Runtime Support for Human-in-the-Loop Feature Engineering Systems

Michael R. Anderson
University of Michigan
mrander@umich.edu

Dolan Antenucci
University of Michigan
dol@umich.edu

Michael Cafarella
University of Michigan
michjc@umich.edu

Abstract

A machine learning system is only as good as its features, the representative properties of a phenomenon that are used as input to a machine learning algorithm. Developing features, or feature engineering, can be a lengthy, human-in-the-loop process, requiring many time-consuming cycles of writing feature code, extracting features from raw data, and model training. However, many opportunities exist to improve this workflow, such as assisting feature code development and speeding up feature extraction.

In this paper, we discuss two projects that take different approaches to accelerate feature engineering, allowing the engineer to spend more time doing what humans do best: applying domain insight to engineer high-impact features. ZOMBIE *is a general-purpose system that reduces the amount of time needed to extract features.* RACCOONDB *is a system that helps users quickly extract features for nowcasting, or using social media to estimate real-world phenomenon. We also discuss several opportunities for future research that would improve the experience and output of feature engineers.*

1 Introduction

Many of today's most interesting software systems depend on machine learning algorithms trained on large datasets; systems like search engines, recommendation systems, and image recognition systems have become drivers of both technological and economic growth. While the specific details of machine learning systems may differ, there is often one common thread: a real-world phenomenon is represented by a set of *features*. Features aim to capture properties relevant to the chosen learning task. For example, the number of times the phrase "lost my job" appears on Twitter in a given week may be a good feature for predicting unemployment levels; a count of the political figures mentioned on a Web page may be a helpful feature to classify it into a category like politics instead of sports; and a patient's weight and blood pressure may be indicators of a risk for heart disease.

The quality of the features is paramount: a machine learning system will ultimately be only as good as its features [2, 20]. That is, if the features fail to sufficiently capture the variation among specific examples of the target phenomenon, the accuracy of the system will suffer. Historically, features have been written by humans, taking advantage of experts to impart domain knowledge to trained models. More recently, deep learning methods have been successful at automatically discovering good features, especially in problem domains that are challenging for humans, such as image and signal processing [13, 18, 35].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

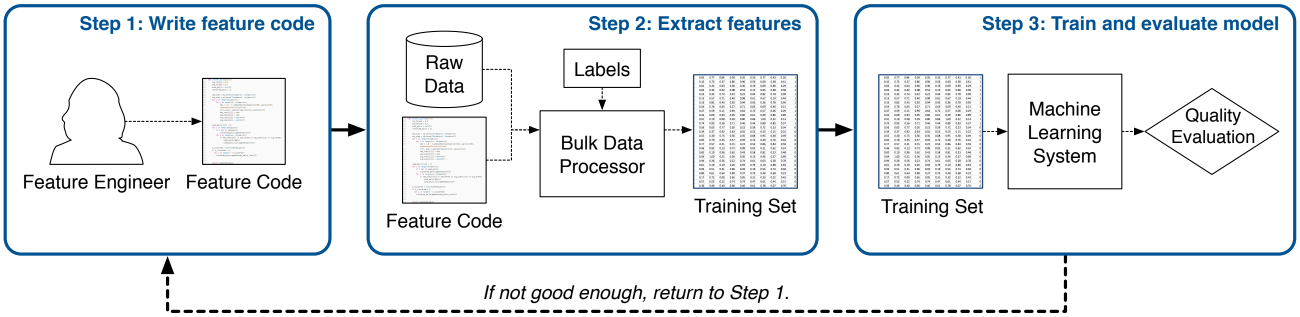


Figure 1: Feature engineering loop.

While both approaches are important to the field of data science [25], in this paper we will focus on systems designed to support the lengthy and often tedious cycle of **human-in-the-loop feature engineering**, which has been a growing focus of research in the database and systems communities [2–5, 8, 30, 34, 40]. Feature engineering is an iterative, data-centric process that we have described in detail in previous work [2, 3], and it has a general workflow followed by many different projects (e.g., [5, 21, 22, 49]).

1.1 Human-in-the-Loop Feature Engineering

In this paper, we will discuss two feature engineering projects that we have personally worked on. ZOMBIE (Section 2), a general-purpose feature engineering system, speeds up feature extraction from raw data for general supervised learning tasks, such as document classification or linear regression-based predictors. RACCOONDB (Section 3) targets a specific problem domain, speeding up the creation of features for *nowcasting models*, which estimate current values of real-world phenomena based on social media messages. Though each targets a different type of machine learning task, both involve the high-level human-in-the-loop process illustrated in Figure 1. This loop takes place while the machine learning system is being developed, before the system's final deployment. It can be described as follows:

1. **Write feature code:** The feature engineer writes code to extract features that represent salient properties of the items in a corpus of raw data. For example, when classifying Web pages, a feature may simply count the number of times a category name appears in the document or perform more complicated functions like named entity recognition using natural language processing techniques. Or, when predicting unemployment levels, a feature might be derived from topics mentioned on social media. The feature engineer may create a set of feature extractors, creating many feature values per raw data item.

2. **Extract features:** The feature code is applied to the raw data to extract the features for each raw data item, often using a bulk data processing system like MapReduce [19] or Spark [47]. When the corpus is large (e.g., a Web crawl or social media database) or when the feature code is computationally complex (e.g., involving natural language processing for document classification or comparison of time-varying signals of topic frequencies in social media), this step can take hours or longer to complete. In this step, each item is also labeled; the label may come from an existing database, be added by a human expert, or created by algorithmic means. The output of this step is a set of labeled feature vectors used to train the machine learning system.

3. **Train and evaluate the model:** Using the labeled set of examples constructed in Step 2, a machine learning system is trained with an appropriate algorithm; a naïve Bayes classifier could be chosen for our document classification example, while our unemployment predictor may use linear regression. Once trained, the model's quality—and thus the effectiveness of the features—is evaluated using an appropriate metric, usually measured over a set of test data. For our document classification example, we may measure

the percentage of test examples that were correctly classified, while for our unemployment predictor, we may use the root mean square error in the test predictions. If the quality is less than what the application requires, the feature engineer begins the process again from Step 1.

A feature engineer can go around this loop many times when developing features for a high-quality machine learning system. The impact of a particular feature to a model is very difficult to predict without actually training the model. First, because datasets are typically large and noisy, it is difficult for the engineer to know how to accurately specify a feature without repeatedly testing it. Further, within feature sets, it is difficult to know whether a new feature will bring any new information to the model [2]. Compounding the tedium, each iteration of the loop can itself take a long time to complete because each step can be lengthy: the engineer must write complicated code, apply it to a large raw dataset, and train a machine learning model. Thus, when designing a system that assists the user in human-in-the-loop feature engineering, we can speed up the engineer's workflow by either reducing the number of loop iterations or by speeding up each iteration.

