Large-Scale Machine Learning at Twitter

Jimmy Lin and Alek Kolcz
Twitter, Inc.

ABSTRACT

The success of data-driven solutions to difficult problems, along with the dropping costs of storing and processing massive amounts of data, has led to growing interest in large-scale machine learning. This paper presents a case study of Twitter's integration of machine learning tools into its existing Hadoop-based, Pig-centric analytics platform. We begin with an overview of this platform, which handles "traditional" data warehousing and business intelligence tasks for the organization. The core of this work lies in recent Pig extensions to provide *predictive* analytics capabilities that incorporate machine learning, focused specifically on supervised classification. In particular, we have identified stochastic gradient descent techniques for online learning and ensemble methods as being highly amenable to scaling out to large amounts of data. In our deployed solution, common machine learning tasks such as data sampling, feature generation, training, and testing can be accomplished directly in Pig, via carefully crafted loaders, storage functions, and user-defined functions. This means that machine learning is *just another Pig script*, which allows seamless integration with existing infrastructure for data management, scheduling, and monitoring in a production environment, as well as access to rich libraries of user-defined functions and the materialized output of other scripts.

Categories and Subject Descriptors: H.2.3 [Database Management]: Languages

General Terms: Languages

Keywords: stochastic gradient descent, online learning, ensembles, logistic regression

1. INTRODUCTION

Hadoop, the open-source implementation of MapReduce [15], has emerged as a popular framework for large-scale data processing. Among its advantages are the ability to horizontally scale to petabytes of data on thousands of commodity servers, easy-to-understand programming semantics, and a high degree of fault tolerance. Although originally designed for applications such as text analysis, web indexing, and graph processing, Hadoop can be applied to manage structured data as well as "dirty" semistructured datasets with inconsistent schema, missing fields, and invalid values.

Today, Hadoop enjoys widespread adoption in organizations ranging from two-person startups to Fortune 500 companies. It lies at the core of a software stack for large-scale analytics, and owes a large part of its success to a vibrant ecosystem. For example, Pig [37] and Hive [47] provide higher-level languages for data analysis: a dataflow language called Pig Latin and a dialect of SQL, respectively. HBase, the open-source implementation of Google's Bigtable [13], provides a convenient data model for managing and serving semistructured data. We are also witnessing the development of hybrid data-processing approaches that integrate Hadoop with traditional RDBMS techniques [1, 34, 3, 30], promising the best of both worlds.

The value of a Hadoop-based stack for "traditional" data warehousing and business intelligence tasks has already been demonstrated by organizations such as Facebook, LinkedIn, and Twitter (e.g., [22, 41]). This value proposition also lies at the center of a growing list of startups and large companies that have entered the "big data" game. Common tasks include ETL, joining multiple disparate data sources, followed by filtering, aggregation, or cube materialization. Statisticians might use the phrase *descriptive statistics* to describe this type of analysis. These outputs might feed report generators, frontend dashboards, and other visualization tools to support common "roll up" and "drill down" operations on multi-dimensional data. Hadoop-based platforms have also been successful in supporting *ad hoc* queries by a new breed of engineers known as "data scientists".

The success of the Hadoop platform drives infrastructure developers to build increasingly powerful tools, which data scientists and other engineers can exploit to extract insights from massive amounts of data. In particular, we focus on machine learning techniques that enable what might be best termed *predictive analytics*. The hope is to mine statistical regularities, which can then be distilled into models that are capable of making predictions about future events. Some examples include: Is this tweet spam or not? What star rating is the user likely to give to this movie? Should these two people be introduced to each other? How likely will the user click on this banner ad?

This paper presents a case study of how machine learning tools are integrated into Twitter's Pig-centric analytics stack for the type of predictive analytics described above. Focus-

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

ing on supervised classification, we have integrated standard machine learning components into Pig loaders, storage functions, and user-defined functions (UDFs) [18], thereby creating a development environment where machine learning tasks (data sampling, feature generation, training, testing, etc.) feel natural. A noteworthy feature is that machine learning algorithms are integrated into Pig in such a way that scaling to large datasets is accomplished through ensemble methods. A key characteristic of our solution is that machine learning is *just another Pig script*, which allows seamless integration with existing infrastructure for data management, scheduling, and monitoring in a production environment, as well as access to rich libraries of existing UDFs and the materialized output of other scripts.

We view this work has having three contributions. First, we provide an overview of Twitter's analytics stack, which offers the reader a glimpse into the workings of a large-scale production platform. Second, we describe Pig extensions that allow seamless integration of machine learning capabilities into this production platform. Third, we identify stochastic gradient descent and ensemble methods as being particularly amenable to large-scale machine learning. We readily acknowledge that this paper does not present any fundamental contributions to machine learning. Rather, we focus on end-to-end machine learning workflows and integration issues in a production environment. Although specifically a case study, we believe that these experiences can be generalized to other organizations and contexts, and therefore are valuable to the community.

2. BACKGROUND AND RELATED WORK

We begin with a brief overview of machine learning. Let X be the input space and Y be the output space. Given a set of training samples $D = \{(x_1, y_1), (x_2, y_2)...(x_n, y_n)\}$ from the space $X \times Y$ (called labeled examples or instances), the *supervised* machine learning task is to induce a function $f : X \rightarrow Y$ that best explains the training data. The notion of "best" is usually captured in terms of minimizing "loss", via a function L which quantifies the discrepancy between the functional prediction $f(x_i)$ and the actual output y_i, for example, minimizing the quantity $\sum_{(x_i,y_i)\in D} L(f(x_i), y_i)$. Once a model is learned, i.e., the best f is selected from a hypothesis space, it can then be applied to make predictions on previously unseen data (hence, *predictive* analytics).

In most cases, x_i is represented as a feature vector, i.e., $\mathrm{x}_i \in \mathbb{R}^d$. For the supervised classification task, y is draw from a finite set, and in the case of binary classification, $y \in \{-1, +1\}$. In practice, multi-class classification can be decomposed into ensembles of binary classifiers. Common strategies include training *one-vs-all* classifiers, pairwise *one-vs-one* classifiers, or classifier cascades. Thus, binary classifiers form primitives on which more complex classifiers can be built. It is, of course, impossible to do justice to the immense literature on machine learning in the space available for this paper; for more details, we refer the reader to standard textbooks [6, 23].

There are three main components of a machine learning solution: the data, features extracted from the data, and the model. Accumulated experience over the last decade has shown that in real-world settings, the size of the dataset is the most important factor [21, 28]. Studies have repeatedly shown that simple models trained over enormous quantities of data outperform more sophisticated models trained on less data [4, 8, 16]. This has led to the growing dominance of simple, data-driven solutions.

Labeled training examples derive from many sources. Human annotators can be paid to manually label examples, and with the advent of crowdsourcing the cost can be quite reasonable [45]. However, the amount of training data that can be manually generated pales in comparison to the amount of data that can be extracted automatically from logs and other sources in an organization's data warehouse. As a simple example, query and interaction logs from commercial search engines can be distilled into relevance judgments [24]. These data tend to be noisy, but modern "learning to rank" [27] algorithms are resilient (by design) to noisy data. By mining log data, an organization can generate practically limitless amounts of training data for certain tasks.

