Journal of Machine Learning Research todo (2017) todo Submitted todo; Published todo

Marvin - Open source artificial intelligence platform

Lucas B. Miguel LUCAS.BONATTO@B2WDIGITAL.COM
Daniel Takabayashi DANIEL.TAKABAYASHI@B2WDIGITAL.COM
Jose R. Pizani JOSE.PIZANI@B2WDIGITAL.COM
Tiago Andrade TIAGO.ANDRADE@B2WDIGITAL.COM
Brody West BRODYW@MIT.EDU

Editor: Claire Hardgrove, Keiran Thompson and Louis Dorard

Abstract

Marvin is an open source project that focuses on empowering data science teams to deliver industrial-grade applications supported by a high-scale, low-latency, language agnostic and standardized architecture platform, while simplifying the process of exploration and modeling. Building model-dependent applications in a robust way is not trivial, one is required to have knowledge in advanced areas of sciences like computing, statistics and math. Marvin aims at abstracting the complexities in the creation process of scalable, highly available, interoperable and maintainable predictive software.

Keywords: Machine Learning, Platform, Predictive Analytics, Artifical Intelligence

1. Introduction

Being able to quickly identify hidden patterns in datasets, wisely choose the best model to train from historical data, and making predictions are the biggest advantages of data-driven organizations against their competitors. Knowing the customer demand for a product before buying it (Chen et al., 2000) or being able to detect a fraud before charging the customer's credit card (Chan and Stolfo, 1998) with a certain level of confidence are examples of how companies doing businesses on the internet are making better decisions and maximizing their earnings.

Capturing and storing large amounts of data is a commonplace for most companies these days. Having more data available is often a positive aspect in order to train models with a lower error rate. However, writing code to effectively process terabytes of data and provide near-real-time predictions supporting high throughput is not a trivial task. One is required to have knowledge in advanced areas of science, such as computing, statistics and math. High scale data processing frameworks (Zaharia et al., 2010) fulfill their role by abstracting some of the complexities related to distributed computing and process orchestration. Libraries like MLLib (Meng et al., 2015) and scikit-learn (Pedregosa et al., 2011) facilitate it by providing high level interfaces to common machine learning algorithms. Even so, building robust model-dependent applications is tricky and requires specialized knowledge. There is an open space for a platform that empowers the data scientist with the tools

and abstractions needed to create scalable, highly available, interoperable and maintainable predictive software.

In this paper we present Marvin (*https://github.com/marvin-ai*), an open source platform that aims to help data scientists with several tasks during the life cycle of an artificial intelligence project. In section 2 we describe the platform's main features, section 3 contains implementation details, section 4 has a sample application and section 5 shows or experiment results. Finally in section 6 we talk about future work and summarize our findings in section 7.

2. Marvin Overview

A model-dependent application is described by an application that processes historical data and train a mathematical model such that the output Y can be explained by the different values of the input X. Some categories of algorithm in the artificial intelligence area fits in this description, e.g. support vector machines, statistical AI, neural networks. Marvin provide tools that will help in different phases of a project with such characteristics. The tools can be executed both on-premise or in a cloud infrastructure, giving the flexibility that is needed for different scenarios. Marvin is different from ML PaaS (AzureMLTeam, 2016), since it can be executed on-premise and allow the use of any algorithm that can be implemented in general-purpose languages (Van Rossum and Drake, 2003) (Odersky et al., 2004). In fact, Marvin could be used as the back-end of such type of platforms.

Marvin was built to enforce the pillars of the basic phases of a model development project, see Figure 1.