1.2 Goals of a Feature Engineering System

A successful feature engineering system can take several approaches to speed up the feature engineering loop described in Section 1.1. These include:

Assistance writing feature code — Helping the feature engineer design features and write feature code can lead to fewer, shorter iterations around the loop. For example, RACCOONDB uses query expansion principles to automatically improve the applicability of feature code. Feature-focused data visualization tools can help engineers better understand data distribution and feature coverage. Development tools for data cleaning and extraction [28,49], as well as domain specific languages targeted at developing features from massive datasets [39], can reduce the time needed to write feature code. Deep learning tools can also be used to automatically create feature extractors, though often at a cost of human interpretability of the resulting features [13, 18, 35].

Faster feature extraction — Extracting features from raw data more quickly can reduce the amount of downtime a feature engineer faces. In many machine learning applications, the amount of data available is far greater than is needed to train a model well enough to evaluate feature effectiveness. For example, when using a Web crawl corpus, pages from e-commerce sites will likely provide little training information when the task is to classify documents as sports, politics, or technology. Pruning or otherwise avoiding the processing of irrelevant raw data items, as is done by both ZOMBIE and RACCOONDB, can greatly reduce the time needed to complete an iteration of the feature engineering loop.

Quicker model training — Because some types of machine learning models can take a very long time to train, speeding this up can greatly reduce the time needed to evaluate the quality of feature code. Approaches here may include scalable, parallelized algorithms [30, 36] or training a fast, approximate model that is good enough to evaluate the feature code.

Further, researchers have begun investigating several additional areas that improve the feature engineer's experience: finding and and collecting appropriate raw data from diverse sources [14]; labeling raw data through crowdsourcing or algorithmic means [21]; and automatically configuring hyperparameters in complex machine learning models [43]. Systems that continue work in these areas would be a benefit to the feature engineer.

1.3 Continuing Relevance of Database Research

Feature engineering may seem to be primarily a machine learning concern, but in fact, database research—especially systems-focused research—is highly relevant. The database community has long focused on optimizing data-centric workflows, and traditional approaches can be starting points for improving feature engineering.

When performing *query optimization*, operation execution is often dynamically ordered to reduce the number of tuples examined in a given query. In feature engineering, similar techniques may be employed to reduce the number of raw data items processed or to minimize the size of the training set used to train a model. Likewise, traditional *indexing* techniques allow for quick access to specific parts of large datasets. In feature engineering systems, methods like these can be used to quickly access parts of a large raw data corpus that have high utility to a particular learning task, allowing for faster feature extraction or training of the machine learning model.

Evaluating the effectiveness of feature code can be recast as a query over the raw data that produces a quality measure Q as a result. A high-precision value of Q may not be needed if a fast, lower-precision value can be computed, suggesting that techniques similar to those used in *approximate query answering* systems (e.g., BlinkDB [1]) may be useful. Extracting features from large datasets often involves *parallel execution* using systems like MapReduce [19] and Spark [47]. Extending these systems to include optimization techniques from parallel databases, as well as making them more supportive of machine learning workloads, as is being done with MLlib [36] and MLbase [30], will certainly accelerate feature engineering workflows.

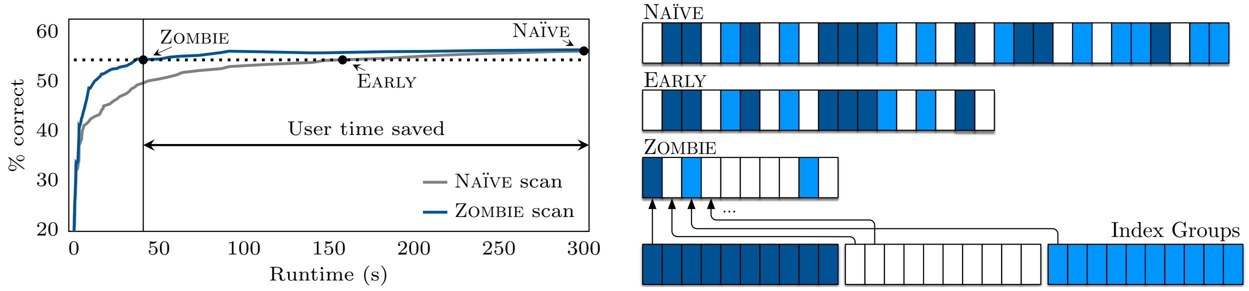
Feature extraction in many cases is an extension of *data cleaning and transformation*: raw data is often in a semi-structured format—e.g., log files, sensor output, and, to an extent, Web pages—and extracting features involves cleaning and transforming that data into a usable format. Data transformation and extraction tools such as Wrangler [28] and DeepDive [49] are currently very helpful in extracting useful data from these sources, but more specialized tools integrated into a feature engineering workflow and geared towards machine learning tasks would be a boon to feature engineers. For cases where data extraction is not straightforward, tools that assist in querying raw data or writing extraction programs using *query synthesis* (e.g, [16]) or *query expansion* (e.g, [29]) principles can speed up the feature engineer's work. Finally, *data visualization* (e.g., [45]) tools can help view raw data in terms of machine learning tasks to help feature engineers design more effective features.

In the remainder of this paper, we will discuss two projects that take different approaches to accelerate the feature engineering loop, allowing the feature engineer to spend more time doing what humans do best: applying domain insight to engineer high-impact features. In Section 2, we go over the optimizations used by ZOMBIE [3] to reduce the amount of time needed to extract features. In Section 3, we show how RACCOONDB [5] allows a user to quickly derive nowcasting features from social media. We also present an informal case study in Section 4 that further demonstrates how RACCOONDB helps users quickly estimate real-world phenomena. Finally, in Section 5, we discuss several opportunities for future research that would improving feature engineering.

2 ZOMBIE: Accelerating Runtime through Input Selection

ZOMBIE [3] is a general-purpose feature engineering system designed to reduce the amount of time needed to evaluate the effectiveness of feature code. In particular, ZOMBIE was designed with our ***faster feature extraction*** goal from Section 1.2 in mind: it significantly shortens the time needed to complete an iteration of the loop shown in Figure 1 by reducing the time needed to apply the code to the raw data (Step 2).

Our Approach — ZOMBIE builds on ideas from query optimization, approximate query processing, and indexing to reduce overall feature code execution time through *input selection*: it dynamically learns which raw inputs are most useful to the machine learning system being trained and selects those items to be processed first. It then stops early once the machine learning system is trained adequately enough to accurately judge its quality. The resulting model is trained with only a small subset of high-impact raw data items; ZOMBIE learns which items in the corpus are likely to be redundant or irrelevant to the trained model, eliminating the need to process them. In this section, we will summarize ZOMBIE's methods and experimental results. Complete system details and results can be found in our full paper [3].



(a) Learning curves for different input selection methods. The light gray line shows a standard bulk scan of the data; ZOMBIE is shown in blue. Dots indicate stopping points.

