# WILLUMP: A STATISTICALLY-AWARE END-TO-END OPTIMIZER FOR MACHINE LEARNING INFERENCE

**Peter Kraft** [1] **Daniel Kang** [1] **Deepak Narayanan** [1] **Shoumik Palkar** [1] **Peter Bailis** [1] **Matei Zaharia** [1]

## 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 paper, we present WILLUMP, an optimizer for ML inference that introduces two statistically-motivated optimizations targeting ML applications whose performance bottleneck is feature computation. First, WILLUMP automatically cascades feature computation for classification queries: WILLUMP classifies most data inputs using only high-value, low-cost features selected through empirical observations of ML model performance, improving query performance by up to 5× without statistically significant accuracy loss. Second, WILLUMP accurately approximates ML top-K queries, discarding low-scoring inputs with an automatically constructed approximate model and then ranking the remainder with a more powerful model, improving query performance by up to 10× with minimal accuracy loss. WILLUMP automatically tunes these optimizations' parameters to maximize query performance while meeting an accuracy target. Moreover, WILLUMP complements these statistical optimizations with compiler optimizations to automatically generate fast inference code for ML applications. We show that WILLUMP improves the end-to-end performance of real-world ML inference pipelines curated from major data science competitions by up to 16× without statistically significant loss of accuracy.

## 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 (Crankshaw et al., 2017; Wang et al., 2018). However, these model serving systems typically approach ML inference as an extension of conventional data serving workloads, missing critical opportunities to exploit the statistical nature of ML inference. Most modern model serving systems, such as Clipper (Crankshaw et al., 2017), 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 (Lee et al., 2018), also apply traditional compiler optimizations such as loop fusion.

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. Two of these properties are:

- **ML models can often be approximated efficiently for 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 much faster inference with negligible change in accuracy (Viola & Jones, 2001; Wang et al., 2017). In contrast, existing multi-purpose model serving systems utilize the same logic for all data inputs.
- **ML models are often used for higher-level queries**, such as top-K queries. However, existing model serving systems do not optimize these query modalities. As we show, tailoring inference to the query (in our work, top-K queries) can improve performance.

To leverage these opportunities for optimization, we present WILLUMP, a system for automatically performing end-to-end optimization of ML inference workloads. 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 (Lee et al., 2018). WILLUMP improves ML

[1]Department of Computer Science, Stanford University. Correspondence to: Peter Kraft <kraftp@cs.stanford.edu>.

*Proceedings of the $3^{rd}$ MLSys Conference*, Austin, TX, USA, 2020.
Copyright 2020 by the author(s).

inference performance through two novel optimizations:

**1) Automatic End-to-end Cascades:** ML inference pipelines often compute many features for use in a model. In classification problems, because ML applications are amenable to approximation, it is often possible to classify data inputs using only a subset of these features. For example, in a pipeline that detects toxic online comments, we may need to compute expensive TF-IDF features to classify some comments, but can classify others simply by checking for curse words.

However, 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 other comments. 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 Viola & Jones (2001), 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.

**2) Top-K Query Approximation:** WILLUMP automatically optimizes an important class of higher-level application queries: top-K queries. Top-K queries request a ranking of the K top-scoring elements of an input dataset. They are fundamentally asymmetric: predictions for high-scoring data inputs must be more precise than predictions for low-scoring data inputs. Existing model serving systems such as Clipper or Pretzel execute top-K queries naively, scoring every element of the input dataset and thus wasting time generating precise predictions for low-scoring data inputs. WILLUMP instead leverages top-K query asymmetry by automatically constructing a computationally simple approximate pipeline to filter out low-scoring inputs, maximizing performance while meeting a target accuracy level. Approximation improves performance on real-world serving workloads by up to 10×, with negligible impact on accuracy.

WILLUMP complements end-to-end cascades and top-K query approximation with compiler optimizations. WILLUMP compiles a subset of Python to machine code using Weld (Palkar et al., 2017; 2018), in the process applying optimizations such as loop fusion and vectorization. These optimizations improve query throughput by up to 4× and median query latency by up to 400×.

We evaluate WILLUMP on a broad range of pipelines curated from entries to major data science competitions hosted by Kaggle, CIKM, and WSDM. Overall, WILLUMP improves query throughput by up to 16× and median query latency by up to 500×. WILLUMP's novel optimizations contribute greatly to this performance: end-to-end cascades improve performance by up to 5× and top-K query approximation by up to 10×. WILLUMP also improves the performance of other model serving systems; integrating WILLUMP with Clipper improves end-to-end query latencies by up to 10×. All performance improvements come without statistically significant accuracy loss.

In summary, we make the following contributions:

- We introduce WILLUMP, a statistically-aware end-to-end optimizer for ML inference pipelines.
- We describe a method for automatically cascading feature computation, improving ML inference performance by up to 5× without statistically significant accuracy loss.
- We describe a method for automatically approximating top-K queries, improving performance by up to 10× with minimal accuracy loss.

## 2 BACKGROUND

In this section, we provide background on ML inference pipelines, cascades, and top-K queries.

### 2.1 ML Inference Pipelines

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 (Lee et al., 2018). Feature computation often dominates performance because it encompasses many common but relatively expensive operations in machine learning, such

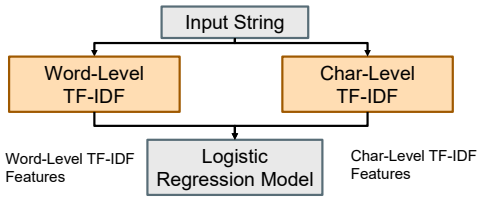


*Figure 1.* A simplified toxic comment classification pipeline. The pipeline computes word- and character-level TF-IDF features from a string and predicts from them with a logistic regression model.

as querying remote data stores (Agarwal et al., 2014).

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 (Lu, 2019) and Deep Feature Synthesis (Kanter & Veeramachaneni, 2015) (whose open-source implementation is widely used (Kanter, 2018)) to automatically generate ML inference pipelines dependent on powerful but computationally expensive features. WILLUMP optimizes the performance of these pipelines.

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, taken from Kaggle (Tunguz, 2018). 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.

### 2.2 Cascades

Cascades are an approximation technique for ML inference first developed for computer vision (Viola & Jones, 2001). ML inference is amenable to approximation because ML models return probabilistic predictions instead of exact answers. However, using an approximate model on every input incurs a high accuracy cost. Cascades reduce this accuracy cost for classification problems by leveraging differing data input difficulty.

In most classification workloads, many data inputs are "easy" to classify in the sense that a computationally simple model can accurately classify them (Viola & Jones, 2001). Therefore, a system need not approximate every data input. Instead, it can accurately classify easy data inputs with a computationally simple approximate model and *cascade* to a more expensive model for hard data inputs. Existing cascades systems such as NoScope (Kang et al., 2017) and Focus (Hsieh et al., 2018) have shown that cascades can dramatically improve ML inference performance with minimal accuracy cost, but they are specialized to one model type.

