It's Data All the Way Down: Exploring the Relationship Between Machine Learning and Data Management

by

Michael R. Anderson

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2019

Michael R. Anderson

mrander@umich.edu

ORCID iD: 0000-0002-0959-4234

To my wife Carol and our wonderful children, Katie and Nicholas.

Acknowledgements

I first off would like to thank my advisor, Mike Cafarella, whose guidance in how to approach research has truly been one of the key takeaways I have from my Ph.D. adventure. Learning how to break down a problem into parts that can be reasoned about and how to effectively and convincingly explain experimental results are things that are probably more important than any particular technical aspect of my research. Mike was also very supportive and understanding of my situation as a non-standard graduate student; being old and married with kids, as I am, presents certain constraints to graduate student life that a student fresh out of an undergraduate degree may not face. Being able to put family first at times and being able to maintain a reasonable work-life balance certainly improved my ability to produce quality (if not entirely speedy) research.

My fellow University of Michigan database students also deserve much thanks. Dolan Antenucci was my collaborator on several projects and on the golf course, and his presence was certainly missed when he graduated a year or two ahead of me. Mark (Zhongjun) Jin was also a great collaborator, full of insight, and one of the nicest lab-mates one could hope to meet. The database group is in good hands for the next couple of years, with Mark as one of the senior students heading things up. Yongjoo Park was always a great sounding board for ideas, especially on our nearly daily walks to Duderstadt for coffee. Dan Fabbri and Matt Burgess, too, were great influences to me as a new grad student.

Most of all, I'd like to thank my family. My wife, Carol, and our two children, Katie and Nicholas gave me loving support through the (many) years this Ph.D. has taken. They put up with me whenever I was particularly stressed out and made things work when I had to fly halfway around the world for a week on relatively short notice. More generally, it was always great to be able to switch out of grad student mode for a while to watch one of the kids' soccer games, go on a lunch date with my wife, or do any of the other little family things that many graduate students aren't lucky enough to take part in while they're still in school. (Also, explaining my research to a 3rd grader was great training in science communication.)

My parents, Bob and Kathy Anderson, were also instrumental in my success, always supporting my goals, academic and otherwise, in a loving and stable home. Through their example, I (and my brothers) learned the value of education and hard work. My grandparents, too, where always there for me. Grandma Norma—a model of a strong, self-sufficient person—passed away before I finished my studies, but would have been proud to see me graduate. Grandma Phyllis and Grandpa Carl came to nearly every baseball game, band concert, and school function throughout my childhood, always cheering me and my brothers on. From all three, I learned that working hard and trying difficult things was something to be celebrated.

To everyone who has supported me throughout my life, I give a heartfelt thanks. I could not have made it here without you.

Table of Contents

List of Figures

List of Tables

Abstract

Data is central to machine learning: models are trained with data, trained models infer their predictions over input data, and the resulting inferences are themselves data. This being the case, there should be a natural relationship between machine learning and data management techniques. Much of machine learning research, perhaps understandably, focusses strictly on algorithmic improvements, chasing ever-increasing state-of-the-art accuracy measurements on their task of choice. Likewise, data management research has been slow to incorporate recent machine learning breakthroughs, like deep learning, to classic data management tasks. In this dissertation, we will demonstrate this relationship between machine learning and data management with a series of projects that improve aspects of machine learning through data management or improve data management with the addition of machine learning.

Specifically, we detail two systems that use database-style methods to improve run-time issues traditionally associated with machine learning and a third project that uses recent machine learning methods to solve data quality issues. Our system Zombie shows that novel data indexing methods can greatly reduce the time needed to evaluate the effectiveness of feature engineering, thereby reducing the time needed to train accurate machine learning models. With our system Tahoma, we show that by using particular physical representations of the images used as input into convolutional neural network classifier cascades, content can be quickly extracted to support binary predicates used in a video analytics database. And our system Grover demonstrates that universal embeddings, like those used in computer vision or natural language processing, can be created for relational data, with both column and table embeddings used to improve the performance of data integration tasks.

Our work shows machine learning and data management go hand-in-hand, and taking a holistic view of both can lead to improvements in each field.

Chapter 1

Introduction

Machine learning is built on data. Models are trained on data, which itself is likely extracted, modified, or engineered from other data. The output of a machine learning model is also data, which—depending on the application—may be loaded into and queried by a database or used as input into some other machine learning model. Because of this centrality of data to the practice of machine learning, data management techniques should also be central to a successful machine learning system. Likewise, because machine learning methods are highly effective at extracting, transforming, predicting, or otherwise acting upon data, there are a number of areas in data management that would be well served by the incorporation of modern machine learning techniques.

One of the key insights that made modern relational data management systems (RDBMS) highly successful was that the logical data structures can be independent from the structure of the physical storage of the data. By removing the need concentrate on the efficiency of physical data access, users could focus on their true task of gleaning insights from their data. This separation allows RDBMS users to declaratively work with their data, leaving the physical management of it to the data management system. Ideally, an RDBMS can decide how to execute a query and take advantage of system-level data structures like indexes with no input from the user.

Recent advances in machine learning have brought with them the development of systems that extract content or knowledge from large corpora of non-relational, unstructured, or otherwise opaque data. A common task for a user of these system is to select data instances based on the outputs inferred by the machine learning system (for example, "What input documents are classified as 'sports'?" or "Which input images contain stop signs?"). While this could be directly compared to a traditional database query that selects tuples based on particular attributes, there is little execution optimization that can be done be in the machine learning case. The attributes must be inferred by the machine learning algorithm before they can be queried, and the system's handling of the data is typically tied directly to data storage decisions made by the user.

These machine learning content extraction systems are often ad hoc, built on top of general execution engines like MapReduce or Spark, or simply coded up from scratch in a language like Python. In these cases, the developer of the system often *is* the user of the system, due in part to the lack of standard of-the-shelf machine end-to-end learning products or because the machine learning field is progressing too fast for such products to be created. While there are currently numerous popular machine learning frameworks and libraries, including TensorFlow [6], Keras [27], PyTorch [112], and scikit.learn [114], to develop and deploy a successful machine learning solution using one of these frameworks still requires much development and system configuration.

Further, these systems focus (perhaps rightfully) on the machine learning aspect of the system's pipeline: training, testing, and inference for a variety of popular algorithms. Data management—loading and transforming inputs, as well as saving outputs—is limited to loading and saving files directly to and from local or networked file systems. Analysis or queries of the system's outputs is performed by analyzing the entire output set using a library like Pandas [101] or loading those results into a standard RDBMS.

In database terms, physical and logical separation of query execution does not exist in machine learning systems. Physical access of the raw data is limited by how the user or system administrator loaded the data into the system: for example, images are read from a certain directory, text files are read in the order they are retrieved from an Amazon Web Services S3 bucket, or sensor readings are read from a series of log files. These inputs are usually processed in the order in which they are accessed (the aforementioned frameworks generally offer a method to shuffle inputs randomly, but this is generally used during training, not inference). The data is not indexed—as it might be in an RDBMS—to allow a query to be executed over the raw data that is *relevant to the query*. Indeed, because the query operates over the output of a (frequently inscrutable) machine learning algorithm, deciding which raw inputs are relevant to a given query is a challenge. Thus, the execution plan for machine learning-based queries is typically all-or-nothing, or at best, accelerated by computing an approximation of an answer on a random sample.

In this dissertation, we show that database-inspired techniques and the facilities they enable—indexes, for example—*can* be created to significantly speed up content extraction and queries over raw text and image data. These ideas can be applied in several areas of a machine learning pipeline, from feature engineering that allows the training of better models to the final inference stage of image classification using deep convolutional neural networks. We also show that recent machine learning innovations can improve data management tasks, with a particular focus on data integration. Our work covers the three following approaches that highlight the relationship between machine learning and data management.

Indexing — In a traditional database system, indexes are used to avoid processing irrelevant data. To do the same for a machine learning system, indexes must be created on the raw input data, while the query is run on the machine learning system's output data. For all but the very simplest linear models, it is very difficult, if not impossible, to determine which raw inputs are relevant to a query over the outputs *without running the actual model on all of the data*, making a standard database-style index impossible to create. As introduced in Section 1.2.1, we show with our Zombie project that effective, general-purpose indexes over the raw data can be created, however, by clustering the raw data using an online learning method at query time to discover which clusters are relevant to a given query.

Alternate data representations — By taking the physical storage responsibilities away from the user, a traditional database system can represent data in ways that directly benefit the execution of queries. Various compression techniques are used, for example, in column-store databases to very effectively reduce the amount of data (and therefore time) needed to load and inspect database table columns [140]. These techniques work because the compression methods—storing counts of identical values, rather than each value individually—work directly on the data being queried. For a machine learning system, we again run into the problem where optimization must be done on the raw data, and it is not clear *a priori* what the correct representation of the input data should be to generate a suitable model. For example, a deep convolutional neural network for image classification will run faster with smaller input images, since there will be fewer connections between the input layer and subsequent layer. But, will the smaller inputs carry enough information to build an effective predictive model?

With our Tahoma project, we demonstrate that using alternate representations of raw input data can lead to significant speedups when building image classifiers by generating a large number of classifiers with variations in architecture and input representations. We show that these classifiers can be used in a huge number of cascades, from which a small set of Pareto optimal cascades can be discovered that allow for huge throughput gains. Further, and perhaps more importantly, we show that the costs of constructing and loading the inputs to the classifiers *must* be considered when determining the optimal architectural configurations of classifier cascades for different deployment scenarios. Our methods, introduced in more detail in Section 1.2.2, allow the construction of the alternate representations and particular cascade configurations to be constructed independently from a user's query over the output of the classifiers.

Embeddings for relational data — While our first two projects demonstrate how techniques inspired by databases can improve machine learning workflows, this final project demonstrates that recent machine learning advances can likewise improve performance on data management tasks. To successfully use a deep learning model, sparse input vectors, like those representing words in a large vocabulary, should be embedded in a dense vector space. In natural language processing (NLP) applications, this is often done using word embedding models like word2vec [104, 105]. These models essentially emit a dense vector in an embedding space that encodes some semblance of semantics for a given word, based on that word's context as used in a large training corpus. These embedding vectors have been shown to greatly improve the performance of many different NLP applications.

In our Grover project, we show how embeddings like these might be constructed for relational data: we create both column embeddings and table embeddings and show how each can be used in several classic data integration tasks.

1.1 Problem Setting

The data managment techniques described in this dissertation are applicable to a range of supervised machine learning systems, including those that perform classification, object detection, and regression. Our methods are not directly applicable for unsupervised machine learning tasks, such as clustering, or for generative systems like generative adversarial networks (GANs) [52]. More specifically, our work targets problems where the task can be formulated as a *machine learning query*.

1.1.1 Machine Learning Queries

A machine learning query is a task in which a trained supervised machine learning model is used to extract inferred content from a corpus of raw data such that a user can run a database-style query over that extracted data. For example, given a large corpus of video frames collected from a car's dashboard camera (dashcam), a user may query:

"Find 100 examples of images with stop signs."[1] In this case, the dashcam images would be processed by a trained image classifier to produce a tuple for each image containing a single Boolean value (or possibly a probabilistic value in the range $[0,1]$) indicating whether a given input contains a stop sign.

More generally, a machine learning query can be run over data produced by a trained supervised machine learning model that outputs data in a relational format, with a single tuple produced for a single raw data input that will allow a query with binary predicates to be answered. In cases where the tuple attributes themselves are not binary, the query predicates can be: "Find images where *stopSignProbability* > 0.8". This class of machine learning task includes several important workloads:

Text classification — Given a large corpus of documents, the task is to classify them into categories. These categories could be, for example, for document organization (e.g., 'sports', 'news', 'comedy', etc.), language identification (e.g., English, French, German, etc.), or sentiment analysis (e.g., 'positive', 'negative' or 'happy', 'sad', etc.) Each of the categories in the output would be a single attribute in the machine learning output tuple. Where our work is less applicable in the text domain is for tasks like machine translation, where there is not a defined tuple beyond a single attribute for the translated text. However, if a binary predicate could be written for the output text, our indexing methods would work. For example, a query like "Does the translated text contain 'Eiffel Tower'?" would benefit from our methods.

Image classification — Much recent work has been done in the computer vision category using deep learning to classify images. With the ImageNet benchmark [127], a host of highly successful deep networks like AlexNet [83], VGGNet [136], and ResNet [56] have been created that can classify images with very high accuracy. In the ImageNet context, these networks take as input a certain sized image and output a tuple with 1,000 attributes, one for each category in the ImageNet dataset, each describing the likelihood of the input image belonging to that particular category. These types of classifiers can clearly benefit from our work, as can other classifiers derived from their architectures.

Object detection — A related area of work is *object detection*, which aims to find particular class of objects in an image and return their location, perhaps as the coordinates of a bounding box, as the YOLO9000 system does [124]. The output for these systems are typically a set of tuples per image, one for each object detected describing its bounding box. This output could be transformed into a single relational tuple, such as:

[1]For clarity, we will often specify user queries in natural language, though in practice such queries may be written in a language like SQL. Query parsing is orthogonal to the work presented in this dissertation.

$< ObjectA_1\ bbox, \ldots, ObjectA_n\ bbox, ObjectB_1\ bbox, \ldots, ObjectB_n\ bbox, \ldots >$. While not as straightforward as typical image classification, this extra transformation step would allow for suitable binary predicates to be written for machine learning queries over object detection systems such that our work would be applicable.

In the most naïve of implementations, these output tuples for any of these tasks could be materialized over the entire corpus of raw data and loaded into an RDBMS, which could then efficiently handle the subsequent query processing. Unfortunately, extracting these tuples is very computationally expensive (perhaps 10ms per input when performing state-of-the-art object detection in images, for example [124]). Processing an entire corpus can quickly become infeasible as the number of inputs or number of individual classifiers or content extractors acting on each data instance grow.

Our main goal, then, is to accelerate the processing of a machine learning query: we want a user to find their answer in seconds instead of hours, or in hours instead of days.

1.1.2 Measures of Machine Learning Success

Machine learning researchers have traditionally concentrated more on improving accuracy of their techniques rather than improving the speed of machine learning applications. (Indeed, "performance" in machine learning papers almost always refers to some measure of accuracy, rather than system speed or throughput, as is usually does in database and systems research.) When speed in considered, the concern is typically with the time required to train a system, not the time needed to run a trained model—the *inference* time—on new inputs to the system. And of far less concern is the time needed to prepare raw data for input into the machine learning system, which may include image resizing or, for text, steps like stop-word removal and stemming. We reviewed 71 papers presented at the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2017) for their mention of any aspects of speed or throughput of their systems. While a majority of them reported training time, only 31% of the papers had any mention of the time required for inference. Fewer than 3% had any mention of data preparation time. (For better or worse, many of these papers highlight accuracy results over pre-prepared benchmark datasets, so real-world data preparation concerns are orthogonal to their goals of algorithmic improvements.)

The research presented in this dissertation, however, primarily focuses on improving the speed of answering machine learning queries. That is, we to machine learning inference faster, either for individual inputs or over the whole set of inputs processed to answer a particular query. The cost of training machine learning models is important, but it often can be considered a one-off, offline cost, similar to those incurred when initially setting up and configuring a typical RDBMS. We also consider the costs of handling the raw data: things like loading and transforming the data into suitable formats incur significant costs that limit overall system throughput when answering machine learning queries.

Our research on applying data management techniques to machine learning systems *does not* focus on improving the accuracy of machine learning models, however. Our optimizations and other techniques are designed to be algorithmically agnostic when it comes to machine learning models, such that improvements in machine learning algorithms can be incorporated into systems implementing our methods.

Likewise, our research on applying machine learning methods to data management tasks—data integration and data quality improvement, particularly—does not aim to directly develop novel algorithms for these tasks. We aim to show that by introducing ideas from the machine learning community—embedding data in a dense, contextual vector space—these data management tasks can be more effective.

1.2 Overview and Contributions

In this section, we will introduce the three different research systems that make up the body of work of this dissertation, as well as briefly discuss other project we have participated in that are tangential to our core work.

1.2.1 Zombie: Indexing Raw Data

Turning raw data into queryable tuples via machine learning has many slow steps: data transformation (or feature extraction), model training, and inference. Speeding up any of those steps can increase the throughput of a system, and each area are important areas of current research. Another way to improve system throughput, however, is to reduce the number of times each of those steps needs to be performed to answer a particular machine learning query. In Chapter 3, we show that because a large corpus of raw data is likely to contain a large number of items that are irrelevant to a particular query, performing intelligent *input selection* can drastically reduce the amount of data that actually needs to be processed to answer a query. Input selection is an online method that determines the utility of potential inputs as the query (and raw data) is processed.

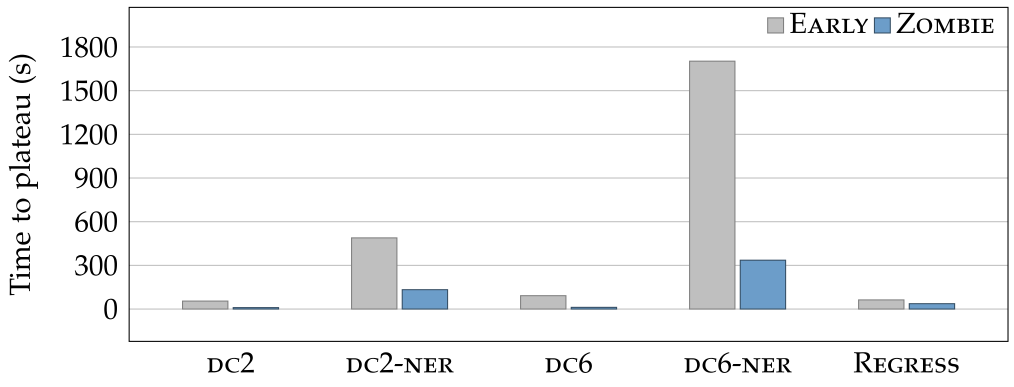


Figure 1.1: Time to answer a feature engineering query for Zombie, compared to a baseline which processes the raw data corpus sequentially and stops early when the query is answered. Each set of bars represents a different feature engineering task. See Chapter 3 for full details on these experiments.

Our system Zombie was designed to support the process of *feature engineering*, and our work presented in Chapter 3 shows how feature engineering and the evaluation of feature sets can be formulated as a query over a corpus of raw data. We build indexes over that data by clustering it on general features and use a multi-armed bandit technique to discover which clusters (called *index groups* in the Zombie system) are most likely to contain raw data inputs that will be useful to answering the feature evaluation query. We show that Zombie can answer these queries up to 8X faster than baseline techniques, as shown in Figure 1.1. While these results are specific to the feature engineering domain, we will show in with our Proposal project that Zombie-style index groups are useful for speeding up the answering of other types of machine learning queries.

1.2.2 Tahoma: Alternative Input Representations

With the proliferation of video capturing devices, from home security cameras to dashcams, being able to query large amounts of collected image data quickly has become increasingly important. Because data is collected quickly and constantly and user may repeatedly ask novel queries—a self-driving car research may wish to find historical examples of newly discovered failure modes, for example—queries must run quickly over raw data for which the desired content has yet to be extracted.

With our Tahoma system, we set out to directly speed up the content extraction—the inference step of the machine learning query processing. Using classifier cascades [25,72, 142,147], Tahoma replaces expensive state-of-the-art deep learning networks with combinations of very small binary classifiers to achieve nearly the same level of accuracy 98 times faster than a fine-tuned ResNet50 classifier [56]. When creating these cascades of

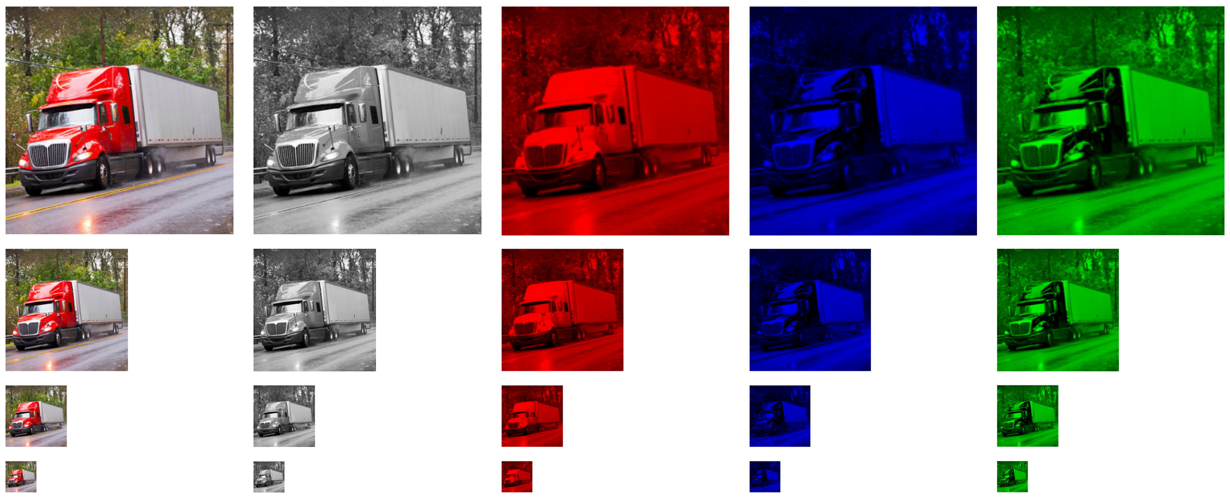


Figure 1.2: Examples of the variations in input representations created to increase the design space of TAHOMA's cascades. Tiny images with a single color channel lead to much faster classifiers, due to a much small number of input parameters. These tiny inputs may carry sufficient information to successfully answer some queries.

small classifiers, TAHOMA instantiates several hundred small convolutional neural network (CNN) classifiers, varying a number of architectural hyperparameters, such as number of layers and number of nodes per layer. Further, the format of the inputs to these networks is varied by changing image size and varying the color depth of the inputs, as shown in Figure 1.2. Adding these varied input representations dramatically increases the design space of possible cascades.

Creating cascades with up to three levels of these small classifiers results in over one million possible cascades. TAHOMA can quickly evaluate the accuracy and throughput of this large set of cascades, providing the user with a small set of Pareto optimal cascades. The user can decide which cascade matches the execution requirements (e.g., high throughput and low accuracy or low throughput and high accuracy) for a particular query. We also show that it is vitally important to include the costs of data handling in a particular system's deployment scenario, such as those for transformation and loading, when evaluating the relative throughput measurements of a set of classifiers. Basing throughput evaluation solely on inference costs (as is typically done) often results presenting the user with a set of "optimal" cascades that are not truly optimal in practice.

1.2.3 GROVER: Embeddings for Relational Data

The use of deep learning and other machine learning methods in the data management world has been quickly increasing in recent years. Index construction [78], query optimization [74,100], and data integration [38] are just some of the areas that have seen the introduction of these methods. One ancillary area that has not seen much focus is that of creating suitable representations of relational data such that it can be input into machine learning models while retaining much of it semantic and contextual information. NLP and computer vision systems often use embeddings, where a data instance is embedded into a dense vector space by a model that has been pre-trained on a large corpus of general data relevant to the machine learning task.

For example, many NLP tasks, like sentiment analysis or machine translation have used word embeddings like word2vec [104,105], GloVe [115], or fastText [20] to embed words into a vector space created by models trained on large text corpora. These models tend to preserve some sense of semantics by measuring how often words appear in context with other words. Using embeddings like these give can act as a head start when training a model, and can be considered a form a transfer learning, where information from the trained embedding model is transferred to the machine learning model being developed. Similarly, in computer vision, deep networks that have been pre-trained on large image classification tasks, like ImageNet [128], are used to embed a new task's images into the vector space learned by the pre-trained network. The pre-trained network has already learned how to extract many salient features from the images, and using these features in a new task leads to more successful models [83].

