Nearest Neighbor Classifiers over Incomplete Information: From Certain Answers to Certain Predictions *

Bojan Karlaš*,†, Peng Li*,‡, Renzhi Wu‡, Nezihe Merve Gürel†,
Xu Chu‡, Wentao Wu§, Ce Zhang†
†ETH Zurich, ‡Georgia Institute of Technology, §Microsoft Research

Abstract

Machine learning (ML) applications have been thriving recently, largely attributed to the increasing availability of data. However, inconsistency and incomplete information are ubiquitous in real-world datasets, and their impact on ML applications remains elusive. In this paper, we present a formal study of this impact by extending the notion of *Certain Answers for Codd tables*, which has been explored by the database research community for decades, into the field of machine learning. Specifically, we focus on classification problems and propose the notion of *"Certain Predictions" (CP)* — a test data example can be *certainly predicted (CP'ed)* if *all* possible classifiers trained on top of all possible worlds induced by the incompleteness of data would yield the *same* prediction. We study two fundamental CP queries: (Q1) *checking query* that determines whether a data example can be CP'ed; and (Q2) *counting query* that computes the number of classifiers that support a particular prediction (i.e., label). Given that general solutions to CP queries are, not surprisingly, hard without assumption over the type of classifier, we further present a case study in the context of nearest neighbor (NN) classifiers, where efficient solutions to CP queries can be developed — we show that it is possible to answer both queries in *linear or polynomial time* over *exponentially* many possible worlds. We demonstrate one example use case of CP in the important application of "data cleaning for machine learning (DC for ML)." We show that our proposed CPClean approach built based on CP can often significantly outperform existing techniques in terms of classification accuracy with mild manual cleaning effort.

1 Introduction

Building high-quality Machine learning (ML) applications often hinges on the availability of high-quality data. However, due to noisy inputs from manual data curation or inevitable errors from automatic data collection/generation programs, in reality, data is unfortunately seldom clean. Inconsistency and incompleteness are ubiquitous in real-world datasets, and therefore can have an impact on ML applications trained on top of them. In this paper, we focus on the question: *Can we reason about the impact of data incompleteness on the quality of ML models trained over it?*

Figure 1 illustrates one dataset with incomplete information. In this example, we have the incomplete dataset D with one missing cell (we will focus on cases in which there are *many* cells with incomplete information) — the age of Kevin is not known and therefore is set as `NULL` (@). Given an ML training algorithm $\mathcal{A}$, we can train an ML model over D, $\mathcal{A}_D$, and given a clean test example t, we can get the prediction of this ML model $\mathcal{A}_D(t)$. The focus of this paper is to understand *how much impact the incomplete information (@) has on the prediction* $\mathcal{A}_D(t)$. This question is not only of theoretical interest but can also have interesting practical implications — for example, if we know that, for a large enough number of samples of t, the incomplete information (@) does not have an impact on $\mathcal{A}_D(t)$ at all, spending the effort of cleaning or acquiring this specific piece of missing information will not change the quality of downstream ML models.

*The first two authors contribute equally to this paper and are listed alphabetically.

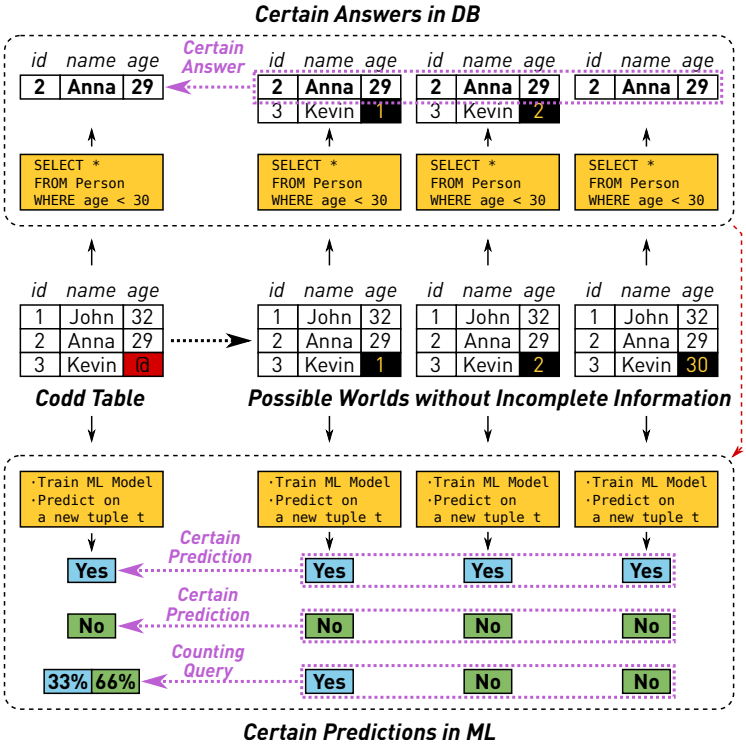

Figure 1: An illustration of the relationship between *certain answers* and *certain predictions*.