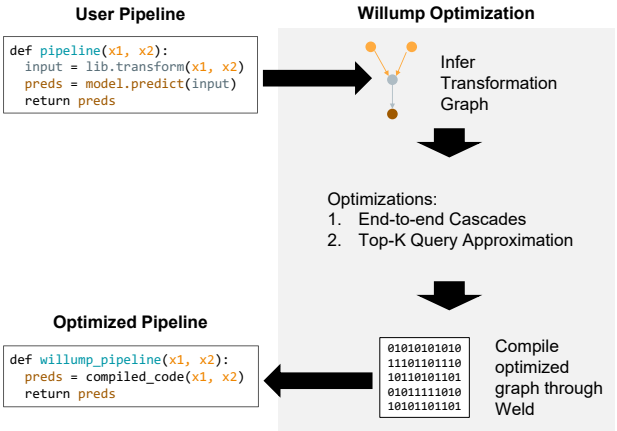


*Figure 2.* A diagram of WILLUMP's architecture. WILLUMP infers a transformation graph from a user pipeline, optimizes it, compiles it through Weld, and returns an optimized pipeline.

Unlike existing cascades systems, WILLUMP automatically optimizes an entire ML inference pipeline, training an approximate model which depends on only a subset of the original model's features. For example, an approximate model for `toxic` might compute only word-level (and not character-level) TF-IDF features. We call WILLUMP's cascades optimization *end-to-end* cascades. We discuss it in more detail in Section 4.

### 2.3 Top-K Queries

Top-K queries are an important class of ML inference query. They request a ranking of the K top-scoring elements of a dataset. Top-K queries are especially common in recommender systems (Cheng et al., 2016). Database researchers have proposed several algorithms for approximating top-K queries (Theobald et al., 2004). However, these algorithms require scoring functions to be monotonic (Ilyas et al., 2008); this is rarely true for ML models. Some ML recommender systems use fast retrieval models to approximate top-K queries (Cheng et al., 2016), but they develop these models manually. Because ML top-K optimization is not automatic, existing ML model serving systems such as Clipper or Pretzel do not optimize top-K queries, instead naively scoring all elements of the input dataset. WILLUMP automatically approximates top-K queries, using an approximate model dependent on a subset of the original model's features to identify and discard low-scoring inputs before ranking remaining inputs with the original model. We describe our approximation algorithm in detail in Section 4.

## 3 WILLUMP OVERVIEW

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.

WILLUMP operates in three stages: graph construction, optimization, and compilation. We diagram WILLUMP's workflow in Figure 2.

**Graph Construction Stage:** WILLUMP's graph construction stage converts an ML inference pipeline into a graph of transformations. Figure 1 is an example transformation graph. We discuss transformation graph construction in Section 5.1.

**Optimization Stage:** WILLUMP's optimization stage applies its end-to-end cascades and top-K query approximation optimizations to the transformation graph. We discuss these optimizations in Section 4.

**Compilation Stage:** WILLUMP's compilation stage transforms the optimized graph back into a Python function that calls the optimized pipeline. In the process, it compiles some graph nodes to optimized machine code using Weld (Palkar et al., 2017). We discuss graph compilation in Section 5.2.

## 4 OPTIMIZATIONS

In this section, we describe WILLUMP's core optimizations: end-to-end cascades and top-K query approximation.

### 4.1 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 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 sketch this procedure

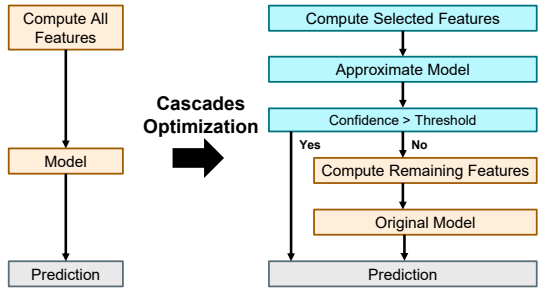


*Figure 3.* WILLUMP's cascades optimization. WILLUMP attempts to predict data inputs using the approximate model, but cascades to the original model if the approximate model is not confident.

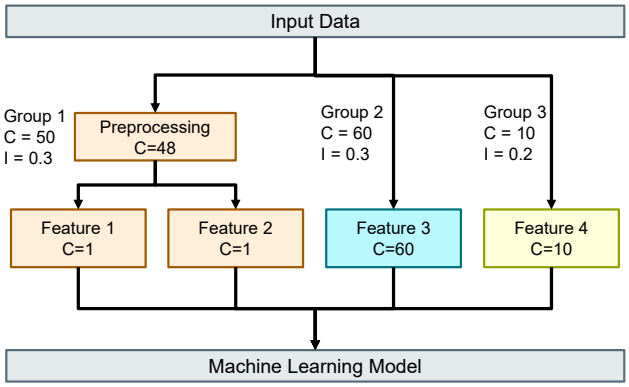


*Figure 4.* An example cascades optimization pipeline. Nodes are assigned to feature groups and marked with their cost (C); groups are also marked with their importance (I). WILLUMP might train an approximate model on Feature Groups 1 and 3 as they maximize sum of feature importance given maximum cost $c_{max} = 60$.

in Algorithm 1 and discuss it in the remainder of this section.

#### 4.1.1 *Partitioning Features*

The first step of constructing cascades is partitioning features into computationally independent sets, which we call *feature groups*. This is necessary because in most ML inference pipelines, many features are generated by the same upstream operators. Such features are not computationally independent: it is inefficient to compute some of them without also computing the rest. For example, in Figure 4, Features 1 and 2 both depend on the expensive Preprocessing node, so it is inexpensive to compute one after computing the other.

To partition features into groups, WILLUMP first identifies the transformation graph node that computed each feature. It then traces those nodes' ancestors to identify their dependencies. For example, in Figure 4, Feature 1 depends on itself and on the Preprocessing node. WILLUMP assigns two features to the same feature group if the cost of their shared dependencies exceeds the cost of either feature's unshared dependencies. For example, in Figure 4, it assigns Features 1

**Algorithm 1** Training an end-to-end cascade.

```
procedure TRAINCASCADE(Graph G, Data D, Accuracy Target a_t)
    T,H ← train_holdout_split(D)
    F ← identify_feature_groups(G)                    §4.1.1
    for f ∈ F do                                      §4.1.2
        imp(f) = permutation_importance(f,T,H)
        cost(f) = computational_cost(f,G,D)
    min_cost = ∞
    for c_max ∈ C do
            ▷ C = { 0.1 · cost(F), 0.2 · cost(F)... }
        S = select_feature_groups(F,c_max)            §4.1.3
        m = train_approximate_model(S,T)              §4.1.4
        t,h_S = cascade_threshold(m,H,a_t)            §4.1.5
            ▷ t is the cascade threshold.
            ▷ h_S is the fraction of H that m can classify.
        cost = h_S·cost(S) + (1−h_S)·cost(F)          §4.1.6
        if cost < min_cost then
            best_S, best_m, best_t = S, m, t
            min_cost = cost
    return best_S, best_m, best_t
```

and 2 to Feature Group 1 because the cost of their shared dependency, the Preprocessing node (C = 48), exceeds the cost of either feature's unshared dependencies (C = 1 for both).