Traditionally, the machine learning community has assumed sequential algorithms on data that fit in memory. This assumption is no longer realistic for many scenarios, and recently we have seen work on multi-core [35] and cluster-based solutions [2]. Examples include learning decision trees and their ensembles [46, 39], MaxEnt models [32], structured perceptrons [33], support vector machines [12], and simple phrase-based approaches [5]. Recent work in online learning (e.g., by Bottou [7] and Vowpal Wabbit[1]) is also applicable to a large-data setting, although in practice such learners are often limited by disk I/O. Online learning also exemplifies the increasingly popular approach where one does not attempt to arrive at exactly the same solution as using an equivalent batch learner on a single machine. Rather, researchers and practitioners often exploit stochastic learners or large ensembles of learners with limited communication [46]. Since "data is king", these approaches work well due to their ability to process massive amounts of data.

Despite growing interest in large-scale learning, there are relatively few published studies on machine learning *workflows* and how such tools integrate with data management platforms: Sculley et al. [42] describe Google's efforts for detecting adversarial advertisements. Cohen et al. [14] advocate the integration of predictive analytics into traditional RDBMSes. It is well known that Facebook's data analytics platform is built around Hive [47], particularly for traditional business intelligence tasks, but we are not aware of any published work on its infrastructure for machine learning. LinkedIn similarly has built significant infrastructure around Hadoop for offline data processing and a variety of systems for online data serving [31], but once again little is known about machine learning.

The increasing popularity of Hadoop has sparked a number of efforts to implement scalable machine learning tools in MapReduce (e.g., the open source Mahout project[2]). A number of algorithms have been "ported" to MapReduce (e.g., [35, 39, 44]), including a declarative abstraction for matrix-style computations [20]. Other research prototypes that attempt to tackle large-scale machine learning include Spark [49], ScalOps [48], and a number of proposals at a recent NIPS workshop on "Big Learning". However, it is not entirely clear how these systems fit into an end-to-end production pipeline. That is, how do we best integrate machine learning into an existing analytics environment? What are the common architectural patterns and the tradeoffs they

[1] http://hunch.net/∼vw/

[2] http://mahout.apache.org/

encode? Are there best practices to adopt? While this paper does not definitively answer these questions, we offer a case study. Since Twitter's analytics stack consists mostly of open-source components (Hadoop, Pig, etc.), much of our experience is generalizable to other organizations.

3. TWITTER'S ANALYTICS STACK

A large Hadoop cluster lies at the core of our analytics infrastructure, which serves the entire company. Data is written to the Hadoop Distributed File System (HDFS) via a number of real-time and batch processes, in a variety of formats. These data can be bulk exports from databases, application logs, and many other sources. When the contents of a record are well-defined, they are serialized using either Protocol Buffers[3] or Thrift.[4] Ingested data are LZO-compressed, which provides a good tradeoff between compression ratio and speed (see [29] for more details).

In a Hadoop job, different record types produce different types of input key-value pairs for the mappers, each of which requires custom code for deserializing and parsing. Since this code is both regular and repetitive, it is straightforward to use the serialization framework to specify the data schema, from which the serialization compiler generates code to read, write, and manipulate the data. This is handled by our system called Elephant Bird,[5] which automatically generates Hadoop record readers and writers for arbitrary Protocol Buffer and Thrift messages.

Instead of directly writing Hadoop code in Java, analytics at Twitter is performed mostly using Pig, a high-level dataflow language that compiles into physical plans that are executed on Hadoop [37, 19]. Pig (via a language called Pig Latin) provides concise primitives for expressing common operations such as projection, selection, group, join, etc. This conciseness comes at low cost: Pig scripts approach the performance of programs directly written in Hadoop Java. Yet, the full expressiveness of Java is retained through a library of custom UDFs that expose core Twitter libraries (e.g., for extracting and manipulating parts of tweets). For the purposes of this paper, we assume that the reader has at least a passing familiarity with Pig.

Like many organizations, the analytics workload at Twitter can be broadly divided into two categories: aggregation queries and *ad hoc* queries. The aggregation queries materialize commonly-used intermediate data for subsequent analysis and feed front-end dashboards. These represent relatively standard business intelligence tasks, and primarily involve scans over large amounts of data, triggered periodically by our internal workflow manager (see below). Running alongside these aggregation queries are *ad hoc* queries, e.g., one-off business requests for data, prototypes of new functionalities, or experiments by our analytics group. These queries are usually submitted directly by the user and have no predictable data access or computational pattern. Although such jobs routinely process large amounts of data, they are closer to "needle in a haystack" queries than aggregation queries.

Production analytics jobs are coordinated by our workflow manager called Oink, which schedules recurring jobs at fixed intervals (e.g., hourly, daily). Oink handles dataflow dependencies between jobs; for example, if job B requires data generated by job A, then Oink will schedule A, verify that A has successfully completed, and then schedule job B (all while making a best-effort attempt to respect periodicity constraints). Finally, Oink preserves execution traces for audit purposes: when a job began, how long it lasted, whether it completed successfully, etc. Each day, Oink schedules hundreds of Pig scripts, which translate into thousands of Hadoop jobs.

[3] http://code.google.com/p/protobuf/

[4] http://thrift.apache.org/

[5] http://github.com/kevinweil/elephant-bird

4. EXTENDING PIG

The previous section describes a mature, production system that has been running successfully for several years and is critical to many aspects of business operations. In this section, we detail Pig extensions that augment this data analytics platform with machine learning capabilities.

4.1 Development History

To better appreciate the solution that we have developed, it is perhaps helpful to describe the development history. Twitter has been using machine learning since its earliest days. Summize, a two year old startup that Twitter acquired primarily for its search product in 2008, had as part of its technology portfolio sentiment analysis capabilities based in part on machine learning. After the acquisition, machine learning contributed to spam detection and other applications within Twitter. These activities predated the existence of Hadoop and what one might recognize as a modern data analytics platform. Since our goal has never been to make fundamental contributions to machine learning, we have taken the pragmatic approach of using off-the-shelf toolkits where possible. Thus, the challenge becomes how to incorporate third-party software packages along with in-house tools into an existing workflow.

Most commonly available machine learning toolkits are designed for a single machine and cannot easily scale to the dataset sizes that our analytics platform can easily generate (although more detailed discussion below). As a result, we often resorted to sampling. The following describes a not uncommon scenario: Like most analytics tasks, we began with data manipulation using Pig, on the infrastructure described in Section 3. The scripts would stream over large datasets, extract signals of interest, and materialize them to HDFS (as labels and feature vectors). For many tasks, it was as easy to generate a million training examples as it was to generate ten million training examples or more. However, generating too much data was counterproductive, as we often had to downsample the data so it could be handled by a machine learning algorithm on a single machine. The training process typically involved copying the data out of HDFS onto the local disk of another machine—frequently, this was another machine in the datacenter, but running experiments on individuals' laptops was not uncommon. Once a model was trained, it was applied in a similarly *ad hoc* manner. Test data were prepared and sampled using Pig, copied out of HDFS, and fed to the learned model. These results were then stored somewhere for later access, for example, in a flat file that is then copied back to HDFS, as records inserted into a database, etc.