We take this idea and put it to use in our Grover system. Starting with fastText embeddings trained on a huge corpus of web tables, we can create embeddings for the contents of each cell in a table, and then aggregate those embeddings into column and table embeddings. We show that the column and table embeddings can then be used in a data integration tasks to improve the quality of data found in web tables.

Chapter 2

Background and Related Work

In this chapter, we briefly review work that demonstrates the relationship between machine learning and data management. Each subsequent chapter in this thesis includes a dedicated and detailed project-specific related work section. Here, though, we will first discuss databases and software systems designed with a machine learning focus. Following that, we will discuss how machine learning has been used in data management tasks.

2.1 Databases and Systems for Machine Learning

There has been a number of projects in recent years that integrate database and systems methods into building machine learning models. In this section, we will discuss three major areas where the database and systems communities have made major contributions to machine learning-related work.

2.1.1 Machine Learning Pipelines

MapReduce [34] was an early distributed processing system that became widely used as part of Apache Hadoop [1]. Though not a machine learning-centric system, MapReduce was used as a platform for systems like SystemML [49], which leveraged MapReduce's parallelism to accelerate machine learning tasks.

Apache Spark [3,154], while also not purposely developed to support machine learning, became an important part in many machine learning systems, especially for the distributed preprocessing of large amounts of data as might me found in an organization's data lake. SystemML has been updated to run on Spark [19], taking advantage of Spark's graph-based execution plans, lazy evaluation, and distributed in-memory caching to improve upon its previous MapReduce-based version. (SystemML has also become part of the Apache ecosystem [4].) MLBase [79] was developed as a system on top of Spark to train machine learning models at scale in a distributed fashion. Its MLlib [103] has become Apache Spark's distributed machine learning library.

KeystoneML [139] is a system for optimizing pipelines for data analytics. It provides a suite of data processing tools for text and images, as well as a library of statistical analysis tools, which are used to build pipelines for analytics tasks. Apache MADLib [2,58] was developed to provide in-database access to machine learning and statistical analytics tools, allowing machine learning operations to be run locally on data with an SQL-based language, without the need for external frameworks.

2.1.2 Feature Engineering and Machine Learning Development

While the previous section detailed some of the many general systems that have been developed to support machine learning tasks, there has also be much interest in improving specific parts of a machine learning workflow. For example, in this thesis, we present Zombie [10], a system designed to accelerate the iterative process of feature engineering [9]. Feature engineering is just a part of the overall workflow of creating successful machine learning models: other portions include data cleaning and preprocessing [35,81,134,156], data loading [135], training set creation [121], hyperparameter tuning [95], and model inference and evaluation [12,71,72].

Because it is difficult to predict the efficacy of each step of this machine learning development workflow without actually performing each (often expensive) step, a machine learning engineer must often perform many iterations over this workflow to determine suitable settings for each step. Ideally, this entire workflow would be optimized holistically, and recent work has been made in this direction. Helix [152] is one such system designed to optimize this entire workflow, using database-style techniques like materialization of intermediate steps that do not change between development iterations and treating this workflow as an optimizable directed acyclic graph.

2.1.3 Machine Learning in Production Environments

Beyond machine learning pipelines, the database community has worked on several projects related to model selection and managing machine learning models for use in production environments [84]. Velox [31], for example, is a system for serving and managing machine learning models, as well as dynamically updating weights and tuning parameters as new data arrives. Clipper [32] is a more recent model serving system that decouples the machine learning models from downstream applications to provide features such as caching, batching, and adaptive model selection.

Our work with the TAHOMA system [12], described in Chapter 4, is designed to speed up what is perhaps the core aspect of production machine learning systems: making classification inferences for novel data. Other systems have also recently taken different approaches to this problem: NoScope [72], for example, uses a classifier cascade approach to speed up image classification in the same vein as TAHOMA, while Willump [76] uses feature selection-based cascades and statistical approximation to speed up top-K queries over classification results.

Another important facet of real-world model use is being able to explain a trained model's inferences, which can be difficult when using deep learning methods, such as convolutional neural networks (CNNs). Krypton [107] frames this challenge in terms of a traditional database-style view maintenance problem to significantly speed up explainable CNN inference.

2.2 Machine Learning for Data Management Tasks

With the widespread popularity of deep learning research, its methods have become increasingly used in database and data management tasks. In this section we will discuss several key areas that have recently benefited from the introduction of these techniques. Our work with the GROVER system detailed in Chapter 5 employs embeddings and deep learning to data integration tasks.

2.2.1 Indexing

For example, learned indexes [78] have recently been demonstrated that use deep learning methods to develop index structures for databases that are more effective than traditional b-tree or hash index structures in many circumstances. These techniques are part of the indexing methods of SageDB [77], a database system that demonstrates that many core components of a traditional database can be replaced by "learned components" that are automatically configured via machine learning.

2.2.2 Query Optimization

Query optimization, too, has seen the introduction of deep learning methods [74, 109], where key parameters in query optimization algorithms, such a table cardinalities for joins have been estimated with higher accuracy than with traditional methods. SageDB [77], mentioned previously, also estimates cardinalities using trained models, improving join optimization.

Deep reinforcement learning methods have also been applied to optimizing join queries; the DQ optimizer [82] examines the plans of previous queries to learn heuristics that improve future searches through the space of possible join plans. ReJOIN [100] uses deep reinforcement learning to enumerate join orderings, as does the system designed by Ortiz et al. [109]

2.2.3 Data Integration

Data integration [38, 73] is perhaps a natural fit to deep learning methods, given text processing similarities it shares with some NLP tasks. Our Grover system uses deep learning models and embeddings to approach several data integration tasks. Grover is described in Chapter 5, along with detailed background information and related work on relevant data integration tasks.

2.2.4 Database Management

Recent work applying machine learning techniques to the management of database systems, with an ultimate goal of developing "self-driving" database systems []. OtterTune [145] uses machine learning models to tune the performance of database systems, using observed historical workloads to automatically configure key system settings, rivaling the performance of expensive human experts. QueryBot 5000 [98] is a framework that learns models to predict future query workloads. Query2Vec [67] uses learned vector representations of SQL queries to perform analytics over database workloads, to aid in tasks like index selection and prediction of query-related memory errors.

These systems and techniques are just a sample of the many methods being currently being researched. Machine learning clearly is a powerful tool that can and should be applied to wide and varied world of data management, and as the projects described above demonstrate, the potential impact on data management is likely to be huge.

Chapter 3

Input Selection for Fast Feature Engineering

3.1 Introduction

Many of today's most compelling software systems, such as Google's core search engine, Netflix's recommendation system, and IBM's Watson question answering system, are *trained systems* that 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 [8,37] and are a crucial bottleneck for many trained systems [22,44,92]. 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

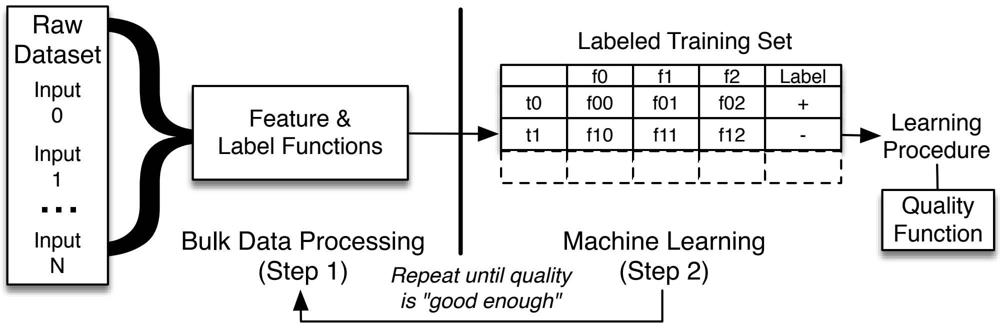


Figure 3.1: In the feature engineering evaluation loop, bulk data processing and machine learning steps are interdependent but to date have been commonly implemented as separate systems.

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*.

System Goal — Figure 3.1 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 [34], Spark [154], or Condor [143]. 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 [8,11,13,80,85,122, 155]. 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 3.1) 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 [66]. We have implemented this method in a prototype data processing system called ZOMBIE.[1]

Contributions and Outline — Our central contributions are:

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

- A proposed query model of the feature engineering workflow that captures current practices (Section 3.2).
- A system design and algorithms for optimizing *input selection* (Sections 3.3–3.4).
- 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 5.5).

We cover related work in Section 5.5.6. Finally, we conclude with a discussion of future work in Section 3.7.

3.2 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 3.1 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** (Figure 3.1), 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 $Q(M) \geq G$.*

[2]We assume labeling is relatively inexpensive; e.g., labels may be drawn from an existing database or provided by distant supervision techniques [106].

[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.

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 3.1: 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.

Algorithm 1 Feature Evaluation Loop

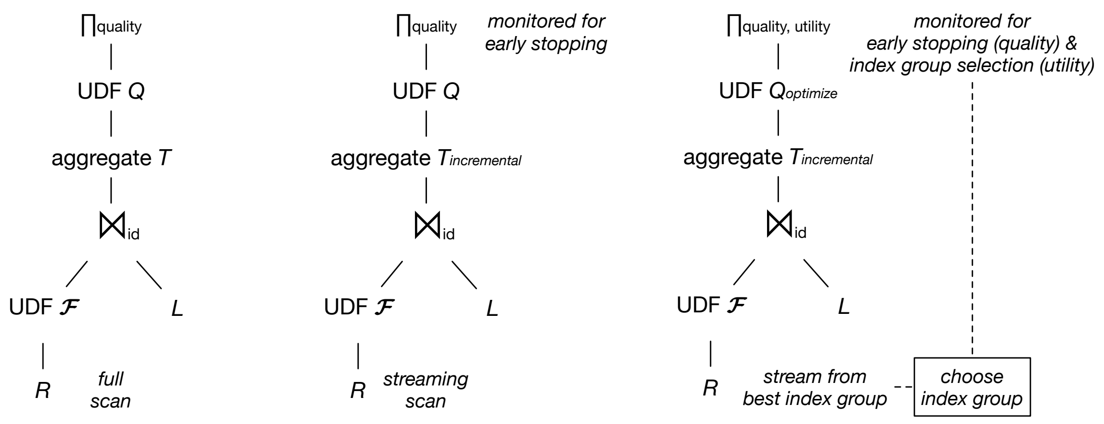
```
Input: Task $(R, L, T, Q, G)$
1: repeat
2:     User writes or modifies feature code $\mathcal{F}$
3:     query
4:         SELECT $Q(T(\mathcal{F}(R.\text{data}), L.label)$ AS $quality$
5:         FROM rawDataSet $R$, labels $L$
6:         WHERE $R$.id = $L$.id
7:     done
8: until $quality \geq G$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

3.2.1 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 3.2a. 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.

3.2.2 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



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

Figure 3.2: 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.

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.

[4]The textbook *Data Mining* does, however, describe an interactive procedure for feature selection that is roughly akin to SUBSET [151].

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).

3.2.3 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 [57]. 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 3.2b 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 3.3) 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 [148,160].

Figure 3.3 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 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

[5]Training overhead is a concern, but many learning algorithms can be trained incrementally. We discuss this further in Section 3.3.2.

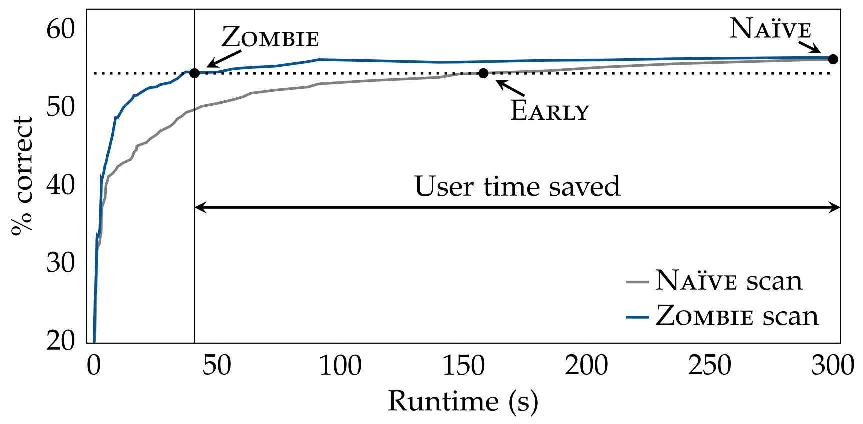


Figure 3.3: Learning curves for our execution plans. The NAÏVE plan performs a bulk scan over R, while EARLY stops the bulk scan early to use a subset R'. ZOMBIE scans the data from index groups and stops early much sooner, translating to a time savings for the user.

full accuracy; with early stopping, the trained model achieves nearly the full accuracy when R' is only half the size of the full R.

3.2.4 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 [133]. 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:     $best = \text{chooseBestExample}(M, examples)$
5:     $trainingSet.\text{append}([best, L(best)])$
6:     $M = T(trainingSet)$
7: until $|trainingSet| == B$
8: return $Q(M)$
```

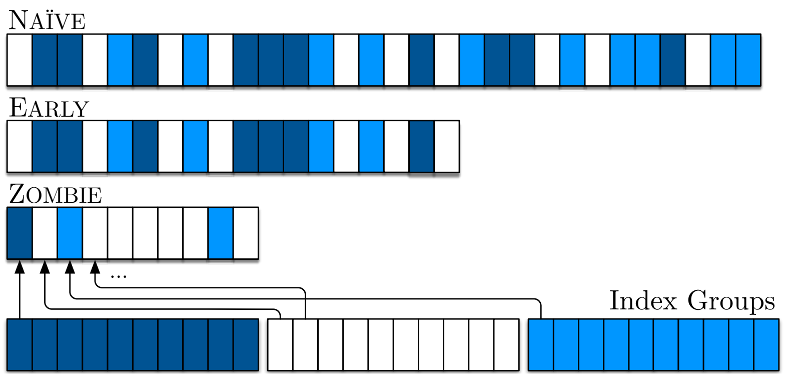


Figure 3.4: Illustration of the subset R' created by each execution plan. Colored rectangles represent raw data items. High-utility items are white; low-utility ones are dark. The Naïve plan processes R in its entirety. Early processes R', chosen by streaming R and stopping early. Zombie patches together R' from index groups.

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 [7, 16, 48], 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 3.3.2.

Figure 3.2c 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.

Figure 3.3 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 3.4 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.

3.2.5 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 3.5.5.) 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.

3.3 System Architecture

We can now describe our design for Zombie, depicted in Figure 3.5. 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*.

3.3.1 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 preprocessing 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

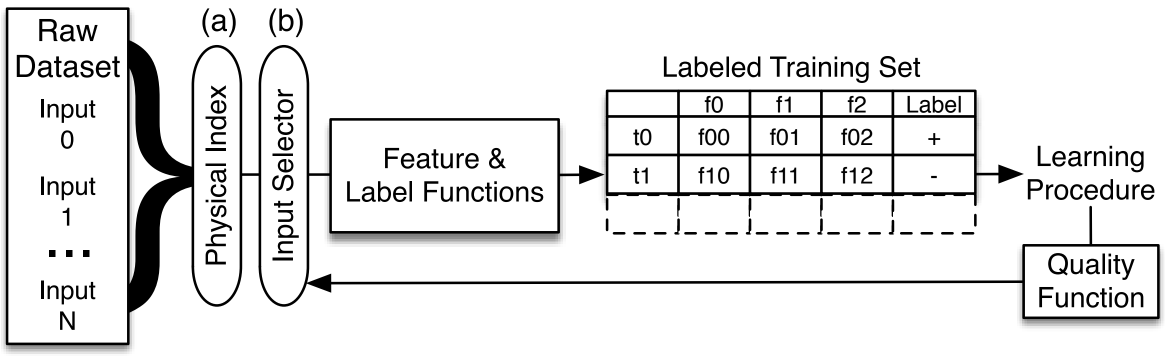


Figure 3.5: Zombie's basic architecture. The novel components are (a) the Physical Index and (b) the Input Selector.

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 3.5. It is driven by the basic framework from Figure 3.1 and the query model from Section 3.2 but includes two novel components. Zombie completely takes over the role of the bulk data processing system from Figure 3.1, 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.

3.3.2 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 [133].