#### 4.1.2 Computing Feature Group Statistics

The second step of constructing cascades is to calculate two statistics for each feature group: its permutation importance (Breiman, 2001; Molnar, 2019) and its computational cost. WILLUMP uses these statistics to select features for the approximate model. The permutation importance of a feature group is a model-agnostic measure of its value to the model's predictions (Fisher et al., 2018). The computational cost of a feature group is defined empirically as the amount of time it takes to compute the feature group on a sample of the training set.

#### 4.1.3 Query Cost Minimization

After identifying feature groups and computing their statistics, WILLUMP selects a set $S$ of computationally inexpensive but predictively powerful feature groups from which to train an approximate model. WILLUMP selects the features that minimize the expected prediction time $p_t$ of an inference query given the accuracy target $a_t$. This time is:

$$p_t = h_S \cdot \text{cost}(S) + (1 - h_S) \cdot \text{cost}(F) \quad (1)$$

Here, $F$ is the set of all features and $h_S$ is the percentage of data inputs that a cascade constructed from $S$ would classify with the approximate model given the accuracy target $a_t$. $h_S$ is computed using the cascade threshold, which we set later.

Unfortunately, $p_t$ is difficult to optimize directly. At a high level, WILLUMP approximates optimizing it by selecting several locally optimal sets of feature groups, measuring the performance of cascades constructed from each, and choosing the best of them.

To select a locally optimal set of feature groups, WILLUMP considers a candidate maximum feature cost $c_{max}$. For each candidate $c_{max}$, WILLUMP selects the set $S$ of feature groups that minimizes $p_t$ given $\text{cost}(S) = c_{max}$. This set cannot score better than the set that maximizes $h_S$ given $\text{cost}(S) \leq c_{max}$. That set, in turn, is equivalent to the set from which one could train an approximate model with maximum accuracy given $\text{cost}(S) \leq c_{max}$.

Unfortunately, determining approximate model accuracy for all combinations of feature groups is impractical. Instead, WILLUMP estimates approximate model accuracy as the sum of the permutation importance scores of the features on which the model was trained. Therefore, for each candidate $c_{max}$, WILLUMP selects the set $S$ of feature groups with maximum sum of permutation importance given $\text{cost}(S) \leq c_{max}$. This is a knapsack problem; we solve it with dynamic programming.

For each selected set of feature groups, WILLUMP constructs cascades and measures $p_t$. This requires training an approximate model and choosing a cascade threshold for each set.

#### 4.1.4 Training Approximate Models

WILLUMP trains an approximate model from a selected set of feature groups by computing the appropriate features from the training set and training a model of the same class as the original on them.

#### 4.1.5 Choosing Cascade Thresholds

To choose the cascade threshold for an approximate model, WILLUMP classifies every element in a held-out portion of the training set using both the approximate and original models, noting the confidences of the approximate model's predictions. For example $i$, we call the approximate model's prediction $s_i$, the original model's prediction $f_i$, and the approximate model's confidence $c_i$. The cascade threshold $t$ is the lowest value such that if we predict every data input $i$ with $s_i$ if $c_i > t$ and $f_i$ otherwise, accuracy on the held-out set would be above the accuracy target $a_t$.

#### 4.1.6 Selecting Optimal Features

Using the approximate models and cascade thresholds, WILLUMP computes $p_t$ (from Equation 1) for each selected set of feature groups. WILLUMP constructs cascades from the selected set of feature groups, and corresponding approximate model and cascade threshold, that minimizes $p_t$.

### 4.2 Top-K Query Optimization

In top-K queries, users request a ranking of the K top-scoring elements in a dataset. Top-K queries are fundamentally asymmetric: they must score high-scoring elements with more precision than low-scoring elements. WILLUMP leverages this asymmetry to approximate top-K queries by filtering out low-scoring candidates with an approximate model, then ranking the remainder with the original model. WILLUMP automatically constructs these approximate models during model training using an algorithm similar to that in Section 4.1. It requires four user-specified parameters: a distribution $\mathcal{K}$ of typical values of $K$, a distribution $\mathcal{N}$ of typical values of the dataset size $N$, an accuracy metric (by default precision), and and a lower accuracy bound $a_t$.

To evaluate a top-K query, WILLUMP scores all dataset elements with an approximate model trained on a selected set $S$ of feature groups, then ranks the $r_S K$ top-scoring elements with the original model. WILLUMP chooses the set $S$ and parameter $r_S$ that minimize the expected execution time $p_t$ of a top-K query given the accuracy bound $a_t$. This cost is:

$$p_t = \text{cost}(S) \cdot \overline{\mathcal{N}} + (\text{cost}(F) - \text{cost}(S)) \cdot r_S \overline{\mathcal{K}} \qquad (2)$$

Here, $F$ is the set of all features, and $\overline{\mathcal{N}}$ and $\overline{\mathcal{K}}$ are the means of $\mathcal{N}$ and $\mathcal{K}$.

To select a set $S$ of feature groups, WILLUMP considers several candidate maximum feature costs $c_{max}$. Like in Section 4.1, for each $c_{max}$, WILLUMP selects the candidate set $S$ of feature groups with maximum sum of permutation importance given $\text{cost}(S) \leq c_{max}$.

For each candidate set $S$ of feature groups, WILLUMP chooses a value of $r_S$, which determines the number of inputs to be ranked by the original model. WILLUMP chooses the smallest value of $r_S$ that satisfies the accuracy bound $a_t$. WILLUMP determines this value by executing sample top-K queries on a held-out portion of the training set. WILLUMP first trains an approximate model and scores all elements of the held-out set. It then draws many values $K$ and $N$ from $\mathcal{K}$ and $\mathcal{N}$. For each pair $(K, N)$, it draws a sample of size $N$ from the held-out set and measures the accuracies of approximate top-K queries run on that sample with different values of $r_S$. WILLUMP chooses the smallest value of $r_S$ for which accuracy is 95% certain to be greater than $a_t$ 95% of the time (these thresholds can be changed by the user). Specifically, it chooses the smallest $r_S$ such that if we consider each sample's outcome to be a Bernoulli random variable (where accuracy on the sample is either $\geq$ or $< a_t$), the 95% binomial proportion confidence interval of this variable is entirely above 95%.

Using this procedure, WILLUMP can compute $r_S$ and, therefore, the expected query execution time (Equation 2) for any candidate set $S$ of feature groups. WILLUMP selects the candidate set $S$ and corresponding $r_S$ value that minimize expected top-K query execution time given $a_t$.