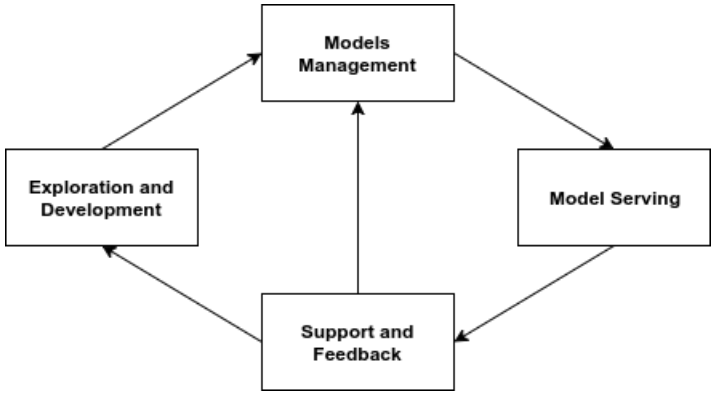

Figure 1: Model application development cycle.

To achieve that, Marvin provide several features, those include:

1. Toolbox - Provide a great set of common tools that are commonly used during the exploration phase of a data science projects and will eventually be carried up to production. The data scientist can take advantage of notebooks, plotting libraries, data frames and so on.

2. Experiment Versioning - During the exploration and exploitation of the problem it is common to test several hypothesis and eventually change approach. Keeping the history of experiments is useful and may serve to explain the final solution.

3. Data Sync CLI - During the model development data is intensively accessed, either to perform feature engineering or backtest the model. In complex environments it's often not a good approach to access production databases during development phase. To solve that Marvin provide a tool to sample data from the official dataset and work locally.

4. Unit Test Framework - In order to ensure a testable application, Marvin provides a built-in unit test framework, encouraging the data scientist to avoid bugs being introduced in the model application in the future.

5. Project Generator - Marvin introduces a design pattern, see Figure 5, to ensure that applications will be built in a decoupled manner. The project generator utility creates the base skeleton for applications, the data scientist is required to only populate the skeleton files with their logic.

6. Artifact Versioning - Marvin keeps track of artifacts generated during the training pipeline. When running the application in production Marvin allows the user to restore the system to a previous state, i.e. publishing a model trained last week because the current model is presenting greater error rate.

7. Large Dataset Processing - Integration with main frameworks for parallel and distributed computing to allow the effective handling of large datasets.

8. Training Pipeline Interface - The phases involved in the training pipeline (data acquisition, data preparation, training) can be started through the CLI or via REST HTTP calls. Allowing the application to be executed by external agents.

9. Feedback Server - In order to allow the model to receive external signals, Marvin provides a feedback server. User and applications can send feedback data to the model, which can be interpreted and perharps start a new training.

10. Predictor Server - When finishing the training pipeline Marvin persists the serialized model in a persistent memory storage. This model is loaded afterwards by the predictor server and is accessible via REST HTTP calls. Users and applications can then make predictions taking advantage of the model.

Marvin is composed by three main components, see Figure 2. The toolbox is both a command line interface and a library that contains a set of utilities to help data scientists during the phase of exploration and development. Using the toolbox the user will build his application, which in the context of Marvin is called an engine. The engine must be built following a design pattern proposed by Marvin, the toolbox will generate a scaffold with the classes corresponding to this pattern. Each phase in the pattern will produce an artifact, that can be persisted and reloaded. Lastly, the engine-executor is the component responsible of orchestrating the execution of engines and also by deploying servers to allow external interaction with it.

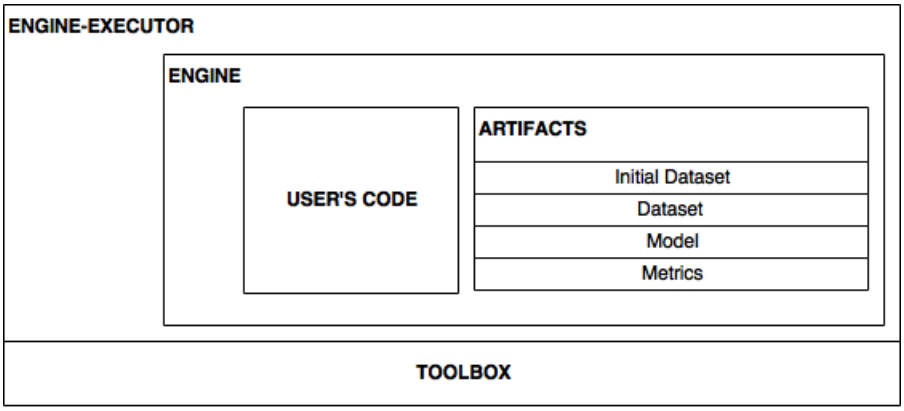

Figure 2: Marvin components.

