Brainwash: A Data System for Feature Engineering

Michael Anderson* Dolan Antenucci* Victor Bittorf† Matthew Burgess*
Michael Cafarella* Arun Kumar† Feng Niu† Yongjoo Park* Christopher Ré†
Ce Zhang†
{mrander,dol,mattburg,michjc,pyongjoo}@umich.edu {bittorf,arun,leonn,chrisre,czhang}@cs.wisc.edu
*University of Michigan, Ann Arbor †University of Wisconsin, Madison

ABSTRACT

A new generation of data processing systems, including web search, Google's Knowledge Graph, IBM's Watson, and several different recommendation systems, combine rich databases with software driven by machine learning. The spectacular successes of these *trained systems* have been among the most notable in all of computing and have generated excitement in health care, finance, energy, and general business. But building them can be challenging, even for computer scientists with PhD-level training. If these systems are to have a truly broad impact, building them must become easier.

We explore one crucial pain point in the construction of trained systems: *feature engineering*. Given the sheer size of modern datasets, feature developers must (1) write code with few effective clues about how their code will interact with the data and (2) repeatedly endure long system waits even though their code typically changes little from run to run. We propose BRAINWASH, a vision for a feature engineering data system that could dramatically ease the *Explore-Extract-Evaluate* interaction loop that characterizes many trained system projects.

1. INTRODUCTION

Trained systems are a new generation of information systems that allow one to answer rich queries over data that is less structured than traditional relational data. The best-known example is the modern search engine, but others include IBM's Watson, Google's Knowledge Graph, and various recommendation systems. Their success has opened a broad range of verticals in health care, finance, energy, and even in traditional enterprises. The potential value of these systems is hard to estimate but likely to be staggering: Web search is a *single vertical*.

Unfortunately, these systems are famously challenging to build, even for highly trained computer scientists. One journalist described Google's search team as engaged in a "relentless slog" [5]. Watson took 20 engineers and researchers 3 years of work before it was competitive with human players [3]. The amount of academic firepower that eventually focused on the Netflix Prize belied the relatively tiny financial reward involved [1]. There is huge demand for such systems, but the current process for building them is daunting for even the richest and most tech-savvy organizations – let alone those organizations that do not routinely attract large numbers of computer science PhDs. If trained systems are to become truly widespread, building them needs to become much easier.

We believe a critical pain point in building these systems is *feature engineering*. Features, sometimes called signals, encode information from raw data that allows machine-learning algorithms to classify an unknown object or estimate an unknown value. For example, consider the observation that the title of a webpage might be more indicative of the page's topic than a typical paragraph-embedded piece of text. The search-engine designer might write a small piece of feature code that returns `true` if the user's search query is present in a page's title and `false` otherwise. A trained system figures out the relative importance of this feature for future tasks during the learning phase.

Features may seem quite ordinary, merely one of the many ingredients that go into a large-scale machine-learning effort (along with raw data, cluster software, human-provided labels, and scalable statistical algorithms). However, a few lessons from the growing body of wisdom surrounding trained systems suggest they are quite remarkable:

1. The More Features, The Merrier. Statistical breakthroughs are academically "sexy," but most trained systems seem to win through lots of features. In 2010, Google's ranking algorithm used 200 distinct features and planned to add 550 more that year [5]. Watson used *"more than 100 different techniques"* for computing an answer, and its designers emphasized the role of employing many loosely coupled experts [3]. These lessons are entirely consistent with some authors' preference for data over model sophistication [4].

2. Feature Man-Months Aren't Mythical. Features can be integrated through light-weight, loosely coupled statistical methods, not the management-heavy techniques required by traditional software components. In principle, large numbers of feature engineers should be able to operate with minimal coordination, apart from the moment when their features' statistical contributions are actually evaluated. Indeed, this organizational principle seems to have been discovered in a roundabout way during the Netflix Prize competition [1]:

6th Biennial Conference on Innovative Data Systems Research (CIDR '13) January 6-9, 2013, Asilomar, California, USA.

Teams that had it basically wrong — but for a few good ideas — made the difference when combined with teams which had it basically right... The top two teams beat the challenge by combining teams and their algorithms into more complex algorithms incorporating everybody's work. The more people joined, the more the resulting team's score would increase.

Unfortunately, writing features can be extremely painful. On the surface, building a feature may seem to be just another software engineering task, albeit one that is run over very large datasets. However, in our experience engineering features requires dramatically more iteration and adjustment. A completed feature is often a small piece of code that is relatively easy to reproduce and does not reflect the large amount of code that was written, tested, and thrown away. We note three ways in which feature engineering is uniquely excruciating:

1. **Grunt-Work Statistics.** First, there is a lot of statistical "grunt work." A feature processes a large dataset — *e.g.*, the set of all webpage titles — whose characteristics are often wholly novel to the developer. Even simple features are hard to write without some commodity metadata, such as frequency distributions of unique values, lists of outlier values, and some simple visualizations. Today the feature developer performs this grunt work for every new feature, by hand.

