Kitana: Efficient Data Augmentation Search for AutoML

Zezhou Huang
zh2408@columbia.edu
Columbia University

Pranav Subramaniam
psubramaniam@uchicago.edu
University of Chicago

Raul Castro Fernandez
raulcf@uchicago.edu
University of Chicago

Eugene Wu
ewu@cs.columbia.edu
DSI, Columbia University

ABSTRACT

AutoML services provide a way for non-expert users to benefit from high-quality ML models without worrying about model design and deployment, in exchange for a charge per hour ($21.252 for VertexAI). However, ML models are only as good as the quality of training data. With the increasing volume of data available, both within enterprises and to the public, there is a huge opportunity for training data augmentation. For instance, vertical augmentation adds predictive features, while horizontal augmentation adds samples. This augmented training data yields potentially much better ML models through AutoML at a lower cost. However, existing AutoML and data augmentation systems either forgo the augmentation opportunities that provide poor models, or apply expensive augmentation searching techniques that drain users' budgets.

We present Kitana, an AutoML system with practical data augmentation search. Kitana manages a corpus of datasets, and exposes an AutoML interface to users. To search for augmentation opportunities at a minimum cost, Kitana applies aggressive precomputation to train a *factorized proxy model* and evaluate each candidate augmentation within 0.1*s*. Kitana also uses a cost model to limit the time spent on augmentation search, supports expressive data access control, and performs request caching to benefit from past similar requests. Using a corpus of 513 open-source datasets, we show that Kitana produces higher quality models than existing AutoML systems in orders of magnitude less time. Across different user requests, we find Kitana increases the model R2 from ~0.16 → ~0.66 while reducing the cost by >100×.

PVLDB Availability Tag:
The source code of this research paper has been made publicly available at https://anonymous.4open.science/r/Kitana-Experiment-D0B2.

1 INTRODUCTION

Cloud-based AutoML services [5, 6, 16, 19] offer the promise to make machine learning (ML) accessible to non-machine learning experts. Given a training set T and target attribute Y, AutoML automatically searches for a good model and parameters. Technical barriers such as model design and tuning, training details, and even deployment barriers are hidden from the user in exchange for a charge per hour (e.g., as of Nov 2022, VertexAI [16] charges $21.252). However, the quality of these services is limited by the initial training data T. Without examples with predictive features, no amount of training will help.

We observe that increasingly large dataset repositories within and across organizations (e.g., data lakes, open data portals, and databases) can be used for *data augmentation*. If we could efficiently search these repositories to find datasets to *augment* the user's training data—by joining with datasets to acquire new features (*vertical augmentation*), unioning with datasets to acquire new samples (*horizontal augmentation*), or a combination of the two—we could dramatically improve AutoML quality, speed, *and* cost. At the same time, not all data in these repositories are, nor should be, accessible to everyone—even within an organization teams have varying levels of access to data. Any augmentation system must respect these access requirements.

We believe a practical data augmentation search for AutoML should meet four criteria. (**C1**) Most importantly, augmentation search *must be fast*. In the cloud, time is money. The search procedure should be independent of the augmented dataset's cardinality, avoid expensive model retraining, and efficiently search a large space of augmentation candidates. (**C2**) The search criteria should be correlated with model improvement. Investing more time (and money) should return *better* models. (**C3**) It should support a mixture of vertical and horizontal augmentations to make the best use of the available dataset repository. (**C4**) Finally, it should support varying levels of data release. For instance, public datasets can be shared in their raw form with the user, whereas private datasets may be shared with the augmentation system to enhance a model's predictions but *not* shareable with the user [46].

Recent data augmentation systems identify a set of candidate tables for augmentation, and for each augmentation, assess its improvement to the model. Techniques like ARDA [28] only support vertical augmentation (C3), materialize the augmented dataset, and retrain a model to assess the augmentation quality. This trades-off time for model quality (C2), but is too slow and expensive for cloud environments (C1). Other approaches such as Li et al. [47] use a proxy "Novelty" metric to estimate only horizontal augmentation candidates (C3). However, our experiments show that "Novelty" is both slow (C1) and can be uncorrelated with actual model improvements (C2). To summarize, existing approaches are slow (C1), may not actually improve the model quality (C2), do not support a mixture of horizontal and vertical augmentation (C3), and do not support different levels of data release (C4).

This paper describes Kitana, a data-augmented AutoML system that satisfies the above four criteria. Kitana combines and extends ideas from factorized learning [57] and indexing [32] to enable the following novel capabilities. We introduce a cheap-to-train *proxy model* that reflects the final model quality and is designed to avoid expensive model retraining for each augmentation candidate. Specifically, we pre-compute cheap sketches for each dataset using ideas from factorized learning and semi-ring aggregation, and use

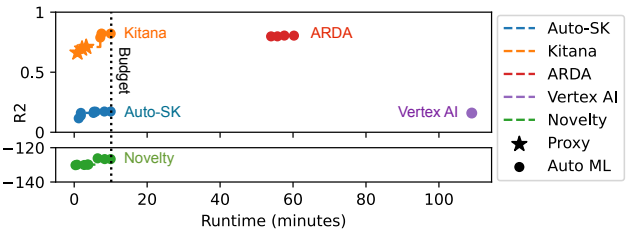


Figure 1: AutoML acc (testing $R2$) for *Gender* dataset with data augmentation using a corpus of 513 datasets. The time budget is 10 minutes (dotted vertical line). Kitana allocates ~ 3 minutes for data augmentation using a proxy model and reaches ~ 0.7 acc; Kitana then sends the augmented dataset to AutoML and further improves acc to 0.82 with the remaining budget. Other baselines are either slow or have low acc.

them to efficiently update the proxy model (a linear model) in $O(n^2 j)$ time[1] for any augmentation.

Given a time budget t, Kitana splits it between augmentation search and AutoML training. Kitana estimates the model search time [50, 61]: when the estimated time exceeds the remaining budget, Kitana stops augmentation search and trains the final model using an existing AutoML service or library. This helps Kitana stay within budget *and* return a model that is more accurate than all existing data augmentation and AutoML systems.

Figure 1 summarizes our major results compared to prior data augmentation techniques (ARDA [28], Novelty [47]); AutoML libraries (AutoSklearn); and major cloud AutoML services (Google's Vertex AI). Since they don't accept a time budget and return a dataset, we run the augmentation techniques to completion and then run AutoSklearn to derive a model. We set AutoSklearn's and Vertex AI's budgets to 10 minutes, and report its $R2$ throughout model search; Vertex AI doesn't enforce the budget.

The training set doesn't have predictive features, so AutoSklearn and VertexAI do not find good models. ARDA takes $\approx$50 minutes for data augmentation, and the final model is still slightly worse than Kitana ($R2 = 0.81$ vs 0.82). Novelty is uncorrelated with model accuracy and degrades the final model. Finally, Kitana's proxy models (stars) are already highly competitive, and it leaves enough time to train the final model using AutoML (circles).

In addition to higher accuracy, faster results, and cheaper cost, Kitana supports sensible dataset access controls via *data release labels*. Labels indicate what can be released to the user, and what data providers need to share with Kitana. A data provider can label each dataset so the raw data can be released to the user (*RAW*), only the trained model can be released (*MD*), or the user can perform inference using the model but the model cannot be released (*API*). A user request can specify whether they want the full augmentation plan and associated datasets, a model trained on augmented data, or a prediction API, and Kitana will automatically adjust its search procedure to satisfy data release labels. Although the implementation to support these controls is relatively straightforward, the labels mirror access control expectations in the wild [31, 46]. They also illustrate the complex interaction between dataset access, system design, and user-facing API semantics. We leave a more complete treatment of access control to future work.

We evaluate Kitana using a corpus of 513 datasets collected from open data repositories and compare its performance against existing AutoML services like Auto-Sklearn and Google's Vertex AI. Across all user requests based on 5 real-world datasets, we find that Kitana is faster, cheaper, and of higher quality than AutoML services. Specifically, Kitana takes less than 1 minute to exceed the accuracy that AutoML reaches after $\geq 10min$; for the same time budget, Kitana increases the model $R2$ scores by $\sim$0.05–0.30.

To summarize, we contribute the following:

- The design and implementation of Kitana, an AutoML system that uses practical data augmentation. Kitana efficiently searches a corpus of datasets for augmentation opportunities that increase the final model as compared to using AutoML services alone.
- Kitana leverages careful pre-computation and factorized learning to evaluate each augmentation within $\sim 100ms$.
- Kitana supports a range of access controls between the data providers and end-users.
- Extensive evaluation against two AutoML systems (Google's VertexAI [16] and AutoSklearn) using a corpus of real-world datasets. We show that Kitana is cheaper and better: Kitana reduces request latency by over 10× for the same model accuracy and increases accuracy by over 30% for the same time budget.

2 PROBLEM DESCRIPTION

This section introduces our terminology, related AutoML services, and the augmentation search problem definition.

2.1 Preliminaries

Kitana focuses on supervised machine learning tasks over tables:

Tables and Dataset Corpus. Let D denote a general relational table and the list of attributes $S_D = [a_1, \ldots, a_m]$ be its schema. A dataset corpus $\mathcal{D} = \{D_1, ..., D_n\}$ is a set of tables. We focus on two forms of augmentation: *horizontal augmentation* unions two tables $D_1 \cup D_2$ that are schema compatible, and *vertical augmentation* joins two tables $D_1 \bowtie D_2$ together to add additional attributes.

Supervised Machine Learning. A supervised machine learning task, such as classification and regression, takes a tabular dataset T that is drawn from an underlying distribution as input. T consists of feature attributes $X \subset S_T$ and a target attribute $Y \in S_T$. Supervised ML finds a model with parameters θ which transforms X in a way that best predicts Y, even when the specific X, Y pair has not been seen before. The model type M may be specified by the user, or automatically found using a meta-search process. During the training process and for the hyperparameter optimization, the performance of models under different hyperparameters is evaluated using cross validation [53] over T. The final returned model is evaluated once on a test dataset V.

AutoML Services. Today's AutoML services take as input T and Y, and automatically search for a good model and parameter θ. Additionally, some AutoML services also deploy the model and return an inference API. This helps non-experts benefit from ML and not worry about model design, model training, and deployment barriers.