Several previous work already help on the task of implementing artificial intelligence applications able to deal with large datasets and achieve good performance. SystemML (Ghoting et al., 2011) provide a high level declarative interface to implement machine learning applications that can process massive amounts of data. Pregel (Malewicz et al., 2010) introduces a computational model suitable for large-scale graph processing. OptiML (Sujeeth et al., 2011) is a domain-specific language (DSL) to achieve implicit parallelism on machine learning applications. MLI (Sparks et al., 2013) offers an API for distributed machine learning that helps turning prototypes into industry-grade ML software. Although these works do a great job abstracting complexity in the batch processing phase, they do not intend to offer tools to help during problem exploration and model serving. Table 1 shows the comparison of Marvin, SystemML and a market solution (Cloudera).

	Marvin	**SystemML**	**Cloudera DSW**
Multi-language support	x		x
HDFS support	x	x	x
Distributed CPU ML API	x	x	
Single node capability	x	x	
Toolbox (development environment)	x		x
Templates	x		x
Integrated notebooks	x		x
Models management	x		x
Data versioning control	x		
Pipeline scaffolding	x		
Unit test integration	x		
REST API	x		
Pre-process optimization		x	
Feedback server	x		

Table 1: Functionality comparison

3. Implementation Details

One of Marvin's main objective is to optimize execution of it's users' applications in order to process large datasets and allow a high number of concurrent model access without penalizing performance. To help users see the boundary of different executions flow within the application we propose the separation of actions in two categories:

- batch - e.g. train, evaluate, prepare
- online - e.g. predict, feedback

Batch actions are executed asynchronously and the result of it's execution will be an artifact, i.e. binary or plain text data. The generated artifacts can be persisted and re-used in the same application or with other applications instances. A practical use of this functionality can be to share the initial dataset artifact between different models, avoiding unnecessary duplicated computation. These characteristics help to perform effective long running and data-intensive jobs.

In order to achieve parallelism in different levels, Marvin applications run on top of common data processing frameworks, see Figure 3. The Marvin context is a component that frees the data scientist of setting up and optimizing the framework, it wraps some methods from the original framework library but also expose the main features of it.

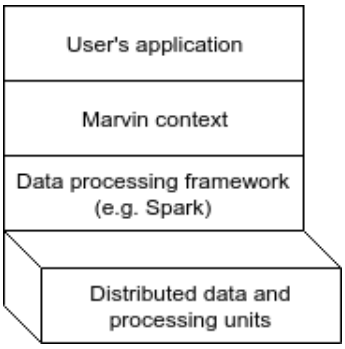

Figure 3: Multi-tier architecture for batch processing.

On the other hand, online actions are executed synchronously and they may generate a valid result. Their result will usually be interpreted by other application, therefore to ensure simple scalability, interoperability, availability and the needed consistency we adopted a microservice-based architecture (Brewer, 2000) (Fowler and Lewis, 2014).

All application's execution are orchestrated by the engine-executor component. This is a configurable component that can be deployed in different formats, depending on the environment complexity. One may want to deploy an engine-executor instance for each pipeline phase (data acquisition, data preparation, model training, etc.), it would avoid a single point of failure and allow independent scalability for each phase. However, projects on a smaller scale may prefer to deploy all the phases and the predictor server in the same instance. To achieve safe and effective concurrent execution, the underlying of engine-executor is implemented according to the Actor model (Hewitt et al., 1973).