(b) Example subsets created by different input selection methods. Each block represents a raw data item; lighter colors show more useful items.

Figure 2: Effects of input selection. ZOMBIE selects more useful raw inputs, yielding a faster learning process.

2.1 Input Selection

ZOMBIE is targeted at feature engineers building systems for supervised learning (e.g., for classification or regression). While developing features, the engineer may be willing to accept a slightly less-precise evaluation of model quality (Step 3 of Figure 1), if that quality value can be determined quickly. ZOMBIE is designed to exploit this tradeoff and generate a fast, approximate measure of the trained model's quality. To see how, consider Figure 2a, which shows the learning curves of a classification system if it were continually re-trained as new training examples are added to the training set (that is, as raw data items are processed by the feature code and labeled to produce training examples). The gray line for the NAÏVE scan shows the learning curve if the corpus is randomized, as it would be for a typical bulk scan of the data using a data processor like MapReduce or Spark. The system's accuracy increases quickly with the initial training examples, but each new example added to the training set has diminishing returns. The system could clearly stop early and get a close estimate of the model's final accuracy, represented by the point at EARLY.

ZOMBIE, however, takes this further by purposely selecting the next raw input to process, as opposed to the random draw of NAÏVE. The inputs chosen are those predicted to have a high utility towards training the model. The utility of an individual raw data item varies between learning tasks, as well as between feature code changes within the same task, making a high-quality static index identifying useful items impossible to create. Doing so would require extracting the current features from the entire corpus for analysis, which is exactly the time-consuming process we wish to avoid. Instead, in a one-time, offline pre-processing step, ZOMBIE organizes the raw data into groups of similar items called *index groups*, which are then used at runtime to identify groups of potentially useful items. ZOMBIE learns which index groups are most likely to contain useful items and selects items from those for processing. Through our experiments, we found that standard clustering methods, like the k-means algorithm, created general purpose index groups that were useful across a range of learning tasks.

The input selection process results in a trained model using just a small subset of the available raw corpus. This is illustrated in Figure 2b, which shows three different input selection methods. NAÏVE is a simple bulk scan over the entire raw data set, whose items are depicted by the multi-colored blocks, where more lightly colored blocks represent raw data items that are more useful to the learning task. The EARLY method uses the same early stopping method as ZOMBIE, but it is run using the same random layout as the full NAÏVE bulk scan. A representative subset of high- and low-quality items are processed. ZOMBIE, on the other hand, draws items from the index groups shown at the bottom of the figure, which have been clustered into groups with different utility values.[1] Once ZOMBIE learns which index groups contain the most useful items, raw data items are drawn

[1] In practice, the index groups are not so cleanly separated, but the method is still successful if there is even a small difference in average utility between the groups.

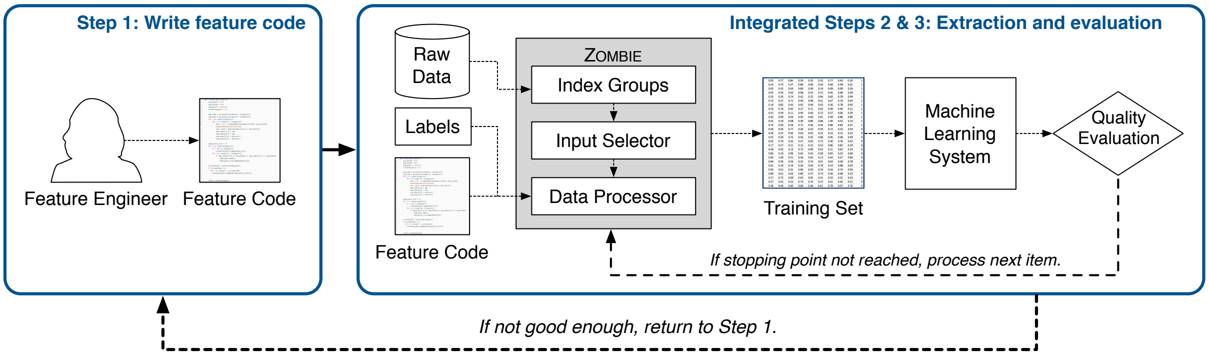

Step 1: Write feature code
Feature Engineer
Feature Code
Integrated Steps 2 & 3: Extraction and evaluation
Raw Data
Labels
Feature Code
ZOMBIE
Index Groups
Input Selector
Data Processor
Training Set
Machine Learning System
Quality Evaluation
If stopping point not reached, process next item.
If not good enough, return to Step 1.

Figure 3: ZOMBIE's basic architecture, which combines Steps 2 and 3 of the feature engineering loop in Figure 1. The system's novel components are the physical index of index groups and the bandit-based input selector. These parts work with standard machine learning components to reduce the time needed to generate features from raw data.

from them, allowing the machine learning model to be trained using much less data.

Because the utility values of the index groups are unknown at the start of each iteration of the feature engineering loop, ZOMBIE faces a classic exploration versus exploitation tradeoff when trying to find the highest-utility index groups in order to maximize the processing of useful items. Thus at runtime, ZOMBIE uses a multi-armed bandit approach to determine which index groups contain the highest-utility raw data items. ZOMBIE draws items from these groups to process with the feature code to generate a new training example, retraining the machine learning model with each addition to the training set. Once the trained model has reached an appropriate stopping point (determined by observing a plateauing of the learning curve), the system stops.

2.2 System Design

Figure 3 shows the basic architecture of ZOMBIE, which combines Steps 2 and 3 of the feature engineering loop in Figure 1. When the system is initialized, the raw data is organized into a task-independent set of index groups, created using a general clustering method appropriate for the data. We create an inverted index $\mathcal{I}$, where the keys are unique identifiers for each group and each key's posting list is an unordered set of raw data items. ZOMBIE's input selector uses a standard upper confidence bound (UCB) multi-armed bandit strategy [11] to explore the index groups in $\mathcal{I}$ and exploit the ones determined to have the highest utility to the learning task. The feature functions are applied to each item selected for processing to create a feature vector for the training set, which is provided to the machine learning algorithm to generate a model. A quality function Q evaluates the model and maintains statistics on the model's improvement with each new item. The output of Q is used as the reward to update the input selector's multi-armed bandit, and the process is repeated. This output is also monitored to detect a plateauing of the learning curve, at which point the processing is stopped and the final quality measure is presented to the feature engineer.