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

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 $model$, $quality$
```

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 3.2.3).

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 3.4.

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 [119], SVMs [88], and decision trees [144]. In Section 5.5, we examine several tasks, including one without incremental retraining.

3.3.3 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 [141], cancer detection [33], and network security [120].

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 3.5.4, 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 3.5.1. 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.

3.3.4 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.

3.4 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 $\mathcal{I}$.

3.4.1 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* [21]. 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 [15]. 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.

3.4.2 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.

3.4.3 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 [55]. 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:

$$u_{\text{balance}}(r) = 1 - \frac{n_{L(r)} - n_{Lmin}}{n_{Lmax} - n_{Lmin}} \tag{3.1}$$

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 [93]. 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:

$$u_{\text{uncert}}(r) = 1 - (p(c_1|r) - p(c_2|r)) \tag{3.2}$$

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:

$$u_{\text{error}}(r) = \begin{cases} 1, & \text{if classifier error} \\ 0, & \text{otherwise} \end{cases} \tag{3.3}$$

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 [130].

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 [133]. 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:

$$u_{\text{info}}(r) = \beta\,(\text{trace}(I_{new}) - \text{trace}(I_{old})) \tag{3.4}$$

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 [126]. We chose $\beta = 0.2$ after testing a range of values $(0 < \beta \leq 1)$ in our experiments.

3.4.4 Selecting an Arm

Arms are selected in our system using the UCB1 algorithm of Auer *et al.* [15]:

$$\mathrm{UCB}_{a,t} = \mu_a + \alpha\sqrt{\frac{2\ln n}{n_a}} \tag{3.5}$$

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.

3.4.5 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 $\mathrm{UCB}_{a,t}$ 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 $\mathrm{UCB}_{a,t}$ value for the pulled arm using that input's previous reward, and then again select the key with highest UCB.

3.5 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 3.2

shows default settings for important system parameters, which were used except where stated otherwise.

We implemented Zombie and our machine learning tasks (using Weka [54]) as a Java application of about 17,500 lines of code. (Our prototype replaces the bulk data processing platform in Figure 3.1, 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.

3.5.1 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 3.5.5. 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 [45] 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

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

Table 3.2: Default experiment settings.

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 [114]. (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 [53, 104] 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.

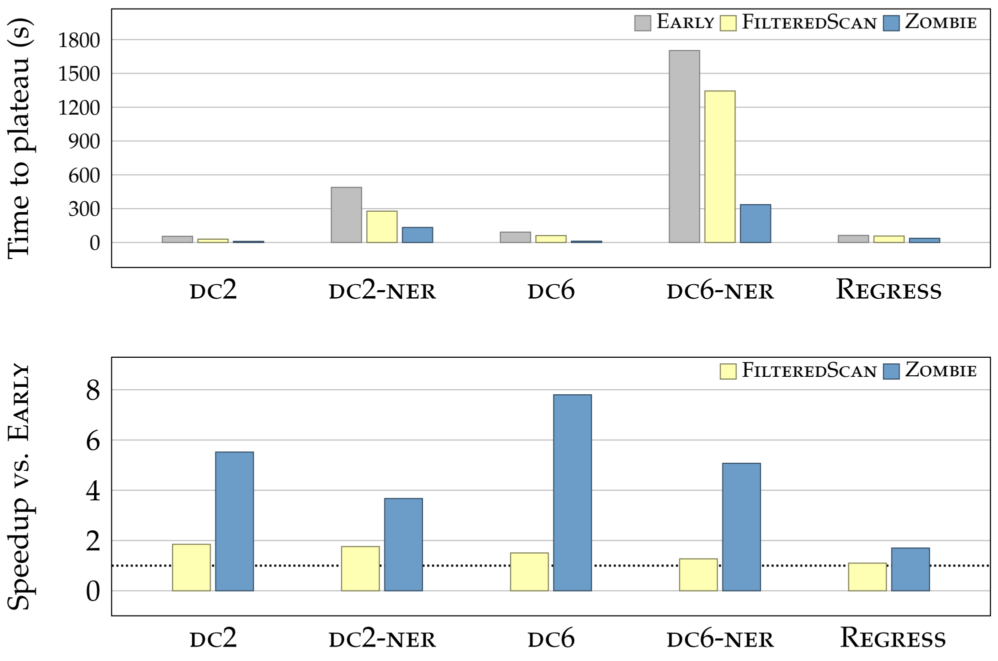


Figure 3.6: Time to stopping point, as well as Zombie's speedup over Early in reaching that point, for all of our experimental tasks. At bottom, the dotted line indicates Early's performance.

3.5.2 Overall Performance

We measured the speedup of all input selection mechanisms relative to the Early technique—the early stopping execution plan discussed in Section 3.2. 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.* [66]. 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.

Methodology — As we showed in Figure 3.3, 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 3.6. 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 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,

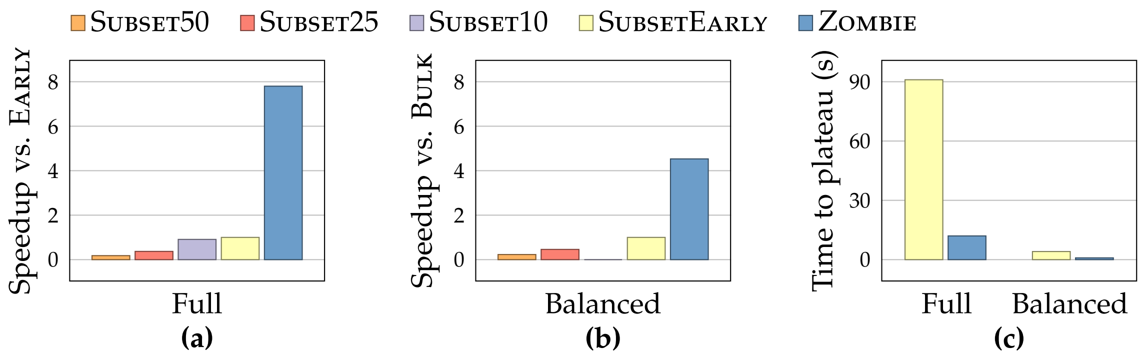


Figure 3.7: SUBSET sizes tested on **DC6** using (a) the full dataset and (b) a class-balanced filtered dataset. SUBSET10 was too small in (b) and had no result. Execution time for both is shown in (c).

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.

3.5.3 Testing SUBSET

The SUBSET method (Section 3.2.2) 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%).

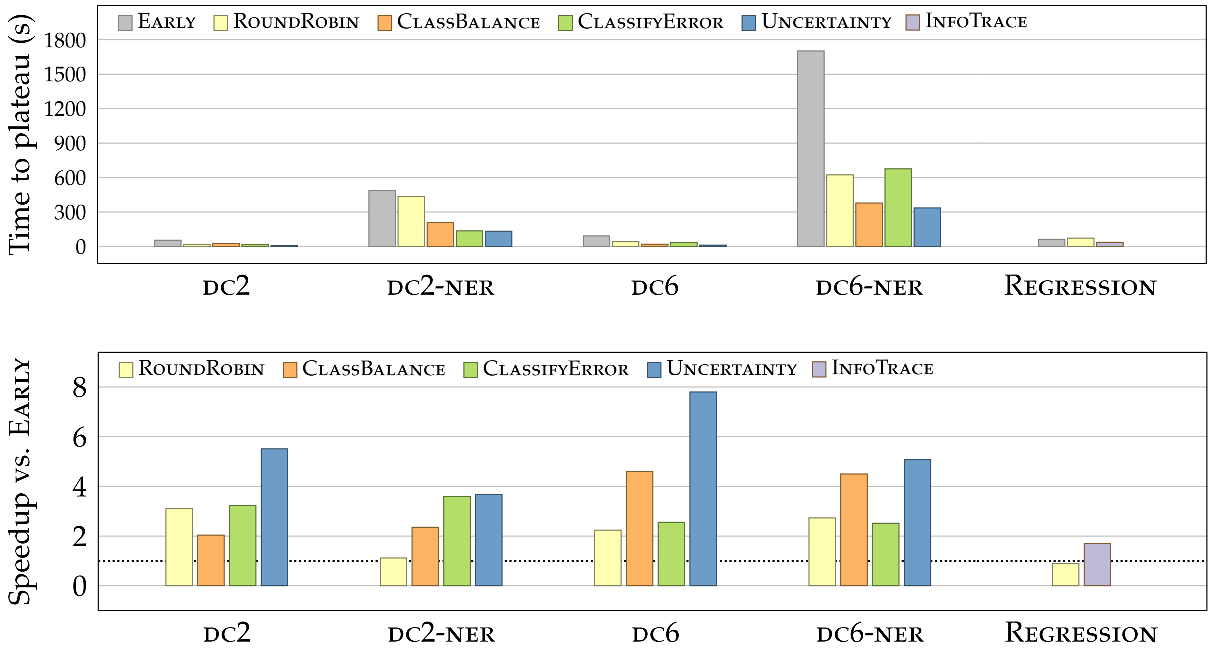


Figure 3.8: Time to stopping point, as well as Zombie's speedup over Early in reaching that point, for our different bandit reward methods. At bottom, the dotted line shows Early's performance.

Figure 3.7 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 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.

3.5.4 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.

Bandit Reward Methods

We evaluated Zombie using the reward functions described in Section 3.4.3. For classification tasks, we tested ClassifyError, ClassBalance, and Uncertainty. For **Regression**, we tested InfoTrace. Finally, we tried a baseline RoundRobin method that just cycled through the list of index groups, choosing one element per group on each pass.

Figure 3.8 shows the impact of choosing different reward functions for our bandit model on the classification tasks. Each of our three proposed reward functions (Uncertainty, ClassBalance, and 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.

Index Grouping Methods

Figure 3.9 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

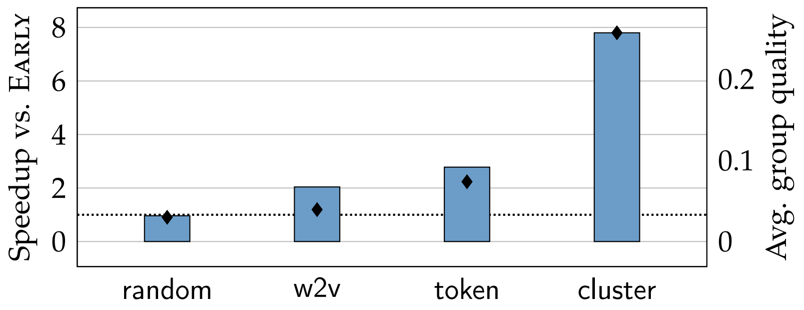


Figure 3.9: Zombie's performance with our index grouping methods, on the **DC6** task, shown as the speedup compared to Early (dotted line). The diamonds show each method's group quality.

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.

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 3.10 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

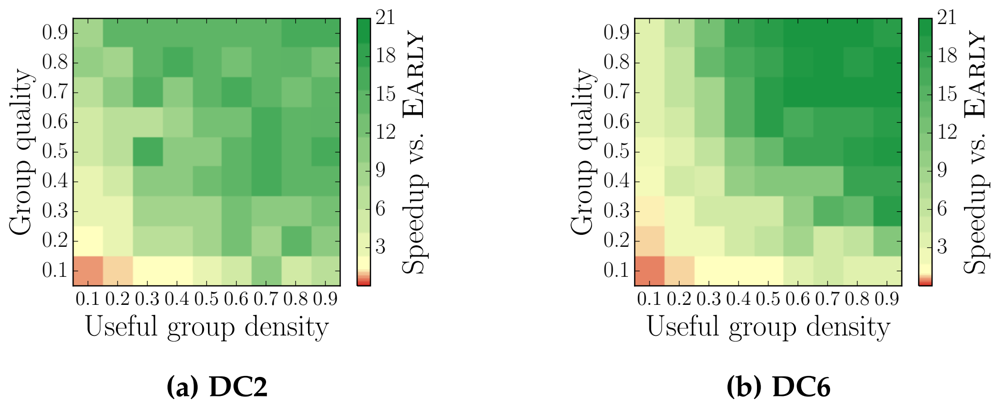


Figure 3.10: ZOMBIE's speedup over EARLY for a range of different index group quality levels on the (a) **DC2** and (b) **DC6** tasks. Green and yellow areas indicate speedups over EARLY. All but the very lowest-quality groupings (lower left, in red) yield a speedup.

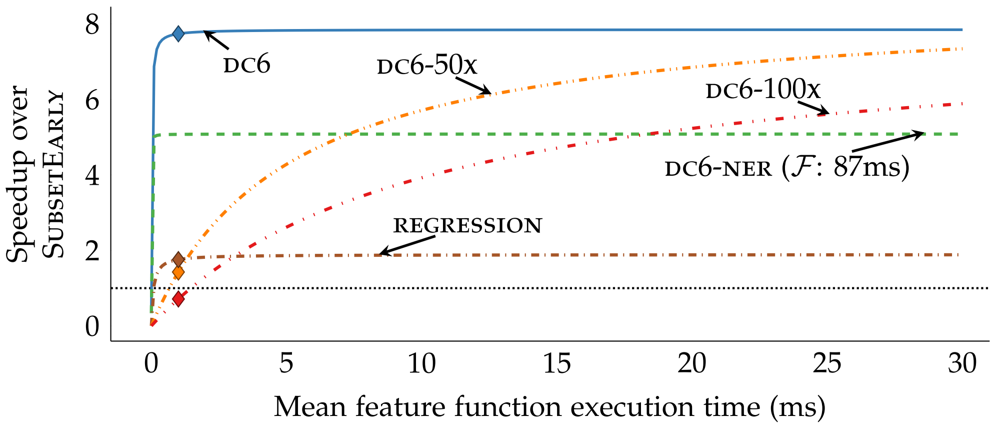


Figure 3.11: Speedup for several tasks, varying the feature function execution time. Diamonds on lines show the task's actual $\mathcal{F}$ execution time. The black dotted line shows SUBSETEARLY.

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.*

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**.

3.5.5 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.

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 3.11 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.

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 3.12 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.

3.6 Related Work

Database researchers have begun to propose frameworks that support feature selection and engineering [8, 80, 155]. Our system's goal—accelerating the feature engineering development cycle—has grown from the vision sketched in Anderson *et al.* [8]. We have demonstrated a user-facing tool [11] built on the techniques detailed in this paper. Zhang *et al.*'s work [155] is more suited for the "generate-and-select" feature generation approach discussed in Section 3.2.5. MLBase [80] focuses on the learning pipeline and

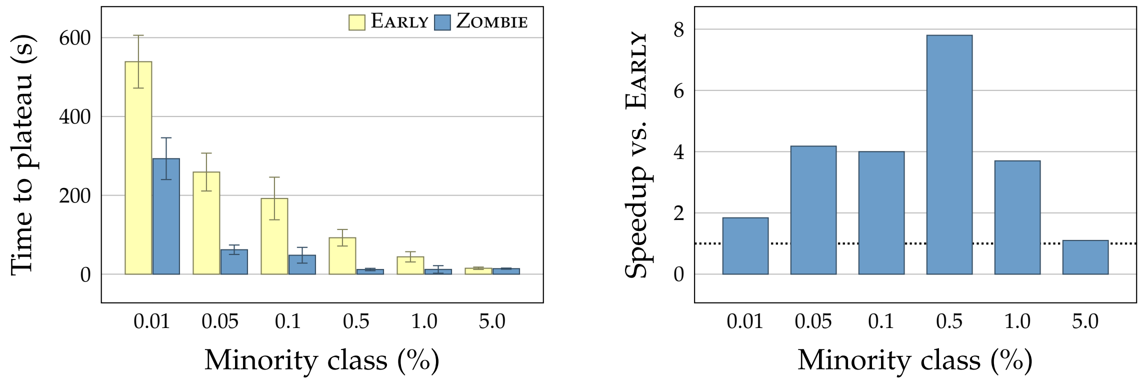


Figure 3.12: Left: Time needed to reach the accuracy plateau for both Early and Zombie on the **DC6** task. Error bars show the 95% confidence bound for mean time over 10 runs. Right: Speedup over Early for the same trials. Early is shown by the dotted line.

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 [28, 34, 70, 96, 99, 102, 154]. MapReduce [34] 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 [7, 16, 48]. 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 [133]. 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 [129]. 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 [29,89]. 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.

3.7 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.

Chapter 4

Physical Representation-based Predicate Optimization for a Visual Analytics Database

4.1 Introduction

Recent developments in computer vision have made feasible a long-term dream for the database community: a *visual analytics database*, which stores image data and answers user questions about its contents. For example, video frames from a city's traffic cameras could be used to count cars per minute. Or photos uploaded to photo storage web sites could be automatically sorted and tagged based on their contents. The sheer volume and diversity of data captured by cameras opens up myriad analytical query possibilities, if the content hidden behind opaque pixel values can be extracted at scale.

Deep convolutional neural networks (CNNs)—the family of methods used in modern computer vision systems—have enabled huge strides in image understanding in the last few years through tasks like image classification and object detection. For example, the classification error rate in the annual ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) [127] dropped from 25% to 18% in 2012 when a deep CNN was first used [83]. Recent results have lowered the error rate to 2% [62], rivaling or exceeding human performance.

Unfortunately, deep networks pose a considerable computational challenge when deployed in an analytical database system: a model's inference for a single image can require a lengthy series of large tensor multiplications. For example, YOLOv2, an object detection system designed for speed, requires 8.52 billion operations per single 416x416 pixel image, processing about 67 images per second on a modern GPU [124]. Since GPU hardware is far more expensive than most image sensors, data from multi-camera applications will soon outpace processing capabilities. Simply, to query huge amounts of image data, we need drastically lower processing costs.

Processing queries over a corpus of image data fits a more general *loop-and-test* pattern that is common to many machine learning tasks: the processor loops over the data, executing an expensive operator on each element to find those satisfying the task's constraints. In this case, content is extracted from each image by the expensive inference stage of a deep network to determine if the image satisfies a binary predicate specified in a user's query. While a loop-and-test process can be shortened by processing fewer items overall—using simple sampling or more sophisticated input selection [10]—we focus here on speeding up the *test* phase by reducing the per-image inference cost.

Recent work has reduced inference times for these types of deep learning systems (e.g., [59, 72]). However, we note that all of the visual data system optimizations to date suffer from a critical defect: they concentrate only on computation and ignore the inevitable data-handling costs, such as loading and transformation. Any query optimization method that focuses only on reducing computational load cannot exploit complex *data-centric tradeoffs* that weigh the amount of image data available, the classifier accuracy, and data handling costs.

As an example, consider an optimizer choosing between two image classification models (M_1 and M_2). M_1 accepts a 3-channel, full-color, 224x224 image as input while M_2 accepts a 1-channel, grayscale, 224x224 image. M_1 has fewer convolutional layers and, despite the larger input, requires fewer tensor operations than M_2, so its inference is faster. M_2 uses less rich data than M_1, but is nonetheless able to obtain comparable accuracy because of its additional convolutional layers. An optimizer that considers only the inference speed would choose M_1. However, a data system using M_2 might be faster, as its inputs load in one-third the time of those of M_1.

Such data-handling tradeoffs are important because visual analytics systems are likely to be architecturally diverse. Some systems may store multiple versions of the same image data (high-res vs. low-res). Others may employ different storage systems (local server vs. cloud vs. in-camera). In some architectures—say, one in which connectivity conditions change, or one in which GPUs are only intermittently available—the highest-payoff query plan may change by the moment. Ignoring data-centric tradeoffs can sacrifices substantial performance. In this work, we propose a framework for handling data-centric tradeoffs when optimizing visual analytical queries.

Technical challenge — We aim to optimize visual analytics queries that might be run in a range of diverse architectures and deployment settings. In this work, we ignore many standard relational query optimization issues, such as query plan rewriting or indexing. We focus exclusively on designing and choosing the CNN-based operator that implements an image-sensitive relational predicate. These operators can be chosen in a manner that trades system throughput against classification accuracy. Making that tradeoff of runtime vs. data quality is an application-specific decision we leave to the user. This paper offers a framework for identifying the best possible operator implementation, subject to a user's desired tradeoff.

Our approach — One method to speed up an expensive-but-accurate CNN is to replace it with cascades of fast, high precision (but low recall) image classifiers [25, 72, 142]. This is effective but focuses exclusively on computational efficiency. We start similarly, training a large number of specialized candidate binary-classification CNN models by varying not only CNN hyperparameters (as in prior work [72]), but also the representations of the inputs; for example, we build an n-layer CNN for large full-color inputs, another for small full-color inputs, more for grayscale inputs, and so on.

From these core candidate models, we then construct a massive number of classifier cascades. All of these cascades have different initially unknown runtime and accuracy characteristics. Our optimization method efficiently evaluates the cascades' accuracy using held-out data, and evaluates their runtime characteristics for the system's current deployment scenario. Finally, it identifies the Pareto-optimal cascades that satisfy the user's application-specific speed and accuracy constraints.

Organization — After formally describing our problem in Section 4.4 we discuss the following contributions:

- We propose a method for identifying high-quality image predicate implementations, by exploring CNN hyperparameters and varying input data representation (Section 4.5).
- We show the dramatic impact on runtime when a system is running in different deployment scenarios and is aware of the deployment-specific data handling costs (Section 4.6).
- We prototype our methods in a system called Tahoma and show that it provides up to a 35x speedup to classifier cascades through input data transformations. Tahoma also achieves up to a 98x speedup over the ResNet50 image classifier with no accuracy loss (Section 5.5).

We follow with a discussion of related work in Section 4.9.

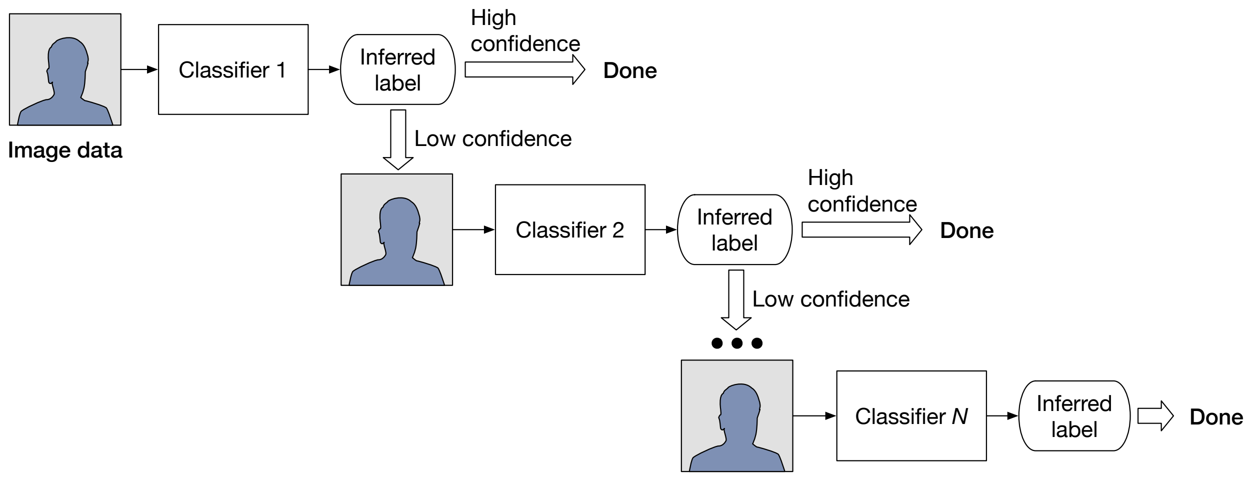


Figure 4.1: A multi-level classifier cascade. If the first classifier's output is uncertain, the input is classified by the second, and so on. If reached, the output of the final classifier in the cascade is accepted as the label.

4.2 Background

In this section, we will briefly give some background on cascade classifiers, since these methods are core building blocks of our work. Numerous systems have used cascades to speed up classifier inference. One of the first was the Viola-Jones face detector [147], which used a series of classifiers based on simple image features to detect faces in subsections of photographs; if any classifier had high confidence in its result, the result was immediately accepted, avoiding the use of further classifiers.

Figure 4.1 illustrates the general cascade process: an image is input into a classifier, whose output is accepted if it has high confidence. Otherwise, a second classifier is used, where again, a low-confidence result will send the input to a third, and so on. If the final level of the cascade is reached, its output is accepted. Ideally, the initial levels of a cascade are fast with high precision, though they may suffer from low recall. In face detection, for example, if most images have no human faces at all, the first classifier may quickly eliminate most cases. Only the few cases containing a face will be processed by the remainder of the cascade, at correspondingly higher cost.

Recently, cascades have been used to accelerate the relatively slow inference speeds of deep neural networks [25, 72, 142]. Our system takes these techniques further by exploiting the representations of the inputs and finding optimal cascades for a user's deployment scenario. Section 4.5 details our model and cascade construction.

4.3 Design Considerations

Several key questions are important to Tahoma's design. We touch on these issues below.

Issue 1: Object detection vs. image classification — In general terms, an object detector (e.g., YOLOv2 [124]) finds the location of particular object classes within an image, while an image classifier (e.g., ResNet50 [56]) identifies the overall contents of an image as one of a particular set of classes. When implementing binary `contains-object` predicates in a visual database application, the extra architectural and computational complexity of object detectors is unnecessary. Our cascades therefore use small and fast CNN image classifiers. Applications requiring object location within images, however, could build cascades from small and fast object detectors.

Issue 2: Online vs. offline classification — If a user's corpus is small or slow-growing enough to allow for offline classification of the entire dataset with available resources, this paper's techniques are unnecessary. Also, our techniques may be unneeded if query predicates are fixed and new ones are unlikely to be introduced. In such cases, the user could materialize the classification results for each image on ingest and store them in a standard database for future queries.

However, there are many exciting real-world applications where assumptions of a small or slowly growing dataset or stable query predicates are unrealistic. Consider, for example, a website for photo storage: Flickr has reported that days of heavy usage can see 25 million photo uploads [5]. Facebook had 350 million daily photo uploads as of 2013 [41]. Video applications can be even more extreme. A single self-driving car can have over a dozen cameras, each gathering 30 frames per second for many hours on a daily basis; a fleet of such cars can generate a huge amount of data.

Further, many applications require retrospective exploration and analysis, where the query predicates may not be known in advance. For example, a self-driving car engineer may wish to find historical examples of a new failure case, requiring the training of a new model to be run over an existing corpus. Or consider a police investigation reviewing thousands of hours of surveillance camera footage to find a local delivery van with a unique logo: a trained object detector would be ideal, but such a specific model is unlikely to already exist. In general, it is unlikely that all possible query predicates can be enumerated in advance for a visual analytics application.

Issue 3: Training costs — A deep CNN image classifier, such as ResNet50 [56], can take days to fully train, due to the model's architectural complexity and the huge training set needed for such a complex model. Such a burden is incompatible with our use case, since requiring this amount of time to install a new predicate in our system is unreasonable. Thankfully, our simple binary predicates typically do not require huge classifiers. The majority of our cascades consist solely of small, specialized classifiers, which train in just minutes.

Further, in cases where a deep network may be needed, training it from scratch is likely to be unnecessary. It is common practice to fine-tune pre-trained deep networks to a particular task [68]. Generally, when fine tuning, most of the deep network is frozen and only the last several layers are modified and retrained to a new task, taking advantage of already-learned features from a similar problem domain. In our experiments, we fine tuned ResNet50 for binary classification tasks using a modern GPU in only 2 to 4 hours.

Issue 4: Deployment scenarios — A visual analytics system may be deployed in a variety of scenarios, requiring accounting for differing data handling costs, in addition to classifier inference costs. Consider the following example scenarios, also later used in our experiments:

- Archive – In this situation, a large archival corpus of historical image data is stored on local drives. Each image must first be loaded from the drive and then transformed into an appropriate input format for the classifier.
- Ongoing – Here, video is continually ingested from its source into a datacenter-based query system, where it is transformed into appropriate representations that are stored on SSD for later queries. Because this data is transformed as it is acquired, only the cost of loading the representations from disk are considered at query time.
- Camera – If compute nodes are at the edge of the network (e.g., connected to surveillance cameras), the images can be directly provided to the classifiers. Only the image transformation costs must be considered, since transfer costs from camera to memory are negligible.

We show in Section 4.8.1 that scenario-awareness while evaluating of the accuracy and speed of classification systems can lead to a large practical increase in a system's throughput.

Table 4.1: Frequently used notation.

Notation	Definition
$\mathcal{I} = (I_1, \ldots, I_n)$	Image data corpus
$T_i = (t_{i1}, \ldots, t_{in})$	Content tuple for image I_i
K	Image classifier, s.t. $K(I_i) = t_i j$
$\mathcal{M} = (M_1, \ldots, M_m)$	Basic classification models
$\mathcal{A} = (A_1, \ldots, A_{n_a})$	Model architecture specifications
$\mathcal{F} = (F_1, \ldots, F_{n_f})$	Input transformation functions
$C = (M_1, \ldots, M_n)$	Cascade of n models belonging to $\mathcal{M}$
$\mathcal{C} = (C_1, \ldots, C_k)$	Collection of cascades for a binary query
$p_{\text{low}}, p_{\text{high}}$	Cascade model decision thresholds

4.4 Definitions and Notation

Here we will formalize the particular image classification problem addressed in this work. Frequently used notation is given in Table 4.1.

Content-based Queries — A query system that operates over images can perform queries over two main types of information: image metadata (e.g., GPS or acceleration information that may accompany frames from dashcams) and content extracted from images themselves (e.g., the image contains a bicycle). Metadata queries are easily handled by existing methods, so we focus this work on processing *content-based queries*.

Definition 2 (Content-based query)**.** *Given a corpus of image data $\mathcal{I}$ containing images $I_1, \ldots, I_n$, the tuples $T_i = (t_{i1}, \ldots, t_{im})$ represent the visual contents for each image I_i, where each element t_{ij} in this tuple represents a content object present in image I_i. A content-based query is constructed of predicates that can be evaluated with the elements of T_i.*

We restrict content-based queries to *binary queries*, which can be combined into a full query over all $T_{\mathcal{I}}$ tuples.

Definition 3 (Binary query)**.** *Given an image's content tuple T_i, a binary query involves a single* `contains-object` *predicate evaluated with a single tuple element, t_{ij}. A binary query thus asks if image I_i contains object t_{ij}.*

