Input Selection for Fast Feature Engineering

Michael R. Anderson
University of Michigan
Ann Arbor, MI 48109
Email: mrander@umich.edu

Michael Cafarella
University of Michigan
Ann Arbor, MI 48109
Email: michjc@umich.edu

***Abstract*—The application of machine learning to large datasets has become a vital component of many important and sophisticated software systems built today. Such *trained systems* are often based on supervised learning tasks that require *features*, signals extracted from the data that distill complicated raw data objects into a small number of salient values. A trained system's success depends substantially on the quality of its features.**

Unfortunately, *feature engineering*—the process of writing code that takes raw data objects as input and outputs feature vectors suitable for a machine learning algorithm—is a tedious, time-consuming experience. Because "big data" inputs are so diverse, feature engineering is often a trial-and-error process requiring many small, iterative code changes. Because the inputs are so large, *each code change* can involve a time-consuming data processing task (over each page in a Web crawl, for example). We introduce ZOMBIE, a data-centric system that accelerates feature engineering through intelligent *input selection*, optimizing the "inner loop" of the feature engineering process. Our system yields feature evaluation speedups of up to 8x in some cases and reduces engineer wait times from 8 to 5 hours in others.

I. 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 employ machine learning techniques over very large datasets. The financial value of these systems far outstrips the traditional database software market, and though the social value is hard to quantify, it is undeniably high. Unfortunately, constructing trained systems is often a difficult endeavor, requiring many years of work even for the most sophisticated technical organizations. One reason for this is the difficulty of *feature engineering*.

A feature engineer writes code to extract representative features from raw data objects; the features are the input to a machine learning system. For example, consider a Web search engine that uses a trained regressor to estimate a Web page's relevance to a user's query; a feature engineer might write functions to determine if the query appears in italics in the document, if the query is in the document's title, and so on. The challenge of feature engineering is that good feature code must not only be programmatically correct; it must also produce features that successfully train the machine learning system.

Unfortunately, good features are difficult to devise [1], [2] and are a crucial bottleneck for many trained systems [3]–[5]. First, using a large set of diverse inputs (say, a set of crawled Web pages) means the engineer never quite knows the "input specification" and must resort to trial-and-error bug fixing as unexpected inputs are discovered. Second, predicting whether a proposed feature will in fact be useful to the machine learning algorithm is difficult; the programmer may implement a feature only to throw it away after it is found to be ineffective. As a result, feature engineers make many small iterated changes to their code and need to evaluate candidate features many, many times before achieving a well-trained machine learning model. But evaluating each change to the feature code can take hours, as it entails applying user-defined functions to a huge number of inputs and retraining a machine learning model. In this paper, we address the sheer amount of time required to perform this *feature code evaluation loop*.

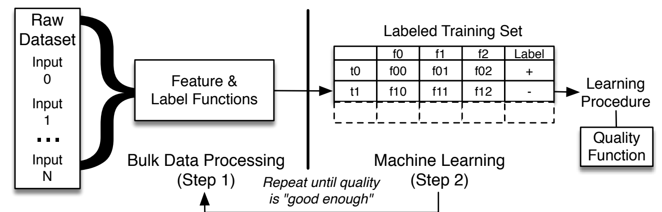


Fig. 1: In the feature engineering evaluation loop, bulk data processing and machine learning steps are interdependent but to date have been commonly implemented as separate systems.

System Goal — Figure 1 shows the feature evaluation loop as conventionally implemented. Having written new feature code, the engineer applies it to a large raw dataset (Step 1) using a bulk data processor, like MapReduce [6], Spark [7], or Condor [8]. This time-consuming step executes the feature code during a scan of the data, producing a training set sent to a machine learning system (Step 2). The engineer evaluates the resulting trained model's accuracy (probably using a holdout set of human-labeled examples). If the model is satisfactory, the engineer is done; otherwise, she modifies the feature code and returns to Step 1.

Feature engineering and feature selection are topics of growing importance in the database field because of their inherent data management challenges [1], [9]–[14]. Here, we model feature evaluation as a specific SQL query over a relation of raw data items. The query implements the feature code as a user-defined function (UDF) and computes a machine learning quality metric with a custom aggregation function. Feature engineering amounts to repeatedly running this query with small changes to the feature UDF. The query is run over large data volumes, so the engineer spends a huge portion of the evaluation loop waiting for the query result. Thus, *a feature engineer's productivity is bound by the feature evaluation query's runtime*. We treat accelerating feature evaluation as a query optimization problem; our metric of success is *the reduction in the time needed to evaluate feature code*.

Technical Challenge — To date, the bulk data processing and machine learning stages have typically been separate systems, with neither module aware of the larger feature evaluation loop. Since the actual bulk data processing system's output only matters insofar as it eventually produces a high-quality trained system, we can use feedback from the machine learning system to perform *input selection optimization*. Rather than scanning over the entire set of raw data—as is standard today—we can perform a variation of *active learning*: choose to process raw inputs that maximize the quality of the trained model, while minimizing runtime by *not* processing inputs that have little effect on the model's quality. Thus, our technical challenge is to build an effective training set while running the user's feature code as little as possible. The system's success can be measured by its speedup over traditional methods when producing the training set for a model of comparable quality.

Our Approach — We propose a version of the bulk data processing system (from Figure 1) that optimizes the feature extraction time through effective rule-based input selection. Our system replaces systems like MapReduce, Spark, and Condor, but our core techniques are orthogonal to their distributed processing methods; in the future, our approach could be combined with those systems. Our system has two stages: (1) offline indexing, where the system organizes the raw dataset into many *index groups* of similar elements before it is used and (2) online querying, where the system dynamically builds a high-quality subset of the data using index groups determined likely to yield useful feature vectors. This subset is used to train the machine learning system for feature code evaluation.

Traditional active learning techniques require computing the features for the entire raw dataset, and thus are much too expensive. Instead, we want to use an online method to quickly discover high-impact raw inputs. Our index groups allow us to use a multi-armed bandit algorithm: runtime identification of high-yield index groups is a good fit for a bandit problem's classic tradeoff of exploration vs. exploitation. By using carefully designed rewards for our bandit, our system quickly identifies relevant index groups for processing, while avoiding irrelevant ones. Our group-and-explore approach yields a substantial speedup over both conventional practice and a previous state-of-the-art method that builds a supervised classifier to choose inputs [15]. We have implemented this method in a prototype data processing system called ZOMBIE.[1]

Contributions and Outline — Our central contributions are:

- A proposed query model of the feature engineering workflow that captures current practices (Section II).
- A system design and algorithms for optimizing *input selection* (Sections III–IV).
- An implemented feature engineering evaluation system that can speed up the feature evaluation loop by **up to 8x** in some settings and has reduced engineer wait times from **8 to 5 hours** in others, compared to conventional methods (Section V).

We cover related work in Section VI. Finally, we conclude with a discussion of future work in Section VII.

[1]Like the undead, ZOMBIE goes straight for the "brains" of the input data.

Parameter	Description	Example
R	Raw dataset	Crawl of news sites with several million pages
$\mathcal{F}$	Feature functions	Boolean indicators of keywords and named entities
L	Label function	Label extractor from in-page tags
T	Training procedure	Multi-class Naïve bayes
Q	Quality function	Accuracy over holdout set
G	Quality goal	90% accuracy

TABLE I: Feature engineering inputs, given by $(R, \mathcal{F}, L, T, Q, G)$, with examples from a classification task: a classifier is trained with crawled news pages to automatically categorize future pages.

II. THE FEATURE EVALUATION QUERY

Feature engineering is a task parameterized with a 6-tuple of inputs $(R, \mathcal{F}, L, T, Q, G)$. The **raw dataset** R is a large corpus, such as a Web crawl. The feature engineer writes a set of **feature functions** $\mathcal{F}$ that extract features from a raw data item $r \in R$. Each function $f \in \mathcal{F}$ accepts an item r as input and emits a single value. Taken together, $\mathcal{F}(r)$ yields an unlabeled feature vector. A **label function** L provides a supervised label for a raw data item.[2] A machine learning **training procedure** T accepts the training set of labeled feature vectors and produces a trained model $T(\mathcal{F}(R), L(R))$. A **quality function** Q determines the quality of the trained model $Q(T(\mathcal{F}(R), L(R)))$. Finally, G is the **quality goal**: the ultimate quality level desired for the trained model.[3] Feature engineering, then, is task of writing and evaluating the feature code $\mathcal{F}$ such that $Q(T(\mathcal{F}(R), L(R))) \geq G$.

