Supporting Fast Iteration in Model Building

Manasi Vartak
MIT CSAIL
Cambridge, MA 02139
mvartak@csail.mit.edu

Pablo Ortiz
MIT
Cambridge, MA 02139
portiz@mit.edu

Kathryn Siegel
MIT
Cambridge, MA 02139
ksiegel@mit.edu

Harihar Subramanyam
MIT
Cambridge, MA 02139
hsubrama@mit.edu

Samuel Madden
MIT CSAIL
Cambridge, MA 02139
madden@csail.mit.edu

Matei Zaharia
MIT CSAIL
Cambridge, MA 02139
matei@mit.edu

Abstract

Building real-world machine learning models involves an iterative process of engineering features, fitting models, and tuning parameters until a model that meets specific acceptance criteria is identified. Over the course of a modeling session, tens to hundreds of models may be built and evaluated. Although the modeling process is iterative, current tools are optimized for individual models as opposed to modeling sessions. In this paper, we argue the need for a system to support the entire modeling process by making it cheap to run and track experiments. We describe the desiderata for such a system spanning systems optimizations to visual interfaces, and describe the ongoing implementation of SHERLOCK, a system optimized for iterative model building.

1 Introduction

Building a real-world machine learning model is an iterative[1] process. Starting with one or more initial hypotheses, a data scientist builds models to test his/her hypotheses, evaluates them, refines them, and repeats this process until he/she finds a model that meets some acceptance criteria (e.g., based on accuracy, interpretability etc.) During the modeling process, the data scientist runs experiments with different types of models (such as logistic regression, SVMs, gradient boosted trees), feature sets (generated through feature engineering), and hyperparameters. Regardless of industry, size of company, and field of research, we find that the process of modeling is the same, succinctly summarized by one data scientist as "rinse-and-repeat." Over the course of a *modeling session*, which starts with the first model being built and ends with the final model put into production, the data scientist tests many tens, if not hundreds, of models. In the field, these modeling sessions can take anywhere from a week to months. To enable scientists in research or industry to leverage the large amount of data at their disposal, we must provide analysts tools to accelerate the modeling process. This means that machine learning systems should optimize the building of not just a single model, but hundreds of models across a modeling session.

Although the process of modeling is known to be iterative, current machine learning tools as well as most research is geared towards building a single model efficienctly. For example, SystemML [2] introduces a higher-level language to express various ML algorithms that can be optimized and executed as MapReduce jobs. Scikit-learn [6], the popular machine learning toolkit in Python, supplies implementations for a variety of machine learning algorithms but provides limited support for fast

[1]iterative as in *repeating*; not to be confused with the number of iterations required by a particular training algorithm

iteration. MLlib and Spark ML [5] provide similar functionality for Apache Spark. Systems such as GraphLab [4] provide efficient, parallel implementations of various machine learning algorithms, while commercial systems such as AzureML provide packaged implementations of various algorithms and feature extraction techniques. Recently, some work (e.g. [10, 9]) has started to look at how database techniques may be used to speed up machine learning. Similar in motivation, also, is recent work such as [8] on supporting iterative probabilistic programming through an IDE. With respect to supporting modeling sessions, however, current tools suffer from the following limitations:

- For the most part, exploration of models takes place *sequentially*. Data scientists must write (repeated) imperative code to explore models one after another.
- Since training of a single model can take anywhere from many minutes to overnight runs, sequential exploration multiplicatively prolongs the time required to build an acceptable model.
- Without a history of previously trained models, each model must be trained from scratch, wasting computation and increasing response times.
- In the absence of history, comparison between models becomes challenging and requires the data scientist to write special-purpose code.
- There is no way to quickly evaluate the impact of changes to models, making experimentation expensive in terms of computation as well as time.

In view of these drawbacks, we believe that building a machine learning system that *treats model iteration as a first-class citizen* and provides tooling to support modeling sessions can help data scientists quickly perform experiments and arrive at usable models.

Our contributions with this paper are three-fold. (1) We argue that while optimizing individual model training is important, usage patterns suggest that supporting modeling sessions can magnify the gains obtained by faster training algorithms. (2) Towards the goal of building systems that support entire modeling sessions, we describe the requirements from such a system and challenges associated with building it. (3) We describe the design and initial implementation of SHERLOCK, a machine learning system we are building to enable fast, iterative modeling.