Despite this restriction, our envisioned system will support complicated queries that can be rewritten as a combinations of metadata and binary query predicates. For example, the query "Find images from Detroit containing a bicycle" can be decomposed into a metadata predicate (location = 'Detroit') and a binary query predicate (`contains_object`(bicycle)). Our focus is on choosing the best classifiers to implement a given `contains_object` predicate. While further query optimization could be done considering multiple binary predicates in concert, we leave that for future work and here concentrate on optimizing single predicates.

Image Classification — The content tuples are generally not available upon image acquisition and must be extracted from the image. In this work, we focus on extraction via *image classification*. In particular, we focus on queries where classification has not been performed in advance, so that classification must be performed as part of query execution.

Definition 4 (Image classification). *Given a corpus of image data $\mathcal{I}$, for each image $I_i \in \mathcal{I}$, we wish to generate a binary label L using a classifier K, such that $L_i = K(I_i)$. The label L_i corresponds to a member t_ij of content tuple T_i.*

The output of a classifier model can be thought of as a virtual column in a relation describing the content objects in images. For example, processing the query predicate `contains_object`(bicycle) would populate the bicycle column of this relation with the output of K_{bicycle}. The classifier K could be generated by a range of methods, including *basic models* like logistic regression or deep CNNs, or may be a collection of basic models, such as *classifier cascades*.

Definition 5 (Basic model). *A basic model M implements a classification method that accepts image I as input and outputs a binary classification result. Our system generates a large set of such CNN-based models $\mathcal{M} = (M_1, ..., M_m)$.*

Two factors parameterize a model M: *model architecture specification A* and *input transformation function F*.

Definition 6 (Model architecture specification). *The internal architecture of a model M is specified by A. For the CNNs used in our system, A_m describes network hyperparameters, such as the number and size of layers. $\mathcal{A} = (A_1, \ldots, A_{n_a})$ gives all potential architectural options for models in $\mathcal{M}$.*

Definition 7 (Input transformations). *Before the classification of an image I by model M, the raw image data is processed by an image transformation function F, such that input image I is transformed into output image I'. Such a function may perform one or more operations such as resizing, normalizing, or reducing color depth. The set $\mathcal{F} = (F_1, \ldots, F_{n_f})$ gives all functions available to pre-process image data for models in $\mathcal{M}$.*

We use the cross product of $\mathcal{F}$ and $\mathcal{A}$ as the model design space, resulting, in practice, in several hundred individual models in $\mathcal{M}$ for each binary query. A goal of this work is to determine which models are most suitable for a user's accuracy and runtime constraints and current deployment scenario.

Classifier cascades — Classification models can be aggregated into collections or ensembles to improve either accuracy or speed. One such method designed to improve classification speed is the *classifier cascade* [147].

Definition 8 (Classifier cascade)**.** *A classifier cascade $C = (M_1, \ldots, M_n)$ is a list of n basic models with probabilistically interpretable output. The models are run in series: image I_i is classified by M_1, and if the output is between two given decision thresholds, p_{low} and p_{high}, it is uncertain. If so, I_i is then classified by M_2. Otherwise, the cascade is stopped and M_1's output is accepted as the label of I_i. This continues to the final classifier M_n, whose output is always accepted.*

Given a set of classification models $\mathcal{M}$, we can construct a large set of cascades $\mathcal{C} = (C_1, \ldots, C_k)$ with up to n levels each, to be evaluated in terms of accuracy and throughput.

Model evaluation — The quality of classifier—either a model M or a cascade K—is given by its *accuracy* and its *throughput*. Accuracy gives the fraction of labels produced by M that are correct. Throughput is the number of classifications per unit time and measures how fast the model's relation is populated.

For a set of classifiers, we can find a subset that is Pareto-optimal over these two criteria. That is, there is a subset that is non-dominated in terms of accuracy and throughput, from which users can select a classifier to meet application needs.

Problem statement — With the preceding definitions, we can formally describe the problem addressed in this paper thusly:

For an image corpus $\mathcal{I}$ and a set of binary classification models $\mathcal{M} = (M_1, \ldots, M_m)$ parameterized by architectural specifications $\mathcal{A} = (A_1, \ldots, A_{n_a})$ and input transformation functions $\mathcal{F} = (F_1, \ldots, F_{n_f})$, find the set of classifier cascades $\mathcal{C} = (C_1, \ldots, C_k)$ constructed from models in $\mathcal{M}$ that are Pareto-optimal in terms of accuracy and throughput over $\mathcal{I}$.

4.5 Cascade Methodology

As discussed in Section 4.4, a classifier cascade is a series of classification models run one after another until a trusted classification result is found. Our methods depend upon building a large number of models that are combined in all possible combinations to cre-

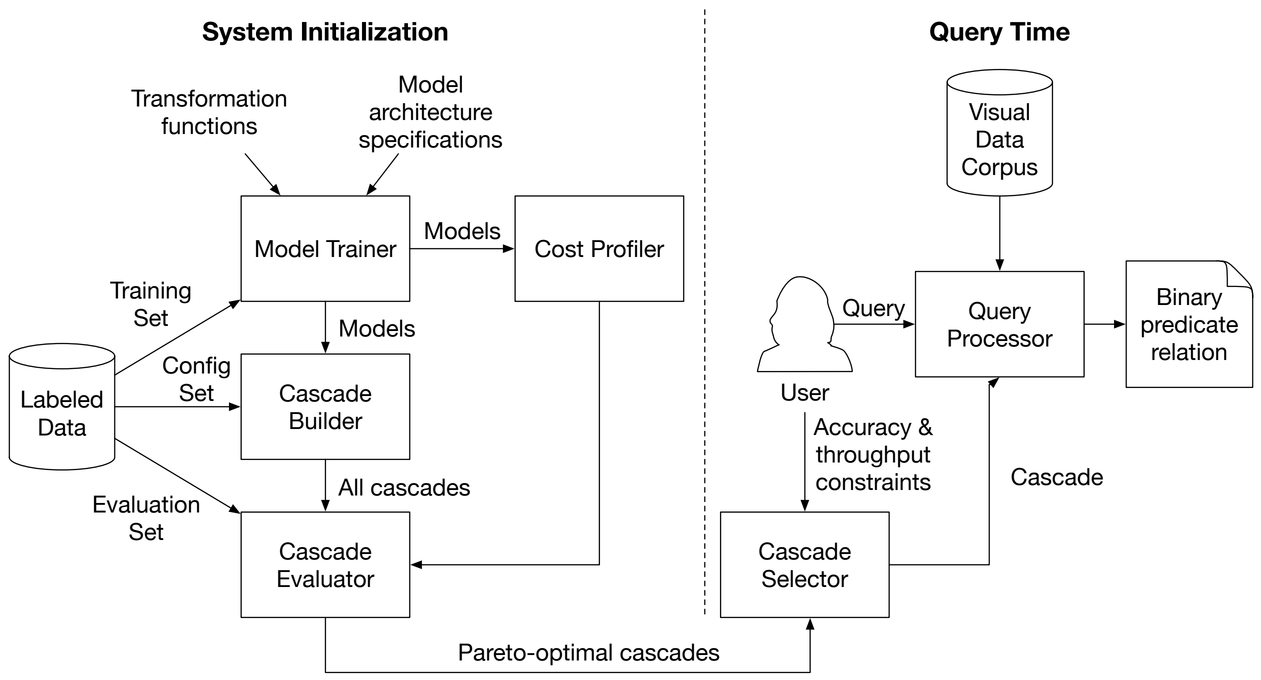


Figure 4.2: TAHOMA architecture

ate a huge number of cascades. After giving a brief overview of our system architecture, we will discuss the details of our models and cascades.

4.5.1 System Architecture

Figure 4.2 sketches out TAHOMA's architecture. TAHOMA has two main modes of operation: system initialization and query execution. During system initialization, model repository is prepared for each binary predicate, which requires a set of labeled data. This dataset is small compared to what is generally used to train deep CNNs: per binary predicate, TAHOMA requires 3,000–4,000 labeled images, with equal numbers of positive and negative examples. The labeled data is split into three sets for training, configuration, and evaluation. Training set $\mathcal{I}_{\text{train}}$, transformation functions $\mathcal{F}$, and architecture specifications $\mathcal{A}$ are provided as input to the model trainer.

For a given binary predicate, a set of models M (each implementing the `contains_-object` operator) is trained and provided to the cost profiler and to the cascade builder. The cost profiler measures the throughput of each model in the current deployment scenario (see Section 4.6). The cascade builder constructs TAHOMA's cascade set $\mathcal{C}$, using all possible combinations of size n of the models in $\mathcal{M}$ and the configuration set $\mathcal{I}_{\text{config}}$ (see Section 4.5.3). Using the evaluation set $\mathcal{I}_{\text{eval}}$, the cascade evaluator measures each cascade's accuracy and throughput (see Section 4.7.1). With this, the system determines the set of Pareto-optimal cascades for use at query time.

Similar to how approximate query systems like BlinkDB [7] and VerdictDB [111] allow users to specify approximation constraints as part of their queries, a Tahoma user provides their constraints on accuracy (U_{acc}) and throughput (U_{thru}) at query time (in the form of the highest tolerable loss in either of those parameters). The cascade selector chooses which of the Pareto-optimal cascades best suits the user's desired tradeoff. For example, the user may wish to maximize throughput as long as the resulting cascade does not suffer more than a 5% loss in accuracy over the most accurate cascade available. The user would set $U_{\text{acc}} = 0.05$ and provide no constraint for U_{thru}. The system would select the cascade from the set of Pareto-optimal cascades has an accuracy closest to (but not below) 95% of that of the most accurate cascade. Because this is a Pareto-optimal choice, there will be no faster cascades at that (or a higher) accuracy level. The selected cascade processes the data in the corpus, extracting the notional relation for the binary predicate in the user's query.

Integration considerations — Because our goal was to explore the optimization methods discussed in this paper, we implemented Tahoma as a standalone query system. However, we believe future deployments of Tahoma's ideas will likely be embodied in RDBMS software. The execution of a `contains_object` operator is analogous to that of a user-defined function (UDF) in a database such as PostgreSQL, and could be wrapped in the RDBMS `CREATE FUNCTION` statement. RDBMS query optimizers could leverage additional metadata relations, such as image location and capture date, to reduce the number of expensive Tahoma UDFs calls for a specific query. Further, UDF output could be stored as a partially materialized table, enabling further query optimization.

Tahoma's initialization process is run at the installation of each new binary predicate. During this, the profiled speeds could be used to inform the RDBMS query optimizer of the execution cost of the UDF (like the PostgreSQL's `COST` parameter for `CREATE FUNCTION`, if each Pareto-optimal cascade was implemented as a separate UDF). While indexing or materializing the output of the UDFs at system installation is not practical in our envisioned deployment scenarios, database triggers could be used to execute the Tahoma UDFs over newly ingested data after system initialization to pre-materialize the output for future queries. In such situations, slower processing may be tolerated for more accurate results, allowing a different Pareto-optimal choice than at query time.

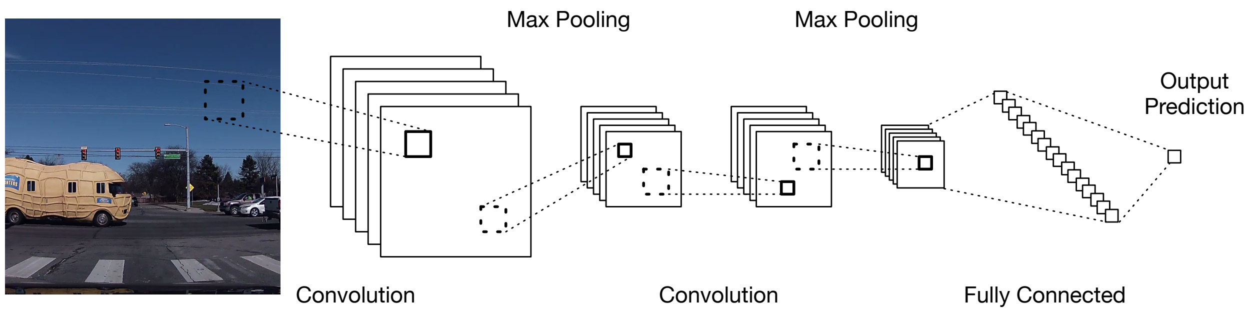


Figure 4.3: CNN architecture used by Tahoma. The number of layers and the number of nodes in each layer are varied as part of the model architecture specifications $\mathcal{A}$.

4.5.2 Building Models

With Tahoma, we create a huge number of cascades $\mathcal{C}$ by first training a large number of individual classification models $\mathcal{M}$. We build the collection of models in two ways: by varying the internal architecture of our CNN-based classifiers with our model architecture specifications $\mathcal{A}$ and by transforming the input images using the input transformation functions $\mathcal{F}$.

Model architecture variations — Tahoma uses convolutional neural networks for its models. Goodfellow et al. provide in-depth discussion on CNNs and deep learning [51]. Our CNNs follow the basic architectural pattern shown in Figure 4.3. Input values are fed into one or more layers of convolutional nodes. Each convolutional layer is followed by a max pooling layer, connected by rectified linear activations (ReLu). The final convolutional layer feeds into a fully connected ReLu layer. A sigmoid output node provides the inferred label. A key point is that our CNNs are small (and thus fast), typically having only one to four convolutional layers. When creating models, we vary these architectural details according to $\mathcal{A}$; for our experimental settings of $\mathcal{A}$, see Section 4.8.1.

Input transformations — We also vary the physical representation of the input to each model. The set of input transformation functions $\mathcal{F}$ comprises functions that perform one or more image processing operations, such as resolution scaling and color channel modification. These types of transformations are useful for building fast, small models: reducing image size and color depth can greatly reduce the number of model input values, directly reducing the size of the CNN's tensor operations. For our experiments, we scaled the image resolution (30x30, 60x60, 120x120, and 224x224 pixels), and for each image size, we used five different color variations (full 3-channel color, each of the individual red, green, and blue color channels, and single-channel grayscale).

The design space defined by these model architecture variations and input transformation functions result in hundreds of different model configurations (360 in our experiments). Once each model is trained on a labeled subset of $\mathcal{I}$, we can compose the models into cascades. Training can take less than a minute for the smallest networks (one convolutional layer with few nodes) with the smallest inputs (30x30 pixels, 1 color channel) to nearly an hour for the largest. Overall, training 360 models for a single binary predicate requires about 12 hours when done serially on an NVIDIA Tesla K80 GPU. Training is parallelizable, so this cost can be greatly reduced in practice.

4.5.3 Computing Decision Thresholds

Each model in $\mathcal{M}$ provides a probabilistic output for a binary classification problem, a real number ranging from 0 to 1. Each model has a pair of decision thresholds, p_{low} and p_{high}, which determine whether the model's labelling decision should be trusted. If the output $o \leq p_{\text{low}}$ or $o \geq p_{\text{high}}$, the model's output is accepted as the output of the cascade. If $p_{\text{low}} < o < p_{\text{high}}$, then we consider the model's output to be uncertain and reclassify the image with the next model in the cascade.

These thresholds are chosen on a per-model basis, such that the precision of classification results with $o \leq p_{\text{low}}$ or $o \geq p_{\text{high}}$ matches a predefined constraint while recall is maximized. Using a small configuration dataset distinct from the training set, the thresholds are determined via a grid search that sweeps through the potential thresholds to find those that provide a precision value greater than or equal to the target precision and selects the thresholds from those that maximize recall.

4.5.4 Constructing Cascades

Each model has its decision thresholds determined independently, rather than in the context of a specific cascade. This assumption of independence allows us to quickly instantiate and evaluate the millions of possible multi-level cascades that can be constructed from the models in $\mathcal{M}$.

To determine the accuracy and throughput of our cascades, we first classify a set of labeled images $\mathcal{I}_{\text{eval}}$ with each model in $\mathcal{M}$. (The images in $\mathcal{I}_{\text{eval}}$ are distinct from those for training and determining decision thresholds so that the resulting accuracy measurements are not the product of overfitting.) Since the cascades in $\mathcal{C}$ comprise combinations of the models in $\mathcal{M}$, the above evaluation need only be done once per model (360 times in our experiments) and not once per cascade (1.3 million).

Ensuring the independence of both model evaluations and decision thresholds (as described in Section 4.5.3) enables extremely fast evaluation of cascades: our evaluation required just over one minute to determine the accuracy and throughput values for 1.3 million cascades. As such, once models have been trained and classified, the selection of a cascade can be part of query planning at query execution time and can thus incorporate query-specific performance criteria (e.g., which storage devices are providing input images).

Using the pre-computed classification results for $\mathcal{I}_{\text{eval}}$ for each model in the cascade, the cascade execution is simulated to obtain its label predictions for $\mathcal{I}_{\text{eval}}$. The cascade's accuracy is then computed by comparing with true labels for $\mathcal{I}_{\text{eval}}$.

4.6 Data Handling Costs

An often overlooked part of image classification is the cost of loading and preparing the data prior to inference. Of the rare projects in the computer vision field that report inference speeds (e.g., the YOLO family of object detectors [123,124]), none report video decoding or image loading time. Likewise, the NoScope video query system [72] explicitly ignores these costs, claiming that GPU-based decoding is so fast as to be negligible or that decoding might be avoided altogether by obtaining raw video frames directly from the generating camera sensor.

This last point hints at why image loading or decoding costs *should* be included when evaluating these systems: deployment scenarios for a given query system may differ drastically. Some may use top-of-the-line GPUs for video decoding, others may store video frames as individual image files on disk, while some may require video to be transported over a network prior to processing. Further, if multiple classification models with a variety of input representations might be used for classification, data handling costs—including preprocessing—must be included in throughput evaluations when deciding which classifier will be used at query time.

More concretely, for accurate model comparisons to be made, the throughput must be measured as the reciprocal of the average classification time, t_{classify}, defined as follows:

$$t_{\text{classify}} = t_{\text{load}} + t_{\text{transform}} + t_{\text{infer}}$$

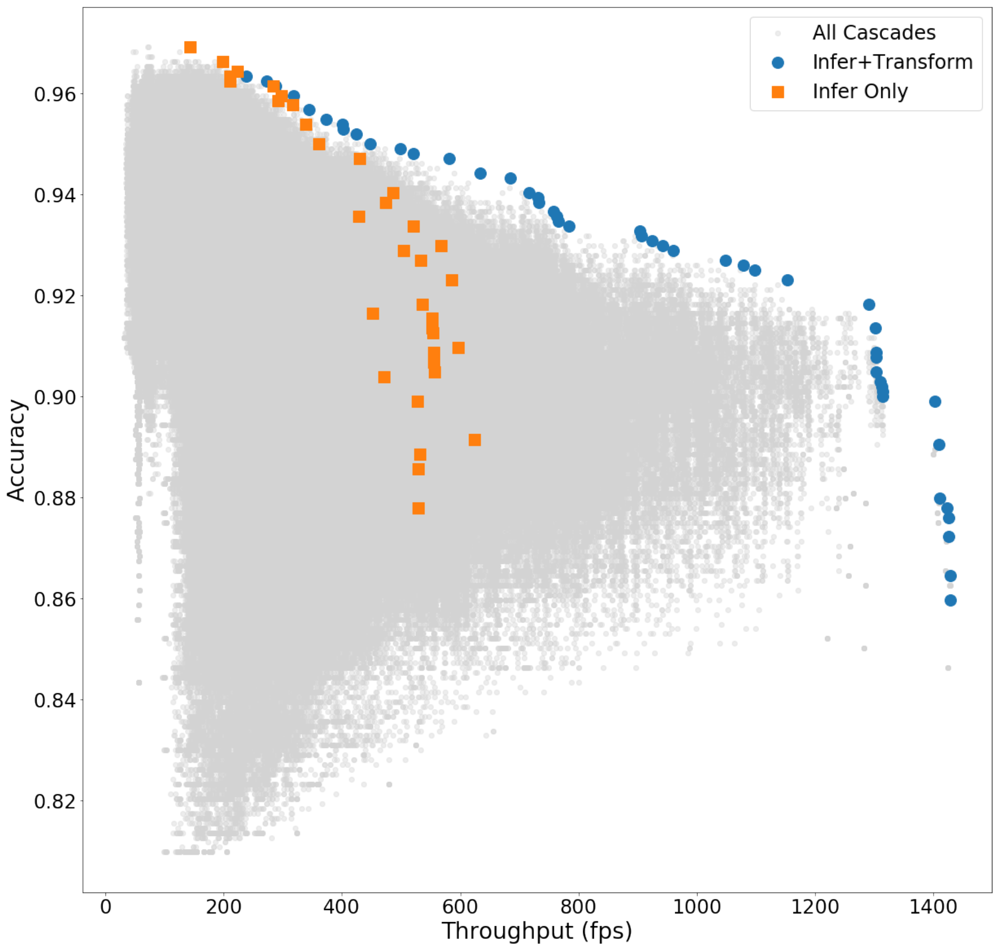


Figure 4.4: Cascades (gray) and Pareto frontier (blue) for an example deployment scenario, compared to the Pareto frontier for a scenario only considering inference costs (orange).

where each time *t* above is the average recorded over some typical set of input images, measured on the deployed system.

Our experiments in Section 4.8.4 demonstrate that choosing among cascades with incorrect cost assumptions can lead to a large decrease in throughput. This throughput loss can be seen in Figure 4.4. The gray points depict all possible cascades for an example binary predicate (e.g., `contains-object(semitruck)`) in a deployment scenario where image loading costs are negligible (full size raw data is already present in memory), but image preprocessing costs are incurred to transform the images into the appropriate resolutions and color representations for each model. The points on the Pareto frontier (shown in blue) represent the cascades that present the best tradeoffs between accuracy and throughput under this deployment scenario. The orange points show the cascades that would be on the Pareto frontier for this binary predicate if only inference costs were considered (pre-processed images are already present in memory). *If the preprocessing costs were not considered, the query throughput would be far below its potential for most accuracy levels.*

4.7 Evaluation Methods

In this section, we detail the methods used to evaluate and compare the cascade sets generated by our system.

4.7.1 Evaluating Cascades

As just discussed, each of the cascades in $\mathcal{C}$ can be displayed on a plot of accuracy versus throughput, as shown in Figure 4.4, where the cascades with the best tradeoffs between accuracy and throughput belong the Pareto frontier. The points on the Pareto frontier are those not dominated by any others, where point is said to dominate another point if it has greater-than-or-equal values for all attributes, and strictly greater values for at least one attribute. [110]. In database literature, this is sometimes referred to as a skyline [75]. Computing the Pareto frontier over just two attributes, as we do here, is $O(n \log n)$ in the number of points [86].

To determine the accuracy and throughput of our cascades, we first classify a set of labeled images $\mathcal{I}_{\text{eval}}$ with each model in $\mathcal{M}$. (The images in $\mathcal{I}_{\text{eval}}$ should be distinct from those used for training and determining decision thresholds so that the resulting accuracy measurements are not the product of overfitting.) Since the cascades in $\mathcal{C}$ comprise combinations of the models in $\mathcal{M}$, the above per-model evaluation need only be done once per model, and can be reused when evaluating alternative cascades.

Ensuring the independence of both model evaluations (as described above) and decision thresholds (as described in Section 4.5.3) enables extremely fast evaluation of cascades: our evaluation required just over one minute to determine the accuracy and throughput values for 1.3 million cascades. As such, once models have been trained and classified, the selection of a cascade can be made as part of query planning at query execution time, and can thus incorporate query-specific performance criteria (e.g., which storage devices are providing input images).

Using the pre-computed classification results for $\mathcal{I}_{\text{eval}}$ for each model in the cascade, the cascade execution is simulated to obtain its predictions of the labels for $\mathcal{I}_{\text{eval}}$. The cascade's accuracy can then be computed by comparing with true labels for $\mathcal{I}_{\text{eval}}$.

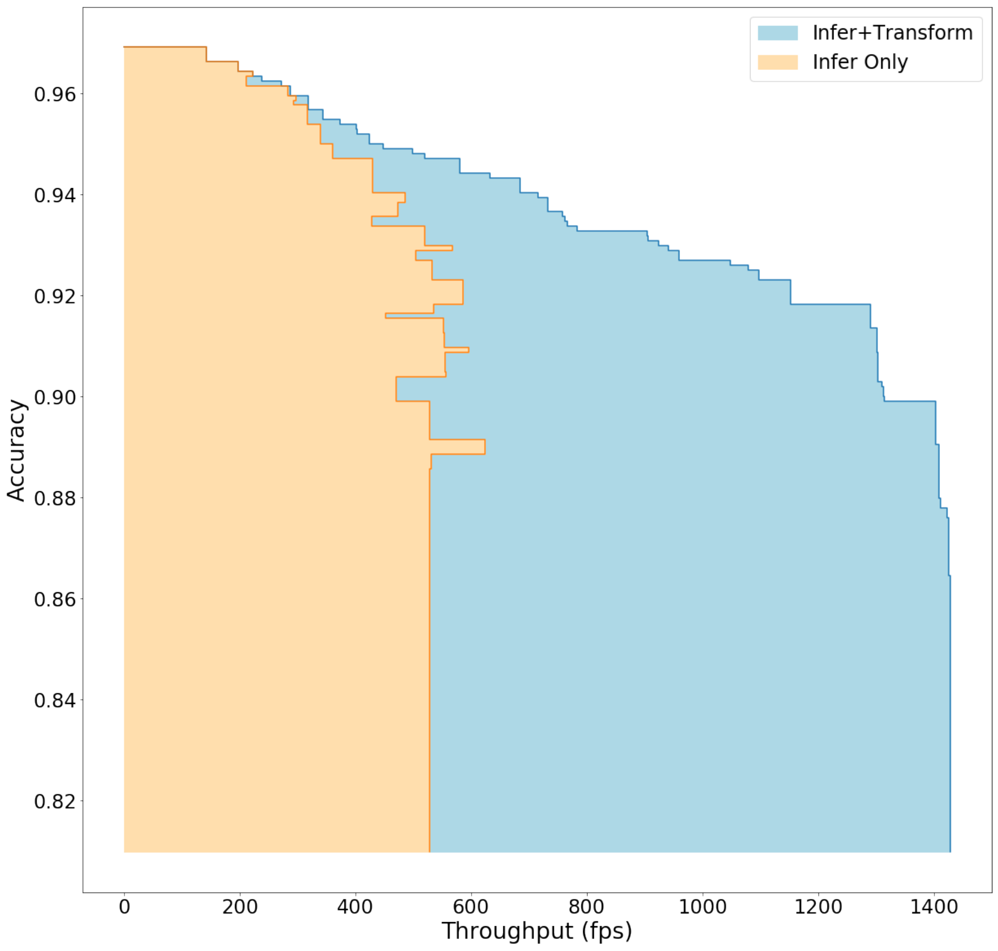


Figure 4.5: Areas to the left of Pareto frontiers, used for evaluation. Frontiers correspond to the cascades in Figure 4.4.

4.7.2 Comparing Cascade Sets

Figure 4.5 illustrates how we compare sets of cascades, such as those belonging to a Pareto frontier, in terms of throughput. We compute the area to the left of the curve (ALC) created by those points in this plot over a given accuracy range. Because a Pareto frontier is a collections of points and not a curve, we interpolate the curve as a step function, as shown in Figure 4.5. Dividing the ALC by the size of the accuracy range give the average throughput for cascades in the Pareto frontier $\mathcal{C}_f$:

$$\textit{Avg. throughput} = \frac{\text{ALC}(\mathcal{C}_f)}{\text{acc}_{\text{high}} - \text{acc}_{\text{low}}}$$

Dividing the ALC of one frontier by another gives the speedup of the former over the latter:

$$\textit{Speedup} = \frac{\text{ALC}(\mathcal{C}_1)}{\text{ALC}(\mathcal{C}_2)}$$

To make comparisons fair, we use the accuracy range for the full set of cascades for each configuration, and then choose the smallest said range. In some cases, we compute the ALC for a Pareto frontier's cascades in a different cost context (detailed in Section 4.6). In this case the cascades are no longer a strict Pareto frontier (see the orange area in Figure 4.5). However, we can still compute ALC and make comparisons.

4.8 Experiments

We have implemented TAHOMA as a prototype system, and in this section, we discuss our experiments that compare TAHOMA's performance against existing baseline methods, investigate the effects of data handling costs of different deployment scenarios, evaluate the effects of the individual input transformations, and analyze the effects of increasing cascade depth.

4.8.1 Experiment Setup

To evaluate the methods and components used in TAHOMA, we designed a series of experiments with the following setup.

Binary predicates

We evaluated TAHOMA's performance over a set of 10 queries with a single `contains-object` binary predicate, randomly chosen from the 1,000 categories in the ImageNet dataset [127]. Each query is of the form "SELECT * FROM images WHERE `contains-object`(*category*)", where *category* is one of those shown in Table 4.2. The ImageNet dataset provides about 1,200 training images for each category. We held out 200 of each category's images as a validation set. The remaining images were labeled as positive examples. We then selected a matching number of random images drawn from the remaining 999 ImageNet categories and labeled them as negative examples. Because these training sets were relatively small, we followed common data augmentation practice by creating a copy of each image that was flipped left-to-right, doubling the amount of training data.

Table 4.2: Binary predicates used in our experiments and their corresponding ImageNet category IDs.

Predicate	ImageNet ID	Query	ImageNet ID
1. acorn	n12267677	6. ferret	n02443484
2. amphibian	n02704792	7. komondor	n02105505
3. cloak	n03045698	8. pinwheel	n03944341
4. coho	n02536864	9. scorpion	n01770393
5. fence	n03930313	10. wallet	n04548362