Relational Queries over Incomplete Information. This paper is inspired by the *algorithmic* and *theoretical* foundations of running *relational queries over incomplete information* [1]. In traditional database theory, there are multiple ways of representing incomplete information, starting from the Codd table, or the conditional table (c-table), all the way to the recently studied probabilistic conditional table (pc-table) [2]. Over each of these representations of incomplete information, one can define the corresponding semantics of a relational query. In this paper, we focus on the weak representation system built upon the Codd table, as illustrated in Figure 1. Given a Codd table T with constants and n variables over domain $\mathcal{D}_v$ (each variable only appears once and represents the incomplete information at the corresponding cell), it represents $|\mathcal{D}_v|^n$ many *possible worlds* $rep(T)$, and a query $\mathtt{Q}$ over T can be defined as returning the *certain answers* that *always* appear in the answer of $\mathtt{Q}$ over each possible world:

$$sure(\mathtt{Q}, T) = \cap\{\mathtt{Q}(I) | I \in rep(T)\}.$$

Another line of work with similar spirit is *consistent query answering*, which was first introduced in the seminal work by Arenas, Bertossi, and Chomicki [3]. Specifically, given an inconsistent database instance D, it defines a set of repairs $\mathcal{R}_D$, each of which is a consistent database instance. Given a query $\mathtt{Q}$, a tuple t is a *consistent answer* to $\mathtt{Q}$ if and only if t appears in *all answers* of $\mathtt{Q}$ evaluated on every consistent instance $D' \in \mathcal{R}_D$.

Both lines of work lead to a similar way of thinking in an effort to reason about data processing over incomplete information, i.e., *to reason about certain/consistent answers over all possible instantiations of incompleteness and uncertainty.*

Learning Over Incomplete Information: Certain Predictions (CP). The traditional database view provides us a powerful tool to reason about the impact of data incompleteness on downstream operations. In this paper, we take a natural step and extend this to machine learning (ML) — given a Codd table T, its $|\mathcal{D}_v|^n$ many possible worlds $rep(T)$, and an ML classifier $\mathcal{A}$, one could train one ML model $\mathcal{A}_I$ for each possible world $I \in rep(T)$. Given a test example t, we say that t can be *certainly predicted (CP'ed)* if $\forall I \in rep(T)$,

$\mathcal{A}_I(t)$ always yields the same class label, as illustrated in Figure 1. This notion of certain prediction (CP) offers a canonical view of the impact from training classifiers on top of incomplete data. Specificlly, we consider the following two CP queries:

(Q1) Checking Query — Given a test data example, determine whether it can be CP'ed or not;

(Q2) Counting Query — Given a test data example that *cannot* be CP'ed, for each possible prediction, compute the number of classifiers that *support* this prediction.

When no assumptions are made about the classifier, Q1 and Q2 are, not surprisingly, hard. In this paper, we focus on (1) developing efficient solutions to both Q1 and Q2 for a specific family of classifiers, while (2) in the meantime, trying to understand the empirical implication and application of CP to the emerging research topic of *data cleaning for machine learning*.

Efficient CP Algorithm for Nearest Neighbor Classifiers. We first study efficient algorithms to answer both CP queries for K-nearest neighbor (KNN) classifier, one of the most popular classifiers used in practice. Surprisingly, we show that, *both CP queries can be answered in polynomial time, in spite of there being exponentially many possible worlds*!

Moreover, these algorithms can be made very efficient. For example, given a Codd table with N rows and at most M possible versions for rows with missing values, we show that answering both queries only take $\mathcal{O}\left(N \cdot M \cdot (\log(N \cdot M) + K \cdot \log N)\right)$. For Q1 in the binary classification case, we can even do $\mathcal{O}(N \cdot M)$! This makes it possible to efficiently answer both queries for the KNN classifier, a result that is both *surprising* (at least to us), *new*, and *technically non-trivial*.

Discussion: Relationship with answering KNN queries over probabilistic databases. As we will see later, our result can be used to evaluate a KNN classifier over a tuple-independent database, in its standard semantics [4–6]. Thus we hope to draw the reader's attention to an interesting line of work of evaluating KNN *queries* over a probabilistic database in which the user wants the system to return the probability of a given (in our setting, training) tuple that is in the top-K list of a query. Despite the similarity of the naming and the underlying data model, we focus on a different problem in this paper as we care about the result of a KNN *classifier* instead of a KNN *query*. Our algorithm is very different and heavily relies on the structure of the classifier.

Applications to Data Cleaning for Machine Learning. The above result is not only of theoretical interest, but also has an interesting empirical implication — *intuitively, the notion of CP provides us a way to measure the relative importance of different variables in the Codd table to the downstream classification accuracy*. Inspired by this intuition, we study the efficacy of CP in the imporant application of "data cleaning for machine learning (DC for ML)" [7, 8]. Based on the CP framework, we develop a novel algorithm CPClean that prioritizes manual cleaning efforts given a dirty dataset.

Data cleaning (DC) is often an important prerequisite step in the entire pipeline of an ML application. Unfortunately, most existing work considers DC as a standalone exercise without considering its impact on downstream ML applications (exceptions include exciting seminal work such as ActiveClean [8] and BoostClean [7]). Studies have shown that such *oblivious* data cleaning may not necessarily improve downstream ML models' performance [9]; worse yet, it can sometimes even degrade ML models' performance due to Simpson's paradox [8]. We propose a novel "DC for ML" framework built on top of certain predictions. In the following discussion, we assume a standard setting for building ML models, where we are given a training set D_{train} and a validation set D_{val} that are drawn independently from the same underlying data distribution. We assume that D_{train} may contain missing information whereas D_{val} is complete.

The intuition of our framework is as follows. When the validation set is sufficiently large, if Q1 returns `true` for every data example t in D_{val}, then with high probability cleaning D_{train} will not have impact on the model accuracy. In this case we can immediately finish without any human cleaning effort. Otherwise, some data examples cannot be CP'ed, and our goal is then to clean the data such that all these examples can be CP'ed. *Why is this sufficient*? The key observation is that, as long as a tuple t can be CP'ed, the prediction will remain the same regardless of further cleaning efforts. That is, even if we clean the whole D_{train}, the

prediction for t (made by the classifier using the *clean* D_{train}) will remain the same, simply because the final clean version is one of the possible worlds of D_{train} that has been included in the definition of CP!