2 Desiderata for a System Optimized for Modeling Sessions

Iteration in the context of modeling involves building models that are variations of one another with respect to types of algorithms, feature sets, and hyperparameters. Since a modeling session involves the building of many tens to hundreds of models, the primary goals of a system optimized for iteration are two fold: (a) make it cheap to run modeling experiments (with respect to time and computation), and (b) provide means to manage the process by tracking experiments and allowing meta-analyses. Specifically, a system optimized for iteration should permit the following functionality:

1. *Efficient Training of Multiple Models at Once.* Most ML toolkits today allow the training of a single model at a time. With external libraries, users may be able to train a handul of models in parallel. However, very few of the existing tools apply optimizations to share scans and computation across multiple models. The advantages of this functionality are obvious: instead of training models, say, with feature sets $\{A, B\}$ and $\{A, B, C\}$ independently, this optimization can enable related models to be trained (as well as evaluated) at the same time. While the system may pay a small price in increased latency, we expect this cost to be counterbalanced by higher throughput.
2. *Incremental Training and Reuse.* Most models built by analysts are variations of past ML models obtained by adding or removing features or changing hyperparameters. As a result, a system should be able to reuse information, e.g. model parameters (e.g. weight vectors), intermediate results, *most relevant* data points etc., from previously trained models to speed up training of new models. For instance, warm starts can enable a model to start from the results of a previously trained model, while boosting may be used to combine two previously trained models. Similarly, if models share a specific computationally expensive intermediate step, then storing intermediate results can speed up training of new models.
3. *Fast Evaluation or Shortcut Evaluation.* Since data scientists run a large number of experiments, a quick — albeit less accurate — evaluation can be sufficient for the user to

determine whether to invest time in running a full experiment. As a result, an iterative system should provide the ability to quickly guage the whether a potential refinement (e.g. new feature, different parameter value) is likely to improve a model. Some techniques for supporting fast evaluation include training and testing models on samples and using analytical techniques to estimate the impact of the change.

4. *History and Tracking of Models.* An often ignored aspect in modeling is tracking of all the hypotheses or models tested. The use of history is three-fold: (a) history can enable better optimization while training new models, (b) it can allow the analyst to perform comparisons across models and identify patterns, and (c) with enough history, the system may be able to eventually automate some aspects of the modeling process.
5. *Human-in-the-Loop.* A significant part of the modeling process is the analyst's creativity, knowledge, and skill in finding patterns. To support the analyst, a machine learning system must provide means to interact with the system at different stages. For example, using history as discussed above, the system must allow the user to easily examine and compare previously built models. Similarly, for models that take a long time to train, the system must provide progress information so the user can choose to alter or pre-empt models. Finally, the system must also provide means to diagnose models and understand the results.

While the above requirements are highly desirable in a machine learning system, we must address several challenges in building such as system: (a) systems and algorithms must be adapted to train and test many models simultaneously, (b) incremental training with samples, feature sets and hyper-parameters must be studied for different classes of algorithms, e.g. [3], (c) as with other analytical tasks, there is a large diversity in the kinds of models that users build and therefore developing a unified optimization algorithm is challenging, and (d) building a machine learning system that keeps the user in the loop requires us to address questions related to visualization and interface design.

3 SHERLOCK: a System for Iterative ML

In view of the desiderata from the previous section, we now describe our design of SHERLOCK, a machine learning system that we are developing to support fast, iterative modeling. Figure 1 shows the architecture for SHERLOCK. We chose to build SHERLOCK on top Spark ML [5] to take advantage of existing implementations of machine learning algorithms and the parallel processing capability of Spark. However, the same techniques can be used to implement such a system on top of scikit-learn and other platforms.

Our system consists of three main components: (a) the *ModelDB* dedicated to storing history and metadata about models; (b) the *Optimizer* responsible for determining the most efficient plan for training and testing new models; and (c) the *Visual Interface* providing a real-time dashboard for the modeling session and means to inspect models. SHERLOCK also exposes two distinct APIs: the *user-facing API* provides users a set of functions to easily specify and run modeling experiments, while the *developer API* specifies the set of functions that a machine learning algorithm must support in order to enable iterative model building. We briefly describe these components in the remainder of this section.