2. **Unknowable Specs.** Most interesting datasets are large and noisy, making the actual feature code "spec" nearly unknowable without repeated testing against the data itself. For example, it is easy to informally describe a feature such as, *"A social media user's name is somewhat indicative of the user's age."* That is, a user with the name *Brittany* is relatively unlikely to be elderly in 2013. But implementing this feature entails implementing a first version, then learning that user names often have numbers and adjectives appended to a human first name, then writing code to strip off these suffixes, then testing the code again.

3. **Unexpected Failure.** Even a perfectly implemented feature can be an unexpected failure, either because it does not capture any useful information or because its information is already captured by a previously implemented feature. For example, it may be that the name-based method above is helpful when considered alone, but does not add any predictive power beyond text-based methods (say, counting the number of *"lol"*s in a social media user's status updates). If the text-based method had been implemented first, then all the work to implement the name feature is wasted effort.

These three burdens of feature engineering — grunt-work statistics, unknowable specs, and unexpected failure — turn the developer's life into an endless cycle of small iterative code changes and tests. Moreover, most feature code runs over huge datasets that require scalable "big data" systems. The few systems that support the user-defined code necessary for feature engineering (*e.g.*, MapReduce [2]) have generally emphasized throughput over latency. Cluster software thus forces developers to endure high-latency waits (perhaps hours or even days) in the "inner loop" of the feature development cycle. As a result, feature engineering is a time-consuming and draining experience.

We envision a system, BRAINWASH,[1] to dramatically improve the productivity of feature engineers. It provides *pervasive programmer hints* to address the constant iteration associated with feature development. In contrast to the hints provided by today's IDEs (which are derived from code snippets or header files), hints in BRAINWASH are derived from the data under inspection as well as code written by other BRAINWASH developers. It is a multiuser system that has three phases:

1. In **Explore**, BRAINWASH speeds the feature engineer through grunt-work statistics by automatically providing commodity information like frequency distributions and automatically-chosen illustrative samples from the dataset.

2. In **Extract**, BRAINWASH attempts to recommend a feature that is both likely to yield benefits and roughly compatible with the developer's code so far. BRAINWASH accomplishes this by repurposing semi- and unsupervised methods for feature induction as recommendation methods.

3. In **Evaluate**, BRAINWASH enables the engineer to evaluate how well or poorly their new features perform in the context of the entire trained system. The challenge for BRAINWASH is to run and evaluate features as quickly as possible. It does so by speculatively executing code it thinks the user will want to run in the future; while features *can* be arbitrary pieces of code, it is possible to make educated guesses about what the code actually does even without directly understanding the source. For example, two functions that were written by the same user just a few minutes apart are likely to be small modifications of one another.

By increasing the velocity of the developer in each phase of the *Explore-Extract-Evaluate* loop, BRAINWASH aims to improve developer effectiveness during feature engineering.

In the next section we review two case studies from our own work, illustrating problems with feature engineering today and showing how it could be improved. In Section 3, we propose BRAINWASH and describe its design.

2. CASE STUDIES

We now describe two feature engineering case studies from trained systems being built by the authors of this paper. GEODEEPDIVE integrates scientific data for geoscientists. AUTOMAN aims to reproduce national economic statistics using social media activity.

2.1 GeoDeepDive

Today, an individual geologist has a *micro view* of geoscience: she has access to measurements from at most a handful of the approximately 30,000 geographical units in North America, usually data collected in her own and partner labs. Other labs' data is buried in the text, figures, and

[1]BRAINWASH is a reference to the artist Mr. Brainwash, or MBW, who mass-produces art in the mold of famous street artist Banksy. Similarily, we propose to mass-produce the artistry required by today's trained systems.

graphs of webpages and journal articles. Even simple *macro-view* questions, such as, *"How much carbon is in the North American rock record?"* are unanswerable without extracting and aggregating these disparate measurements.

GeoDeepDive aggregates data from multiple sources to build a single macro-view database.[2] A geologist can browse geological regions on a map, filter by various criteria, and see relevant extracted data along with textual provenance.

The machine learning problem is largely one of *entity linking*. Geological data is often tied to a simple tuple consisting of a spatial component, a temporal component, and a hierarchy of "formation" types. Geologists use precise language to refer to formation names, but may refer to locations only obliquely, *e.g.*, *"near the panhandle."* The feature engineering task for GeoDeepDive is to find hints that accurately map these text-embedded "geo-tuples" to elements in a database. The feature developers encountered the usual ration of troubles:

1. **Grunt-Work Statistics.** The input dataset held roughly 20,000 geology-oriented papers of unknown content and style. The quantity and distribution of figures, tables, footnotes, formations, and unambiguous place-names naturally have a large impact on which features are even possible. Developers had no way to obtain that information except to write a series of fairly dull statistics-gathering programs.

