Machine Learning at Microsoft with ML.NET

Matteo Interlandi
Microsoft
Redmond, WA
mainterl@microsoft.com

Sergiy Matusevych
Microsoft
Redmond, WA
sergiym@microsoft.com

Saeed Amizadeh
Microsoft
Redmond, WA
saamizad@microsoft.com

Shauheen Zahirazami
Microsoft
Redmond, WA
shzahira@microsoft.com

Markus Weimer
Microsoft
Redmond, WA
mweimer@microsoft.com

Abstract

We are witnessing an explosion of new frameworks for building Machine Learning (ML) models [14, 17, 7, 24, 13, 11, 18, 23, 10, 15, 4]. 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 ways that are very different from those of other software artifacts. While typical software packages are seamlessly compiled and ran on a myriad of heterogeneous devices, machine learning models are often relegated as services in relatively inefficient containers [6, 19, 5, 22, 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 other available 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 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 entire end-to-end 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 [26].

The need for ML pipelines has been recognized previously. Python libraries such as Scikit-learn [23] 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, 16]) and breadth of libraries (e.g., numpy [25], pandas [21], matplotlib [9]). However, Python-based libraries inherit many syntactic idiosyncrasies and language constraints (e.g. interpreted execution, dynamic typing, global interpreter lock that restrict parallelization, etc.), making them suboptimal for high-performance applications targeting wide range of devices.

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

In this paper we introduce ML.NET: a recently open-sourced [2] 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 in 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 thousands 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. ML.NET is open source and publicly available [2]; a recent demo showcasing ML.NET capabilities can be found at [1], while we refer readers to [3] for example pipelines.

Next we will give an overview of ML.NET main concepts using a simple pipeline.

ML.NET: An Overview

ML.NET is a .NET machine learning library that allows developers to build complex machine learning pipelines, evaluate them, and seamlessly deploy them 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. Below we illustrate how these tasks can be accomplished in ML.NET on a short but realistic example. We will also exploit this example to introduce the main concepts of ML.NET.

```
1 var loader = new TextLoader().From<SentimentData>();
2 var featurizer = new TextFeaturizer("Features", "Text");
3 var learner = new FastTreeBinaryClassifier() { /* Some parameters */ };
4
5 var pipeline = new LearningPipeline()
6     .Add(loader)
7     .Add(featurizer)
8     .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. Loader produces 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 column `Text` (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 create a *learner* (i.e. a trainable model)— in this case, a binary classifier called `FastTree`: an efficient implementation of the MART gradient boosting algorithm [20]. 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 plan.

```
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 used directly for prediction serving (Figure 4). To evaluate model performance, ML.NET provides specific components called *evaluators*. An evaluator accepts a previously trained model as input alongside test datasets and produces 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.

System Requirements and Implementation

ML.NET is the solution Microsoft developed to empower developers with a machine learning framework to author, test, and deploy ML pipelines. ML.NET is implemented with the following goals in mind:

1. *Unification*: ML.NET must act as a unifying framework that can host a variety of models and components (with related idiosyncrasies). Once ML pipelines are trained, the same pipeline must be deployable into any production environment (from data centers to IoT devices) with close to zero engineering cost. In the last decade 100s of products and services have employed ML.NET, validating its success as a unifying platform.
2. *Extensibility*: Data scientists are interested in experimenting with different models and features with the goal of obtaining the best accuracy. Therefore, it should be possible to add new components and algorithms with minimal reasonable effort via a general API that supports a variety of data types and formats. Since its inception, ML.NET has been extended with many components. In fact, a large fraction of the built-in operators started life as extensions shared between data scientists.
3. *Scalability and Performance*: ML.NET must be scalable and allow maximum hardware utilization—i.e., be fast and provide high throughput. Because production-grade datasets are often very large and do not fit in RAM, scalability implies the ability to run pipelines in out-of-memory mode, with data paged in and processed incrementally. In our internal benchmarks ML.NET achieves impressive scalability and performance (up to several orders-of-magnitude) when compared to other publicly available toolkits.

In its current implementation, ML.NET comprises of 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 provides more then 80 featurizers and 40 machine learning models.

References

[1] ML.NET Demo. https://www.youtube.com/watch?v=zXn10vy8F6E.

[2] ML.NET Github Repository. https://github.com/dotnet/machinelearning.

[3] ML.NET Samples. https://github.com/dotnet/machinelearning-samples.

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

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

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

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

[15] TransmogrifAI. https://transmogrif.ai/, 2018.

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

[17] 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.

[18] 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.

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

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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