Optimizing WahooML, A Query Optimizer for the Spark Machine Learning Pipeline

M.Eng Thesis Proposal

M.Eng. Candidate:	Faculty Advisor:	Mentor:
Kathryn Siegel	Sam Madden	Manasi Vartak
ksiegel@mit.com	madden@csail.mit.edu	mvartak@mit.edu
MIT	MIT Database Group	MIT Database Group

November 2015

Abstract

Existing machine learning tools allow data scientists to iteratively engineer features and train models. Much of a data scientist's time is spent determining the optimal model by tweaking dozens of inputs. WahooML is a data scientist productivity tool that expedites the process for discovering optimal models. The system optimizes the entire model development process by allowing data scientists to train sets of models, taking advantage of the parallels between models and storing trained models in an external database. My research will focus on several significant optimization components of WahooML: incremental training, k-folds cross validation, and ensemble models.

1 Introduction

A data scientist's pipeline for creating and training machine learning (ML) models is a multi-step, repetitive process. Specifically, the process involves defining the model specifications, training the model on a dataset, cross- validating the results on test data, tweaking the model specifications and/or dataset, and beginning the whole cycle over again. Eventually, a model is produced that meets a reasonably high standard of accuracy in the cross- validation step, and this model is the one that is selected for further use. Because of this cyclical iteration pattern, existing machine learning tools require users to test model hypotheses sequentially. Users must train one model after the other, manually specifying parameters such as the datafolds and the number of training iterations. This manual component incurs a significant time cost.

WahooML is a system that accelerates this iterative process by optimizing model training runs and allowing multiple models to be trained together. The system also exposes a developer API that allows users to specify custom models and estimators. WahooML revolves around the concept of a "hypothesis," which refers to an insight that a data scientist is trying to reach by training models on a set of data. WahooML accepts a full data set and the specifications of the training model. The query optimizer component of the system then optimally groups and tests datafolds such that it finds the ideal output in the least computationally expensive way possible. The ModelDB component of the system stores intermediate computation, as well as the outputted model, such that WahooML can strategically reuse past computation if it were to see an identical or similar model specification in the future. Finally, the WahooML user interface (UI) displays interesting visualizations of the entire process.

My work thus far has focused on creating a basic system that stores trained models in an external database, such that results do not have to be recomputed. My thesis work will mainly focus on the query optimizer component of the system. In this proposal, I describe the research and decisions that led to the current design of the system, as well as the optimizations I plan to study and implement in WahooML.

2 Previous Work

In this section, I provide an overview of the existing ML tools Weka and AzureML and evaluate the benefits and shortcomings of each tool. On the whole, WahooML will provide significant performance ad-

vantages over these popular existing tools.

2.1 Weka

Weka is a data mining program written in Java. It provides a GUI through which a user can import measurements and view them in a table. Through this GUI, data scientists can select machine learning classification algorithms and the parameters associated with these algorithms. One can also select which subset of features to use with an ML algorithm; there is a supervised learning feature to help users select attributes. Weka then runs the ML algorithm and reports detailed results of the model back to the user. Morever, users may write a new classifier or filter using Weka's Java API and use it alongside the already-existing tools. [5]

The system has an flexible, intuitive interface that provides ample data for data scientiests. However, there are several significant drawbacks to using Weka. Iterating on different feature sets and algorithms can be slow and cumbersome due to the manual selection process for data. A lot of information is provided by Weka, but Weka provides no guidance as to how that information could help training future models. Moreover, the system forces the user to conduct model refinement sequentially, without any method to avoid unnecessary recomputation. Overall, there are many components of Weka, such as the intuitive UI and wealth of outputted data, that we should emulate in WahooML. However, WahooML would provide benefits such as parallel model training and strategic storage to reduce overall computation time.