## 5 WILLUMP API AND COMPILATION

In this section we describe WILLUMP's API and compilation procedure.

### 5.1 WILLUMP API and Graph Construction

WILLUMP can automatically optimize an ML inference pipeline implemented as a Python function which follows three rules. First, the user must register model training, inference, and scoring functions that conform to a simple API; for classification pipelines, the inference function must return a confidence metric. This rule guarantees WILLUMP's optimizations can be agnostic to model APIs. Second, every statement in the pipeline must be an explicit Python function call; the final statement, whose output is returned, must be to the model prediction function. This rule guarantees WILLUMP can trace data flow. Third, the pipeline must represent all data as NumPy arrays, SciPy sparse matrices, or Pandas DataFrames. This rule guarantees WILLUMP can marshal data to and from Weld. We show a simple example pipeline function using `scikit-learn` below:

```
def toxic_comment_classification(strings):
  wf = word_vectorizer.transform(strings)
  cf = char_vectorizer.transform(strings)
  return predict(model, wf, cf)
```

If an ML inference pipeline conforms to the above rules, WILLUMP can use standard compiler techniques to parse it into a directed acyclic graph of transformations. Each function called by the pipeline becomes a node in the transformation graph. While parsing, WILLUMP checks these functions against a list of known functions which it can compile to Weld. If the function is in the list, WILLUMP marks it compilable, otherwise WILLUMP leaves it in Python.

### 5.2 Transformation Graph Compilation

After optimizing a transformation graph, WILLUMP compiles the nodes marked compilable to optimized machine code using Weld (leaving the remaining nodes in Python) and packages them back into a Python function. Weld (Palkar et al., 2018) is an intermediate representation (IR) and compiler for data processing operations. Weld implements many compiler optimizations over its IR, including loop fusion, data structure preallocation, and vectorization. These are similar to the compiler optimizations implemented by some existing model serving systems such as Pretzel (Lee et al., 2018). The set of operators WILLUMP can compile to Weld is extensible; users can add Weld IR for custom operators. WILLUMP compiles graphs in four stages:

| Benchmark | Description | Feature-Computing Operators | Prediction Type | Model |
|---|---|---|---|---|
| Toxic (Tunguz, 2018) | Predict whether online comment is toxic. | String processing, N-grams, TF-IDF | Classification | Linear |
| Music (rn5l, 2018) | Predict whether user will like song. | Remote data lookup, data joins | Classification | GBDT |
| Product (Nguyen et al., 2017) | Classify quality of online store listing. | String processing, N-grams, TF-IDF | Classification | Linear |
| Instant (Kikani, 2019) | Approximate a function over tabular data. | Model Stacking | Classification | Ensemble |
| Purchase (Koehrsen, 2019) | Predict customer's next purchase. | Automatically Generated Features | Classification | GBDT |
| Price (Lopuhin, 2018) | Predict price of online good. | Feature encoding, string processing, TF-IDF | Regression | NN |
| Credit (Aguiar, 2018) | Predict probability a loan is defaulted. | Remote data lookup, data joins | Regression | GBDT |

*Table 1.* Properties of WILLUMP's benchmark workloads.

**Sorting:** In the first stage of compilation, WILLUMP sorts the transformation graph into an ordered list of nodes which minimizes the number of transitions between compilable nodes (which are executed in Weld) and Python nodes. This is desirable because each transition requires marshaling data between languages and because the Weld optimizer can apply end-to-end optimizations like loop fusion over large Weld blocks. WILLUMP sorts the graph topologically, then heuristically minimizes the number of transitions by moving each Python node to the earliest allowable location. We observe that this is effective in our evaluation because Python nodes are typically either preprocessing nodes (which execute before any Weld nodes) or parts of the model (which execute after all Weld nodes).

**Code Generation:** In the second stage of compilation, WILLUMP compiles every compilable node to the Weld IR. WILLUMP compiles nodes using parameterized Weld templates. For example, it compiles a TF-IDF vectorization node with a template containing a TF-IDF implementation written in Weld and parameterized to use several different tokenizers, n-gram ranges, and norms. After generating Weld code, WILLUMP coalesces Weld and uncompiled Python code segments together, creating *blocks* of Weld and Python code.

**Drivers:** In the third stage of compilation, WILLUMP generates a *driver*, a Python extension that calls into Weld, for each block of Weld code. WILLUMP identifies each block's input and output variables, then generates C++ driver code that marshals each block's input variables into Weld data structures, executes its Weld code, and marshals its output variables into Python data structures. To minimize driver latency, we developed several new Weld types for WILLUMP, including dataframe and sparse matrix types, which drivers can create in $O(1)$ time from their Python equivalents.

**Compilation:** In the final stage of compilation, WILLUMP combines the Python blocks and Weld drivers into a Python program. WILLUMP compiles drivers into Python C extensions, inserts calls to them in the appropriate places, packages the resulting all-Python program as a Python function, and returns it.

## 6 EVALUATION

We evaluate WILLUMP and its optimizations on seven high-performing entries to major data science competitions at CIKM, Kaggle, and WSDM. We demonstrate that:

1. WILLUMP improves batch inference throughput by up to 16× and point query latency by up to 500×. Of this speedup, up to 5× comes from end-to-end cascades, and the rest from compilation.
2. WILLUMP improves top-K query performance by up to 31×. Of this speedup, up to 10× comes from top-K query approximation, and the rest from compilation.
3. WILLUMP improves the performance of other model serving systems. WILLUMP improves end-to-end Clipper performance by up to 9×. Moreover, WILLUMP's optimizations improve performance over compiler optimizations similar to Pretzel's[1].

### 6.1 Experimental Setup

We ran all benchmarks on an n1-standard-8 Google Cloud instance with four Intel Xeon CPUs running at 2.20 GHz with 30 GB of RAM. We stored remote data tables on a Redis 3.2.6 server on a GCP instance with a single Intel Xeon CPU running at 2.20 GHz with 30 GB of RAM.

### 6.2 Benchmarks

We benchmark WILLUMP on the seven real-world workloads described in Table 1, all curated from entries in major data science competitions. All benchmarks are single-threaded. One benchmark, `purchase`, uses features automatically generated by the Deep Feature Synthesis algorithm (Kanter & Veeramachaneni, 2015); these features consist largely of aggregations such as averages and counts over relational data. Two benchmarks, `credit` and `music`, query data from stores which can be located either locally or remotely. We query `music` remotely in all benchmarks; we query `credit` remotely only in top-K benchmarks because it performs regression and cannot be cascaded. We show all benchmarks' transformation graphs in Figure 5.