2.3 Summary of Experimental Results

ZOMBIE was tested on several different machine learning tasks, using feature functions of varying complexity. Compared to a baseline method of a random bulk scan of the raw data that used the same plateau-based early stopping method, ZOMBIE showed an average of nearly 2x speedup for regression tasks and up to an 8x speedup for classification tasks (Table 3), meaning that each iteration of the feature evaluation loop was significantly shorter when using ZOMBIE. Compared to a full, single-processor bulk scan, as might be done in practice, ZOMBIE yielded up to a 90x speedup. For each of the tasks in Table 3, we used the same raw data corpus (1 million Wikipedia pages),

Task	Speedup
2-class classify	5.5x
6-class classify	7.8x
Regression	1.7x

Table 3: ZOMBIE's speedup results vs. a bulk scan with early stopping on several machine learning tasks.

index groups (k-means clusters), and feature set (counts of 40 specific words in each document). The range in speedup values highlights how the applicability of a set of index groups can vary between learning tasks. Our index groups represented the variability in the token-based features well, so when the features were highly relevant to the learning task, ZOMBIE could quickly find the high-utility raw data items, as it did in the document classification tasks. The feature set was less relevant for the regression task, so there were few actual high-utility items, making it difficult for ZOMBIE to provide more than a modest speedup. Still, the processing time was cut nearly in half compared to the early stopping baseline. For detailed experimental results, including evaluation of index group creation methods and design choices for the multi-armed bandit, see our full paper [3].

2.4 Related Work

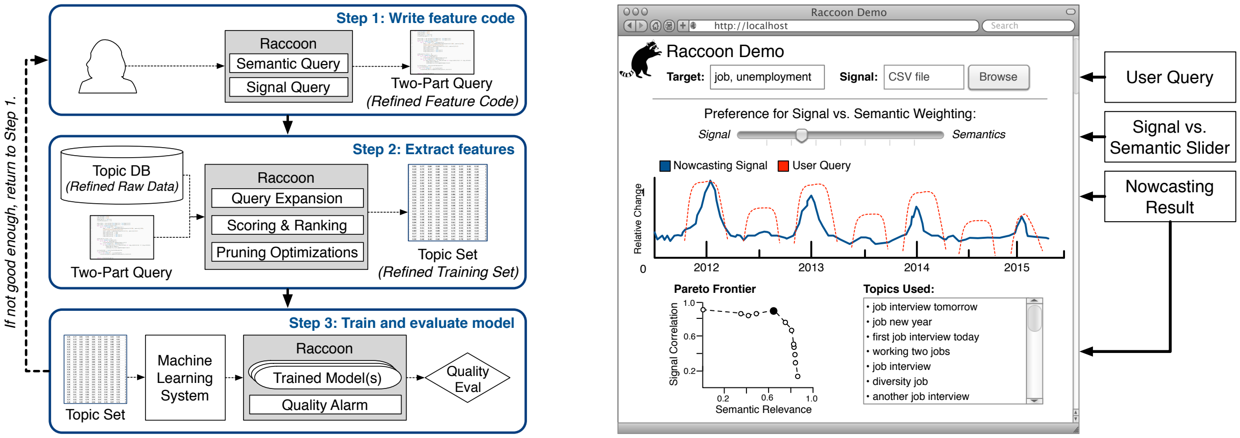
ZOMBIE [3] draws on work from a diverse set of fields. Feature engineering and feature selection have been the focus of some recent research in the database community [2, 30, 48]. ZOMBIE's goal of approximating the feature quality metric is similar to approximate query answering systems [1,12,23]. The practice of selecting the inputs for a training set based on their predicted utility is related to active learning [42], though in active learning the goal is typically to minimize the amount of labels needed to generate, based on a pre-existing feature set. We have also presented a demonstration development environment that uses ZOMBIE's input selection method [4].

3 RACCOONDB: Finding Features in Social Media

RACCOONDB [5] is a feature engineering system we designed for a specific problem domain: generating *nowcasting* features for economic and social science models. Nowcasting is the process of extracting trends from social media to generate a time-varying signal that accurately describes and quantifies a real-world phenomenon. For example, the weekly trend of US unemployment claims can be modeled using the frequency of tweets about unemployment-related topics [7]. Nowcasting has been used to model flu activity [24], mortgage refinancing [10], and more [17, 41, 46]. These models, however, have been typically created as one-off and highly manual endeavors, with researchers tediously searching through a social media corpus to find topics that are appropriate proxies for their target phenomena. By focusing on this specific problem, RACCOONDB is able to be more aggressive than a general feature engineering system could be with improving the nowcasting feature engineering loop.

The process used to created these models follows the feature engineering loop in Figure 1: the researcher writes code or a query—based on either textual or time series matching—to specify her desired topics (Step 1). Next, she executes the code or query over a large social media corpus to generate a training set based on these topics (Step 2). Finally, she uses the training set to build a model of the target phenomenon, and then evaluates its predictiveness using a holdout dataset (Step 3). This process is repeated many times by the researcher, in part because the social media corpus can be very diverse, making it difficult for the researcher to know precisely which topics to select. And because the corpus can be very large (over 150 million distinct topics in our experiments), each iteration can be lengthy, taking dozens of minutes to many hours, depending on the system used.

Our Approach — RACCOONDB targets two of the goals of feature engineering systems we discussed in Section 1.2: ***assistance writing feature code*** and ***faster feature extraction***. First, we used query expansion techniques to assist users in writing queries (i.e., "feature code") that find relevant topics, even when their initial queries inadequately describe their target phenomena. Second, we used a number of query optimization techniques to assist with faster feature extraction, allowing queries to run in interactive time and users to quickly modify their queries to improve their results. In the remainder of this section, we will briefly describe the design of RACCOONDB and present several experimental results. For an in-depth discussion of nowcasting and RACCOONDB's design and technical evaluation, see our full paper [5].



Raccoon Demo

Target: job, unemployment **Signal:** CSV file

Preference for Signal vs. Semantic Weighting:

Signal — Semantics

Topics Used:

- job interview tomorrow
- job new year
- first job interview today
- working two jobs
- job interview
- diversity job
- another job interview

(a) Feature engineering with RACCOONDB

(b) RACCOONDB user interface

Figure 4: In Step 1 of (a), users provide *refined* feature code in the form of a two-part query using the interface in (b). In Step 2, RACCOONDB generates a set of topics that serve as a *refined* training set. In Step 3, RACCOONDB generates several results, which the user explores in real-time with the slider interface in (b).

3.1 System Design

RACCOONDB is a system that allows a user to query a database of *topics* extracted from social media messages to produce a nowcasting result. Each topic represents a collection of similar messages (e.g., those discussing the loss of a job) and contains both a textual label (e.g., “job loss”) and a time-varying signal with the frequency these messages appear (e.g., daily). It is this set of tuples that RACCOONDB uses to answer nowcasting queries.