3.1 ModelDB

Many of the requirements for an iterative learning system are tied to having a central repository of models both for tracking modeling experiments and for reusing previous computation. The central repository in SHERLOCK is called ModelDB. ModelDB stores tables corresponding to model specifications (*schema* of a model), trained models, quality information (i.e. results from evaluations), and intermediate results from training models. The model specification and quality information are most relevant for performing comparisons across models and finding closely related models, while intermediate results indicate computations that can be reused.

3.2 Optimizer

Given a set of model specifications, the optimizer is responsible for determining the best plan to train and test these models. There are three main cases that the optimizer supports for training: (a)

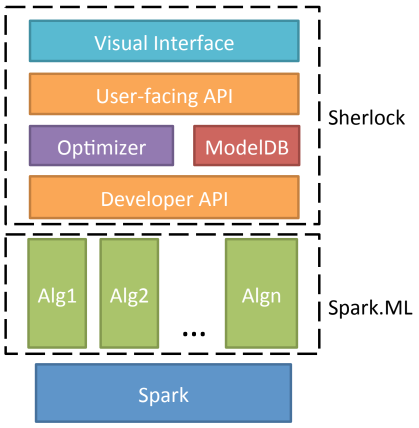


Figure 1: SHERLOCK *Architecture*

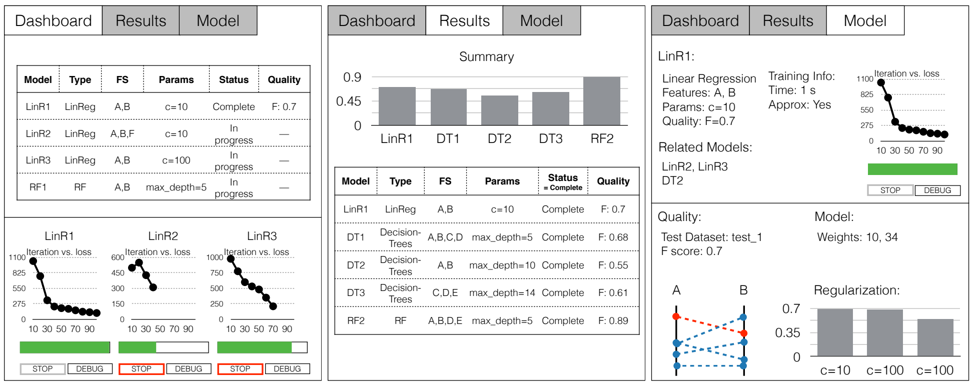
training multiple models at the same time (e.g., for hyperparameter tuning), (b) training variations of models based on previously trained models (e.g., with different feature sets), and (c) incrementally training models using more data (e.g., for quick experiments or for model updates). To support each of these cases, SHERLOCK reuses techniques from databases and online machine learning including multi-query optimization [7], warm starts, and model-specific techniques such as [3] for incremental processing. We are also developing new techniques that can incrementally update models in response to changing features and parameters. In addition to the above techniques that focus on efficient training, the optimizer optimizes testing of models during procedures such as cross-validation. We incorporate recent work on robust machine learning using differential privacy [1] to ensure that evaluations are statistically valid. SHERLOCK explicitly exposes the tradeoff between latency of training and accuracy of the model. Where possible, SHERLOCK provides progressive information about training and testing of models so the user can choose to pre-empt models with poor performance.

3.3 Visual Interface

A key requirement for an iterative learning system is to keep the analyst in the loop throughout the modeling session. The different parts of our system are tied together via a visual interface shown in Figure 2. The purpose served by the visual interface is three-fold. First, the visual interface provides a real-time dashboard (Figure 2a) of what models are being trained at the current time. Users can view current models, examine loss characteristics, and update or prune models based on performance. Second, it provides a means to explore previous experiments stored in the ModelDB. The user can query for specific models (e.g., those with a given set of features) and use a variety of visualizations to compare different models (Figure 2b). Last, the visual interface provides means to diagnose the performance of individual models via model-specific visualizations (e.g. Figure 2c).

