Buzzsaw: A System for High Speed Feature Engineering

Andrew Stanton and Liangjie Hong and Manju Rajashekhar
Etsy Inc.
117 Adams Street
Brooklyn, NY
{astanton,lhong,mrajashekhar}@etsy.com

ABSTRACT

Feature Engineering is a critical part of machine learning systems. Most feature engineering solutions focus on a single language or computational abstraction for their pipelines, forcing practitioners to either embrace a restricted ecosystem or incur operational and orchestrational complexity. In this paper, we describe *Buzzsaw*: an independent system for constructing and evaluating complex feature pipelines, designed for embedded usage across multiple languages and platforms. We describe its core interfaces, data types, and algorithms for implementing the system and compare it to three other common feature engineering frameworks.

1 INTRODUCTION

The widespread application of machine learning techniques in recent years has rapidly expanded the capabilities of both consumer and business products. With the commoditization of ETL platforms and machine learning suites, much of the work of a data scientist is spent on feature engineering, plugging and playing different sets of features to improve models [7]. It is often considered the "secret sauce" of model improvement [4, 5]. However, as data size and model complexity increases, a divide has appeared between where feature engineering is performed and where learning algorithms are actually implemented.

As industry has consolidated around platforms like Spark [17] and MapReduce [6] for cluster compute, large investments have been made in tooling for broadening platform capabilities for handling diverse workloads. It is natural, then, that feature engineering gravitates toward those ecosystems; after all, raw features are readily available, usually as outputs of upstream jobs. Furthermore, cluster compute is exceptionally well suited for large scale processing of feature transforms: most transformations can be implemented with the main primitive of the paradigm, a Map. However, there are real tradeoffs with these platform-specific feature pipelines, especially in early stage experimentation:

(1) High speed implementations are typically developed independently of cluster compute libraries [1, 8–10, 13].
(2) Cluster-compute pipelines are hard to iterate on due to "Pipelines as Code". Iteration leads to pipeline jungles [15], hindering maintenance.
(3) It requires touching multiple systems and platforms even for simple experimentation.
(4) It is hard, or impossible, to bundle feature pipelines with external models.
(5) Debugging is difficult: features are engineered in a completely separate environment, creating a large surface area for failure.

In this paper, we present *Buzzsaw*: a standalone, high performance feature engineering library, designed with the following considerations in mind:

- Embeddable. It should work directly within other languages.
- Extensible. Adding new feature transforms should be easy and safe.
- Scalable. It needs to work effectively in both local and cluster compute environments.
- Configurable. It needs to be config driven, not code driven.
- Performance. It needs to be fast for both batch and real-time uses.
- Maintainance. It should focus on common interfaces to minimize glue code [15].
- Deterministic. It should produce consistent results at all stages of execution.

2 BUZZSAW

Buzzsaw operates on structured features, evaluating each through a user defined directed acyclic graph (DAG). Nodes are defined as Feature Processors (FP): interfaces which compute a transform on a set of inputs and produce an output. Edges describe the dependencies between outputs of upstream feature processors to downstream consumers. Structured data is submitted to the graph as JSON objects, where keys describe the feature names and values the actual features. *Buzzsaw* is written in Rust, facilitating safety and integration over foreign function interfaces (FFI).

2.1 High Level Architecture

Buzzsaw operates on JSON formatted data. On execution, it begins by packing the raw features into the *Buzzsaw*'s internal data representation, a DType. The processor graphs are then evaluated, each node consuming either the original input data or the output of a previously evaluated Feature Processor. After evaluation, an Output Formatter aggregates the necessary fields, transforming the associated DTypes into the format specified by a user provided *Buzzsaw* config.

There are two independent DAGs specified in *Buzzsaw*: Features and Supervised. When generating training data, both DAGs are evaluated, allowing the Output Formatter to produce supervised features. However, when only features are required, such as during prediction, only the features graph is executed. Figure 1 examines a simple DAG.

2.2 Buzzsaw DAG

The core contribution of *Buzzsaw* is in the composition and evaluation of the Feature Processor DAG. We start by describing the

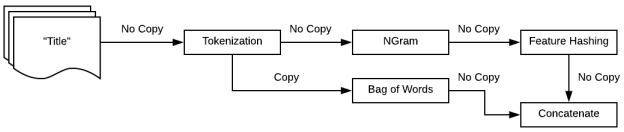

Figure 1: Example of a feature DAG

Framework	Platform	DPS	X Slower
Buzzsaw	Native	**1,274,044**	1.00x
Scikit-Learn	Python	66,948	19.03x
PySpark	Python	16,624	76.64x
PySpark*	Python	17,406	73.19x
PS+B Single	Library	107,996	11.80x
PS+B Single*	Library	152,531	8.35x
PS+B Batch	Library	201,299	6.33x
PS+B Batch*	Library	441,656	2.88x
KeystoneML	Scala	232,805	5.47x

Table 1: Experiment: Feature Hashing + TF-IDF

primitive data types and move on to defining the minimal interfaces needed for constructing and evaluating the DAG. We close out by discussing Output Formatters. Input documents are composed of namespaced, strongly typed data, similar to the DataFrame representation used in Spark ML [2]. Feature Processors are executed according to their topological sort order, caching resulting outputs for consumption by downstream processors. *Buzzsaw* passes full ownership of the input channels into an FP, allowing it reuse data structures and utilize other performance tricks, only copying data when needed (see Figure 1).