Assistance Writing Feature Code — Figure 4a shows how RACCOONDB fits in the feature engineering loop. First, the user writes a two-part query (Step 1) that declaratively describes her target phenomenon both semantically—as a text string—and as a time-varying signal (Figure 4b). This can be considered a refined version of the feature code in Figure 1. Ideally, this signal would represent historical ground truth data for the target phenomenon, but when this is not available, the user can use her domain knowledge to describe this in a distant supervision manner [37]. For example, if targeting box office sales, she may indicate peaks during the summer when blockbuster movies are generally released. Because users may not have perfect knowledge about the target, RACCOONDB employs query expansion principles during feature extraction (Step 2) to expand the semantic part of the query and allows partial time series to be input as the signal part. This makes RACCOONDB more robust to error in the nowcasting query, as our quality experiments in Section 3.2 show.

Further, during evaluation (Step 3), RACCOONDB assists users with evaluating their models in two ways. Since users may have more confidence in either of their semantic or signal query components, multiple models are trained, each weighting the query components differently. During evaluation, users can explore these models’ results in real time with a signal vs. semantic weighting slider (Figure 4b). Additionally, for users who lack ground truth data and provide a query signal that only roughly estimates their target phenomena, RACCOONDB features a *query quality alarm*. This alarm uses a model trained on a set of labeled low- and high-quality queries, alerting users if the model classifies their query as low quality.

Faster Feature Extraction — Processing this two-part, computationally intensive query over a huge corpus of social media can take a significant amount of time if done naïvely. For instance, our experiments in Section 3.2 show that popular data systems like Apache Spark and PostgreSQL took dozens of minutes to many hours to query 150 million topics. Our goal in designing RACCOONDB was to return results in interactive time to allow users to quickly hone their nowcasting results. We accomplished this goal (with queries returned often in about one second) through a number of optimization techniques (Step 2).

First, RACCOONDB uses *semantic-based candidate pruning* to avoid the expensive scoring of many topics. This works by first semantically expanding the user's query with a thesaurus-based method. Each word in each topic (as well as the query) is expanded into a set of synonyms, and a similarity score is computed based on the overlap of the expanded sets of words. This score is used to rank candidate topics for potential inclusion in the query answer. Computing this score for every topic is very expensive, so RACCOONDB prunes the candidates by analyzing the graph of synonym sets in the thesaurus, as well as by dynamically computing a threshold score below which candidates are excluded. In practice, this can reduce query computation time by over 90%.

Second, RACCOONDB uses *signal-based candidate pruning* to further avoid fully scoring many topics. To do this, RACCOONDB employs confidence interval pruning to limit the amount of expensive correlation computations it performs. Correlations are first calculated using a very low-resolution version of the topic and query signals (20 data points, rather than nearly 200). Using the standard 95% confidence interval for Pearson correlations, we eliminate all topics whose upper confidence bound was less than the lower confidence bound of the k-th highest ranked topic based on the low-resolution correlation. The full, expensive correlation was calculated for the remaining topics. When combined with our semantic-based pruning, this further reduces runtimes to interactive speeds (averaging 3.6 seconds with 1 core and 1.2 seconds with 30 cores in our experiments).

Finally, RACCOONDB takes advantage of several *low-level optimizations*. Semantic similarity and signal correlations are implemented as vector operations, allowing the system to use SIMD operations. Signal and topic label data is stored using compact integer representations, speeding up lookup times. The topic database is partitioned, allowing many calculations to be highly parallelized.

3.2 Experimental Results

We performed a number of experiments that tested RACCOONDB's output quality and runtime performance. In this section, we will briefly summarize the runtime results and present a novel evaluation of querying RACCOONDB with less than perfect information. The experiments center around estimating several different target phenomena (box office sales, flu activity, etc.), which we chose because of the availability of historical ground truth data and their use in past nowcasting research. The topics we used were n-grams of up to four words in length extracted from 40 billion tweets collected over four years. An in-depth discussion of these experiments and others is available in our full paper [5].

	PostgreSQL	Spark	RACDB
1 core	> 6 h	–	3.6 s
30 cores	–	1173 s	1.2 s

Table 4: Average nowcasting query processing times for PostgreSQL, Apache Spark, and RACCOONDB.

One of the goals of RACCOONDB is to process nowcasting queries in interactive time—something we have found traditional data processing systems are unable to do. Table 4 compares the query processing times for RACCOONDB and two popular data systems (PostgreSQL and Apache Spark), using two different levels of parallelization. We averaged runtimes across our six target phenomena. Because PostgreSQL and Spark are very inefficient at nowcasting query processing, there runtimes are quite poor. In contrast, RACCOONDB is able to process queries orders of magnitude faster, achieving interactive runtimes using few resources.

3.2.1 Evaluating Quality

The feature engineering loop is human-driven, and while that human may be an expert in the problem domain, she may be less informed about what constitutes quality features for a nowcasting model. When this is the case, further iterations of feature engineering are often needed to narrow in on the high-quality features. In the following experiments, we show that RACCOONDB can still produce quality results under different levels of user knowledge and with varied amounts of error in the user's query. These experiments are compared to a

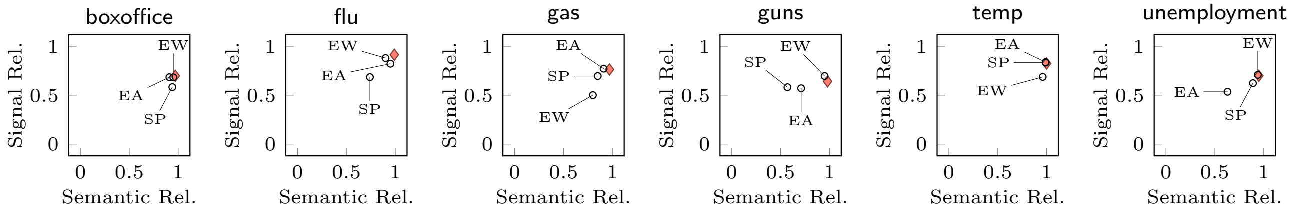
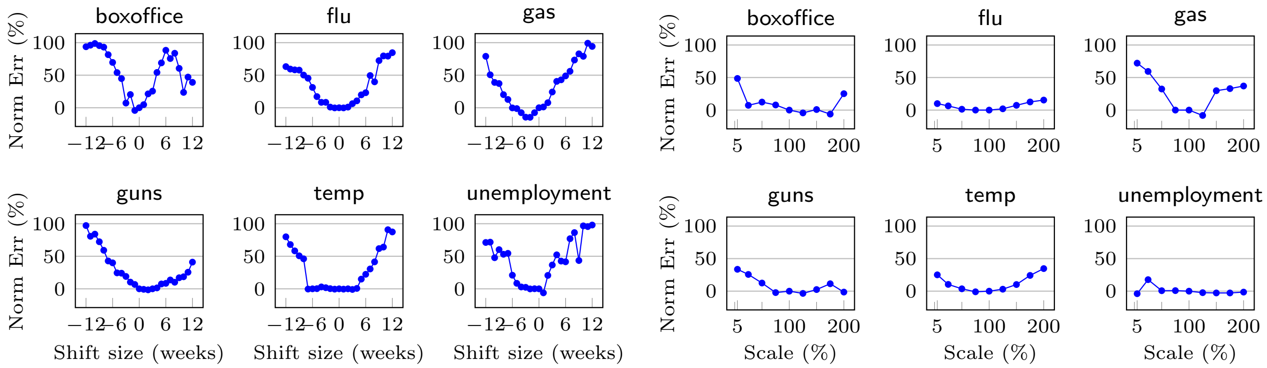


Figure 5: Our less-informed users EqWidth (EW), EqAmp (EA), and SeasonalPeaks (SP), compared to the fully-informed user (red diamond).



