HAL
open science

A Factorized Version Space Algorithm for "Human-In-the-Loop" Data Exploration

Luciano Di Palma, Yanlei Diao, Anna Liu

To cite this version:

Luciano Di Palma, Yanlei Diao, Anna Liu. A Factorized Version Space Algorithm for "Human-In-the-Loop" Data Exploration. 19th IEEE International Conference in Data Mining, Nov 2019, Beijing, China. hal-02274497v1

HAL Id: hal-02274497

https://inria.hal.science/hal-02274497v1

Submitted on 29 Aug 2019 (v1), last revised 3 Sep 2019 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Factorized Version Space Algorithm for "Human-In-the-Loop" Data Exploration

Luciano Di Palma
Ecole Polytechnique, France
luciano.di-palma@polytechnique.edu

Yanlei Diao
Ecole Polytechnique, France
yanlei.diao@polytechnique.edu

Anna Liu
University of Massachusetts Amherst, USA
anna@math.umass.edu

***Abstract*—While active learning (AL) has been recently applied to help the user explore a large database to retrieve data instances of interest, existing methods often require a large number of instances to be labeled in order to achieve good accuracy. To address this slow convergence problem, our work augments version space-based AL algorithms, which have strong theoretical results on convergence but are very costly to run, with additional insights obtained in the user labeling process. These insights lead to a novel algorithm that factorizes the version space to perform active learning in a set of subspaces. Our work offers theoretical results on optimality and approximation for this algorithm, as well as optimizations for better performance. Evaluation results show that our factorized version space algorithm significantly outperforms other version space algorithms, as well as a recent factorization-aware algorithm, for large database exploration.**

***Index Terms*—Active learning, version space, data exploration**

I. Introduction

In the setting of interactive data exploration (IDE), a user that comes to explore a large database is often driven by the goal of understanding some particular phenomenon. Such goals are treated as building a classification model over all objects in the database from no or very few labeled instances [1]–[4]; examples include classifying all the models in a car database as relevant or irrelevant to the interest of a customer, or classifying all the observations in a digital sky survey as relevant or irrelevant to the interest of a scientist. For IDE, active learning [5] has been explored to derive an accurate model with minimum user labeling effort while offering interactive performance in presenting next data instances for the user to label [2], [3].

In IDE, however, existing active learning techniques [5]–[8] often fail to provide satisfactory performance when such models need to be built over large databases. For example, our evaluation results show that on a Sloan Digital Sky Survey (SDSS) dataset of 1.9 million data instances, the state-of-the-art technique for IDE [3] requires the user to label 200-300 instances in order to learn a classification model over 6 attributes with F-score[1] of 95%. Similar results can be observed for other active learning techniques [5]–[7]. Asking the user to label a large number of instances to achieve accuracy, referred to as *slow convergence*, is undesirable.

[1]F-score is a better measure for exploring a large database than classification error given that the user interest, i.e., the positive class, often covers only a small fraction of the database; classifying all instances to the negative class has low classification error but poor F-score.

In this work, we aim to design new techniques to overcome the slow convergence problem by exploring two ideas:

Version Space Algorithms: First, we would like to leverage Version Space (VS) algorithms [5], [7], [9], [10] because they present a strong theoretical foundation for convergence. These algorithms model all possible configurations of a classifier that can correctly classify the current set of labeled data as a set of hypotheses forming a *version space* $\mathcal{V}$, and aim to seek the next instance such that its acquired label will enable $\mathcal{V}$ to be reduced. The most well known example is the bisection rule [9], [10], which among all unlabeled instances, looks for the one whose label allows $\mathcal{V}$ to be reduced by half or most close to that. It is shown theoretically to have near-optimal performance for convergence.

Implementing the bisection rule, however, is prohibitively expensive due to the exponential size of a version space. To reduce cost, various approximations have been proposed. Some of the most popular techniques are Simple Margin [7], Query-by-Disagreement [5], Query-by-Committee (QBC) [11], and ALuMA [6]. The first two methods often suffer from suboptimal performance since they are very rough approximations of the bisection rule. On the other hand, QBC and ALuMA can better approximate the bisection rule by sampling the version space, which, however, is very costly to run on large databases. For example, for *each* unlabeled instance ALuMA runs Hit-and-Run sampling with thousands of steps to sample a single hypothesis, and repeats this procedure to sample 1000 hypotheses for estimating the instance's reduction power of the version space; such cost is repeated for other unlabeled instances in the database. As can be seen, new techniques are needed to enable VS algorithms to achieve both fast convergence and high efficiency on large databases.

Factorization. To make VS algorithms practical for large databases, our second idea is to augment them with additional insights obtained in the user labeling process. In particular, we observe that when a user labels a data instance, her decision making process often can be broken into a set of smaller questions, and the answers to these questions can be combined to derive the final answer. For example, when a customer decides whether a car model is of interest, she may have a set of questions in mind: "Is gas mileage good enough? Is the vehicle spacious enough? Is the color a preferred one?" While the user may have high-level intuition that each question is related to a subset of attributes (e.g., the space depends on the

body type, length, width, and height), she is not able to specify these questions precisely. It is because she may not know the exact threshold value or the exact relation between a question and related attributes (e.g., how the space can be specified as a function of body type, length, width, and height). The above insight allows us to design new version space algorithms that leverage the high-level intuition a user has for breaking the decision making process, formally called a *factorization structure*, to combat the slow convergence problem.

More specifically, we make the following contributions:

1. *A Factorized Version Space Algorithm* (Section III): We propose a novel algorithm that leverages the factorization structure to create subspaces and factorizes the version space accordingly to perform active learning in the subspaces. Compared to recent work [3] that also used factorization for active learning, our work explores it in the new setting of VS algorithms and completely eliminates the strong assumptions made in [3] such as convex and conjunctive properties of user interest patterns, resulting in significant performance improvement.