There are many issues with this workflow, the foremost of which is that downsampling largely defeats the point of working with large data in the first place. Beyond the issue of scalability, using existing machine learning tools created

workflow challenges. Typically, data manipulation in Pig is followed by invocation of the machine learning tool (via the command line or an API call), followed perhaps by more data manipulation in Pig. During development, these context switches were tolerable (but undoubtedly added friction to the development process). In production, however, these same issues translated into brittle pipelines. It is common to periodically update models and apply classifiers to new data, while respecting data dependencies (e.g., data import schedules). To accomplish this, we often resorted to brittle shell scripts on cron, and in a few cases, having individuals *remember* to rerun models (by hand) "once in a while". Due to application context switches between data manipulation in Pig and machine learning in separate packages, we could not take advantage of our Oink workflow manager, which was designed only for Pig jobs. Although internally, we have another system called Rasvelg for managing non-Pig jobs, coordination between the Pig and non-Pig portions of the workflow was imperfect. Disparate systems rendered error reporting, monitoring, fault handling, and other production considerations needlessly complex.

It became clear that we needed tighter integration of machine learning in Pig. Recognizing the deficiencies described above, we set out on a redesign with the following two goals:

- **Seamless scaling to large datasets.** First and foremost, the framework must support large datasets. We should be able to exploit as much data as we can generate. However, we recognize that there remains value in small-scale experiments on sampled data, particularly for development purposes. Thus, our goal is an architecture that can scale seamlessly from tens of thousands of examples to tens of millions of examples and beyond. Specifically, we want to avoid separate processes for developing on a laptop, prototyping on a server in the datacenter, and running at scale on our Hadoop cluster.
- **Integration into production workflows.** The value of analytics manifests itself only when deployed in a production environment to generate insights in a timely fashion. Twitter's analytics platform already has well-defined processes for deployment in production, and it is important that our machine learning tools integrate naturally into these existing workflows.

Clearly, certain machine learning capabilities are awkward to directly implement in Pig: for these, existing code needs to be "wrapped" in a manner that presents a clean abstraction. On the other hand, other common machine learning tasks seem suitable to directly implement in Pig. So the design challenge boils down to this: what are the core non-Pig machine learning primitives, and how can they be combined with built-in Pig language constructs to create a machine learning platform that meets the above design goals?

At the high-level, our solution is simple: feature extractors are written as user-defined functions; the inner loop of a single-core learner is encapsulated in a Pig storage function; and prediction using learned models is performed using UDFs. Everything else is considered "glue" and directly handled by Pig. In this overall design, the question of what code we *actually* use for the non-Pig machine learning primitives is mostly an implementation detail. After exploring a few options, we integrated mostly in-house code, for reasons explained below.

As previously mentioned, Vowpal Wabbit is a fast online learner. It would have been a great candidate to integrate into our Pig framework as a core learner and classifier, except it is implemented in C++, which doesn't fit well with Twitter's JVM-centric runtime environment.

The other package we examined was Mahout, a large-scale machine learning toolkit for Hadoop. Although the Mahout project began in 2008, the pace at which capabilities matured was uneven in its early days. Many components in Twitter's codebase for machine learning reached maturity before or at around the same time as similar capabilities in Mahout. Maturity aside, Mahout was designed as a standalone package, complete with its own learners, internal representations, mechanisms for flow control, etc. It wasn't practical to simply take our Pig output and "run Mahout", for exactly the workflow issues discussed above. Pig integration would have required pulling apart Mahout components, deciding which we should wrap and expose as Pig primitives, and which we should ignore—this is exactly our challenge to begin with, so having Mahout doesn't actually get us closer to the solution. Ultimately, we decided not to build around Mahout, because it would have required significant reworking of legacy code already in production.

Although we ultimately did not use Mahout components in our implementation, it is entirely possible to integrate Mahout with Pig using the same techniques we introduce in this paper: wrapping core machine learning functionality as Pig primitives, and leveraging Pig itself to manage large, complex dataflows. In more recent work, we take this route to leverage some of Mahout's other capabilities (beyond classification). Twitter has recently open-sourced Pig bindings for Mahout's internal data representation (SequenceFiles of VectorWritables) in its Elephant Bird package. This in turn has enabled an outside prototype that demonstrates deeper integration with Mahout's learners, along the same lines as presented in this paper.[6] In many ways, this illustrates the flexibility of our design and the generalizability of our solution. As discussed earlier, our contributions lie in the overall framework of how machine learning primitives can be composed in Pig—whether these primitives are backed by in-house code or open-source components is an implementation detail. In the future, we envision an ecosystem of Pig primitives which are merely thin wrappers around existing machine learning components, all interoperating in complex dataflows orchestrated by Pig.

4.2 Core Libraries

Our machine learning framework consists of two components: a core Java library and a layer of lightweight wrappers that expose functionalities in Pig.

The core library is worth a passing description, but is not terribly interesting or innovative. It contains basic abstractions similar to what one might find in Weka, Mallet, Mahout, and other existing packages. We have a representation for a feature vector, essentially a mapping from strings to floating point feature values, mediated by integer feature ids for representational compactness. A classifier is an object that implements a `classify` method, which takes as input a feature vector and outputs a classification object (encapsulating a distribution over target labels). There are two different interfaces for training classifiers: Batch trainers implement a builder pattern and expose a `train` method that

[6] http://github.com/tdunning/pig-vector

takes a collection of (label, feature vector) pairs and returns a trained classifier. Online learners are simply classifiers that expose an `update` method, which processes individual (label, feature vector) pairs. Finally, all classifiers have the ability to serialize their models to, and to load trained models from abstract data streams (which can be connected to local files, HDFS files, etc.).

Our core Java library contains a mix of internally built classifiers and trainers (for logistic regression, decision trees, etc.), as well as adaptor code that allows us to take advantage of third-party packages via a unified interface. All of these abstractions and functionalities are fairly standard and should not come as a surprise to the reader.

4.3 Training Models

For model training, our core Java library is integrated into Pig as follows: feature vectors in Java are exposed as maps in Pig, which we treat as a set of feature id (int) to feature value (float) mappings. Thus, a training instance in Pig has the following schema:

```
(label: int, features: map[])
```

As an alternative, training instances can be represented in SVMLight format, a simple sparse-vector encoding format supported by many off-the-shelf packages. SVMLight is a line-based format, with one training instance per line. Each line begins with a label, i.e., $\{+1, -1\}$, followed by one or more space separated (feature id, feature value) pairs internally delimited by a colon (`:`). We have Pig loaders and storage functions for working with instances in this format, as well as UDFs to convert to and from Pig maps.

One of the primary challenges we had to overcome to enable Pig-based machine learning was the mismatch between typical Pig dataflows and dataflows in training machine learning models. In typical Pig scripts, data flow from sources (HDFS files, HBase rows, etc.), through transformations (joins, filters, aggregations, etc.), and are written to sinks (other HDFS files, another HBase table, etc.). In this dataflow, UDFs might read "side data", for example, loading up dictionaries. When training a classifier, the input data consist of (label, feature vector) pairs and the output is the trained model. The model is not a "transformation" of the original data in the conventional sense, and is actually closer (both in terms of size and usage patterns) to "side data" needed by UDFs.