(a) Error from shifting user signal query peaks, relative to our distant supervision user model.

(b) RACCOONDB user interface

Figure 6: Effects of various types of error in the user signal query.

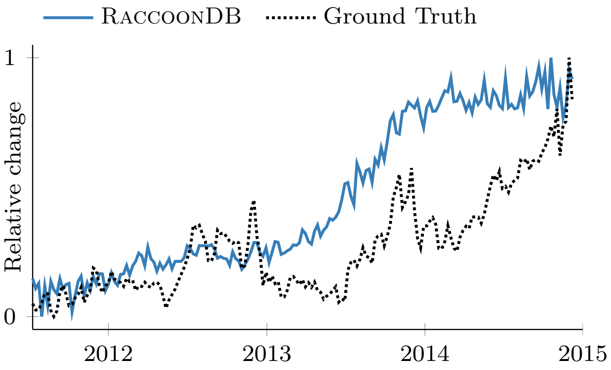
baseline *distant supervision user model*, where we assumed the user did not have access to accurate ground truth signal data and created the query signal based on a rough estimate of the target phenomenon. To simulate this, we created signals based on the actual ground truth signals by finding major peaks in the signal and synthesizing partial user signals consisting only of smooth representations of the actual peaks.

Performance of Less-Informed Users — To test how resilient RACCOONDB is to varying levels of user knowledge, we created three variants of our distant supervision user model, referred to as our *less-informed user models*. Assuming that high-quality semantic components are relatively easy for a user to construct, we focused on the more difficult part: the query signal. For the EqWidth user model, our simulated users only know when major peaks occur in the signal and not how long they last, so the synthesized peaks all have the same three-month width. For EqAmp, users have no intuition about peak height, so the synthesized peaks have the correct width, but all have equal amplitudes. Finally, for SeasonalPeaks, users only know about peaks within the last year (2014), so the synthesized peaks from 2014 are seasonally repeated over previous years (2011 - 2013).

Figure 5 shows the accuracy of our different less-informed user models using RACCOONDB. Displayed in the figure are EqWidth (EW), EqAmp (EA), and SeasonalPeaks (SP). In most cases, the less-informed user models still perform reasonably well compared to the distant supervision user model. Certain target phenomena are not as resilient to these variations though. For example, unemployment has a significant drop in semantic relevance when using the EqAmp user model. This is due to a multi-year downward trend in unemployment during this time period that reflected a general improvement in the US economy. Since EqAmp did not reflect this trend, RACCOONDB selected some unrelated topics in addition to the unemployment-related topics, obtaining a poor 0.63 semantic relevance. On average, though, the user models do not show a large decrease in accuracy: EqWidth's semantic and signal relevance decreases by just 6% and 8%; EqAmp by just 13% and 7%; and SeasonalPeaks by just 15% and 12% respectively.

MANUAL Query	Ground Truth Correlation
“solar panels”	0.13
“solar panel installation”	0.00
“solar installations”	-0.22
“solar savings”	-0.19
“solar”	0.06
“solar city”	0.30
“alternative energy”	-0.54
“photovoltaics”	-0.37
“solar energy”	-0.10
“purchase solar”	-0.16
“solar manufacturing”	-0.22
“home solar”	-0.36
RACCOONDB Query	
User signal + “solar panels”	0.73
User signal + “solar panel energy”	0.75
User signal + “solar panel installation”	0.71
User signal + “solar revenue”	0.79
User signal + “solar”	0.80

(a) User queries



Result topics	
pct solar	less with diy solar
solar pool	sunmodule solar panel
#solar #solars	deals on diy solar
now solar power	uv solar
solar wind	our solar panels
tenesol solar	green with diy solar
...	...

(b) RACCOONDBresult

Figure 7: (a) Case study queries for predicting *solar panel installations* using a MANUAL approach and RACCOONDB, along with correlations with a holdout ground truth signal; *user signal* is a user-chosen rough estimate for the target and is the quarterly revenue for Solar City Corporation. (b) RACCOONDB’s result signal and topics for the “solar” query.

Resilience to User Query Errors — We also set out to demonstrate RACCOONDB’s resilience to errors within a user’s query. For instance, the user may have intuition about where her target trend peaks but may be off by several weeks. We introduced two types of errors to the distant supervision user model signals to simulate this. For the first, we shifted the peaks in weekly increments, between -12 and +12 weeks. Figure 6a shows how error increases with respect to a non-shifted signal. As would be expected, the result quality worsened as the shifts increased in either direction. However, the error was not catastrophic: even if a user is off by a few weeks with her estimated peaks, she could still achieve decent results. It is worth noting the behavior of boxoffice, where its results began to improve again when shifted more than six weeks into the future. This occurred due to the bimodal nature of movie box office sales, which peak in both the summer and winter months. With enough error added, the user’s signal query began to accurately describe a different peak in the actual box office data.

The second error we explored is with regards to the width of each peak. We scaled the peak widths from 5% up to 200% of the original peak widths from our distant supervision user model signals (Figure 7b). Results were similar to our peak shifting experiment, where only the extreme errors resulted in a poor-quality result. RACCOONDB gives users some room for error when defining their query signals.

3.3 Related Work

The idea of RACCOONDB [5] as an optimized, general nowcasting query system draws upon our experience with designing previous nowcasting systems [7–9] and various other areas of database research: feature selection [26, 48], query optimization [45], and multi-criteria ranking [27]. We have also presented a demonstration system highlighting the user interaction of RACCOONDB [6].

4 A RACCOONDB Case Study

While our experiments have validated our technical claims for RACCOONDB, we wanted to see how useful our system is in a real-world use case, where a researcher wants to estimate a novel target phenomenon. How does the output quality and time required by the user compare to a traditional approach to nowcasting? To test this, we assigned one of our authors to be a test subject to perform such a task, while the other two authors were assigned to choose a target phenomenon, with the ground truth data for the target hidden from the test subject. Note that this study should not be confused with a formal user study; this is instead a simple, informal evaluation.

The chosen target was "US solar panel installations in the last five years," which would be evaluated against a ground truth signal of weekly photovoltaic installation counts from the Open PV Project [44]. The test subject was instructed to estimate this target using two different approaches. In the first (MANUAL), he was instructed to follow a traditional approach to nowcasting, manually searching for topics by keyword in our topic database and then use the signals of the topics found to build a model to estimate the target. In the second, he was instructed to use our RACCOONDB demonstration system [9] to estimate the target.