2. *Theoretical results* (Section IV): We also provide theoretical results on the optimality of our factorized VS algorithm.

3. *Optimizations* (Section V): We further provide optimizations for sampling over a large version space.

4. *Evaluation* (Section VI): Evaluation results show that 1) for low dimensional problems, our optimized VS algorithm, without factorization, already outperforms existing VS algorithms including Simple Margin [7], Query-by-Disagreement [5], and ALuMA [6]; 2) for higher dimensional problems, our factorized VS algorithm outperforms VS algorithms [5]–[7], as well as DSM [3], a factorization-aware algorithm, often by a wide margin while maintaining interactive speed. For example, for a complex user interest pattern tested, our algorithm achieves F-score of over 90% after 100 iterations, while DSM is still at 40% and all other VS algorithms are at 10% or lower.

II. Related Work

In this section, we present most relevant results in Active Learning and Data Exploration Systems.

Active Learning. The recent results on active learning are surveyed in [5]. Our work focuses on a common form called pool-based active learning. In this setting, there is a small set of labeled data $\mathcal{L}$ and a large pool of unlabeled data $\mathcal{U}$ available. In active learning, an example is chosen from the pool in a greedy fashion, according to a utility measure used to evaluate all instances in the pool (or, if $\mathcal{U}$ is large, a subsample thereof). In our setting of database exploration, the labeled data $\mathcal{L}$ is what the user has provided thus far. The pool $\mathcal{U}$ is a subsample of size m of the unlabeled part of the database. The utility measure varies with the classifier and algorithm used. We focus on version space algorithms below.

Version Space Algorithms. Version Space (VS) algorithms are a particular class of active learning procedures. In such a procedure, the learner starts with a set of possible classifiers (hypotheses), which we denote by $\mathcal{H}$. The Version Space $\mathcal{V}$ is defined as the subset of classifiers $h \in \mathcal{H}$ consistent with the labeled data, i.e. $h(x) = y$ for all labeled points (x, y). As new labeled data is obtained, the set $\mathcal{V}$ will shrink, until we are left with a single hypothesis. Various strategies have been developed to select new instances for labeling.

Generalized Binary Search [9], [10]: Also called the *Version Space Bisection* rule, this algorithm searches for a point x for which the classifiers in $\mathcal{V}$ disagree the most; that is, the sets $\mathcal{V}^{x,y} = \{h \in \mathcal{V} : h(x) = y\}$ have roughly the same size, for all possible labels y. It has strong theoretical guarantees on convergence: the expected number of iterations needed to reach 100% accuracy is at most a constant factor larger than the optimal algorithm on average. Implementing the bisection rule, however, is prohibitively expensive: for each unlabeled instance x in the database, one must evaluate $h(x)$ for each hypothesis h in the version space, which is exponential in size $O(m^d)$, where m is number of unlabeled instances and d is the VC dimension [12]. A number of approximations of the bisection rule have been introduced in the literature:

Simple Margin [7]: As a rough approximation of the bisection rule for SVM classifiers, it leverages an heuristic that data points close to the SVM's decision boundary closely bisect the version space, $\mathcal{V}$. However, it can suffer from suboptimal performance, specially when $\mathcal{V}$ is very asymmetric.

Query By Disagreement [5]: This algorithm approximates the version space $\mathcal{V}$ by a positively biased and a negatively biased hypothesis, and selects a data point for which these two biased hypothesis disagree. Again, it also suffers from suboptimal performance since the selected point is only guaranteed to "cut" $\mathcal{V}$, but possibly not by a large amount.

Query by Committee (QBC) [11] and *ALuMA* [6]: QBC [11] attempts to estimate how much each point reduces the version space $\mathcal{V}$ through a sampling technique. It is a much better approximation of the bisection rule, but works by taking a single pass of the entire dataset, hence not suitable for pool-based sampling. ALuMA [6] is a "pool-based" version of QBC and uses a different technique for sampling the version space. It is shown to outperform QBC. Hence, we use ALuMA as a baseline version space algorithm in this work.

Data Exploration Systems. In the data exploration domain, a main objective is to design a database system that guides the user towards discovering relevant records in a large database. One example is Snorkel [13], a *data programming* framework where an expert user writes several *labeling functions* representing simple heuristics used for labeling a data point. By leveraging the prediction of several such functions, Snorkel is capable of building an accurate classifier without having the user manually label any data point.

Another line of work is "explore-by-example" systems [1]–[4], which leverage active learning to obtain a small set of user labeled data points for building an accurate model of the user interest. These systems require minimizing both the user labeling effort and running time in each iteration to find a new data point for labeling. In particular, recent work [3] is shown to outperform prior work [1], [2] via the following technique:

Dual-Space Model (DSM) [3]: In this work, the user interest

pattern P is assumed to form a convex object in the data space. For such a pattern, this work proposes a "dual-space model" (DSM), which builds not only a classifier but also a polytope model of the data space $\mathcal{D}$, offering information including the areas known to be positive and areas known to be negative. It uses both the polytope model and the classifier to decide the best instance to choose for labeling next. In addition, DSM explores *factorization* in a limited form: when the user pattern P is a conjunction of sub-patterns on non-overlapping attributes, it factorizes the data space into low-dimensional spaces and runs the dual-space model in each subspace.

III. A Factorized Version Space Algorithm

To improve the efficiency of version space (VS) algorithms on large databases, we aim to augment them with additional insights obtained in the user labeling process. In particular, we observe that when a user labels a data instance, her decision making process often can be broken into a set of smaller "yes" or "no" questions, which can be answered independently, and these answers can be combined to derive the final answer. Revisit the previous example: when a customer decides whether a car model is of interest, she has three questions in mind:

Q_1: Is gas mileage good enough?

Q_2: Is the vehicle spacious enough?

Q_3: Is the color a preferred one?