Our solution is to embed the learner inside a Pig storage function. Typically, the storage function receives output records, serializes them, and writes the resulting representation to disk. In our case, the storage function receives output records and feeds them to the learner (without writing any output). Only when all records have been processed (via an API hook in Pig) does the storage function write the *learned model* to disk. That is, the input is a series of tuples representing (label, feature vector) pairs, and the output is the trained model itself.

This design affords us precise control of how the learning is carried out. Since storage functions are called in the final reduce stage of the overall dataflow, by controlling the number of reducers (with the Pig `parallel` keyword), we can control the number of models that are learned. For example, if we set the parallel factor to one, then all training instances are shuffled to a single reducer and fed to exactly one learner. By setting larger parallel factors $n > 1$, we can learn simple ensembles—the training data is partitioned n-ways and models are independently trained on each partition. This design is illustrated in Figure 1.

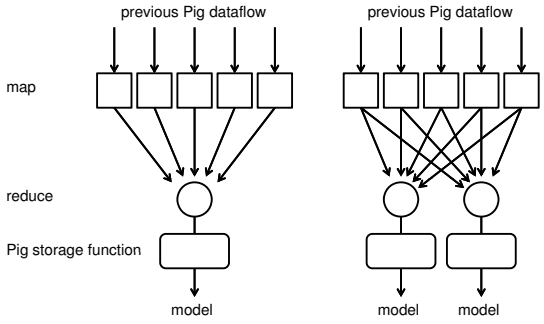

Figure 1: Illustration of how learners are integrated into Pig storage functions. By controlling the number of reducers in the final MapReduce job, we can control the number of models constructed: on the left, a single classifier, and on the right, a two-classifier ensemble.

In short, the parallelization provided by running multiple reducers corresponds naturally to training ensembles of classifiers, and using this mechanism, we can arbitrarily scale out (overcoming the bottleneck of having to shuffle all training data onto a single machine). As is often the case for many machine learning problems, and confirmed in our experiments (see Section 6), an ensemble of classifiers trained on partitions of a large dataset outperforms a single classifier trained on the entire dataset (more precisely, lowers the variance component in error).

The reader here might object that our design violates an abstraction barrier, in that Pig is an abstract dataflow language, and we require the programmer to have an understanding of the underlying MapReduce-based physical plan. However, we argue that Pig is so tightly coupled to Hadoop in practice that being a competent Pig programmer requires at least passing knowledge of how the dataflow translates into Hadoop jobs. Even failing that, for the novice Pig user we can supply a standard idiom (see below) that is guaranteed to trigger a (final) Hadoop job just prior to training, i.e., we can offer the following instructions: "when in doubt, include these lines at the end of your Pig script, and set the `parallel` there for training n classifiers in parallel."

Now putting the pieces together: in our design, training a classifier can be as simple as two lines in Pig:

```
training = load 'training.txt'
               using SVMLightStorage()
               as (target: int, features: map[]);
store training into 'model/'
    using FeaturesLRClassifierBuilder();
```

In this case, `FeaturesLRClassifierBuilder` is a Pig storage function that wraps the learner for a logistic regression classifier from our core Java library.

Our machine learning algorithms can be divided into two classes: batch learners and online learners. Batch learners require all data to be held in memory, and therefore the Pig storage functions wrapping such learners must first internally buffer all training instances before training. This presents a scalability bottleneck, as Hadoop reduce tasks are typically allocated only a modest amount of memory. Online learners, on the other hand, have no such restriction: the Pig storage function simply streams through incoming instances,

feeds them to the learner, and then discards the example. Parameters for the model must fit in memory, but in practice this is rarely a concern. Online learners are preferable for large datasets, but often batch learners perform better on less data and thus remain useful for experiments. We provide more details in Section 5.

In addition to training models, many other types of data manipulations common in machine learning can be straightforwardly accomplished in Pig. For example, it is often desirable to randomly shuffle the labeled instances prior to training, especially in the case of online learners, where the learned model is dependent on the ordering of the examples. This can be accomplished by generating random numbers for each training instance and then sorting by it, as follows:

```
training = foreach training generate
              label, features, RANDOM() as random;
training = order training by random parallel 1;
```

Shuffling data just prior to training is not only good practice, it triggers a final MapReduce job and provides an opportunity to control the number of models learned.

Another common operation is to separate data into training and test portions (or fold creation for cross-validation). Generating a 90/10 training/test split can be accomplished in Pig as follows:

```
data = foreach data generate target, features,
           RANDOM() as random;
split data into training if random <= 0.9,
    test if random > 0.9;
```

We have shown how training models can be accomplished entirely in Pig (in as few as two lines). In addition, many common ways of manipulating data become idiomatic in Pig, which makes scripts easier to understand.

4.4 Using Learned Models

Once a classification model has been learned, its accuracy on held out test data needs to be verified. Only after that can the solution be deployed. Deployment of machine learning solutions is dependent on the nature of the problem, but the two primary modes are: to make periodic batch predictions on new incoming data and to make predictions in an online setting over live data streams. The second is beyond the scope of this paper,[7] and the first is also handled by our analytics infrastructure—for example, a model might be applied on an hourly basis to classify newly imported log data. As it turns out, applying learned models to make batch predictions and the problem of verifying model accuracy are basically the same, and require the same functionalities to be exposed in Pig.

We have developed wrappers that allow us to use classifiers directly in Pig. For each classifier in our core Java library, we have a corresponding Pig UDF. The UDF is initialized with the model, and then can be invoked like any other UDF. Thus, we can evaluate classifier effectiveness on held out test data in the following manner:

```
define Classify ClassifyWithLR('model/');
data = load 'test.txt' using SVMLightStorage()
           as (target: double, features: map[]);
data = foreach data generate target,
           Classify(features) as prediction;
```

[7]Note, however, that after a model is trained, it can be easily used anywhere (with a JVM).

With additional processing, we can compute standard metrics such as precision, recall, area under the curve, etc. See Section 6 for a complete example. In this specific case, we are evaluating classifier effectiveness, but the only difference between this and making batch predictions on new data is whether or not we have ground truth labels.

To take advantage of ensembles, we have a separate UDF that encapsulates the desired functionality:

```
define Classify ClassifyWithEnsemble('model/',
   'classifier.LR', 'vote');
```

Here, the `model` directory contains multiple models. The UDF specifies the base classifier (logistic regression) and the evidence combination method (simple majority voting). It instantiates the ensemble, and on each call, passes the feature vector to each classifier and combines the evidence appropriately—this is transparent to the programmer. In Section 5.2 we describe our ensembles in more detail.

4.5 Discussion

The key characteristic of our design is that a machine learning job is *just another Pig script*, which allows seamless integration with existing infrastructure. Oink schedules a multitude of production jobs that run daily, outputting results in directories on HDFS along the lines of `/processed/task/YYYY/MM/DD/` (while managing data dependencies, monitoring job completion, etc.). Machine learning scripts can take advantage of Oink, for example, to build daily models. Another job could then use the most recent model to make predictions on incoming data every hour—also scheduled by Oink, which figures out the data dependencies. Thus, productionizing machine learning jobs is no more difficult than Pig scripts that count records.