[1]Where n is the number of features and j is the join key domain size; note the time complexity is independent of the cardinality of the augmented dataset

Developers can benefit from open-source packages such as Auto-Sklearn [35] and FLAML [60], and commercial solutions such as H2O [17]. End users can benefit from cloud-based AutoML services, such as Google's Vertex AI [16], Microsoft's Automated ML [6], Amazon's SageMaker AutoPilot [5], and Oracle's AutoML [19].

Model training and search are time-critical in AutoML as AutoML services adopt an hourly pricing model. For instance, as of November 2022, VertexAI charges \$21.252/hr to train on tabular data, AWS Sagemaker charges \$0.115 – 5.53/hr depending on the machine, and Azure AutoML charges \$0.073 – 27.197/hr depending on the machine. Running AutoML longer will search a larger space and increase the chances of finding an accurate model. For instance, the blue line in Figure 6 shows how the model accuracy improves from 0 when Auto-Sklearn starts to 0.4 – 0.6 after 10 minutes. Since the search process can potentially run for a very long time, users typically set a time budget t.

More data is available than ever before—through public government portals, enterprise data lakes, data marketplaces, logging tools, and IoT devices. This is a huge opportunity to go beyond merely model search [55] for AutoML, and to also vertically and horizontally augment training datasets [28, 47].

2.2 Problem Definition

Let $\mathcal{D}$ be a corpus of datasets. Each dataset $D \in \mathcal{D}$ is mapped to an access label $l(D) \in \{RAW, MD, API\}$ from data providers for access control. The labels are totally ordered $RAW < MD < API$. RAW allows D to be released to the user. MD allows the models θ trained over D to be released, but not D. API disallows θ to be released, but allows a prediction API that internally uses θ.

The user submits a tuple (t, T, M, R), which consists of a time budget t, training dataset T, an optional model types M (l for linear regression, $\cup$ for all models), and return labels $R \subseteq \{RAW, MD, API\}$ that describe what Kitana should return. RAW returns the raw augmentation data along with the augmentation plan introduced next, MD returns model θ and API returns prediction API. The user additionally has a testing dataset V with the same underlying distribution as T, but only accessible by the user.

We search for a good *Augmentation Plan* $\mathcal{P} = [A_1, \cdots, A_n]$ that specifies a sequence of horizontal or vertical augmentation A: Horizontal augmentation contains augmentation data D_A and vertical augmentation additionally contains join key j_A. $A(I)$ applies an augmentation to a dataset I, and is defined as $A(I) = I \cup D_A$ for horizontal augmentation, and $A(I) = I \bowtie_{j_A} D_A$ for vertical augmentation[2]. Applying augmentation plan to training set T is defined as $\mathcal{P}(T) = A_n(A_{n-1}(\cdots A_1(T)))$.

The access label $l(D)$ on each corpus dataset, along with the user's requested return type R, place access control constraints on the search problem: the search space is restricted to $\sigma_{l(D) \le min(R)}(\mathcal{D})$. Note that if $R = \{MD\}$, the augmentation type is restricted to horizontal augmentation. This is because the user will not be able to vertically augment new records for prediction using θ. Therefore, R offers users a trade-off between model performance and explainability: for high model performance, $R = \{API\}$ can leverage all of $\mathcal{D}$, but if users want to inspect raw data and model parameters, the search is over a restricted subset.

Putting everything together, we present the formal problem definition for one user request:

Problem 1. *Given request (t, T, M, R) and testing dataset V not available during search, find an augmentation plan $\mathcal{P}^*$ that maximizes test accuracy:*

$$\begin{aligned} \mathcal{P}^* = \quad & \underset{\mathcal{P}}{\operatorname{argmax}}\, acc(\theta, \mathcal{P}(V)) \\ s.t. \quad & \theta = AutoML.train(\mathcal{P}(T)), \\ & t_{data} + t_{md} \le t, && \textit{(time budget constraint)} \\ & \theta \in M, && \textit{(model type constraint)} \\ & \forall A \in \mathcal{P}^*, l(D_A) \le min(R). && \textit{(access control constraint)} \end{aligned}$$

where t_{data} and t_{md} are the times for augmentation search and AutoML model search, respectively.

Hardness. Feature selection is an NP-hard problem [27] that can be reduced to finding the optimal augmentation plan, where each feature is a normalized vertical augmentation. From a practical perspective applying AutoML to evaluate every possible augmentation plan is expensive and not scalable.

3 KITANA OVERVIEW

We now present the system architecture, how Kitana wraps the core problem definition to provide an AutoML-like service, and the lifecycle of a user's request.

3.1 Kitana Architecture

Figure 2 presents the main system components and control flow, where the black components are part of Kitana and the blue components can be outsourced to an external service. Section 5 describes the components in greater detail.

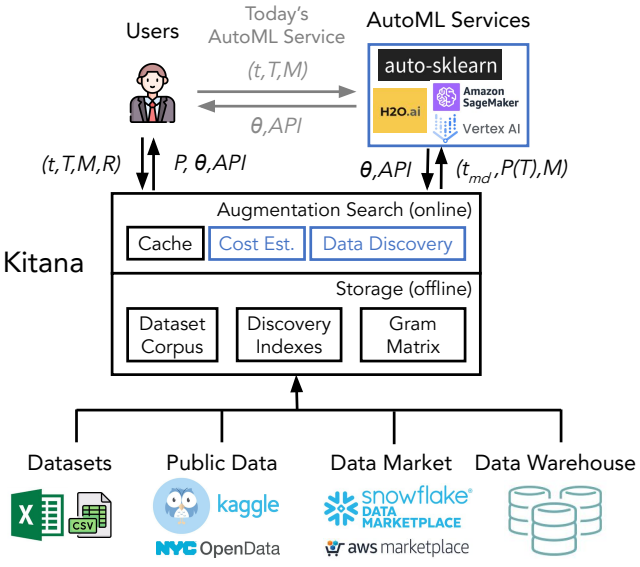


Figure 2: Overview of Kitana Architecture.

[2]For notational convenience, the text assumes the j_A is the same in I and D_A. More generally, Kitana uses Aurum for candidate discovery, which supports equijoins; the factorized learning techniques are general to any join, including theta and anti-joins.

Existing AutoML services follow the gray path, where the user submits a request and receives model parameters and/or a prediction API. In contrast, Kitana is an intermediary between the user and AutoML. During online operation, the user submits a request. Kitana finds the optimal augmentation plan $\mathcal{P}^*$ (Problem 1), and passes the augmented dataset to an AutoML library or service. Depending on the request's return labels, Kitana returns the trained model parameters, prediction API, and/or augmented dataset.

To search for $\mathcal{P}^*$ in a tractable way, Kitana iteratively adds the next best augmentation in a greedy fashion that mirrors bottom-up feature selection algorithms [37]. It evaluates augmentations using *proxy models* that avoid model retraining. Then, Kitana either returns the *proxy model* if the desired model type is linear $M = l$, or materializes and sends the augmented training dataset to a separate AutoML library or service. During offline setup, data providers register datasets, and Kitana builds sketches and indexes over them to accelerate the augmentation search.

Offline Phase. Data providers call Kitana's $upload(D, l)$ function to upload dataset D with access level l (Section 2.2). Kitana then computes sketches for data discovery and augmentation search—it pre-aggregates attribute profiles for data discovery systems (Kitana uses Aurum [32], see Section 5.1.2) and factorized sketches to accelerate augmentation candidate assessment (Section 4). Kitana also applies simple cleaning, standardization, and feature transformations. Optionally, the data provider could upload the profiles and sketches, so Kitana does not have access to the raw datasets.

Online Phase. The user submits request (t, T, M, R), and the *Augmentation Search* component first checks the cache for an augmentation plan $\mathcal{P}'$ returned from prior requests with the same training schema S_T. It then uses a data discovery service (Aurum in our implementation) to find augmentable (union- or join-compatible) datasets; Aurum takes attribute profiles of training data T as input (e.g., MinHash, covariance matrix), and returns a set of candidate augmentations. After that, Kitana evaluates each augmentation candidate via the *proxy model* and greedily builds the augmentation plan. Kitana further combines *proxy model* and factorized learning with pre-processing techniques to reduce augmentation candidate evaluation cost to $\sim 100ms$ per dataset (Section 4).

Kitana needs to judiciously use the time budget, especially if there are no good augmentations. To do so, Kitana uses the sketches to compute the output size of an augmentation candidate. It uses the size to estimate the time to run AutoML t_{md} on the augmented dataset using existing estimation techniques [20, 50, 61]. If the augmentation candidate does not improve the proxy model's cross-validation accuracy, or when t_{md} exceeds the remaining budget (Section 5.2), Kitana materializes the augmented training dataset $T' = \mathcal{P}^*(T)$ and calls the AutoML service.

Finally, based on user-requested return types R, Kitana returns the augmentation plan $\mathcal{P}^*$ along with raw augmentation datasets D, model θ, and/or the prediction API. The API takes non-augmented records T' as input, applies $\mathcal{P}^*(T')$, makes a prediction using θ, and returns the prediction to the user.

3.2 Request Processing Flow

Algorithm 1 outlines the core logic in Kitana when handling a request. The user sends a request (t, T, M, R) and Kitana outputs $API, \theta, \mathcal{P}$ based on user-requested returned types R (Section 2.2).

To start, Kitana initializes an empty augmentation plan $\mathcal{P}$. If T is schema compatible with a prior request, Kitana evaluates the cached plans $\mathcal{P}'$ and chooses a plan that most improves the proxy model performance by $\geq \delta$ (L2,3). Kitana then greedily builds the augmentation plan until the estimated time to materialize and train the model would exceed the remaining time budget (L4-18).

In each iteration (L4), Kitana uses the data discovery system to find a set $\mathcal{A}$ of horizontal and vertical augmentation candidates that are compatible with the request's return types and the datasets' data release labels (L6). Each augmentation candidate A specifies the augmentation *type* ("horiz" or "vert"), the annotated relation D (Section 4.1), and join key j for vertical augmentations (L8). To simplify the search procedure, Kitana applies horizontal before vertical augmentation (L9). The intuition is that the data discovery system already finds union compatible projections, thus any dataset D that is union compatible to T after vertical augmentation will have a union compatible projection before vertical augmentation (though the resulting tables may not be exactly the same).