We performed similar labeling for the validation set. For performance evaluation, we collected an additional dataset for each category using the image search functionality of the Google and Bing search engines, resulting in roughly 500 positive images per category. To find negative examples, we first created a large collection of images by performing image searches on Google and Bing for each of the remaining 999 ImageNet categories. From that, we randomly chose our negative examples to match the number of positive ones.

Cascade configurations

We used the Keras [27] deep learning library running TensorFlow [6] to train and execute the CNNs used in our cascades. We varied several network architecture hyperparameters to create a range of models for each binary predicate: the number of convolutional layers (1, 2, 4), the number of convolutional nodes in each layer (16, 32), and the number of nodes in the final dense layer (16, 32, 64). These hyperparameters provide a range of reasonable options similar to those used in other systems (e.g., [72]). We also varied the size of the input images (30x30, 60x60, 120x120, 224x224). For each input size, we used five different image representations: full 3-channel color, each of the individual red, green, and blue color channels, and single-channel grayscale. In all, we constructed 360 simple CNNs for each binary predicate.

We also included in our pool of classifiers a fine-tuned implementation of ResNet50 [56] that had been pre-trained on ImageNet. Fine-tuning was done using standard techniques; the final 1000-class classification layer was replaced with 64-node ReLu dense layer, followed by a 2-node softmax layer for the output of the binary prediction. These layers were trained using the same training set as the smaller, specialized classifiers.

For each classifier, we calibrated its decision thresholds using the validation dataset for five precision settings: 0.91, 0.93, 0.95, 0.97, and 0.99. We use each classifier variation to construct cascades of one and two levels, as well as three-level cascades with ResNet50 as the final layer (see Section 4.8.6), resulting in 1,301,405 possible cascades per predicate.

Deployment Scenarios

The data handling and preprocessing costs of particular deployment scenarios can have a large effect on the optimal choices in classifier cascades. To demonstrate this, we analyzed our cascades under four different scenarios, corresponding to those in Section 4.3:

- INFER ONLY — This scenario ignores data handling and transformation costs—only inference costs (i.e., only the time required to evaluate the CNN) are considered when computing throughput, a practice commonly used in computer vision literature. However, as we show, the fastest inference often does not imply the fastest end-to-end query performance in practical deployments.
- ARCHIVE — This scenario includes the cost of loading a full-size image off an SSD hard disk, as well as the cost of resizing that image to the appropriate input size for a given classifier, as might be done when querying an existing corpus of archived image or video data.
- ONGOING — This scenario corresponds to deployments where images are resized on ingest before saving to disk. Load times are smaller, since the full-sized image is not loaded from disk if not needed for a particular classification. This scenario may occur when setting up a data collection system in tandem with a query system; proper image sizes for object detectors are known and initial transform costs can amortized over many queries.
- CAMERA — This scenario only includes the computation costs of resizing the images, as in deployments where loading costs are negligible (e.g., images are loaded to memory directly from a connected camera sensor).

Data handling costs for the above scenarios only occur once for a given input: if a cascade includes two classifiers that use, for example, a 30x30 pixel red channel input, the costs to create that input are incurred only once per image.

Cascade evaluation

To compare two sets of cascades in terms of throughput, we compute the area to the left of the curve (ALC) created by those points in this plot over a given accuracy range. Because a Pareto frontier is a collection of points and not a curve, we interpolate the curve as a step function. Dividing the ALC by the size of the accuracy range give the *average throughput* for cascades in the Pareto frontier. Dividing the ALC of one frontier by another gives the *speedup* of the former over the latter. For fair comparisons, we use the accuracy range for the full set of cascades for each configuration and choose the smallest said range. In some cases, we compute the ALC for a Pareto frontier's cascades in a different cost context. These cascades are no longer a strict Pareto frontier, but we can still compute ALC for comparisons. When comparing TAHOMA to a single classifier, such as against our ResNet50 baseline system, we choose the optimal cascade whose accuracy is both higher and closest to the accuracy of the single classifier.

Hardware

We used an Amazon EC2 p2.xlarge instance with an NVIDIA Tesla K80 GPU to do training, inference, and throughput measurements for all CNNs. For operations suited to parallel CPU processing—such as finding the Pareto frontiers for our cascade sets—we used a 32-core (2.8GHz) Opteron 6100 server with 512GB RAM.

4.8.2 Comparison Against Baselines

Summary: TAHOMA yielded significant throughput gains over our baselines under a variety of deployment scenarios.

Overview — We compare TAHOMA against a fine-tuned, pre-trained ResNet50 implementation [56], as well as a set of non-optimized cascades, which comprise a subset of TAHOMA's design space. These two-level cascades terminate in a full-cost classifier (i.e., ResNet50) and use full-color 224x224 images as input. These are similar to design to CNN-based cascades in previous work [72]. An illustration of the difference in the design spaces is shown in Figure 4.6, with TAHOMA's available cascade options being markedly larger due both the use of data transformations on the inputs and the additional cascade depth.

Throughput gains — Figure 4.7 shows the performance of TAHOMA compared to our baselines for the four deployment scenarios from Section 4.8.1. When only considering inference speed (i.e., INFER ONLY), TAHOMA yielded a 98x speedup over using a fine-tuned ResNet50 classifier alone, averaged over our 10 predicates. TAHOMA showed a 35x average speedup over the accuracy range provided by the Baseline cascades. At the accuracy level provided by the fastest Baseline cascade for each predicate, TAHOMA yielded a 59x speedup on average. For the other scenarios, data handling overheads reduce the

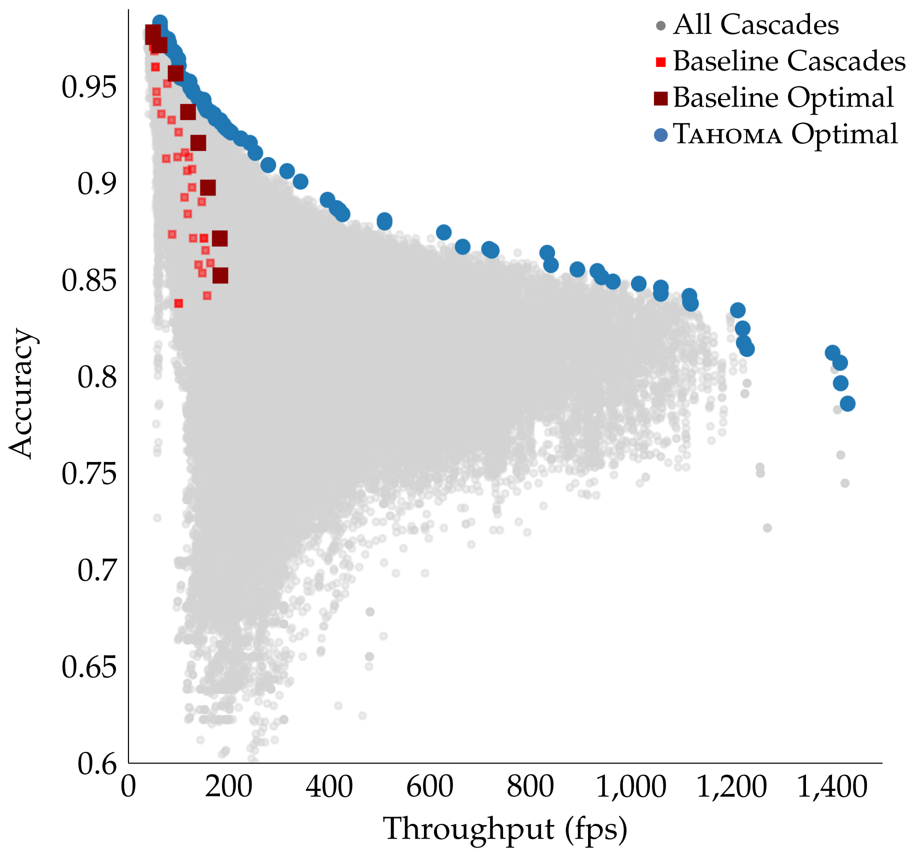


Figure 4.6: A comparison of the cascade space of Tahoma (gray) compared to that of our Baseline cascades (red). Tahoma's Pareto-optimal points are in blue. This example uses our komondor binary predicate under the Camera cost model.

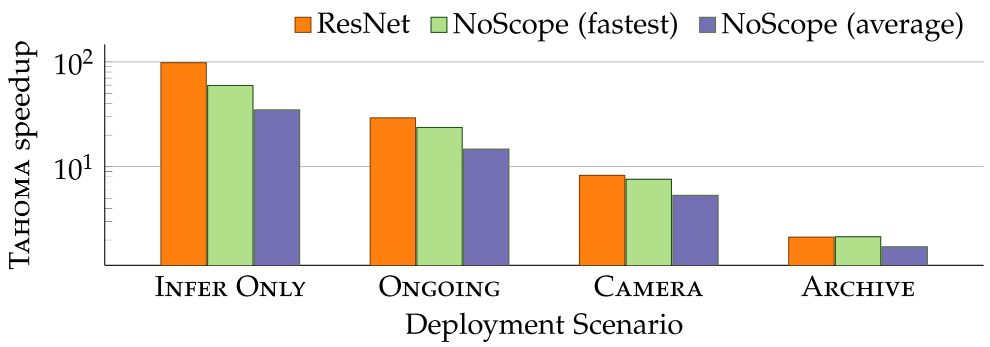


Figure 4.7: Average speedup values of Tahoma over baselines. *ResNet50* and *Baseline (fastest)* comparisons use the optimal cascade with the nearest higher accuracy to ResNet50 and the fastest Baseline cascade, respectively. *Baseline (average)* shows average speedups over the Baseline accuracy range.

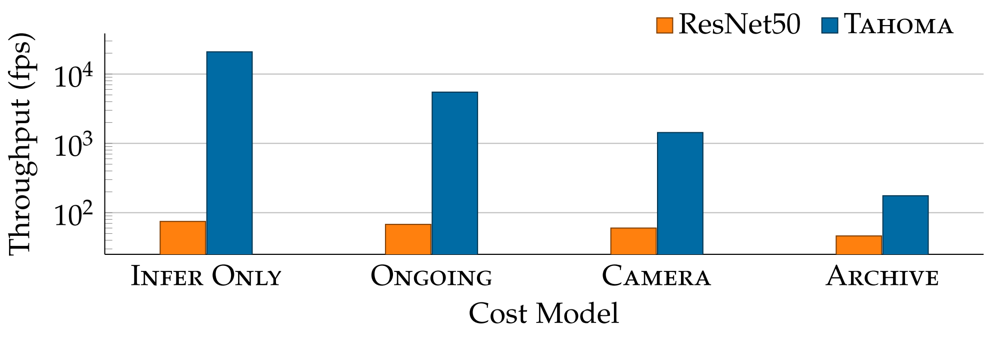


Figure 4.8: Throughput of Tahoma and ResNet50 of fastest cascades for each cost model, averaged over 10 binary predicates.

speedup gains. Nevertheless, Tahoma achieves substantial speedup in all scenarios, and even the most expensive scenario that requires costly loading and transformation costs (Archive) shows a nearly 2x speedup versus both ResNet50 and Baseline.

If speed is the priority, Tahoma allows a user to trade accuracy for a large throughput boost. Figure 4.8 shows Tahoma's fastest optimal cascade compared to ResNet50. Across all predicates, the fastest cascades were not true cascades at all: they comprised a single specialized classifier with adequate accuracy and high throughput. In the Infer Only scenario, Tahoma achieved an average throughput of 20,926 frames per second—280 times faster than our fine-tuned ResNet50 models, which had an average throughput of about 75 frames per second. The more realistic Ongoing scenario still achieves 5484 frames per second—an 81x speedup. These large speedups come at a price, though: under Infer Only, the Tahoma's fastest cascade was on average 12% less accurate than ResNet50. Of course, as Figure 4.6 shows, the optimal cascades provide a rich space of throughput and accuracy tradeoffs, so users can find the right balance for their needs.

4.8.3 Comparison with NoScope

Because NoScope [72] is the existing system most closely aligned with our work, we ran experiments to directly compare the two systems. For these experiments, we used the code and datasets (coral and jackson) provided by the NoScope authors[1]. The other datasets presented in the NoScope paper were not publicly available. We used the default parameters provided in the NoScope code for each dataset and report results for both systems with a target precision of 0.95 used to select cascade thresholds. YOLOv2 [124] was used as the final, expensive classifier for both systems. Both NoScope and Tahoma were run on AWS p2.xlarge instances. Note that CPU and GPU specifications differ be-

[1]`https://github.com/stanford-futuredata/noscope`

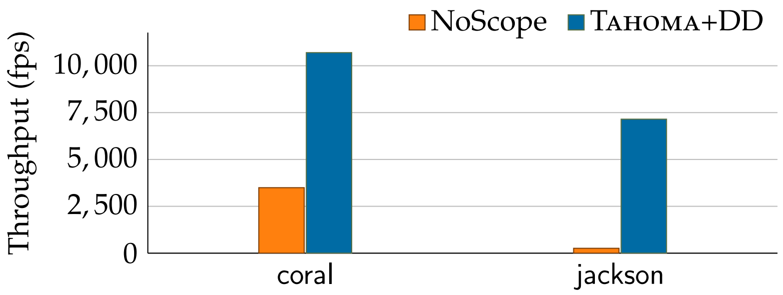


Figure 4.9: Comparison with NoScope. Tahoma+DD is Tahoma with a simulated NoScope-style difference detector.

tween our NoScope installation and the one in the NoScope paper, so raw performance numbers differ between our experiments and theirs.

To compare NoScope and Tahoma on an equal footing, we implemented Tahoma+DD, which is Tahoma with a simulated difference detector equivalent to that used by NoScope. The difference detector measures the similarity between the current frame and previously seen ones and reuses previous results if the compared frames meet a similarity threshold. This mechanism is orthogonal to our work and increase NoScope's throughput by avoiding many classifier executions. To create Tahoma+DD, we recorded frame similarity using NoScope's difference detector and reused Tahoma's results for frames meeting NoScope's threshold instead of classifying them.

Additionally, both systems used basic frame skipping, only processing one of every 30 frames. The results shown here include only those frames actively processed by each system, not those skipped this way. Tahoma+DD results are measured in the Infer Only deployment scenario, which matches NoScope's throughput measurements. Tahoma+DD results use the Pareto-optimal cascade with the closest but higher accuracy level to that of the NoScope for each dataset.

Figure 4.9 compares the throughput for NoScope and Tahoma+DD for the two public NoScope datasets. For both datasets, Tahoma+DD significantly outperformed NoScope. On coral, Tahoma+DD system reached a throughput of 10,700 fps, while NoScope's throughput was 3494 fps, giving Tahoma+DD a 3.1x speedup over NoScope. On jackson, Tahoma+DD system reached a throughput of 7,150 fps, while NoScope's throughput was 260 fps, giving Tahoma+DD a 27.5x speedup over NoScope[2].

[2]As may be apparent from the results, the coral dataset was a much simpler classification task than jackson, with far more reused results from the difference detector for coral (25.2% reused) than for jackson (3.8% reused). NoScope used the expensive YOLOv2 model for a significant number of frames on jackson, as well, leading to its slow performance. Tahoma+DD's much larger cascade design space allowed it to find an accurate cascade that was able to avoid calling YOLOv2 for all but a few frames.

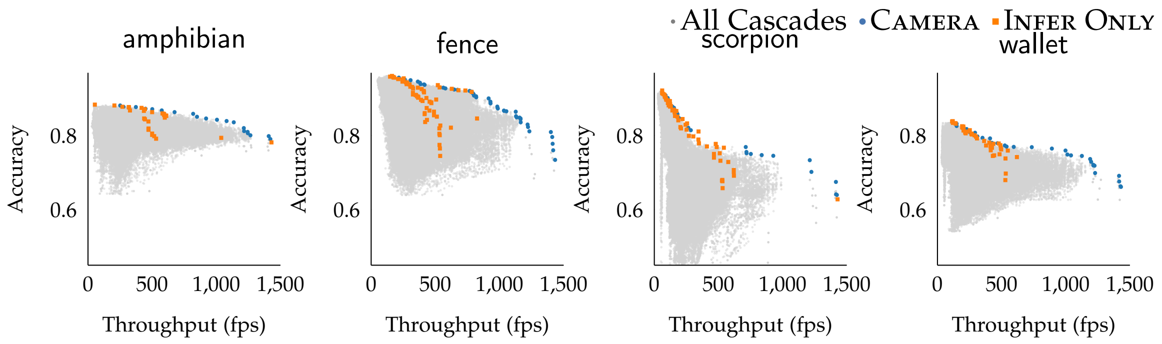


Figure 4.10: Pareto frontiers for several of our binary predicates, under the Camera cost model (blue), compared to cascades that appear in the Pareto frontier for the Infer Only model (orange).

4.8.4 Deployment Scenario Awareness

Figure 4.10 shows (in blue) the Pareto frontier of all classifier cascades for the Camera cost model for several of our binary predicates. Additionally, the cascades that would be Pareto-optimal for each query under the Infer Only model are shown in orange. These orange points form a non-convex curve, since they are *not a Pareto frontier* under the depicted Camera cost model. Each cascade may be impacted differently by loading and transformation costs, so their throughputs can change relative to one another in different deployment scenarios. With few exceptions, the optimal cascades under Camera are different than the Infer Only ones. It is clear that if the data handling costs of a scenario like Camera were ignored and the "optimal" cascades were chosen only considering inference costs, considerable throughput gains would be missed.

Table 4.3 shows the difference in throughput in our deployment scenarios when cascades are chosen in a scenario-oblivious way (i.e., when only inference costs are considered, as in Infer Only) versus when the cascades are chosen taking scenario costs into consideration. Because Tahoma provides a tradeoff between accuracy and throughput, we show results for four different levels of permissible accuracy loss. A user, for example, may decide that a 5% decrease in accuracy is acceptable in order to process images faster. Then, in the Camera scenario, the system's throughput would increase by 59.5% if cascades were chosen taking data handling costs into consideration, instead of being oblivious to these costs and only considering the classifier's inference.

Table 4.3: Throughputs, given in frames per second (fps), for various deployment scenarios when the cascade choices chosen in either oblivious or aware of scenario data handling costs. Here, permissible accuracy loss indicates how much accuracy the user is willing to trade for an increase in throughput. Scenario awareness can lead to significant throughput increases, shown in parentheses.

Permissible accuracy loss	Scenario: ARCHIVE		Scenario: CAMERA		Scenario: ONGOING	
	Oblivious	Aware	Oblivious	Aware	Oblivious	Aware
0% loss	57.5	58.3 (+1.4%)	107.1	107.1 (+0.0%)	111.9	111.9 (+0.0%)
2% loss	85.1	91.1 (+7.1%)	267.5	324.6 (+21.3%)	985.2	985.3 (+0.0%)
5% loss	103.1	117.1 (+13.5%)	344.7	549.9 (+59.5%)	1938.7	2000.8 (+3.2%)
10% loss	130.6	142.0 (+8.7%)	568.0	806.8 (+42.0%)	3669.1	3669.1 (+0.0%)

4.8.5 Analysis of Input Transformations

TAHOMA uses several different input transformations to expand the space of simple classifiers used to construct cascades. To see how these affect TAHOMA's performance, we constructed four cascade sets that used varying subsets of the transformations: *None*, which used no input transformations (i.e., all inputs are 224x224 three-color-channel images); *Color Variations*, which used only the transformations that extract the color channels or create grayscale images; *Resizing*, which used only transformations that reduce resolution; and Full, which used the full set of transforms included in TAHOMA.

Figure 4.11 shows the average throughput for each cascade set for each binary predicate, computed using the ALC method described in Section 4.8.1. We computed these values over the accuracy range of the *Full* cascade set for each predicate. Image resizing operations by far have the largest impact on throughput, giving nearly a ten-fold increase over *None*. The resized 3-channel, 30x30 pixel input images equate to 2,700 input values, while the full-sized 3-channel, 224x224 pixel images equate to 150,528 values; this huge reduction in input size results in orders of magnitude fewer tensor operations during CNN inference. Likewise, reducing the color depth of an image from three channels to one reduces the CNN's computational requirement by two thirds. These transforms, especially resolution reduction, are critical to high query throughput; they enable much smaller CNNs, more than paying for the transforms' computational costs.

4.8.6 Analysis of Increased Cascade Depth

Each additional level added to the cascade exponentially increases the size of the cascade design space. Evaluating the 1.3 million cascades used in results reported elsewhere in this section is fast, taking about 1 minute on average per binary predicate. However,

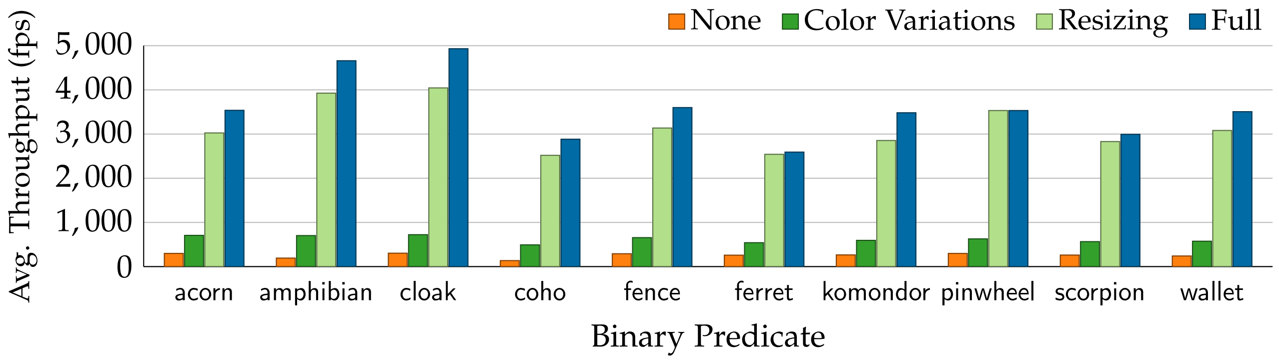


Figure 4.11: Average throughput of optimal cascades for cascade sets that use different input transformations.

this cascade set includes a full cross-product only for one- and two-level cascades, with three-level cascades restricted to those with fine-tuned ResNet50 as the final classifier. If we considered all possible three-level cascades from all available models, our cascade set balloons to about 45 million distinct cascades, requiring about 40 minutes of evaluation time per predicate. Evaluating a full cross-product of four-level cascades is intractable (roughly 360^4 total cascades).

Figure 4.12 shows how a cascade set's Pareto frontier evolves as the maximum depths of the cascades increase. Each set of cascades includes all depths up to its maximum. A "one level" cascade simply corresponds to our set of basic classification models. Where a cascade depth indicates "+ ResNet50", we append ResNet50 as an additional final level of the cascade. That is, "Two level + ResNet" cascades comprise two levels populated by any model from our collection, followed by a final ResNet50 level. The addition of a ResNet50 level effectively doubles the number of cascades in a cascade set. Adding a full additional level (drawn from our full set of models) increases the cascade set size and evaluation time exponentially.

As can be seen, increasing the depth of the cascades has diminishing returns to throughput at query time, while greatly increasing computational cost of cascade evaluation during system initialization. Moving from "Two level + ResNet50" to "Three level (full)" only increases average throughput by 1.0%, while increasing evaluation time nearly 40-fold to just over 40 minutes. Thus, for our experiments, we have restricted cascades to at most "Two level + ResNet50".The minimal increase in throughput capabilities of additional layers is not worth the huge increase in evaluation time.

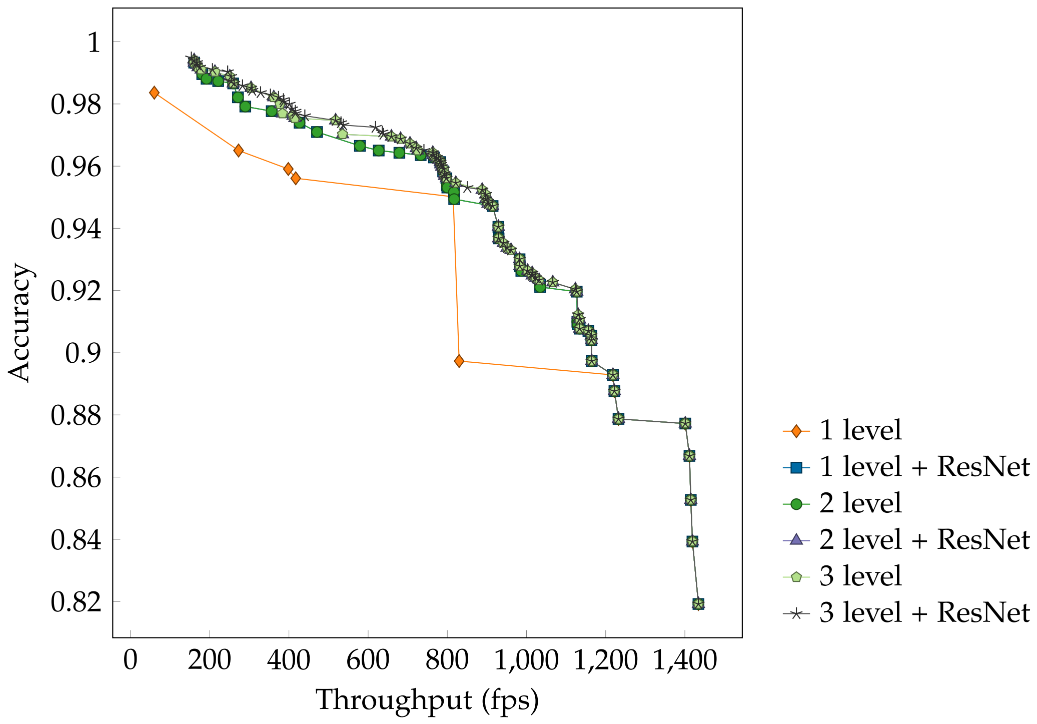


Figure 4.12: Evolution of Pareto frontier as cascades depth increases, shown for our fence predicate in the CAMERA scenario. Other predicates and scenarios showed similar results. As cascades get deeper, Pareto frontier improvements become negligible.

4.9 Related Work

TAHOMA builds upon a rich history of work in the database, image processing, and machine learning communities.

Query systems for visual data — A number of database and query systems targeted at visual data have been developed over the past two decades [63, 138]. Early approaches involved manual textual labeling (e.g. [137]) or extracting rudimentary low-level features (e.g. [46, 125]). Later systems performed additional semantic object extraction, using hand-written functions and statistical methods to define objects [117, 118]. That work, however, precedes the recent deep learning revolution in computer vision and relies on highly manual object extraction methods that limit system applicability and scale.

Work that extended relational query methods over visual data [50,87,94] may be useful in extending query capabilities in a system like ours. Fagin's work with the Garlic system [42], for example, deals with a number of issues particular to querying multimedia databases and the fuzziness of data extracted from multimedia data. However, the data handled by TAHOMA does not exhibit the same kind of fuzziness as Garlic, which is concerned with approximate matches and image similarity. Our system deals with binary predicates that—while based on probabilistic classifier output—are considered strictly true or false at query time.

There has been a recent explosion of interest in query systems for visual data due to advances in classifier accuracy made possible by convolutional neural networks and to the massive datasets made possible by cheap storage and image sensors. NoScope [72], for example, uses several techniques, including classifier cascades, to accelerate query processing over video. Our key departure from this prior work is to include transformations of the input image representation within our query plans, drastically expanding the design space of classifier cascades and enabling much smaller models and order-of-magnitude throughput improvements. We compare our our system with NoScope in Section 4.8.3.

BlazeIt [71] optimizes queries for objects found in video, in part by using small, specialized CNNs to quickly answer queries where possible. Like NoScope, these specialized CNNs do not make use of Tahoma's input transformations. The rich space of Tahoma's specialized CNNs, however, could potentially be integrated into a query optimizer like BlazeIt. Focus [61] is a system that indexes objects in live video. It too, uses specialized CNNs to speed up queries. Focus varies the resolution of input images when creating the set of specialized CNNs, though it does not use TAHOMA's other input transformations. Additionally, TAHOMA's use of cascades of multiple specialized CNNs (rather than a single CNN) creates a design space of millions of possible specialized cascades and a much richer set of Pareto-optimal choices. Further, TAHOMA can quickly evaluate its cascades in deployment-specific settings, determining the cascades optimal for the deployment's current operating characteristics. This could be particularly useful in dynamic scenarios, such as with networked cameras with varying bandwidth constraints. VideoStorm [157] is a system that automatically adjusts parameters like resolution and frame rate to maximize output quality of computer vision algorithms, based on computing resource availability in large clusters.

Image classification — Deep CNNs have revolutionized the field of computer vision, leading to breakthroughs in image classification and detection capabilities in recent years. The ImageNet competition [127] has had as one of its core challenges a 1000-category, million image classification task. A deep CNN was first used in 2012 and greatly reduced the best error rate [59], and the error rate has dropped in the subsequent years to near or better than human performance on the same task (e.g, [56,62,65]). Deep CNNs can now facilitate semantic content extraction from images and videos, supporting the development of large scale visual analytics databases. Deep CNNs are too slow, however, in their current incarnations to be applied at scale. Research in speeding up

CNNs has seen some recent interest (e.g., SqueezeNet [64] and MobileNets [**?**]). Tahoma is essentially classifier-agnostic, so these and future networks can be incorporated into our cascading techniques.

Classifier cascades — Our work leans heavily on previous research into classifier cascades. One of the first works to use the technique was Gama's Cascade Generalization [47], in which data was classified by a cascade of classifiers that each subsequently added additional features to the input vector. Another famous early use was in the Viola-Jones face detector [147], which used a series of classifiers based on simple image features to detect faces in photographs. More recently, cascades have been used to accelerate the slow inference speeds of deep neural networks [25,72,142]. Other work has used cascades to improve accuracy, rather than speed [146]. Chen et al. [26] used classifier cascades to reduce the cost of feature extraction for text. Our system, rather, uses feature extraction (in the form of input transformations) in cascades to create smaller, faster models. We show how classifier cascade-driven query optimization can be exploited in a visual data analytics system.