We do not expect the user to specify her questions precisely as classification models (which requires knowing the exact shape of the decision function and all constants used), but rather to have a high-level intuition of the set of questions and to which attributes each question is related.

Factorization Structure. Formally, we model such an intuition of the set of questions and the relevant attributes as a factorization structure: Let us model the decision making process using a complex question Q defined on an attribute set $\boldsymbol{A}$ of size d. Based on the user intuition, Q can be broken into smaller independent questions, $Q_1, \ldots, Q_K$, where each Q_k is posed on a subset of attributes $\boldsymbol{A}^k = \{A_{k1}, \cdots, A_{kd_k}\}$. The family of attribute sets, $(\boldsymbol{A}^1, \cdots, \boldsymbol{A}^K)$, $|\boldsymbol{A}^1 \cup \cdots \cup \boldsymbol{A}^K| \leq d$ may be disjoint, or overlapping in attributes as long as the user believes that decisions for these smaller questions can be made independently. In our work $(\boldsymbol{A}^1, \cdots, \boldsymbol{A}^K)$ is referred to as the *factorization structure*.

Note that the factorization structure needs to be provided by the user as it reflects her understanding of her own decision making process. The independence assumption in the decision process should not be confused with the data correlation issue. For example, the color and the size of cars can be statistically correlated, e.g., large cars are often black in color. But the user decision does not have to follow the data characteristics; e.g., the user may be interested in large cars that are red. As long as the user believes that her decision for the color and that for the size are independent, the factorization structure ({color}, {size}) applies. To the contrary, if the two attributes are not independent in the decision making process, e.g., the user prefers red if the car is small and black if the car is large, but the year of production is an independent concern, then the factorization structure can be ({color size}, {year}).

Decision functions. Given a data instance x, we denote $x^k = proj(x, \boldsymbol{A}^k)$ the projection of x over attributes $\boldsymbol{A}^k$. We use $Q_k(x^k) \rightarrow \{-,+\}$ to denote the user decision on x^k. Then we assume that the final decision from the answers to these questions is a boolean function $F : \{-,+\}^K \rightarrow \{-,+\}$:

$$Q(x) = F(Q_1(x^1), \ldots, Q_K(x^K)) \tag{1}$$

In this work, we assume that the decision function F is given by the user. The most common example is the conjunctive form, $Q_1(x^1) \wedge \ldots \wedge Q_K(x^K)$, meaning that the user requires each small question to be $+$ to label the overall instance with $+$. Given that any boolean expression can be written in the conjunctive normal form, $Q_1(x^1) \wedge \ldots \wedge Q_K(x^K)$ already covers a large class of decision problems, while our work also supports other decision functions that use $\vee$.

Given the decision function F, we aim to learn the subspatial decision functions, $\{Q_1(x^1), \ldots, Q_K(x^K)\}$, efficiently from a small set of labeled instances. For a labeled instance x, the user provides a collection of *subspatial labels* $(y_1, \ldots, y_K) \in \{+,-\}^K$ to enable learning.

Generality. We note the differences of our factorization framework from [3]: First, one of the main assumptions in [3] is that the set $\{x^k : Q_k(x^k) = +\}$ or the set $\{x^k : Q_k(x^k) = -\}$ must be a convex object, which is eliminated in this work. Second, the global decision function F must be conjunctive in [3], which is relaxed to any boolean expression in our work. Third, our factorization is applied to version space algorithms, as shown below, while [3] does not consider them at all.

A. Overview of A Factorized Version Space Algorithm

Based on the above insight, we propose a new active learning algorithm, called a *factorized version space algorithm*. It leverages the factorization structure provided by the user to create subspaces, and factorizes the version space accordingly to perform active learning in the subspaces.

Algorithm 1 A Factorized Version Space Algorithm

Input: database $\mathcal{D}$, initial labeled set $\mathcal{L}_0$, per iteration subsample size m, version space sample size M

$\mathcal{L} \leftarrow \mathcal{L}_0, \mathcal{U} \leftarrow \mathcal{D}$
while user is still willing **do**
  $\mathcal{U}' \leftarrow$ subsample$(\mathcal{U}, m)$
  for each subspace k **do**
    $\mathcal{U}^k \leftarrow \{x^k,$ for $x \in \mathcal{U}'\}$
    $\mathcal{L}^k \leftarrow \{(x^k, y^k),$ for $(x, y) \in \mathcal{L}\}$
    $\{p_k(x)\}_{x \in \mathcal{U}'} \leftarrow$ positive_cut_probability$(\mathcal{U}^k, \mathcal{L}^k, M)$
  end for
  $x^* \leftarrow \arg\min_{x \in \mathcal{U}'} \prod_k (1 - 2(1 - p_k(x))p_k(x))$
  $y^* \leftarrow$ get_labels_from_user(x^*)
  $\mathcal{L} \leftarrow \mathcal{L} \cup \{(x^*, y^*)\}, \mathcal{U} \leftarrow \mathcal{U}/\{x^*\}$
end while
$C^k \leftarrow$ train_majority_vote_classifier$(\mathcal{L}^k)$, $k = 1, \ldots, K$
return $x \mapsto F(C^1(x^1), \ldots, C^K(x^K))$