2. **Unknowable Specs.** At the start of the project it was not obvious to the computer scientists, and perhaps too obvious to the geologists, that "black shale" refers not just to carbon but to carbon along with a very specific range of possible temporal values. There was no way for developers to learn that "black shale" needed a specialized processor without actually slogging through the data.

3. **Unexpected Failure.** To link phrases like "this formation" to actual geological formations, developers used a baseline pronoun-coreference feature that maps phrases like "this formation" to the nearest formation-name mention in the previous paragraph. One might expect more sophisticated features to lead to significant quality improvement. Yet developers found that more sophisticated features that consult either document-level statistics or deep linguistic parsing have essentially the same accuracy as the baseline feature. Moreover, when both types of features are used, there was no notable improvement in quality.

2.2 Automan

Automan's goal is to use social media data to collect economic statistics.[3] For example, an increase in the number of users who write "I need a job" may indicate a growth in unemployment. Statistics derived from social media hold the promise of being much faster and less expensive to gather than traditional survey-driven data.

An important part of economic data is the demographics of the individuals involved. For example, a 20-year-old college student who needs a summer job occupies a very different economic position from a 55-year-old factory worker who has been laid off. While traditional surveys collect reliable demographic data, most social media systems do not.

Thus, Automan estimates the age of each social media user, even though most users do not reveal any explicit age information at all. In this work, we have focused on Twitter data, which comprises timestamped messages and the users' social network. Building this trained system entailed the same three problems.

1. **Grunt-Work Statistics.** Just knowing the basic structure of the input data is enough to suggest a number of statistics that would be useful for feature engineering: the number of messages, the number of users, the rate of message production, how many friends different users have, how many messages are roughly useful for economics statistics, *etc.* Computing each one of these numbers required dedicated code.

2. **Unknowable Specs.** The student working on this project actually encountered the name-formatting issue described in Section 1. Another example concerns messages where the author reveals his own age, *e.g.*, "I am 35 years old." It initially seemed users have no motivation to be misleading. But people in some cases will ironically exaggerate their age, as in, "I threw out my back today. I'm 100 years old." Removing these false statements improved the age predictor. It is hard to imagine that the feature engineer could know ahead of time that an irony-detector could be a useful tool.

3. **Unexpected Failure.** For all the work that went into the *Brittany* feature that exploits name popularity information, it did not turn out to be very useful. Experiments so far have shown it adds almost no predictive power beyond a system that examines word choice and the friend network. The name-popularity feature seemed initially compelling but was a poor investment of engineering resources.

We now describe how brainwash addresses these challenges.

3. BRAINWASH: PRELIMINARY DESIGN

We can now propose a preliminary design based on an *Explore-Extract-Evaluate* interaction loop with the developer. It aims to improve feature engineering via *pervasive programmer hints* and *low-latency program execution.*

We observe that our previous projects used data sets from a wide variety of sources: structured data, Tweets, images, and text corpora stored in flat files. To process such diverse data, we view it as crucial that brainwash not be tied to any particular data processing system, but instead sit as a layer between the developer and the data storage and processing substrates. Our current plan is to create adapters for a variety of storage and processing systems: main memory, relational databases, key-value stores, flat files, distributed filesystems, and MapReduce.

One consequence of our choice to support a wide variety of data sources is that brainwash's data model must be relatively flexible. The model comprises sparse tuples that have *list* as a first-class type — they are roughly similar to Protocol Buffers [6]. brainwash models features as user-defined functions that consume and produce lists of tuples. The developer explicitly declares input and output schemas for each feature.

[2]For an overview, including a demonstration video, see `http://hazy.cs.wisc.edu/geodeepdive`.

[3]See a demo video at `http://youtu.be/iq_IW34QeJQ`.

We treat feature development as a workflow of developer-written functions $udf_0, udf_1, ..., udf_N$. A "run" i consists of applying udf_i to each tuple in the input dataset (*e.g.*, each webpage, each academic paper, or each Tweet). Like the `map()` from MapReduce, a *udf* function invocation takes a single tuple from the input and yields zero or more tuples that are placed into the run's output.[4] Explicit schemas allow BRAINWASH to guess what the *udf* is doing, even if the function consists entirely of opaque compiled code.

We envision BRAINWASH as a centralized system that supports many developers and projects simultaneously. Even if developers do not explicitly collaborate, the system stores and sometimes repurposes developers' work. BRAINWASH can thus exploit knowledge about the *udf* workflow, the dataset, and developer activity to address each of the three problems encountered in feature engineering.

