A Demonstration of Willump: A Statistically-Aware End-to-end Optimizer for Machine Learning Inference

Peter Kraft, Daniel Kang, Deepak Narayanan,
Shoumik Palkar, Peter Bailis, Matei Zaharia
Stanford DAWN Project
{kraftp, ddkang, deepakn, shoumik, pbailis, matei}@cs.stanford.edu

ABSTRACT

Systems for ML inference are widely deployed today, but they typically optimize ML inference workloads using techniques designed for conventional data serving workloads and miss critical opportunities to leverage the statistical nature of ML. In this demo, we present Willump, an optimizer for ML inference that introduces statistically-motivated optimizations targeting ML applications whose performance bottleneck is feature computation. Willump automatically cascades feature computation for classification queries: Willump classifies most data inputs using only high-value, low-cost features selected by a cost model, improving query performance by up to 5× without statistically significant accuracy loss. In this demo, we use interactive and easily-downloadable Jupyter notebooks to show VLDB attendees which applications Willump can speed up, how to use Willump, and how Willump produces such large performance gains.

PVLDB Reference Format:
Peter Kraft, Daniel Kang, Deepak Narayanan, Shoumik Palkar, Peter Bailis, Matei Zaharia. A Demonstration of Willump: A Statistically-Aware End-to-end Optimizer for Machine Learning Inference. *PVLDB*, 13(12): 2833-2836, 2020.
DOI: https://doi.org/10.14778/3415478.3415487

1. INTRODUCTION

The importance of machine learning in modern data centers has sparked interest in model serving systems, which perform ML inference and serve predictions to users [2]. However, these model serving systems typically approach ML inference as an extension of conventional data serving workloads, missing critical opportunities to exploit its statistical nature. Most modern model serving systems, such as Clipper [2], Amazon Sagemaker, and Microsoft AzureML, treat ML inference as a black box and implement generic systems optimizations such as caching and adaptive batching. Some systems, such as Pretzel [5], also apply traditional compiler optimizations such as loop fusion.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415487

These optimizations are useful for ML inference applications, just as they are for web applications or database queries. However, unlike other serving workloads, ML inference workloads have unique statistical properties that these optimizations do not leverage. One such property is that *ML models can often be approximated efficiently on many inputs*. For example, the computer vision community has long used "model cascades" where a low-cost model classifies "easy" inputs and a higher-cost model classifies inputs where the first is uncertain, resulting in faster inference with negligible change in accuracy [7].

In recent work, we have developed a system to leverage this opportunity for optimization: Willump, a statistically-aware end-to-end optimizer for ML inference [4]. Willump targets a common class of ML inference applications: those whose performance bottleneck is feature computation. In these applications, a pipeline of transformations converts raw input data into numerical features that are then used by an ML model to make predictions. These applications are common, especially when performing ML inference over tabular data. For example, a recent study of ML inference at Microsoft found that feature computation accounted for over 99% of the runtime of some production ML inference applications [5]. Willump improves ML inference performance through an automatic approximation algorithm we call end-to-end cascades.

The key observation underlying end-to-end cascades is that ML inference pipelines often compute many features for use in a model, but can classify some data inputs using only a subset of these features. For example, a pipeline that detects toxic online comments may need to compute expensive TF-IDF vectorizations to classify some comments, but can classify others simply by checking for curse words.

Selectively computing features is challenging because features vary by orders of magnitude in computational cost and importance to the model and are often computationally dependent on one another. Therefore, one cannot pick an arbitrary set of features (e.g,. the least computationally intensive) and expect to efficiently classify data inputs with them.

To address these challenges, Willump uses a cost model based on empirical observations of ML model performance to identify important but inexpensive features. With these features, Willump trains an approximate model that can identify and classify "easy" data inputs, but *cascade* "hard" inputs to a more powerful model. For example, an approximate model for toxic comment classification might classify comments with curse words as toxic but cascade others. Willump automatically tunes cascade parameters to maximize query performance while meeting an accuracy target. The concept of cascades has a long history in the ML literature, beginning with [7],

but to the best of our knowledge, Willump is the first system to automatically generate feature-aware and model-agnostic cascades from input programs. Willump's cascades deliver speedups of up to 5× on real-world ML inference pipelines without a statistically significant effect on accuracy [4].