Algorithm 1 shows the pseudo-code of our algorithm. It starts by taking a labeled dataset (which can be empty), and creating a memory-resident sample from the underlying large database as an unlabeled pool $\mathcal{U}$ (line 1) to enable efficiency for interactive performance. It then proceeds to an iterative procedure (lines 2-11): In each iteration, it may further subsample the unlabeled pool to obtain $\mathcal{U}'$ to expedite learning (line 3). Then the algorithm considers each subspace (lines 4-8), including the both labeled instances, $(x^k, y^k) \in \mathcal{L}^k$, and unlabeled instances, $x^k \in \mathcal{U}'$, projected to this subspace. The key step is to compute for each unlabeled instance, x, how much its projection x^k can reduce the version space V^k once its label is acquired (line 7). This step requires efficient sampling of the version space, V^k, which is a main focus of Section V. Once the above computation completes for all subspaces, the algorithm chooses the next unlabeled instance that can offer best reduction of the factorized version space, $\tilde{V} = V^1 \times \ldots V^K$ (line 9). The derivation of this strategy, as well as the proof of its optimality, is detailed in Section IV. The selected instance is then presented to the user for labeling and the unlabeled pool is updated. The algorithm then proceeds to the next iteration. Once the user wishes to stop exploring, we train a majority vote classifier [6] for each subspace k. The majority vote classifier, C, is constructed by first computing a sample of hypotheses from the version space. Then for any point x, $C(x)$ is computed as the most frequent label across the sample. Given the classifiers, $(C^1, \ldots, C^K)$, for the subspaces, we build a final classifier $F(C^1, \ldots, C^K)$ (line 14), which can then be used to retrieve all the data instances of interest to the user from the database $\mathcal{D}$.

IV. Theoretical Analysis

A. Bisection Rule over Factorized Version Space

Let $\mathcal{X} = \{x_i\}_{i=1}^N$ be the collection of unlabeled data points, and let $y_i \in \mathcal{Y}$ represent the unknown label of x_i. The user interest pattern can be modeled as a *hypothesis function* $h : \mathcal{X} \to \mathcal{Y}$, and we denote by $\mathcal{H}$ the set of all hypotheses. We also assume a known probability distribution $\pi(h)$, representing our prior knowledge over which hypotheses are more likely to match the user preference.

The *version space* is the set of all hypotheses consistent with the labeled set $\mathcal{L}$: $V = \{h \in \mathcal{H} : h(x) = y, \forall (x, y) \in \mathcal{L}\}$.

Our factorized version space algorithm is based on a well known concept called **Version Space Bisection** rule (or Generalized Binary Search). It searches for the point x which most evenly divides the version space across all classes:

$$\arg\max_x 1 - \sum_{y \in \mathcal{Y}} p_{x,y}^2 \tag{2}$$

where $p_{x,y} = \pi_V(V_{x,y})$. Here π_V is the probability distribution π normalized over the version space V and $V_{x,y} = \{h \in V : h(x) = y\}$. Thus, the greedy strategy searches for the point x for which the sets $V_{x,y}$ have approximately the same probability mass for every possible label y of x.

Now, let's suppose that a factorization structure $(\boldsymbol{A}^1, \cdots, \boldsymbol{A}^K)$ is given. For each subspace k, the user labels the projection x^k of x over $\boldsymbol{A}^k$ based on a hypothesis from a hypothesis class $\mathcal{H}^k$ with prior probability distribution π^k. The user then provides a binary label $\{+, -\}$ for each subspace; in other words, for each x a collection of *subspatial labels* $(y_1, \ldots, y_K) \in \mathcal{Y}_F = \{+, -\}^K$ is provided by the user.

Definition 1: **Factorized hypothesis function and factorized hypothesis space**: Let $\mathcal{Y}_F = \{+, -\}^K$. We define the factorized hypothesis function as the function $H : \mathcal{X} \to \mathcal{Y}_F$ such that

$$H(x) = (h_1, \ldots, h_K)(x) = (h_1(x^1), \ldots, h_K(x^K)).$$

H belongs to the product space $\tilde{\mathcal{H}} = \mathcal{H}^1 \times \ldots \times \mathcal{H}^K$, which we call the factorized hypothesis space.

We assume that the user labels the subspaces independently and consistently within each subspace.

Definition 2: **Factorized version space**: Let $\mathcal{L}$ be the set of instances that have been labeled over subspaces at any iteration of active learning. Define the factorized version space as $\tilde{V} = \{H \in \tilde{\mathcal{H}} : H(x) = y \text{ for all } (x, y) \in \mathcal{L}\}$. We can see that $\tilde{V} = \prod_k V^k = V^1 \times \ldots \times V^K$, where $V^k = \{h \in \mathcal{H}^k : h(x^k) = y^k \text{ for all } (x, y) \in \mathcal{L}\}$ is the version space at subspace k.

Given the assumption of independent labeling among the subspaces, the prior probability distribution over the factorized hypothesis space is $\tilde{\pi} = \pi^1 \times \ldots \times \pi^K$.

Definition 3: **Factorized greedy selection strategy**: Let $p_{x,y} = \tilde{\pi}_{\tilde{V}}(\tilde{V}_{x,y})$, the strategy is defined as

$$\arg\max_x 1 - \sum_{y \in \mathcal{Y}_F} p_{x,y}^2. \tag{3}$$

B. Optimal properties

For the version space bisection rule, it has been shown [10] that the greedy strategy in (2) enjoys a near-optimal performance guarantee: the average number of labeled instances by the greedy algorithm is no larger than:

$$OPT \cdot \left(1 + \ln \frac{1}{\min_h \pi(h)}\right)^2, \tag{4}$$

where OPT is the minimum expected number of iterations across all active learning algorithms that continue until the hypothesis matching the user interest has been found.

Factorized greedy selection strategy. Applying the bisection rule to $(\mathcal{X}, \mathcal{Y}, \tilde{\mathcal{H}}, \tilde{\pi})$, and noting that $\min_{H \in \tilde{\mathcal{H}}} \tilde{\pi}(H) = \prod_k \min_{h_k \in \mathcal{H}^k} \pi^k(h_k)$, the following theorem is the direct extension of the above result.

Theorem 1 (Number of iterations with factorization): The factorized greedy strategy in (3) takes at most:

$$OPT_f \cdot \left(1 + \sum_k \ln\left(\frac{1}{\min_{h_k} \pi_k(h_k)}\right)\right)^2 \tag{5}$$

iterations in expectation to identify a hypothesis randomly drawn from $\tilde{\pi}$, where OPT_f is the minimum number of iterations across all strategies that continue until the true hypothesis over all subspaces has been found.