Table I summarizes these elements and shows examples from a document classification task. Five of the elements—R, L, T, Q, and G—are pre-determined and remain static for the duration of the task (or even across many tasks). A feature engineer adds to or modifies the functions $\mathcal{F}$ to maximize the value of Q. To do this, she writes and evaluates feature code in the **feature evaluation loop** in Figure 1 and defined as follows:

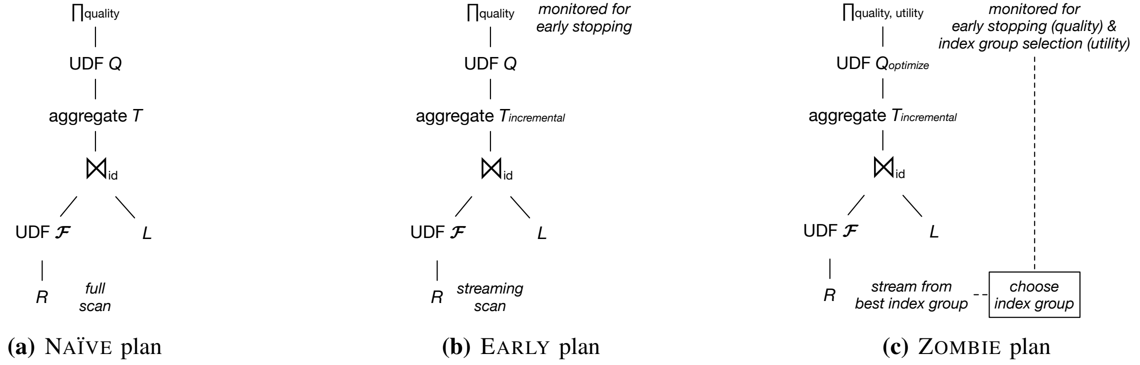
Definition 1 (Feature Evaluation Loop): Starting with R, L, T, Q and G, the feature engineer writes a set of feature functions $\mathcal{F}$, and then applies $\mathcal{F}$ and L to R to create a trained model $M = T(\mathcal{F}(R), L(R))$. The engineer evaluates the features in $\mathcal{F}$ by comparing the quality $Q(M)$ with goal G. If $Q(M)$ is less than G, the feature engineer modifies or adds to $\mathcal{F}$ and repeats the process until $Q(M) \geq G$.

A. Feature Evaluation Loop as a Query

We can model the inner loop of this workflow as a database-style query, shown in Algorithm 1. Lines 4 to 6 show the query as a hypothetical SQL statement. We consider the raw data R and the labels L to be relations. The feature code $\mathcal{F}$ is a UDF that produces the input for an aggregation function T that trains the learning system. Q is a UDF that accepts the trained model and emits a quality metric. The NAÏVE execution plan for this query is shown in Figure 2a. Because $\mathcal{F}(R)$ is computed by applying an expensive UDF with a full scan over a large dataset, it is slow; our goal is speed it up.

[2]We assume labeling is relatively inexpensive; e.g., labels may be drawn from an existing database or provided by distant supervision techniques [16].

[3]G might also be defined in terms of time: a feature engineer may have, say, 8 hours to produce the highest quality model possible.



(a) NAÏVE plan **(b)** EARLY plan **(c)** ZOMBIE plan

Fig. 2: Query execution plans. In (a), the UDF is applied to the entire dataset before T is invoked. In (b), the raw data items are pipelined to $T_{incremental}$ and Q. The output *quality* is continuously monitored to detect the early stopping point. In (c), data are pipelined to $T_{incremental}$ and $Q_{optimize}$. The output *quality* is monitored for early stopping, and *utility* informs the index group selection.

Algorithm 1 Feature Evaluation Loop

Input: Task (R, L, T, Q, G)
1: **repeat**
2: User writes or modifies feature code $\mathcal{F}$
3: **query**
4: SELECT $Q(T(\mathcal{F}(R.\text{data}), L.label)$ AS *quality*
5: FROM rawDataSet R, labels L
6: WHERE R.id = L.id
7: **done**
8: **until** *quality* $\geq G$
9: **return** $\mathcal{F}$

B. Common Practice: Subset

One popular method for speeding up the feature evaluation loop is an informal method we call SUBSET. The feature engineer creates a task-specific program S that consumes the entire raw input R and generates a smaller dataset $R' \subseteq R$. She then enters the evaluation loop, running the SQL query with R' instead of R. Because R' is small, the evaluation is fast. After exiting the loop, the engineer may perform an additional run with the full R to produce the final trained model.

In a series of conversations with feature engineers, we have found that SUBSET is popular, though it does not appear to be a topic of academic investigation.[4] The implicit assumption behind SUBSET is that $Q(T(\mathcal{F}(R'), L(R')))$ accurately approximates $Q(T(\mathcal{F}(R), L(R)))$. While SUBSET's popularity suggests it provides some benefits, its drawbacks are clear:

1) The program S takes extra effort to develop.
2) Applying S means scanning R at least once.
3) If R is large and diverse, it may be difficult to write a filter program S that identifies a high-quality subset.
4) When the engineer changes $\mathcal{F}$, the set of useful inputs may change. She may need to rewrite and rerun S.
5) Even if S produces a relevant subset, it may still contain unproductive "redundant" inputs and be unnecessarily large. If, for example, the engineer identifies useful Web domains, only a few examples from each domain may actually be useful.

SUBSET has overhead costs associated with writing and running S. Thus, we believe it is likely only useful when the costs can be amortized over many runs of the same feature code, probably when debugging the code itself is the goal. Further, choosing the right size for R' is difficult: too few inputs lead to an inaccurate estimate of the Q value, while too many quickly reduce the time advantage of using S. Moreover, the optimal size for R' will *change per task*.

The method we propose in this paper can be seen as an attempt to remove the weaknesses of SUBSET. An ideal input selection method would reduce development time and runtime overhead (weaknesses 1 and 2), choose good subsets automatically (weakness 3), and respond quickly to feature code changes (weakness 4). Finally, it would respond to the learner's changing requirements (weakness 5).

C. Approximation by Early Stopping

To address weaknesses 1–4 of the SUBSET method, we can use a method similar in spirit to the approximation and early stopping of online aggregation [18]. We can build the subset by adding one item at a time, retraining and re-evaluating the model after each addition. Once the model reaches a desired state, we can *stop early* and have an appropriately sized subset that contains enough useful inputs for the learning task.[5]

Figure 2b illustrates this EARLY execution plan. The raw data items, stored in random order on disk and accessed sequentially, are pipelined to $\mathcal{F}$, to $T_{incremental}$, which is retrained as new items arrive, and then to Q, whose output is monitored by a process that stops the query when a stopping criterion is met. In this paper, we stop the query when the learning curve (e.g., Figure 3) begins to plateau, though there are a number of valid stopping criteria, including reaching a certain accuracy level or after a specific amount of runtime. Researchers have also investigated algorithmic stopping criteria, though these are tailored to specific learning tasks [19], [20].

Figure 3 shows the effects of early stopping on a single iteration of the feature evaluation loop. The gray line is a learning curve taken from our experiments. The x-axis shows the runtime, which increases as more raw items from R are added to R' and processed by the UDF; the y-axis shows the classifier's accuracy after each item is added to R'. As R' grows, the accuracy curve flattens out as the marginal return

[4]The textbook *Data Mining* does, however, describe an interactive procedure for feature selection that is roughly akin to SUBSET [17].

[5]Training overhead is a concern, but many learning algorithms can be trained incrementally. We discuss this further in Section III-B.

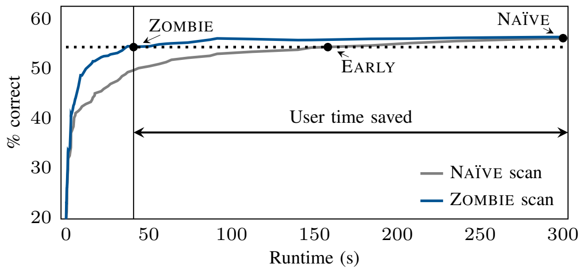


Fig. 3: Learning curves for our execution plans. The NAÏVE plan performs a bulk scan over R, while EARLY stops the bulk scan early to use a subset R'. ZOMBIE scans the data from index groups and stops early much sooner, translating to a time savings for the user.

for each new item in the training set decreases. Using all items in R, the classifier—with this particular feature set—achieves an accuracy of 56%. The dotted line shows 98% of the full accuracy; with early stopping, the trained model achieves nearly the full accuracy when R' is only half the size of the full R.

D. Optimizing the Approximate Query

With ZOMBIE, we also address SUBSET's weakness 5. We construct R' using only a minimal number of corpus's low-utility items (i.e., items that are redundant or irrelevant to the task), so R' consists mainly of high-utility items, and thus a high-quality model can be trained with a relatively small amount of data. This is similar to traditional active learning, described by Algorithm 2 [21]. The crucial difference is that for active learning, $\mathcal{F}(R)$ is already computed (on line 2) for all potential training examples—exactly what we wish to avoid. An active learning-based feature evaluation method would require the full scan of the NAÏVE plan plus significant overhead from using an active learning method to build a subset R'.