Before evaluating candidate A, Kitana adds it to the current plan $\mathcal{P}^*$(L10) and cheaply estimates the shape (# of attributes and rows) of augmented training data (L11) by an optimized count query (Section 4.1). If AutoML is needed ($M \neq l$), Kitana uses a cost model to estimate how long AutoML would take on it—if the estimate exceeds the remaining time budget t_{remain}, Kitana skips the candidate (L12). In addition, a separate timer thread terminates

Algorithm 1 HandleRequest(t, T, M, R)

1: $\mathcal{P}^*$ = initEmptyPlan() ▷ Empty augmentation plan
2: $\mathcal{P}'$ = cacheLookup(T)
3: $\mathcal{P}^* = \mathcal{P}'$ **If** $\mathcal{P}'$ increases accuracy $\geq \delta$
4: **while** true **do** ▷ Greedily add augmentations
5: $(acc^*, A^*) = (-\infty, \emptyset)$ ▷ Best accuracy, augmentation found
6: $\mathcal{A}$= dataDiscovery$(\mathcal{P}^*(T).profile, R)$
7: **for** $A \in \mathcal{A}$ **do**
8: $type, D, j = A$ ▷ Aug. type, data, (opt.) join key
9: **continue If** type = "horiz" $\wedge$ $\mathcal{P}^*$ has vert. aug
10: $\mathcal{P}' = \mathcal{P}^*.add(A)$
11: $T' = \mathcal{P}'(T)$ ▷ Augment data (not materialized).
12: **continue If** $M \neq l \wedge t_{remain} < cost(T', M)$
13: $acc, \theta = model.trainAndEvaluate(T')$
14: $(acc^*, \theta^*, A^*, \mathcal{P}^{*\prime}) = (acc, \theta, A, \mathcal{P}')$ **If** $acc > acc^*$
15: **end for**
16: **break If** $\Delta acc^* < \delta \vee (M \neq l \wedge t_{remain} < cost(\mathcal{P}^{*\prime}(T), M))$
17: $\mathcal{P}^* = \mathcal{P}^{*\prime}$
18: **end while**
19: **if** $M \neq l$ **then**
20: θ^* = AutoML$(t_{remain}, \mathcal{P}^*(T).materialize(), M)$
21: **end if**
22: SaveToCache(T, $\mathcal{P}^*$)
23: **return** $\mathcal{P}^*$ **If** $RAW \in R$; θ^* **If** $MD \in R$;
24: $constructAPI(\theta^*, \mathcal{P}^*)$ **If** $API \in R$

the augmentation search if the estimated time to run AutoML on the best augmentation plan so far may exceed the remaining budget.

To evaluate candidate A, Kitana trains the proxy model and evaluates the model accuracy using 10-folds cross-validation (L13). Internally, these two steps are factorized: T' in L11 is not materialized and only the aggregates needed for training and evaluation in L13 are computed. Our pre-processing optimization further accelerates this evaluation (Section 4.2), and Kitana keeps the candidate if it improves the best accuracy so far (L14). After evaluating the candidate plans, Kitana keeps the augmentation if it improves the accuracy by $\geq \delta$ over the previously chosen augmentation (L4) and there is time to train the augmented dataset $\mathcal{P}^*(T)$ using AutoML (L16). Otherwise, it breaks the loop.

If the user requested models other than linear regression (L19), Kitana materializes the final augmented training dataset $\mathcal{P}^*(T)$ and sends it along with the remaining time budget t_{remain} and model types M to the AutoML service (L20). Otherwise, Kitana skips this step and uses the proxy model's parameters. Kitana uses the final model parameters (from AutoML or the proxy model) to construct an API that first applies $\mathcal{P}^*$ to a test example before using the model to make a prediction (Section 5.2.4); if the AutoML service returns an API, Kitana can wrap that as well, however the user will also incur the AutoML service's costs to perform inference. Kitana caches the new plan on L22. Finally, depending on R, Kitana returns the augmentation plan (along with the raw datasets), trained model θ, and/or new prediction API based on R (L23).

4 FACTORIZED DATA AUGMENTATION

Factorized learning [8, 43, 56] trains machine learning models over the output of a join query $\mathbf{X} = \mathbf{R}_1 \bowtie ... \bowtie \mathbf{R}_k$ without materializing the join. It treats training as an aggregation function and distributes the aggregation through the joins. As such, the cost is linear in the relation sizes rather than the join results. Factorized learning has been extended to many models, including linear regression [57], ridge regression with regularization, factorization machines [56], classification models using ridge regression [49], generalized linear models [39], K-means [29], and SVMs [42].

Our primary contribution is to show that factorized learning is a natural fit for data augmentation. Evaluating an augmentation candidate—materializing the augmented dataset, retraining the model, and cross-validation—is by far the biggest bottleneck [28, 47] in Kitana, and factorization avoids materialization altogether. Thus, Kitana trains and cross-validates a factorized model as the proxy to evaluate an augmentation candidate, and uses linear regression by default. In fact, proxy models are often more accurate than AutoML models on the user's original training data (without augmentation), take orders of magnitude less time, and are often preferred in practice for interpretability and reliability reasons [62].

However, naively using factorized learning still needs to repeatedly compute these "training" aggregates for each augmentation candidate. Instead, Kitana identifies aggregates sharable across augmentations and pre-computes them—they are computed independently for each dataset in the repository, and in theory could be offloaded to the data provider during dataset registration. This lets Kitana outperform naive factorized learning by over two orders of magnitude on augmentation benchmarks. The rest of this section first introduces the basic concepts in factorized learning, then describes the working sharing opportunities Kitana exploits for efficient data augmentation search.

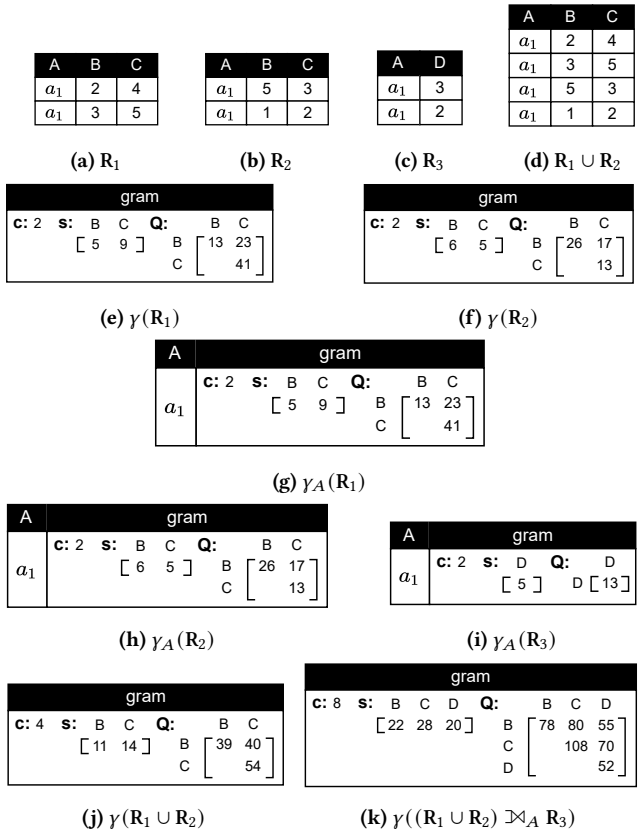


Figure 3: Factorized Learning Example

4.1 Basic Concepts

We now present semi-ring annotations and aggregation push-down, which encapsulate the core factorized learning optimizations. We then illustrate these concepts using linear regression as an example.

4.1.1 Semi-ring Annotations. A semi-ring is a set $\mathbf{S}$ that contains 0 and 1 elements, and supports the binary operators + and ×; it is typically defined as the tuple $(\mathbf{S}, 0, 1, +, \times)$. Different semi-rings are designed to support different model types. Each tuple is annotated with an element from $\mathbf{S}$, and $R[t] \in \mathbf{S}$ denotes tuple t's annotation in relation R. Relational operators are extended to compute their output relation's annotations: $\bowtie$ multiplies the annotations of matching input tuples, while γ sums the annotations in the same group-by bin. Formally, given relations R, T with schemas S_R, S_T:

$$(R \bowtie T)[t] = R[\pi_{S_R}(t)] \times T[\pi_{S_T}(t)] \quad (1)$$

$$(\gamma_{\mathbf{A}} R)[t] = \sum \{R[t_1] | t_1 \in R, t = \pi_{\mathbf{A}}[t_1]\} \quad (2)$$

Here, t is the output tuple, and $R[\pi_{S_R}(t)]$ is the input annotation of t projected onto R.

Example 1. *Consider aggregation query $\gamma_{A,SUM(B)}(R_1 \bowtie R_3)$ (Figure 3a and Figure 3c). To support $SUM(B)$ aggregation, we use natural number semi-ring, and initially annotate $R_1[t] = \pi_B(t), R_3[t] = 1$. $R_1 \bowtie R_3$ will produce a cartesian product (as A has the same value) of four tuples with annotations $2\times1, 3\times1, 2\times1, 3\times1$. $\gamma_{A,SUM(B)}$ then sums the annotations: for the only a_1 group, the sum is $2+3+2+3 = 10$.*

The core factorized learning optimization is to push aggregation through joins by exploiting the fact that join distributes over addition as in elementary algebra. For the previous example, we can rewrite $\gamma_{A,SUM(B)}(R_1 \bowtie R_3) = (\gamma_{A,SUM(B)}(R_1)) \bowtie_A (\gamma_{A,SUM(B)}(R_3))$; the sum for group a_1 then is computed as $(2+3)\times(1+1) = 10$. More generally, assuming j is the set of join keys between R and T:

$$\gamma_{\mathbf{A}}\left(R \bowtie_j T\right) = \gamma_{\mathbf{A}}\left(\gamma_{\mathbf{A},j}(R) \bowtie_j \gamma_{\mathbf{A},j}(T)\right)$$

This reduces intermediate sizes and thus join-aggregation costs.

4.1.2 Factorized Learning. We now illustrate how linear regression can be expressed as a single semi-ring join-aggregation query.

Linear regression uses training data $\mathbf{X}$ and target variable Y to learn parameters θ that minimize the model's squared loss: $\hat{\theta} = arg\,min_\theta||Y-\mathbf{X}\theta||^2$. The closed form solution is $\theta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T Y$. By treating Y as a special feature, we can see that the bottleneck computation is the gram matrix $\mathbf{X}^T\mathbf{X}$, where each cell is the sum of products between pairs of features. $\mathbf{X}^T\mathbf{X}$ has shape $m\times m$, where m is the number of features and is assumed to be much smaller than the number of rows. As a result, computing θ could be considered as a cheap postprocessing over the small $\mathbf{X}^T\mathbf{X}$.