To *minimize* the number of tuples in D_{train} being cleaned until all data examples in D_{val} are CP'ed, we further propose a novel optimization algorithm based on the principle of *sequential information maximization* [10], exploiting the counts in Q2 for each example in D_{val} that cannot be certainly predicted. The optimization algorithm is *iterative*: Each time we pick the next example in D_{train} (to be cleaned) based on its potential impact on the "degree of certainty" of D_{train} after cleaning (see Section 4.1 for more details).

Summary of Contributions In summary, this paper makes the following contributions:

(C1) We propose *certain predictions*, as well as its two fundamental queries/primitives (checking and counting), as a tool to study the impact of incomplete data on training ML models.

(C2) We propose efficient solutions to the two fundamental CC queries for nearest neighbor classifiers, despite the hardness of these two queries in general.

(C3) We propose a novel "DC for ML" approach, CPClean, built on top of the CP primitives that significantly outperforms existing work in terms of classification accuracy, with mild manual cleaning effort.

Moving Forward Just like the study of consistent query answering that focuses on specific subfamilies of queries, in this paper we have focused on a specific type of classifier, namely the KNN classifier, in the CP framework. This allows us to design efficient algorithms specific to this workload. In the future, it is interesting to extend our study to a more diverse range of classifiers — either to develop efficient exact algorithms or to explore efficient approximation algorithms. It is also interesting to extend our CP-based data cleaning framework to more types of classifiers.

Paper Organization This paper is organized as follows. We formalize the notion of certain predictions, as well as the two primitive queries Q1 and Q2 (Section 2). We then propose efficient algorithms in the context of nearest neighbor classifiers (Section 3). We follow up by proposing our novel "DC for ML" framework exploiting CP (Section 4). We report evaluation results in Section 5, summarize related work in Section 6, and conclude the paper in Section 7.

2 Certain Prediction (CP)

In this section, we describe the *certain prediction* (CP) framework, which is a natural extension of the notion of *certain answer* for query processing over Codd tables [1] to machine learning. We first describe our data model and then introduce two CP queries.

Data Model We focus on standard supervised ML settings:

1. Feature Space $\mathcal{X}$: without loss of generality, we assume that every data example is drawn from a domain $\mathcal{X} = \mathbb{D}^d$, i.e., a d dimensional space of data type $\mathbb{D}$.
2. Label Space $\mathcal{Y}$: we assume that each data example can be classified into one of the labels in $\mathcal{Y}$.
3. Training Set $D_{train} \subseteq \mathcal{X} \times \mathcal{Y}$ is drawn from an *unknown* distribution $\mathcal{P}_{\mathcal{X},\mathcal{Y}}$.
4. Test Set $D_{test} \subseteq \mathcal{X}$ (Validation Set D_{val}) is drawn from the marginal distribution $\mathcal{P}_{\mathcal{X}}$ of the joint distribution $\mathcal{P}_{\mathcal{X},\mathcal{Y}}$.
5. Training Algorithm $\mathcal{A}$: A training algorithm $\mathcal{A}$ is a functional that maps a given training set D_{train} to a function $\mathcal{A}_{D_{train}} : \mathcal{X} \mapsto \mathcal{Y}$. Given a test example $t \in D_{test}$, $\mathcal{A}_{D_{train}}(t)$ returns the prediction of the trained classifier on the test example t.

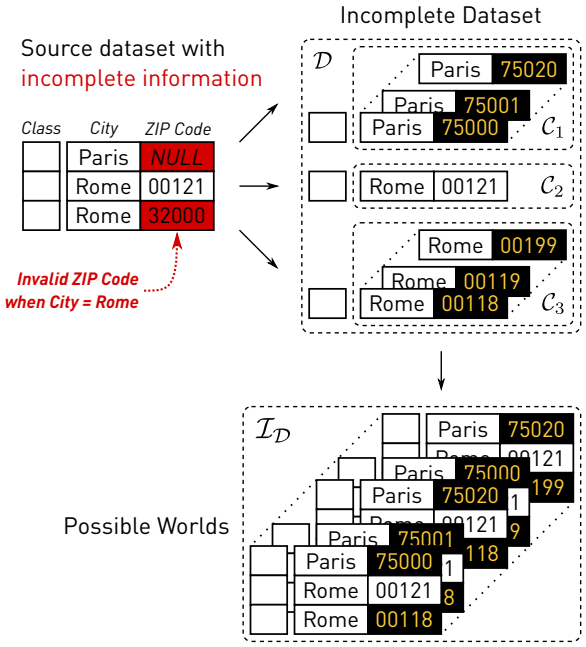

Figure 2: Example of a dataset with incomplete information, its representation as an *incomplete dataset*, and the induced set of *possible worlds*.

Incomplete Information in the Training Set In this paper, we focus on the case in which there is incomplete information in the training set. We define an *incomplete training set* as follows.

Our definition of an incomplete training set is very similar to a block tuple-independent probabilistic database [2]. However, we do assume that there is no uncertainty on the label and we do not have access to the probability distribution of each tuple.

Definition 1 (Incomplete Dataset)**.** *An* incomplete dataset

$$\mathcal{D} = \{(\mathcal{C}_i, y_i) : i = 1, ..., N\}$$

is a finite set of N pairs where each $\mathcal{C}_i = \{x_{i,1}, x_{i,2}, ...\} \subset \mathcal{X}$ is a finite number of possible feature vectors of the i-th data example and each $y_i \in \mathcal{Y}$ is its corresponding class label.