Willump complements end-to-end cascades with powerful compiler optimizations. Willump compiles a subset of Python to machine code through the Weld system [6], in the process applying optimizations such as loop fusion and vectorization. Compilation improves query throughput by up to 4× and query latency by up to 400× [4].

In this demo paper, we explain use cases for Willump in more detail (Section 2), discuss Willump's design and optimizations (Section 3), and sketch how we plan to demonstrate Willump (Section 4), so VLDB attendees can visualize Willump transforming and optimizing real-world ML inference pipelines for dramatic performance increases.

2. BACKGROUND

Willump optimizes ML inference applications whose performance is bottlenecked by feature computation. In such applications, ML inference is performed by a pipeline of transformations which receives raw input from clients, transforms it into numerical features (such as by computing statistics about a raw string input), and executes an ML model on the features to generate predictions. In this paper we define *features* as numerical inputs to an ML model.

It is relatively common for ML inference applications to be bottlenecked by feature computation, especially when using less expensive ML models such as linear classifiers and boosted trees. For example, a recent study of ML inference at Microsoft found feature computation accounted for over 99% of the runtime of some production ML inference applications [5]. Feature computation often dominates performance because it encompasses many common but relatively expensive operations in machine learning, such as querying remote data stores [1].

Recent developments in automated machine learning (AutoML) on tabular data have increased the importance of feature computation. Researchers have developed algorithms such as Google AutoML Tables and Deep Feature Synthesis [3] to automatically generate ML inference pipelines dependent on powerful but computationally expensive features. Willump optimizes the performance of these pipelines [4].

We diagram an ML inference pipeline in Figure 1. This pipeline, which we call `Toxic`, is a simplified version of one of our real-world benchmark pipelines [4]. It predicts whether an online comment is toxic. `Toxic` transforms an input string into numerical features with two TF-IDF vectorizers: one word-level and one character-level. `Toxic` then executes a logistic regression model on these features to predict whether the input was toxic. In the real pipeline `Toxic` is based on, feature computation accounts for over 99% of runtime.

3. WILLUMP OVERVIEW

3.1 Architecture

Willump is an optimizer for ML inference pipelines. Willump users write ML inference pipelines in Python as functions from raw inputs to model predictions. Specifically, these functions must register model training, prediction, and scoring functions, must be written as a series of explicit Python function calls, and must represent data using NumPy arrays, SciPy sparse matrices, or Pandas DataFrames.

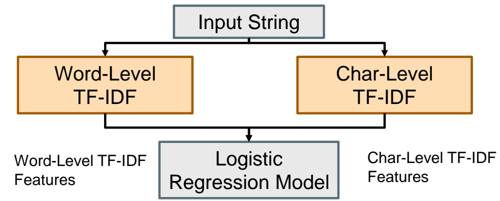

Figure 1: A simplified toxic comment classification pipeline. The pipeline computes word- and character-level TF-IDF features from a string and predicts from them with a logistic regression model.

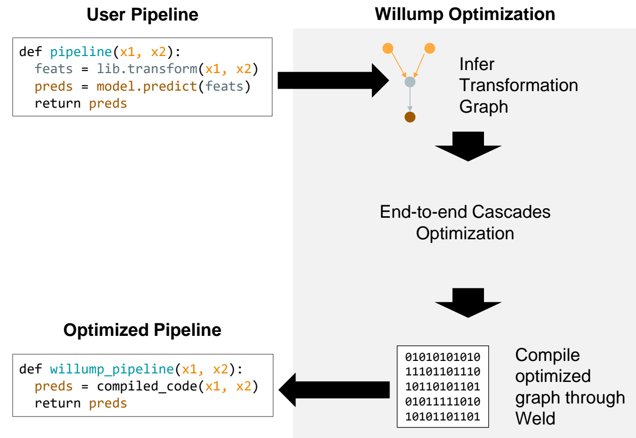

Figure 2: A diagram of Willump's architecture. Willump infers a transformation graph from a user pipeline, optimizes it, compiles it through Weld, and returns an optimized pipeline.

Willump operates in three stages: graph construction, optimization and compilation. First, Willump's graph construction stage converts an ML inference pipeline into a graph of transformations, such as Figure 1. Then, Willump's optimization stage applies the end-to-end cascades optimization to the transformation graph. Finally, Willump's compilation stage transforms the optimized graph back into a Python function. In the process, it compiles some graph nodes to optimized machine code using Weld [6]. We diagram this architecture in Figure 2. This demo paper will focus on the optimization stage and end-to-end cascades.