3.4 SHERLOCK API

By examining typical examples of modeling workflows, we have identified two APIs, one user-facing and one developer-oriented, that capture key pieces of functionality for iterative modeling. The goal of the user-facing API is to make it easy to specify and run a large number of modeling experiments. For instance, with `createSpec`, we make it easy for a user to specify many models in a single line of code. On the other side, we have identified a set of functions that a machine learning algorithm must support to allow iterative modeling using that algorithm. Called the developer API, this API includes functions to incrementally update models given more data, different set of parameters, and different features. Table 1 shows a subset of functions from the SHERLOCK API.



Model	Type	FS	Params	Status	Quality
LinR1	LinReg	A,B	c=10	Complete	F: 0.7
LinR2	LinReg	A,B,F	c=10	In progress	—
LinR3	LinReg	A,B	c=100	In progress	—
RF1	RF	A,B	max_depth=5	In progress	—

Model	Type	FS	Params	Status = Complete	Quality
LinR1	LinReg	A,B	c=10	Complete	F: 0.7
DT1	Decision-Trees	A,B,C,D	max_depth=5	Complete	F: 0.68
DT2	Decision-Trees	A,B	max_depth=10	Complete	F: 0.55
DT3	Decision-Trees	C,D,E	max_depth=14	Complete	F: 0.61
RF2	RF	A,B,D,E	max_depth=5	Complete	F: 0.89

LinR1:

Linear Regression
Features: A, B
Params: c=10
Quality: F=0.7

Training Info:
Time: 1 s
Approx: Yes

Related Models:
LinR2, LinR3
DT2

Quality:
Test Dataset: test_1
F score: 0.7

Model:
Weights: 10, 34

(a) Dashboard *(b) Model Comparison* *(c) Model Diagnosis*

Figure 2: SHERLOCK *Visual Interface*

Type	Description	API
User-facing	Specify Models	createSpec(modelType: String, params1=[], featureSet={}...)
	Update Features	addFeatures(specs: Array[ModelSpec], new_features=[])
	Train Models	train(dataset: DataFrame, specs: Array[ModelSpec])
	Cross-Validation	cv(dataset: DataFrame, specs: Array[ModelSpec])
Developer	Multi-model Training	fit(dataset: DataFrame, specs: Array[ModelSpec])
	Training Model Variants	fit(dataset: DataFrame, old_model: Model, new_spec: ModelSpec)
		fit(dataset: DataFrame, old_model: Model, new_featureSet: FeatureSet)
	Incremental Training	fit(old_model: Model, new_data: DataFrame)

Table 1: SHERLOCK *APIs (subset)*

4 Conclusion

Building real-world machine learning models is an iterative process where the user builds and evaluates a large number of models before arriving at one that meets the acceptance criteria. Current tools support efficient building only of single models as opposed to supporting the full modeling process. In this paper, we argue that while optimizing single models is important, usage patterns suggest that supporting entire modeling sessions is important and requires different tradeoffs from those used in traditional systems. We discuss the requirements that must be satisfied by an end-to-end modeling system including the ability to train many models at the same time, incremental training, and fast evaluation. As an example of an end-to-end system, we describe the design of SHERLOCK, a system currently under development. We discuss the APIs provided by our system, optimizations employed, and key system components.

References

[1] C. Dwork, V. Feldman, M. Hardt, T. Pitassi, O. Reingold, and A. Roth. The reusable holdout: Preserving validity in adaptive data analysis. *Science*, 349(6248):636–638, 2015.

[2] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 231–242. IEEE, 2011.

[3] P. Laskov, C. Gehl, S. Krüger, and K.-R. Müller. Incremental support vector learning: Analysis, implementation and applications. *The Journal of Machine Learning Research*, 7:1909–1936, 2006.

[4] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[5] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. Mllib: Machine learning in apache spark. *CoRR*, abs/1505.06807, 2015.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[7] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[8] S. Singh, S. Riedel, L. Hewitt, and T. Rocktaschel. Designing an ide for probabilistic programming: Challenges and a prototype. In *NIPS Workshop on Probabilistic Programming*, 2014.

[9] E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 368–380, New York, NY, USA, 2015. ACM.

[10] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 265–276. ACM, 2014.