The gram matrix semi-ring [57] helps efficiently compute $\mathbf{X}^T\mathbf{X}$. For a training set with m features, the gram matrix annotation is a triple $(c, \mathbf{s}, \mathbf{Q}) \in (\mathbb{Z}, \mathbb{R}^m, \mathbb{R}^{m\times m})$ that respectively contain the tuple count, sum of features, and sum of products between each pair of features. The zero and one elements are $\mathbf{0} = (0, \mathbf{0}^m, \mathbf{0}^{m\times m})$ and $\mathbf{1} = (1, \mathbf{0}^m, \mathbf{0}^{m\times m})$. The $+$ and $\times$ operators between two annotations $a = (c_a, \mathbf{s}_a, \mathbf{Q}_a)$ and $b = (c_b, \mathbf{s}_b, \mathbf{Q}_b)$ are defined as:

$$a + b = (c_a + c_b, \mathbf{s}_a + \mathbf{s}_b, \mathbf{Q}_a + \mathbf{Q}_b) \tag{3}$$

$$a \times b = (c_a c_b, c_b\mathbf{s}_a + c_a\mathbf{s}_b, c_b\mathbf{Q}_a + c_a\mathbf{Q}_b + \mathbf{s}_a\mathbf{s}_b^T + \mathbf{s}_b\mathbf{s}_a^T) \tag{4}$$

All rules for combining annotations remain the same.

Finally, computing $\mathbf{X}^T\mathbf{X}$ is reduced to executing a single join-aggregation query $\gamma(\mathbf{R}_1 \bowtie \ldots \bowtie \mathbf{R}_k)$, where aggregation can be pushed down before join as discussed before.

4.1.3 Factorized Model Evaluation. Given the model parameters θ, the semi-ring annotations of the validation table also help accelerate cross-validation. Suppose the evaluation metric is the squared loss: $\sum(y-\theta x)^2 = \sum(y^2 - 2\theta xy + \theta^2x^2) = \sum y^2 - 2\theta\sum xy + \theta^2\sum x^2$. This expression decomposes into the sums of pairwise products, which are readily available in the gram matrix semi-ring annotations.

4.2 Factorized Data Augmentation

Although factorized learning quickly trains a single model in one query, augmentation search still needs to execute $O(KU(N+M))$ training queries, where there are U user requests, each final augmentation plan contains K augmentations, and the data discovery service respectively returns M and N horizontal and vertical augmentation candidates. For instance, for an AutoML service with $U = 1000$ user requests over $M+N = 500$ augmentation candidates and $K = 3$ average augmentations per plan, the total number of training queries to execute is around $1000 \cdot 3 \cdot 500 = 1.5$ million. Kitana borrows ideas from factorized IVM [48] and view materialization for factorized queries [8] to aggressively pre-compute and share aggregates between these queries. Since each search iteration evaluates every horizontal and vertical candidate, we will first describe optimizations for individual augmentations, and then describe sharing across iterations.

4.2.1 Horizontal Augmentation. Given the current augmentation plan $\mathcal{P}$, horizontal augmentation Q_k^h will union the plan with the candidate dataset D_k^h. We can use IVM to push the aggregation through union [48].

$$Q_k^h = \gamma(\mathcal{P}(T) \cup D_k^h) = \gamma(\mathcal{P}(T)) \cup \gamma(D_k^h)$$

Example 2. *Suppose we union $\mathbf{R}_1$ and $\mathbf{R}_2$ (Figure 3(a,b)). The gram matrix of the union is defined as $\gamma(\mathbf{R}_1 \cup \mathbf{R}_2)$ (Figure 3j), which is equivalent to $\gamma(\mathbf{R}_1) \cup \gamma(\mathbf{R}_2)$ (Figure 3(e,f)).*

To fit the linear regression model using the gram matrix (Figure 3j), we treat B as feature and C as the target variable. $\theta = [\theta_B, \theta_0]$ is then:

$$\theta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T Y = \begin{bmatrix}\sum B^2 & \sum B \\ \sum B & \sum 1\end{bmatrix}^{-1}\begin{bmatrix}\sum BC \\ \sum C\end{bmatrix} = \begin{bmatrix}39 & 11 \\ 11 & 4\end{bmatrix}^{-1}\begin{bmatrix}40 \\ 14\end{bmatrix}$$

The key optimization is to pre-compute $\gamma(\mathcal{P}(T))$ at the start of the search iteration, and $\gamma(D_k^h)$ when data providers upload the dataset. $\gamma(\mathcal{P}(T))$ is shared across all M candidate horizontal augmentations, and $\gamma(D_k^h)$ is shared across U user requests where D_k is a horizontal candidate. Now, horizontal augmentation simply adds the pre-computed aggregates in near-constant time.

4.2.2 Vertical Augmentation. Vertical augmentation is more complex than horizontal augmentation because pushing aggregation through the join needs to take the join key j into account (so the join can be evaluated). Consider Q_k^v, which augments the current augmentation plan $\mathcal{P}$ with D_k^V using join key j[3]:

$$Q_k^v = \gamma(\mathcal{P}(T) \bowtie_j D_k^v) = \gamma(\gamma_j(\mathcal{P}(T)) \bowtie_j \gamma_j(D_k^v))$$

Example 3. *Suppose we have already horizontally augmented $\mathbf{R}_1$ with $\mathbf{R}_2$, and want to assess the vertical augmentation $(\mathbf{R}_3, A)$. To do so, we want to compute $\gamma((\mathbf{R}_1 \cup \mathbf{R}_2) \bowtie_A \mathbf{R}_3)$ (Figure 3k). We can push down the aggregation to derive $\gamma((\gamma_A(\mathbf{R}_1) \cup \gamma_A(\mathbf{R}_2)) \bowtie_A \gamma_A(\mathbf{R}_3))$ (Figure 3(g,h,i)).*

$\gamma_j(\mathcal{P}(T))$ is shared among all N vertical augmentation candidates with join key j. Thus, Kitana pre-computes $\gamma_{j'}(\mathcal{P}(T))$ for all of its valid join keys j'. Kitana also pre-computes $\gamma_{j'}(D_k^v)$ for all of its valid join keys, and shares them across all U requests where D_k^v is a vertical candidate. Vertical augmentation is now independent of the dataset size and bound by the cardinality of j (typically small).

[3]Note for Left Join: When the join keys from vertical augmentations have missing values, performing an inner join will remove tuples and skew training data. Following prior works [28] Kitana performs a left join between the user and augmentation datasets such that the tuples in the user dataset remain without reducing the cardinality. To impute missing values in the left join result, Kitana uses the rules in Section 5.1.2 and computes annotations for imputed values. Kitana does not impute missing values when sending the materialized augmented relation to AutoML, because AutoML systems search for the best imputation method during hyper-parameter optimization [36].

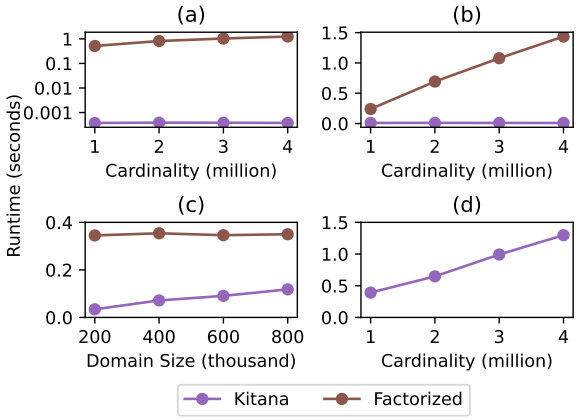


Figure 4: Factorized Augmentation Benchmarks. (a) Horizontal Augmentation Runtime (log) when varying augmentation dataset cardinality. (b) Vertical Augmentation Runtime when varying user dataset cardinality. (c) Vertical Augmentation Runtime when varying join key domain size. (d) Offline Pre-computation Runtime when varying the cardinality.

4.2.3 Sharing Between Augmentation Plans. Once Kitana finds the best vertical augmentation (D^*, j^*) at the end of a search iteration, the next plan is defined as $\mathcal{P}'(T) = \mathcal{P}(T) \bowtie_{j^*} \gamma_{j^*}(D^*)$. At the start of the next iteration, Kitana will pre-compute aggregations $\gamma_j(\mathcal{P}'(T))$ for all valid join keys j. Pre-computing $\gamma_j(\mathcal{P}'(T))$ can re-use aggregations from previous plans. We illustrate this re-use opportunity using a special case where $j \in S_{D^*} \wedge j \notin S_{\mathcal{P}(T)}$:

$$\gamma_j(\mathcal{P}'(T)) = \gamma_j(\mathcal{P}(T) \bowtie_{j^*} D^*) = \gamma_j(\gamma_{j^*}(\mathcal{P}(T)) \bowtie_{j^*} \gamma_{j^*,j}(D^*))$$

Here, $\gamma_{j^*}(\mathcal{P}_{i,j}(T_i))$ has already been computed in a previous search iteration. In general, Kitana aggressively pushes aggregations through joins which reduces the join cost and canonicalizes the plans; it then finds subsets of the query plan that is identical to subsets from a previous iteration (as in the example) and re-uses them.

4.3 Factorized Augmentation Benchmarks

Does pre-computation help, or simply add additional overhead? We now benchmark horizontal and vertical augmentation to evaluate our pre-computation optimizations (Kitana) against naive factorized learning (Factorized). Specifically, we evaluate 1) what factors influence the performance of Kitana? and 2) what are the costs of offline pre-computation?

Dataset. We generate a synthetic dataset $T[f_1, f_2, f_3, Y, j]$ with three features, a Y variable, and a join key with a default domain size of 30. The feature and Y values are randomly generated. $|T| = 1M$ tuples by default. We will horizontally and vertically augment T.

4.3.1 Horizontal Augmentation Performance. We create augmentation dataset D^h using the above procedure, vary $|D^h| \in [1M, 4M]$ tuples, and measure the runtime to compute $\gamma(T \cup D^h)$. Figure 4(a) shows that Kitana is >3 orders of magnitude faster than Factorized. At $|D^h| = 4M$, Factorized takes ~1.2s while Kitana takes ~400μs. Factorized grows linearly in relation cardinality as it needs to compute the aggregates online, while Kitana is constant.