2.2 Azure ML

Azure Machine Learning is a system that allows users to run machine learning workflows. It provides a UI for users as well as cloud hosting and storage. Users interact with Azure ML by creating dataflow diagrams that detail experiments. Individual nodes in these experiments represent operations on data ranging from transformations to model training. Azure ML then processes the experiment and returns the results and computed statistics to the user. Users can programmatically create their own modules and import tham into Azure ML. [2]

Similarly to Azure ML, WahooML will provide data pipeline visualizations and allow users to integrate their custom code with the system. However, Azure ML limits user capabilities in several ways. The system forces users to use the Azure cloud instead of a custom hosting site, restricts custom additions to live within individual modules, and is very slow when compared to alternatives. WahooML will avoid these shortcomings, while also offering benefits such as warm-start runs and incremental training capabilities.

3 Initial Progress

In the first half of the semester, we have developed an initial prototype that caches trained models in a model database built using MongoDB. A web interface displays trained models and logging messages from Spark. In this section, I discuss the initial design discussions and the thought processes that helped formulate the preliminary system.

3.1 Spark-WahooML Integration

To integrate WahooML with Spark, we use the Spark package name and override methods in the Spark codebase by using traits and extending certain classes. Our implementation requires using the same package name as Spark ML, as we use and modify private constructors and methods only accessible from within the Spark ML package. Traits, also known as mix-ins, allow overriding methods of a class and adding attributes to a class. By using a combination of traits and custom classes that extend Spark classes, we can override the $fit()$ and $train()$ methods of the Spark ML Estimator subclasses and add additional functionality. From a user standpoint, this design is minimally cumbersome to integrate into an existing data analysis workflow, as the user merely has to use the WahooML version of a particular Estimator class; the rest of the training code stays the same. Additionally, this design avoids copying over any boilerplate code from Spark. We will have to copy some logic from Spark, but we can incrementally copy over this logic as we explore more and more optimizations.

We chose this design because it involves copying over the minimal amount of boilerplate code and complex logic from the Spark codebase. As the code stands now, we do not have any boilerplate code in the WahooML codebase that exists for the sole purpose of integration with Spark. Using traits allows us to add features and optimizations incrementally, which significantly sped up the development time of the MVP. WahooML also avoids needing to modify any method signatures in the Spark ML codebase and black-boxes Spark as a separate library.

3.2 ModelDB

The Model Database, or ModelDB, is built on top of MongoDB using the Casbah framework. For a Spark Estimator, the class used to train models, we present an analogous WahooML version. We have implemented WahooML logistic regression and WahooML linear regression classes, and intend to support further training algorithms in the future. These WahooML estimator classes extend their Spark ML counterparts, simply adding traits that call into the underlying database. This design choice lends itself well to adding additional WahooML Estimators, as the storage logic for a single estimator type is contained within that single estimator class.

The ModelDB trait, when extended by a WahooML Estimator class, checks the database for an already-trained model whenever the $fit()$ function is called. If a matching model is found, it is returned to the user. Otherwise, the $fit()$ function proceeds as usual, and before returning the trained model to the user, the system stores it in the database. For each model in the database, we store the fields encapsulated by the trained model, a database object representation of the model specification, and a hash of the dataset on which the model was trained. Doing so allows us to query for a trained model by its model specification and training dataset. Future research will investigate what other computation we can store in the database to reduce overall computation.

3.3 User Interface

The User Interface (UI) is built using Express and Node.js. The WahooML Spark jobs pass log messages to the scala WahooConnector class, which then sends these messages to the Node.js backend using POST requests. The application then uses Socket.io to push these log messages to the frontend. Users can browse the models in the ModelDB and see these log messages using the UI.

4 Proposed Work

I will continue research on WahooML for my thesis. In general, the WahooML development team will work on the three main components of the system, which are as follows.

1. The user interface (UI) is a web interface that allows users to launch Spark ML runs using WahooML optimizations and to monitor the progress of these runs.
2. The ModelDB stores a global repository of models with information about the trained model, the model specification, and the data used to train the model. My work thus far this semester has focused on creating the ModelDB, as described in the "Initial Progress" section.
3. The optimizer determines the best plan for training a group of models on a dataset. Optimizing a training run involves training many models in parallel and reusing computation among each parallel training run. Additionally, the optimizer will query the ModelDB for useful computational results and use this information to further reduce the necessary computation for the run.