[1]We cannot compare against Pretzel directly as its code and benchmarks are not publicly available

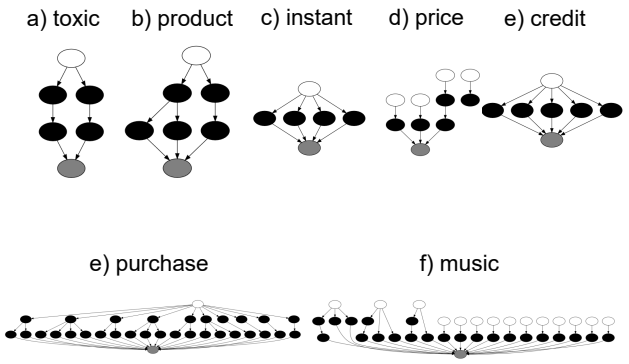


*Figure 5.* Transformation graphs of all benchmark workloads. Inputs are in white, models in gray, and transformation nodes in black.

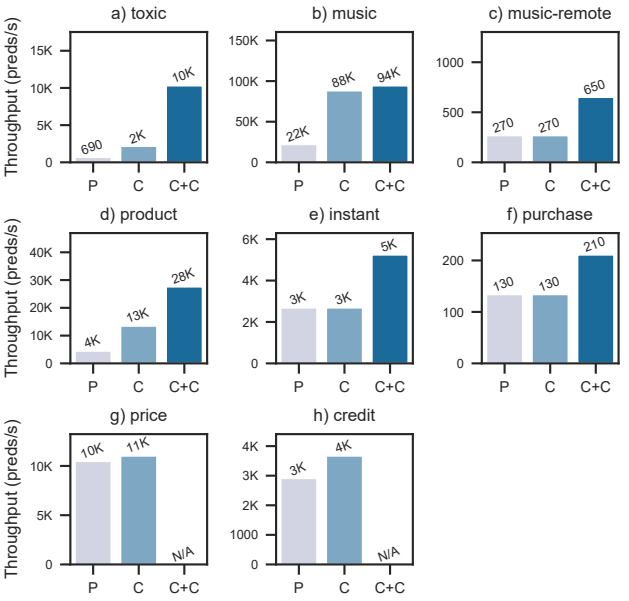


*Figure 6.* WILLUMP performance on offline batch queries. P means (unoptimized) Python, C means compiler optimizations only, C+C means compiler and cascades optimizations.

### 6.3 Evaluating WILLUMP

**WILLUMP Throughput.** We first evaluate WILLUMP on offline batch queries, showing results in Figure 6. First, we apply WILLUMP's compiler optimizations. These improve the performance of all compilable benchmarks by up to 4.3×.

Then, we apply end-to-end cascades to all classification benchmarks. For all benchmarks, we set an accuracy target of 0.1% less than the accuracy of the original model, but we did not observe a statistically significant change in accuracy for any benchmark.

End-to-end cascades improve benchmark performance by up to 5×. Interestingly, cascades are least effective on `music`, which queries pre-computed features from an in-memory database. This is very fast when compiled, so

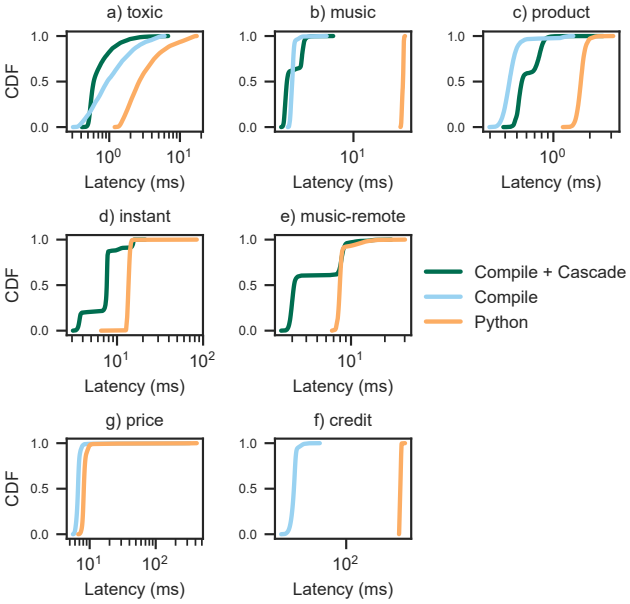


*Figure 7.* WILLUMP latency CDFs on online point queries, with one outstanding query at a time. Benchmarks in the second row contain no compilable operators; we only apply cascades. Benchmarks in the third row do not perform classification; we only apply compiler optimizations.

feature computation accounts for a small portion of overall runtime and potential gains from cascades are limited. In `music-remote`, we moved the features to a remote database, so querying them was more costly and cascades became more effective, providing a 2.4× speedup.

**WILLUMP Latency.** We next evaluate WILLUMP on online point queries, showing results in Figure 7. We evaluate all benchmarks except `purchase`; the Python implementation of the deep feature synthesis algorithm was not designed for low-latency queries. When evaluating, we make one query at a time; each query contains one data input. We first apply WILLUMP's end-to-end compiler optimizations to all compilable benchmarks. These decrease p50 and p99 latency by 1.3-400×. The large speedup on `music` and `credit` is enabled by WILLUMP's low-latency Weld drivers, which call into WILLUMP's Weld code far faster than the benchmarks' original Pandas implementations can call into their underlying C code; this is critical for point operations.

We then apply end-to-end cascades to all classification benchmarks. These improve p50 latency by up to 2.5× for most benchmarks. However, because cascades only speed up prediction of some data inputs (those classified by the approximate model), they do not improve p99 latency. Cascades are least effective on `product` and `music`; in both cases this is because the latency contribution of feature computation is small compared to that of the model. We demonstrate this with an alternate version of `music`, `music-remote`,

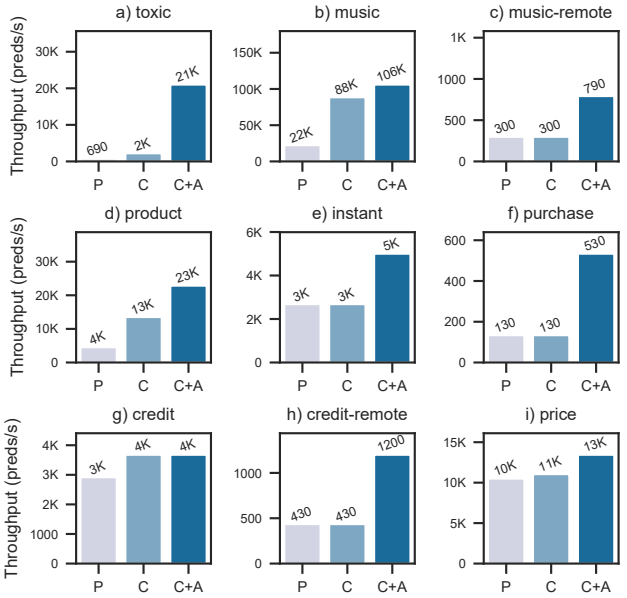


*Figure 8.* WILLUMP performance on top-K queries. P means (unoptimized) Python, C means compiler optimizations only, C+A means compiler and top-K approximation optimizations.

which queries features from remote databases instead of an in-memory store. As in the previous section, more expensive feature computation makes cascades more effective; they decrease p50 latency of `music-remote` by 2.5×.