According to the semantics of $\mathcal{D}$, the i-th data example can take any of the values from its corresponding *candidate set* $\mathcal{C}_i$. The space of all possible ways to assign values to all data points in $\mathcal{D}$ is captured by the notion of possible worlds. Similar to a block tuple-independent probabilistic database, an incomplete dataset can define a set of *possible worlds*, each of which is a dataset without incomplete information.

Definition 2 (Possible Worlds)**.** *Let $\mathcal{D} = \{(\mathcal{C}_i, y_i) : i = 1, ..., N\}$ be an incomplete dataset. We define the set of possible worlds $\mathcal{I}_\mathcal{D}$, given the incomplete dataset $\mathcal{D}$, as*

$$\mathcal{I}_\mathcal{D} \;=\; \{D = \{(x_i', y_i')\} : |D| = |\mathcal{D}| \wedge \forall i.\; x_i' \in \mathcal{C}_i \wedge y_i' = y_i\}\,.$$

In other words, a *possible world* represents one complete dataset D that is generated from $\mathcal{D}$ by replacing every candidate set $\mathcal{C}_i$ with one of its candidates $x_j \in \mathcal{C}_i$. The set of all distinct datasets that we can generate in this way is referred to as the *set of possible worlds*. If we assume that $\mathcal{D}$ has N data points and the size of each $\mathcal{C}_i$ is bounded by M, we can see $|\mathcal{I}_\mathcal{D}| = \mathcal{O}(M^N)$.

Figure 2 provides an example of these concepts. As we can see, our definition of incomplete dataset can represent both possible values for missing cells and possible repairs for cells that are considered to be potentially incorrect.

Connections to Data Cleaning. In this paper, we use data cleaning as one application to illustrate the practical implication of the CP framework. In this setting, each possible world can be thought of as one possible *data repair* of the dirty/incomplete data. These repairs can be generated in an arbitrary way, possibly depending on the entire dataset [11], or even some external domain knowledge [12]. Attribute-level data repairs could also be generated independently and merged together with Cartesian products.

We will further apply the assumption that any given incomplete dataset $\mathcal{D}$ is *valid*. That is, for every data point i, we assume that there exists a *true value* x_i^* that is unknown to us, but is nevertheless included in the candidate set $\mathcal{C}_i$. This is a commonly used assumption in data cleaning [13], where automatic cleaning algorithms are used to generate a set of candidate repairs, and humans are then asked to pick one from the given set. We call $D_{\mathcal{D}}^*$ the *true possible world*, which contains the true value for each tuple. When $\mathcal{D}$ is clear from the context, we will also write D^*.

2.1 Certain Prediction (CP)

When we train an ML model over an incomplete dataset, we can define its semantics in a way that is very similar to how people define the semantics for data processing over probabilistic databases — we denote $\mathcal{A}_{D_i}$ as the classifier that was trained on the possible world $D_i \in \mathcal{I}_{\mathcal{D}}$. Given a test data point $t \in \mathcal{X}$, we say that it can be *certainly predicted* (CP'ed) if all classifiers trained on all different possible worlds agree on their predictions:

Definition 3 (Certain Prediction (CP))**.** *Given an incomplete dataset $\mathcal{D}$ with its set of possible worlds $\mathcal{I}_{\mathcal{D}}$ and a data point $t \in \mathcal{X}$, we say that a label $y \in \mathcal{Y}$ can be* certainly predicted *with respect to a learning algorithm $\mathcal{A}$ if and only if*

$$\forall D_i \in \mathcal{I}_{\mathcal{D}}, \mathcal{A}_{D_i}(t) = y.$$

Connections to Databases. The intuition behind this definition is rather natural from the perspective of database theory. In the context of Codd table, each `NULL` variable can take values in its domain, which in turn defines exponentially many possible worlds [1]. Checking whether a tuple is in the answer of some query `Q` is to check whether such a tuple is in the result of each possible world.

Two Primitive CP Queries Given the notion of *certain prediction*, there are two natural queries that we can ask. The query $Q1$ represents a *decision problem* that checks if a given label can be predicted in *all* possible worlds. The query $Q2$ is an extension of that and represents a *counting problem* that returns the number of possible worlds that support each prediction outcome. Figure 3 illustrates both queries and we formally define them as follows.

Definition 4 (Q1: Checking)**.** *Given a data point $t \in \mathcal{X}$, an incomplete dataset $\mathcal{D}$ and a class label $y \in \mathcal{Y}$, we define a query that checks* if all possible world *permits y to be predicted:*

$$Q1(\mathcal{D}, t, y) := \begin{cases} \texttt{true}, & \text{if } \forall D_i \in \mathcal{I}_{\mathcal{D}}, \mathcal{A}_{D_i}(t) = y; \\ \texttt{false}, & \text{otherwise.} \end{cases}$$

Definition 5 (Q2: Counting)**.** *Given a data point $t \in \mathcal{X}$, an incomplete dataset $\mathcal{D}$ and a class label $y \in \mathcal{Y}$, we define a query that returns the* number of possible worlds *that permit y to be predicted:*

$$Q2(\mathcal{D}, t, y) := |\{D_i \in \mathcal{I}_{\mathcal{D}} : \mathcal{A}_{D_i}(T) = y\}|.$$

Computational Challenge. If we do not make any assumption about the learning algorithm $\mathcal{A}$, we have no way of determining the predicted label $y = \mathcal{A}_{D_i}(t)$ except for running the algorithm on the training dataset. Therefore, for a general classifier treated as a black box, answering both $Q1$ and $Q2$ requires us to apply a brute-force approach that iterates over each $D_i \in \mathcal{I}_{\mathcal{D}}$, produces $\mathcal{A}_{D_i}$, and predicts the label. Given an incomplete dataset with N data examples each of which has M clean candidates, the computational cost of this naive algorithm for both queries would thus be $\mathcal{O}(M^N)$.