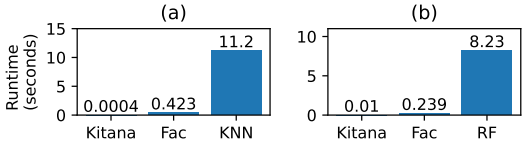


Figure 5: Factorized Augmentation Compared to models from previous works. Fac is factorized learning without pre-computations. (a) Horizontal Augmentation Runtime. (b) Vertical Augmentation Runtime.

4.3.2 Vertical Augmentation Performance. We create augmentation dataset $D^v[j, f]$ containing the join key j and one feature f. The performance of Kitana is bounded by the join key domain size and is independent of the cardinality of the query dataset. Figure 4(b) varies $|D^v| \in [1M, 4M]$ but fixes $|j| = 30$, and we see that Kitana is constant while Factorized grows linearly. Figure 4(c) fixes $|D^v| = 1M$ but varies $|j| \in [200k, 400, 600, 800k]$. Kitana grows linearly because the aggregates are larger. However, Kitana still avoids recomputing aggregates from scratch, and is ultimately upper-bounded by Factorized.

4.3.3 Sharing Between Augmentation Plans. We create three datasets $R_1[A, B, Y], R_2[B, C, f_1], R_3[C, D, f_2]$ with two features, a Y variable, four join keys A, B, C, D with domain size $200k$, and each with $1M$ tuples. We consider original plan $\mathcal{P} = R_1 \bowtie_B R_2$ and new plan $\mathcal{P}' = R_1 \bowtie_B R_2 \bowtie_C R_3$. The task is to compute $\gamma_j(\mathcal{P}')$ for $j \in \{A, B, C, D\}$. Without pre-computation, the four aggregates take 3.13s to complete. With aggregates of original plan $\gamma_j(\mathcal{P})$ for $j \in \{A, B, C\}$ pre-computed and re-used, we can compute the aggregates for $\mathcal{P}'$ in 1.75s, thus 1.8× speedup.

4.3.4 Offline Pre-computation Overhead. Kitana's pre-computation essentially shifts the cost for Factorized to offline rather than during augmentation search. Figure 4(d) reports pre-computation time while varying $|D^v| \in [1M, 4M]$. It grows linearly with $|D^v|$, but is only ~1.3s for $4M$ tuples. This is because the gram-matrix semi-ring decomposes into sum-aggregation operations that are optimized in analytical DBMSes and data science libraries like numpy [38]. Finally, notice that pre-computation is run once per dataset, and can be delegated to the data provider by extending Kitana.*upload*(D, l).

4.3.5 Comparison against Other Models. Although Kitana improves considerably over naive factorized learning, prior data augmentation works don't apply factorized learning and are even slower. To highlight the performance disparities, we now compare Kitana, factorized learning (Fac), and prior augmentation techniques.

We consider previous models for both horizontal and vertical augmentations. For horizontal augmentations, Li et al. [47] computes novelty by training a 3-NN regressor (KNN) for each augmentation candidate. We test the training time on a horizontal augmentation candidate with $|D^h| = 1M$. Figure 5(a) shows that Kitana is four orders of magnitude faster. For vertical augmentation, ARDA [28] joins all tables at once, injects random control features, and trains a random forest to perform feature selection. We generate $|D^v|$ following Section 4.3.2 (without feature injection), and limit ARDA's random forest to a max depth of 3 and 0.1 sampling rate. Figure 5(b) shows that Kitana is three orders of magnitude faster.

In conclusion, prior works incur unacceptably high search costs that will consume the user's budget. Kitana uses factorized learning and prudent pre-computation to accelerate augmentation search by 2-4 orders of magnitude, at a level that makes data augmentation search for AutoML realistic.

5 KITANA DESIGN AND IMPLEMENTATION

This section describes each component in Kitana's architecture (Figure 2). We will present the offline and then online phases.

5.1 Offline Phase

While offline, Kitana collects a large volume of datasets for augmentations, preprocesses them, and builds the necessary indexes to serve requests efficiently during the online phase.

5.1.1 Data Collection. Enterprise users can make the best of datasets already in the data warehouses by uploading them to Kitana with different access labels. They can also upload additional datasets bought from data markets [4, 21], or use existing web crawlers [40] to collect public dataset [9, 14, 18] with *RAW* access labels.

5.1.2 Data Preprocessing. Kitana preprocesses data corpus offline for efficient online *Data Discovery* and *Factorized Learning*.

Feature Engineering. Kitana performs standard feature engineering [45], including missing values imputation[4], feature transformations (e.g., ImageNet [44] to featurize images, and BERT [30] to featurize text) and standardization (center and re-scale numeric attributes).

Data Discovery. Data discovery finds augmentation candidates for a given table. It does so by computing a profile (e.g., minhash and other statistics) for each table and building an index over them. Kitana uses Aurum [32] by default, which builds a *discovery index* offline. The index finds union-able tables based on syntactic schema matching and value similarity, and join-able tables based on a combination of similarity, containment, and semantic similarity [33] metrics. Kitana can use other data discovery systems [24, 26, 63] as well, and they build similar types of indexes offline.

Gram Matrix. To support efficient online factorized learning, Kitana pre-computes aggregated gram-matrix for each dataset D in the corpus as discussed in Section 4.2; the aggregates include $\gamma(D)$ and $\gamma_j(D)$ for each valid join key j. Building the gram-matrix has low overhead as shown in Section 4.3, and can potentially be offloaded to the data providers during dataset upload.

Re-weighting. For vertical augmentation, one-to-many or many-to-many join results will have a much larger number of tuples and potentially skew the training data distribution. To avoid this, we re-weight the semi-ring such that, for each join key value, its semi-ring has a count of 1. Given the aggregated gram-matrix semi-ring (c, s, Q) for each join key, we multiply it with $(1/c, 0, 0)$ to $(1, s/c, Q/c)$.

5.1.3 Handling Updates. Kitana efficiently supports deletes, inserts, and updates of columns and rows to datasets in the corpus. First, the data discovery system we use supports incremental maintenance, as discussed in [32]. Second, we use factorized IVM [48] to update the semi-rings incrementally.

[4]In addition to dealing with missing values offline, Kitana also applies the missing value imputation methods during online vertical augmentations on tables with join keys containing missing values. This is because Kitana uses a left join to avoid reducing the cardinality of the output join, which can introduce nulls.

5.2 Online Phase

During the online phase, Kitana preprocesses the user's base table and checks the request cache in case it can save computation. Then it triggers the augmentation process described in 3.2, while balancing the budget usage. It uses an AutoML service to find a good model and finally it constructs a response to the user.

5.2.1 Request Preprocessing. Given the user's training dataset T, Kitana computes a profile and uses it to probe the discovery index to find augmentation candidates (L6). Kitana also splits T into 10-folds for cross-validation. At the beginning of each iteration (L4-18), Kitana pre-computes $\gamma(\mathcal{P}(T))$ and $\gamma_j(\mathcal{P}(T))$ for all valid join keys to share computations (Section 4.2).

5.2.2 Request Cache. Caching augmentation plans helps when the training data in two requests share the same schema (L2, Algorithm 1). We use the most recent plan that improves the cross-validation accuracy by $\geq\delta$, where δ is the parameter used for early stopping in Algorithm 1 (0.02 by default). We design a two-level *Request Cache*: each schema stores a list of K plans ($K = 1$ in the experiments). The list is ordered and replaced using LRU; a plan is considered used if it improved the model by $\geq\delta$.

5.2.3 Cost Model. Kitana shortcuts to AutoML once there is not "enough time" left (L16, Algorithm 1), and predicted by a cost model. The cost model $f(n, m) \rightarrow \mathbb{R}^+$ maps the shape $n \times m$ of the augmented training dataset T' as input, and outputs the time to train user-requested model K times ($K = 5$ by default). The function should over-predict to ensure Kitana is not worse than AutoML without augmentation search, and is typically tuned to specific hardware and model types.

Since Kitana uses Auto-Sklearn [36] as its default AutoML library, we use scitime [20] to construct the cost model. Scitime is an open-sourced project for training time estimation for Sklearn algorithms given training data shape; we found it accurate in our experimental evaluation. ML cost estimation is an active area of research [50, 61], and we expect cost estimators to improve over time.

5.2.4 Prediction API Construction. Kitana constructs and exposes a prediction REST API if the request R includes *API* in the return labels. The API takes as input a dataset T' with the same schema as T, and outputs the prediction values for each record. Internally, it applies all vertical augmentations from $\mathcal{P}^*$ to T', and then uses the final model θ to perform the predictions. If Kitana internally uses a cloud AutoML service rather than Auto-Sklearn, then Kitana will call the service's own prediction API. Note that this will incur monetary charges that the user is responsible for.

6 EVALUATION

Our evaluation strives to understand three main questions: **Q1**: Can Kitana improve task performance beyond real-world AutoML services? **Q2**: How does Kitana adapt to varying budgets? **Q3**: how sensitive is Kitana performance to different components and their configurations?

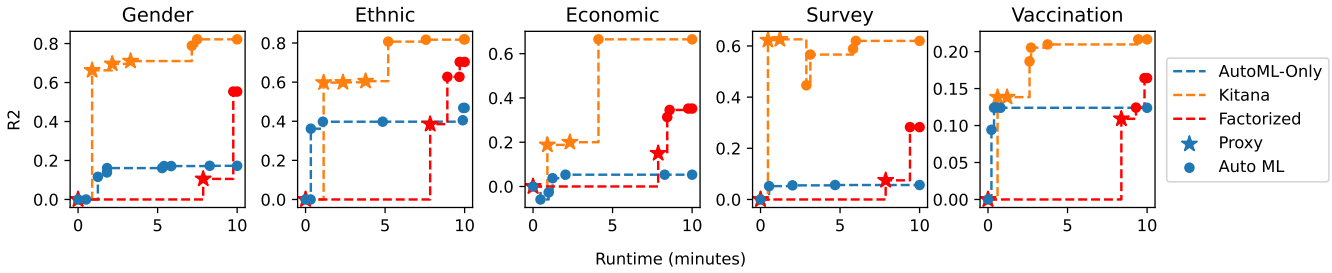


Figure 6: Baseline performance (test R2) over time. ★ is for proxy model and • is for AutoML model performance.

6.1 Setup

Datasets. We collected 513 datasets from NYC open data [18] (361 datasets, 3.1 GB) and CMS Data [9] (152 datasets, 7.4 GB). We then created 5 user requests based on the following datasets; for each, we split the dataset into 80% training data and 20% data.