Also, the machine learning code has access to all data stored on HDFS (complete with schemas where available) since Pig loaders already exist (via Elephant Bird). The machine learning code has access to hundreds of UDFs in Twitter libraries that have accumulated over the years for manipulating everything from tweets to timestamps to IP addresses. We have thus far not explicitly addressed feature generation, because it is largely dependent on the problem domain and requires the creativity of the engineer to be able to cull the relevant signals from vast quantities of data. However, we describe a sample application in Section 6 to give the reader a feel for the complete process. It suffices to say that the algorithm designer has access the entire Pig analytics stack. In many cases, basic feature generators are simply repurposed existing UDFs.

Finally, Pig allows us to seamlessly scale down onto individual servers or even laptops by running in "local mode". Pig local mode allows evaluation of scripts on a single machine, without requiring access to a cluster, which is very useful for rapid prototyping and debugging. After a Pig script has been debugged and verified to run correctly in local mode, it can be deployed to the cluster at scale with minimal modification. Although there is overhead to running Pig in a single-machine environment, this disadvantage is more than compensated for by the ability to seamlessly work with datasets whose sizes differ by orders of magnitude.

5. SCALABLE MACHINE LEARNING

The field of machine learning is incredibly rich and diverse, but from the vast literature, we have identified two classes

of techniques that are particularly amenable to large-scale machine learning. The first is stochastic gradient descent, representative of online learners that can easily scale to large datasets. The second is ensemble methods, which allow us to parallelize training in an nearly embarrassingly parallel manner, yet retain high levels of effectiveness. Both are well known in the machine learning literature, and together they form a powerful combination. These techniques occupy the focus of our implementation efforts.

5.1 Stochastic Gradient Descent

In recent years, the machine learning community has seen a resurgence of interest in online learning methods that do not require multiple iterations over a dataset. Aside from the fact that these methods naturally adjust to changing data conditions (e.g., temporal variation in the dependence of the target variable on the input features), they are also amenable to handling massive datasets that do not fit into memory. For applications involving text it has been shown that online learning algorithms lead to competitive classification accuracies compared to support vector machines (SVMs) while being orders of magnitude faster to train [7, 2]. While there are many different flavors of online learning, we focus on stochastic gradient descent (SGD) for logistic regression, which is a well known linear model that automatically outputs well-calibrated estimates of the posterior class probabilities.

Stochastic gradient descent is an approximation to gradient descent methods commonly used in batch learning [43]. In standard gradient descent, one computes the gradient of the objective loss function using all training examples, which is then used to adjust the parameter vector in the direction opposite to the gradient [6]. The process is repeated each iteration till convergence. Subgradient methods represent an approximation in which only a subset of all training examples is used in each gradient computation, and when the size of the subsample is reduced to a single instance, we arrive at stochastic gradient descent.

Assuming a two-class problem with a training set $D = \{(x_1, y_1), (x_2, y_2)...(x_n, y_n)\}$, where x's are feature vectors and $y \in \{-1, +1\}$, we define a class of linear discriminative functions of the form:

$$F(x) : R^N \rightarrow \{-1, +1\}$$
$$F(x) = \begin{cases} +1 \text{ if } w \cdot x \geq t \\ -1 \text{ if } w \cdot x < t \end{cases}$$

where t represents a decision threshold and w is the weight vector. The modeling process usually optimizes the weight vector based on the training data D, and the decision threshold is subsequently tuned according to various operational constraints (e.g., taking precision, recall, and coverage into account). A range of methods for learning w have been proposed, where generally different methods optimize a different form of a loss or objective function defined over the training data. One particularly well-established technique is known as logistic regression, where the linear function $w \cdot x$ is interpreted as the logarithmic odds of x belonging to class $+1$ over -1, i.e.,

$$\log \left[\frac{p(y = +1|x)}{p(y = -1|x)} \right] = w \cdot x$$

The objective of regularized logistic regression (using Gaussian smoothing) is to identify the parameter vector w that maximizes the conditional posterior of the data.

$$L = \exp\left(-\lambda w^2/2\right) \cdot \prod_i p(y_i|x_i)$$

where

$$p(y = +1|x) = \frac{1}{(1 + \exp(-w \cdot x))}$$

and

$$p(y = -1|x) = 1 - p(y = +1|x) = \frac{1}{(1 + \exp(w \cdot x))}$$

Maximizing L is equivalent to maximizing $\log(L)$, which for gradient descent is accomplished by adjusting the weight vector in the direction opposite to the gradient of $\log(L)$:

$$\begin{aligned} -\nabla \log(L) &= \lambda w + \sum_i \frac{1}{p(y_i|x_i)} \frac{\partial}{\partial w} p(y_i|x_i) \\ &= \lambda w + \sum_i y_i p(y_i|x_i)(1 - p(y_i|x_i)) \end{aligned}$$

Let γ represent the update rate, then in stochastic gradient update where the gradient is computed based on a single training instance, the update to the weight vector upon seeing the ith training example is given by

$$w \leftarrow w + \gamma \left[\lambda w + y_i \cdot p(y_i|x_i)(1 - p(y_i|x_i))\right]$$

Note that all elements of the weight vector w are decayed by factor $(1 - \gamma\lambda)$ at each iteration, but in situations where the feature vectors of training instances are very sparse (as is true for text) we can simply delay the updates for features until they are actually seen.

The SGD variant of logistic regression depends on the choice of two regularization parameters, λ of the Gaussian prior and the update rate γ. Both can be selected *a priori* or tuned using validation data. The update rate is commonly decreased as the training progresses, e.g., as $\propto 1/t$ where t is the index of the current update. A recently proposed extension of SGD, called Pegasos [43], modifies the update process such that after each modification of the weight vector, the weights are rescaled to fit with an L2 ball of radius γ. This is coupled with an update rate defined as $\propto 1/\lambda\dot{t}$. While Pegasos is generally a subgradient method (i.e., it performs updates on small batches of training data), it also supports traditional stochastic gradient learning with updates performed for each training instance. Our machine learning library includes an implementation of SGD regularized logistic regression with fixed and decaying update rates, as well as its Pegasos variant.

In applications of machine learning to large datasets comprising different and diverse data sources, it is common to find features with drastically different properties: binary, multi-valued discrete, integer-valued, real-valued, etc. This calls for data pre-processing and normalization, especially if common regularization parameters are used. In our case, we typically discretize the features prior to model training using the MDL algorithm [17].

5.2 Ensemble Methods

Recall that in our Pig framework, the `parallel` keyword associated with the final MapReduce job determines the

number of reducers, and hence, the number of independent models that are learned. When applying these learned models, evidence from individual classifier instances must be combined to produce a final prediction. This is where ensemble methods come into play.

Given n independently trained, but possibly correlated, classifiers $C_1, C_2 \ldots C_n$, there are many known methods to combine evidence from individual classifier decisions [32]. The simplest is majority voting, where the output y predicted for an input x is:

$$y = \arg\max_{y \in Y} \sum_{k=1}^{n} \alpha_k \mathrm{I}\left(\arg\max_{y' \in Y} [p_k(y'|\mathrm{x}) = y]\right)$$

where I is an indicator function of the predicate it takes as argument. Alternatively, the class probabilities could be taken into account:

$$y = \arg\max_{y \in Y} \sum_{k=1}^{n} \alpha_k p_k(y|\mathrm{x})$$

This evidence combination method allows, for example, two classifiers that are "unsure" of their decisions to be overruled by a third classifier that is highly confident in its prediction. We have implemented both evidence combination methods.

