Machine Learning at Microsoft with ML.NET

Matteo Interlandi
Microsoft
Redmond, WA
mainterl@microsoft.com

Sergiy Matusevych
Microsoft
sergiym@microsoft.com

Misha Bilenko
Yandex
mbilenko@yandex-team.ru

Saeed Amizadeh
Microsoft
saamizad@microsoft.com

Shauheen Zahirazami
Microsoft
shzahira@microsoft.com

Markus Weimer
Microsoft
mweimer@microsoft.com

Abstract

Machine Learning is transitioning from an art and science into a technology available to every developer. In the near future, every application on every platform will incorporate trained models to encode data-driven decisions that would be impossible for developers to author. This presents a significant engineering challenge, since currently data science and modeling are largely decoupled from standard software development processes. This separation makes incorporating machine learning capabilities inside applications unnecessarily costly and difficult, and furthermore discourage developers from embracing ML in first place.

In this paper we introduce ML.NET, a framework developed at Microsoft over the last decade in response to the challenge of making it easy to ship machine learning models in large software applications. We present its main abstractions and design choices. Specifically, we introduce DataView, the core data abstraction of ML.NET which allows it to capture full predictive pipelines efficiently and consistently across training and inference lifecycles.

1 Introduction

We are witnessing an explosion of new frameworks for building Machine Learning (ML) models [16, 19, 7, 32, 14, 11, 21, 31, 10, 17, 1]. This profusion is motivated by the transition from machine learning as an art and science into a set of technologies readily available to every developer. An outcome of this transition is the abundance of applications that rely on trained models for functionalities that evade traditional programming due to their complex statistical nature. Speech recognition and image classification are only the most prominent such cases. This unfolding future, where most applications make use of at least one model, profoundly differs from the current practice in which data science and software engineering are performed in separate and different processes and sometimes even organizations. Furthermore, in current practice, models are routinely deployed and managed in completely distinct ways from other software artifacts. While typical software packages are seamlessly compiled and run on a myriad of heterogeneous devices, machine learning models are often relegated to be run as web services in relatively inefficient containers [6, 22, 2, 30, 12]. This pattern not only severely limits the kinds of applications one can build with machine learning capabilities, but also discourages developers from embracing ML as a core component of applications.

At Microsoft we have encountered this phenomenon across a wide spectrum of applications and devices, ranging from services and server software to mobile and desktop applications running on PCs, Servers, Data Centers, Phones, Game Consoles and IOT devices. A machine learning toolkit for such diverse use cases, frequently deeply embedded in applications, must satisfy additional constraints compared to the recent cohort of toolkits. For example, it has to limit library dependencies that are uncommon for applications; it must cope with datasets too large to fit in RAM; it has to be

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

portable across many target platforms; it has to be model class agnostic, as different ML problems lend themselves to different model classes; and, most importantly, it has to capture the full prediction pipeline that takes a test example from a given domain (e.g., an email with headers and body) and produces a prediction that can often be structured and domain-specific (e.g., a collection of likely short responses). The requirement to encapsulate predictive pipelines is of paramount importance because it allows for effectively decoupling application logic from model development. Carrying the complete train-time pipeline into production provides a dependable way for building efficient, reproducible, production-ready models [36].

The need for ML pipelines has been recognized previously. Python libraries such as Scikit-learn [31] provide the ability to author complex machine learning cascades. Python has become the most popular language for data science thanks to its simplicity, interactive nature (e.g., notebooks [8, 18]) and breadth of libraries (e.g., numpy [34], pandas [28], matplotlib [9]). However, Python-based libraries inherit many syntactic idiosyncrasies and language constraints (e.g., interpreted execution, dynamic typing, global interpreter locks that restrict parallelization), making them suboptimal for high-performance applications targeting a myriad of devices.