- **Gender** [1] from NYC open data, which contains 4.1 MB of Early Learning Assessment (ELA) data from 2013-2016, including gender. The regression task predicts the `Mean Scaled Score` (test scores).
- **Ethnic** [2] from NYC open data. Contains 6.9 MB of ELA data from 2013-2017, including ethnicity. The regression task predicts the `Mean Scale Score`.
- **Economic** [11] from NYC open data. Contains 4.2 MB of data from 2013-2017. It contains school math results and socio-economic factors. The task classifies individuals as `Econ Disadv` or `Not Econ Disadv`.
- **Survey** [22] from CMS data. Contains 10MB of nursing home health data over three years. The regression task predicts the `Total Number of Health Deficiencies`.
- **Vaccination** [12] from CMS data. Contains 867KB of COVID-19 Vaccination Rate data. The regression task predicts the `Percent of Vaccinated Residents`.

Baselines:[5]

- AutoML-Only directly sends the request to an AutoML service. Although Kitana supports any AutoML service, our evaluation primarily uses the popular open-source library Auto-sklearn [36]. Section 6.2.2 also evaluates Google's Vertex AI.
- Factorized is similar to Kitana, but does not pre-compute sketches. It computes the aggregates online and fully re-trains the *factorized proxy model* to assess each candidate augmentation plan.
- Kitana balances the budget between augmentation and AutoML, and applies our full suite of optimizations to speed up the augmentation search.

Setup. All experiments run in a single thread on a GCP n1-standard-16 VM, running Debian 10, Xeon 2.20GHz CPU, and 60GB RAM. We implemented Kitana in Python 3. To compute monetary costs, we report computation costs because they dwarf storage costs. For instance, Google Cloud storage costs $0.046/GB/month, while running an n1-standard-16 instance costs about $388.36/month.

[5]We also tried other baselines like ARDA [28]. However, ARDA's data augmentation search process fails to finish within the budget; thus it doesn't output any model.

6.2 Q1: Does Kitana use Data Effectively?

We first evaluate the three baselines using all five user requests. The AutoML service incrementally adds models to an ensemble, and thus periodically returns updated model parameters and accuracy whenever a new model is added. Thus, for each request we report the R2 score over time. For Kitana, we also report the proxy model in each iteration during the augmentation search. Section 6.2.1 reports results using Auto-Sklearn, and Section 6.2.2 then evaluates different AutoML services by (monetary) cost.

6.2.1 Main Results. Figure 6 plots model quality (y-axis) by runtime (x-axis) for different baselines (color). The star and circle marks correspond to the proxy and AutoML models. In general, R2 increases over time, and Kitana dominates both naive and AutoML on both runtime and R2. For **Gender**, Kitana reaches an accuracy of > 0.6 in less than 1 minute, whereas AutoML-only never exceeds 0.2 even after 10 minutes. These results highlight the importance of high-quality features over pure compute [28, 55], as well as the competitive performance of linear regression.

Zooming in, Factorized ultimately outperforms AutoML-Only, but takes a long time to find a helpful augmentation. This leaves less time to search for a good model during the AutoML phase, which explains its lower R2 scores compared to Kitana. In general, with a large budget, we expect Factorized to have a similar curve as Kitana, but shifted towards the right.

The drop in **Survey** occurs when switching from the proxy model to AutoML because AutoML starts training from scratch and ultimately does not perform better than the proxy. The same patterns, though magnified, occur when using Google's VertexAI.

6.2.2 Effect of AutoML System. We now directly compare Kitana with existing AutoML services on price and model accuracy, and show that Kitana can 1) reach the same or greater model accuracy for considerably lower cost, 2) reach far higher model accuracy for the same cost, and 3) often times reach a higher accuracy at a lower cost. In many cases, the proxy model alone is sufficient to outperform AutoML services.

Setup. To do so, we evaluate Auto-Sklearn (SK) and Google's Vertex AI [16]. In Vertex AI, users can specify a time budget of at least 1 hour, so we chose a 1-hour budget. Unlike Auto-Sklearn, Vertex AI may exceed the requested budget. Further, it does not report model performance for data over time, so we only report the final model performance.

We evaluate each AutoML service in isolation and as part of Kitana. Since SK is not hosted, we evaluate SK and SK+Kitana on a

		SK (-Only)		SK + Kitana		Vertex AI (-Only)			Vertex AI + Kitana		
Data	**Metric**	**Score**	**Cost**	**Score**	**Cost**	**Score**	**Time**	**Cost**	**Score**	**Time**	**Cost**
Gender	R2	0.17	$0.13	0.82	$0.13	0.16	109	$38.61	**0.91**	118	$41.80
Ethnic	R2	0.39	$0.13	0.83	$0.13	0.38	133	$47.11	**0.85**	121	$42.86
Survey	R2	0.06	$0.13	**0.64**	$0.13	0.16	113	$40.02	**0.64**	130	$46.05
Vaccination	R2	0.12	$0.13	**0.22**	$0.13	0.11	116	$41.09	0.20	115	$40.73
Economic	Accuracy	0.77	$0.13	0.92	$0.13	0.78	118	$41.80	**0.99**	118	$41.80

Table 1: Comparison of Auto-Sklearn (SK) and Vertex AI with and without Kitana. Vertex AI's budget is set to 1 hour (the minimum allowed by the service) while SK is set of 10 minutes (sufficient to converge). Kitana dominates both AutoML services.

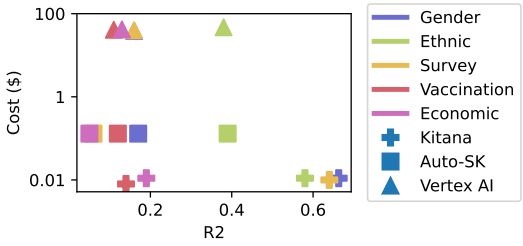


Figure 7: Cost for Kitana to reach or exceed the model quality of Auto-Sklearn and Vertx AI for different user requests (color). Kitana trains higher quality models at ~10× to >100× less cost compared to AutoSklean and Vertex AI, respectively.

GCP n1-standard-16 VM that costs $0.13 for 10 minutes. Although Auto-Sklearn can continue running for a long time, we find that the model converges by 10 minutes, so we terminate it after 10 minutes—this is why SK and SK+Kitana costs are fixed.

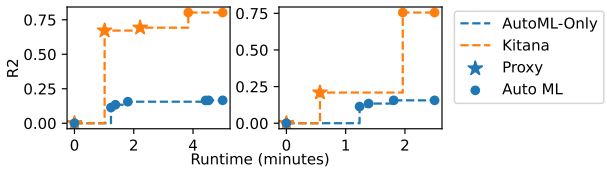
Results. Table 1 summarizes the model accuracy, monetary cost, and run time in minutes for VertexAI; we highlight the highest model accuracies in **red**. These results fix the time budget (though Vertex AI exceeds the 1 hour minimum budget), and focus on model accuracy and cost. When looking at AutoML in isolation, Vertex AI takes an order of magnitude longer to run and results in comparable model accuracies as Auto-Sklearn. Adding data augmentation (Kitana) improves the model quality by a large margin (e.g., 0.16 → 0.91 for Gender) across all user requests.

Although existing AutoML services focus solely on compute, Kitana provides an opportunity to balance data augmentation and AutoML. Figure 7 reports the cost for Kitana to achieve the same or higher model quality as SK and VertexAI (lower right is better). In the **Gender** dataset, current users simply choose between e.g., Auto-Sklearn ($0.13, $R2 = 0.17$) or Vertex AI ($38.61, $R2 = 0.16$). Instead, Kitana merely costs $0.01 to reach $R2 = 0.66$; in fact, all Kitana points are based on the proxy model alone.

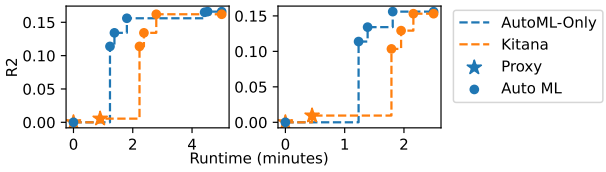
6.3 Q2: Kitana Adaptability

Our previous experiments used a generous time budget and a corpus that contains good augmentation candidates. We now study how Kitana adapts under varying time budgets, and when there are no good augmentation opportunities.

6.3.1 Varying Budget. We repeat the experiment from Section 6.2 but reduce the budget by 2× (5 minutes) and 4× (2.5 minutes). For space limits, we report results for **Gender**, but the other requests have similar results.



(a) (left) 5min and (right) 2.5min budgets.



(b) Kitana when the corpus lacks good augmentations.

Figure 8: Kitana adapts to low budgets by choosing cheap but effective augmentations. When Kitana lacks predictive datasets, it early stops and achieves similar performance.

Figure 8a shows that Kitana adapts to shorter budgets by finding augmentations that require less AutoML training time. The stars refer to the proxy model, so we see that the 5-minute budget lets Kitana run two iterations of augmentation search to find a good augmentation. With only 2.5 minutes, Kitana terminates the search ~30s into the first iteration and picks the best plan so far. In both cases, the proxy model is of higher quality than the result for AutoML-Only. A more accurate cost model can potentially allow Kitana to try more augmentation candidates; however, we find that the simple cost model works well in practice.

6.3.2 Kitana Without Predictive Augmentations. What if there are no augmentation opportunities? Since the time spent in the augmentation search is wasted, how much worse will Kitana be, especially when the budget is tight? Simply deleting the predictive datasets from the corpus will not work because the data discovery component will simply not return any results, and Kitana will skip the augmentation search altogether. Thus, we instead create false positive augmentations by replacing all the candidate datasets' features with random values. The setup is otherwise the same as the preceding one.

Figure 8b shows that Kitana follows the same curve as AutoML, but shifted to the right by 1min (for 5-minute budget) and ~30s (for 2.5-minute budget). The final model quality is nearly identical (R2 differs by <0.005) for both budgets because Kitana early-terminates in order to provide AutoML ample time to train.

Metric	Time	Training R2	Testing R2
Kitana	0.01s	0.995	0.994
Novelty	9.72s	0.773	-0.232

Table 2: Kitana Performance for horizontal augmentation with different metrics.

6.4 Q3: Component Sensitivity

We now evaluate Kitana when restricted to horizontal augmentation only against recent data augmentation work by Li et al. [47], and also evaluate the effects of request caching across a sequence of user requests.