The test subject started the case study by spending several minutes searching online for a rough estimate of the target's trend, and he found the quarterly revenue for Solar City Corporation, which he assumed would share a similar trend with the target. Table 7a summarizes the test subject's queries for both estimation approaches. When using MANUAL to estimate the target, the test subject queried for several minutes, trying to find topic signals that matched the Solar City revenue signal, but he had very little success in his searches, averaging just 0.23 correlation (using absolute value) with the ground truth signal.[2] When using RACCOONDB over the course of a few minutes, he submitted five queries with various keywords and included the Solar City revenue signal with each query. When presented with the results, he adjusted the signal vs. semantic weighting slider to find the result with "relevant-looking" topics and a signal that "matched well" with the revenue signal. Figure 7 shows the user-chosen RACCOONDB result for his "solar" query. Across his five queries and chosen results, the results averaged 0.75 correlation with the ground truth signal.[3]

While the total time spent querying with the two approaches was not drastically different (under 10 minutes for MANUAL and under 5 minutes for RACCOONDB), the test subject was able to find a high-quality result with his first RACCOONDB query; subsequent queries were simply explorations into improving that result. In contrast, the test subject submitted many reasonable queries with MANUAL and still received very low-quality results. When processed by RACCOONDB, many of these same queries (e.g., "solar panels") returned high-quality results due to RACCOONDB's use of query expansion principles and its two-part query. This case study, while informal, provides additional confidence in RACCOONDB's real-world usage.

5 Other Opportunities

Human-in-the-loop feature engineering will likely remain an important part of building successful machine learning systems. The feature engineering loop shown in Figure 1 contains many areas that are prime opportunities for research by the database and systems communities.

Step 1 of the feature engineering loop—writing feature code—is where the feature engineer applies their domain expertise and creativity to create effective features. RACCOONDB assists the user in this area by semantically expanding the user's textual query, allowing more coverage over a diverse dataset than a less-informed user may be able to provide. Tools like FeatureInsight [15] have begun to help feature engineers visualize the effects of features on classification problems, but more advanced feature visualization systems (perhaps a visualization query system like SeeDB [45]) could help feature engineers discover subtle properties of raw data

[2] We also tried performing PCA on the chosen topics' signals and using the first factor as the target's estimate—similar to what RACCOONDB does to aggregate topics into a final estimate—but the results did not improve.

[3] Certain scientific fields will find this correlation to be quite low, but social sciences generally find anything above 0.5 to be strong.

that would make effective features. Other potential directions include incorporating automated or assisted data cleaning and transformation directly into the feature engineering workflow; integrating a data cleaning system (e.g., [28, 31, 38]) with real-time feedback from a machine learning pipeline; and applying deep learning methods [13, 18, 35] to automatically create feature extractors.

Related to feature code development is the problem of labeling training examples in datasets that are too large to feasibly label by hand. Research in distant supervision techniques [37] and label generation from noisy human-provided data [21] has shown promise. Further reducing this bottleneck would reduce the end-to-end time needed to create a successful machine learning system.

A source of significant downtime for a feature engineer is waiting for feature code to be applied to a large dataset. ZOMBIE attacks this problem by automatically finding a good subset of the data to process, while RACCOONDB takes advantage of application-specific properties of the data to aggressively prune its large corpus for each query. Other optimization approaches might include caching and shared computation [45] that take into account the evolution of feature code: feature sets may change incrementally, allowing some results from previous iterations of the feature engineering loop to be reused. On the other hand, raw data items that have the same features on one iteration of feature code may have vastly different features on another. Efficiently partitioning the raw data and parallelizing the computation under these dynamic conditions also pose interesting challenges.

Training a machine learning system can also be a lengthy part of the feature engineering workflow. Several projects have targeted training large models in a distributed fashion, such as MLlib [36] and MLbase [30, 43], as well as model selection management systems [32]. Further, determining the ideal hyperparameters for a machine learning model has seen some attention (e.g., from the MLlib team) but remains an open problem. Feature selection [26] (as opposed to feature engineering) is important to optimizing the performance of machine learning systems and has been looked at by several research groups [33, 48]. The goals of these projects are generally orthogonal to those of feature engineering; they aim to quickly train and configure the final machine learning model, rather than help the feature engineer evaluate the effectiveness of a particular feature set. Applying these optimization methods to evaluating feature effectiveness may require a different set of system requirements and novel techniques, such as developing approximate machine learning models.

6 Conclusion

Feature engineering is an import area of research that is inherently data-centric. In this paper, we presented two examples of systems aimed at improving the feature engineer's workflow by minimizing runtime and assisting with the creation of effective features from large raw datasets. The database community is in an ideal position to contribute to new feature engineering projects with its decades of experience in managing, optimizing, visualizing, and otherwise working with data.

Acknowledgements

This work, along with that of ZOMBIE and RACCOONDB, is supported by NSF grants 0903629, 1054913, 1064606, and 1131500, as well as by gifts from Yahoo! and Google.

References

[1] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica, *BlinkDB: Queries with bounded errors and bounded response times on very large data*, EuroSys, 2013.

[2] Michael Anderson, Dolan Antenucci, Victor Bittorf, Matthew Burgess, Michael J Cafarella, Arun Kumar, Feng Niu, Yongjoo Park, Christopher Ré, and Ce Zhang, *Brainwash: A data system for feature engineering.*, CIDR, 2013.

[3] Michael R Anderson and Michael Cafarella, *Input selection for fast feature engineering*, ICDE, 2016.

[4] Michael R. Anderson, Michael Cafarella, Yixing Jiang, Guan Wang, and Bochun Zhang, *An integrated development environment for faster feature engineering*, PVLDB **7** (2014), no. 13, 1657–1660.

[5] Dolan Antenucci, Michael R. Anderson, and Michael Cafarella, *A declarative query processing system for nowcasting*, PVLDB **10** (2017), no. 3, 145–156.

[6] Dolan Antenucci, Michael R Anderson, Penghua Zhao, and Michael Cafarella, *A query system for social media signals*, ICDE, 2016.

[7] Dolan Antenucci, Michael Cafarella, Margaret C Levenstein, Christopher Ré, and Matthew D Shapiro, *Using social media to measure labor market flows*, Working Paper 20010, National Bureau of Economic Research, March 2014.

[8] Dolan Antenucci, Michael J Cafarella, Margaret Levenstein, Christopher Ré, and Matthew Shapiro, *Ringtail: Feature selection for easier nowcasting.*, WebDB, 2013.

[9] Dolan Antenucci, Erdong Li, Shaobo Liu, Bochun Zhang, Michael J Cafarella, and Christopher Ré, *Ringtail: A generalized nowcasting system*, PVLDB **6** (2013), no. 12, 1358–1361.