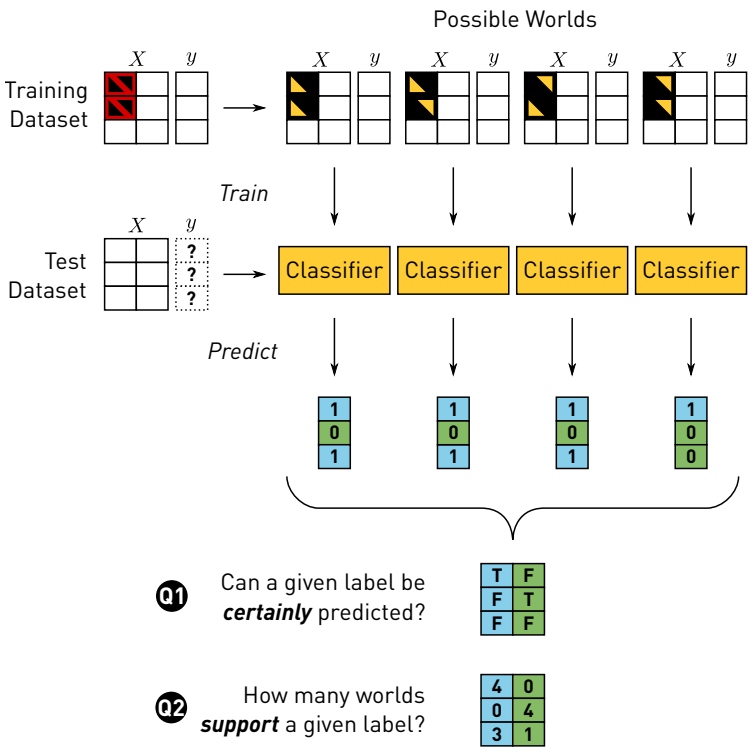

Figure 3: Illustration of certain prediction, and two queries: checking query (Q1) and counting query (Q2).

K	$\lvert\mathcal{Y}\rvert$	Query	Alg.	Complexity in $O(-)$	Section
1	2	Q1/Q2	SS	$NM \log NM$	3.1.2
K	2	Q1	MM	NM	3.2
K	$\lvert\mathcal{Y}\rvert$	Q1/Q2	SS	$NM(\log(NM) + K^2 \log N)$	3.1.3

Figure 4: Summary of results (K and $|\mathcal{Y}|$ are constants).

This is not surprising. However, as we will see later in this paper, for certain types of classifiers, such as K-Nearest Neighbor classifiers, we are able to design efficient algorithms for both queries.

Connections to Probabilistic Databases. Our definition of certain prediction has strong connection to the theory of probabilistic database [2] — in fact, Q2 can be seen as a natural definition of evaluating an ML classifier over a block tuple-independent probabilistic database with uniform prior.

Nevertheless, unlike traditional relational queries over a probabilistic database, our "query" is an ML model that has very different structure. As a result, despite the fact that we are inspired by many seminal works in probabilistic database [4–6], they are not applicable to our settings and we need to develop new techniques.

Connections to Data Cleaning. It is easy to see that, if $Q1$ returns `true` on a test example t, obtaining more information (by cleaning) for the original training set will not change the prediction on t at all! This is because the true possible world D^* is one of the possible worlds in $\mathcal{I}_\mathcal{D}$. Given a large enough test set, if Q_1 returns `true` for all test examples, cleaning the training set in this case might not improve the quality of ML models at all!

Of course, in practice, it is unlikely that all test examples can be CP'ed. In this more realistic case, Q_2 provides a "softer" way than Q_1 to measure the *degree of certainty/impact*. As we will see later, we can use this as a principled proxy of the impact of data cleaning on downstream ML models, and design efficient algorithms to prioritize which uncertain cell to clean in the training set.

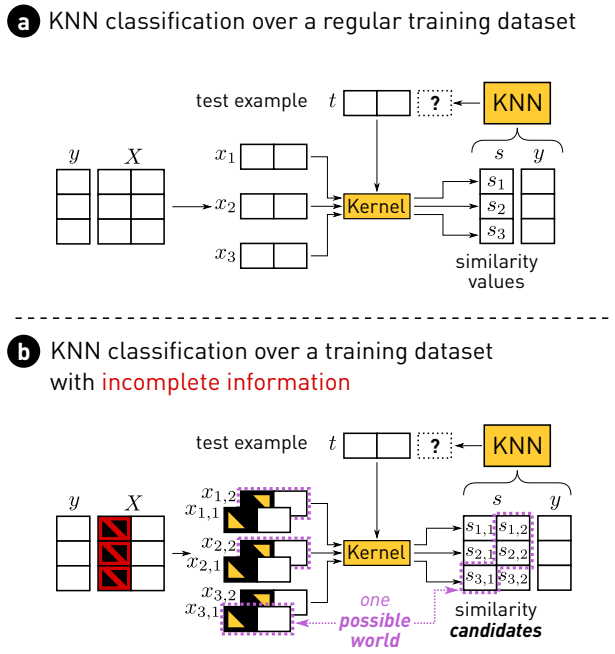

Figure 5: Illustration of KNN classifiers.

3 Efficient Solutions for CP Queries

Given our definition of certain prediction, not surprisingly, both queries are hard if we do not assume any structure of the classifier. In this section, we focus on a specific classifier that is popularly used in practice, namely the K-Nearest Neighbor (KNN) classifier. As we will see, for a KNN classifier, we are able to answer both CP queries in *polynomial* time, even though we are reasoning over *exponentially* many possible worlds!