6.4.1 Horizontal Augmentation Comparisons. Recent works, such as the Acquisition algorithm by Li et al. [47], also study the problem of horizontal augmentation. At a high level, given a corpus of datasets, Acquisition seeks sample "novel" records from datasets, based on a custom "Novelty" metric. Novelty is estimated by unioning subsets of the user and augmentation candidates, and fitting a 3-NN classifier to predict which dataset a given record is from. The intuition is that novel data is easier to classify.

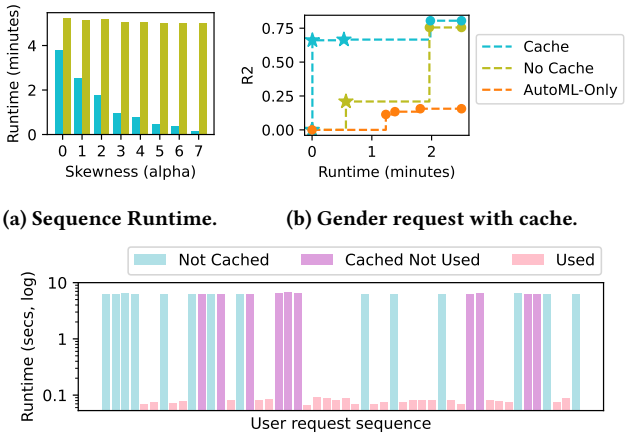
The key limitation of the novelty measure is that it is oblivious to the actual prediction task, as defined by the test distribution. If the data corpus offers union-compatible data that is irrelevant to the test distribution (common for the data lake in the wild), this data will have high novelty simply if they are dissimilar to the training data. In contrast, Kitana directly evaluates augmentations with respect to the training data through cross validation, which is expected to be representative of the test data.

Setup. We compare Kitana with a variation that replaces Lines 13-14 in Algorithm 1 to rank augmentation candidates using Novelty. This also means that augmentation search does not benefit from factorized learning. We measure the time to search for the top one horizontal augmentation candidates, and the model accuracy after running AutoML for 2 minutes over the augmented dataset. Following Li et al. [47], we use the **RoadNet** [3] mapping dataset (434,874 tuples) with schema `(lat, lon, altitude)`, and partition it along `lat` and `lon` into a 8×8 grid. Each partition is a candidate horizontal augmentation. The regression task uses `lat`, `lon` to predict `altitude`, and the user's training T and testing V datasets are both 0.5% samples from partition 1. In this way, most horizontal augmentations are dissimilar but irrelevant.

Acquisition uses a reinforcement learning framework to estimate a dataset's novelty because there is a cost to sample from each augmentation candidate. In our case, we directly evaluate the true novelty in order to study the upper bound of their approach.

Results. Table 2 reports that Kitana is orders of magnitude faster than Novelty because the latter trains a classification model for each augmentation but does not benefit from factorized learning. Although both approaches have a high R2 on the training data, Novelty prefers augmentations *dissimilar to* partition 1, which skews the augmented training data and leads to a low testing R2 score.

6.4.2 Request Cache. Our experiments so far focused on a single user request, however an AutoML service is expected to serve many requests over time, from the same or different users that perform the same or different tasks. How much does the request take advantage of the similarity between multiple user requests? Further, what is the caching overhead when user requests are mostly dissimilar?



(a) Sequence Runtime. **(b) Gender request with cache.**

(c) Cache performance (log) zoom in for alpha = 1.

Figure 9: Request Cache Experiment Result. The runtime of a sequence of user requests has been reduced because of the cache. The reduction in run-time can improve final model performance as Kitana has more budgets.

Setup. We create a synthetic benchmark consisting of 20 users and 300 candidate vertical augmentations per user, for a total of 6000 augmentation candidates. Each user dataset requires 2 augmentations to train a perfect proxy model (R2=1). To test the case of failed cached augmentation plans, we group the users into 10 pairs and assign each pair a unique schema. Thus, each pair of users (A, B) will match each other's cached augmentation plans, but A's plan will not improve B's model and vice versa.

We generate sequences of 50 user requests, where the i^{th} user is drawn from a Zipfian distribution over the 20 users. We vary the parameter $\alpha \in [0, 7]$; 0 is uniform, while 7 is heavily skewed. To pressure test the request cache, we limit the cache size to accommodate 5 schemas and 1 plan per schema.

Results. Figure 9a reports the runtime for all 50 requests as the Zipfian skew increases. Compared with no caching, the cache reduces the runtime by 26.8% given a uniform distribution ($\alpha = 0$) and by 96.7% when the requests are nearly all the same ($\alpha = 7$).

Figure 9c examines the per-request runtimes when $\alpha = 1$. We see that cache hits reduce the request runtime by over 100×, and also that evaluating unhelpful cached plans (purple) incurs negligible overhead (~1%) as compared to a cache miss (blue).

A cache hit both reduces the augmentation search time, and gives AutoML more time to find a better model. To see this, we first cache the augmentation plan generated by the **Gender** user request from Section 6.3 that has a 5-minute budget. We then generate a similar request with a resampled training dataset and a reduced budget of 2.5 minutes. Figure 9b shows that the cached plan immediately increases the proxy model's quality to an R2 of ~0.7. This leads to a better final augmentation plan, and ultimately a higher quality model as compared to a no-cache setting (~0.75 → ~0.8).

7 RELATED WORK

We now distinguish Kitana from related work.

AutoML: AutoML services enable users with limited ML backgrounds to train high-quality models. AutoML is time-critical as it charges on an hourly basis (e.g., VertexAI charges $21.252/hr). Existing AutoML services [5, 6, 16, 17] are model-centric: all the user budgets are spent on training and searching for the models, however they are only as good as their initial training dataset. Kitana provides an API-compatible interface as existing AutoML services but is data-centric by allocating part of the user budgets to search for predictive data augmentations. Our experiments show that Kitana can be orders of magnitude more cost-effective than existing AutoML services, and achieve almost the same result in the worst case (when there is no helpful data augmentation).

Model Augmentation: Recent works propose horizontal or vertical augmentation techniques to improve machine learning models. For instance, Li et al. [47] focuses on horizontal augmentation. They sample records from a set of relations based on a "Novelty" measure, assuming that record diversity will improve model accuracy, which may not hold for the data lake in the wild. Our experiments show that the model accuracy can decrease after augmentation.

Other works such as ARDA [28] apply specialized feature selection techniques that require iterative training of complex models (e.g., random forest). These techniques are ill-suitable for time-critical tasks such as AutoML: they take orders of magnitudes more time than Kitana, and even at such high costs, they empirically don't show better results than Kitana (Figure 1).

Data Discovery: Nowadays data warehouses and data lakes leverage data discovery [32] or catalogs [7, 10, 13, 14] systems to find candidate datasets or sellers. Data discovery search interfaces typically take natural language, datasets, or schemas as input. Kitana uses existing data discovery systems [32] to identify vertical and horizontal augmentation candidates and is agnostic to the specific discovery mechanism.

Data Marketplaces: With the unprecedentedly large volume of datasets available, there is a rising interest in the data marketplaces [4, 21]. However, current data marketplaces release raw data directly, which involves a manual assessment process. When there is a potential buyer of the data products, sellers need to, e.g., contact the buyer, send a sample of data to the user for assessment, and engage in lengthy price negotiations [41]. To solve this, many works have proposed theoretical models of data markets for ML tasks, such as [23]. Unlike us, their contribution is in pricing data to avoid strategic behavior and allocate revenue to data providers, and consequently, they must resort to simplifications in their model, such as supporting only one buyer.

Views are commonly used for access control [54]. They prevent access to the underlying data and only expose query results to the user. Since ML models can be expressed as aggregation queries, Kitana uses "ML Model" views to protect seller datasets from release.

Differentially private queries [51] add noise to aggregated statistics to provide plausible deniability for individual input records. These techniques are compatible with Kitana—noise can be added to the final model parameters, to the predictions, or potentially even by sellers before uploading their data to the service.

8 TOWARDS DATA MARKETS

A data market is a service that matches data users with datasets offered by providers [34, 41, 58, 59]. This type of market may be within an organization (e.g., data scientists search for datasets provided by other teams that improve their BI or ML tasks), or between organizations. For example, John Deere vehicles generate agricultural sensor data that can benefit crop health monitoring and supply-chain optimization [25]. Financial institutions can monetize access to data to help others improve their risk assessments and their decision-making models [15]. Doctors want to collect radiology or genomic data about patients in other departments or hospitals that have similar symptoms as current patients [52].

Unfortunately, data markets are stymied by the simple fact that matching users with useful datasets—arguably its core functionality—is non-trivial. Existing data marketplaces such as AWS Data Marketplaces [4], Snowflake's [21], or Narrative simply index metadata about datasets, and the burden of actually assessing the value of a candidate dataset is on the user. In many cases, the market simply returns a reference to a data provider, and the user needs to contact the provider, ask for a sample to assess, and negotiate a contract [41]. For data markets to be effective, this matching process must be simple, cheap, fast, and provide predictably good matches.

Kitana provides such an efficient matching process for ML training tasks, and we believe it can bootstrap a useful data market within organizations by reducing the barriers on the "demand side" (data users) and encouraging participation from the "supply side" (data providers). On the demand side, consider the following:

Example 4. *An online real estate company has teams for different US regions, as well as for different types of data—rental listings, recent home purchases, neighborhood demographics and schooling, local business listings, and more. A data scientist has created a Jupyter notebook, and is building a model to predict a neighborhood's safety score. She is primarily using local police records for theft and misdemeanors. A notebook plugin recognizes that she is trying to build a model, and constructs a Kitana request. She then sees recommendations to use records from three nearby police stations, and features from local school quality datasets. She clicks "use" to automatically augment her training data, and sees substantial model improvement.*

Kitana also benefits the supply side in two ways. First, demand-side tasks become benchmarks that inform data producers and integration. Second, it paves the way for data-rich organizations to monetize data access. For example, they could act as a Kitana data provider, and recieve compensation if their data is used to augment a request. Evaluating the plethora of economic mechanisms to match supply and demand is an exciting direction for future work.

9 CONCLUSIONS

This paper presented Kitana, an AutoML system with practical data augmentation search. To identify augmentation opportunities efficiently, Kitana leverages factorized learning with aggressive pre-computations, and a cost model for balancing the search cost and model training. Our experimental evaluation shows Kitana finds considerably more accurate models in orders of magnitude less time than SOTA open-source and commercial AutoML services.