3.2 Automatic End-to-End Cascades

End-to-end cascades speed up ML inference pipelines that perform classification by classifying some data inputs with an *approximate model* dependent on a subset of the original model's features. When using cascades, Willump first attempts to classify each data input with the approximate model. Willump returns the approximate model's prediction if its confidence exceeds a threshold, which we call the *cascade threshold*, but otherwise computes all remaining features and classifies with the original model. This is shown in Figure 3.

Willump automatically constructs end-to-end cascades from an ML inference pipeline, its training data, and an accuracy target. First, Willump partitions features into computationally independent groups and computes their computational cost and importance to the model. Then, Willump identifies

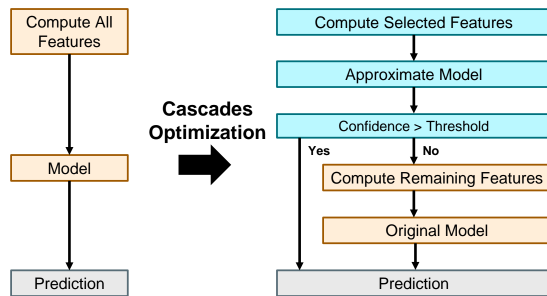

Figure 3: Willump's cascades optimization. Willump attempts to predict data inputs using the approximate model, but cascades to the original model if the approximate model is not confident.

several sets of computationally inexpensive but predictively powerful features. For each selected set of features, Willump trains an approximate model, chooses a cascade threshold based on the accuracy target, and uses these to estimate the cost of accurately making predictions using cascades with those features. Willump constructs cascades using the selected set of features that minimizes this cost. We discuss this algorithm in more detail in the original Willump paper [4].

4. DEMONSTRATION

At VLDB 2020, we will present a demo showcasing how Willump's optimizations improve the performance of real-world ML inference pipelines. To facilitate virtual attendance and maximize Willump's accessibility, we have created a simplified version of Willump specifically for VLDB, implementing Willump's cascades optimization in < 500 lines of code with no external dependencies and a user-friendly interface. It is available on GitHub here: `https://github.com/stanford-futuredata/Willump-Simple`. We have implemented our demo in Jupyter notebooks using this simplified Willump implementation so attendees can access it from the comfort of their homes. In the notebooks, attendees will interactively step through the stages of Willump's cascades optimization, learning how choices of features or parameters affect cascades behavior. After the demo, attendees will understand the performance benefits Willump offers and how Willump produces them.

Attendees first select an ML inference application to demonstrate. We provide Jupyter notebook demos corresponding to several of the applications benchmarked in the original Willump paper. These applications, curated from high-accuracy entries in major data science competitions, cover a diverse range of problem domains from music recommendation to content moderation.

In each notebook, attendees interactively step through the process of Willump optimizing an ML inference pipeline and experiment with Willump's optimizations. The notebooks begin with the Python code for the target ML application and Willump's optimization interface, demonstrating how Willump is used in practice. In Figure 4, we show the code for the `Music` benchmark, which performs music recommendation by querying precomputed features from a remote data store. We expect attendees will first run this code unmodified, but they can freely modify it (for example, by removing features) to observe the effects on performance.

```
@willump_execute(predict_function=music_predict,
                 confidence_function=music_confidence,
                 predict_cascades_params=cascades_dict)
def music_eval_pipeline(data, model):
    user_latent_features = get_features_from_redis(data, name="features_uf")
    song_latent_features = get_features_from_redis(data, name="features_sf")
    user_cluster_features = get_features_from_redis(data, name="uc_features")
    song_cluster_features = get_features_from_redis(data, name="sc_features")
    artist_cluster_features = get_features_from_redis(data, name="ac_features")
    user_features = get_features_from_redis(data, name="us_features")
    song_features = get_features_from_redis(data, name="ss_features")
    artist_features = get_features_from_redis(data, name="as_features")
    genre_features = get_features_from_redis(data, name="gs_features")
    city_features = get_features_from_redis(data, name="cs_features")
    ages_features = get_features_from_redis(data, name="ages_features")
    language_features = get_features_from_redis(data, name="ls_features")
    gender_features = get_features_from_redis(data, name="gender_features")
    composer_features = get_features_from_redis(data, name="composer_features")
    lyrs_features = get_features_from_redis(data, name="lyrs_features")
    sns_features = get_features_from_redis(data, name="sns_features")
    stabs_features = get_features_from_redis(data, name="stabs_features")
    stypes_features = get_features_from_redis(data, name="stypes_features")
    regs_features = get_features_from_redis(data, name="regs_features")
    return music_predict(model,
                         [user_latent_features, song_latent_features,
                          user_cluster_features, song_cluster_features,
                          artist_cluster_features, user_features, song_features,
                          artist_features, genre_features, city_features, ages_features,
                          language_features, gender_features, composer_features,
                          lyrs_features, sns_features, stabs_features, stypes_features,
                          regs_features])
```