K-Nearest Neighbor Classifiers. A textbook KNN classifier works in the following way, as illustrated in Figure 5(a): Given a training set $D = \{(x_i, y_i)\}$ and a test example t, we first calculate the similarity between t and each x_i: $s_i = \kappa(x_i, t)$. This similarity can be calculated using different kernel functions κ such as linear kernel, RBF kernel, etc. Given all these similarity scores $\{s_i\}$, we pick the top K training examples with the largest similarity score: $x_{\sigma_1}, ..., x_{\sigma_K}$ along with corresponding labels $\{y_{\sigma_i}\}_{i\in[K]}$. We then take the majority label among $\{y_{\sigma_i}\}_{i\in[K]}$ and return it as the prediction for the test example t.

Summary of Results. In this paper, we focus on designing efficient algorithms to support a KNN classifier for both CP queries. In general, all these results are based on two algorithms, namely SS (SortScan) and MM (MinMax). SS is a generic algorithm that can be used to answer both queries, while MM can only be used to answer $Q1$. However, on the other hand, MM permits lower complexity than SS when applicable. Figure 4 summarizes the result.

Structure of This Section. In Section 3.1 we will focus on the SS algorithm as it is more generic. We will explain a simplified version of the SS algorithm for the special case ($K = 1$, $|\mathcal{Y}| = 2$) in greater details as it conveys the intuition behind this algorithm. We will follow by describing the SS algorithm in its general form. We will summarize the MM algorithm in Section 3.2, which can be significantly more efficient than SS in some cases, but leave the full details to the appendix.

3.1 SS Algorithm

We now describe the SS algorithm. The idea behind SS is that we can calculate the similarity between all candidates $\cup_i \mathcal{C}_i$ in an incomplete dataset and a test example t. Without loss of generaility, assume that $|\mathcal{C}_i| = M$, this leads to $N \times M$ similarity scores $s_{i,j}$. We can then sort and scan these similarity scores.

The core of the SS algorithm is a dynamic programming procedure. We will first describe a set of basic building blocks of this problem, and then introduce a simplified version of SS for the special case of $K = 1$ and $|\mathcal{Y}| = 2$, to explain the intuition behind SS. We follow this by the general version of the SS algorithm.

3.1.1 Two Building Blocks

In our problem, we can construct two building blocks efficiently. We start by articulating the settings precisely. In the next section, we will use these two building blocks for our SS algorithm.

Setup We are given an incomplete dataset $\mathcal{D} = \{(\mathcal{C}_i, y_i)\}$. Without loss of generality, we assume that each $\mathcal{C}_i$ only contains M elements, i.e., $|\mathcal{C}_i| = M$. We call $\mathcal{C}_i = \{x_{i,j}\}_{j\in[M]}$ the i^{th} incomplete data example, and $x_{i,j}$ the j^{th} candidate value for the i^{th} incomplete data example. This defines M^N many possible worlds:

$$\mathcal{I}_{\mathcal{D}} = \{D = \{(x_i^D, y_i^D)\} : |D| = |\mathcal{D}| \wedge y_i^D = y_i \wedge x_i^D \in \mathcal{C}_i\}.$$

We use $x_{i,j_{i,D}}$ to denote the candidate value for the i^{th} data point in D. Given a test example t, we can calculate the similarity between each candidate value $x_{i,j}$ and t: $s_{i,j} = \kappa(x_{i,j}, t)$. We call these values *similarity candidates*, as shown in Figure 5 (b). We assume that there are no ties in these similarities scores (we can always break a tie by favoring a smaller i and j or a pre-defined random order).

Furthermore, given a candidate value $x_{i,j}$, we count, for each candidate set, how many candidate values are less similar to the test example than $x_{i,j}$. This gives us what we call the *similarity tally* α. For each candidate set $\mathcal{C}_n$, we have

$$\alpha_{i,j}[n] = \sum\nolimits_{m=1}^{M} \mathbb{I}[s_{n,m} \leq s_{i,j}].$$

Example 1. *In Figure 6 we can see an example of a similarity tally $\alpha_{2,2}$ with respect to the data point $x_{2,2}$. For i^{th} incomplete data example, it contains the number of candidate values $x_{i,j} \in \mathcal{C}_i$ that have the similarity value no greater than $s_{2,2}$. Visually, in Figure 6, this represents all the candidates that lie left of the vertical yellow line. We can see that only one candidate from $\mathcal{C}_1$, two candidates from $\mathcal{C}_2$, and none of the candidates from $\mathcal{C}_3$ satisfy this property. This gives us $\alpha_{2,2}[1] = 1$, $\alpha_{2,2}[2] = 2$, and $\alpha_{2,2}[3] = 0$.*

KNN over Possible World D Given one possible world D, running a KNN classifier to get the prediction for a test example t involves multiple stages. First, we obtain *Top-K Set*, the set of K examples in D that are in the K-nearest neighbor set

$$Top(K, D, t) \subseteq [N],$$

which has the following property

$$|Top(K, D, t)| = K,$$

$$\forall i, i' \in [N]. \quad i \in Top(K, D, t) \wedge i' \notin Top(K, D, t) \implies s_{i,j_{i,D}} > s_{i',j_{i',D}}.$$

Given the top-K set, we then tally the corresponding labels by counting how many examples in the top-K set support a given label. We call it the *label tally* γ^D:

$$\gamma^D \in \mathbb{N}^{|\mathcal{Y}|} : \gamma_l^D = \sum\nolimits_{i\in Top(K,D,t)} \mathbb{I}[l = y_i].$$

