Shahin: Faster Algorithms for Generating Explanations for Multiple Predictions

Sona Hasani
Google
USA
sonah@google.com

Saravanan Thirumuruganathan
QCRI, HBKU
Qatar
sthirumuruganathan@hbku.edu.qa

Nick Koudas
University of Toronto
Canada
koudas@cs.toronto.edu

Gautam Das
University of Texas at Arlington
USA
gdas@cse.uta.edu

ABSTRACT

Machine learning (ML) models have achieved widespread adoption in the last few years. Generating concise and accurate explanations often increases user trust and understanding of the model prediction. Usually, the implementations of popular explanation algorithms are highly optimized for a single prediction. In practice, explanations often have to be generated in a batch for *multiple* predictions at a time. To the best of our knowledge, there has been no work for efficiently generating explanations for more than one prediction. While one could use multiple machines to generate explanations in parallel, this approach is sub-optimal as it does not leverage higher-level optimizations that are available in a batch setting. We propose a principled and lightweight approach for identifying redundant computations and several effective heuristics for dramatically speeding up explanation generation. Our techniques are general and could be applied to a wide variety of perturbation based explanation algorithms. We demonstrate this over a diverse set of algorithms including, LIME, Anchor, and SHAP. Our empirical experiments show that our methods impose very little overhead and require minimal modification to the explanation algorithms. They achieve significant speedup over baseline approaches that generate explanations in a sequential manner.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Information systems** → **Query optimization**.

KEYWORDS

explanations; multi-query optimization; machine learning; LIME; Anchor; SHAP

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457332

ACM Reference Format:
Sona Hasani, Saravanan Thirumuruganathan, Nick Koudas, and Gautam Das. 2021. Shahin: Faster Algorithms for Generating Explanations for Multiple Predictions. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3448016.3457332

1 INTRODUCTION

The widespread use of ML models has necessitated the development of algorithms for explaining their predictions. Explanations increase user trust and understanding of the model with diverse applications including model transparency [11, 12, 21], debugging [9], accountability [33], auditing [11, 29], fairness [6], explanation summarization [9, 24] and others [7, 25]. An increasing number of countries espouse a "right to explanation" [11]. However, current algorithms are optimized for explaining *individual* predictions. In applications such as responsible AI [6, 33] or explanations summarization [9, 24], explaining data cleaning [5, 9, 35] there is a need to generate explanations for multiple predictions in a *batch* setting. Generating explanations often cannot be done in real-time (in milliseconds). For example, generating a single explanation using LIME takes 17, 15, 6, 6 and 5 seconds respectively for the 5 datasets evaluated in the paper. So, an organization might pre-compute all the explanations in a batch setting and retrieve them as needed.

Sequentially processing one explanation at a time could take too much time. An alternate approach of using a cluster and parallelizing the explanation generation would give results faster but could waste precious computing resources. Given the rapidly increasing carbon footprint of ML algorithms [31], and the widespread deployment of explanation algorithms, there is a pressing need for smarter algorithms for this critical problem.

We propose a principled and scalable framework, Shahin,[1] for generating explanations for multiple predictions. There are a number of redundant computations that could be avoided by leveraging techniques such as materialization and reuse. Our techniques were inspired by Multi-Query Optimization (MQO) [28, 32]. Given a query workload, MQO seeks to identify common sub-expressions across queries, so that the reevaluation cost could be minimized. In our paper, we focus on perturbation based explanation algorithms. We describe a general set of heuristics and discuss how these ideas

[1] A subspecies of Peregrine Falcon, the fastest bird in the world

could be instantiated for popular perturbation based explanation algorithms (LIME, Anchor and SHAP) requiring minimal changes and very low overhead. These techniques are widely used for explaining tabular data and have been incorporated into the ML offerings of Google [12], AzureML [21], and others [14]. Our proposed approach achieves significant speedup *without compromising the explanation quality*. In short, we adapt the techniques pioneered by the database community to solve a practical problem in data science.

Opportunities for Optimization. Intuitively, algorithms such as LIME [26] and Anchor [27] work by perturbing the data, applying the blackbox classifier on the perturbations, and using the resulting predictions to generate explanations. We trained a Random Forest classifier on the widely used Census-Income dataset [37], and used LIME and Anchor to explain a single instance. Invoking the classifier on the perturbations accounted for 88% of the execution time for LIME and 92% for Anchor. Our key insight is that it is possible to avoid the classifier invocation by reusing perturbation generated for explaining one tuple. For example, if two tuples t_i and t_j have some overlap, one could generate perturbations that could be used for the explanations of both of them. Naive optimization techniques such as persisting all the perturbations or greedily choosing perturbations are not viable and provide only minor improvements. We perform a lightweight preprocessing of the dataset and use the collected statistics to generate perturbations smartly. We identify a number of such optimization opportunities and propose effective heuristics for speeding up the explanations.

Summary of Contributions.

- We identify an important problem of generating explanations for multiple predictions over *tabular* data.
- We analyze popular explanation algorithms, identify multiple redundant computations and develop scalable algorithms inspired by database techniques.
- We conduct extensive experiments that show that SHAHIN achieves significant speedups with very little overhead.

2 PRELIMINARIES

In this section, we formally define the problem of generating explanations for multiple predictions.

2.1 Problem Statement

We are given a *batch* of tuples $B = \{t_1, t_2, \ldots, t_n\}$, a classifier C and an explainer $\mathcal{E}$. Let $y_i = C(t_i)$ be the prediction for tuple t_i and $e_i = \mathcal{E}(t_i, C)$ be the corresponding explanation. Our goal is to generate explanations for the predictions of all the tuples in B. The explanations could be in the form of a rule "IF $A_i = u$ then class=Positive". Alternatively, it could be in the form of weights associated with each attribute such that attributes vital for the prediction getting a higher value. We shall describe the different type of explanations in Section 3.

> EXPLANATIONS FOR MULTIPLE PREDICTIONS (EMP) PROBLEM: Given a batch of tuples B, classifier C and explainer $\mathcal{E}$, efficiently generate explanations for each of tuples t_i such that the cumulative cost of computing explanations is minimized.