In the default case, all α's are set equal to one, indicating that each vote is equally important, although this need not be the case. Uniform weighting is particularly suited to scenarios where all members of the ensemble represent the same underlying learner and trained over similar quantities of data. According to the bias-variance decomposition of the generalization error [10] applied to ensembles, averaging models characterized by similar inductive biases leads to a reduction of the variance component of the ensemble when compared to an individual model [26]. Large datasets are amenable to this type of learning via sampling and partitioning, but for smaller ones bagging has been used, where each member of the ensemble is trained with data representing a bootstrap sample (i.e., sampling with replacement) from the original dataset [9].

There are also cases where different α (importance) weights are assigned to each classifier. For classifiers of different types such an assignment may stem from prior knowledge of their strength, but more generally they can be learned, e.g., in a stacking framework. Here, individual classifiers are trained first and then applied to held-out validation instances. These outcomes are then used as input features to a gating or combiner model that learns the values of α.

Randomization does not need to be restricted to choosing which sets of instances are used to learn which model. More generally, it can also be used to decide which subset of features is available to a model. This is particularly relevant to high dimensional problems. Lowering the dimensionality of the feature subspace seen by a model not only speeds up the induction process but also makes the model more robust by preventing the dominance of a few strong features over the rest. Random forests [11] are an example of models that are particularly well suited to randomizing both the feature space and the instance space; they are also well suited to parallel implementations since all trees in the forest are created independently from one another (unlike in boosting where tree induction is sequential).

One current limitation of Pig is that it is not well suited for iterative processing. There have been recent efforts to adapt MapReduce to iterative-style algorithms (e.g., boosting [39]), as well as similar efforts using different cluster-based architectures (e.g., MPI [46]). We elected to stay within the confines of Pig, and it will be interesting to see when simple acyclic dataflows are no longer sufficient.

6. SENTIMENT ANALYSIS APPLICATION

In this section, we present an application of our machine learning tools to the problem of sentiment analysis. Although the problem is intrinsically interesting, our discussion primarily exists to illustrate the various features of our machine learning framework and show how all the pieces "come together".

6.1 Methodology

Sentiment analysis, and more broadly, opinion mining, is an area of natural language processing that has received significant interest in recent years; Pang and Lee provide a nice overview [40]. These technologies have also seen widespread commercial interest, particularly as applied to social media: sentiment analysis and related technologies promise solutions to brand and customer relations management, as well as insights into consumer behavior in the marketplace. Sentiment analysis applied to tweets has naturally received attention [38, 36, 25]. In contrast to previous approaches, which use some form of linguistic processing, we adopt a knowledge-poor, data-driven approach. It provides a baseline for classification accuracy from content, given *only* large amounts of data.

More specifically, we tackle the binary polarity classification task. That is, given a tweet known in advance to express some sentiment, the classifier's task is to predict $y_i \in \{\textsc{Negative}, \textsc{Positive}\}$. To generate labeled training data for polarity classification, we use the well-known "emoticon trick".[8] That is, we simply assume that tweets with positive emoticons, e.g., :-) and variants, are positive training instances, and tweets with negative emoticons, e.g., :-(and variants, are negative training instances. Obviously, these assumptions are not completely valid and do not capture phenomena such as sarcasm, irony, humor, etc., but, overall, data gathered in this manner are quite reasonable. For illustrative purposes, the "emoticon trick" is typical of a mechanism for generating a large number of labeled, albeit noisy, training examples.

We prepared a test set consisting of one million English tweets with emoticons from Sept. 1, 2011, at least 20 characters in length. The test set was selected to contain an equal number of positive and negative examples. For training, we prepared three separate datasets containing 1 million, 10 million, and 100 million English training examples from tweets before Sept. 1, 2011 (also containing an equal number of positive and negative examples). In preparing both the training and test sets, emoticons are removed.

Our experiments used a simple logistic regression classifier learned using online stochastic gradient descent (as described in Section 5), using hashed byte 4-grams as features. That is, the feature extractor treated the tweet as a raw byte array, moved a four-byte sliding window along

[8] It is not exactly clear who "discovered" this trick. Pak and Paroubek [38] is the earliest reference in the academic literature we could find, although Twitter search has had the ability to retrieve tweets by emoticons since the very beginning.

the array, and hashed the contents of the bytes, the value of which was taken as the feature id. Features were treated as binary (i.e., feature values were always one, even if the tweet contained multiple occurrences of the same byte 4-gram). Thus, we made no attempt to perform any linguistic processing, not even word tokenization.

For each of the $\{1, 10, 100\}$ million example datasets, we trained a single logistic regression model with all the data (i.e., one reducer at the end received all training instances). For the $\{10, 100\}$ million example datasets, we additionally experimented with ensembles of different sizes. For each condition, we conducted 10 different trials—the dataset was fixed, but each trial yielded a random shuffle of the training examples and the partitions (for ensembles). Each learned model was then evaluated on the 1 million instance held-out test set. For evidence combination in the ensemble experiments, the final prediction was made by taking into account the probabilities of individual classifiers (but otherwise the importance weights assigned to each classifier α was simply 1). We measured accuracy of the predictions—in our case, accuracy is exactly the same as precision and recall, since this is a forced-choice binary classification problem. Because our training and test sets were balanced by construction, the baseline is 0.5 (random guessing).

Table 1 shows the Pig script for training the sentiment classifiers. We begin by loading tweets using a specialized Pig loader (which handles uncompressing and unmarshalling the data). A randomly generated value is associated with each tweet (used later for sampling) and the instances are filtered to exclude non-English tweets based on a UDF that performs language identification. The training script then splits into two separate branches, one handling positive examples, and the other handling negative examples. In each, the label is generated along with the tweet (sans emoticons). A fixed number of examples is then selected using the random values previously generated. Finally, the positive and negative examples are unioned together: generation of random numbers and then sorting by them shuffles the positive and negative examples together.

In the training script, processed data are directly fed into the `TextLRClassifierBuilder` storage function that wraps the actual learner (logistic regression with SGD). As a common-case optimization, the feature extractor is folded into the learner itself to avoid materializing features. An entirely equivalent method would be to explicitly apply the feature extractor, then feed into the `FeaturesLRClassifierBuilder`. The variable `$PARTITIONS` controls the number of reducers in the final Hadoop job, and therefore the size of the ensemble built. Our actual experiments, however, followed a slightly different procedure: to conduct multiple trials on the same dataset, we first materialized the $\{1, 10, 100\}$ million training instances on HDFS. Scanning over large numbers of tweets to sample is fairly expensive, so we avoided resampling on every trial. For experiments involving 10 and 100 million instances, the instances were materialized in smaller chunks, which we then combined using the Pig `union` operator. Each trial involved a random shuffle of the training set (i.e., sorting by randomly generated values).

Note that to productionize this simple example would not require much more effort. It would consist of adding Oink bindings, which include specifying data dependencies (for example, that this script depends on tweets) and additional metadata such as job frequency. After that, Oink would handle model training at the correct periodicity, materializing results to a known HDFS directory (which can then be picked up by other systems). In addition, all alerting, monitoring, fault handling, etc. would come "for free".