As we understand that there is no mother programming language for artificial intelligence applications, we built Marvin under the assumption that it should be language agnostic. It means that users are free to implement their applications using their preferred language, provided that its support is implemented. The first Marvin's version supports Python language. Figure 4 shows how Marvin is using the RPC (Srinivasan, 1995) protocol to execute code written in different languages.

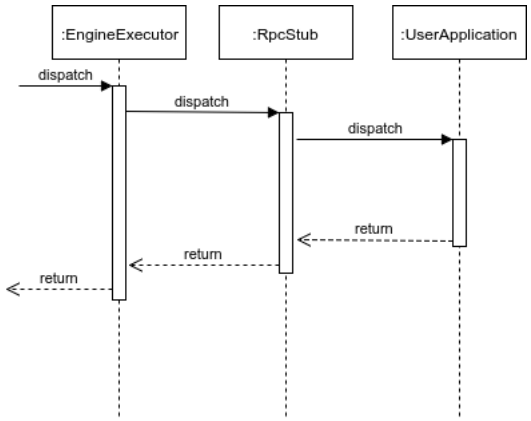

Figure 4: Simplified sequence of engine-executor executing online action on user's code.

4. Sample Engine

Marvin applications are also labeled as engines. An engine is composed by the application's source code, a file containing parameters for the application and the metadata file. We encourage users to implement decoupled engines that are easy to maintain and less bug prone. To induce that we propose the DASFE design pattern (Figure 5). This pattern is strongly based on the DASE pattern (Chan et al., 2013), however we added the feedback phase to it. This evolution intends to enable the engine to receive input from external applications or users, this kind of feature allows the model to be modified online or add hooks to start a new training pipeline.

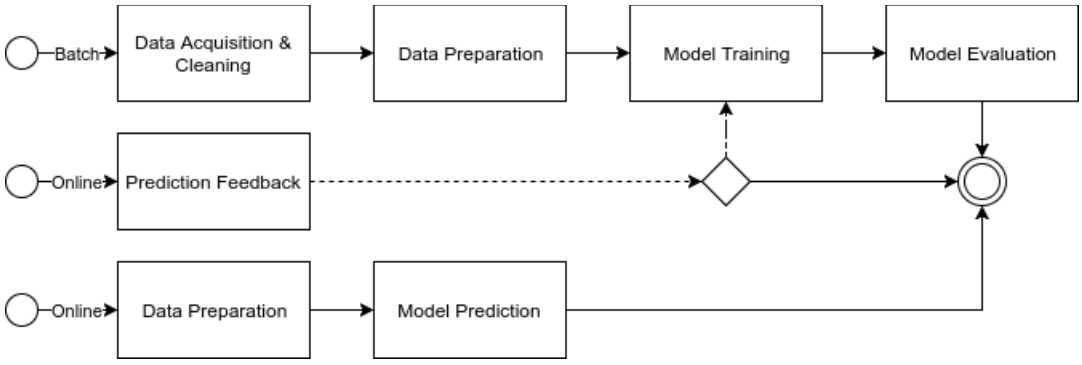

Figure 5: DASFE pattern pipeline.

We provide a project generator utility that generates the necessary base code for an engine, by doing that we reduced the complexity of the data scientist's job, requesting them to just populate these base files with the program logic. It is not necessary to care about passing the data to the next phase, or serializing the artifact in some persistence.

The next paragraphs will present a simplified sample engine as reference. The sample is a Python engine able to classify products using a linear classifier with stochastic gradient descent (SGD). The engine uses a mix of Spark framework data structures and Pandas dataframe.