A straightforward approach would assign more resources by running the explanation algorithms in parallel on many machines. Instead, we propose a more promising and simpler approach inspired by multi-query optimization [28, 32] that achieves speed up by avoiding redundant computations by materializing them. As we shall show in our experiments, our approach outperforms the parallelization strategy even for small batches.

Batch and Streaming Variants. In the *batch* variant, we are provided with a batch of individual predictions that must be explained. In a number of data science applications, the set of tuples on which the predictions need to be made and explained is available beforehand. Emerging scenarios in responsible AI and explanation summarization work by generating explanation for each tuple in the test set and post-processing the generated explanations. This allows a number of optimization opportunities by performing a lightweight pre-processing to identify the redundant computations. These could then be pre-computed and materialized for later use. The other is the *streaming* scenario where the predictions arrive one at a time and we need to compute explanations for them *immediately*. We do not have the luxury of pre-hoc identifying the redundant computations and might also have additional constraints on resource consumption. We need to identify promising candidates for redundant computations in a principled manner. Our proposed approaches are generic enough to handle both the scenarios. Figure 1 illustrates the key components of our proposed approach.

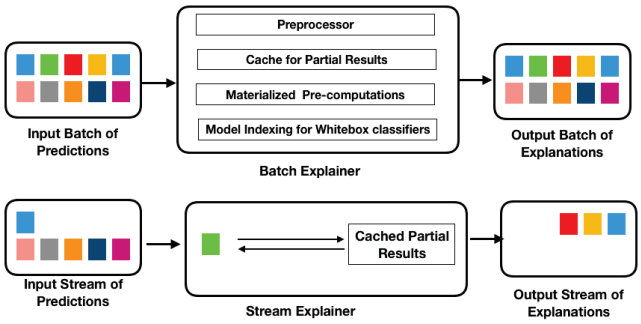

Figure 1: Illustration of our proposed approach.

3 EXPLAINING MULTIPLE PREDICTIONS FOR LIME, ANCHOR AND SHAP

We focus on three perturbation based algorithms – LIME [26], Anchor [27], and KernelSHAP [20]. Internally, they use diverse techniques and are exemplars in showing the generality of our approach. We provide the necessary background and focus on the optimizations that provide the most bang for the buck, easy to implement, and are generalizable to other explanation algorithms.

Key Idea. Each of the perturbation algorithms follows the same template. Given an input tuple t_i, they generate multiple perturbations P_i, and post-process the perturbations and their classifier predictions to generate explanations. We could also abstract the perturbation scheme. Given a tuple t_i, each of the explainers generate a perturbed sample by 'freezing' a subset of t_i's attributes and then replacing the rest of the values randomly according to a

perturbation scheme. For example, LIME uses the frequency distribution of an attribute A_k when replacing the attribute A_k of t_i with the most frequent value having a higher likelihood of being filled.

Suppose that two tuples t_i and t_j have an overlap in attribute values. It is possible that the explanation algorithm could choose to freeze this overlap and fill the remaining attribute values. Hence, one could generate a common perturbation and reuse them for both t_i and t_j. We could extend this intuition further. Given a batch of tuples D, frequent itemsets (by definition) are subsets of attribute values that co-occur in multiple tuples. This increases the likelihood that the explanation algorithm would choose to freeze these itemsets and generate perturbations around them. Hence, they are also promising candidates to cache so that they could be used for explaining multiple tuples.

Shahin takes a uniform random sample of the batch and applies a traditional frequent itemset algorithm. Each frequent itemset f is of the form $\{A_i = u, A_j = v, \ldots\}$ where $A_i, A_j, \ldots$ are arbitrary features and $u, v, \ldots$, are the corresponding values of those features that are frequent in the batch. The sample size is heuristically chosen as *max(1000, 1% of batch)*.

3.1 LIME for EMP Problem

LIME Primer. Local interpretable model-agnostic explanations (LIME) [26] is a seminal work that trains an interpretable surrogate model to approximate the blackbox classifier C and generate explanations. The explanation of LIME corresponds to a small set of attributes with *relative weights*. For a two class classification problem, attributes that contribute to positive class will have positive weights. We can obtain an ordering of the importance of the attributes to the prediction by sorting based on the weights. LIME consists of four key operations: (1) perturbing t_i to obtain samples S; (2) running blackbox classifier C on S; (3) training an interpretable model C_{int} on S that acts as a surrogate for C; (4) generating explanations for t_i by analyzing C_{int}. Profiling on LIME shows that the steps (1) and (2) account for more than 95% of total execution time.

Adapting LIME for EMP. One can simultaneously minimize the cost of both steps (1) and (2). First, let us dig deeper into how the perturbations are done for tabular data. For categorical attributes, LIME perturbs them by sampling values according to the data distribution from the training dataset. By default, it discretizes numerical data and treats it as categorical. Alternatively, it can perturb a numerical feature by sampling from a unit Normal distribution and performing inverse operation of mean-centering and scaling [26]. Shahin relies on two key insights. First, the perturbations are performed for each feature independently and based on a distribution that is *fixed* for each tuple. For example, consider two arbitrary tuples t_i and t_j. During perturbation, the probability that a feature A_i will be set the value u, for both t_i and t_j, is exactly same as the proportion of u in the training dataset. Second, given two arbitrary perturbations made by LIME, we must prefer those that could be reused for multiple tuples.

We compute the frequent itemsets F and for each $f \in F$, we compute τ perturbations. For example, if $f = (A_i = u, A_j = v)$, then we create τ perturbations such that all of them have $(A_i = u, A_j = v)$ while the values for other features are obtained using LIME's perturbation techniques. The parameter τ is set automatically by Shahin based on the resource constraints. The classifier is invoked on each of the perturbations and its output is stored. Given a new tuple t_i for explanation, we check if t_i contains any of the frequent itemsets. If so, we pool the τ perturbations corresponding to those itemsets. For the remaining perturbations, we follow the same procedure as LIME. The reused perturbations and their labels result in savings in terms of both classifier invocations and needless creation of perturbations. The pseudocode can be found in Algorithm 1.