In this paper we introduce ML.NET: a machine learning framework allowing developers to author and deploy in their applications complex ML pipelines composed of data featurizers and state of the art machine learning models. Pipelines implemented and trained using ML.NET can be seamlessly surfaced for prediction without any modification: training and prediction, in fact, share the same code paths, and adding a model into an application is as easy as importing ML.NET runtime and binding the inputs/output data sources. ML.NET's ability to capture full, end-to-end pipelines has been demonstrated by the fact that 1,000s of Microsoft's data scientists and developers have been using ML.NET over the past decade, infusing 100s of products and services with machine learning models used by hundreds of millions of users worldwide.

ML.NET supports large scale machine learning thanks to an internal design borrowing ideas from relational database management systems and embodied in its main abstraction: DataView. DataView provides *compositional processing* of *schematized data* while being able to *gracefully and efficiently* handle *high dimensional* data in datasets *larger than main memory*. Like views in relational databases, a DataView is the result of computations over one or more base tables or views, and is generally immutable and lazily evaluated (unless forced to be materialized, e.g., when multiple passes over the data are requested). Under the hood, DataView provides streaming access to data so that working sets can exceed main memory. Next we will give an overview of ML.NET main concepts using a simple pipeline.

2 ML.NET: Overview

ML.NET is a .NET machine learning library that allows developers to build complex machine learning pipelines, evaluate them, and then utilize them directly for prediction. Pipelines are often composed of multiple transformation steps that featurize and transform the raw input data, followed by one or more ML models that can be stacked or form ensembles. We next illustrate how these tasks can be accomplished in ML.NET on a short but realistic example. Furthermore, we will exploit this example to introduce the main concepts in ML.NET.

```
1  var loader = new TextLoader().From<SentimentData>();
2  var featurizer= new TextFeaturizer("Features","Text")
3  var learner = new FastTreeBinaryClassifier()
4  {
5    // Some parameter
6  };
7
8  var pipeline = new LearningPipeline()
9      .Add(loader)
10     .Add(featurizer)
11     .Add(learner);
```

Figure 1: A text analysis pipeline whereby sentences are classified according to their sentiment.

Figure 1 introduces a Sentiment Analysis pipeline (SA). The first item required for building a pipeline is a *Loader* (line 1) which specifies the raw data input parameters and its schema. In the example pipeline, the input schema (`SentimentData`) is specified explicitly with a call to `From`, but in other situations (e.g., CSV files with headers) schemas can be automatically inferred by the loader. Loaders generate a DataView object, which is the core data abstraction of ML.NET. DataView provides a fully schematized non-materialized view of the data, and gets subsequently transformed by pipeline components. The second step is feature extraction from the input `Text` column (line 2). To achieve this, we use the `TextFeaturizer` *transform*. Transforms are the main ML.NET operators for manipulating data. Transforms accept a DataView as input and produce another DataView. `TextFeaturizer` is actually a complex transform built off a composition of nine base transforms that perform common tasks for feature extraction from natural text. Specifically, the input text is first normalized and tokenized. For each token, both char- and word-based ngrams are extracted and translated into vectors of numerical values. These vectors are subsequently normalized and concatenated to form the final `Features` column. Some of the above transforms (e.g., normalizer) are *trainable*: i.e., before producing an output DataView they are required to scan the whole dataset to determine internal parameters (e.g., scalers). Subsequently, in line 3 we apply a *learner* (i.e., a trainable model)— in this case, a binary classifier called FastTree: an efficient implementation of the MART gradient boosting algorithm [24]. Once the pipeline is assembled (line 8), we can train it by calling the homonym method on the pipeline object with the expected output prediction type (Figure 2). ML.NET evaluation is *lazy*: no computation is actually run until the train method (or other methods triggering pipeline execution) is called. This allows ML.NET to (1) properly validate that the pipeline is well-formed before computation; and (2) deliver state of the art performance by devising efficient execution plans.

```
var model = pipeline.Train<SentimentPrediction>();
```

Figure 2: Training of the sentiment analysis pipeline. Up to here no execution is actually triggered.