1. **Explore: Grunt-Work Statistics.** BRAINWASH can assist with statistics collection by leveraging the formal schema and previous users' *udf*s. Exploiting schema information for the raw input file should be straightforward: the system can automatically read the input, counting tuples and fields as it goes. The result should be a rough overview similar to what is offered in data integration tools such as Google Refine. However, this schema information is not sufficient when the developer wants to see statistics on "derived objects" that come from running *udf*s on a subfield of the raw input, such as the named entities embedded in a webpage.

We can automate statistics on derived objects by examining the output schemas and datasets from previous *udf*s that share an input schema with the current one under development. For example, consider a developer who has written one *udf* that transforms a raw input file of webpage tuples into a series of paragraph instances and then writes a series of different *udf*s that transform paragraph into a large range of outputs with different schema formats. The body of *udf*s not only provides evidence that paragraph is an interesting and broadly useful object, it also provides BRAINWASH with the code necessary to produce paragraph. Further, we do not need to limit these automated statistics to the user who actually wrote paragraph-producing code: it can be used to help any programmer who needs to process webpages.

2. **Extract: Unknowable Specs.** BRAINWASH should preemptively suggest code that will address unknowable specification problems before the developer even encounters them. For example, any developer who needs to process first names in social media is likely to want to remove all the strange username suffixes described in Section 1.

We can do this by pairing schema matching with code suggestion. Imagine a user A who has written udf_i^A which extracts usernames from Tweets, as well as udf_{i+1}^A that normalizes the usernames. Now user B writes a function udf_j^B that has an input schema similar to udf_i^A and generates output that is similar to that produced by udf_i^B. BRAINWASH should then suggest that B run udf_{i+1}^A and can present sample input/output pairs to explain why.

There may be a large number of candidate functions that B could apply — ranking these will be a core technical challenge. Like other ranking systems, *udf* suggestion should improve as developers use the system, providing additional evidence about which *udf*s are appropriate in which cases.

[4]BRAINWASH also allows aggregation, similar to `reduce()`, but we omit our description of this mechanism to simplify our exposition.

3. **Evaluate: Unexpected Failure.** Improving the overall performance of the trained system is, in the end, the feature developer's real goal. But measuring a feature's real contribution to the end-to-end system can entail training and testing many different permutations of features, each step of which may need to process a huge dataset. As a result, developers sidestep system-wide evaluation for far too long in the development process. BRAINWASH encourages frequent and full evaluation by saving data on previous feature permutations, and speculatively training statistical systems using features under development.

The system automatically formulates code according to the above three features, then sends the results to the user's IDE. In many of the above cases, the system both formulates code and speculatively executes it on the user's behalf.

This speculative formulate-and-execute cycle is what allows BRAINWASH to not just give programmer hints, but reduce execution latency as well. In some cases, such as the **explore** component above, BRAINWASH reduces user-apparent latency simply by removing the work from the user's todo list — we do the work when there's slack time in the back-end processing system. In other cases, as with **extract**, the user's coding target is unclear; the system tries to guess it, executes the resulting code, and hopes that the effort was not wasted. Even if the user's goal is not predicted, the system can use previous iterations of the current *udf* to determine which input tuples are likely to yield useful outputs and prioritize these during processing.

4. CURRENT STATUS AND NEXT STEPS

We are building an initial prototype of BRAINWASH based on the code of GEODEEPDIVE, AUTOMAN, and an earlier web-scale prototype called DEEPDIVE (`http://hazy.cs.wisc.edu/deepdive`). Our plan is to refine BRAINWASH and these active research projects in parallel.

BRAINWASH raises several research questions. One that we have examined is the efficacy of different feature-engineering methods. Earlier this year, we performed the first study that compares two popular methods used in the explore-extract-evaluate loop, *distant supervision* and *crowdsourcing*, to extract relationships at web scale [7]. This study sparked our interest in using distant supervision to recommend features in the explore and extract phases. We plan to use BRAINWASH as a vehicle to continue such studies.

5. REFERENCES

[1] E. V. Buskirk. How the Netflix Prize Was Won. *Wired*, 2009.
[2] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.
[3] D. Ferrucci. An Overview of the DeepQA Project. *AI Magazine*, 2012.
[4] A. Y. Halevy, P. Norvig, and F. Pereira. The Unreasonable Effectiveness of Data. *IEEE Intelligent Systems*, 2009.
[5] S. Levy. How Google's Algorithm Rules the Web. *Wired*, 2010.
[6] `https://developers.google.com/protocol-buffers/`.
[7] C. Zhang, F. Niu, C. Ré, and J. Shavlik. Big Data versus the Crowd: Looking for Relationships in All the Right Places. In *ACL*, 2012.