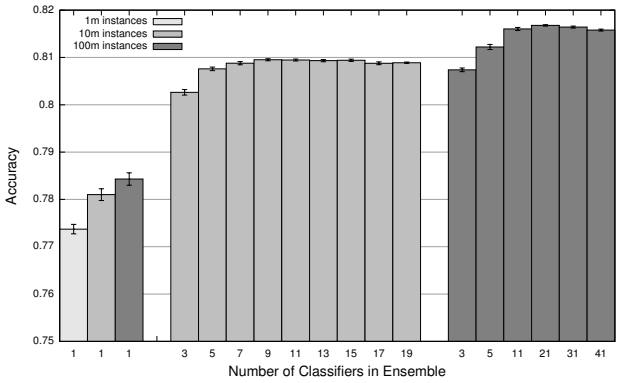

Figure 2: Accuracy of our tweet sentiment polarity classifier on held out test set of 1 million examples. Each bar represents 10 trials of a particular setting, with $\{1, 10, 100\}$ million training examples and varying sizes of ensembles. Error bar denote 95% confidence intervals.

Testing the learned models is accomplished by the script shown in Table 2. The `TextScoreLR` UDF is initialized with the model. We have another UDF called `EnsembleTextScoreLR` for ensembles; it is initialized with a directory containing multiple models as well as the evidence combination mode. The UDF outputs $p(\text{POSITIVE}|x)$ if the predicted label is POSITIVE or $-p(\text{NEGATIVE}|x)$ if the predicted label is NEGATIVE. For `EnsembleTextScoreLR`, the evidence combination logic is handled within the UDF, and so it has the same output behavior as `TextScoreLR`. After running the classifier on all the test instances, the Pig script makes hard classification decision, compares with the ground truth labels, and separately tallies the number of correct and incorrect predictions.

6.2 Results

Results of our polarity classification experiments are shown in Figure 2. The bars represent mean across 10 trials for each condition, and the error bars denote 95% confidence intervals. The leftmost group of bars show accuracy when training a single logistic regression classifier on $\{1, 10, 100\}$ million training examples. We get an accuracy boost going from 1 to 10 million examples, but a smaller increase moving from 10 to 100 million examples.

The middle and right group of bars in Figure 2 show the results of learning ensembles. In the middle we present the 10 million examples case, with ensembles of $\{3, 5, 7 \ldots 19\}$ separate classifiers. On the right we present the 100 million examples case, with ensembles of $\{3, 5, 11, 21, 31, 41\}$ classifiers. As expected, ensembles lead to higher accuracy—and note that an ensemble trained with 10 million examples outperforms a single classifier trained on 100 million examples.

We are not able to report accurate running times because all our experiments were run on a production cluster with concurrently-running jobs of different types and sizes. However, informal observations confirm that the algorithms behave as they should: ensembles take shorter to train because

```
status = load '/tables/statuses/$DATE' using TweetLoader() as (id: long, uid: long, text: chararray);

status = foreach status generate text, RANDOM() as random;
status = filter status by IdentifyLanguage(text) == 'en';

-- Filter for positive examples
positive = filter status by ContainsPositiveEmoticon(text) and not ContainsNegativeEmoticon(text)
             and length(text) > 20;
positive = foreach positive generate 1 as label, RemovePositiveEmoticons(text) as text, random;
positive = order positive by random;   -- Randomize ordering of tweets.
positive = limit positive $N;          -- Take N positive examples.

-- Filter for negative examples
negative = filter status by ContainsNegativeEmoticon(text) and not ContainsPositiveEmoticon(text)
             and length(text) > 20;
negative = foreach negative generate -1 as label, RemoveNegativeEmoticons(text) as text, random;
negative = order negative by random;   -- Randomize ordering of tweets
negative = limit negative $N;          -- Take N negative examples

training = union positive, negative;

-- Randomize order of positive and negative examples
training = foreach training generate $0 as label, $1 as text, RANDOM() as random;
training = order training by random parallel $PARTITIONS;
training = foreach training generate label, text;

store training into '$OUTPUT' using TextLRClassifierBuilder();
```

Table 1: Pig script for training binary sentiment polarity classifiers. The script processes tweets, separately filtering out those containing positive and negative emoticons, which are unioned together to generate the final training set. As detailed in Section 4.3, the learner (in this case, SGD logistic regression) is embedded inside the Pig store function, such that the learned model is written directly to HDFS.

```
define TextScoreLR TextScoreLR('hdfs://path/model');

data = load 'testdata' using PigStorage() as (label: int, tweet: chararray);
data = foreach data generate label, (TextScoreLR(tweet) > 0 ? 1 : -1) as prediction;

results = foreach data generate (label == prediction ? 1 : 0) as matching;
cnt = group results by matching;
cnt = foreach cnt generate group, COUNT(results);

-- Outputs number of incorrect and correct classification decisions
dump cnt;
```

Table 2: Pig script for evaluating the trained sentiment polarity classifiers. Note that the classifier is wrapped in a Pig UDF that is applied to test instances. Classification decisions are checked against ground truth labels; grouping and counting arrives at the final correct and incorrect predictions.

models are learned in parallel, and there is no single machine onto which all data must be shuffled. In terms of applying the learned models, running time increases with the size of the ensembles—since an ensemble of n classifiers requires making n separate predictions.

7. STATUS AND FUTURE WORK

Work began on Pig integration of machine learning tools in Spring 2011, using existing code that was refactored to comprise our core Java library. Although code for machine learning had existed long before that, this represented the first attempt to pull together a common set of libraries that would be shared across the entire company. Working prototypes were quickly completed, and we had our first production deployment during Summer 2011. Currently, there are a handful of machine-learned solutions running in production; applications include spam analysis, ranking functions for user search, and linked-based friend suggestions.

Experiences with the framework have been very positive—the learning curve for engineers and data scientists who already work with Pig on a daily basis is very shallow. It involves pointing people at internal wiki pages, using which they quickly come up to speed. As is common in devising machine learning solutions, most of the creativity lies in formulating the problem, and most of the programming effort is spent implementing feature extractors. For the first, there is no substitute for domain understanding and insight. This independent of our machine learning framework itself, since large classes of non-machine-learning problems also require implementing custom UDFs. Using our machine learning framework is mostly a matter of learning a few Pig idioms; the rest feels just like typical analysis tasks. Put it another way: machine learning becomes a natural extension of data science, where insights gleaned from data are operationalized in computational models.

From what we can tell, broader adoption of our machine learning framework is not limited by inherent capabilities of our design, but rather by the "readiness" of groups in the organization to embrace machine-learning solutions. We observe that groups often progress through a series of "phases" when tackling complex problems—usually beginning with manual intervention, hand-crafted rules, regular expression patterns, and the like. There is, of course, nothing inherently wrong with such solutions, and through them one learns a great deal about a problem domain. However, growing complexity beyond that which can be humanly managed eventually compels groups to seek alternative solutions, based on machine learning approaches or hybrid solutions that combine elements from automatic and manual techniques. Thus, the challenges in moving toward predictive analytics are cultural more than technical. We are presently actively engaging individuals and groups within the organization to evangelize both machine learning solutions in general and our approach in particular.