Algorithm 1 LIME for EMP Problem

```
1:  Input: A batch B, blackbox classifier C, number of samples N
2:  Compute frequent itemsets F over B
3:  Generate τ perturbations ∀f ∈ F
4:  Invoke C on all perturbations and store the output in P
5:  for each tuple t_i ∈ B do
6:      S = retrieve reusable samples and labels from P
7:      S' = Obtain N − |S| perturbations, invoke C on them
8:      S = S ∪ S'
9:      Compute proximity between t_i and each s ∈ S
10:     Train interpretable model M using S
11:     Generate explanations for t_i using M
12: return explanations for B
```

3.2 Anchor for EMP Problem

Anchor Primer. Anchor [27] is another popular perturbation based explantion algorithm that outputs easy to understand rules of the form IF $A_i = u$ AND $A_j = v$ THEN class=1. For each rule, Anchor also provides two metrics – precision and coverage. Precision is the proportion of tuples in which the rule holds. Coverage is the fraction of tuples on which the predicates of the rule holds. Given a tuple t_i and a desired threshold on precision, Anchor provides a rule with high coverage whose precision exceeds the bound. Anchor consists of three key steps: (1) Generating candidate rules; (2) Estimating their precision; (3) Identify K best candidates with high precision and coverage and repeating from step 1 till the precision constraints are satisfied.

Adapting Anchor for EMP. Similar to LIME, Anchor provides multiple opportunities for optimization. The key bottleneck in Anchor is the estimation of precision of a candidate rule. For example, let IF $A_i = u$ THEN class=1 be such a rule. A naive approach would be to generate various samples where $A_i = u$ and the other attributes are obtained by using training data distribution. For each of the sample data points, we invoke the classifier and report the proportion in which class=1 occurs. Since classifier invocation is very expensive, Anchor uses a sophisticated multi-armed-bandits to minimize the number of such calls.

We begin by identifying the frequent itemsets F. For each $f \in F$, we estimate the precision of rules. If the precision of a rule containing f as its predicate is higher than the user provided threshold, then the rule could be used as an Anchor for all tuples containing f. Since f was a frequent itemset, it is likely to have a high coverage. The second optimization is to bootstrap the computation of precision for candidate rules containing a superset of frequent itemsets. Let $A_i = u$ be a frequent itemset. Then the process of

estimating its precision requires the generation of various sample data points and the invocation of classifier on it. Consider a rule IF $A_i = u$ AND $A_j = v$ THEN class=1. Instead of estimating the precision from scratch, we can scan the samples generated for $A_i = u$, find the subset that also contains $A_j = v$. By computing the proportion of those that have class=1, we can obtain a preliminary estimate that could be refined as needed. Finally, the coverage of the rules are fixed for each candidate rule. Hence, we materialize the coverage of all the candidate rules so that they are not recomputed again and again. Given multiple candidate rules satisfying the requirements, we pick the rule with least number of predicates. Algorithm 2 describes coarse-grained pseudocode.

Algorithm 2 Anchor for EMP Problem

```
Input: A batch B, blackbox classifier C
Compute frequent itemsets F over B
Generate candidate rules R using F and estimate precision
for each tuple t_i ∈ B do
    R' = {}
    loop
        Generate candidate rules by appending new predicates from t_i to R'
        if precision of any candidate rule r ∈ R' satisfies precision constraints then
            Use r as Anchor
        else
            Bootstrap precision for the candidate rules R' from materialized samples
            Add current precision estimates of rules to R
            Find best candidate rules R'' ⊆ R'
            Update precision of rules R''
return explanations for B
```

3.3 KernelSHAP for EMP Problem

Shapley Values. Given a tuple t_i, SHAP [20] is a family of algorithms that use Shapley values for allocating contribution/importance of each feature value $A_j = v$ in t_i. Intuitively, Shapley values computes the marginal contribution for each feature over all possible subsets of features. Computing the exact Shapley value requires exponential time. Hence, the values are computed approximately through sampling [34]. Feature importances computed via Shapley values have a number of appealing theoretical properties.

KernelSHAP Primer. KernelSHAP can estimate the feature importances of any blackbox functions. It consists of four major steps: (1) Generate multiple random feature subsets, estimate their weight through SHAP kernel and convert each feature subset to a random perturbations through sampling from the training data distribution; (2) Apply blackbox classifier on the random data perturbations; (3) Build a weighted linear and interpretable model; (4) Compute the Shapley values for each feature.

Adapting KernelSHAP for EMP**.** First, we obtain the frequent itemsets F and generate τ random perturbations. Suppose the feature subset is $A_i = u, A_j = v$. Then for each of the τ perturbations, we set their A_i to u and A_j to v. The other attribute values are filled by sampling according to their data distribution. We invoke the classifier on each of these random data perturbations and store the predictions. KernelSHAP computes M random data perturbations before feeding them to an interpretable model. Given a tuple t_i for explanation, we identify if it contains any feature subset that is frequent. If so, then we can reuse all random perturbations and their labels. For the remaining budget, we randomly chose a feature subset and check if it is a superset of any other frequent itemset. If so, we can again reuse those data perturbations. For example, if $A_i = u$ is frequent but $A_i = u, A_j = v$ is not, we can still scan the random data perturbations of $A_i = u$ for those that also have $A_j = v$. Another key optimization is to choose random feature subsets in proportion to the weight provided by SHAP kernel defined as [20].

$$\pi(m, s) = \frac{m - 1}{\binom{m}{s} \times s \times (m - s)} \tag{1}$$

Here m is the maximum size of the feature subset while s is the size of current feature subset. We can see that when s is small or $s \sim m$, the value $\pi(m, s)$ is large. In other words, generating feature subsets that are either very small (1 or 2) or very large (as large as m or $m-1$) is preferable to feature subsets of intermediate size such as $m/2$. This optimization has been previously observed in [22]. Algorithm 3 puts together all these ideas.