Model selection and management — Tahoma's general method of generating a large number of model (and cascade) variations and selecting among those can be seen as a form of model selection [84]. A number of recent data systems have been proposed or developed to assist in machine learning model creation, management, and deployment [17,31,43,79]. These systems have been developed with general machine learning tasks in mind, while Tahoma focuses directly on performing queries and analytics over visual data. Because of this tight focus, we can take advantage of characteristics of CNN architectures and properties of image and video data. Our model generation and selection methods could be integrated with systems like Velox [31] or MacroBase [17], whose functionality in serving models and handling large amounts of fast data would be complementary to our work.

4.10 Conclusion and Future Work

In this paper, we have presented a method of accelerating content extraction from large corpora of visual data, with the aim of supporting visual analytics queries. We showed how constructing a huge number of classifier cascades from a wide variety of CNN-based classification models can yield large speedups in content extraction. Our cascades applied input transformations on the raw image corpus to further reduce classification costs. We also demonstrated the necessity of being deployment scenario-aware—that is, considering costs such as those for data loading and image transformation—when evaluating the accuracy and throughput tradeoffs of classifiers.

While this paper primarily focused on image classification tasks, it is just the beginning of series of work that will develop a data analytics for visual data that will take full advantage of spatio-temoporal locality present in adjacent video frames to further accelerate content extraction. We hope to include new state-of-the-art computer vision methods to extract more complex data, which will then allow the processing of complex analytical queries over video.

Chapter 5

Column and Table Embeddings for Data Integration Tasks

5.1 Introduction

Myriad tasks in data management can benefit from the application of machine learning models: index construction [78], query optimization [74, 100], and data integration [38] have all recently seen the application of deep learning to their domains. A problem with applying deep learning (and most other machine learning methods) to the data management domain is how to best represent the data being input into the models. Relational tables, for example, do not naturally translate into a typical vector used as machine learning input. Our goal with this work is to demonstrate suitable representations for relational tables for deep learning for data management tasks, with a focus on data integration.

One advantage of deep learning over other, simpler machine learning methods is that input features often do not have to be manually created; raw data (e.g., pixel values for an image) can be input directly into the network, and given enough training data, the complex architecture of the deep network can learn the features (essentially, as encoded by the weights of the network) on its own.

Indeed, this works well with raw data that is represented by a dense vector that encodes some sort of relationships between each element, as with the pixel values of images: an image is represented by a vector where each member is a consecutive pixel (or color channel of a pixel for color images where each pixel consists of a red, green, and blue intensity value). In an image, there is typically some sort of relationship between adjacent pixels, and the deep network can learn higher order features—edges and corners, perhaps—from those simple building blocks, given a large enough training set.

In other problem domains, such as natural language processing (NLP), however, input data might not be naturally represented by a dense vector.

In many NLP tasks, the input to a model is a word or series of words that are part of a vocabulary. A common way to represent input like this is with a *one-hot* encoding, where the input is a vector with the length of the size of the vocabulary, with each element representing a word. The element for a given input word has the value 1, while the rest of elements have the value 0. This has a major shortcoming, though: this representation is a very long, extremely sparse vector, with little or no semantic relationships between elements. A very large training set would be needed to learn such relationships, and for many tasks, such a dataset (at least a labeled one) does not exist and collecting such data would incur great expense.

Thankfully, researchers have developed a technique to encode members of a vocabulary as a dense vector, or an *embedding* in a vector space. Perhaps one of the most famous such embedding techniques is the now classic word2vec [104, 105]. Others include the successful GloVe embeddings [115] and the more recent fastText [20] and ELMo [116] embeddings. In all of the cases, semantic relationships between words (or even parts of words, in the case of fastText and ELMo) are learned and encoded in the vector representations of each word, based on models trained over large corpuses of text. To perhaps oversimplify, these models learn the relationships between words based on their context in the training corpus, such that the resulting vector encodings for two semantically similar words will be relatively close to one another in the vector space, compared to two words that have less semantic relatedness to one another. For example, Mikolov et al. demonstrate that word2vec associates "Beijing" with "China" much more so than with "Portugal" [105].

Data management tasks using deep learning are similar in some aspects to NLP, especially in the data integration domain. Consider schema matching, where two tables are compared to see if their schemas match sufficiently to perform an operation between the two tables like a union or a join. In schema matching, the semantic meaning of columns can be compared to determine match suitability. This has been done by matching the contents of a column with entities in a knowledge base (e.g., YAGO or DBPedia) and then using the majority class of the matched entities as the semantic type of the column [18, 36]. When a table corpus is derived from heterogeneous sources of varying quality, however, it can be rare to match table values with a knowledge base (Lemhberg and Bizer [90] found that only roughly 5% of the 90 million tables in the Web Data Commons (WDC) 2015 web tables corpus [91] contained at least one entity that could be found in DBPedia). A better method to represent the semantic relationships between relational data—like those captured by word embeddings in the NLP domain—-would at the least allow for better coverage for real-world datasets.

Technical challenge — Relational data is central to many tasks that benefit from machine learning. Performance on these tasks would be improved by finding a way to represent attribute values, columns, and entire tables that is suitable for input into machine learning models, while capturing semantic relationships between elements in the corpus. This project aims to develop an effective way to embed relational data into a vector space that preserves relationships between table elements, similar to word embedding methods like word2vec. A further challenge is that within a large web tables corpus, the variety of elements within the tables is huge and of a wide range of sizes, making creating a definitive vocabulary of reasonable size difficult.

Our approach — Our approach to embedding relation data in a vector space is to begin with a existing method that builds word embeddings based on letter n-grams within words to allow for words not seen in the training corpus to still be represented in the vector space. We train fastText [20] on a large subset of a web tables corpus, where each "word" is an element in a table, whose context is the column in which it appears. We then aggregate these elements by finding the mean for all elements in a column to create a column embedding–similar to a bag-of-words method of creating sentence embeddings from word embeddings. Likewise, a table embedding is created by finding the mean of all elements of the table. Each of these embeddings has a use in various data management tasks, which we describe in Section 5.3.

Organization — This chapter is organized as follows:

- We discuss relevant background material, including word embeddings and the neural network architectures used in our models, in Section 5.2.
- We describe our target data management tasks—column labeling, table titling, key column identification, and schema matching—in Section 5.3.
- We detail our methods in Section 5.4.
- We prototype our methods in a system called Grover and show our experimental results in Section 5.5.

We follow with a discussion of related work in Section 5.5.6.

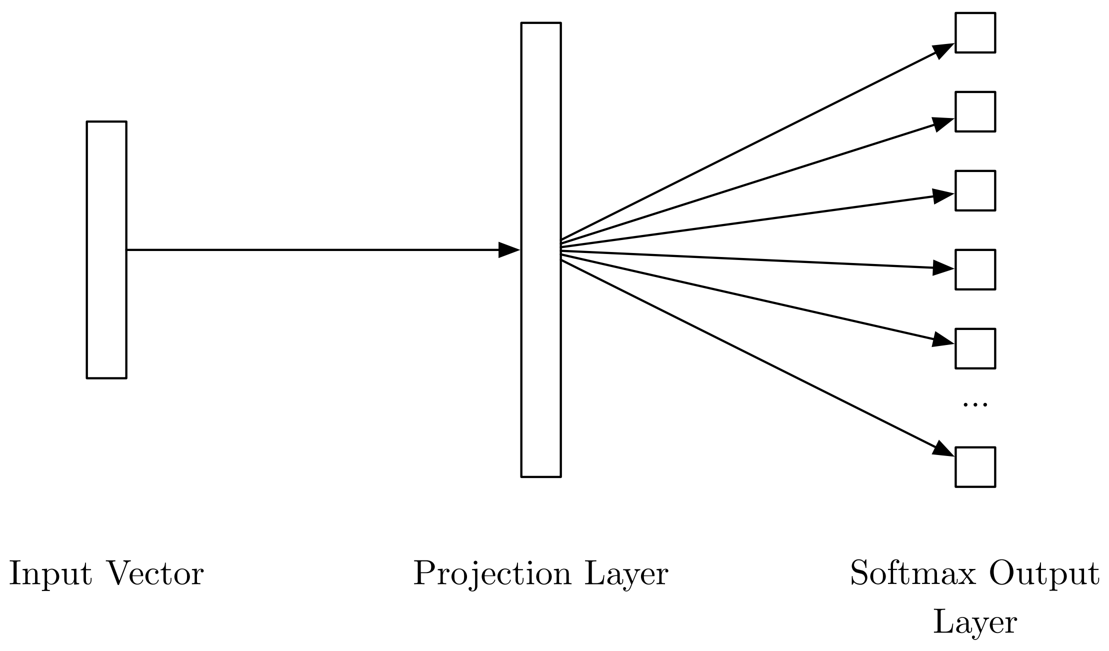


Figure 5.1: The basic skip-gram model used by word2vec is a neural network with one hidden layer. The input vector is a one-hot vector describing the vocabulary used by the model. The projection layer has n nodes with no activation functions. The output layer has nodes representing the vocabulary members, with softmax activations to predict the context words. The output of the projection layer is what is used as the word embedding vector.

5.2 Background

In this section, we will provide some background on methods and algorithms that influence this work.

5.2.1 Word and Sentence Embeddings

Word and sentence embeddings have become a key component of deep learning-based NLP systems. The most successful methods are sometimes called *universal embeddings*, because they are pre-trained on a large corpus of text and act as a form of transfer learning, where relationships between words or sentences are learned from that large corpus and then used as a starting point for an NLP system. Embeddings used in this way can greatly improve the performance of a downstream application, especially if that application has a smaller training set that would preclude learning some of those pre-learned relationships.

The word2vec [104,105] embedding method had a huge impact on the field, producing dense word vectors for text corpora that still rival more recent methods. The method uses a skip-gram model[1] to produce embeddings. The skip-gram model (illustrated in Figure 5.1) essentially tries to predict a word that appears in context with a given word. That is, given two words that appear close to one another in the text (*first*, *second*), the model will be trained to predict *second* when given *first* as input. For example, given the text “a long time ago in a galaxy far far away” and a context window of size 2, for the word “ago”, the model will be trained with the word pairs (*ago*, *long*), (*ago*, *time*), (*ago*, *in*), and (*ago*, *and*). A large text corpus produces a very large number of these pairs, which can lead to long training times. Mikolov et al. introduced number of efficiency tricks to train a word2vec embedding model quickly [104].

Several years later, a project which might be considered a successor to word2vec was introduced: fastText [20]. This method, rather than using a word as input and output, the model uses the sum of vectors representing character n-grams derived from a word as input to a skip-gram model. Thus, for the word “galaxy”, 3-length n-grams would be *gal*, *ala*, *lax*, and *axy*, for the input word “galaxy”. Vectors representing these n-grams are summed and after training, this summed vector for the input word is used as a word embedding. This n-gram-level encoding allows for the model to represent words that may not appear in the original corpus, with the assumption that words that share n-grams have some amount of similarity to one another.

Many other methods of producing word embeddings have been developed, including GloVe [115], which is based on global co-occurance counts of words in a corpus, and ELMo [116], which uses a more sophisticated LSTM-based model to develop a language model over a text corpus. In this paper, we use fastText as the basis for our embeddings, because of the speed in training of the model and high quality of the results.

There has also been much interest in creating *sentence embeddings*, which embed an entire sentence in a dense vector space. While a number of effective sophisticated methods have been developed to create sentence embeddings, it has been found that a surprisingly effective yet simple method is to take a bag-of-words approach and average a sentence’s word vectors [14,30,149]. Because of this simplicity and effectiveness, we use this method to create embeddings for columns and tables.

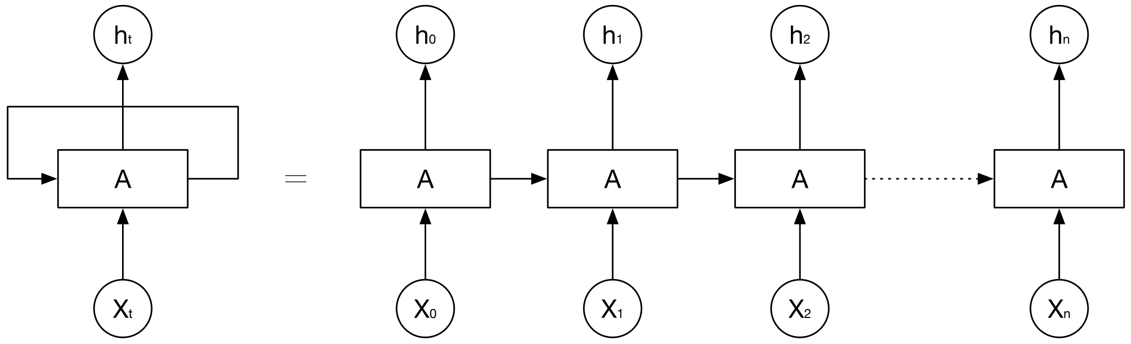


Figure 5.2: A recurrent neural network, on the left. An RNN can be conceptually "unrolled," as is shown on the right side of the diagram.

5.2.2 Recurrent Neural Networks

A column in a data table can be seen as a sequence of values, and as such, a natural model for making predictions with a data column as input is a recurrent neural network (RNN) [40, 97]. An RNN accepts as input a sequence of values $(X_0, X_1, \ldots, X_n)$, and as shown in Figure 5.2, when each value X_t is processed as input, the RNN's previous hidden state is also provided as input. This allows the model to essentially have a memory of previous inputs when providing the output for all subsequent outputs.

One particular architecture of RNNs is the long short term memory, or LSTM [60]. LSTMs were designed to combat certain deficiencies found in early RNNs in handling long-term dependencies (e.g., trying to predict the last word in long sentence when important context is given early in the sentence). LSTMs maintain a "cell state" in addition to a typical hidden state. As a sequence is processed, the LSTM can forget or modify this cell state, depending on what it has seen previously. This flexibility in its memory allows it to maintain long-term dependencies, while ignoring information that is no longer important to the remainder of the sequence. We use LSTMs in this paper as our RNN architecture.

5.3 Tasks

There are a number of important data integration tasks that can be improved by using table embeddings. In this project, we focus on four particular tasks to illustrate the utility of column and table embeddings with our Grover prototype:

[1]They also present a continuous bag of words (CBOW) model, but the skip-gram method has tended to be the better of the two and is typically the model used.

1. **Column Labeling** – The aim of this task is, when given a relational table without labels for its columns, determine the best labels for each column.

   Previous methods [18, 36] have used databases derived from an ontology such as Google Knowledge Graph or YAGO to look up the entity each cell corresponds to, and then to look up the class of each entity. If a class is 50% or more of a column, that class is used as the label, otherwise, the highest scoring class is used.

2. **Table Titling** – Perhaps less important when performing relational operations on data tables than when presenting them in human-readable form, determining appropriate titles for tables is nonetheless important in providing proper context to data users.

   Previous methods [91] extract titles for web tables from caption text surrounding the table on its host web page. In cases where no suitable text can be found, titles cannot be provided for the data.

3. **Key Column Identification** – Identifying the key column of a table can be very important for later operations, such as table joins or entity matching. The Web Data Commons 2015 Web Table Corpus [91] defines a key column of a table as the column that contains the names of the entities represented by each row. For that corpus, key columns were identified by determining if a column had a high number of unique values that were strings of relatively short length.

4. **Schema Matching** – Schema matching involves determining whether two tables have compatible schemas, such that operations like JOIN or UNION can be successfully performed between them. Many schema matching techniques have been developed over the past decades; we will be concentrating on those useful for performing table unions [108] and table joins over a corpus of web tables [24, 39, 91].

Other tasks, that we save for future work include data augmentation [23], where missing data in a web table can be extracted from surround text on the containing web page, and knowledge base matching [132, 159], where web tables are matched to existing knowledge bases to facilitate downstream applications like knowledge base completion or table extension.

5.4 Algorithms and Methods

In this section, we detail the methods used to create various embeddings for relational data, as well as the machine learning architectures used to employ these embeddings towards the tasks described in Section 5.3.

5.4.1 Column-value Embedding

In order to use a single column value as an input into a neural network, we first created a vector representation of the value, similar to the use of word embeddings in NLP applications. Because considerable work has been done in this area in the NPL domain, we used an existing method, fastText [20], to create these vector representations. As we discussed in Section 5.2, fastText is based on the continuous skip-gram model used in word2vec [105], but instead of limiting the model to the words appearing in the training corpus, fastText builds its model using character n-grams, allowing for vectors to be constructed for out-of-dictionary words. This is important when constructing vector representations for values found in a huge corpus of general web tables, since the contents of individual table cells are incredibly varied and are often multi-word phrases (and thus, creating a comprehensive dictionary is infeasible).

5.4.2 Column Labeling

For the column labeling task, we have tried several methods, with experimental results reported in Section 5.5.2.

LSTM on a Single Column

The architecture for this first method comprises an LSTM network, whose input is a sequence of column value vectors and whose output is a vector of the possible column labels. Though a prediction is generated after each element of the sequence is processed by the LSTM, only the output from the final element is used as the predicted label.

Each column is treated independently in this method, which limits its performance. Consider the case of a column containing a series of integers: with only the integers as a source of information, a model may be unable to determine if the values represent temperatures, ages, or strike outs. We find that in cases like these, the model was able to discern *some* column labels based on apparent properties of the column, like value range[2]: for example, a column with values ranging from 20 to 40 was correctly labeled "low temp" while one with values between 60 and 80 was correctly labeled "high temp". (Note that prior to any processing, we normalized the column labels in our training set by converting all to lower case. We did not combine similar labels—both "low temp" and "lo temp" are present in our dataset, for example. Combining semantically similar labels such as these is a standard preprocessing step that could be done to further improve accuracy if these methods were to be deployed in a real world system.)

Full-table LSTM Label Predictions

To overcome the problems created by treating columns independently, we can train a model that uses *all* of the columns in a table as input. Like we can treat a column of data as a sequence for input into an LSTM model, we can think of a table as a sequence of columns that can likewise be used as a model's input. Of course, a column of values cannot be directly used as a single input, so we must create a suitable vector representation, or *column embedding*.

Column Averaging — The idea of a column embedding is similar to that of a sentence embedding in NLP applications. Sentence embedding creates a vector representation of a sentence, which can then be used as input into a classifier or other model. As we discussed in Section 5.2, simply averaging the results of word embeddings has been found to be surprisingly effective [14, 69, 150]. Extending this to data tables, we can create column embeddings by averaging the value embeddings within the column. These column embeddings can then be used for our column labeling task by using the columns embeddings for the table as an input sequence to an LSTM model, with the output being the sequence of labels. Given that we have the entire sequence on hand, and that we are predicting labels for each member of the input sequence, we can use a bidirectional LSTM model [131]. This type of model essentially combines two LSTM networks, one which looks at the sequence beginning-to-end, and the other from end-to-beginning. The outputs of both directional LSTMs are concatenated and used as input into a final fully connected linear layer with softmax outputs to predict the sequence of column labels.

LSTM Output Vectors — Beyond simply averaging the values in a column to produce a column embedding, we can create column embeddings by using the output of intermediate layers of a neural network trained on a related task, similar to the common practice of using image embeddings extracted from deep networks trained to classify

[2]Properties like these were not explicitly provided to the model; all features were automatically learned by the model from the input vectors.

ImageNet images [83]. Given that we have already created an LSTM model to predict column labels for a single column, then, we can use the final output of the LSTM layer as a vector representation of the entire column. The architecture for a full table column labeling network using the vectors is the same as the previously described network using column averaging embeddings. We test both of these methods for column embeddings in Section 5.5.2.

5.4.3 Table Titling

When predicting an appropriate title for a data table, the entire table should be considered as input to the model; thus, for this task we based our methods on the full-table LSTM models used for column labeling. Further, we focused on using the column-averaging method to create column embeddings over using LSTM output vectors due to the much faster training speed (essentially training a single LSTM versus training one per column plus one for the table) and comparable accuracy (see Section 5.5.2) between the two methods. One difference between this task and column labeling is that there is just a single prediction for the entire input sequence of column embeddings. Thus, we limited evaluation to the model to the output of the model for the final element of the sequence.

5.4.4 Key Column Identification

The key column of a web table is the column which contains the name of the entity represented by a row of data [91]. Any column could potentially be the key column; however, in a random sample of 100,000 web tables in the Web Data Commons 2015 Web Table Corpus, nearly 75% of the tables had no key column or had the first column as the key column. (A further 15% had the second column as the key column.) For the purposes demonstrating the effectiveness of column embeddings in this paper, then, we simplified the task from predicting *which* column of a table was the key column to one of predicting whether or not the first column of the table was a key column. We used the same column-averaging embedding LSTM architecture as the column labeling and table titling tasks, with the output of the model being a boolean value representing key column status of the table's first column. We only used the model's output for the final element of the input sequence for this model's predicted value.

5.4.5 Schema Matching

An important task applicable to many data integration tasks is schema matching, or finding tables (typically with unlabeled schemas) within a corpus that have matching or overlapping schemas and are thus suitable for join or union operations. In this work, we will show that the column embeddings derived from fastText can be leveraged to find schema-matched tables within our web table corpus.