Algorithm 2 Active Learning-based Feature Evaluation

Input: Task $(R, \mathcal{F}, L, T, Q, B)$
1: *trainingSet* $= []$, $M = \emptyset$
2: *examples* $= \mathcal{F}(R)$
3: **repeat**
4: *best* $=$ chooseBestExample(M, *examples*)
5: *trainingSet*.append([*best*, L(*best*)])
6: $M = T$(*trainingSet*)
7: **until** |*trainingSet*| $== B$
8: **return** $Q(M)$

What ZOMBIE does instead is estimate the average utility of groups of similar raw data items in real time. Like the two-phase operation of many approximate query databases [22]–[24], our system first creates many sub-samples of the data in an offline phase; the raw data in R is organized into many groups of similar items, called *index groups*. Then, during the runtime query phase, the most relevant index groups are used to answer the user's query. Again, we cannot pre-compute the utility values; an item's usefulness directly depends on the features generated by $\mathcal{F}$. ZOMBIE learns the index group utility values in real time with a multi-armed bandit algorithm. We describe this algorithm in detail in Section III-B.

Figure 2c shows the optimized query execution plan. Like the previous early stopping method, the raw data is pipelined through $T_{incremental}$, which incrementally trains the learning system, and $Q_{optimize}$, which, in addition to the *quality* value monitored for early stopping, also produces a *utility* value that quantifies the usefulness of the item just processed. The *utility* value is used by the "choose index group" operation to estimate which index group has the highest average utility. This operation is key to our input selection optimization method and is the focus of the remainder of this paper.

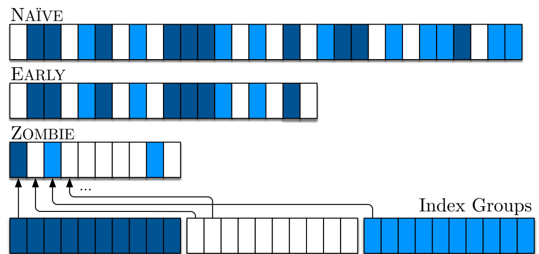


Fig. 4: Illustration of the subset R' created by each execution plan. Colored rectangles represent raw data items. High-utility items are white; low-utility ones are dark. The NAÏVE plan processes R in its entirety. EARLY processes R', chosen by streaming R and stopping early. ZOMBIE patches together R' from index groups.

Figure 3 illustrates the benefit of using ZOMBIE's input selection method. The blue line shows a full scan done with ZOMBIE, with the dot showing the early stopping point. With ZOMBIE, the classifier can reach nearly its full potential in a much shorter runtime than with EARLY and in just a small fraction of the time needed by NAÏVE. Figure 4 illustrates the difference in the subset R' created by each method. NAÏVE processes all of R, EARLY processes truncated version R, and ZOMBIE patches together R' from its index groups, as it learns which will provide the highest-utility items.

E. Deployment and Limitations

There are settings in which ZOMBIE may not be helpful. First, our user model itself may not apply. In domains where it is difficult for humans to provide insight (e.g., signal processing), a "generate-and-select" approach—a huge number of candidate features are hypothesized and data-centric methods select the best ones—may be more useful than the "engineer-and-test" approach described here. Even when it is possible to provide domain insight, "generate-and-select" needs little human attention, and the resulting features may be sufficient if the high accuracy enabled by domain knowledge is not required.

Alternatively, deployment details may reduce the need for ZOMBIE. Some machine learning systems have huge training or evaluation costs; any reduction in feature extraction time may be negligible compared to the learning system's inherent costs. (We examine this point experimentally in Section V-E1.) Finally, a few organizations may have massive parallel infrastructures that make all but the most burdensome computations irrelevant.

Despite these limitations, ZOMBIE directly targets a setting that we believe is extremely common (from both published engineering accounts and personal experience): a feature engineer trying to improve a trained system's accuracy by creating features that embody domain expertise.

III. SYSTEM ARCHITECTURE

We can now describe our design for ZOMBIE, depicted in Figure 5. First, we discuss ZOMBIE's two-phase execution model. We then detail the system's two major components: the *input selector* and the *physical indexes*.

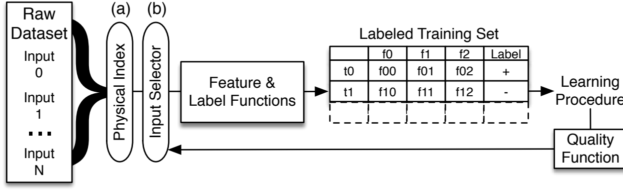


Fig. 5: ZOMBIE's basic architecture. The novel components are (a) the Physical Index and (b) the Input Selector.

A. Execution Model

ZOMBIE executes in two stages, similar to the indexing and query processing stages of a relational database. In *system initialization*, the system is given dataset R but has no access to any other part of the task description. System initialization is a one-time pre-processing of R that builds a physical index $\mathcal{I}$— a set of index groups that each contain a set of similar raw inputs. System initialization may be costly, but its runtime can be amortized over many rounds of code modification (and further, many distinct learning tasks if they use the same input set R). In the *feature evaluation query*, the system is given the feature engineering parameters $(R, \mathcal{F}, L, T, Q)$. The query is repeated with a modified $\mathcal{F}$ until the quality goal G is achieved.

Our design is shown in Figure 5. It is driven by the basic framework from Figure 1 and the query model from Section II but includes two novel components. ZOMBIE completely takes over the role of the bulk data processing system from Figure 1, but it invokes the machine learning system as an external library component. The raw data items are organized into a physical index (a) consisting of a series of index groups built during system initialization. Instead of simply scanning and fully processing R during feature evaluation, ZOMBIE uses an *input selector* module (b) that repeatedly chooses the next raw data item to process from the index groups. The system stops when the trained model meets the user's stopping criterion.

B. Input Selector

ZOMBIE's input selector is the core of the system, repeatedly choosing the next raw data input $r \in R$ to process with the feature functions $\mathcal{F}$. As items are processed, the input selector learns which index groups are most likely to produce high-utility inputs and uses those groups as the source of the next selected inputs. By prioritizing inputs that are most likely to improve Q, our approach is roughly comparable to the active learning strategy of expected error reduction [21].

Input Selection Algorithm — The selector's basic execution loop is shown in Algorithm 3. On line 4, it chooses an index group from which to select an input, using current statistics and utility information. On lines 5 and 6, it fetches the item and applies $\mathcal{F}$ and L to create a labeled feature vector. On line 7, T trains a new model. On lines 8 and 9, it measures and records the model's quality the chosen input's utility with Q. The loop ends on line 10 when R is exhausted or `shouldStop()` returns true (as discussed in Section II-C).

ZOMBIE's performance is linked to its ability to accurately predict the utility of applying $\mathcal{F}$ to an input from a given index group. We track the utility of a single raw input r by observing changes in the model after adding $\langle \mathcal{F}(r), L(r) \rangle$ to the training set (by using T to train a new model and using Q to evaluate it). We track index group utility by aggregating utility values of previously processed inputs from the group. We discuss this important part of our algorithm—embedded in the `chooseIndexGroup()` function—in Section IV.

Algorithm 3 Input Selection Algorithm

Input: Task $(R, \mathcal{F}, L, T, Q)$, index $\mathcal{I}$
1: *trainSet* = {}; *utilities* = []
2: *quality* = 0; *model* = $\emptyset$;
3: **repeat**
4: *bestIdx* = chooseIndexGroup(*utilities*)
5: $r = \mathcal{I}$(*bestIdx*).getNext()
6: *trainSet* = *trainSet* $\cup \langle \mathcal{F}(r), L(r) \rangle$
7: *model = T(trainSet)*
8: *(quality, utility) = Q(model)*
9: *utilities*[*bestIdx*].append(*utility*)
10: **until** |*trainSet*| == $|R|$ **or** shouldStop(*quality*, *model*)
11: **return** *model*, *quality*

Managing Overhead — Depending on how T and Q are implemented, running them could be computationally expensive. Indeed, the expense might undermine any advantage gained by avoiding unproductive raw inputs. If T can incrementally retrain the model with each input, doing so should yield large performance benefits. ZOMBIE does not strictly assume incremental retraining, but the lack of it may add substantial overhead. However, incremental procedures exist for a range of popular machine learning methods, including neural networks [25], SVMs [26], and decision trees [27]. In Section V, we examine several tasks, including one without incremental retraining.

C. Index Groups

At system initialization, we group the raw inputs in R into a task-independent set of index groups. We create an inverted index $\mathcal{I}$ that contains one entry for each index group. The key is a unique identifier for the group, while the key's indexed posting list is an unordered set of raw inputs. An input can be present in multiple groups.