Figure 4: The Python code for `Music`, performing music recommendation by querying pre-computed features from a remote data store.

```
Feature: user_latent_features        Cost:  1.000 Importance:  0.102
Feature: user_cluster_features       Cost:  1.000 Importance:  0.040
Feature: song_cluster_features       Cost:  1.000 Importance:  0.039
Feature: sns_features                Cost:  1.000 Importance:  0.033
Feature: song_latent_features        Cost:  1.000 Importance:  0.023
Feature: stabs_features              Cost:  1.000 Importance:  0.016
Feature: user_features               Cost:  1.000 Importance:  0.015
Feature: artist_cluster_features     Cost:  1.000 Importance:  0.011
Feature: stypes_features             Cost:  1.000 Importance:  0.008
Feature: regs_features               Cost:  1.000 Importance:  0.007
Feature: city_features               Cost:  1.000 Importance:  0.006
Feature: genre_features              Cost:  1.000 Importance:  0.005
Feature: ages_features               Cost:  1.000 Importance:  0.004
Feature: language_features           Cost:  1.000 Importance:  0.002
Feature: artist_features             Cost:  1.000 Importance:  0.002
Feature: gender_features             Cost:  1.000 Importance:  0.001
Feature: composer_features           Cost:  1.000 Importance:  0.000
Feature: song_features               Cost:  1.000 Importance: -0.001
Feature: lyrs_features               Cost:  1.000 Importance: -0.001
```

Figure 5: `Music` feature statistics. Willump measures the computational cost and permutation importance of all features.

```
Total feature cost: 19.000  Selected feature cost cutoff: 4.750
Selected features:
        user_latent_features
        user_cluster_features
        song_cluster_features
        sns_features
Confidence threshold: 0.600  Percentage of inputs approximated: 0.835
Expected Query Cost = Percent Approximated * Selected Feature Cost +
                      Percent Not Approximated * Total Feature Cost
                    = 0.835 * 4.000 + 0.165 * 19.000 = 6.480
Projected speedup: 2.932
```

Figure 6: Feature selection and cost estimation in `Music`. Willump selects high-value, low-cost features, trains an approximate model from them, empirically determines what percentage of inputs can be accurately approximated, and estimates the resulting speedup from approximation.

In the next part of the demo, attendees watch Willump optimize the chosen ML inference pipeline, learning about Willump's optimizations in the process. First, Willump computes the computational cost and importance to the model of all the pipeline's features and reports them to the user. For example, in Figure 5, Willump determines that all features in `Music` have the same cost (because they are precomputed and need only be queried from a remote database), but importances vary greatly.

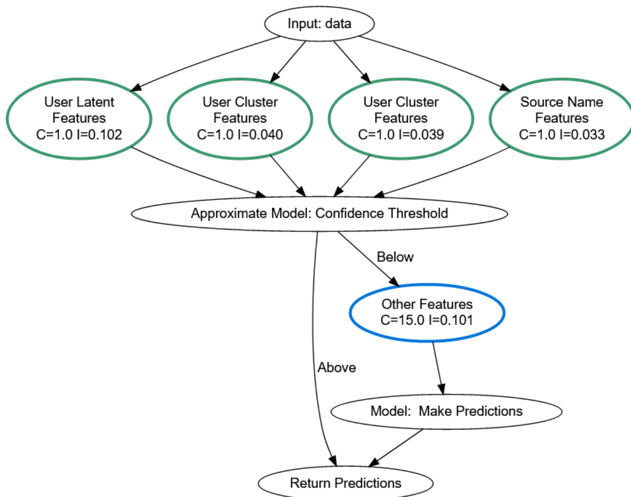

Figure 7: A graph of the optimized `Music`. Willump attempts to predict each data input with an approximate model trained on selected features, but only returns this result if its confidence is above a threshold, otherwise computing all features and predicting with the original model.