Nearest Neighbors

Because our column embeddings are vectors embedded in a multidimensional space, we can compute the distance between, say, a query column and the rest of the columns in our corpus. A query column in this case is one which we wish to find similar columns in the corpus that belong to tables that are candidates for schema matching. The ability to quickly find candidate tables is important, since a web tables corpus can be huge and running expensive schema matching algorithms on the entire data set can be infeasible. Finding similar columns (and thus candidate tables) based simply on column embeddings can be problematic, however, especially if the column contains simply a set of integers or other values that are difficult to semantically understand without additional information like column headers or the column's context within its table.

Like column embeddings can be effectively created by simply averaging the FastText embeddings of its member values, we can create table embeddings by averaging the table's column embeddings. This single vector embeds the table into n-dimensional space, allowing for nearest neighbor search to be performed to find candidate tables for schema matching to a query table. We show in Section 5.5.5 that while both column embeddings and table embeddings generate suitable candidate tables for schema matching, table embeddings are much more effective.

We found that some tables do not make semantic sense for joining or unions, even if they share nearly all the same column types. For example, one query table contained baseball batting statistics, but each row was an aggregation over a particular batting count (e.g., 1 ball and 1 strike, or 2 balls and 0 strikes). It does not make sense to join or union this table with the vast majority of baseball statistics tables in our corpus, which contain stats for individual players aggregated over entire seasons, even though they match on all but one column (i.e., batting count versus name).

Schema Matching Classifier

While schema matching itself is not the focus of this work, we created an LSTM-based schema matching classifier to predict whether or not a pair of tables are suitable for a schema matching operation, such as a table union. Our model consists of two LSTMs, each based on the column average LSTM used for column labeling. The two LSTMs—one for each table—take as input a sequence of column embeddings and the final output of each LSTM is concatenated together to form the input of a sigmoid later that outputs a boolean prediction of whether or not the two tables match. Note that this method does not include any schema information and solely relies on the embeddings created from the table contents. A classifier like this would likely be improved by including such information, but we leave implementing an improved classifier to future work. Such a classifier may include embeddings created for the column labels themselves to encode schema information in a suitable format.

5.5 Experiments

We performed experiments on several different tasks to demonstrate the feasibility of applying neural network-style embeddings to data integration tasks. Experiments were performed using our GROVER prototype implemented using PyTorch [113] and executed on NVidia GTX 1080 GPUs.

5.5.1 Data Set

For our experiments, we used data from the Web Data Commons (WDC) 2015 Web Table Corpus [91]. To build fastText [20] vectors for column values, one million tables were randomly sampled from the corpus, with an additional constraint that only 1000 were selected from each web site domain. (Certain domains, such as sports statistics websites, would be overly represented, leading the model to be biased towards those types of tables.)

For training the various neural network models (with a slight variation for our predicting table titles experiments, described in Section 5.5.3), an additional 100,000 tables were randomly sampled, with a limit of 100 tables per web domain. From these 100,000 tables, 10% were extracted randomly to build a validation set used to configure training parameters, and 10% were held out at random to build a separate test set, which was used to calculate our reported results.

5.5.2 Column Labeling

We tested several different methods for labeling columns, as described in Section 5.4.

Single Column LSTM

The single column LSTM treated each column independently, using only that column's values to predict the column's label, as described in Section 5.4.2. Each value was first converted to a 64-element fastText vector, and each column in the table was treated as an independent sequence of these vectors. The set of column labels was minimally preprocessed by converting each label to lower case. Labels that appeared less than 10 times were replaced with an `OTHER` token, which represented 17% of the final column data set. There were 4475 labels in the resulting data set.

Each column in the data set was represented by a (*sequence*, *label*) pair, where *sequence* was used as the input to the neural network and *label* was the ground truth label. Model parameters such as number and size of LSTM hidden layers were chosen by performing a grid search over reasonable values and selecting the best performing values as evaluated using the validation data set. For these experiments, the LSTM had a single hidden layer with 256 features. The output of the LSTM was input into a fully connected 4475-node linear layer with a softmax output to predict the *label* associated with the input *sequence*. The network was trained using a batch size of 32 over 200 epochs, where each epoch took roughly 5 minutes to complete. The model with the lowest loss on the validation set was selected as the final trained model.

Results — As shown in Table 5.1, the trained model for the single column LSTM (SingleCol) task had an accuracy of 45.5% over the test data set. Table 5.1 also gives the results for up to top-k accuracy (up to $k = 5$), where the model's prediction is considered correct if the ground truth label appears in the top k ranked outputs of the model. (A top-k interpretation of the output would represent a realistic use case, where a user of model was presented with k possible choices for labeling a column.) For a top-5 result, the SingleCol model was 70.7% accurate.

Column Averaging LSTM

We tested two column averaging LSTM methods, one with a standard LSTM and the other with a bidirectional LSTM. Both otherwise used the same architecture. The fastText vectors for each column in the dataset were averaged using a simple unweighted mean, giving a single 64-element embedding representing each column. Each table in the

	SingleCol	ColAvgLSTM	ColAvgBiLSTM	LSTMOut
Top 1	45.5%	70.0%	76.0%	74.3%
Top 2	57.5%	78.5%	82.9%	81.1%
Top 3	63.6%	82.4%	85.7%	83.7%
Top 4	67.8%	84.5%	87.3%	85.1%
Top 5	70.7%	85.9%	88.5%	86.1%

Table 5.1: Top-k accuracy for column labeling on test set.

dataset was represented by a (*column sequence*, *label sequence*) pair, where the former was the sequence of column embeddings for a single table, while the latter is the sequence of labels for that table. Labels were preprocessed in the same way as for the single column LSTM method, and the training regime was the same, though each epoch for these methods took between 20 and 30 seconds, due the much smaller input sequences. The trained models produced an output for each element of the column sequence, with each output predicting the label for the corresponding column sequence.

Results — Table 5.1 shows results for both the standard LSTM (ColAvgLSTM) and bidirectional LSTM (ColAvgBiLSTM) methods. As expected the ColAvgBiLSTM method performed better than ColAvgLSTM, since the ColAvgLSTM model more often predicted incorrect labels for columns early in the sequence. The bidirectional LSTM gave those columns additional information for predictions by also processing the columns in reverse order, improving the accuracy on those columns. Both column averaging methods far outperformed the single column method, which was an expected result due to the additional context in which the predictions were made.

LSTM Output Embeddings

For LSTM output embeddings, rather than averaging the column entry embeddings, we started with the same architecture as for the single column LSTM method above. After training that model, the final output of the LSTM node is stored as a column embedding, rather than used as input into a softmax layer to predict the column labels. These embeddings are then used in a bidirectional LSTM as is done in the column averaging LSTM method. (A standard single directional LSTM was not tested, due to the obvious advantage of the bidirectional model shown for the column averaging LSTMs.) This model was trained with the same regime as previous methods, with each epoch taking roughly 5 minutes to complete.

Results — The accuracy results for the LSTM output embeddings (LSTMOut) were slightly less than for ColAvgBiLSTM (74.3% vs. 76.0%). This deficiency, plus the much longer training time (5 minutes vs. 30 seconds per epoch), leads to the conclusion that of these tested methods, the column averaging bidirectional LSTM method is superior.

Error Analysis

Upon inspection, many of the errors in label prediction for all of the models were simply grading errors. That is, the model predicted a reasonable label for the column, but that predicted label did not *exactly* match the actual label. For example, a weather table had a column labeled “lo temp”, while the model predicted “low temp”. In other cases, suitable synonyms for actual labels were predicted, but were reported as being incorrect. As previously stated, we performed very little preprocessing and normalization on the labels; if a thorough mapping of equivalent labels was performed, the ColAvgBiLSTM model could have reached 84% top-1 accuracy, based on an analysis of 100 randomly sampled tables in the test set. Likewise, the ColAvgLSTM model would have reached 79% top-1 accuracy, LSTMOut reached 81% top-1 accuracy, and SingleCol reached 65% top-1 accuracy.

5.5.3 Predicting Table Titles

In addition to labeling columns, column embeddings can be used to label entire tables with appropriate titles, as well. In this section, we describe our experimental results using a similar LSTM architecture with column embedding inputs to predict table titles.

Data Set

For these experiments, we again used the WDC 2015 Web Table Corpus and extracted 100,000 tables that had titles present in their corpus entries. Like the previous experiment, we limited each domain to a maximum of 100 tables. With this extracted data set, we removed tables whose titles did not appear 10 or more times (after performing simple normalization on the titles), leaving a final dataset with 57,662 tables and 1,115 different titles. We split this final data set into three parts, with 90% of the tables in the training set and 10% in each of the validation and test sets. The tables were partitioned into these subsets such that tables from a particular web domain would only be present in one subset. Domains were randomly assigned to the subsets.

	ColAvgLSTM	ColAvgBiLSTM
Top 1	69.0%	69.2%
Top 2	72.7%	76.4%
Top 3	75.9%	80.6%
Top 4	77.2%	83.3%
Top 5	77.6%	85.1%

Table 5.2: Top-k accuracy for predicting table titles on test set.

Model Architecture

Due to the success of the LSTM and BiLSTM column averaging approaches in the previous experiments, we used these same methods to predict the titles for the tables in the above described data set. For each column, the fastText vector for each value was averaged as before, and the 64-element column vectors was used as input to standard LSTM and BiLSTM models. The output of each model was a produced by a softmax layer with 1,115 elements, one for each potential column label. Only the output given by the model for the final member of the sequence was evaluated as the predicted title.

Results and Discussion

Table 5.2 shows the top-5 accuracies for the table title prediction task using column averaging embeddings for both the LSTM and BiLSTM models. While both the LSTM and BiLSTM models provided similar accuracy for the top prediction (69.0% and 69.2%, respectively), the BiLSTM provided much higher top-5 accuracy at 85.1% versus 77.6% for the LSTM model.

In cases where the model provided an incorrect predicted title for a table, the predicted title was often close semantically. For example, for a schedule of television programs, the correct title was a certain date and the predicted title was a different date. Or, a table had a specific title ("women's shirts") and a more general title was predicted ("women's apparel"). For many of these error cases, a more sophisticated title normalization scheme would provide a significant increase in model accuracy. An analysis of 100 randomly sampled tables in the test set shows that the BiLSTM model would have reached 81% top-1 accuracy if semantically similar titles were normalized into a single equivalent title. The LSTM model would have reached 75% accuracy in the same situation.

	ColAvgLSTM	ColAvgBiLSTM
Accuracy	80.5%	81.4%

Table 5.3: Accuracy when predicting whether a table's first column is the key column.

5.5.4 Key Column Identification

In this section, we demonstrate how column embeddings can be used to identify the key column in a web table—a key step in efficiently using web tables in a relational database.

Data Set

The WDC Web Table Corpus metadata includes a boolean flag that indicates whether or not a table has a key column and if so, also identifies which column it is. In the 100,000 table data set used in the column labeling experiments (Section 5.5.2), the first column of nearly 40% of the tables was the key column, and nearly 35% of the tables had no key column at all. Since this covered a large portion of the dataset, we simplified the task from identifying *which* column was a key column to one of determining if the first column was the key column. We limited the dataset to those tables with no key column and those where the first column was the key column, resulting in a total of 73,305 tables. As in previous experiments, we split this data set into three parts, with 90% of the tables in the training set and 10% in each of the validation and test sets. The tables were partitioned into these subsets such that tables from a particular web domain would only be present in one subset. Domains were randomly assigned to the subsets.

Model Architecture

Like the previous experiments, we used column-average embeddings as input into both LSTM and BiLSTM models and used the same training regime. The output of the model was a softmax layer predicting true (the first column is the key column) or false (the first column is not the key column). Only the output for the final element of the input sequence is considered when evaluating the output of the model.

Results and Discussion

Table 5.3 shows the model accuracies for predicting whether or not the first column of a table is the table's key column, for both the LSTM and BiLSTM models. The LSTM and BiLSTM models both provided similar accuracy levels for this task, at 80.5% and 81.4%, respectively.

5.5.5 Schema Matching

In this section, we discuss our experiments on schema matching using embeddings. We first identified candidates using a nearest neighbor search with full table embeddings, and then developed a simple classifier to predict whether the candidate was suitable for a table union operation.

Labeling Data

Training data for our classifier was identified by comparing two candidate tables by hand and determining if the two shared at least one semantically identical column. For example, if one table listed basketball players along with their height, weight, and position, while the other listed basketball player names and statistics, the pair would be labelled as a match. A union of the two tables would clearly have empty values where the columns do not overlap, but other downstream data integration operations could be used to fill those spots. If the pair included the basketball player stats table and another with stats of basketball teams, it would not be a match, since, even if columns have the same label (e.g., "points"), the values are arguably semantically different and do not belong in the same table.

Data Set

The schema matching task uses a small subset of full dataset used for column labeling. Tables were chosen at random and examined to determine if they were suitable as query tables for a table union operation. A considerable number of tables in the overall dataset are calendars, tables that describe a single store inventory items (where the first column lists attributes and a second lists values—these would be suitable for use, perhaps, after a transposition operation, but that is out of scope of this experiment), and other similarly unusable tables.

Candidate Search

Candidates were selected by doing a nearest neighbor search in the table embedding space for a given query table. Table embeddings were created by averaging the column embeddings for each table. The top 100 nearest tables in terms of euclidean distance were considered candidates for the table union task, filtering out duplicate tables and any tables from the same web domain as the query table (otherwise, the task proved simple and unrealistic, as the top 100 was primarily from the same domain for the

Query table	Matches
patents	99
baseball	70
keywords	45
soccer	37
basketball-roster	27
hockey	27
tennis	25
football-roster	7
basketball-stats	1
aussie-football	0

Table 5.4: Number of schema matching tables within the top 100 nearest neighbors in the table embedding vector space.

majority of query tables). Due to the amount of labor involved with evaluating the pairs, only candidates 10 query tables (chosen at random from suitable tables) were examined.

Results — The number of matches with the top 100 candidates are shown in Table 5.4. Note that no explicit schema information was used in this search, just the averaged table embeddings. The candidate searches resulted in nearly 99% schema matches to 0% matches. The patents table was perhaps surprisingly matchable, given the large amount of varied text present in patent descriptions and titles. Many of the sport-related tables—which dominated the randomly selected tables—had mediocre to fair results, since the table values were primarily numeric, and there were matches from other sports interspersed throughout the candidates. The basketball-stats table was particularly bad in this regard, but this occurred because the query table had a very high proportion of zeros listed in the statistics, making it difficult to differentiate the table from other tables that had mostly zeroes for entries. The aussie-football table was from one of the very few domains in the corpus dedicated to Australian rules football, so filtering out results from that same domain severely limited the potential matches.

Classifier

As we described in Section 5.4.5, our schema match classifier consisted of two LSTMs: each took in a sequence column-average embeddings—one for the query table and one for the candidate table—and the final output of each LSTM was concatenated together and provided to a sigmoid layer that predicted whether there was a schema match or not. Testing was limited by the small amount of labeled training data, as we used the 1000 candidate pairs (top 100 for 10 query tables) generated by the nearest neighbor search described in the previous section. We ran the training and test procedure 10 times, using 9 of the query table sets to train and using the remaining one as the test set for evaluation of the model's accuracy. This testing method resulted in 68% accuracy, averaged over all runs.

While this accuracy level is not incredibly high (and, indeed, simply predicting "no match" for every pair would have resulted in somewhat nearly as accurate of a result), this classifier model did not use any schema information to make these predictions. Other models can easily be envisioned that incorporate schema information; future work should include creating embeddings for the column labels, allowing schema information to easily be included in models like these.

5.5.6 Related Work

The work in this chapter draws on work from several areas in both the machine learning and database communities.

Embeddings — We discussed related work in word and sentence embeddings in Section 5.2. The use of embeddings for transfer learning-type tasks is not limited to NLP. Many computer vision projects use pre-trained deep network architectures as a starting point for novel tasks. One popular pre-training corpus is ImageNet [128], a collection of 1 million photos labeled into 1,000 classes. Deep network architectures like AlexNet [83] or ResNet [56] are trained on the ImageNet classification task, and then reused in other computer vision tasks, often by removing the final classification layer and replacing it with one suitable for the new task. The model is then fine-tuned on a training set for the new task. The output of the model after the final classification layer is removed is analogous to word embeddings like word2vec [104, 105].

Data Integration — There is a rich history of data integration research. The area most related to our work is that using web tables as a data corpus. Cafarella et al. [24] performed some of the earliest web tables work, including some simple column labeling and schema matching tasks. Further tasks were performed in the follow-on Octopus project [23]. Other web tables data integration work was done in conjunction with the Web Data Common's Web Table Corpora [91]. Much of this work was done to combine web table data with knowledge bases (e.g, [90]). The Infogather [153, 158] system also performed data integration tasks over web tables; in particular, the project aimed to fill in missing data or extend existing tables using tables extracted from the web. None of these projects used deep learning or embeddings as we have done in this work. Some embedding techniques have been used in data quality tasks, like entity linking. The IDEL system [73] creates embeddings for entities in relational tables by concatenating vector representations for each attribute, attribute type, and foreign key in a tuple, which is used in a model along with text embeddings for entity candidates being matched. This differs from our method, in that by using the character n-gram approach as in fastText, to create embeddings for each element in a web table, we do not have to know (and potentially do not concretely know types and foreign keys). Our methods also create column- and table-wise embeddings, rather than for individual tuples.

As for non-web tables data integration, a similar project to ours was that of Nargesian et al. [108], which used pre-generated (on natural language text) fastText embeddings to represent attributes in open data databases. They found that there were many attributes not representable, due to many attributes not being present in natural language corpora. We avoid that problem by training a fastText model directly on a large number of web tables, such that nearly all attributes in our training and test data sets can be represented.

5.6 Conclusion and Future Work

With our prototype Grover system, we have demonstrated that column and table embeddings can be created and successfully used in several data integration tasks. Column labeling, table titling, and key column identification on a web table corpus was accurately done using column embeddings as input into a bidirectional LSTM model. Schema matching candidate search was demonstrated using nearest neighbor search in a table embedding vector space, and a simple classifier that predicted a schema match between a pair of tables was also shown to have fair success.