Choosing a good set of index groups for $\mathcal{I}$ is a core challenge for ZOMBIE. Because feature code in $\mathcal{F}$ changes quickly and often (and re-indexing for each change would be too expensive), we assume $\mathcal{I}$ is built just once and serves a range of feature engineering tasks. $\mathcal{I}$ must be broadly useful over many runs of the feature engineering loop. We expect, though, a single *task-independent* $\mathcal{I}$ will serve many different versions of $\mathcal{F}$. If there are index groups within $\mathcal{I}$ with a concentration of useful inputs even slightly higher than the corpus as a whole, ZOMBIE can perform better than other methods.

This paper's core contribution lies in how the system exploits the grouped data, not in particular grouping methods. Our work assumes that the qualities that make a raw input useful for the learner will be reflected in a grouping of the input data. Our method relies on correlation between $\mathcal{I}$ and the output of the $\mathcal{F}$; if there is no relationship between the two, our method will be no better than random selection. Though this assumption might seem unreasonably strong, we find off-the-shelf, general-purpose clustering effective, for several reasons:

1) Experience has shown general-purpose clustering to be broadly useful across a huge range of domains, including Web page clustering [28], cancer detection [29], and network security [30].

2) The output of $\mathcal{F}$ and $\mathcal{I}$ can be correlated in unexpected ways that are useful for input selection. E.g., a feature describing document length may seem unrelated to a token-based $\mathcal{I}$, but our method will work if some tokens exist primarily in long documents.
3) As we show in Section V-D3, ZOMBIE yields substantial speedups over traditional input selection methods using even low-quality input groupings.

Index groups tailored to a particular run of ZOMBIE would surely allow for successful input selection, but the time advantage gained by selecting good raw inputs would be lost while waiting for the data to be grouped. Thus, we depend on a general, task-independent grouping done using standard clustering algorithms known to be successful with a wide variety of data types, such as k-means. We examine grouping methods in depth in Section V-A. In certain cases, if the index groups and features are truly uncorrelated (a situation we view as unlikely) it might make sense to re-group the data using a different user-defined method, similar to reindexing a database.

D. Physical Access

ZOMBIE is designed for processing large datasets, so the selector should be able to handle raw input sets larger than available memory. Our physical indexes are essentially inverted index posting lists and so are compatible with handling larger-than-RAM datasets. However, even modest memory sizes are quite large; we assume each posting list has a buffered in-memory portion continually replenished by a background process scanning items from disk.

IV. PREDICTING INPUT UTILITY

In this section, we cover how the input selector can effectively implement the `chooseIndexGroup()` function from Algorithm 3. Our solution is to learn at runtime a notional inverted mapping from user feature vectors output by $\mathcal{F}$ to the index groups in the index $\mathcal{I}$. The system creates this mapping by observing the utility of the feature vectors. The `chooseIndexGroup()` function then uses the inverted mapping to find high-utility index groups in $\mathcal{I}$.

A. Design Discussion

By processing many raw inputs $r \in R$ with $\mathcal{F}$, and thereby generating many $(r, \mathcal{F}(r))$ example pairs, we can likely build a high-quality mapping from the space of feature vectors to that of index groups. Such a mapping would be useful in choosing raw inputs, but building a high-quality mapping would require the costly processing of a large amount of example data and would often be unnecessary. Instead, exploiting a quickly built, medium-quality mapping may be better. In other words, we face the classic tradeoff between *exploration* (processing novel items to improve the mapping) and *exploitation* (using the current mapping to select the most useful data items).

Bandits — Researchers in fields ranging from online advertising to robotics have developed a number of solutions for such problems under the banner of reinforcement learning. A standard problem formulation in this area is the *multi-armed bandit* [31]. Consider a gambler choosing to pull an arm from one of a number of slot machines with different but unknown payout rates. Each pull yields a *reward* drawn from a distribution of values tied to that arm. The gambler must balance explorative arm-pulling to gather additional payout information against exploitative arm-pulling to maximize the reward using current knowledge. Our system fits well with this model: the input selector (the gambler) must choose which of the index groups (the arms) will supply the next raw data item.

Strategies — Many arm-pulling strategies have been developed to minimize *regret*, the difference between actual and optimal payouts. One popular strategy bases arm selection on comparing upper-confidence bounds (UCB) of estimated rewards rather than the estimated rewards themselves. UCB-based strategies have been shown to have near optimal regret bounds [32]. We use a UCB strategy to choose arms to pull—that is, index groups from which to fetch raw inputs—in our input selector. We now describe more precisely how we model the task.

B. Our Bandit Model

The multi-armed bandit in our system determines which index groups from the task-independent grouping $\mathcal{I}$ of the raw dataset R are most useful to the current task and which can be safely ignored. We define our bandit problem as follows:

- **A set of bandit arms.** We create one bandit arm for each key $k \in \mathcal{I}$. "Pulling arm a_k" means reading a random raw data item from $\mathcal{I}[k]$, processing it with $\mathcal{F}$ and L, and adding the result to the training set.
- **A reward function** to compute a pulled arm's payout. This is the *utility* value in Algorithm 3.

For each index group (i.e., key in $\mathcal{I}$), we must record the rewards received so far (lines 8 and 9 of Algorithm 3). The rest of our `chooseIndexGroup()` procedure comes from carefully defining the reward function.

C. Rewarding a Pull

The reward for an arm pull is how the bandit learns the pulled arm's utility to the current task. Careful design of the reward function lets us encourage the selector to prefer certain index groups over others. Consider, for example, a classification task where examples of one of the labeled classes are rare. Giving a high reward (i.e., utility value) to members of the rare class and a low reward otherwise will lead to the selection of index groups more likely to contain members of the rare class. For our experiments, we implemented four reward functions based on active learning and other machine learning techniques:

ClassBalance is designed to produce a training set that is more balanced in terms of relative class populations than is the raw data corpus. Real-world datasets are often heavily imbalanced, and restoring a degree of balance is often a first step in building a learning system [33]. The reward (or utility) $u_{\text{balance}}(r)$ for a selected item r is based on the ratio of that item's class in the current training set:

$$u_{\text{balance}}(r) = 1 - \frac{n_{L(r)} - n_{Lmin}}{n_{Lmax} - n_{Lmin}} \quad (1)$$

where $n_{L(r)}$, n_{Lmin}, and n_{Lmax} are the counts of the items in the training set belong to r's class, the least populated class, and the most populated class, respectively. The label function L determines the item's class.

Uncertainty is a reward based on the active learning technique of uncertainty sampling, where the item selected is the one the classifier is most uncertain about when predicting its class label [34]. Using a probabilistic classifier, a prediction's uncertainty is defined in terms of the probabilities of the two most likely class labels. That is, for a raw data item $r \in R$ with feature vector $\mathcal{F}(r)$, $p(c_1|r)$ is the probability of the most likely label for r, $p(c_2|r)$ is the probability of the next most likely label, and the reward $u_{\text{uncert}}(r)$ is given by the uncertainty:

$$u_{\text{uncert}}(r) = 1 - (p(c_1|r) - p(c_2|r)) \tag{2}$$

ClassifyError gives a high reward to items for which the currently trained classifier predicts the wrong class label. The reward $u_{\text{error}}(r)$ is defined as:

$$u_{\text{error}}(r) = \begin{cases} 1, & \text{if classifier error} \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

This method is somewhat related to boosting, where a series of weak classifiers are trained on different subsets of training data and then combined. When a weak classifier's training set is constructed, previously misclassified examples are preferred, to emphasize the "hardest" training examples [35].

InfoTrace is our reward for regression tasks, based on the idea of minimizing the training set's variance by maximizing the Fisher information, a technique from active learning and optimal experiment design [21]. To encourage the selection inputs that yield a large increase in the trace of the learner's Fisher information matrix I, the reward $u_{\text{info}}(r)$ is given by:

$$u_{\text{info}}(r) = \beta\,(\text{trace}(I_{new}) - \text{trace}(I_{old})) \tag{4}$$

For linear regression with a constant variance σ^2, $I = \frac{1}{\sigma^2}\mathbf{X}^{\mathrm{T}}\mathbf{X}$, where $\mathbf{X}$ is the model's design matrix [36]. We chose $\beta = 0.2$ after testing a range of values ($0 < \beta \leq 1$) in our experiments.

D. Selecting an Arm

Arms are selected in our system using the UCB1 algorithm of Auer *et al.* [32]:

$$\text{UCB}_{a,t} = \mu_a + \alpha\sqrt{\frac{2\ln n}{n_a}} \tag{5}$$

where μ_a is the average reward of arm a, n_a is the number of pulls of arm a, and n is the overall number of pulls so far. The parameter α controls the size of the confidence bound, which sets the balance between exploration and exploitation in the system. We chose $\alpha = 2$ after testing a range of values.