Once a pipeline is trained, a model object containing all training information is created. The model can be saved to a file (in this case, the information of all trained operators as well as the pipeline structure are serialized into a compressed file), or evaluated against a test dataset (Figure 3) or directly used for prediction serving (Figure 4). To evaluate model performance, ML.NET provides specific components called *evaluators*. Evaluators accept a previously trained model as input alongside test datasets and produce a set of metrics. In the specific case of the `BinaryClassifierEvaluator` used in Figure 3, relevant metrics are those used for binary classifiers, such as accuracy, Area Under the Curve (AUC), log-loss, etc.

```
    var evaluator = new BinaryClassifierEvaluator();
    var metrics = evaluator.Evaluate(model, testData);
```

Figure 3: Evaluating mode accuracy using a test dataset.

Finally, serving the model for prediction is achieved by calling the `Predict` method with a list of `SentimentData` objects. Predictions can be served natively in any OS (e.g., Linux, Windows, Android, macOS) or device (x86/x64 and ARM processors) supported by the .NET Core framework.

```
var predictions = model.Predict(PredictionData);
```

Figure 4: Serving predictions using the trained model.

3 Related Work

While several machine learning frameworks [29, 10, 17, 16, 19, 14, 7, 32, 11, 4] have been proposed in the past years, Scikit-learn [31] and H20 probably remains the most related to ML.NET. Scikit-learn has been developed as a machine learning tool for Python and, as such, it mainly targets interactive uses cases running over datasets fitting in main memory. Given its characteristic, Scikit-learn has several limitations when it comes to experimenting over Big Data: runtime performance are often inadequate; large datasets and feature sets are not supported; datasets cannot be streamed but instead they can only be accessed in batch from main memory. ML.NET solves the aforementioned problems thanks to the DataView abstraction (Section 4.1) and several other techniques inspired by databases.

While ML.NET uses DataView, H2O employs H2O Data Frames as abstraction of data. Differently than DataView however, H2O Data Frames are not immutable but “fluid”, i.e., columns can be added, updated and removed by modifying the base data frame. Fluid vectors are compressed so that larger than RAM working sets can be used. H2O provides several interfaces (R, Python, Scala) and large variety of algorithms. H2O is JVM-based and provides performance for large dataset mainly through in-memory distributed computation (based on Apache Spark [35, 15]). Conversely, ML.NET main focus is efficient single machine computation.

4 System Design Principles and Abstractions

ML.NET borrows ideas from the database community. ML.NET's main abstraction is called DataView (Section 4.1). Similarly to (intensional) database relations, the DataView abstraction provides compositional processing of schematized data, but specializes it for machine learning pipelines. The DataView abstraction is generic and supports both primitive operators as well as the composition of multiple operators to achieve higher-level semantics such as the `TextFeaturizer` transform of Figure 1 (Section 4.2). Under the hood, operators implementing the DataView interface are able to gracefully and efficiently handle high-dimensional and large datasets thanks to *cursoring* (Section 4.3) which resembles the well-known iterator model of databases [25].

4.1 The DataView Abstraction

In relational databases, the term *view* typically indicates the result of a query on one or more tables (base relations) or views, and is generally immutable. Views (and tables) are defined over a *schema* which expresses a sequence of *columns names* with related *types*. The semantics of the schema is such that each data row outputs of a view must conform to its schema. Views have interesting properties which differentiate them from tables and make them appropriate abstractions for machine learning: (1) views are *composable*—new views are formed by applying transformations (queries) over other views; (2) views are *virtual*, i.e., they can be lazily computed on demand from other views or tables without having to materialize any partial results; and (3) since a view does not contain values, but merely computes values from its source views, it is *immutable* and *deterministic*: the same exact computation applied over the same input data always produces the same result. Immutability and deterministic computation enable transparent data caching (for speeding up iterative computations such as ML algorithms) and safe parallel execution. DataView inherits the aforementioned database view properties, namely: schematization, composability, lazy evaluation, immutability, and deterministic execution.