Algorithm 3 KernelSHAP for EMP Problem

```
Input: A batch B, blackbox classifier C, number of samples N
Compute frequent itemsets F over B
Generate τ random data perturbations ∀f ∈ F
Invoke C on all perturbations and store the output in P
for each tuple t_i ∈ B do
    S = S' = {}
    if t_i contains any frequent itemset then
        S = S∪ retrieve the perturbations and their labels from P
    while |S| + |S'| < N do
        Pick feature subset size according to Equation 1
        Pick a random subset s
        if s is a superset of any frequent itemset then
            S' = S'∪ any relevant perturbations from P
    Invoke C on perturbations from S'
    S = S ∪ S'
    Compute weight of each s ∈ S using SHAP kernel
    Train an interpretable model M
    Generate explanations for t_i using M
return explanations for B
```

3.4 Optimization Principles used by SHAHIN

The ideas behind SHAHIN are generic and could be used to speedup other perturbation based explanation algorithms over tabular data.

Materialization and Reuse of Perturbations. The key insight is to identify the expensive computations that are repeated and materialize the intermediate results. For example, invoking blackbox classifiers is often the biggest bottleneck accounting for 88% of the execution time for LIME and 92% for Anchor for Census-Income dataset. However, it is often unlikely that random perturbations of two tuples would have produced a common sample whose classifier

invocation could be minimized. We need to *engineer* opportunities for reuse. By preferentially selecting perturbations (such as by leveraging frequent itemsets) that could be reused, we achieved tremendous speedups.

Caching Other Invariant Results. The precision and coverage of a rule are invariant and do not change for different tuples. Even if they are inexpensive, it is sub-optimal to repeatedly calculate them. One can design a cache to store these invariant results for reuse. In some cases such as coverage, it is often clear that it is an invariant. In other cases, such as precision in Anchor, the parameter is often derived/estimated using a complex approach like multi-armed bandit which makes the invariance non-obvious. Similarly, other notions of invariance might exist and one could achieve good speedup by pre-computing them.

3.5 Streaming Variant of Shahin

Shahin could also be invoked in a streaming setting where individual predictions arrive one at a time and explanations have to be generated for them. Shahin uses a simple adaptation to retrofit the ideas developed for the batch setting for application in the streaming setting. Shahin is given a memory budget that constrains the amount of auxiliary information that could be saved such as frequent itemsets and perturbations.

Let us consider the LIME explanation algorithm. At the beginning, the tuples arrive one at a time and we do not have sufficient information to identify which of them are frequent itemsets. For the first tuple t_1, we generate and store all the perturbations along with their labels to a repository P. Clearly, there is no saving yet. For the second tuple to be explained t_2, we check if we can reuse any of the perturbations of t_1. If so, we reuse the perturbations as appropriate. If not, we generate perturbations of t_2, invoke the classifier and store these perturbations with labels to P. We also store the set of tuples that are being explained $\{t_1, t_2, \ldots, \}$.

We repeat this process until either: (a) P exceeds the memory budget or (b) number of tuples exceed a certain threshold (automatically chosen by Shahin such as 100). When the former happens, we kick out perturbations based on the LRU (least recently used) policy. At the limit, this approach implicitly ensures that perturbations containing frequent itemsets will not be kicked out. When the latter (b) happens, we run a frequent itemset mining algorithm and also store the negative border of the frequent itemsets. An itemset $\{A_i = u, A_j = v\}$ is in the negative border if it is not frequent but all of its *immediate subsets* i.e. $\{A_i = u\}$ and $\{A_j = v\}$ are frequent.

Let F be the set of frequent itemsets and their negative border. For each item $f \in F$, we compute the frequency of f in the set of tuples. Once this is done, we purge the tuples as they no longer are needed. Due to the way in which F is constructed, the perturbation repository P already consists of perturbations containing itemsets from F. If not, we purge that perturbation and use the obtained savings to generate perturbations of $f \in F$.

When new set of tuples arrive, we update the frequency of itemsets in F and reuse perturbations as appropriate from P. We also persist the next set of tuples so that the frequent itemsets can be recomputed. Any frequent itemset $f \in F$ that becomes infrequent is kicked out along its perturbations from P. We can see that this intuitive approach computes useful perturbations while keeping fresh by periodically recomputing the frequent itemsets. This property makes the algorithm dynamic and responsive to changes in the input stream.

3.6 Discussion

Handling Numeric Data. Both LIME and Anchor *discretize* numeric data (such as by quartile discretization) before generating perturbations. Shahin computes the frequent itemset over the discretized data. Of course, if discretization is not done the opportunity for identifying redundant computation decreases.

Anchor Explanation Quality. Both of the optimizations used by Shahin to speed up Anchor are exact and do not affect explanation quality. First, Shahin caches invariant results (such as precision and coverage) of a candidate rule and avoids recomputing it. Second, Shahin reuses perturbations to accelerate the precision computation of a candidate rule. For e.g. the precision of a rule R_1 such as "IF $A_i = u$ and $A_j = v$ THEN class=1" is obtained by generating various perturbations where $A_i = u$ and $A_j = v$, invoking classifier and finding the fraction for which class=1. We can see that one could reuse the perturbation of R_1 for candidate rules R_2 ("IF $A_i = u$ THEN class=1") and R_3 ("IF $A_j = v$ THEN class=1"). Furthermore, the reverse is also possible. We can reuse any of the perturbations of R_2 and R_3 that matches the predicate of R_1 to compute the precision of R_1. Once again, this optimization is exact.

LIME/KernelSHAP Explanation Quality. Both LIME and KernelSHAP generate a perturbation of tuple t_i as follows. First, they fix the value of a random subset of attributes (say that of A_1, A_2). For each of the other attribute ($A_j \in \{A_3, A_4, \ldots\}$), they choose a value from *Domain(A_j)* according to frequency distribution. This perturbation is passed to a classifier for getting the label. Then the whole process is repeated to get another perturbation. Shahin pre-generates perturbations for frequent itemsets. If tuple t_i has a frequent itemset f, one could reuse the pre-computed perturbations of f as and when LIME or KernelSHAP picks the set of attributes f. We can see that this on-demand reuse of cached perturbations does not introduce any approximation.