E. Putting It All Together

We can now summarize our overall bandit algorithm for choosing raw inputs. At each invocation of `chooseInputGroup()`, we use the learner's statistics about previous pulls to estimate the reward and update the $\text{UCB}_{a,t}$ values (as in "Selecting an Arm" above) for each a_k in index $\mathcal{I}$. We then return the key with maximal UCB. Depending on the initial grouping method, a raw input can appear in more than one index group, so we may encounter previously processed raw inputs. If so, we do not invoke $\mathcal{F}$ but instead update the $\text{UCB}_{a,t}$ value for the pulled arm using that input's previous reward, and then again select the key with highest UCB.

Parameter	Setting
Index grouping method	cluster
Reward method	UNCERTAINTY
Minority class rarity	0.5%

TABLE II: Default experiment settings.

V. EXPERIMENTS

We ran four types of experiments to demonstrate ZOMBIE performs effective input selection under a range of tasks and system settings. First, we compared ZOMBIE's overall speedup to several baselines on a family of learning tasks. Second, we compared ZOMBIE to a common input selection method, SUBSET. Third, we varied internal algorithmic decisions: the reward function, the input grouping method, and the quality of the index groups. Finally, we varied two kinds of difficulty that impact input selection: feature function execution time and rarity of high-value data items in the raw corpus. Table II shows default settings for important system parameters, which were used except where stated otherwise.

We implemented ZOMBIE and our machine learning tasks (using Weka [37]) as a Java application of about 17,500 lines of code. (Our prototype replaces the bulk data processing platform in Figure 1, but our method could be integrated into existing data processing systems.) We deployed the system on an Amazon EC2 r3.xlarge instance with 30 GB of RAM.

A. Feature Engineering Workloads

We evaluated the system using two different learning tasks and four different index group creation methods.

Document Classification — We first tested a document classification task. The raw input dataset (R in the feature engineer's tuple) was a corpus of one million WikiText documents randomly drawn from Wikipedia, after removing all pages marked as "deleted" or "redirect." We labeled each document with a function (L) that derived a label from *Category* tags. The trained artifact predicted a novel page's class label. We tested two variants of this task: **DC2** labeled documents as either geography or other. The **DC6** task distinguished among six different labels: geography, politics, science, sports, videoGames, and other. For both tasks, other was the majority class. The remaining classes each comprised 0.5% of the corpus, except when explicitly varied for the experiments discussed in Section V-E2. The engineer's Q function returned the trained model's classification accuracy over a holdout set of 2,500 documents from each labeled class.

The training mechanism T for **DC2** and **DC6** was Weka's updatable multinomial naïve Bayes classifier. We tested a set of 40 feature functions as $\mathcal{F}$; each applied a single regular expression to the raw input document and returned the number of matches; thus, $\mathcal{F}$ resulted in a 40-element feature vector for each input. We also tried a more time-consuming $\mathcal{F}$ that applied the Stanford Named Entity Recognizer [38] to each text, yielding a vector of counts of the three NER types (organization, person, and location); the tasks that use NER features are **DC2-NER** and **DC6-NER**. The regular expression $\mathcal{F}$ averaged 1 ms per execution per input, while the NER $\mathcal{F}$ averaged 87 ms. We held $\mathcal{F}$ constant over each experiment to measure ZOMBIE's impact on a single iteration of the feature engineering loop.

Linear Regression — To test ZOMBIE on a non-classification problem, we created a regression task using the same Wikipedia raw dataset. The trained model predicts the page's length. This simple task has several desirable methodological qualities: (1) this prediction should be possible, as some topics naturally lend themselves to longer articles; (2) it is not trivially obvious how to make this prediction, so it is a good target for a trained system; (3) we can use the same data and features as our classification task, making feature function times comparable across the tasks; and (4) page length is an easy value for others to compute when trying to reproduce our results. We counted the page's length in bytes, normalized by the standard deviation of the page lengths. The Q function measured root mean squared error over a holdout set. For training procedure T, we used Weka's least squares regression module. The feature function set $\mathcal{F}$ was the 40-element vector described above.

Index Group Construction — We tested four methods to create index groups. For ZOMBIE's default cluster method, we used tf-idf normalized token counts as features for the k-means clustering implementation from scikit-learn [39]. (We chose $k = 500$ after testing values of k ranging from 100 to 10,000.) The remaining three methods were designed to test a variety of grouping methods and are not proposed for practical use. For the w2v index, we used Google's word2vec tool [40] to build a set of index groups. To do so, we provided the tokenized corpus to the tool to create a feature vector for every token. We then clustered these vectors using k-means clustering, with $k = 2000$ (chosen from a range of tested values) and created an index group for each cluster by assigning a document to a group if one of its tokens belonged to that index group's corresponding cluster. Documents could belong to multiple index groups. For the token index, we tokenized the Wikipedia documents and removed stop-listed and rare tokens (those with fewer than 50 occurrences). We created one index group per unique token, adding to each group the documents containing the corresponding token. The resulting index had roughly 35,000 index groups. For random, we assigned documents randomly to one of 500 index groups. Finally, to test the impact of index group quality on ZOMBIE, we evaluated a range of synthetic groupings, each with a different distribution of "useful" items.

B. *Overall Performance*

We measured the speedup of all input selection mechanisms relative to the EARLY technique—the early stopping execution plan discussed in Section II. For EARLY, raw data items were stored on disk in random order and processed sequentially. EARLY allows for a direct comparison to ZOMBIE's early stopping, though the NAÏVE full scan method may better represent current practice. Comparing to EARLY understates the actual speedup that feature engineers would see with ZOMBIE, since standard systems do not perform early stopping.

We also compared ZOMBIE to our implementation of another proposed input selection method called FILTEREDSCAN, from Ipeirotis *et al.* [15]. FILTEREDSCAN first runs in standard bulk processing mode, collecting example pairs of raw inputs and observed accuracy improvements. It trains a classifier with the gathered data to label new items as helpful or not. Helpful inputs are those that are expected to yield large accuracy gains. Then, candidate inputs are classified; helpful inputs are processed first and the rest deferred until helpful inputs have been exhausted. For our experiments, FILTEREDSCAN chose its first n inputs randomly and used commodity bag-of-words features to train the helpful-or-not model. For document classification, helpful items were in the minority class. For regression, the labels were based on the INFOTRACE reward: helpful items increase the information matrix's trace more than the average of previous inputs. We then trained a naïve Bayes classifier that, starting with the $n+1$ input, labeled new items as helpful or not. For each experiment, we tested a range of n values and chose the one yielding the best speedup.

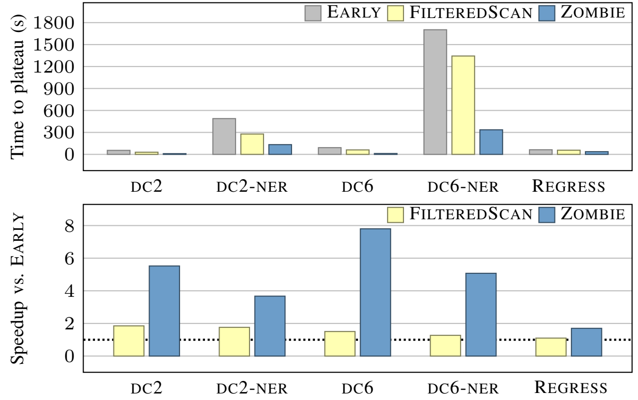


Fig. 6: Time to stopping point, as well as ZOMBIE's speedup over EARLY in reaching that point, for all of our experimental tasks. At bottom, the dotted line indicates EARLY's performance.

Methodology — As we showed in Figure 3, better input selection systems can reach high quality values rapidly, so the system can stop early. We measure performance by recording how long either EARLY or ZOMBIE took to reach a plateau in accuracy in its learning curve. This plateau is defined as the point where the tested accuracy values over a window of time change less than a user-specified *minimal change* value. In our experiments, we chose a value equal to ϵ times the final accuracy achieved by running the task to completion. We ran each experiment ten times. To show our system is robust to the user's choice of stopping point, we evaluated each result with ϵ varying from 0.01 to 0.05. We averaged the measured time to the stopping point over all ten runs and minimal change values. When we report speedup vs. EARLY, it is EARLY's stopping point time divided by ZOMBIE's stopping point time.

Results — Our basic results for the document classification and regression tasks are summarized in Figure 6. ZOMBIE yielded a gain over EARLY in all cases, with speedups up to nearly 8x. It also beat the FILTEREDSCAN method in all cases, usually substantially. ZOMBIE performed especially well on **DC6**; our index groups correlate well with all of our tasks, but EARLY displayed a slower learning rate on **DC6** than on the other tasks, giving ZOMBIE more room for improvement. The speedup numbers are important, but so are actual time savings. When using ZOMBIE, the feature engineer could reach the accuracy plateau for the **DC6** task in 12 seconds (processing about 11,800 items or 1.2% of the corpus), nearly eight times faster than the 92 seconds (9.1% of the corpus) required to reach the same level of accuracy with EARLY. FILTEREDSCAN was better than EARLY but still required 61 seconds.