My work will focus on the optimizer component of the system. First, I will research and implement optimizations for incremental training on increasing amounts of data and number of iterations. This will involve creating a dataframe wrapper module that captures changes in training data and tracks these changes in the model training process. Furthermore, we will define two separate APIs for users to access WahooML's features: (1) an API that trains the model on increasing amounts of data, strategically using both the dataframe wrapper module and results from previous runs, and (2) an API that trains models with a "warm start," i.e. begins training on an already-trained model with a specification involving fewer iterations.

Next, I will research optimizations for training models on many different folds of a dataset. A dataset is often split into K folds, with each model trained on a different subset of the K datafolds. Training these models simultaneously allows the system to reuse computation for each datafold.

Once the core system is built, I will research optimizations on ensemble models. Spark ML already supports the capability for ensemble models, so WahooML will provide optimizations via batched and reusable computation.

4.1 Incremental Data

A common pattern in a data scientist's workflow involves training the same model on increasing amounts of data. Existing machine learning tools retrain a model from scratch on the entire dataset when new data is added. We will avoid this large amount of recomputation by optimizing WahooML to reuse information from previous data runs.

At a high level, the system will use a new $addData()$ function, which will be a method in the

model classes and accept a DataFrame as a parameter. This function will train the model on additional data, changing the model parameters to reflect both the new data and the old data. The system will also store each incrementally trained model in the ModelDB. A workflow would work as follows.

1. The user trains the model with dataset DF1.
2. The user trains the same model on a set of additional rows, which make up dataset DF2. So, the model has been trained on DF1 + DF2. Both of these models are now in the ModelDB.

This design is specifically targeted towards the addition of rows and does not allow for any other types of data transform. Implementation will involve modifying the underlying machine learning library supporting Spark ML using traits or subclassing. We will need to enable this library, Breeze, to support "warm starts," or submitting an already-trained model to be trained further.

4.2 Incremental Training Runs

Another common pattern in a data scientist's workflow involves training a model with an increasing number of iterations, with the purpose of obtaining a more accurate model. Existing machine learning tools simply retrain all models from stratch, instead of taking advantage of models that may have been previously trained with fewer iterations. Our system will avoid this unnecessary computation by using "warm starts," or training runs initialized with an already-trained model.

At a high level, before training a given model X, WahooML will search in the model database for models with the same DataFrame and model specification as model X. The system will modify the search parameters to allow for results with model specifications containing a lesser number of training iterations than model X's specification. The system will find the difference between the number of iterations with which the stored model was trained and the number of iterations with which model X should be trained. It will then perform this delta number of iterations on the already-trained model. Since Spark does not explicitly provide for warm starts, this functionality must be added to the codebase.

4.3 K-Folds

A third common pattern when training models is splitting the total training dataset into multiple "folds" of data. Data scientists train models on subsets of folds and use cross validation to find the best model. Cross validation is an important tool to test whether a trained model is accurate on other data points on which it was not trained. A model's performance on the test fold is predictive of its performance on datasets with unknown categorization. Spark ML has a CrossValidator class that handles cross-validation and selection of the optimal model. Consider a dataset with three folds: A, B, and C. The typical process the Spark cross-validator uses to find an optimal model would proceed as follows.

1. A model is trained with folds A and B and is cross-validated on C.
2. The model is trained with folds B and C and is cross-validated on A.
3. The model is trained with folds A and C and is cross-validated on B.
4. The data scientist selects the model with the highest accuracy on the cross validation step.

As a result, a large number of folds implies an large number of subsets. Currently, each of the models must be trained serially, with each of K folds being used in some N<K subsets. WahooML will provide a more computationally efficient WahooCrossValidator class that extends the existing Spark CrossValidator class. The system will reuse training computation for the same fold across the different training runs conducted by the cross-validator. WahooML can also detect suboptimal models early on in the process and eliminate them, decreasing overall computation.