In the following, we derive a simplified computation of the factorization greedy strategy (3).

Theorem 2: Let $p_{x^k,+} = \pi^k_{V^k}(V^k_{x^k,+})$, the factorized greedy selection strategy (3) is equivalent to

$$\arg\max_x 1 - \prod_k (1 - 2p_{x^k,+}(1 - p_{x^k,+})). \tag{6}$$

Proof: First, by noting that $\tilde{V}_{x,y} = \prod_k V^k_{x^k,y^k}$, it implies $\tilde{\pi}_{\tilde{V}}(\tilde{V}_{x,y}) = \prod_k \pi^k_{V^k}(V^k_{x^k,y^k})$. Therefore $\sum_{y\in\mathcal{Y}} p^2_{x,y} =$

$$\sum_{y_1=\pm} \cdots \sum_{y_K=\pm} \prod_k \pi^k_{V^k}(V^k_{x^k,y^k})^2 = \prod_k \sum_{y_k=\pm} \pi^k_{V^k}(V^k_{x^k,y^k})^2 =$$
$$\prod_k (p^2_{x^k,+} + (1 - p_{x^k,+})^2) = \prod_k (1 - 2p_{x^k,+}(1 - p_{x^k,+})).$$

■

V. Optimizations

The main difficulty in implementing the greedy selection strategies, as discussed in the previous section, lies in efficient computation of the cut probability $p_{x,+} = \pi_V(V_{x,+})$. For this, we adopt a *sampling approximation* [6], [8], [11] :

$$p_{x,+} = \mathbb{P}(H(x) = +) \approx \frac{1}{M}\sum_{i=1}^{M} \mathbb{1}(H_i(x) = +) \tag{7}$$

where $\{H_i\}_{i=1}^M$ is a i.i.d. sample from π_V. Thus, our problem is to develop a sampling algorithm for hypotheses.

Our sampling procedure closely follows [6]. We first consider the class of homogeneous linear classifiers:

$$\mathcal{H}^{lin} = \{w \in B(0,1) : h_w(x) = \text{sign}(w^T x)\} \tag{8}$$

We can then improve the generalization power of this class in two ways. First, we can add a bias b to the linear classifier by simply adding a dummy feature of value 1 to each data point. Second, we can choose a kernel function $k(\cdot,\cdot)$ and replace each data point x by its kernel representation $(k(x,x_1),\ldots,k(x,x_t))$, where $\{x_1,\ldots,x_t\}$ is the collection of data points labeled so far.

Now, assume we have a labeled set $\mathcal{L} = \{(x_i, y_i)\}$, with $y_i \in \{-1, 1\}$. The version space is the set of all h_w with w restricted to the convex set:

$$W = \{w \in B(0,1) : y_i x_i^T w > 0\} \tag{9}$$

Luckily, sampling from convex sets is a well-know problem in computational geometry. It can be solved by the Hit-and-Run algorithm [14], which creates a random-walk inside the polytope which converges to the uniform distribution. A more detailed explanation on how to implement this step can be found in the appendix A.

Rounding. The hit-and-run algorithm itself can have a large mixing time, specially in cases where the convex body is very elongated in one direction. To solve this problem, we adopt a *rounding procedure* [15], [16]. Basically, it consists of finding a linear transformation T for which the convex body $T(W)$ is more evenly elongated across all directions. The Hit-and-Run sampling is run over $T(W)$, with the final sampling over W being obtained through the inverse transformation T^{-1}. Computing the rounding transformation is done in two steps:

1) Find (an approximation of) the ellipsoid $\mathcal{E}$ of minimum volume containing W.
2) Set T as any linear isomorphism taking $\mathcal{E}$ into a ball or radius 1.

More details on how to implement these steps can be found in the appendix B.

Hit-and-Run's starting point. Hit-and-Run starts by computing a point W_0 inside W. Although this could be done by solving a linear programming task, it can take a significant amount of time. Instead, we rely on the rounding procedure: the ellipsoid $\mathcal{E}$ computed satisfies $\gamma\mathcal{E} \subset W \subset \mathcal{E}$, where $\gamma\mathcal{E}$ is obtained by shrinking all axes of $\mathcal{E}$ by some $0 < \gamma < 1$. This property guarantees that the center of $\mathcal{E}$ must be inside W, which can then be used as initial point.

VI. Experiments

We implemented all of our techniques in a Java-based prototype, which connects to a PostgreSQL database. In this section, we evaluate these techniques against state-of-the-art active learning algorithms [3], [5]–[7] in terms of accuracy (using F-score) and efficiency (execution time in each iteration).

A. Experimental Setup

We evaluate our techniques using the Sloan Digital Sky Survey dataset (SDSS, 190 million tuples). This dataset contains the "PhotoObjAll" table with 510 attributes[2] and 190 million sky observations. We used a 1% sample (1.9 million tuples, 4.9GB) for running active learning algorithms. SDSS also offers a query release, where the SQL queries reflect the data interest of scientists. We extracted 11 queries to build a benchmark, which can be found in Appendix C, and treated them as the ground truth of the positive classes of 11 classifiers to be learned — our system **does not** need to know these queries in advance, but can learn them via active learning. These queries reflect different dimensionality (2D-7D) and complexity of the decision boundary (e.g., various combinations of linear, quadratic, log patterns).