**Top-K Queries.** We now evaluate WILLUMP on top-K queries, showing results in Figure 8. We use $K = 20$, query over the entire validation set, and set a minimum precision of 0.95.

We first apply WILLUMP's end-to-end compiler optimizations to compilable benchmarks; these perform the same as in the batch setting. We then apply WILLUMP's top-K query approximation optimization. This produces performance improvements ranging from 1.3-10× with precision always above the minimum. Smaller speedups occur in benchmarks with relatively expensive models, such as `music`, `credit`, and `price`, as well as in benchmarks where differences between scores of high-scoring candidates were small (less than a hundredth of a percent), like `product`. Benchmarks with less expensive models and more differentiation between high-scoring candidates, like `music-remote`, `credit-remote`, and `toxic`, have larger speedups.

**Integration with Clipper.** We next evaluate integration of WILLUMP with Clipper, showing results in Figure 9. We optimize `product` and `toxic` with WILLUMP's compiler and cascades optimizations and serve with Clipper, evaluating end-to-end query latencies. At a batch size of 1, WILLUMP improves p50 latency by 2.5-3× and p99 latency by 4.5-12×. At a batch size of 100, WILLUMP improves p50 latency by 4-9× and p99 latency by 5-25×. These speedups are slightly smaller than in prior experiments but increase

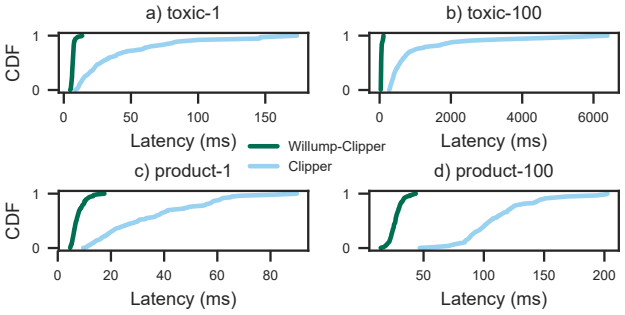


*Figure 9.* End-to-end latency CDFs on queries made using Clipper with and without WILLUMP optimization at batch sizes of 1 and 100.

| | Toxic | Music | Product | Instant | Purchase | Price | Credit |
|---|---|---|---|---|---|---|---|
| Cascades | 1.1× | 6.8× | 2.2× | 1.0× | 1.7× | N/A | N/A |
| Top-K | 1.1× | 6.9× | 1.8× | 1.0× | 1.6× | 3.8× | 3.9× |

*Table 2.* Ratios of WILLUMP to Python training times when training end-to-end cascades and top-K query approximations.

with batch size because Clipper has significant overheads, including serialization, RPC processing time, etc.

**WILLUMP Overhead.** Finally, we evaluate the training times of WILLUMP's end-to-end cascades and top-K query approximation optimizations, showing results in Table 2. WILLUMP's end-to-end training times are 1.0-6.9× as long as the end-to-end training times of the original pipelines. The increase in training time is largest for benchmarks whose end-to-end training time is dominated by model training and comes largely from the need to train several approximate models during approximate model feature selection. We believe WILLUMP's tradeoff of train performance for inference performance is acceptable because for many real ML applications, training consumes far fewer resources than inference (Hazelwood et al., 2018).

## 7 RELATED WORK

**Model Serving:** Researchers and commercial vendors have developed many model serving systems. Some are general-purpose, serving different models from different frameworks. These include research systems, such as Clipper (Crankshaw et al., 2017) and Rafiki (Wang et al., 2018), and commercial platforms, such as Amazon's Sagemaker and Microsoft's Azure ML, among others (Olston et al., 2017; Apache, 2019). These systems aim to reduce the difficulty of deploying ML models. Typically, they consider models to be black boxes and implement only pipeline-agnostic optimizations such as end-to-end caching and adaptive batching (Crankshaw et al., 2017). However, recently researchers have developed more powerful optimizations, such as the use of erasure codes to improve accuracy in the face of unavailability (Kosaian et al., 2019). As an optimizer for ML inference pipelines, WILLUMP synergizes with general-purpose model serving systems, significantly

improving their performance with its statistically-aware optimizations, as we show in Section 6.3.

Other model serving systems are application-specific. For example, Noscope (Kang et al., 2017) and Focus (Hsieh et al., 2018) improve performance of neural nets querying large video datasets. Major web companies have also developed specialized systems for video recommendation (Davidson et al., 2010) and ad-targeting (Cheng et al., 2016), among other tasks. Also related is LASER (Agarwal et al., 2014), designed for linear models used in advertising, which provides latency guarantees by dropping slow-to-compute features. WILLUMP generalizes ideas from these application-specific systems, such as their use of cascades.

**Cascades:** Cascades were initially developed for rapid object detection by applying more complex classifiers to more object-like regions of an image (Viola & Jones, 2001). They have been widely applied to image and video tasks such as pedestrian detection (Cai et al., 2015) and face recognition (Sun et al., 2013). Some application-specific model serving systems, such as NoScope and Focus, utilize cascades.

**Cost-Sensitive Training Algorithms:** ML researchers have proposed many algorithms for incorporating feature cost into model training, similar to WILLUMP's cascades. Prominent examples include Xu et al. (2014), Wang et al. (2011), and Raykar et al. (2010). However, unlike WILLUMP, these systems are neither general nor automatic. They perform no graph or dataflow analysis, assume all features are computationally independent, require users to provide costs for all features, and restrict users to specific types of models (such as linear models for Wang et al. (2011) or cascades of CART trees for Xu et al. (2014)).

**Top-K Approximation:** Database researchers have proposed many algorithms for optimizing and approximating top-K queries. These algorithms, including MPro (Chang & Hwang, 2002) and approximate TA (Theobald et al., 2004), among others (Ilyas et al., 2008), score candidates by querying one data source at a time; they drop candidates whose probability of scoring in the top K given the queried data sources is too low. However, they do not translate well to ML top-K queries. They assume that scoring functions execute on a set of data sources which can be queried independently, but ML models execute on features which are often computationally interdependent. They also assume that scoring is done by a monotonic and usually linear aggregation function, but ML models are typically nonlinear and offer no monotonicity guarantees.

Some systems for performing ML top-K queries use a *retrieval model* to select high-scoring inputs to rank with a more powerful model (Cheng et al., 2016). This is similar to WILLUMP's use of an approximate model, but retrieval models are typically manually constructed while WILLUMP constructs approximate models automatically. Therefore, to the extent of our knowledge, WILLUMP is the first system to automatically optimize ML top-K queries.