Schema with Hidden Columns. Each DataView carries schema information specifying the name and type of each view's column. DataView schemas are ordered and, by design, multiple columns can share the same name, in which case, one of the columns *hides* the others: referencing a column by name always maps to the latest column with that name. Hidden columns exist because of immutability and can be used for debugging purposes: having all partial computations stored as hidden columns allows the inspection of the provenance of each data transformation. Indeed, hidden columns are never fully materialized in memory (unless explicitly required by the algorithm or user) therefore their resource cost is minimal.

High Dimensional Data Support with Vector Types. Machine learning and advanced analytics applications often involve high-dimensional data. For example, common techniques for learning from text uses bag-of-words (e.g., `TextFeaturizer`), one-hot encoding or hashing variations to represent non-numerical data. These techniques typically generate an enormous number of features (e.g., with hashing, it is common to use 20 bits or more, producing 2^{20} features or more). Representing each feature as an individual column is far from ideal, both from the perspective of how the user interacts with the information and how the information is managed in the schematized system. The DataView solution is to represent each set of features as a single *vector* column. This technique is similar to the *column families* concept in NoSQL systems such as BigTable [20] or Cassandra [5]. A vector type specifies an item type and optional dimensionality information. The item type must be a primitive, non-vector, type. The optional dimensionality information specifies the number of items in the corresponding vector values. When the size is unspecified, the vector type is variable-length. For example, the `TextTokenizer` transform (contained in `TextFeaturizer`) maps a text value to the sequence of individual terms in that text. This transformation naturally produces variable-length vectors of text. Conversely, fixed-size vector columns are used, for example, to represent a range of column from an input dataset.

4.2 Composing Computations using DataView

ML.NET includes several standard operators and the ability to compose them using the DataView abstraction to produce efficient machine learning pipelines. *Transform* is the main operator class: transforms are applied to a DataView to produce a derived DataView and are used to prepare data for training, testing, or prediction serving. *Learners* are machine learning algorithms that are trained on data (eventually coming from some transform) and produce predictive models. *Evaluators* take scored test datasets and produced metrics such as precision, recall, F1, AUC, etc. Finally, *Loaders* are used to represent data sources as a DataView, while *Savers* serialize DataViews to a form that can be read by a loader. We now details some of the above concepts.

Transforms. Transforms take a DataView as input and produce a DataView as output. Many transforms simply "add" one or more computed columns to their input schema. More precisely, their output schema includes all the columns of the input schema, plus some additional columns, whose values are computed starting from some of the input columns. It is common for an added column to have the same name as an input column, in which case, the added column hides the input column, as we have previously described. Multiple primitive transforms may be applied to achieve higher-level semantics: for example, the `TextFeaturizer` transform of Figure 1 is the composition of 9 primitive transforms.

Trainable Transforms. While many transforms simply map input data values to output by applying some pre-defined computation logic (e.g., `Concat`), other transforms require "training", i.e., their precise behavior is determined automatically from the input training data. For example, normalizers and dictionary-based mappers translating input values into numerical values (used in `TextFeaturizer`) build their state from training data. Given a pipeline, a call to `Train` triggers the execution of all trainable transforms (as well as learners) in topological order. When a transform (learner) is trained, it produces a DataView representing the computation up to that point in the pipeline: the DataView can then be used by downstream operators. Once trained and later saved, the state of a trained transform is serialized such that, once loaded back the transform is not retrained.