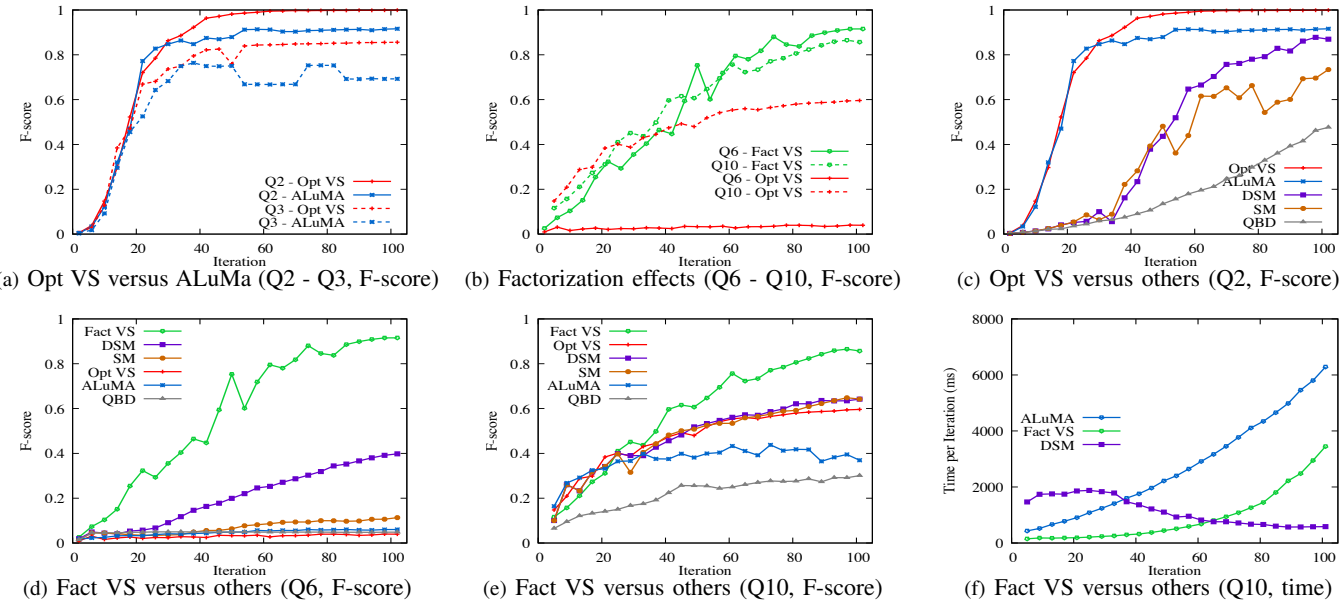
Algorithms: We compare our algorithms to state-of-the-art VS algorithms, Simple Margin (SM) [7], Query By Disagreement (QBD) [5], and ALuMA [6], and another factorization-aware algorithm, DSM [3]. In our experiments, each active learning algorithm starts with one positive example and one negative example, and runs up to 100 additional labeled examples.

Servers: Our experiments were run on four servers, each with 40-core Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz, 128GB memory, OpenJDK 1.8.0 on CentOS 7. Our factorized algorithm used one core for sampling in each subspace.

B. Evaluating Our Techniques using SDSS

Expt 1 (Optimization of Hit-and-Run): We first investigate the effect of Rounding on the Hit-and-Run sampling, as proposed in Section V. The resulting algorithm, denoted as "Opt VS", is compared to a baseline without such optimization, which is ALuMA [6]. For all 11 SDSS queries, Opt VS

[2]http://www.sdss3.org/dr8/



(a) Opt VS versus ALuMa (Q2 - Q3, F-score) (b) Factorization effects (Q6 - Q10, F-score) (c) Opt VS versus others (Q2, F-score)

(d) Fact VS versus others (Q6, F-score) (e) Fact VS versus others (Q10, F-score) (f) Fact VS versus others (Q10, time)

Fig. 1. Comparison between our proposed algorithms and the state-of-the-art over SDSS dataset

significantly outperforms ALuMa, while Figure 1(a) shows the F-scores of Q2 and Q3.

Expt 2 (Factorization): We next study the effect of factorization, by comparing Opt VS with its extension to factorization, denoted as "Fact VS". For factorization, each predicate in the target query corresponds to its own factorized subspace. Q1-Q4 are 2D queries and not factorized. Q5-Q7 are factorized where each predicate corresponds to each subspace with no overlap of attributes across subspaces, while Q8-Q11 are also factorized but with overlap of attributes across subspaces. For Q5-Q11, Fact VS outperforms Opt VS, often by a wide margin. Figure 1(b) show the F-score for Q6 and Q10. In the particular case of Q6, the factorized version reaches $> 90\%$ F-Score after 100 iterations while the non-factorized version is still at less $< 10\%$.

C. Comparing to Other Methods using SDSS

Expt 3 (A comparative study): In figures 1(c)-1(e), we compare to three VS algorithms, Simple Margin (SM) [7], Query By Disagreement (QBD) [5], and ALuMA [6], as well as DSM [3], a factorization-aware algorithm.

We first consider the case without factorization. Our Opt VS algorithm outperforms the three VS algorithms [5]–[7] and DSM [3] most time for the 2D queries, Q1-Q4, that do not use factorization. Figure 1(c) show the results for Q2. During the initial iterations, Opt VS improves much faster than other algorithms, and remains the best across all iterations.

We next consider factorization. Our Fact VS almost always outperforms others, including DSM that uses factorization under stronger assumptions. Figures 1(d) and 1(e) show the results for Q6 and Q10. In general, Fact VS outperforms DSM, which in turn outperforms all other (non-factorized) algorithms. In the particular case of 1(d), after 100 iterations Fact VS is at $> 90\%$ accuracy, while DSM is still at merely 40% and all of the other alternatives are at 10% or lower.

Finally, Figure 1(f) shows the running time of Fact VS, DSM, and ALuMA for Q10, our most expensive query. ALuMA is consistently more expensive than Fact VS. The two factorized algorithms take at most a couple of seconds per iteration, thus better suiting the interactive data exploration scenario. Moreover, DSM exhibits a large warm-up time and is slower than Fact VS during the first 60 iterations. Thus, Fact VS may be preferred to DSM and ALuMA given its better accuracy and lower time per iteration in initial iterations.