[10] Nikos Askitas, Klaus F Zimmermann, et al., *Detecting mortgage delinquencies*, Tech. report, IZA, 2011.

[11] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer, *Finite-time analysis of the multiarmed bandit problem*, Machine Learning **47** (2002), no. 2-3, 235–256.

[12] Brian Babcock, Surajit Chaudhuri, and Gautam Das, *Dynamic sample selection for approximate query processing*, SIGMOD, 2003.

[13] Yoshua Bengio, Aaron Courville, and Pascal Vincent, *Representation learning: A review and new perspectives*, IEEE Transactions on Pattern Analysis and Machine Intelligence **35** (2013), no. 8, 1798–1828.

[14] Anant Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J Elmore, Samuel Madden, and Aditya G Parameswaran, *Datahub: Collaborative data science & dataset version management at scale*, CIDR, 2015.

[15] Michael Brooks, Saleema Amershi, Bongshin Lee, Steven M Drucker, Ashish Kapoor, and Patrice Simard, *FeatureInsight: Visual support for error-driven feature ideation in text classification*, VAST, 2015.

[16] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden, *Optimizing database-backed applications with query synthesis*, ACM SIGPLAN Notices **48** (2013), no. 6, 3–14.

[17] H. Choi and H. Varian, *Predicting the present with Google Trends*, Tech. report, Google, Inc., 2011.

[18] Adam Coates, Andrew Y. Ng, and Honglak Lee, *An analysis of single-layer networks in unsupervised feature learning*, AISTATS, 2011.

[19] Jeffrey Dean and Sanjay Ghemawat, *MapReduce: Simplified data processing on large clusters*, OSDI, 2004.

[20] Pedro Domingos, *A few useful things to know about machine learning*, Communications of the ACM **55** (2012), no. 10, 78.

[21] Henry R Ehrenberg, Jaeho Shin, Alexander J Ratner, Jason A Fries, and Christopher Ré, *Data programming with DDLite: Putting humans in a different part of the loop*, HILDA, 2016.

[22] David Ferrucci, *An Overview of the DeepQA Project*, AI Magazine (2012).

[23] Minos N Garofalakis and Phillip B Gibbons, *Approximate query processing: Taming the terabytes.*, VLDB, 2001.

[24] J. Ginsberg, M. H. Mohebbi, R. Patel, L. Brammer, M. S. Smolinksi, and L. Brilliant, *Detecting influenza epidemics using search engine query data*, Nature (2009).

[25] Anthony Goldbloom, *Kaggle: The home of data science*, Extract SF, 2015.

[26] Isabelle Guyon and André Elisseeff, *An introduction to variable and feature selection*, Journal of Machine Learning Research **3** (2003), 1157–1182.

[27] Ihab F Ilyas, Walid G Aref, and Ahmed K Elmagarmid, *Supporting top-k join queries in relational databases*, VLDB Journal **13** (2004), no. 3, 207–221.

[28] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer, *Wrangler: Interactive visual specification of data transformation scripts*, SIGCHI, 2011.

[29] Nodira Khoussainova, YongChul Kwon, Magdalena Balazinska, and Dan Suciu, *SnipSuggest: Context-aware autocompletion for SQL*, PLVDB **4** (2010), no. 1, 22–33.

[30] Tim Kraska, Ameet Talwalkar, John C. Duchi, Rean Griffith, Michael J. Franklin, and Michael I. Jordan, *MLbase: A distributed machine-learning system*, CIDR, 2013.

[31] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J Franklin, and Ken Goldberg, *ActiveClean: Interactive data cleaning for statistical modeling*, PLVDB **9** (2016), no. 12, 948–959.

[32] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M Patel, *Model selection management systems: The next frontier of advanced analytics*, ACM SIGMOD Record **44** (2016), no. 4, 17–22.

[33] Arun Kumar, Jeffrey Naughton, Jignesh M Patel, and Xiaojin Zhu, *To join or not to join? thinking twice about joins before feature selection*, SIGMOD, 2016.

[34] Arun Kumar, Feng Niu, and Christopher Ré, *Hazy: Making it easier to build and maintain big-data analytics*, Communications of the ACM **56** (2013), no. 3, 40–49.

[35] Quoc V. Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Greg Corrado, Kai Chen, Jeffrey Dean, and Andrew Y. Ng, *Building high-level features using large scale unsupervised learning*, ICML, 2012.

[36] Xiangrui Meng, Joseph Bradley, B Yuvaz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D Tsai, Manish Amde, Sean Owen, et al., *MLlib: Machine learning in apache spark*, Journal of Machine Learning Research **17** (2016), no. 34, 1–7.

[37] Mike Mintz, Steven Bills, Rion Snow, and Dan Jurafsky, *Distant supervision for relation extraction without labeled data*, ACL-IJCNLP, 2009.

[38] John Morcos, Ziawasch Abedjan, Ihab Francis Ilyas, Mourad Ouzzani, Paolo Papotti, and Michael Stonebraker, *DataXFormer: An interactive data transformation tool*, SIGMOD, 2015.

[39] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause, *Learning programs from noisy data*, ACM SIGPLAN Notices, vol. 51, ACM, 2016, pp. 761–774.

[40] Christopher Ré, Amir Abbas Sadeghian, Zifei Shan, Jaeho Shin, Feiran Wang, Sen Wu, and Ce Zhang, *Feature engineering for knowledge base construction*, IEEE Data Engineering Bulletin **37** (2014), no. 3.

[41] Steven L Scott and Hal Varian, *Bayesian variable selection for nowcasting economic time series*, Economics of Digitization, University of Chicago Press, 2014.

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

[43] Evan R Sparks, Ameet Talwalkar, Daniel Haas, Michael J Franklin, Michael I Jordan, and Tim Kraska, *Automating model search for large scale machine learning*, SoCC, 2015.

[44] *The Open PV Project*, `https://openpv.nrel.gov/`.

[45] Manasi Vartak, Sajjadur Rahman, Samuel Madden, Aditya Parameswaran, and Neoklis Polyzotis, *SeeDB: Efficient data-driven visualization recommendations to support visual analytics*, VLDB, 2015.

[46] Lynn Wu and Erik Brynjolfsson, *The future of prediction: How google searches foreshadow housing prices and sales*, Economics of Digitization, U. of Chicago Press, 2014.

[47] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica, *Spark: Cluster computing with working sets*, HotCloud, 2010.

[48] Ce Zhang, Arun Kumar, and Christopher Ré, *Materialization optimizations for feature selection workloads*, SIGMOD, 2014.

[49] Ce Zhang, Jaeho Shin, Christopher Ré, Michael Cafarella, and Feng Niu, *Extracting databases from dark data with DeepDive*, SIGMOD, 2016.