4.4 Ensemble Models

An ensemble model is a machine learning algorithm created from a set of other models. In Spark, an ensemble model is either a gradient-boosted tree or a random forest, both of which use decision trees as their base models. My research will focus on random forests, which combine sets of decision trees to minimize overfitting.

The algorithm for random forests first "bags," or randomly samples, the dataset; one decision tree is grown with each of these random samples. The algorithm uses some standard equation to combine the predictions of the individual decision trees. The most common implementation simply averages the individual outcomes. [1] This repetitive structure lends itself to several possible optimizations:

1. WahooML could store these trees in a compressed format in the ModelDB; the trees could be flattened into arrays of nodes. Because random forests tend to be dominated by sets of important variables, WahooML could

prune the decision trees before storage to conserve space.
2. The system could abort growing portions of decision trees that deal only with weak predictors; WahooML would learn information about weak predictors from other decision trees.
3. Since random forests inject randomness through bagging and through node splits, the system could assist a data scientist by running the training process multiple times and selecting the most predictive set of decision trees.
4. The system could assist users with tuning the training parameters. Spark allows users to specify four main parameters: the number of trees in the forest, the maximum depth of each tree in the forest, the subsampling rate used to select data points for each bag, and the number of features used to split the data at each node. WahooML could accept ranges of these parameters and train ensemble models with each possible combination of parameters, outputting the optimal ensemble model of the set. The system could share repetitive computation across runs with similar parameters to further decrease the computational load.

Because decision trees are one of the most common techniques used by data scientists, optimizations in training time will benefit users. WahooML will also automate some manual derivation of optimal parameters, resulting in more predictive models.

5 Benchmarks

We intend to evaluate the results and performance of our system using several open-source datasets. Websites that host these datasets include Kaggle and various university sites. We will evaluate the performance of WahooML by two main metrics: accuracy of the resulting model and the computation time taken to derive that model. So, we will need to roughly approximate the time needed to manually derive the optimal model via iterative training, and then compare this time to WahooML.

6 Timeline

My past and intended timeline for completing the work required for my Master's thesis is comprised of the following deadlines. Note that these deadlines are heavily based off of the WahooML team's development deadlines.

1. (Done) End of September - Conduct literature survey, familiarize myself with existing tools, set up Spark.
2. (Done) Mid-October - Complete interfaces for project.
3. (Done) Beginning of November - Demoable MVP with model storage in an external database.
4. (Done) Thanksgiving - Scope out individual work for the optimizer component of WahooML, finish Master's thesis proposal, complete design documents for training optimizations, and iterate on MVP implementation.
5. End of Semester - Implement optimizations for incremental data and incremental training iterations. Also implement the K-folds optimization.
6. End of IAP - Improve optimizations from November-December sprint, research ensemble models.
7. Spring Break (end of March) - Complete WahooML component optimizing ensemble models.
8. Mid-April - Wrap up work on system optimizations.
9. Mid-May - Finish Thesis

7 Conclusion

WahooML will aid data scientists by expediting the model building process. The system will allow users to quickly test hypotheses about optimal models via its batch training and model database features. My research will contribute several core optimizations to the system, allowing WahooML to support incremental training, batched k-folds cross-validation, and optimal ensemble model-finding. This proposal discussed the design of each optimization and how each will fit into the overall system.

8 References

1. Blau, Gerard. "Analysis of a Random Forests Model." Ed. Bin Yu. Journal of Machine Learning Research 13 (2012): 1063-095. Print.
2. "Machine Learning." - Predictive Analytics. N.p., n.d. Web. 15 Nov. 2015. <https://azure.microsoft.com/en-us/services/machine-learning/>.
3. "Spark." Spark Overview. Apache, n.d. Web. 15 Nov. 2015. <http://spark.apache.org/docs/latest/>.

4. Vartak, Manasi. Supporting Fast Iteration in Model Building. Tech. N.p.: n.p., n.d. Web.
5. "Weka 3: Data Mining Software in Java." Weka 3. N.p., n.d. Web. 15 Nov. 2015. <http://www.cs.waikato.ac.nz/ml/weka/>.