**ML Optimizers and Compilers:** Several prior systems have developed optimizers and compilers for ML workloads. Unlike WILLUMP, many specialize in improving neural network performance. For example, TVM (Chen et al., 2018) compiles deep neural nets to different architectures and NVIDIA TensorRT (Sharma & Moroney, 2018) is a library optimized for fast inference performance on NVIDIA GPUs. Additional optimization techniques include knowledge distillation (Hinton et al., 2015) and approximate caching (Kumar et al., 2019). Some systems, like WILLUMP, optimize ML pipeline performance. For example, KeystoneML (Sparks et al., 2017) optimizes distributed training pipelines. Closer to WILLUMP is Pretzel (Lee et al., 2018), which improves ML inference performance through end-to-end compiler optimizations such as loop fusion and vectorization. These are similar to the optimizations WILLUMP implements through Weld compilation, but unlike WILLUMP Pretzel does not implement statistical optimizations such as cascades or top-K approximation.

## 8 CONCLUSION

This paper presents WILLUMP, a statistically-aware end-to-end optimizer for ML inference. WILLUMP leverages unique properties of ML inference applications to automatically improve their performance through statistical optimizations, such as end-to-end cascades and top-K query approximation. WILLUMP improves the performance of real-world ML inference pipelines by up to an order of magnitude over existing systems.

## 9 ACKNOWLEDGMENTS

This research was supported in part by affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, Infosys, NEC, and VMware—as well as Toyota Research Institute, Northrop Grumman, Cisco, SAP, and the NSF under CAREER grant CNS-1651570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Toyota Research Institute ("TRI") provided funds to assist the authors with their research but this article solely reflects the opinions and conclusions of its authors and not TRI or any other Toyota entity.

## REFERENCES

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

*Mining*, pp. 173–182. ACM, 2014.

Aguiar, J. LightGBM with Simple Features, October 2018. URL https://www.kaggle.com/jsaguiar/lightgbm-with-simple-features.

Apache. Prediction IO, Apr 2019. URL https://predictionio.apache.org/start/.

Breiman, L. Random Forests. *Machine learning*, 45(1): 5–32, 2001.

Cai, Z., Saberian, M., and Vasconcelos, N. Learning Complexity-aware Cascades for Deep Pedestrian Detection. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 3361–3369, 2015.

Chang, K. C.-C. and Hwang, S.-w. Minimal Probing: Supporting Expensive Predicates for Top-k Queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pp. 346–357. ACM, 2002.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018.

Cheng, H.-T., Koc, L., Harmsen, J., Shaked, T., Chandra, T., Aradhye, H., Anderson, G., Corrado, G., Chai, W., Ispir, M., et al. Wide & Deep Learning for Recommender Systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, pp. 7–10. ACM, 2016.

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

Davidson, J., Liebald, B., Liu, J., Nandy, P., Van Vleet, T., Gargi, U., Gupta, S., He, Y., Lambert, M., Livingston, B., et al. The Youtube Video Recommendation System. In *Proceedings of the fourth ACM Conference on Recommender Systems*, pp. 293–296. ACM, 2010.

Fisher, A., Rudin, C., and Dominici, F. All Models are Wrong but Many are Useful: Variable Importance for Black-Box, Proprietary, or Misspecified Prediction Models, using Model Class Reliance. *arXiv preprint arXiv:1801.01489*, 2018.

Hazelwood, K., Bird, S., Brooks, D., Chintala, S., Diril, U., Dzhulgakov, D., Fawzy, M., Jia, B., Jia, Y., Kalro, A., et al. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 620–629. IEEE, 2018.

Hinton, G., Vinyals, O., and Dean, J. Distilling the Knowledge in a Neural Network, 2015.

Hsieh, K., Ananthanarayanan, G., Bodik, P., Venkataraman, S., Bahl, P., Philipose, M., Gibbons, P. B., and Mutlu, O. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 269–286, 2018.

Ilyas, I. F., Beskales, G., and Soliman, M. A. A Survey of Top-k Query Processing Techniques in Relational Database Systems. *ACM Computing Surveys (CSUR)*, 40 (4):11, 2008.

Kang, D., Emmons, J., Abuzaid, F., Bailis, P., and Zaharia, M. NoScope: Optimizing Neural Network Queries over Video at Scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.

Kanter, J. M. and Veeramachaneni, K. 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*, pp. 1–10. IEEE, 2015.

Kanter, M. Featuretools Year in Review, Dec 2018. URL https://blog.featurelabs.com/featuretools-2018-in-review/.

Kikani, P. IG PCA + NuSVC + KNN + LR Stack, July 2019. URL https://www.kaggle.com/prashantkikani/ig-pca-nusvc-knn-lr-stack.

Koehrsen, W. A Machine Learning Framework with an Application to Predicting Customer Churn, August 2019. URL https://github.com/Featuretools/predict-customer-churn.

Kosaian, J., Rashmi, K., and Venkataraman, S. Parity Models: Erasure-Coded Resilience for Prediction Serving Systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 30–46, 2019.

Kumar, A., Balasubramanian, A., Venkataraman, S., and Akella, A. Accelerating Deep Learning Inference via Freezing. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association. URL https://www.usenix.org/conference/hotcloud19/presentation/kumar.

Lee, Y., Scolari, A., Chun, B.-G., Santambrogio, M. D., Weimer, M., and Interlandi, M. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 611–626, 2018.

Lopuhin, K. Mercari Golf, Feb 2018. URL https://www.kaggle.com/lopuhin/kernels.

Lu, Y. An End-to-End AutoML Solution for Tabular Data, May 2019. URL https://ai.googleblog.com/2019/05/an-end-to-end-automl-solution-for.html.

Molnar, C. *Interpretable Machine Learning*. 2019. https://christophm.github.io/interpretable-ml-book/.

Nguyen, T. T., Fani, H., Bagheri, E., and Titericz, G. Bagging Model for Product Title Quality with Noise. 2017.

Olston, C., Fiedel, N., Gorovoy, K., Harmsen, J., Lao, L., Li, F., Rajashekhar, V., Ramesh, S., and Soyke, J. Tensorflow-Serving: Flexible, High-Performance ML Serving. *arXiv preprint arXiv:1712.06139*, 2017.

Palkar, S., Thomas, J. J., Shanbhag, A., Narayanan, D., Pirk, H., Schwarzkopf, M., Amarasinghe, S., and Zaharia, M. Weld: A Common Runtime for High Performance Data Analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017.

Palkar, S., Thomas, J., Narayanan, D., Thaker, P., Palamuttam, R., Negi, P., Shanbhag, A., Schwarzkopf, M., Pirk, H., Amarasinghe, S., et al. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Proceedings of the VLDB Endowment*, 11(9):1002–1015, 2018.