ZOMBIE speedups over EARLY for the NER tasks are 3.6x and 5.1x for **DC2-NER** and **DC6-NER**, respectively. These speedups were smaller than **DC6**, but the high cost of the

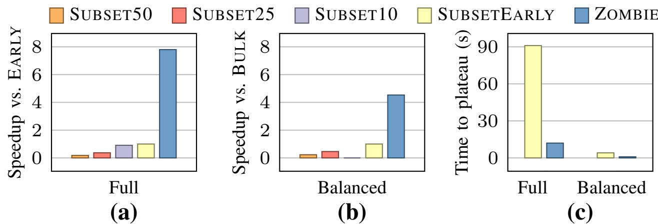


Fig. 7: SUBSET sizes tested on **DC6** using (a) the full dataset and (b) a class-balanced filtered dataset. SUBSET10 was too small in (b) and had no result. Execution time for both is shown in (c).

NER function still made the absolute time savings substantial. ZOMBIE reached the stopping point for **DC2-NER** in 2.2 minutes instead of the 8.1 minutes required for EARLY. For **DC6-NER**, ZOMBIE could stop after 5.6 minutes instead of EARLY's 28.3 minutes. Saving over 20 minutes per iteration of the feature engineering cycle would certainly improve an engineer's productivity over the course of a work day. For the **Regression** task, ZOMBIE showed a 1.7x speedup over EARLY, needing only 36 seconds versus EARLY's 63 seconds. FILTEREDSCAN's speedup over EARLY was relatively poor, with a less than 2x speedup for even the simplest **DC2** task. As tasks became more difficult—that is, with more classes to label—FILTEREDSCAN declined to almost the same level as EARLY. We could almost certainly improve FILTEREDSCAN's performance by tuning it to particular tasks, but this is exactly the type of extra human labor we aim to avoid.

C. Testing SUBSET

The SUBSET method (Section II-B) is a common approach to speeding up feature evaluation. The user has two main choices when building a subset: the method used to choose its contents and its size. Recall that in addition to the manual labor involved in writing the SUBSET program and its execution overhead, the user has a real challenge in formulating the SUBSET size; choosing too few samples will mean the system fails to meet the plateau-based stopping point, while too many will make the system run longer than is necessary. The feature engineer using SUBSET must make this guess in a preliminary phase *before* running any machine learning procedures.

We tested two subset selection methods on our **DC6** task: the *Full* method sampled randomly from the entire dataset, while the *Balanced* method sampled from a dataset that was filtered so it has a balanced number of class labels. The Full dataset contained one million items and the Balanced dataset had 30,000 items. We tested different user guesses for size: SUBSET10 (10%), SUBSET25 (25%), and SUBSET50 (50%), as well as the unrealistic SUBSETEARLY, which contained the exact number of items needed to reach the task's stopping point. This method acts as if the user could perfectly predict the number of items used by our EARLY baseline method. Averaged over 10 runs, SUBSETEARLY contained about 91,000 items for Full (9.1%) and 3,400 for Balanced (11.3%).

Figure 7 shows the results for these experiments. As expected, SUBSETEARLY was better than any of the subsets that represent more realistic user guesses (SUBSET10, SUBSET25, and SUBSET50). In the Balanced experiment, we show no result for SUBSET10, since it was too small to reach the stopping point (and the feature engineer would not accept the subset's trained model). For SUBSET25 and SUBSET50, the Balanced

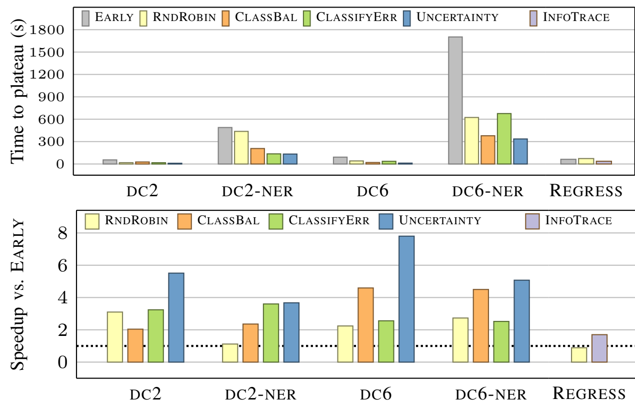


Fig. 8: Time to stopping point, as well as ZOMBIE's speedup over EARLY in reaching that point, for our different bandit reward methods. At bottom, the dotted line shows EARLY's performance.

method was fast in terms of raw execution time, taking less than 10 seconds to complete the task. ZOMBIE was the best approach on both the Full and Balanced sets; it beat SUBSETEARLY by avoiding the processing of redundant items unlikely to improve the end system. The filtering done to create the Balanced subsets does not generalize across learning tasks and must be repeated for each subsequent task using the dataset. In contrast, ZOMBIE's initialization is performed just once per dataset.

D. Varying System Parameters

We also tested ZOMBIE's configuration, testing several bandit rewards, several index grouping methods, and ZOMBIE's robustness in the face of index groups of varying quality.

1) Bandit Reward Methods: We evaluated ZOMBIE using the reward functions described in Section IV-C. For classification tasks, we tested CLASSIFYERROR, CLASSBALANCE, and UNCERTAINTY. For **Regression**, we tested INFOTRACE. Finally, we tried a baseline ROUNDROBIN mechanism that just cycled through the list of index groups, choosing one element per group on each pass.

Figure 8 shows the impact of choosing different reward functions for our bandit model on the classification tasks. All of our three proposed reward functions (UNCERTAINTY, CLASSBALANCE, CLASSIFYERROR), had at least a 2x speedup over EARLY for all of the document classification tasks. It may seem surprising that the baseline ROUNDROBIN method fared reasonably well on several tasks: in these cases, just the index grouping alone is enough to increase the likelihood of processing an example of one of the minority classes. Perhaps not surprisingly, UNCERTAINTY either performed the best or tied for best method in all cases and is our recommended reward function for classification tasks. CLASSIFYERROR is likely to learn incorrectly from bad past decisions. CLASSBALANCE cannot avoid candidate inputs that might have been useful in the past but are no longer useful to the learner.

For the **Regression** task, INFOTRACE outperformed EARLY, showing a speedup of 1.7x. ROUNDROBIN performed slightly worse than EARLY, with a 0.9 speedup. Our **Regression** results were not as good as those for the other tasks: overhead is higher from retraining the linear regression learner, as well as from the expensive INFOTRACE. Also, active learning for regression is relatively understudied, and the field's best techniques are unsuitable due to high computational costs.

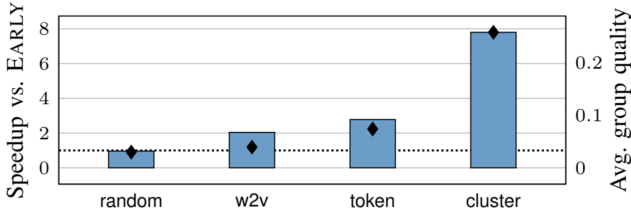


Fig. 9: ZOMBIE's performance with our index grouping methods, on the **DC6** task, shown as the speedup compared to EARLY (dotted line). The diamonds show each method's group quality.

2) Index Grouping Methods: Figure 9 shows ZOMBIE's speedup over EARLY using the UNCERTAINTY reward on the **DC6** task for four index group construction methods: cluster (our recommended method) and, for comparison, random, token, and w2v. We found similar results for the other tasks, but omit them due to space constraints. The figure's left axis describes the bars, showing the speedup over EARLY. The right axis describes the black diamonds, showing the mean *group quality* for each method, defined as an index group's ratio of minority class items (our proxy for useful items), weighted by its fraction of all the minority items. The random index performs poorly. For all others, ZOMBIE yields a speedup over EARLY.

Unsurprisingly, ZOMBIE's speedup was roughly correlated with the quality of the grouping. In the case of random, there were simply no high-quality index groups for our bandit model to find and exploit. The results show that ZOMBIE can yield a very large speedup when the group quality is middling and can even yield some speedup when the group quality is quite poor (as with w2v and token). These results give us increased confidence in our recommendation of ZOMBIE's use of standard, task-independent clustering (e.g., our k-means cluster method).

The time needed to compute these indexes for our Wikipedia dataset was non-trivial but manageable, ranging from 30 to 60 minutes for token and cluster and several hours for w2v. Because these indexes are designed for use with many different learning tasks, their construction costs (as when building traditional database indexes) can be amortized over their long life.

3) Effects of Index Group Quality: To better understand the impact of index group quality on ZOMBIE's performance independent of any particular grouping method, we created a range of synthetic indexes with specific quality measures. We fixed the raw dataset while distributing the items among index groups to vary two parameters. *Useful group density* measures the fraction of index groups containing at least one minority class item (again, our useful items proxy). *Group quality* measures the fraction of useful items in each useful index group, as defined previously. That is, the former is how likely a random input will be useful; the latter is how likely a random item from a useful index group will itself be useful.