In terms of building out additional capabilities, we are adopting a pragmatic approach of implementing new features only when there is a need—we believe strongly in letting the problem drive the tools. Nevertheless, looking into the future we can imagine other types of machine learning tasks being integrated into Pig in much the same way. Although the focus of this paper has been on classification, regression and ranking can receive the same treatment. The same goes with clustering. However, without a concrete application to drive the development, tools will forever remain solutions in search of problems.

8. CONCLUSION

As the cost of storage and processing continues to drop, organizations will accumulate increasing amounts of data from which to derive insights. Inevitably, the sophistication of analyses will increase over time. Business intelligence tasks such as cubing to support "roll up" and "drill down" of multi-dimension data are already commonplace, with mature best practices both in the context of traditional data warehouses and Hadoop-based stacks. We are, however, witnessing the transition from simple *descriptive* analytics to more powerful *predictive* analytics, which promises to unlock greater troves of insights. There has not yet emerged a consensus on architectures and best practices for these types of activities. Nevertheless, we hope that through the accumulation of experiences, the community will converge on a body of shared knowledge. We hope that our experiences in integrating machine learning tools in a Pig-centric analytics environment contribute to this end goal.

9. ACKNOWLEDGMENTS

We would like to recognize Abdur Chowdhury, former chief scientist at Twitter, without whom this work would not be possible. We are also indebted to the countless engineers at Twitter whose efforts we build upon, especially the analytics team: the authors get the recognition, but they deserve the credit. Finally, we'd like to thank the anonymous reviewers for detailed comments that have greatly improved this paper.

10. REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *VLDB*, 2009.

[2] A. Agarwal, O. Chapelle, M. Dudik, and J. Langford. A reliable effective terascale linear learning system. arXiv:1110.4198v1, 2011.

[3] K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and E. Paulson. Efficient processing of data warehousing queries in a split execution environment. *SIGMOD*, 2011.

[4] M. Banko and E. Brill. Scaling to very very large corpora for natural language disambiguation. *ACL*, 2001.

[5] R. Bekkerman and M. Gavish. High-precision phrase-based document classification on a modern scale. *KDD*, 2011.

[6] C. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag, 2006.

[7] L. Bottou. Large-scale machine learning with stochastic gradient descent. *COMPSTAT*, 2010.

[8] T. Brants, A. Popat, P. Xu, F. Och, and J. Dean. Large language models in machine translation. *EMNLP*, 2007.

[9] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[10] L. Breiman. Arcing classifiers. *Annals of Statistics*, 26(3):801–849, 1998.

[11] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[12] E. Chang, H. Bai, K. Zhu, H. Wang, J. Li, and Z. Qiu. PSVM: Parallel Support Vector Machines with incomplete Cholesky factorization. *Scaling up Machine Learning: Parallel and Distributed Approaches*. Cambridge University Press, 2012.

[13] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *OSDI*, 2006.

[14] J. Cohen, B. Dolan, M. Dunlap, J. Hellerstein, and C. Welton. MAD skills: New analysis practices for big data. *VLDB*, 2009.

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

[16] C. Dyer, A. Cordova, A. Mont, and J. Lin. Fast, easy, and cheap: Construction of statistical machine translation models with MapReduce. *StatMT Workshop*, 2008.

[17] U. Fayyad and K. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. *IJCAI*, 1993.

[18] A. Gates. *Programming Pig*. O'Reilly, 2011.

[19] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of MapReduce: The Pig experience. *VLDB*, 2009.

[20] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on MapReduce. *ICDE*, 2011.

[21] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, 2009.

[22] J. Hammerbacher. Information platforms and the rise of the data scientist. *Beautiful Data: The Stories Behind Elegant Data Solutions*. O'Reilly, 2009.

[23] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009.

[24] T. Joachims, L. Granka, B. Pan, H. Hembrooke, F. Radlinski, and G. Gay. Evaluating the accuracy of implicit feedback from clicks and query reformulations in Web search. *ACM TOIS*, 25(2):1–27, 2007.

[25] E. Kouloumpis, T. Wilson, and J. Moore. Twitter sentiment analysis: The good the bad and the OMG! *ICWSM*, 2011.

[26] L. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. Wiley-Interscience, 2004.

[27] H. Li. *Learning to Rank for Information Retrieval and Natural Language Processing*. Morgan & Claypool, 2011.

[28] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool, 2010.

[29] J. Lin, D. Ryaboy, and K. Weil. Full-text indexing for optimizing selection operations in large-scale data analytics. *MAPREDUCE Workshop*, 2011.

[30] Y. Lin, D. Agrawal, C. Chen, B. Ooi, and S. Wu. Llama: Leveraging columnar storage for scalable join processing in the MapReduce framework. *SIGMOD*, 2011.

[31] LinkedIn. Data infrastructure at LinkedIn. *ICDE*, 2012.

[32] G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. Walker. Efficient large-scale distributed training of conditional maximum entropy models. *NIPS*, 2009.

[33] R. McDonald, K. Hall, and G. Mann. Distributed training strategies for the structured perceptron. *HLT*, 2010.

[34] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan. Distributed cube materialization on holistic measures. *ICDE*, 2011.

[35] A. Ng, G. Bradski, C.-T. Chu, K. Olukotun, S. Kim, Y.-A. Lin, and Y. Yu. Map-reduce for machine learning on multicore. *NIPS*, 2006.

[36] B. O'Connor, R. Balasubramanyan, B. Routledge, and N. Smith. From Tweets to polls: Linking text sentiment to public opinion time series. *ICWSM*, 2010.

[37] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. *SIGMOD*, 2008.

[38] A. Pak and P. Paroubek. Twitter as a corpus for sentiment analysis and opinion mining. *LREC*, 2010.

[39] B. Panda, J. Herbach, S. Basu, and R. Bayardo. MapReduce and its application to massively parallel learning of decision tree ensembles. *Scaling up Machine Learning: Parallel and Distributed Approaches*. Cambridge University Press, 2012.

[40] B. Pang and L. Lee. Opinion mining and sentiment analysis. *FnTIR*, 2(1-2):1–135, 2008.

[41] D. Patil. *Building Data Science Teams*. O'Reilly, 2011.

[42] D. Sculley, M. Otey, M. Pohl, B. Spitznagel, J. Hainsworth, and Y. Zhou. Detecting adversarial advertisements in the wild. *KDD*, 2011.

[43] Y. Singer and N. Srebro. Pegasos: Primal estimated sub-gradient solver for SVM. *ICML*, 2007.

[44] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. *VLDB*, 2010.

[45] R. Snow, B. O'Connor, D. Jurafsky, and A. Ng. Cheap and fast—but is it good? Evaluating non-expert annotations for natural language tasks. *EMNLP*, 2008.

[46] K. Svore and C. Burges. Large-scale learning to rank using boosted decision trees. *Scaling up Machine Learning: Parallel and Distributed Approaches*. Cambridge University Press, 2012.

[47] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive—a petabyte scale data warehouse using Hadoop. *ICDE*, 2010.

[48] M. Weimer, T. Condie, and R. Ramakrishnan. Machine learning in ScalOps, a higher order cloud computing language. *Big Learning Workshop*, 2011.

[49] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical Report UCB/EECS-2011-82, Berkeley, 2011.