Raykar, V. C., Krishnapuram, B., and Yu, S. Designing Efficient Cascaded Classifiers: Tradeoff between Accuracy and Cost. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 853–860. ACM, 2010.

rn5l. WSDM Music Recommendation Challenge, Jan 2018. URL https://github.com/rn5l/wsdm-cup-2018-music.

Sharma, S. and Moroney, L. Announcing TensorRT integration with TensorFlow 1.7 , Mar 2018. URL https://developers.googleblog.com/2018/03/tensorrt-integration-with-tensorflow.html.

Sparks, E. R., Venkataraman, S., Kaftan, T., Franklin, M. J., and Recht, B. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pp. 535–546. IEEE, 2017.

Sun, Y., Wang, X., and Tang, X. Deep Convolutional Network Cascade for Facial Point Detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3476–3483, 2013.

Theobald, M., Weikum, G., and Schenkel, R. Top-k Query Evaluation with Probabilistic Guarantees. In *Proceedings of the Thirtieth International Conference on Very Large Databases-Volume 30*, pp. 648–659. VLDB Endowment, 2004.

Tunguz, B. Logistic Regression with Words and Char N-grams, Mar 2018. URL https://www.kaggle.com/tunguz/kernels.

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

Wang, L., Lin, J., and Metzler, D. A Cascade Ranking Model for Efficient Ranked Retrieval. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 105–114. ACM, 2011.

Wang, W., Gao, J., Zhang, M., Wang, S., Chen, G., Ng, T. K., Ooi, B. C., Shao, J., and Reyad, M. Rafiki: Machine Learning as an Analytics Service System. *Proceedings of the VLDB Endowment*, 12(2):128–140, 2018.

Wang, X., Luo, Y., Crankshaw, D., Tumanov, A., Yu, F., and Gonzalez, J. E. IDK Cascades: Fast Deep Learning by Learning Not to Overthink. *arXiv preprint arXiv:1706.00885*, 2017.

Xu, Z., Kusner, M. J., Weinberger, K. Q., Chen, M., and Chapelle, O. Classifier Cascades and Trees for Minimizing Feature Evaluation Cost. *The Journal of Machine Learning Research*, 15(1):2113–2144, 2014.

## A MICROBENCHMARKS

In this section, we analyze the behavior and overhead of WILLUMP's optimizations in more detail.

**Cascades Tradeoffs.** We first examine the behavior of WILLUMP's cascades optimization. For each classification benchmark, we graph performance and accuracy at varying cascade thresholds in Figure 10. On each graph, a blue circle marks the performance of the original model and an orange X that of the approximate model; points in between are cascaded models with varying cascade thresholds. As a reminder, the cascade threshold is the confidence the approximate model must have in a prediction to return the prediction and not cascade to the original model.

For all benchmarks, cascaded models with high cascade thresholds are faster than the original model but have similar accuracy. WILLUMP's cascades algorithm automatically chooses these cascade thresholds to maximize performance without statistically significant accuracy loss. On each graph, the point marked with a red diamond is the chosen threshold.

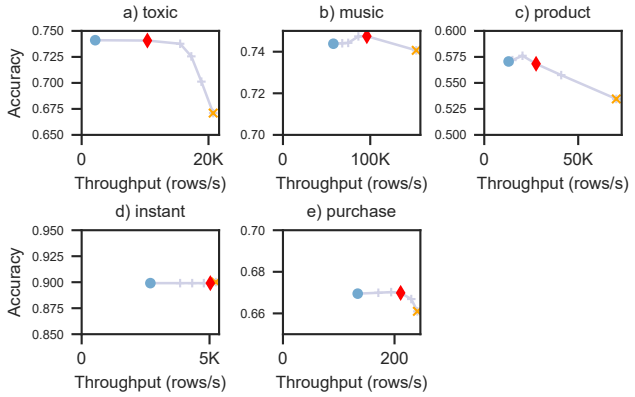


*Figure 10.* Throughput versus accuracy on all classification benchmarks with varying cascade thresholds. A blue circle marks performance of the original model, an orange X marks performance of the approximate model, and a red diamond marks WILLUMP's chosen cascade threshold.

For most benchmarks, as the cascade threshold decreases, performance continues to improve but accuracy falls off. This shows cascades are working as intended. At high cascade thresholds, the approximate model classifies easy data inputs and the original model classifies hard data inputs, so accuracy is high. At low thresholds, the approximate model must classify hard data inputs, so accuracy falls.

**Compilation Times** We also evaluate WILLUMP's pipeline compilation times. We find these rarely exceed thirty seconds. The exceptions are benchmarks which query in-memory data stores (`music` and `credit`); it takes WILLUMP several minutes to optimize them as it must convert the data stores into a format which Weld can query.

# B ARTIFACT APPENDIX

## B.1 Abstract

This artifact contains the implementation of the WILLUMP optimizer described in this paper. Furthermore, this artifact contains the scripts and datasets needed to reproduce the key performance results in this paper, specifically the batch, point, and top-K performance comparisons in Figures 6, 7, and 8.

## B.2 Artifact check-list (meta-information)

- **Program:** The WILLUMP optimizer implemented in Python.
- **Data set:** Competition datasets for each benchmark, all provided or linked to.
- **Run Time Environment:** Python 3 (Tested with Python 3.6.8).
- **Metrics:** For batch (Figure 6) and top-K (Figure 8) experiments, pipeline throughput. For point (Figure 7) experiments, pipeline latency.
- **Output:** Throughputs and latencies (p50 and p99) printed to stdout.
- **Experiments:** Run each benchmark in the batch, point, and top-K setting with no optimizations, compiler optimizations only, and compiler optimizations plus cascades or top-K approximation optimizations.
- **Publicly available?:** Yes.

## B.3 Description

### *B.3.1 How delivered*

The artifact is hosted on GitHub at `https://github.com/stanford-futuredata/Willump`. The artifact, its instructions, its benchmark scripts, and its datasets are publicly available. Additionally, an archival version of the artifact is available at `https://doi.org/10.5281/zenodo.3687193`.

### *B.3.2 Software dependencies*

Experiments were run using Ubuntu 18.04 with Python 3.6.8. Full installation instructions including dependencies are here: `https://github.com/stanford-futuredata/Willump/blob/master/README.md`.

### *B.3.3 Data sets*

The datasets of the competitions our benchmarks were curated from were used. All are included in the artifact or linked to from `https://github.com/stanford-futuredata/Willump/blob/master/BENCHMARKS.md`.

## B.4 Installation

Full installation instructions are here: `https://github.com/stanford-futuredata/Willump/blob/master/README.md`.

## B.5 Experiment workflow

Full benchmark instructions are here: `https://github.com/stanford-futuredata/Willump/blob/master/BENCHMARKS.md`.

## B.6 Evaluation and expected result

Artifact evaluation is expected to reproduce the performance comparisons in Figures 6, 7, and 8 of the paper. We expect the results of evaluation to show a similar performance trend as these figures.