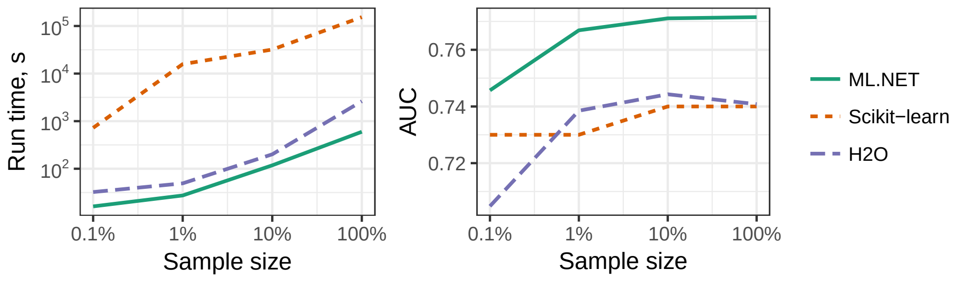
Learners. Alike trainable transforms, learners are machine learning algorithms that take DataView as input and produce "models": transforms that can be applied over input DataViews and output predictions. ML.NET supports learners for *binary classification*, *regression*, *multi-class classification*, *ranking*, *clustering*, *anomaly detection*, *recommendation* and *sequence prediction* tasks.

4.3 Cursoring over Data

ML.NET uses DataView as a representation of a computation over data. Access to the actual data is provided through the concept of *row cursor*. While in databases queries are compiled into a chain of operators, each of them implementing an iterator-based interface, in ML.NET, ML pipelines are compiled into chains of DataViews where data is accessed through cursoring. A row cursor is a movable window over a sequence of data rows coming either from the input dataset or from the result of the computation represented by another DataView. The row cursor provides the column values for the current row, and, as iterators, can only be advanced forward (no backtracking is allowed).

Columnar Computation. In data processing systems, it is common for a down-stream operator to only require a small subset of the information produced by the upstream pipeline. For example, databases have columnar storage layouts to avoid access to unnecessary columns [33]. This is even more so in machine learning pipelines where featurizers and ML models often work on one column at a time. For instance, `TextFeaturizer` needs to build a dictionary of all terms used in a text column, while it does not need to iterate over any other columns. ML.NET provides columnar-style computation model through the notion of *active columns* in row cursors. Active columns are set when a cursor is initialized: the cursor then enforces the contract that only the computation or data movement necessary to provide the values for the active columns are performed.

Pull-base Model, Streaming Data. ML.NET runtime performance are proportional to data movements and computations required to scan the data rows. As iterators in database, cursors are pull-based: after an initial setup phase (where for example active columns are specified) cursors do not access any data, unless explicitly asked to. This strategy allows ML.NET to perform at each time only the computation and data movements needed to materialize the requested rows (and column values within a row). For large data scenarios, this is of paramount importance because it allows efficient streaming of data directly from disk, without having to rely on the assumption that working sets fit into main



(a) Runtime (b) Accuracy

Figure 5: Experimental results for the Criteo dataset.

memory. Indeed, when the data is known to fit in memory, caching provides better performance for iterative computations.

5 System Implementation and Experiments

ML.NET is the solution Microsoft developed for the problem of empowering developers with a machine learning framework to author, test and deploy ML pipelines. ML.NET is implemented with the goal of providing a tool that is easy to use, scalable over large datasets while providing good performance, and able to unify under a single API data transformations, featurizers, and state of the art machine learning models. In its current implementation, ML.NET comprises 2773K lines of C# code, and about 74K lines of C++ code, the latter used mostly for high-performance linear algebra operations employing SIMD instructions. ML.NET supports more then 80 featurizers and 40 machine learning models. ML.NET is extensible [27, 13] and allows users to import pre-trained TensorFlow and ONNX [3] models.

We tested ML.NET over the Criteo dataset [23] and we report here a comparison against Scikit-learn and H2O. The reported experiments are carried out on a standard data science virtual machine on Azure with 56GB of RAM, 112 GB of Local SSD and a single Intel(R) Xeon(R) @ 2.40GHz processor. We used Scikit-learn version 0.19.1 and H2O version 3.20.0.7. For the three systems we build a pipeline which (1) fills in the missing values in the numerical columns of the dataset; (2) encodes a categorical columns into a numeric matrix using a hash function; and (3) applies a Gradient Boosting Classifier (LightGBM [26] for ML.NET). Figure 5a shows the total runtime (including training and testing), while Figure 5b depicts the AUC on the test dataset. As we can see, ML.NET has the best performance. H2O shows good runtime performance, especially for smaller datasets. In this experiment Scikit-learn has the worst running time: with the full training dataset, ML.NET trains in around 10 minutes while Scikit-learn takes more than 2 days. Regarding the accuracy, we can notice that the results from ML.NET dominate Scikit-learn/H2O by a large margin. This is mainly due to the superiority of LightGBM versus the Gradient Boosting algorithm used in the latters.