Finally, we pick the label with the largest count:

$$y_D^* = \arg\max_l \gamma_l^D.$$

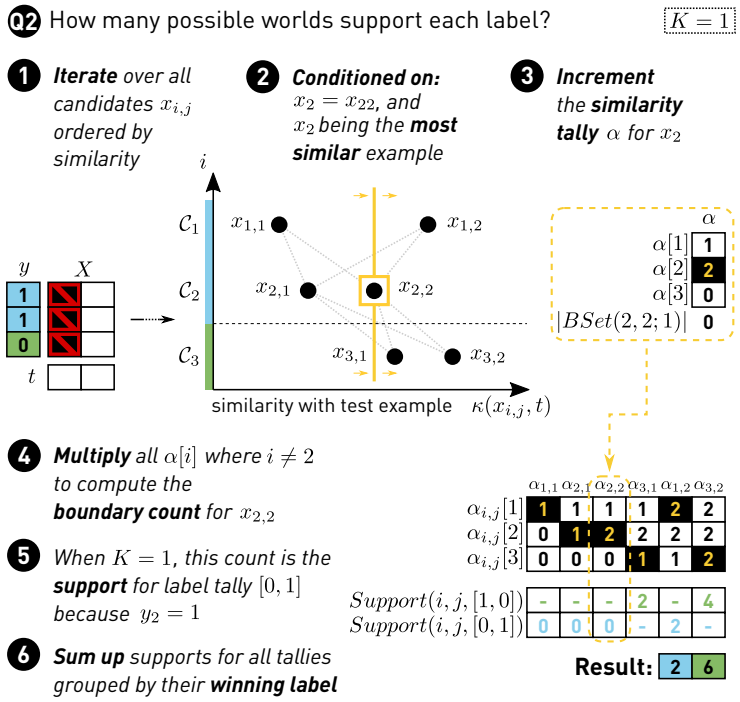

Figure 6: Illustration of SS when $K = 1$ for $Q2$.

Example 2. *For $K = 1$, the Top-K Set contains only one element x_i which is most similar to t. The label tally then is a $|\mathcal{Y}|$-dimensional binary vector with all elements being equal to zero except for the element corresponding to the label y_i being equal to one. Clearly, there are $|\mathcal{Y}|$ possible such label tally vectors.*

Building Block 1: Boundary Set The first building block answers the following question: *Out of all possible worlds that picked the value $x_{i,j}$ for $\mathcal{C}_i$, how many of them have $x_{i,j}$ as the least similar item in the Top-K set?* We call all possible worlds that satisfy this condition the *Boundary Set* of $x_{i,j}$:

$$BSet(i, j; K) = \{D : j_{i,D} = j \wedge i \in Top(K, D, t) \wedge i \notin Top(K-1, D, t)\}.$$

We call the size of the boundary set the *Boundary Count*.

We can enumerate all $\binom{N}{(K-1)}$ possible configurations of the top-(K-1) set to compute the boundary count. Specifically, let $\mathcal{S}(K-1, [N])$ be all subsets of $[N]$ with size $K-1$. We have

$$|BSet(i, j; K)| = \sum_{\substack{S \in \mathcal{S}(K-1,[N]) \\ i \notin S}} \left(\prod_{n \notin S} \alpha_{i,j}[n] \right) \cdot \left(\prod_{n \in S} (M - \alpha_{i,j}[n]) \right).$$

The idea behind this is the following — we enumerate all possible settings of the top-(K-1) set: $\mathcal{S}(K-1, [N])$. For each specific top-(K-1) setting S, every candidate set in S needs to pick a value that is more similar than $x_{i,j}$, while every candidate set not in S needs to pick a value that is less similar than $x_{i,j}$. Since the choices of value between different candidate sets are independent, we can calculate this by multiplying different entries of the similarity tally vector α.

We observe that calculating the boundary count for a value $x_{i,j}$ can be efficient when K is small. For example, if we use a 1-NN classifier, the only S that we consider is the empty set, and thus, the boundary count merely equals $\prod_{n \in [N], n \neq i} \alpha_{i,j}[n]$.

Example 3. *We can see this, in Figure 6 from Step 3 to Step 4, where the size of the boundary set* $|BSet(2,2;1)|$ *is computed as the product over elements of* α*, excluding* $\alpha[2]$*. Here, the boundary set for* $x_{2,2}$ *is actually empty. This happens because both candidates from* $\mathcal{C}_3$ *are more similar to* t *than* $x_{2,2}$ *is, that is,* $\alpha_{2,2}[3] = 0$*. Consequently, since every possible world must contain one element from* $\mathcal{C}_3$*, we can see that* $x_{2,2}$ *will never be in the Top-*1*, which is why its boundary set contains zero elements.*

If we had tried to construct the boundary set for $x_{3,1}$*, we would have seen that it contains two possible worlds. One contains* $x_{2,1}$ *and the other contains* $x_{2,2}$*, because both are less similar to* t *than* $x_{3,1}$ *is, so they cannot interfere with its Top-*1 *position. On the other hand, both possible worlds have to contain* $x_{1,1}$ *because selecting* $x_{1,2}$ *would prevent* $x_{3,1}$ *from being the Top-*1 *example.*

Building Block 2: Label Support To get the prediction of a KNN classifier, we can reason about the label tally vector γ, and not necessarily the specific configurations of the top-K set. It answers the following question: *Given a specific configuration of the label tally vector* γ*, how many possible worlds in the boundary set of* $x_{i,j}$ *support this* γ*?* We call this the *Support* of the label tally vector γ:

$$Support(i,j,\gamma) = |\{D : \gamma^D = \gamma \wedge D \in BSet(i,j;K)\}|.$$

Example 4. *For example, when* $K = 3$ *and* $|\mathcal{Y}| = 2$*, we have* 4 *possible label tallies:* $\gamma \in \{[0,3],[1,2],[2,1],[3,0]\}$*. Each tally defines a distinct partition of the boundary set of* $x_{i,j}$ *and the size of this partition is the support for that tally. Note that one of these tallies always has support 0, which happens when* $\gamma_l = 0$ *for the label* $l = y_i$*, thus excluding* $x_{i,j}$ *from the top-*K *set.*