4 EXPERIMENTS

Goals. Our goals for the evaluation are threefold: (a) how does the optimizations proposed by Shahin compare against baseline approaches such as using a cluster of machines? (b) can the batch and streaming variants of Shahin achieve significant speedups for datasets with diverse characteristics? (c) what is the computational and space overhead imposed by Shahin?

4.1 Experimental Setup

Hardware and Platform. All our experiments were performed on a quad-core 2.2 GHz machine with 16 GB of RAM. Shahin and the various explanation algorithms are all implemented in Python. We compared Shahin against the author provided implementations of LIME, Anchor and SHAP.

Datasets. We conduct experiments over five diverse benchmark datasets. The Census-Income dataset [37] consists of 42 demographic and employment related variables and predicts whether

Dataset	#Tuples	#CatA	#NumA	#MaxDC	LIME (s)	Anchor (s)	SHAP (s)
Census-Income (KDD)	299285	27	15	18	15.67, 1.84, 5.59	1.67, 0.19, 0.59	0.95, 0.12, 0.33
Recidivism	9549	14	5	20	5.62, 0.624, 1.872	8.4 , 0.95, 3.11	1.62, 0.21, 0.46
lendingclub	42536	26	24	837	17.5, 1.7, 5.48	2.9, 0.28, 0.85	1.1, 0.15, 0.4
KDD Cup 1999	4000000	13	27	490	6.1, 0.51, 1.6	7.8, 0.64, 2.05	1.27, 0.15, 0.33
Covertype	581012	44	10	7	6.04, 0.55, 1.68	19, 1.69, 5.27	1.56, 0.19, 0.43

Table 1: Dataset Characteristics and Performance of SHAHIN. #CatA, #NumA are the number of categorical and numerical attributes. #MaxDC is largest domain cardinality of the categorical attribute. The last three columns display the average time taken in seconds for explaining a single tuple (for a batch of 1000) for the sequential baseline, SHAHIN-Batch and SHAHIN-Streaming variants of LIME, Anchor and SHAP respectively.

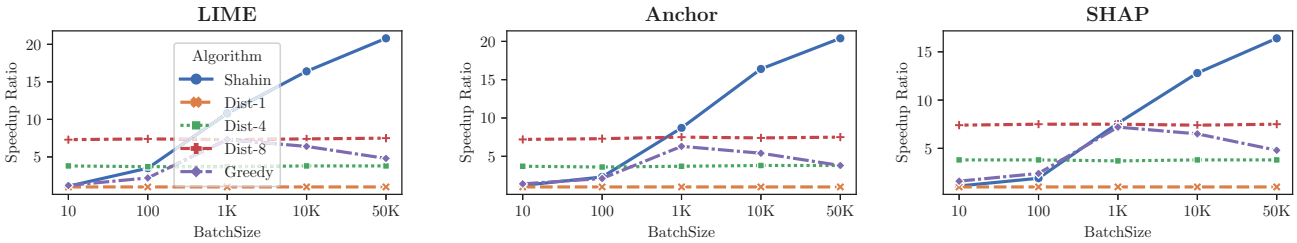

Figure 2: Comparison of Speedups achieved by SHAHIN and Baseline Algorithms

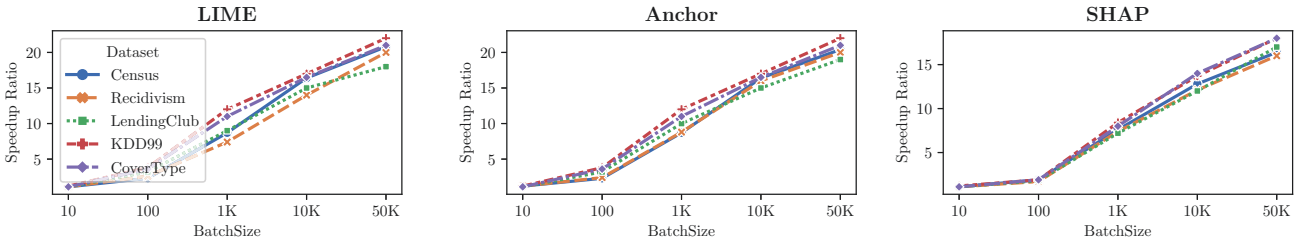

Figure 3: Speedup Ratio of SHAHIN-Batch for LIME, Anchor and SHAP for diverse datasets.

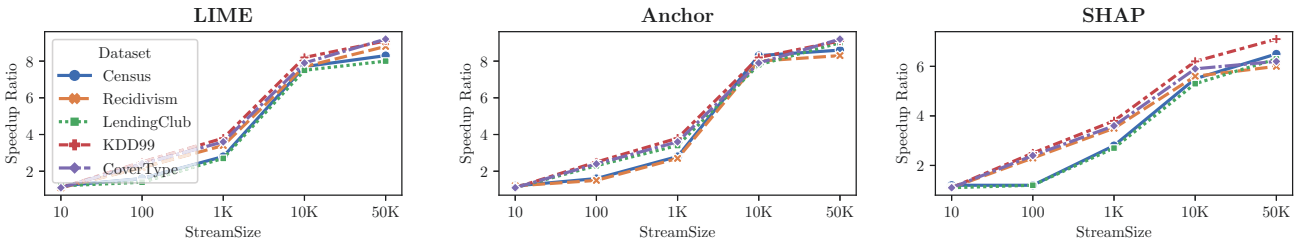

Figure 4: Speedup Ratio of SHAHIN-Streaming for LIME, Anchor and SHAP for diverse datasets.