The code to obtain data from data sources and remove unnecessary rows, i.e. data cleaning, should be placed in the execute method of AcquisitorAndCleaner class:

```
class AcquisitorAndCleaner(EngineBaseDataHandler):

    def execute(self, **kwargs):
        data = self.spark.sql("""select p.bscprd_desc as name,
        h.misphr_line as tag from core.mis_product_hierarchy as h,
        core.bsc_product as p
        where h.misphr_id_product = p.bscprd_id_product
        and h.misphr_line in ('SMARTPHONE', 'TABLETS')""")
        self.initial_dataset = data.toPandas()
```

Then it is necessary to prepare the acquired data before training. The execute method on TrainingPreparator should contain preparation logic, e.g. inputation of missing values and data type transformation:

```
class TrainingPreparator(EngineBaseDataHandler):

    def execute(self, **kwargs):
        data = self.initial_dataset
        vectorizer = TfidfVectorizer(encoding='utf-8')
        vectorizer.fit(data['name'])
        X_train = vectorizer.transform(data['name'][10:])
        y_train = data['tag'][10:]
        X_test = vectorizer.transform(data['name'][0:10])
        y_test = data['tag'][0:10]
        self.dataset = {
            "vectorizer": vectorizer,
            "X": (X_train, X_test),
            "y": (y_train, y_test)
        }
```

The model is finally trained at the Trainer class, when the execute method completes Marvin will serialize the model in the configured persistence. The data scientist do not need to implement the serialization logic, Marvin has serialization mechanisms implemented in all supported languages. If it is necessary to use custom serialization it can be achieved by extending Marvin application programming interface. The Trainer code will look like:

```
class Trainer(EngineBaseTraining):

    def execute(self, **kwargs):
        data = self.dataset
        clf = SGDClassifier(**self.params).fit(data['X'][0], data['y'][0])
        self.model = {"clf": clf, "vectorizer": self.dataset["vectorizer"]}
```

The MetricsEvaluator class is the appropriated class to place code related with model evaluation, in this example we're computing model error in a confusion matrix:

```
class MetricsEvaluator(EngineBaseTraining):

    def execute(self, **kwargs):
        pred = self.model['clf'].predict(self.dataset['X'][1])
        m1 = classification_report(self.dataset['y'][1], pred)
        m2 = confusion_matrix(self.dataset['y'][1], pred)
        self.metrics = [m1, m2]
```

At PredictionPreparator the sample engine is just performing data transformation:

```
class PredictionPreparator(EngineBasePrediction):

    def execute(self, input_message, **kwargs):
        return self.model['vectorizer'].transform([input_message['msg']])
```

Finally the Predictor class contains the logic that will be called for every valid HTTP request made to the Predictor server. Marvin's engine-executor will load the serialized object from the configured persistence and inject the trained model in the self.model variable. The code will be as follows:

```
class Predictor(EngineBasePrediction):

    def execute(self, input_message, **kwargs):
        return np.array_str(self.model["clf"].predict(input_message))
```

Deploying this code on Marvin's engine-executor will provide control over the training pipeline execution, dataframes and model serialization and versioning, metrics evaluation, model serving and feedback input interface.

5. Performance Assessment

Aiming to evaluate the parallelism ability of the predictor server, we conducted a set of experiments predicting classes of iris in a Support Vector Machine (SVM) (Hearst et al., 1998) model trained using the classical Fisher's iris dataset (Fisher, 1936) for classification. The overall objective of this experiment was to ensure that the predictor server is able to take advantage of multi-core architectures while not impacting significantly negative on the response time of predictions or the consistency of the results due to many queued threads or timeout exhaustion.

The test setup consisted of two dedicated machines, one simulating the users and other running Marvin's engine-executor with the SVM model that was previously trained. The client machine had 24 cores with Hyper-threading available and 48GB of RAM. The server machine also contained 24 cores with Hyper-threading and 64GB of RAM. Both machines were running Debian GNU/Linux.