Next, Willump selects features for its approximate model. Attendees enter a feature cost cutoff into the notebook. Willump then selects the features from which it can train the most accurate approximate model given that cost cutoff, estimating accuracy as the sum of the feature importance scores of the selected features. For example, in Figure 6, we set a cost cutoff of one quarter the total feature cost, which is enough to select four features. Since in this example all features have the same cost, Willump chooses the four features with the highest importance scores: the user latent features, user and song cluster features, and source name features.

After selecting features, Willump trains an approximate model and uses a validation set to empirically estimate the percentage of data inputs that it can accurately classify. It then uses a cost model to estimate the projected speedup provided by cascades constructed from that model. It displays these computations to the user, as shown in Figure 6. Attendees can experiment with different cost cutoffs to maximize projected speedup. By doing this, they take the role of Willump's optimizer, which automatically experiments with several sets of features before choosing the best-performing one. To help attendees visualize how cascades use selected features, we graph each cascaded pipeline, as in Figure 7

After attendees have chosen an approximate model, Willump uses cascades to efficiently make predictions. It predicts the application's test set using both the original and cascaded pipelines. It reports accuracy and graphs performance, showing attendees the dramatic speedups that cascades produce without statistically significant accuracy loss. For example, in Figure 8, Willump improves `Music` performance by 2.8×.

5. TAKEAWAYS

Our demo will showcase the power of Willump to improve the performance of real-world ML inference applications. Attendees will come away understanding what applications Willump can speed up, how to use Willump, and how Willump's op-

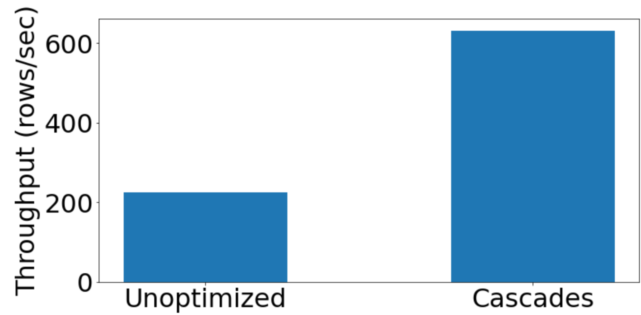

Figure 8: The performance improvement Willump produces in `Music`.

timizations work. Feature computation is the performance bottleneck in many widely used ML inference applications, but it is not yet well-studied, and state-of-the-art systems still use generic approaches such as operator compilation [5]. We hope that by demonstrating the large performance improvements that Willump produces with ML-specific optimizations such as end-to-end cascades, we can spark new interest in this area.

6. REFERENCES

[1] D. Agarwal, B. Long, J. Traupman, D. Xin, and L. Zhang. Laser: A Scalable Response Prediction Platform for Online Advertising. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, pages 173–182. ACM, 2014.

[2] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[3] J. M. Kanter and K. Veeramachaneni. Deep Feature Synthesis: Towards Automating Data Science Endeavors. In *2015 IEEE International Conference on Data Science and Advanced Analytics, DSAA 2015, Paris, France, October 19-21, 2015*, pages 1–10. IEEE, 2015.

[4] P. Kraft, D. Kang, D. Narayanan, S. Palkar, P. Bailis, and M. Zaharia. Willump: A statistically-aware end-to-end optimizer for machine learning inference. In *Proc. Conference on Systems and Machine Learning, MLSys 2020*, 2020.

[5] Y. Lee, A. Scolari, B. Chun, M. 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 18)*, pages 611–626, 2018.

[6] S. Palkar, J. Thomas, D. Narayanan, P. Thaker, R. Palamuttam, P. Negi, A. Shanbhag, M. Schwarzkopf, H. Pirk, S. Amarasinghe, et al. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. volume 11, pages 1002–1015. VLDB Endowment, 2018.

[7] P. Viola and M. Jones. Rapid Object Detection Using a Boosted Cascade of Simple Features. page 511. IEEE, 2001.