REFERENCES

[1] 2022. 2013-16 School ELA Data Files By Grade - Gender. https://data.cityofnewyork.us/Education/2013-16-School-ELA-Data-Files-By-Grade-Gender/436j-ja87.

[2] 2022. 2013-2017 School ELA Results - Ethnic. https://data.cityofnewyork.us/Education/2013-2017-School-ELA-Results-Ethnic/ynau-kwze.

[3] 2022. 3D Road Network (North Jutland, Denmark) Data Set. https://networkrepository.com/3D-spatial-network.php.

[4] 2022. Amazon Marketplace. https://aws.amazon.com/marketplace.

[5] 2022. Amazon SageMaker. https://aws.amazon.com/sagemaker/.

[6] 2022. Azure: Automated Machine Learning. https://docs.microsoft.com/en-us/azure/machine-learning/concept-automated-ml.

[7] 2022. Azure: Data Catalog. https://azure.microsoft.com/en-us/services/data-catalog/.

[8] 2022. Calibration: A Simple Trick for Fast Analytics over Joins (Technical Report). https://anonymous.4open.science/r/CJT-2DF1/tech_report/Calibrated_Junction_Hypertree.pdf.

[9] 2022. CMS Data. https://data.cms.gov/.

[10] 2022. Collibra Data Intelligence Cloud: Data Catalog. https://www.collibra.com/us/en/platform/data-catalog.

[11] 2022. COVID-19 Vaccination Rates - Provider Data. https://data.cityofnewyork.us/Education/2013-2017-School-Math-Results-Economic/9vgx-wa3i.

[12] 2022. COVID-19 Vaccination Rates - Provider Data. https://data.cms.gov/provider-data/dataset/pvax-cv19.

[13] 2022. Data Catalog: data discovery | Google Cloud. https://cloud.google.com/data-catalog.

[14] 2022. Data.World. https://data.world/.

[15] 2022. Golman Sachs Modeling Platform. https://www.linuxfoundation.org/press-release/goldman-sachs-open-sources-its-data-modeling-platform-through-finos/.

[16] 2022. Google Cloud: Vertex AI. https://cloud.google.com/vertex-ai.

[17] 2022. H2O: Driverless AI Transformations. https://docs.h2o.ai/driverless-ai/latest-stable/docs/userguide/transformations.html.

[18] 2022. NYC Open Data. https://opendata.cityofnewyork.us/.

[19] 2022. Oracle Machine Learning AutoML. https://docs.oracle.com/en/database/oracle/machine-learning/oml-automl-ui/index.html.

[20] 2022. scitime. https://github.com/scitime/scitime.

[21] 2022. Snowflake Data Marketplace. https://www.snowflake.com/data-marketplace/.

[22] 2022. Survey Summary. https://data.cms.gov/provider-data/dataset/tbry-pc2d.

[23] Anish Agarwal, Munther Dahleh, and Tuhin Sarkar. 2019. A marketplace for data: An algorithmic solution. In *Proceedings of the 2019 ACM Conference on Economics and Computation*. 701–726.

[24] Alex Bogatu, Alvaro AA Fernandes, Norman W Paton, and Nikolaos Konstantinou. 2020. Dataset discovery in data lakes. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 709–720.

[25] Kelly Bronson and Irena Knezevic. 2016. Big Data in food and agriculture. *Big Data & Society* 3, 1 (2016), 2053951716648174.

[26] Sonia Castelo, Rémi Rampin, Aécio Santos, Aline Bessa, Fernando Chirigati, and Juliana Freire. 2021. Auctus: a dataset search engine for data discovery and augmentation. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2791–2794.

[27] Bin Chen, Jiarong Hong, and Yadong Wang. 1997. The minimum feature subset selection problem. *Journal of Computer Science and Technology* 12, 2 (1997), 145–153.

[28] Nadiia Chepurko, Ryan Marcus, Emanuel Zgraggen, Raul Castro Fernandez, Tim Kraska, and David Karger. 2020. ARDA: automatic relational data augmentation for machine learning. *arXiv preprint arXiv:2003.09758* (2020).

[29] Ryan Curtin, Benjamin Moseley, Hung Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Rk-means: Fast clustering for relational data. In *International Conference on Artificial Intelligence and Statistics*. PMLR, 2742–2752.

[30] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[31] Paul Dourish, Rebecca E Grinter, Jessica Delgado De La Flor, and Melissa Joseph. 2004. Security in the wild: user strategies for managing security as an everyday, practical problem. *Personal and Ubiquitous Computing* 8, 6 (2004), 391–401.

[32] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A data discovery system. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1001–1012.

[33] Raul Castro Fernandez, Essam Mansour, Abdulhakim A Qahtan, Ahmed Elmagarmid, Ihab Ilyas, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2018. Seeping semantics: Linking datasets using word embeddings for data discovery. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 989–1000.

[34] Raul Castro Fernandez, Pranav Subramaniam, and Michael J. Franklin. 2020. Data Market Platforms: Trading Data Assets to Solve Data Problems. *Proc. VLDB Endow.* 13, 12 (sep 2020), 1933–1947. https://doi.org/10.14778/3407790.3407800

[35] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. 2020. Auto-sklearn 2.0: The next generation. *arXiv preprint arXiv:2007.04074* 24 (2020).

[36] Matthias Feurer, Aaron Klein, Jost Eggensperger, Katharina Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and Robust Automated Machine Learning. In *Advances in Neural Information Processing Systems 28 (2015)*. 2962–2970.

[37] Isabelle Guyon and André Elisseeff. 2003. An introduction to variable and feature selection. *Journal of machine learning research* 3, Mar (2003), 1157–1182.

[38] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[39] Jonathan Huggins, Ryan P Adams, and Tamara Broderick. 2017. PASS-GLM: polynomial approximate sufficient statistics for scalable Bayesian GLM inference. *Advances in Neural Information Processing Systems* 30 (2017).

[40] Md Abu Kausar, VS Dhaka, and Sanjeev Kumar Singh. 2013. Web crawler: a review. *International Journal of Computer Applications* 63, 2 (2013).

[41] Javen Kennedy, Pranav Subramaniam, Sainyam Galhotra, and Raul Castro Fernandez. 2022. Revisiting Online Data Markets in 2022: A Seller and Buyer Perspective. *SIGMOD Rec.* 51, 3 (nov 2022), 30–37. https://doi.org/10.1145/3572751.3572757

[42] Mahmoud Abo Khamis, Ryan R Curtin, Benjamin Moseley, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Functional Aggregate Queries with Additive Inequalities. *ACM Transactions on Database Systems (TODS)* 45, 4 (2020), 1–41.

[43] Mahmoud Abo Khamis, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. AC/DC: in-database learning thunderstruck. In *Proceedings of the second workshop on data management for end-to-end machine learning*. 1–10.

[44] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012).

[45] Max Kuhn and Kjell Johnson. 2019. *Feature engineering and selection: A practical approach for predictive models*. CRC Press.

[46] Mathias Lécuyer, Riley Spahn, Roxana Geambasu, Tzu-Kuo Huang, and Siddhartha Sen. 2017. Pyramid: Enhancing Selectivity in Big Data Protection with Count Featurization. *2017 IEEE Symposium on Security and Privacy (SP)* (2017), 78–95.

[47] Yifan Li, Xiaohui Yu, and Nick Koudas. 2021. Data acquisition for improving machine learning models. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1832–1844.

[48] Milos Nikolic and Dan Olteanu. 2018. Incremental view maintenance with triple lock factorization benefits. In *Proceedings of the 2018 International Conference on Management of Data*. 365–380.

[49] Chong Peng and Qiang Cheng. 2020. Discriminative ridge machine: A classifier for high-dimensional data or imbalanced data. *IEEE Transactions on Neural Networks and Learning Systems* 32, 6 (2020), 2595–2609.

[50] Adrian Daniel Popescu, Andrey Balmin, Vuk Ercegovac, and Anastasia Ailamaki. 2013. Predict: towards predicting the runtime of large scale iterative analytics. *Proceedings of the VLDB Endowment* 6, CONF (2013).

[51] Davide Proserpio, Sharon Goldberg, and Frank McSherry. 2014. Calibrating Data to Sensitivity in Private Data Analysis. *Proc. VLDB Endow.* 7 (2014), 637–648.

[52] Mohammad Rasouli and Michael I Jordan. 2021. Data sharing markets. *arXiv preprint arXiv:2107.08630* (2021).

[53] Payam Refaeilzadeh, Lei Tang, and Huan Liu. 2009. Cross-validation. *Encyclopedia of database systems* 5 (2009), 532–538.

[54] Shariq J. Rizvi, Alberto O. Mendelzon, S. Sudarshan, and Prasan Roy. 2004. Extending query rewriting techniques for fine-grained access control. In *SIGMOD '04*.

[55] Nithya Sambasivan, Shivani Kapania, Hannah Highfill, Diana Akrong, Praveen Paritosh, and Lora M Aroyo. 2021. "Everyone wants to do the model work, not the data work": Data Cascades in High-Stakes AI. In *proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–15.

[56] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q Ngo, and XuanLong Nguyen. 2019. A layered aggregate engine for analytics workloads. In *Proceedings of the 2019 International Conference on Management of Data*. 1642–1659.

[57] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*. 3–18.

[58] Florian Stahl, Fabian Schomm, and Gottfried Vossen. 2012. Marketplaces for Data: An Initial Survey. *ERCIS Working Paper* Working Papers, European Research Center for Information Systems (07 2012). https://doi.org/10.1145/2481528.2481532

[59] Florian Stahl, Fabian Schomm, and Gottfried Vossen. 2014. The data marketplace survey revisited.

[60] Chi Wang, Qingyun Wu, Markus Weimer, and Erkang Zhu. 2021. FLAML: a fast and lightweight AutoML Library. *Proceedings of Machine Learning and Systems* 3 (2021), 434–447.

[61] Kewen Wang and Mohammad Maifi Hasan Khan. 2015. Performance prediction for apache spark platform. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. IEEE, 166–173.

[62] Doris Xin, Hui Miao, Aditya Parameswaran, and Neoklis Polyzotis. 2021. Production machine learning pipelines: Empirical analysis and optimization opportunities. In *Proceedings of the 2021 International Conference on Management of Data*. 2639–2652.

[63] Yi Zhang and Zachary G Ives. 2020. Finding related tables in data lakes for interactive data science. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1951–1966.