The experiment strategy was to keep the amount of resources available and increase the throughput of concurrent requests being sent to the server until a significantly increase on the response time or failed requests was observed. The requests from client to server were made through the REST HTTP protocol and the machines were located in the same physical data center. As it is possible to see in Figures 6 and 7, we achieved 500 predictions

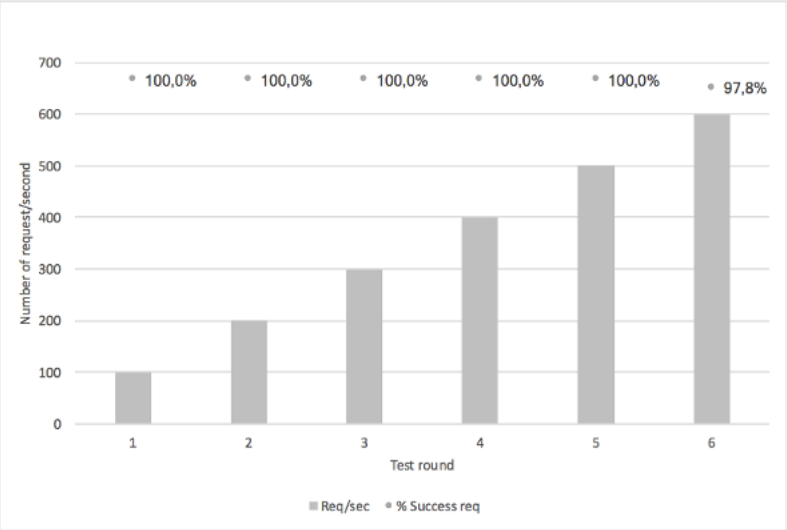

Figure 6: Predictor load test (reqs/second).

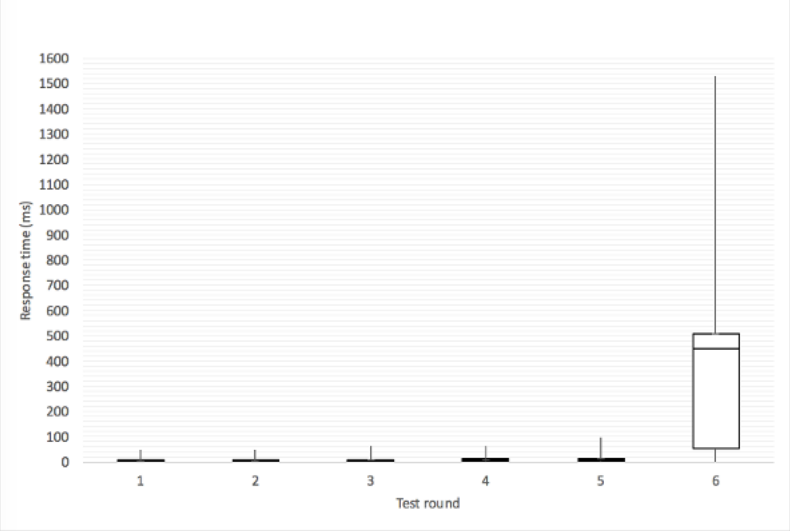

Figure 7: Predictor load test (Response time box plot).

per second maintaining a stable response time and none failed requests, at round 6 the mean response time increased significantly and the error rate has raised. The behavior of test round 6 indicates that we crossed the edge of efficient parallelism of Marvin's engine-executor predictions. Although we consider this a good number, we encourage administrators to deploy several instances of engine-executors behind a load-balancer when more than 500 predictions per second must be served. The current performance result is proved to be enough for large scale e-commerce platforms, like B2W Digital.

6. Future Work

Although Marvin is already being used in production setups, several improvements can be made to turn it into the de facto choose for data science teams which need to build production-facing models. The current version of Marvin has independent setups for each engine, it means that the user is responsible for having a layer on top of it if he desires to have a single management console of his engines. To build a cluster of engines the user needs to make specific configurations, like set the persistence folder for each engine under the same parent folder, and also maintain engine's parameters per instance. These could be challenging when it becomes to maintaining dozens of engines. Thus there is space to build a cluster admin on top of Marvin's engines.