Figure 10 shows ZOMBIE's performance using index groupings with parameter settings ranging from 0.1 to 0.9, averaged over 10 runs, for **DC2** and **DC6**. Each square's color represents ZOMBIE's speedup using the grouping with the corresponding parameters, compared to EARLY. Red regions show where ZOMBIE underperformed EARLY, while green regions show where ZOMBIE had a significant speedup. In the lower left corner are truly bad (and likely unrealistic) groupings: most index groups contain no useful items, and in those that do, useful items are very rare. Here, ZOMBIE has a very difficult job finding any useful items. The upper right represents near perfect (and, again, unrealistic) groupings: most index groups contain high concentrations of useful items. Here, our bandit method can quickly start exploiting the good groups. Between these extremes is a large area of the parameter space showing a positive speedup. *Even with low-quality index groups,* ZOMBIE *is successful at speeding up the feature engineering loop.*

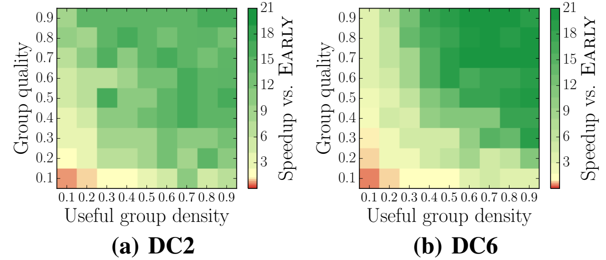


Fig. 10: ZOMBIE's speedup over EARLY for a range of different index group quality levels on the (a) **DC2** and (b) **DC6** tasks. Green and yellow areas indicate speedups over EARLY. All but the very lowest-quality groupings (lower left, in red) yield a speedup.

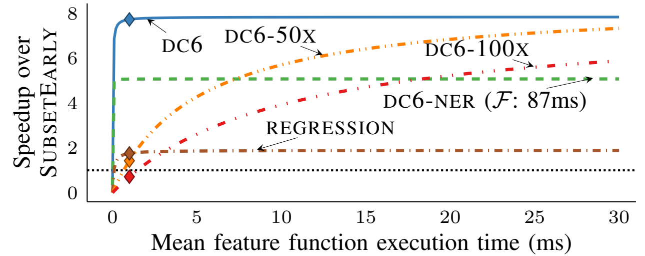


Fig. 11: Speedup for several tasks, varying the feature function execution time. Diamonds on lines show the task's actual $\mathcal{F}$ execution time. The black dotted line shows SUBSETEARLY.

We expect standard clustering methods to fall within ZOMBIE's effective range. As a point of comparison, our k-means cluster method exhibits density 0.55 and group quality 0.11 on **DC2** and density 0.86 and group quality 0.24 on **DC6**.

E. Varying Task Parameters

We performed two types of experiments to test properties of the learning task, first exploring the impact of the feature function costs. We then tested how the difficulty of finding useful inputs in the dataset affects ZOMBIE's performance.

1) Feature Function Costs: ZOMBIE's performance gains are tied to the computational difficulty of $\mathcal{F}$ and T. Reducing the number of function invocations is more meaningful when $\mathcal{F}$ is expensive. However, ZOMBIE incurs overhead from T and from the input selection loop. If $\mathcal{F}$ is fast enough, this overhead will swamp any gains from avoiding calls to $\mathcal{F}$.

Figure 11 shows the results of our experiment to discover how slow a feature function has to be before it can benefit from using ZOMBIE. We compared five tasks to SUBSETEARLY, chosen for a baseline because it only runs T after the subset has been created and so does not incur additional retraining runtime costs; these results show the worst-case effects of ZOMBIE's overhead. The y-axis shows ZOMBIE's speedup over SUBSETEARLY. For the x-axis, we simulated different average $\mathcal{F}$ runtimes. We kept the other aspects of the tasks—the number of inputs processed, the outputs from $\mathcal{F}$, and the observed learner accuracy—exactly the same.

A line on this graph shows ZOMBIE's speedup ratio for a task, if a single invocation of the user's $\mathcal{F}$ took the time indicated on the x-axis. The diamonds show the actual $\mathcal{F}$ average runtimes. As a line increases, the effect of ZOMBIE's overhead (largely from retraining) decreases. When a line is "flat," the overhead is negligible compared to $\mathcal{F}$'s invocation time. For longer $\mathcal{F}$ times, ZOMBIE's ability to avoid function invocation yields a larger speedup. A line's height is task-specific, showing how fast we reach the task's stopping point.

For three of the tasks, ZOMBIE yields a speedup larger than 1x except when $\mathcal{F}$ is extremely fast. Even with the observed average runtime of 1 ms in the case of **DC6** and **Regression**, ZOMBIE is useful. For **DC6-NER**'s 87 ms runtime, ZOMBIE provides a healthy speedup. For **DC6** and **DC6-NER**, retraining is incremental and the training costs are negligible. **Regression**'s retraining costs are about five times higher.

We also introduced two synthetic tasks: **DC6-50X** and **DC6-100X**. These are identical to the **DC6** task, except with artificially-inflated training times of 50 and 100 times **DC6**'s average training time, simulating tasks with an extremely computationally intensive training procedure T. The gentler curves of **DC6-50X** and **DC6-100X** show that when T is extremely time-consuming, $\mathcal{F}$ must also be more time-consuming for ZOMBIE to yield a substantial speedup. In some cases, a very fast $\mathcal{F}$ might mean that ZOMBIE does not yield a speedup over EARLY. Indeed, with the 1 ms $\mathcal{F}$ we observed for **DC6**, the **DC6-100X** task would be slower than SUBSETEARLY. However, ZOMBIE yields a real speedup even when running with a training procedure *50x slower than our real system*. And when training is 100x slower, even a tiny increase in feature function invocation time means ZOMBIE is worth running.

2) Difficulty of Finding Good Inputs: We varied the relative rarity of the labeled classes in the document classification task in order to test how ZOMBIE responds when it is difficult to find good items. As minority items become more rare, we expect the input selection task to become more difficult, as it is harder to find the minority-class examples needed to train a high-quality learner. EARLY should also suffer in this situation.

Figure 12 shows that ZOMBIE does well for minority class sizes that range from 0.01% to 5% of the corpus for the **DC6** task. For the timing plot on the left, error bars indicate a 95% confidence interval. Results for our other tasks were consistent with these results but are omitted due to space constraints. For the most difficult case, only 100 examples of each minority class were present in the 1M document test corpus. This extreme ratio would not be surprising in many settings: for example, a task that predicts e-commerce prices might use a feature that exploits rare pages that list prices but is useless on news or social media pages. While ZOMBIE's speedup ratio over EARLY was relatively low in the 0.01% case, the engineer's wait time was reduced from almost 10 minutes to 5 minutes, which could have a major impact on workflow. When the feature function execution time is much larger, absolute savings can be much larger: ZOMBIE saves 182 minutes, or *over 3 hours*, on **DC6-NER** at 0.01% rarity, reducing the evaluation loop from 8 to 5 hours. This would allow an engineer to perform two feature iterations in a workday instead of just one.

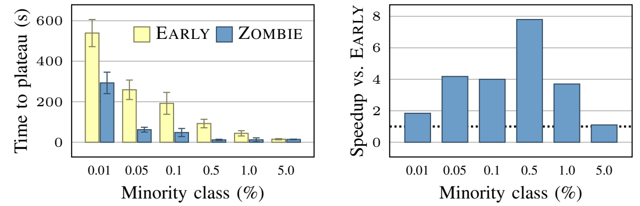


Fig. 12: Left: Time needed to reach the accuracy plateau for both EARLY and ZOMBIE on the **DC6** task. Error bars show the 95% confidence bound for mean time over 10 runs. Right: Speedup over EARLY for the same trials. EARLY is shown by the dotted line.

VI. RELATED WORK

Database researchers have begun to propose frameworks that support feature selection and engineering [1], [13], [14]. Our system's goal—accelerating the feature engineering development cycle—has grown from the vision sketched in Anderson *et al.* [1]. We have demonstrated a user-facing tool [9] built on the techniques detailed in this paper. Zhang *et al.*'s work [13] is more suited for the "generate-and-select" feature generation approach discussed in Section II-E. MLBase [14] focuses on the learning pipeline and does not specifically address feature engineering; ZOMBIE may be complementary to its methods.

Our system draws intellectually on several other areas of data management. Large-scale distributed data processing has seen intensive research for at least a decade [6], [7], [41]–[45]. MapReduce [6] was the first in the modern wave of systems, but its simple scan-and-process model has been largely eclipsed by systems that use familiar database techniques: indexes, high-level query languages, query optimization, etc. Few, though, optimize execution of user-defined code, let alone opaque feature code in a learning task, as we do with ZOMBIE.