For future work, we would like to first test the embeddings described in this work in more complicated data integration tasks, including knowledge base completion and data augmentation. Testing on additional datasets would also further show the efficacy of using column and table embeddings for these tasks. We would also like to investigate more sophisticated embedding creation methods; including additional context when training the embedding model, such as table schemas and row information, should lead to more effective embeddings. Finally, we would like to provide the public with downloadable pre-trained embedding models suitable for a wide range of data management tasks.

Chapter 6

Conclusion and Future Work

The work covered in this dissertation touches both the machine learning and the data management fields. We have shown that each community can learn from the other and borrow techniques to improve the results of their own work, especially when implementing real-world systems. There are a number of areas that our work can be extended or be used to influence directions of future work. We discuss these below.

6.1 Databases Built on Machine Learning

Our Zombie and Tahoma projects have a key similarity: relational-style data is extracted from instances of raw data by a machine learning model. While Zombie is focused primarily on speeding up the process of feature engineering, the core indexing method could easily be transferred to a more traditional data management domain. Consider a visual analytics database, as envisioned as a later stage of the Tahoma project. The classifier cascades developed as part of Tahoma speed up the inference step of content extraction, but still require processing all (or a large part) of an image corpus to answer the queries using content extracting binary predicates described in that work. A Zombie-style index over the images would allow queries to be answered much more quickly by processing less data. Combining the goals of these two projects would speed up query processing in two orthogonal ways: processing raw data faster and processing less raw data overall.

Interesting directions could be taken when designing such a system. It is not obvious how to best leverage a Zombie-style index when answering a query with multiple content extracting functions. Further complications come when a previous query has partially materialized the extracted content, leaving some raw data unprocessed. While a future query might be answered quickly using that already materialized data, depending on the downstream application of the query's answer, the unprocessed raw data might be the best choice for use. For example, if previous query execution had partially materialized a CONTAINS('stop sign') predicate in conjunction with a CONTAINS('bicycle') predicate, a future query for stop signs and pedestrians may not be best answered with the previously materialized stop sign images. Indeed, the distribution of the previously materialized data may differ enough from that of a future query that answering the query with the previous data may, for example, bias a machine learning model trained with that data. Thus, building an end-to-end system that employs a database to extract structured data from raw data objects for use in machine learning applications must take a holistic view when optimizing queries.

6.2 Embeddings for Relational Data

Our Grover project demonstrated that embeddings can be created for relational data to encode columns and tables in a dense vector space suitable for use in machine learning models. The applications shown in the project where fairly rudimentary, however, and relational data embeddings could be used in much more far ranging applications. We discussed some additional data integration tasks, such as knowledge base completion from web table data and data augmentation. Beyond data integration, query optimization seems to be a prime target. Part of query optimization is estimating table cardinalities for joins, and certainly some form of table embedding would be useful to deep learning-based models used to perform these estimations. One could also imagine a use for column or table embeddings, combined with more traditional word embeddings, when processing queries expressed in natural language rather than SQL. Future work with embeddings for relational data should include schema and metadata context to support applications such as these.

Bibliography

[1] Apache Hadoop. https://hadoop.apache.org/.

[2] Apache MADLib. https://madlib.apache.org/.

[3] Apache Spark. https://spark.apache.org/.

[4] Apache SystemML. https://systemml.apache.org/.

[5] A year without a byte. http://code.flickr.net/2017/01/05/a-year-without-a-byte/. Accessed: 2018-09-04.

[6] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[7] 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. In *EuroSys*, 2013.

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

[9] Michael R 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. In *CIDR*, 2013.

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

[11] Michael R. Anderson, Michael Cafarella, Yixing Jiang, Guan Wang, and Bochun Zhang. An integrated development environment for faster feature engineering. *Proceedings of the VLDB Endowment*, 7(13):1657–1660, 2014.

[12] Michael R Anderson, Michael Cafarella, German Ros, and Thomas F Wenisch. Physical representation-based predicate optimization for a visual analytics database. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1466–1477. IEEE, 2019.

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

[14] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. A simple but tough-to-beat baseline for sentence embeddings. In *ICLR*, 2017.

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

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

[17] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. Macrobase: Prioritizing attention in fast data. In *SIGMOD*, 2017.

[18] Sreeram Balakrishnan, Alon Halevy, Boulos Harb, Hongrae Lee, Jayant Madhavan, Afshin Rostamizadeh, Warren Shen, Kenneth Wilder, Fei Wu, and Cong Yu. Applying webtables in practice. In *CIDR*, 2015.

[19] Matthias Boehm, Michael W Dusenberry, Deron Eriksson, Alexandre V Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R Reiss, Prithviraj Sen, Arvind C Surve, et al. Systemml: Declarative machine learning on Spark. *Proceedings of the VLDB Endowment*, 9(13):1425–1436, 2016.

[20] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

[21] Sébastien Bubeck and Nicolo Cesa-Bianchi. Regret analysis of stochastic and non-stochastic multi-armed bandit problems. *Machine Learning*, 5(1):1–122, 2012.

[22] Eliot Van Buskirk. How the Netflix Prize Was Won. *Wired*, 2009.

[23] Michael J Cafarella, Alon Halevy, and Nodira Khoussainova. Data integration for the relational web. *Proceedings of the VLDB Endowment*, 2(1):1090–1101, 2009.

[24] Michael J Cafarella, Alon Halevy, Daisy Zhe Wang, Eugene Wu, and Yang Zhang. Webtables: exploring the power of tables on the web. *Proceedings of the VLDB Endowment*, 1(1):538–549, 2008.

[25] Zhaowei Cai, Mohammad Saberian, and Nuno Vasconcelos. Learning complexity-aware cascades for deep pedestrian detection. In *ICCV*, 2015.

[26] Minmin Chen, Zhixiang Xu, Kilian Weinberger, Olivier Chapelle, and Dor Kedem. Classifier cascade for minimizing feature evaluation cost. In *Artificial Intelligence and Statistics*, pages 218–226, 2012.

[27] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[28] Cloudera Impala. https://github.com/cloudera/impala.

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

[30] Alexis Conneau, German Kruszewski, Guillaume Lample, Loïc Barrault, and Marco Baroni. What you can cram into a single vector: Probing sentence embeddings for linguistic properties. *arXiv preprint arXiv:1805.01070*, 2018.

[31] Daniel Crankshaw, Peter Bailis, Joseph E Gonzalez, Haoyuan Li, Zhao Zhang, Michael J Franklin, Ali Ghodsi, and Michael I Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. *arXiv preprint arXiv:1409.3809*, 2014.

[32] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[33] Marcilio CP de Souto, Ivan G Costa, Daniel SA de Araujo, Teresa B Ludermir, and Alexander Schliep. Clustering cancer gene expression data: A comparative study. *BMC Bioinformatics*, 9(1), 2008.

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

[35] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K Elmagarmid, Ihab F Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. The Data Civilizer system. In *CIDR*, 2017.

[36] Dong Deng, Yu Jiang, Guoliang Li, Jian Li, and Cong Yu. Scalable column concept determination for web tables using large knowledge bases. *Proceedings of the VLDB Endowment*, 6(13):1606–1617, 2013.

[37] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78, 2012.

[38] Xin Luna Dong and Theodoros Rekatsinas. Data integration and machine learning: a natural synergy. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1645–1650. ACM, 2018.

[39] Julian Eberius, Maik Thiele, Katrin Braunschweig, and Wolfgang Lehner. Top-k entity augmentation using consistent set covering. SSDBM, 2015.

[40] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[41] Facebook, Ericsson, and Qualcomm. A Focus on Efficiency (whitepaper). Technical report, 2013.

[42] Ronald Fagin. Fuzzy queries in multimedia database systems. In *PODS*, 1998.

[43] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. Towards a unified architecture for in-RDBMS analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 325–336. ACM, 2012.

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

[45] Jenny Rose Finkel et al. Incorporating non-local information into information extraction systems by Gibbs sampling. In *ACL*, 2005.

[46] Myron Flickner, Harpreet Sawhney, Wayne Niblack, et al. Query by image and video content: The QBIC system. *Computer*, 28(9):23–32, 1995.

[47] Joao Gama. Combining classifiers by constructive induction. In *European Conference on Machine Learning*, pages 178–189. Springer, 1998.

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

[49] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. SystemML: Declarative machine learning on MapReduce. In *2011 IEEE 27th International Conference on Data Engineering*, pages 231–242. IEEE, 2011.

[50] Forouzan Golshani and Nevenka Dimitrova. A language for content-based video retrieval. *Multimedia Tools and Applications*, 6(3):289–312, 1998.

[51] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[52] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[53] Google word2vec. https://code.google.com/p/word2vec/.

[54] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[55] Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284, 2009.

[56] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[57] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. Online aggregation. *ACM SIGMOD Record*, 26(2):171–182, 1997.

[58] Joseph M Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, et al. The madlib analytics library: or mad skills, the sql. *Proceedings of the VLDB Endowment*, 5(12):1700–1711, 2012.

[59] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. In *NIPS Deep Learning Workshop*, 2014.

[60] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[61] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Paramvir Bahl, Matthai Philipose, Phillip B Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *OSDI*, 2018.

[62] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. *arXiv preprint arXiv:1709.01507*, 2017.

[63] Weiming Hu, Nianhua Xie, Li Li, Xianglin Zeng, and Stephen Maybank. A survey on visual content-based video indexing and retrieval. *IEEE Trans. on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 41(6):797–819, 2011.

[64] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5mb model size. *arXiv:1602.07360*, 2016.

[65] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

[66] Panagiotis G. Ipeirotis, Eugene Agichtein, Pranay Jain, and Luis Gravano. To search or to crawl?: Towards a query optimizer for text-centric tasks. In *SIGMOD*, 2006.

[67] Shrainik Jain, Bill Howe, Jiaqi Yan, and Thierry Cruanes. Query2vec: An evaluation of nlp techniques for generalized workload analytics. *arXiv preprint arXiv:1801.05613*, 2018.

[68] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *International Conference on Multimedia*. ACM, 2014.

[69] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pages 427–431. Association for Computational Linguistics, April 2017.

[70] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[71] Daniel Kang, Peter Bailis, and Matei Zaharia. BlazeIt: Fast exploratory video queries using neural networks. *arXiv preprint arXiv:1805.01046*, 2018.

[72] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. No-Scope: Optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.

[73] Torsten Kilias, Alexander Löser, Felix A. Gers, Richard Koopmanschap, Ying Zhang, and Martin Kersten. Idel: In-database entity linking with neural embeddings. *Proceedings of the VLDB Endowment*, 11(5), 2018.

[74] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR*, 2019.

[75] Donald Kossmann, Frank Ramsak, and Steffen Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 275–286. VLDB Endowment, 2002.

[76] Peter Kraft, Daniel Kang, Deepak Narayanan, Shoumik Palkar, Peter Bailis, and Matei Zaharia. Willump: A statistically-aware end-to-end optimizer for machine learning inference. *arXiv preprint arXiv:1906.01974*, 2019.

[77] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. SageDB: A learned database system. In *CIDR*, 2019.

[78] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM, 2018.

[79] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, Michael J Franklin, and Michael I Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, volume 1, pages 2–1, 2013.

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

[81] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J Franklin, and Ken Goldberg. ActiveClean: Interactive data cleaning for statistical modeling. *Proceedings of the VLDB Endowment*, 9(12):948–959, 2016.

[82] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.

[83] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

[84] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M Patel. Model selection management systems: The next frontier of advanced analytics. *ACM SIGMOD Record*, 44(4):17–22, 2016.

[85] Arun Kumar, Feng Niu, and Christopher Ré. Hazy: Making it easier to build and maintain big-data analytics. *Communications of the ACM*, 56(3):40–49, 2013.

[86] Hsiang-Tsung Kung, Fabrizio Luccio, and Franco P Preparata. On finding the maxima of a set of vectors. *Journal of the ACM (JACM)*, 22(4):469–476, 1975.

[87] Tony CT Kuo and Arbee LP Chen. A content-based query language for video databases. In *Multimedia Computing and Systems*, 1996.

[88] Pavel Laskov, Christian Gehl, Stefan Krüger, and Klaus-Robert Müller. Incremental support vector learning: Analysis, implementation and applications. *The Journal of Machine Learning Research*, 7:1909–1936, 2006.

[89] 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. In *ICML*, 2012.

[90] Oliver Lehmberg and Christian Bizer. Stitching web tables for improving matching quality. *Proceedings of the VLDB Endowment*, 10(11):1502–1513, 2017.

[91] Oliver Lehmberg, Dominique Ritze, Robert Meusel, and Christian Bizer. A large public corpus of web tables containing time and context metadata. In *WWW*, 2016.

[92] Steven Levy. How Google's Algorithm Rules the Web. *Wired*, 2010.

[93] David D Lewis and Jason Catlett. Heterogenous uncertainty sampling for supervised learning. In *ICML*, 1994.

[94] John Z Li, M Tamer Ozsu, Duane Szafron, and Vincent Oria. MOQL: A multimedia object query language. In *Intl. Workshop on Multimedia Info. Systems*, 1997.

[95] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[96] Harold Lim, Herodotos Herodotou, and Shivnath Babu. Stubby: A transformation-based optimizer for MapReduce workflows. *Proceedings of the VLDB Endowment*, 5(11):1196–1207, 2012.

[97] Zachary C Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015.

[98] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. Query-based workload forecasting for self-driving database management systems. In *SIGMOD*, pages 631–645. ACM, 2018.

[99] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[100] Ryan Marcus and Olga Papaemmanouil. Towards a hands-free query optimizer through deep learning. In *CIDR*, 2019.

[101] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

[102] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[103] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. MLlib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[104] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *Workshop at International Conference on Learning Representations (ICLR)*, 2013.

[105] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

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

[107] Supun Nakandala, Arun Kumar, and Yannis Papakonstantinou. Incremental and approximate inference for faster occlusion-based deep cnn explanations. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1589–1606. ACM, 2019.

[108] Fatemeh Nargesian, Erkang Zhu, Ken Q Pu, and Renée J Miller. Table union search on open data. *Proceedings of the VLDB Endowment*, 11(7):813–825, 2018.

[109] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. Learning state representations for query optimization with deep reinforcement learning. *arXiv preprint arXiv:1803.08604*, 2018.

[110] Christos H. Papadimitriou and Mihalis Yannakakis. Multiobjective query optimization. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '01, pages 52–59, New York, NY, USA, 2001. ACM.

[111] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. VerdictDB: Universalizing approximate query processing. In *SIGMOD*, 2018.

[112] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS Autodiff Workshop*, 2017.

[113] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

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

[115] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[116] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proc. of NAACL*, 2018.

[117] M Petkovic and W Jonker. A framework for video modelling. In *Intl. Conf. on Applied Informatics*, 2000.

[118] Milan Petkovic and Willem Jonker. *Content-Based Video Retrieval: A Database Perspective*, volume 11. Springer, 1999.

[119] R. Polikar, L. Upda, S.S. Upda, and V. Honavar. Learn++: An incremental learning algorithm for supervised neural networks. *IEEE Transactions on Systems, Man, and Cybernetics*, 31(4):497–508, 2001.

[120] Leonid Portnoy et al. Intrusion detection with unlabeled data using clustering. In *Workshop on Data Mining Applied to Security*, 2001.

[121] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. Snorkel: Rapid training data creation with weak supervision. *Proceedings of the VLDB Endowment*, 11(3):269–282, 2017.

[122] Christopher Ré, Amir Abbas Sadeghian, Zifei Shan, Jaeho Shin, Feiran Wang, Sen Wu, and Ce Zhang. Feature engineering for knowledge base construction. *IEEE Data Eng. Bulletin*, 37(3), 2014.

[123] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *CVPR*, 2016.

[124] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. In *CVPR*, 2017.

[125] Wei Ren, Sameer Singh, Maneesh Singh, and Yuesheng S Zhu. State-of-the-art on spatio-temporal information-based video retrieval. *Pattern Recognition*, 42(2):267–282, 2009.

[126] Simon Rogers and Mark Girolami. *A first course in machine learning*. Chapman & Hall/CRC, 2011.

[127] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[128] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[129] Maytal Saar-Tsechansky, Prem Melville, and Foster Provost. Active Feature-Value Acquisition. *Management Science*, 55(4):664–684, 2009.

[130] Robert E Schapire. The boosting approach to machine learning: An overview. In *Nonlinear estimation and classification*, pages 149–171. Springer, 2003.

[131] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[132] Yoones A Sekhavat, Francesco Di Paolo, Denilson Barbosa, and Paolo Merialdo. Knowledge base augmentation using tabular data. In *LDOW*, 2014.

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

[134] Vraj Shah and Arun Kumar. The ML Data Prep Zoo: Towards semi-automatic data preparation for ML. In *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning*, page 11. ACM, 2019.

[135] Vraj Shah, Arun Kumar, and Xiaojin Zhu. Are key-foreign key joins safe to avoid when learning high-capacity classifiers? *Proceedings of the VLDB Endowment*, 11(3):366–379, 2017.

[136] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ILSVRC*, 2014.

[137] Thomas G Aguierre Smith and Glorianna Davenport. The stratification system a design environment for random access video. In *International Workshop on Network and Operating System Support for Digital Audio and Video*, 1992.

[138] Cees GM Snoek and Marcel Worring. Concept-based video retrieval. *Foundations and Trends in Information Retrieval*, 2(4):215–322, 2008.

[139] Evan R Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J Franklin, and Benjamin Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *2017 IEEE 33rd international conference on data engineering (ICDE)*, pages 535–546. IEEE, 2017.

[140] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: A column-oriented dbms. In *VLDB*, 2005.

[141] Alexander Strehl, Joydeep Ghosh, and Raymond Mooney. Impact of similarity measures on web-page clustering. In *Workshop on AI for Web Search*, 2000.

[142] Yi Sun, Xiaogang Wang, and Xiaoou Tang. Deep convolutional network cascade for facial point detection. In *CVPR*, 2013.

[143] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4), 2005.

[144] Paul E Utgoff. Incremental induction of decision trees. *Machine learning*, 4(2):161–186, 1989.

[145] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*, pages 1009–1024. ACM, 2017.

[146] Rodrigo Verschae, Javier Ruiz-del Solar, and Mauricio Correa. A unified learning framework for object detection and classification using nested cascades of boosted classifiers. *Machine Vision and Applications*, 19(2):85–103, 2008.

[147] Paul Viola and Michael J Jones. Robust real-time face detection. *International Journal of Computer Vision*, 57(2):137–154, 2004.

[148] Andreas Vlachos. A stopping criterion for active learning. *Computer Speech & Language*, 22(3):295–312, 2008.

[149] Alex Wang, Amapreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[150] John Wieting, Mohit Bansal, Kevin Gimpel, and Karen Livescu. Towards universal paraphrastic sentence embeddings. In *ICLR*, 2016.

[151] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2011.

[152] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. Helix: Holistic optimization for accelerating iterative machine learning. *Proceedings of the VLDB Endowment*, 12(4):446–460, 2018.

[153] Mohamed Yakout, Kris Ganjam, Kaushik Chakrabarti, and Surajit Chaudhuri. Infogather: entity augmentation and attribute discovery by holistic matching with web tables. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 97–108. ACM, 2012.

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

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

[156] Ce Zhang, Arun Kumar, and Christopher Ré. Materialization optimizations for feature selection workloads. *ACM Transactions on Database Systems (TODS)*, 41(1):2, 2016.

[157] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. Live video analytics at scale with approximation and delay-tolerance. In *NSDI*, 2017.

[158] Meihui Zhang and Kaushik Chakrabarti. Infogather+: semantic matching and annotation of numeric and time-varying attributes in web tables. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 145–156. ACM, 2013.

[159] Xiaolu Zhang, Yueguo Chen, Jinchuan Chen, Xiaoyong Du, and Lei Zou. Mapping entity-attribute web tables to web-scale knowledge bases. In *International Conference on Database Systems for Advanced Applications*, pages 108–122. Springer, 2013.

[160] Jingbo Zhu, Huizhen Wang, Eduard Hovy, and Matthew Ma. Confidence-based stopping criteria for active learning for data annotation. *Transactions on Speech and Language Processing*, 6(3):3, 2010.