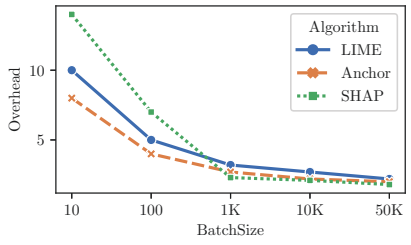

Figure 5: Overhead of SHAHIN

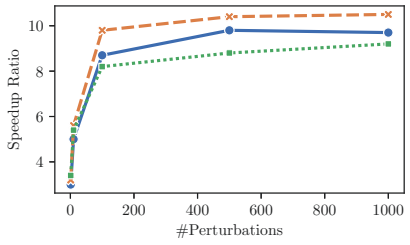

Figure 6: Impact of #Perturbations, τ

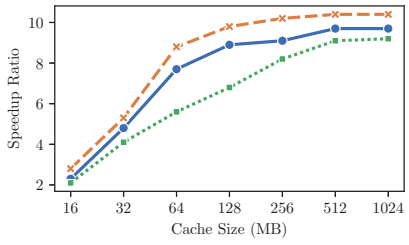

Figure 7: Impact of Cache Size (in MB)

the person makes $\$50K$ annually. The recidivism dataset is used to predict recidivism for individuals released from prison [30]. The lending club dataset predicts whether a loan will result in default or late payment. These three datasets have been used in prior explanation work such as [27]. KDDCup 1999 dataset seeks to build a network intrusion detector by predicting whether a connection is normal or abnormal [8]. The CoverType dataset from UCI repository tries to predict forest cover type from cartographic variables [8]. Each of these datasets are diverse in number of total, categorical and numerical attributes. The categorical attributes also have a wide spectrum in terms of domain cardinality. The details about these datasets are provided in Table 1. We partition the dataset into 1/3 and 2/3. We used the former for training the ML model and used the latter for prediction and explanation. These datasets exhibit diversity along key dimensions that influence the performance of explanation algorithms such as the number of categorical / numerical attributes and the frequency distribution of domain values for the categorical attributes.

Explanation Algorithms. We focus on three representative perturbation based algorithms – LIME, Anchor and SHAP as they use different algorithmic techniques and produce very different explanations. Our implementation of Shahin was based on Microsoft's InterpretML library [25]. Due to space limits, we conduct our experiments on random forest classifier. Since Shahin achieves speedup by minimizing the *number of classifier invocations*, this does not materially affect the conclusions. We used the default hyperparameters for the explanation algorithms (such as $\epsilon = 0.1, \delta = 0.05$ for Anchor). The default value of τ was set to 100.

Baseline Algorithms. We consider two baseline approaches. The first approach generates explanations sequentially one prediction at a time. Specifically, we consider a distributed variant where the explanation generation is equally spread across 1, 4 or 8 machines dubbed as Dist-1, Dist-4 and Dist-8 respectively. Given a batch of 10000 predictions to explain, Dist-8 will split them into 1250 predictions and spread them to 8 machines. The second baseline is Greedy. Given a memory budget, this approach stores all the perturbations until the budget is exhausted. When generating explanation, it reuses existing perturbations and their labels if possible. Otherwise, it generates new explanations and uses the LRU (least recently used) policy to replace unused perturbation. By default, we assumed that the space budget is 10x the size of the batch.

Batch Sizes. We evaluate both the batch and streaming variants of Shahin using the same set of tuples processed. We vary the batch size from 10 to 50K tuples. The order in which the tuples are processed is the same for all the algorithms. In the batch scenario, this denotes the set of tuples for which explanations has to be generated. In the streaming scenario, Shahin receives an explanation request one at a time. We randomly generated 10 different permutations and report the average results.

Performance Measures. We measure our proposed algorithms through two key metrics – speedup ratio and overhead. The speedup ratio is defined as the ratio of time taken by the sequential approach to the time taken by Shahin. The running time of both baselines and Shahin varies based on various factors such as the number of categorical attributes, their domain cardinality, classifier used and so on. However, these variations are removed when using speedup ratio allowing us to quantify the speedup achieved by Shahin's optimizations. While Shahin does use multiprocessing, we disable it to show that our superior performance comes from algorithmic improvements that minimize the number of classifier invocations. For the distributed version of the baseline (say Dist-8), we report the average time taken by the 8 machines as the runtime. By default, Shahin runs only on a single core of a single machine. The computational overhead measures the percentage of time taken by Shahin for housekeeping purposes such as computing frequent itemsets and retrieving relevant perturbations. The space overhead quantifies the space used for caching the intermediate results.

4.2 Experimental Results

Comparison against Baseline Approaches. First, we measure the speedup achieved by Shahin against the distributed and greedy baselines for explaining the predictions of a random forest classifier trained over the Census-Income dataset for a batch of tuples whose size is varied from 10 to 50K. Figure 2 shows that Shahin achieves the best speedup ratio regardless of the explanation algorithm used. We found similar trends for other datasets. We also report the average time taken in seconds for explaining a single tuple (for a batch of 1000) for the sequential baseline, Shahin-Batch and Shahin-Streaming variants of LIME, Anchor and SHAP respectively in Table 1. Even though Shahin ran on a single machine, it outperforms Dist-8 for batch sizes of 1000 and higher. For LIME explainer, Shahin achieves a speedup of 10.8 for a batch size of 1000 which increases to 20.8 for a batch size of 50K. The disparity in performance increases as the batch size is increased with Shahin being 2.6x faster than Dist-8 for a batch size of 50K. Larger batches allow Shahin to make informed decisions about the redundant computations through frequent itemsets.

The other baseline Greedy achieves substantial speedup for small batches that decreases for larger batches. This decrease is due to the algorithm's sub-optimal decisions regarding which perturbations to persist and which to remove. Blindly persisting all perturbations is not an effective strategy. In contrast, Shahin takes a holistic approach and produces consistent and significant speedups.

Performance of Shahin-Batch. Figure 3 shows the speedup for three explainers (LIME, Anchor and SHAP) for all the datasets. Regardless of the dataset, Shahin can speedup explanation generation by more than 20x for LIME and Anchor. The corresponding value for SHAP is around 18. Since invoking blackbox classifier accounts for more than 90% of the time taken, we can infer that speedup is primarily achieved by minimizing the number of calls to the classifier by intelligently materializing and reusing the perturbations. Similar to Figure 2, the speedup ratio increases with the batch size. Larger batch sizes allow Shahin to identify promising intermediate results to cache based on frequent itemsets.