ZOMBIE's two-phase operation is similar to approximate query processing systems; samples of the data with specific statistical properties are pre-computed and then used to answer an approximate query [22]–[24]. Our system also answers queries using pre-computed data subsets. Unlike approximate query processors, it is unclear what useful statistical properties could be pre-calculated because of the users changing feature code; feature evaluation is more suited to an online approach.

Active learning is a well-known topic within machine learning [21]. Our work shares the main goal of active learning: minimizing the cost of constructing a training set through careful selection of training examples. We differ from typical active learning in that we cannot examine the features of potential training examples to guide our selection, which would require unwanted feature code runtime costs. The most related line of active learning research is *active feature-value acquisition*, which attempts to avoid very expensive features, like medical tests, by estimating the utility of every object in the raw dataset [46]. This assumes a dramatic cost split between features that are nearly free and features that are so expensive that it is worth paying almost any computational cost to avoid them. While useful, this does not apply in our setting: our work assumes functions with comparable runtimes, so total runtime is best reduced by processing less raw data.

Deep learning has become a hot area of research, promising high-accuracy models that do not require traditional explicit feature engineering [47], [48]. Deep learning methods may somewhat displace human feature engineers in the future,

though we believe there will always be a strong role for human-provided domain knowledge. Deep learning, despite its many successes, faces several challenges. Its applicability to text is still unclear, and it is extremely computationally intensive. Moreover, its operation is notoriously opaque: engineers may face trouble maintaining and debugging these systems. In any case, our system can be used alongside such approaches.

VII. Conclusions and Future Work

We have described the ZOMBIE input selection system. For the critical feature engineering evaluation loop, ZOMBIE obtains speedups of up to 8x, and reduces wait from 8 to 5 hours in some cases. It is a promising tool in the effort to accelerate the feature engineering iteration cycle. We view ZOMBIE as just one component of an integrated feature engineering system. We would like to improve ZOMBIE by incorporating arm statistics across multiple iterations of the evaluation loop, and by giving the engineer explicit feature design hints. We will also explore other applications for our bandit method, such as automatically choosing among data sources of varying quality.

Acknowledgments

This project is supported by NSF grants 0903629, 1054913, and 1064606, as well as by gifts from Yahoo! and Google.

References

[1] M. Anderson *et al.*, “Brainwash: A data system for feature engineering,” in *CIDR*, 2013.

[2] P. Domingos, “A few useful things to know about machine learning,” *Communications of the ACM*, vol. 55, no. 10, p. 78, 2012.

[3] S. Levy, “How Google’s Algorithm Rules the Web,” *Wired*, 2010. [Online]. Available: http://www.wired.com/2010/02/ff_google_algorithm/

[4] E. V. Buskirk, “How the Netflix Prize Was Won,” *Wired*, 2009. [Online]. Available: http://wired.com/2009/09/how-the-netflix-prize-was-won/

[5] D. Ferrucci, “An Overview of the DeepQA Project,” *AI Magazine*, 2012.

[6] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in *OSDI*, 2004.

[7] M. Zaharia *et al.*, “Spark: Cluster computing with working sets,” in *HotCloud*, 2010.

[8] D. Thain, T. Tannenbaum, and M. Livny, “Distributed computing in practice: The Condor experience.” *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, 2005.

[9] M. R. Anderson *et al.*, “An integrated development environment for faster feature engineering,” *PVLDB*, vol. 7, no. 13, pp. 1657–1660, 2014.

[10] D. Antenucci *et al.*, “Ringtail: Feature selection for easier nowcasting.” in *WebDB*, 2013.

[11] A. Kumar, F. Niu, and C. Ré, “Hazy: Making it easier to build and maintain big-data analytics,” *Communications of the ACM*, vol. 56, no. 3, pp. 40–49, 2013.

[12] C. Ré *et al.*, “Feature engineering for knowledge base construction,” *IEEE Data Eng. Bulletin*, vol. 37, no. 3, 2014.

[13] C. Zhang, A. Kumar, and C. Ré, “Materialization optimizations for feature selection workloads,” in *SIGMOD*, 2014.

[14] T. Kraska *et al.*, “MLbase: A distributed machine-learning system,” in *CIDR*, 2013.

[15] P. G. Ipeirotis *et al.*, “To search or to crawl?: Towards a query optimizer for text-centric tasks,” in *SIGMOD*, 2006.

[16] M. Mintz *et al.*, “Distant supervision for relation extraction without labeled data,” in *ACL-IJCNLP*, 2009.

[17] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2011.

[18] J. M. Hellerstein, P. J. Haas, and H. J. Wang, “Online aggregation,” *ACM SIGMOD Record*, vol. 26, no. 2, pp. 171–182, 1997.

[19] A. Vlachos, “A stopping criterion for active learning,” *Computer Speech & Language*, vol. 22, no. 3, pp. 295–312, 2008.

[20] J. Zhu *et al.*, “Confidence-based stopping criteria for active learning for data annotation,” *Transactions on Speech and Language Processing*, vol. 6, no. 3, p. 3, 2010.

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

[22] M. N. Garofalakis and P. B. Gibbons, “Approximate query processing: Taming the terabytes.” in *VLDB*, 2001.

[23] B. Babcock, S. Chaudhuri, and G. Das, “Dynamic sample selection for approximate query processing,” in *SIGMOD*, 2003.

[24] S. Agarwal *et al.*, “BlinkDB: Queries with bounded errors and bounded response times on very large data,” in *EuroSys*, 2013.

[25] R. Polikar *et al.*, “Learn++: An incremental learning algorithm for supervised neural networks,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 31, no. 4, pp. 497–508, 2001.

[26] P. Laskov *et al.*, “Incremental support vector learning: Analysis, implementation and applications,” *The Journal of Machine Learning Research*, vol. 7, pp. 1909–1936, 2006.

[27] P. E. Utgoff, “Incremental induction of decision trees,” *Machine learning*, vol. 4, no. 2, pp. 161–186, 1989.

[28] A. Strehl, J. Ghosh, and R. Mooney, “Impact of similarity measures on web-page clustering,” in *Workshop on AI for Web Search*, 2000.

[29] M. C. de Souto *et al.*, “Clustering cancer gene expression data: A comparative study,” *BMC Bioinformatics*, vol. 9, no. 1, 2008.

[30] L. Portnoy *et al.*, “Intrusion detection with unlabeled data using clustering,” in *Workshop on Data Mining Applied to Security*, 2001.

[31] S. Bubeck and N. Cesa-Bianchi, “Regret analysis of stochastic and nonstochastic multi-armed bandit problems,” *Machine Learning*, vol. 5, no. 1, pp. 1–122, 2012.

[32] P. Auer, N. Cesa-Bianchi, and P. Fischer, “Finite-time analysis of the multiarmed bandit problem,” *Machine Learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[33] H. He and E. A. Garcia, “Learning from imbalanced data,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 9, pp. 1263–1284, 2009.

[34] D. D. Lewis and J. Catlett, “Heterogenous uncertainty sampling for supervised learning.” in *ICML*, 1994.

[35] R. E. Schapire, “The boosting approach to machine learning: An overview,” in *Nonlinear estimation and classification*. Springer, 2003, pp. 149–171.

[36] S. Rogers and M. Girolami, *A first course in machine learning*. Chapman & Hall/CRC, 2011.

[37] M. Hall *et al.*, “The WEKA data mining software: An update,” *SIGKDD Explorations*, vol. 11, no. 1, pp. 10–18, 2009.

[38] J. R. Finkel *et al.*, “Incorporating non-local information into information extraction systems by Gibbs sampling,” in *ACL*, 2005.

[39] F. Pedregosa *et al.*, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[40] “Google word2vec,” https://code.google.com/p/word2vec/.

[41] S. Melnik *et al.*, “Dremel: Interactive analysis of web-scale datasets,” *PVLDB*, vol. 3, no. 1-2, pp. 330–339, 2010.

[42] G. Malewicz *et al.*, “Pregel: A system for large-scale graph processing,” in *SIGMOD*, 2010.

[43] R. Kallman *et al.*, “H-store: A high-performance, distributed main memory transaction processing system,” *PVLDB*, vol. 1, no. 2, pp. 1496–1499, 2008.

[44] H. Lim, H. Herodotou, and S. Babu, “Stubby: A transformation-based optimizer for MapReduce workflows,” *PVLDB*, vol. 5, no. 11, pp. 1196–1207, 2012.

[45] “Cloudera Impala,” https://github.com/cloudera/impala.

[46] M. Saar-Tsechansky, P. Melville, and F. Provost, “Active Feature-Value Acquisition,” *Management Science*, vol. 55, no. 4, pp. 664–684, 2009.

[47] A. Coates, A. Y. Ng, and H. Lee, “An analysis of single-layer networks in unsupervised feature learning,” in *AISTATS*, 2011.

[48] Q. V. Le *et al.*, “Building high-level features using large scale unsupervised learning,” in *ICML*, 2012.