Buzzsaw internally represents features in an algebraic data type (ADT) called DType. Currently, *Buzzsaw* supports the following: booleans, 64-bit signed integers, 64-bit floats, string, set of strings, sparse vectors, dense vectors, and string to float mappings.

Feature Processors (FP) are the backbone of *Buzzsaw*. An implementation provides a single function, `evaluate`, which takes in a vector of DType and returns a vector of DType. They operate over "columns" of data, providing computational opportunities, such as vectorization over adjacent rows (e.g. L2 normalization of dense vectors). Feature Processors have a companion "SerDe" (an abbreviation of Serialize/Deserialize) object. It's responsible for learning a FPs settings from a data sample as well as constructing its companion FP from saved configurations. All Feature Processors SerDe's (FPS) tie versions to their underlying representations. When loading a saved FP, SerDe's validate the saved configuration is compatible with the current code base.

Output Formatters produce common machine learning formats from the DAGs, producing both supervised and feature-only records. As of writing, *Buzzsaw* produces libsvm [3] format, Vowpal Wabbit [10] format, and a generic TSV format.

2.3 Configuration

Users provide a config describing how features flow through the DAG. Users describe feature data types, the pipeline, and how to map features to a specified output format. *Buzzsaw* follows two phases: a learning phase, called `fit`, and a `generate` phase which produces the actual feature vectors.

During `fit`, *Buzzsaw* parses the user configuration and a sample of the dataset to construct the underlying Feature Processors, learning any statistics relevant to the FP (e.g. mean and variance in the case of Z-Whitening). It produces a new configuration file, the *Buzzsaw* pipeline config (BPC) file, consisting of the serialized pipelines.

The `generate` phase loads the pipelines from a BPC file and processes documents in mini-batches provided by the user. *Buzzsaw* is guaranteed thread-safe by use of language primitives, simplifying parallel computation in languages like Python and Java.

3 EVALUATION

To evaluate the efficacy of our approach, we compare *Buzzsaw* to other popular frameworks on feature pipeline throughput using three variants: native, a standalone executable wrapping the *Buzzsaw* library, embedded into Python, and finally compare it to the state-of-the-art pipeline, KeystoneML[16].

For Python, we look at two common frameworks for feature engineering: Scikit-learn[14] and PySpark [12, 17], using the ML Pipeline's interface built and maintained by Databricks [11]. We compare the three frameworks using a sample of the May 2015 Reddit comment dataset, released on Kaggle [1], resulting in a dataset of 29, 590, 949 samples. For hardware, we use a dual CPU, Intel Xeon CPU E5-2630 v2 at 2.60GHz machine with 128GB of RAM and 4.5TB of disk space, running CentOS 7.

We measure performance for a multi-channel feature pipeline; we convert the 'body' field into a feature vector via TF-IDF, the 'author' field into a feature vector via feature hashing, and finally concatenate the resulting vectors to create the output. A more complete evaluation is available in the full paper.

3.1 Framework Comparisons

We measure overall framework throughput with Documents per Second (DPS). We measure the relative throughput between libraries as compared to our Buzzsaw native implementation, labelled as "X Slower". In an attempt to better replicate performance across a cluster, we measure and remove fixed cost overhead from PySpark. These are marked in 1 with an *. Experiments were run three times, using the fastest time reported.

Buzzsaw was the fastest framework tested, over an order of a magnitude faster than other Python frameworks. PySpark + Buzzsaw showed good scaling, finishing over 23 times faster than vanilla PySpark. Compared to the KeystoneML framework, *Buzzsaw* registered nearly a 5.5X improvement in throughput over the equivalent pipeline.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1383–1394.

[1] https://www.kaggle.com/reddit/reddit-comments-may-2015

[3] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)* 2, 3 (2011), 27.

[4] Olivier Chapelle and Yi Chang. 2011. Yahoo! learning to rank challenge overview. In *Proceedings of the Learning to Rank Challenge*. 1–24.

[5] Ildefons Magrans de Abril and Masashi Sugiyama. 2013. Winning the kaggle algorithmic trading challenge with the composition of many models and feature engineering. *IEICE transactions on information and systems* 96, 3 (2013), 742–745.

[6] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[7] Jeffrey Dunn. 2016. Introducing FBLearner Flow: Facebook's AI backbone. *Facebook blog, https://code.facebook.com/posts/1072626246134461/introducing-fblearner-flow-facebook-s-ai-backbone. html* (2016).

[8] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2016. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759* (2016).

[9] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*. 3149–3157.

[10] John Langford, Lihong Li, and Alex Strehl. 2007. Vowpal wabbit online learning project. (2007).

[11] Xiangrui Meng, Joseph Bradley, Evan Sparks, and Shivaram Venkataraman. 2015. 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).

[12] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

[13] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. 2017. PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration. (2017).

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

[15] D Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems*. 2503–2511.

[16] Evan R Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J Franklin, and Benjamin Recht. 2017. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*. IEEE, 535–546.

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