VII. Conclusions

To overcome the slow convergence of active learning (AL) in large database exploration, we presented a new algorithm that augments version space-based AL algorithms, which have strong theoretical results on convergence but are costly to run, with insights on the factorization structure employed in the user labeling process. The resulting algorithm factorizes the version space to perform active learning in a set of subspaces, with provable results on optimality and optimizations for performance. Evaluation results using real world datasets show that our algorithm significantly outperforms state-of-the-art version space algorithms, as well as a recent factorization-aware algorithm, for large database exploration.

References

[1] K. Dimitriadou, O. Papaemmanouil, and Y. Diao, "Explore-by-example: an automatic query steering framework for interactive data exploration," in *SIGMOD Conference*, 2014.

[2] K. Dimitriadou, O. Papaemanouil, and Y. Diao, "AIDE: An active learning-based approach for interactive data exploration," *TKDE*, 2016.

[3] E. Huang, L. Peng, L. D. Palma, A. Abdelkafi, A. Liu, and Y. Diao, "Optimization for active learning-based interactive database exploration," *Proceedings of the VLDB Endowment*, 2018.

[4] W. Liu, Y. Diao, and A. Liu, "An analysis of query-agnostic sampling for interactive data exploration," *Communications in Statistics – Theory and Methods*, 2017.

[5] B. Settles, *Active Learning*, 2016.

[6] A. Gonen, S. Sabato, and S. Shalev-Shwartz, "Efficient Active Learning of Halfspaces: an Aggressive Approach," *JMLR*, 2013.
[7] S. Tong and D. Koller, "Support Vector Machine Active Learning with Applications to Text Classification," *JMLR*, 2001.
[8] K. Trapeznikov, V. Saligrama, and D. Castañón, "Active Boosted Learning (ActBoost)," *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, 2011.
[9] S. Dasgupta, "Analysis of a greedy active learning strategy," *NIPS*, 2005.
[10] D. Golovin and A. Krause, "Adaptive Submodularity : A New Approach to Active Learning and Stochastic Optimization," *Time*, 2007.
[11] R. Gilad-Bachrach, A. Navot, and N. Tishby, "Query by Committee Made Real," *NIPS*, 2005.
[12] M. J. Kearns and U. Vazirani, *An Introduction to Computational Learning Theory*, 2018.
[13] A. J. Ratner, C. M. De Sa, S. Wu, D. Selsam, and C. Ré, "Data programming: Creating large training sets, quickly," in *NIPS*, 2016.
[14] L. Lovász, "Hit-and-run mixes fast," *Mathematical Programming*, 1999.
[15] L. Lovász, *An Algorithmic Theory of Numbers, Graphs and Convexity*, 1986.
[16] D. De Martino, M. Mori, and V. Parisi, "Uniform sampling of steady states in metabolic networks:Heterogeneous scales and rounding," *PLoS ONE*, 2015.
[17] G. Marsaglia, "Choosing a Point from the Surface of a Sphere," *The Annals of Mathematical Statistics*, 2007.
[18] H. S. Haraldsdóttir, B. Cousins, I. Thiele, R. M. Fleming, and S. Vempala, "CHRR: Coordinate hit-and-run with rounding for uniform sampling of constraint-based models," *Bioinformatics*, 2017.

Appendix A
Hit-and-Run implementation

Let $W \subset \mathbb{R}^d$ be a convex body. The Hit-and-Run algorithm is a randomized algorithm for sampling a point $x \in W$ uniformly at random. More precisely, it generates a Markov Chain inside W which converges to the uniform distribution; starting at any given point $X_0 \in W$, it iteratively performs two steps:

1) Sample a direction vector D uniformly at random over the unit sphere
2) Set X_{t+1} as a random point on the line segment $\{s \in \mathbb{R} : X_t + sD\} \cap W$

Implementing step 1 can be easily done through the Marsaglia Algorithm [17]: simply sample $D \sim \mathcal{N}(0, I_d)$ and set $D \leftarrow D/\|D\|$.

As for step 2, the main difficulty is to find the intersections of the line $L = \{s \in \mathbb{R} : X_t + sD\}$ with the boundary of W. Although there are methods for finding these extremes for a general convex body W, we will focus to the particular case of a polytope $P = \{x : Ax \geq 0\}$ intersected with the unit ball: $W = P \cap B(0,1)$. In this case, it is easy to see that for the polytope:

$$\begin{aligned} X_t + sD \in P &\iff A(X_t + sD) \geq 0 \\ &\iff sAD \geq -AX_t \\ &\iff sa_i^T D \geq -a_i^T X_t, \text{ for all } i \\ &\iff \max_{a_i^T D > 0} -\frac{a_i^T X_t}{a_i^T D} \leq s \leq \min_{a_i^T D < 0} -\frac{a_i^T X_t}{a_i^T D} \end{aligned}$$

and, for the unit ball (keeping in mind that $\|D\| = 1$):

$$\begin{aligned} X_t + sD \in B(0,1) &\iff \|X_t + sD\|^2 \leq 1 \\ &\iff s^2 + 2(X_t^T D)s + \|X_t\|^2 - 1 \leq 0 \\ &\iff -X_t^T D - \sqrt{\Delta} \leq s \leq -X_t^T D + \sqrt{\Delta} \end{aligned}$$

where $\Delta = (X_t^T D)^2 - \|X_t\|^2 + 1$. Note that $\Delta > 0$ since $X_t \in B(0,1)$. Finally, we simply need to sample S uniformly from the intersection of the above two intervals and set $X_{t+1} = X_t + SD$.

Appendix B
Rounding implementation

The Rounding algorithm [15], [16], [18] is a preprocessing method devised to improve the mixing time of the Hit-and-Run Markov chain. Essentially, the mixing time tends to be very high when a convex body $W \subset \mathbb{R}^d$ is very elongated into some direction. In order to counter this problem, the rounding procedure looks for a linear transformation $T : \mathbb{R}^d \to \mathbb{R}^d$ for which the image $T(W)$ is "rounder", i.e. evenly elongated into all directions. With this, Hit-and-Run can be run over $T(W)$ and the final sampling over W can be retrieved via T^{-1}.