Marvin platform was built on an architecture that allows the engine-executor to run engines implemented in different programming languages through the RPC protocol. As the date of this paper, there is a Python toolbox that facilitates the work of a user implementing engines in this language. In the near future we plan to focus our efforts on implementing toolboxes for different languages, e.g. R, Julia, Scala, Go and Java.

7. Conclusion

Implementing artificial intelligence applications with enterprise software characteristics is a hard task. Several contributions were made in libraries offering algorithms implementation and frameworks for distributed computing of data-intensive applications. Marvin adds tools and integrate with libraries and data frameworks to support the exploration and model development of such kind of applications, it introduces a framework that speeds up the task of turning model prototypes into industry-grade software. Lastly Marvin's engine-executor is a model server that takes care of pipeline execution, artifacts serialization and offers a standard interface to allow other applications to access the model, it takes into account non-functional requirements to allow safe concurrency and effective parallelism on shared and distributed memory. The experiments demonstrated that engine-executor is able to serve 500 predictions per second while maintaining stable response time and 0 failed requests.

Marvin engines are helping companies to be data-driven organizations, serving algorithms that can automate decisions such as optimizing its products prices to increase revenue, detecting fraud at the earliest stage and customizing sorting of many offers of the same product to customer clusters. The platform meshes the necessary components to empower data scientists pursuing to deliver production level applications that can support high throughput and process large datasets.

References

AzureMLTeam. Azureml: Anatomy of a machine learning service. In Louis Dorard, Mark D. Reid, and Francisco J. Martin, editors, *Proceedings of The 2nd International Conference on Predictive APIs and Apps*, volume 50 of *Proceedings of Machine Learning Research*, pages 1–13, Sydney, Australia, 06–07 Aug 2016. PMLR. URL `http://proceedings.mlr.press/v50/azureml15.html`.

Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

Philip K Chan and Salvatore J Stolfo. Toward scalable learning with non-uniform class and cost distributions: A case study in credit card fraud detection. In *KDD*, volume 1998, pages 164–168, 1998.

Simon Chan, Thomas Stone, Kit Pang Szeto, and Ka Hou Chan. Predictionio: a distributed machine learning server for practical software development. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 2493–2496. ACM, 2013.

Frank Chen, Zvi Drezner, Jennifer K Ryan, and David Simchi-Levi. Quantifying the bullwhip effect in a simple supply chain: The impact of forecasting, lead times, and information. *Management science*, 46(3):436–443, 2000.

Cloudera. Cloudera Data Science Workbench (DSW). `https://www.cloudera.com/products/data-science-and-engineering/data-science-workbench.html`. Accessed: 2017-10-10.

Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of human genetics*, 7(2):179–188, 1936.

Martin Fowler and James Lewis. Microservices. *ThoughtWorks. http://martinfowler.com/articles/microservices. html [last accessed on February 17, 2015]*, 2014.

Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 231–242. IEEE, 2011.

Marti A. Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. Support vector machines. *IEEE Intelligent Systems and their applications*, 13(4):18–28, 1998.

Carl Hewitt, Peter Bishop, and Richard Steiger. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference*, volume 3, page 235. Stanford Research Institute, 1973.

Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

Xiangrui Meng, Joseph Bradley, Evan Sparks, and Shivaram Venkataraman. Ml pipelines: a new high-level api for mllib. *Databricks blog, https://databricks.com/blog/2015/01/07/ml-pipelines-a-new-high-level-api-for-mllib. html*, 2015.

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

Evan R Sparks, Ameet Talwalkar, Virginia Smith, Jey Kottalam, Xinghao Pan, Joseph Gonzalez, Michael J Franklin, Michael I Jordan, and Tim Kraska. Mli: An api for distributed machine learning. In *Data Mining (ICDM), 2013 IEEE 13th International Conference on*, pages 1187–1192. IEEE, 2013.

Raj Srinivasan. Rpc: Remote procedure call protocol specification version 2. 1995.

Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. Optiml: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616, 2011.

Guido Van Rossum and Fred L Drake. *Python language reference manual*. Network Theory, 2003.

Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.