Performance of Shahin-Streaming. Figure 4 shows the performance of Shahin in a streaming setting. Overall, the trends largely mirror those from the batch scenario. One can see that the speedups achieved in the batch setting dwarves that from the streaming setting. This is not surprising due to the ability of Shahin to preprocess the data and carefully chose the perturbations to materialize.

While the streaming based approach has a slower start obtaining only 25% of the speedup of that of batch setting, it improves to more than 60% for larger batches. Of course, one could achieve higher speedups through smarter frequent itemset computation.

SHAHIN **Overhead.** We evaluate the overhead of SHAHIN for LIME explainer for random forest over Census-Income dataset. SHAHIN mines the frequent itemsets, computes the perturbations and invokes the classifiers on the perturbations. For each tuple in the batch, it retrieves the relevant perturbations. The computation of frequent itemsets is done on a uniform random sample of size 1% of the batch size or 1000 whichever is larger). The small size ensures that this step is not very expensive. The computation of perturbations and classifier invocation gets amortized overall. Retrieving relevant perturbations is also a lightweight operation. Figure 5 shows that the percentage overhead imposed by SHAHIN is as little as 3% and 2% for batch size of 10K and 50K respectively.

We next study the impact of varying the resource budget for SHAHIN for random forest classifier for different explanation algorithms. We vary the number of perturbations stored for each frequent itemset between 1 to 1000. Figure 6 shows the result. Even storing 10 perturbations provides a 5x speedup for LIME. Interestingly, storing beyond 100 perturbations per frequent itemset does not provide any additional benefit. This is due to the fact that each tuple usually contains a handful of frequent itemsets. Hence, the number of perturbations generated by the traditional LIME explainer that contains them is also limited. Figure 7 demonstrates the results of speedup achieved when we vary the cache size between 16 MB to 1024 MB. The results are similar to that of Figure 6. Not surprisingly, smaller cache sizes results in lower speedup ratios. Increasing the cache size does not have a linear relationship with speedup ratio. For both LIME and Anchor, the performance peaks at around 128 MB and then plateaus for larger caches. This shows that there is a diminishing utility for storing perturbations of frequent itemsets. As the frequency of an itemset decreases, the number of tuples that could benefit from caching those perturbations also decreases. We found similar trends for other datasets.

Explanation Quality. The optimizations of SHAHIN such as caching are exact and does not affect the explanation quality. We empirically measure the fidelity of explanations generated by the sequential approach along two metrics: feature importance values and rank. For LIME and SHAP, we can represent the contribution of each feature as a real-valued number. We measure the Euclidean distance between the explanation generated by original LIME/SHAP and the SHAHIN variants. Using the importance values, we can compute a ranked list of features. We use Kendall-τ to measure the rank correlation between the ranking produced by original LIME/SHAP and SHAHIN variant. Given a batch of tuples, we compute the Kendall-τ for each of the tuple in the batch and compute its average.

SHAHIN achieves the *same ranking* of features for all three explanation algorithms. For Anchor and SHAP, the explanation generated by SHAHIN is identical to the one generated by the sequential baseline. For both of these algorithms, SHAHIN achieves the speedup through caching the redundant and invariant computations such as classifier invocations. Clearly, such a caching based approach will produce identical explanations. For LIME, we found that the maximum deviation in explanation values to be as small as 0.1. This discrepancy was due to a subtle reason. LIME relies on Ridge regression for estimating the feature importance weights that are computed internally using a randomized algorithm – stochastic average gradient descent solver. Since SHAHIN invokes the random number generator less than the baseline (due to caching), it results in a minor discrepancy. However, the magnitude of the discrepancy is comparable to the deviation between two explanations generated by LIME for two different random seeds.

5 RELATED WORK

Explanations of ML Models. ML explanation has applications in model debugging, detecting bias, ensuring operational safety, among others [7, 25]. In the last few years, there has been a groundswell of work for explaining and/or interpreting ML models sometimes with different motivations [19, 29]. Explaining complex models is often very challenging [1, 10, 13]. Hence, a number of recent research focus instead on generating explanations for individual predictions. Some local explanation algorithms explain the model introducing interpretable surrogate models in the local neighborhood of an individual prediction such as LIME [26] or Anchor [27]. Some popular explanation techniques provide explanations in the form of computing the contribution of individual features (such as SHAP [20]). These algorithms are incorporated into ML platforms of the industry sector including Google [12] and AzureML [21] among others. Robustness of the existing explanation methods are studied in [2]. Recently, there has been a number of work on generating explanations for database and data curation settings [5, 9, 35, 36].

Speeding up ML through Database Techniques. There has been prior work speeding up ML algorithms using database techniques. Some have used the concept of Materialization and reuse to speedup the ML model construction [4, 15]. Similarly, other database techniques like pipelining and operator pushdown are used to speedup ML tasks [17]. Query optimization techniques have been used to speedup random forest inference [3] and video queries [16, 18]. A recent work [23] used Multi query optimization [28, 32] techniques to speedup the *individual* explanations of a CNN based model.

6 CONCLUSION

We introduced a practical data science problem – efficiently generating explanations for a batch of predictions. Given the increasing popularity of explanation algorithms and their adoption in academia and industry, there is a pressing need to develop scalable algorithms. Our proposal, SHAHIN, introduces a number of optimizations for the ML context in order to identify redundant computations and achieves significant speedups over baselines. There are a number of interesting next steps such as extending these techniques for non-tabular data and other major classes of explanation algorithms. While the proposed optimizations are exact, it is possible that one could achieve substantial speedup by allowing certain approximation in the explanations generated.

7 ACKNOWLEDGMENTS

The research of Nick Koudas is supported in part by COHESA NOE. The research of Gautam Das was supported in part by grants 2008602, 1745925, and 1937143 from NSF.

REFERENCES

[1] A. Adadi and M. Berrada. Peeking inside the black-box: A survey on explainable artificial intelligence (xai). *IEEE Access*, 6:52138–52160, 2018.

