a window where the feature engineer can write feature code and configure details of the learning task. A realistic user of our system would spend all of her programming time using this interface. For our demonstration system, the user can either write novel feature code or choose from several pre-written functions. We plan to add more features to the IDE's editor in the future, as described in Anderson, *et al.* [1].

The second page of the interface is shown in Figure 4. With this runtime control panel, the engineer starts the feature function application and evaluation process. The lower third of the page is for job control: the user chooses how many worker machines to use and when to start and stop.

The upper two-thirds of the page is devoted to a live display of ZOMBIE's ability to find high-value inputs. This display

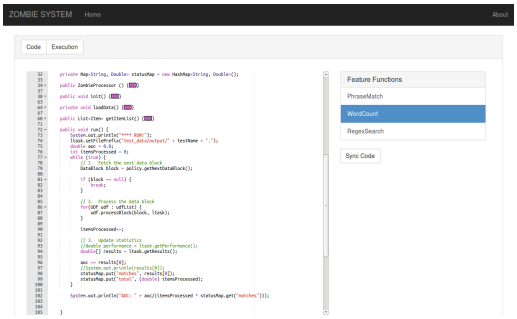

Figure 3: The IDE's code editor. For demonstration purposes, users can either write code from scratch or choose from among several pre-written functions.

is a live version of the basic performance curves shown in Figure 1 and is updated every few seconds. The user can watch this display and decide when to terminate processing early. Our model of the feature engineer suggests she will repeatedly modify feature code using the editor (Figure 3) and then evaluate the code by switching to the runtime control panel (Figure 4).

4. DEMONSTRATION DETAILS

Our demonstration system allows conference attendees to test ZOMBIE on a real text classification task. Users can either choose from several pre-written feature functions or can write their own arbitrary feature code. To allow users to observe meaningful differences between BASELINE and ZOMBIE in just a short demo-style interaction, we will configure the system to process a text corpus that is unrealistically small but functions well for demonstration purposes.

If attendees choose to write their own feature code, we will suggest the following script to illustrate how a feature engineer can be more productive when using our IDE:

1. The feature engineer is interested in building a classifier for news pages on the Web, wanting to mark each page as *politics*, *sports*, *entertainment*, or *other*. The raw dataset is a dump of Wikipedia pages. The engineer wants to iteratively develop a set of features, but of course, each change to the feature code can require a time-consuming evaluation process. For this task, the *other* category is much more common than the others; ZOMBIE should prioritize the non-*other* raw inputs.

2. The feature engineer writes two feature functions. Feature function A is fast and simply tests for the presence of a handful of *politics*-relevant words: president, politician, *etc.* Feature function B is time-consuming, entailing a natural language parse of the input page.

3. The feature engineer applies function A, followed by function B. For each function, the IDE runs ZOMBIE in parallel with BASELINE. ZOMBIE should find relevant feature vectors faster than BASELINE, despite a modest amount of additional system overhead. The conference attendee will see that ZOMBIE offers a very large advantage when evaluating the time-consuming function B, but a lesser one when evaluating A. The attendee can further modify the feature code to see its impact on ZOMBIE's performance delta over BASELINE.

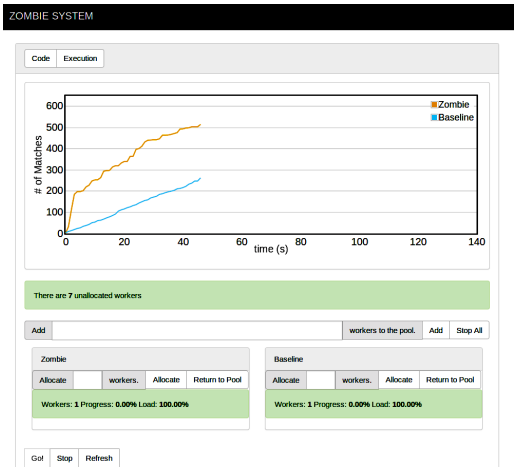

Figure 4: The IDE's runtime control panel.

Finally, we will log all feature functions written by conference attendees. These functions will inform our own research and suggest interesting test scenarios for future users.

5. CONCLUSION

Our demonstration feature engineering IDE showcases the ZOMBIE system. By choosing raw data inputs intelligently, rather than the flat scan that is standard today, it can substantially reduce the amount of time that feature engineers idle away unproductively. We believe our demonstration IDE is an important first step toward a data-centric development environment for feature engineers.

6. ACKNOWLEDGMENTS

This project is supported by National Science Foundation grants IGERT-0903629, IIS-1054913, and IIS-1064606, as well as by gifts from Yahoo! and Google.

7. REFERENCES

[1] M. Anderson, D. Antenucci, V. Bittorf, M. Burgess, M. Cafarella, A. Kumar, F. Niu, Y. Park, C. Ré, and C. Zhang. Brainwash: A data system for feature engineering. In *CIDR*, 2013.

[2] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

[3] S. Bubeck and N. Cesa-Bianchi. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Machine Learning*, 5(1):1–122, 2012.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[5] D. Ferrucci et al. Building Watson: An overview of the DeepQA project. *AI Magazine*, 31(3):59–79, 2010.

[6] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. *SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[7] S. Levy. How Google's Algorithm Rules the Web. *Wired*, February 2010.

[8] B. Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[9] C. Zhang, F. Niu, C. Ré, and J. W. Shavlik. Big data versus the crowd: Looking for relationships in all the right places. In *ACL*, 2012.