6 Conclusions

Machine learning is rapidly transitioning from a niche field to a core element of modern application development. This raises a number of challenges Microsoft faced early on. ML.NET addresses a core set of them: it brings machine learning onto the same technology stack as application development, delivers the scalability needed to work on datasets large and small across a myriad of devices and environments, and, most importantly, allows for complete pipelines to be authored and shared in an efficient manner. These attributes of ML.NET are not an accident: they have been developed in response to requests and insights from thousands of data scientists at Microsoft who used it to create hundreds of services and products used by hundreds of millions of people worldwide every day. ML.NET is open source and publicly available at `https://github.com/dotnet/machinelearning`.

Acknowledgments

We would like to thank Yiwen Zhu, Pete Luferenko, Tom Finley, Monte Hoover and the ML.NET team for suggestions and help with early drafts of the paper.

References

[1] H2O Algorithms Roadmap. `https://github.com/h2oai/h2o-3/blob/master/h2o-docs/src/product/flow/images/H2O-Algorithms-Road-Map.pdf`, 2015.

[2] TensorFlow serving. `https://www.tensorflow.org/serving`, 2016.

[3] Open Neural Network Exchange (ONNX). `https://onnx.ai`, 2017.

[4] Caffe2. `http://caffe2.ai/`, 2018.

[5] Cassandra. `http://cassandra.apache.org/`, 2018.

[6] Clipper. `http://clipper.ai/`, 2018.

[7] CNTK. `https://docs.microsoft.com/en-us/cognitive-toolkit/`, 2018.

[8] Jupyter. `http://jupyter.org/`, 2018.

[9] Matplotlib. `https://matplotlib.org/`, 2018.

[10] Michelangelo. `http://eng.uber.com/michelangelo/`, 2018.

[11] MXNet. `https://mxnet.apache.org/`, 2018.

[12] MXNet Model Server (MMS). `https://github.com/awslabs/mxnet-model-server`, 2018.

[13] NimbusML. `https://github.com/Microsoft/NimbusML`, 2018.

[14] PyTorch. `https://pytorch.org/`, 2018.

[15] Spark. `http://spark.apache.org/`, 2018.

[16] TensorFlow. `https://www.tensorflow.org`, 2018.

[17] TransmogrifAI. `https://transmogrif.ai/`, 2018.

[18] Zeppelin. `https://zeppelin.apache.org/`, 2018.

[19] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[20] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 205–218, 2006.

[21] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[22] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. 2017.

[23] Criteo. Kaggle display advertising challenge dataset, 2014.

[24] J. H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

[25] G. Graefe. Volcano: An extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, Feb. 1994.

[26] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3146–3154. Curran Associates, Inc., 2017.

[27] Y. Lee, A. Scolari, B. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi. PRETZEL: opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, pages 611–626, 2018.

[28] W. Mckinney. pandas: a foundational python library for data analysis and statistics. 01 2011.

[29] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. 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. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[30] C. Olston, F. Li, J. Harmsen, J. Soyke, K. Gorovoy, L. Lao, N. Fiedel, S. Ramesh, and V. Rajashekhar. Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML Systems at NIPS*, 2017.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, Nov. 2011.

[32] F. Seide and A. Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 2135–2135, New York, NY, USA, 2016. ACM.

[33] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[34] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011.

[35] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[36] M. Zinkevich. Rules of machine learning: Best practices for ML engineering. https://developers.google.com/machine-learning/rules-of-ml.