[2] D. Alvarez-Melis and T. S. Jaakkola. On the robustness of interpretability methods. *arXiv preprint arXiv:1806.08049*, 2018.

[3] A. Ardalan, A. Doan, A. Akella, et al. Smurf: self-service string matching using random forests. *Proceedings of the VLDB Endowment*, 12(3):278–291, 2018.

[4] A. Asudeh, A. Nazi, J. Augustine, S. Thirumuruganathan, N. Zhang, G. Das, and D. Srivastava. Leveraging similarity joins for signal reconstruction. *Proc. VLDB Endow.*, 11(10):1276–1288, June 2018.

[5] V. Di Cicco, D. Firmani, N. Koudas, P. Merialdo, and D. Srivastava. Interpreting deep learning models for entity resolution: an experience report using lime. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–4, 2019.

[6] J. Dodge, Q. V. Liao, Y. Zhang, R. K. Bellamy, and C. Dugan. Explaining models: an empirical study of how explanations impact fairness judgment. In *Proceedings of the 24th International Conference on Intelligent User Interfaces*, pages 275–285, 2019.

[7] F. Doshi-Velez and B. Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.

[8] D. Dua and C. Graff. UCI machine learning repository, 2017.

[9] A. Ebaid, S. Thirumuruganathan, W. G. Aref, A. Elmagarmid, and M. Ouzzani. Explainer: entity resolution explanations. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 2000–2003. IEEE, 2019.

[10] L. H. Gilpin, D. Bau, B. Z. Yuan, A. Bajwa, M. Specter, and L. Kagal. Explaining explanations: An overview of interpretability of machine learning. In *DSAA*, pages 80–89. IEEE, 2018.

[11] B. Goodman and S. Flaxman. European union regulations on algorithmic decision-making and a "right to explanation". *AI magazine*, 38(3):50–57, 2017.

[12] Google. Increasing transparency with google cloud explainable ai, 2020. https://cloud.google.com/explainable-ai, Last accessed on 2020-02-12.

[13] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi. A survey of methods for explaining black box models. *ACM computing surveys (CSUR)*, 51(5):1–42, 2018.

[14] H20.AI Interpretability. H20.ai driverless ai, 2020. http://docs.h2o.ai/driverless-ai/latest-stable/docs/userguide/interpret-non-ts.html, Last accessed on 2020-02-12.

[15] S. Hasani, S. Thirumuruganathan, A. Asudeh, N. Koudas, and G. Das. Efficient construction of approximate ad-hoc ml models through materialization and reuse. *Proceedings of the VLDB Endowment*, 11(11):1468–1481, 2018.

[16] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: Optimizing neural network queries over video at scale. *Proc. VLDB Endow.*, 10(11):1586–1597, Aug. 2017.

[17] Z. Kaoudi, J.-A. Quiane-Ruiz, S. Thirumuruganathan, S. Chawla, and D. Agrawal. A cost-based optimizer for gradient descent optimization. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 977–992, New York, NY, USA, 2017. Association for Computing Machinery.

[18] N. Koudas, R. Li, and I. Xarchakos. Video monitoring queries. *arXiv preprint arXiv:2002.10537*, 2020.

[19] Z. C. Lipton. The mythos of model interpretability. *Queue*, 16(3):31–57, June 2018.

[20] S. M. Lundberg and S.-I. Lee. A unified approach to interpreting model predictions. In *NeurIPS*, pages 4765–4774, 2017.

[21] Microsoft. Model interpretability in azure machine learning, 2020. https://docs.microsoft.com/en-us/azure/machine-learning/how-to-machine-learning-interpretability, Last accessed on 2020-02-12.

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

[23] S. Nakandala, A. Kumar, and Y. Papakonstantinou. Incremental and approximate inference for faster occlusion-based deep cnn explanations. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1589–1606, 2019.

[24] K. Natesan Ramamurthy, B. Vinzamuri, Y. Zhang, and A. Dhurandhar. Model agnostic multilevel explanations. *arXiv*, pages arXiv–2003, 2020.

[25] H. Nori, S. Jenkins, P. Koch, and R. Caruana. Interpretml: A unified framework for machine learning interpretability. *arXiv preprint arXiv:1909.09223*, 2019.

[26] M. T. Ribeiro, S. Singh, and C. Guestrin. " why should i trust you?" explaining the predictions of any classifier. In *KDD*, pages 1135–1144, 2016.

[27] M. T. Ribeiro, S. Singh, and C. Guestrin. Anchors: High-precision model-agnostic explanations. In *AAAI*, 2018.

[28] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, pages 249–260, 2000.

[29] C. Rudin. Please stop explaining black box models for high stakes decisions. *CoRR*, abs/1811.10154, 2018.

[30] P. Schmidt and A. D. Witte. *Predicting recidivism in north carolina, 1978 and 1980*. Inter-university Consortium for Political and Social Research, 1988.

[31] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni. Green ai. *arXiv preprint arXiv:1907.10597*, 2019.

[32] T. K. Sellis. Multiple-query optimization. *TODS*, 13(1):23–52, 1988.

[33] A. Smith-Renner, R. Fan, M. Birchfield, T. Wu, J. Boyd-Graber, D. Weld, and L. Findlater. No explainability without accountability: An empirical study of explanations and feedback in interactive ml.

[34] E. Štrumbelj and I. Kononenko. Explaining prediction models and individual predictions with feature contributions. *Knowledge and information systems*, 41(3):647–665, 2014.

[35] S. Thirumuruganathan, M. Ouzzani, and N. Tang. Explaining entity resolution predictions: Where are we and what needs to be done? In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, pages 1–6, 2019.

[36] S. Thirumuruganathan, N. Tang, M. Ouzzani, and A. Doan. Data curation with deep learning. *EDBT*, 2020.

[37] UCI Repository. Census income dataset, 2020. https://archive.ics.uci.edu/ml/datasets/Census-Income+%28KDD%29, Last accessed on 2020-04-01.