First, let's see how the rounding transformation T affects the hit-and-run chain generation. Let $X_0 \in W$ be the chain's starting point, and let's define $Y_0 = T(X_0) \in T(W)$. The usual Hit-and-Run algorithm over $T(W)$ gives rise to a chain $\{Y_t\}$, which is incrementally defined by $Y_{t+1} = Y_t + s_{t+1} D_{t+1}$. By applying T^{-1} on the previous equation, and setting $X_t = T^{-1}(Y_t)$, we obtain a revised version of the Hit-and-Run update rule:

$$X_{t+1} = X_t + s_{t+1} T^{-1} D_{t+1} \tag{10}$$

Now, all it remains is how to compute T^{-1}. For this, we follow the algorithm described in [15], [16]. In general terms, this algorithm finds an approximation of the minimum volume ellipsoid $\mathcal{E}$ containing W. The rounding transformation can then be chosen as any linear transformation taking $\mathcal{E}$ into a unit-radius ball. Implementation details can be found on algorithm 2.

As a last remark, this algorithm assumes that the convex body W possesses a *separation oracle*; in other words, for any point $x \notin W$, we can find a hyperplane $H(x) = \{y : b_x + w_x^T y = 0\}$ such that $W \subset H(x)^+ = \{y : b_x + w_x^T y \geq 0\}$ and $x \in H(x)^- = \{y : b_x + w_x^T y \leq 0\}$. In the particular case of $W = \{x : Ax \geq 0\} \cap B(0,1)$, $H(x)$ is given by:

$$H(x) = \begin{cases} \{y : -1 + \frac{1}{\|x\|} x^T y = 0\}, \text{ if } \|x\| > 1 \\ \{y : a_i^T y = 0\}, \text{ if } a_i^T x < 0 \end{cases} \tag{11}$$

Appendix C
SDSS Queries

Q1 (**2D**, selectivity 0.1%): $rowc \in (662.5, 702.5)$ AND $colc \in (991.5, 1053.5)$

Q2 (**2D**, 0.1%): $(rowc - 682.5)^2 + (colc - 1022.5)^2 < 29^2$

Q3 (**2D**, 0.1%): $ra \in (190, 200)$ AND $dec \in (53, 57)$

Algorithm 2 Rounding algorithm

Input: convex body $W \subset \mathbb{R}^d$, any $R > 0$ satisfying $W \subset B(0, R)$

Output: T^{-1}, the inverse of the rounding transformation

```
z ← 0, P ← (1/R) I_d
converged ← false
while not converged do
    converged ← true
    if z ∉ W then
        converged ← false
        H ← get_separating_hyperplane(W, z)
        z, P ← ellipsoid_method_update(z, P, H)
    else
        Q, D ← eigendecomposition(P)
        a_k^± ← z ± (1/(d+1)) q_k
        if any a_k^± ∉ W then
            converged ← false
            H ← get_separating_hyperplane(W, a_k^±)
            z, P ← ellipsoid_method_update(z, P, H)
        end if
    end if
end while
return Cholesky(P)
```

Algorithm 3 Ellipsoid Method update

Input: ellipsoid $\mathcal{E}(z, P)$ in $\mathbb{R}^d$, cutting hyperplane $H(b, w)$

Output: The minimum volume ellipsoid containing $\mathcal{E} \cap H^-$

```
α ← (b + w^T z) / sqrt(w^T P w)
z' ← z − ((dα − 1)/(d+1)) (P w / (w^T P w))
P' ← (d^2(1−α^2)/(d^2−1)) ( P − (2(1−dα)/((d+1)(1−α))) P w w^T P )
return E(z', P')
```

Q4 (**2D**, 0.1%): $rowv^2 + colv^2 > 0.5^2$

Q5 (**4D**, 0.01%): $(rowc - 682.5)^2 + (colc - 1022.5)^2 < 90^2$ AND $ra \in (180, 210)$ AND $dec \in (50, 60)$

Q6 (**6D**, 0.01%): $(rowc - 682.5)^2 + (colc - 1022.5)^2 < 280^2$ AND $ra \in (150, 240)$ AND $dec \in (40, 70)$ AND $rowv^2 + colv^2 > 0.2^2$

Q7 (**4D**, 7.8%) : $x_1 > (1.35 + 0.25 * x_2)$ AND $x_3 + 2.5 * \log(2 * 3.1415 * petror50_r^2) < 23.3$

Q8 (**7D**, 5.5%): $(dered_r - dered_i) < 2$ AND $cmodelmag_i - extinction_i \in [17.5, 19.9]$ AND $(dered_r - dered_i) - (dered_g - dered_r)/8. > 0.55$ AND $fiber2mag_i < 21.7$ AND $devrad_i < 20.$ AND $dered_i < 19.86 + 1.60 * ((dered_r - dered_i) - (dered_g - dered_r)/8. - 0.80)$

Q9 (**5D**, 1.5%): $u - g < 0.4$ AND $g - r < 0.7$ AND $r - i > 0.4$ AND $i - z > 0.4$

Q10 (**5D**, 0.5%): $(g <= 22)$ AND $(u - g \in [-0.27, 0.71])$ AND $(g - r \in [-0.24, 0.35])$ AND $(r - i \in [-0.27, 0.57])$ AND $(i - z \in [-0.35, 0.70])$

Q11 (**5D**, 0.1%): $((u - g > 2.0)$ OR $(u > 22.3))$ AND $(i \in [0, 19])$ AND $(g - r > 1.0)$ AND $((r - i < 0.08 + 0.42 * (g - r - 0.96))$ OR $(g - r > 2.26))$ AND $(i - z < 0.25)$