For $K = 1$*, a label tally can only have one non-zero value that is equal to* 1 *only for a single label* l*. Therefore, all the elements in the boundary set of* $x_{i,j}$ *can support only one label tally vector that has* $\gamma_l = 1$ *where* $l = y_i$*. This label tally vector will always have the support equal to the boundary count of* $x_{i,j}$*.*

Calculating the support can be done with dynamic programming. First, we can partition the whole incomplete dataset into $|\mathcal{Y}|$ many subsets, each of which only contains incomplete data points (candidate sets) of the same label $l \in |\mathcal{Y}|$:

$$\mathcal{D}_l = \{(\mathcal{C}_i, y_i) : y_i = l \wedge (\mathcal{C}_i, y_i) \in \mathcal{D}\}.$$

Clearly, if we want a possible world D that supports the label tally vector γ, its top-K set needs to have γ_1 candidate sets from $\mathcal{D}_1$, γ_2 candidate sets from $\mathcal{D}_2$, and so on. *Given that* $x_{i,j}$ *is on the boundry, how many ways do we have to pick* γ_l *many candidate sets from* $\mathcal{D}_l$ *in the top-K set?* We can represent this value as $C_l^{i,j}(\gamma_l, N)$, with the following recursive structure:

$$C_l^{i,j}(c,n) = \begin{cases} C_l^{i,j}(c,n-1), & \text{if } y_n \neq l, \\ C_l^{i,j}(c-1,n-1), & \text{if } x_n = x_i, \text{otherwise} \\ \alpha_{i,j}[n] \cdot C_l^{i,j}(c,n-1) + (M - \alpha_{i,j}[n]) \cdot C_l^{i,j}(c-1,n-1). \end{cases}$$

This recursion defines a process in which one scans all candidate sets from $(\mathcal{C}_1, y_1)$ to $(\mathcal{C}_N, y_N)$. At candidate set $(\mathcal{C}_n, y_n)$:

1. If y_n is not equal to our target label l, the candidate set $(\mathcal{C}_n, y_n)$ will not have any impact on the count.
2. If x_n happens to be x_i, this will not have any impact on the count as x_i is always in the top-K set, by definition. However, this means that we have to decrement the number of available slots c.
3. Otherwise, we have two choices to make:
   (a) Put $(\mathcal{C}_n, y_n)$ into the top-K set, and there are $(M - \alpha_{i,j}[n])$ many possible candidates to choose from.
   (b) Do not put $(\mathcal{C}_n, y_n)$ into the top-K set, and there are $\alpha_{i,j}[n]$ many possible candidates to choose from.

It is clear that this recursion can be computed as a dynamic program in $\mathcal{O}(N \cdot M)$ time. This DP is defined for $c \in \{0...K\}$ which is the exact number of candidates we want to have in the top-K, and $n \in \{1...N\}$ which defines the subset of examples $x_i : i \in \{1...N\}$ we are considering. The boundary conditions of this DP are $C_l^{i,j}(-1, n) = 0$ and $C_l^{i,j}(c, 0) = 1$.

Given the result of this dynamic programming algorithm for different values of l, we can calculate the support of label tally γ:

$$Support(i, j, \gamma) = \prod_{l \in \mathcal{Y}} C_l^{i,j}(\gamma_l, N),$$

which can be computed in $\mathcal{O}(NM|\mathcal{Y}|)$.

Example 5. *If we assume the situation shown in Figure 6, we can try for example to compute the value of* $Support(3, 1, \gamma)$ *where* $\gamma = [1, 0]$. *We would have* $C_0^{3,1}(1, N) = 1$ *because* x_3 *(the subset of* D *with label 0) must be in the top-*K*, which happens only when* $x_3 = x_{3,1}$. *On the other hand we would have* $C_1^{3,1}(0, N) = 2$ *because both* x_1 *and* x_2 *(the subset of* D *with label 1) must be out of the top-*K*, which happens when* $x_1 = x_{1,1}$ *while* x_2 *can be either equal to* $x_{2,1}$ *or* $x_{2,2}$. *Their mutual product is equal to* 2*, which we can see below the tally column under* $x_{3,1}$.

3.1.2 $K = 1$, $|\mathcal{Y}| = 2$

Given the above two building blocks, it is easy to develop an algorithm for the case $K = 1$ and $|\mathcal{Y}| = 2$. In SS, we use the result of $Q2$ to answer both $Q1$ and $Q2$. Later we will introduce the MM algorithm that is dedicated to $Q1$ only.

We simply compute the number of possible worlds that support the prediction label being 1. We do this by enumerating all possible candidate values $x_{i,j}$. If this candidate has label $y_i = 1$, we count how many possible worlds have $x_{i,j}$ as the top-1 example, i.e., the boundry count of $x_{i,j}$. We have

$$Q2(\mathcal{D}, t, l) = \sum_{i \in [N]} \sum_{j \in [M]} \mathbb{I}[y_i = l] \cdot |BSet(i, j; K = 1)|,$$

which simplifies to

$$Q2(\mathcal{D}, t, l) = \sum_{i \in [N]} \sum_{j \in [M]} \mathbb{I}[y_i = l] \cdot \prod_{n \in [N], n \neq i} \alpha_{i,j}[n].$$

If we pre-compute the whole α matrix, it is clear that a naive implementation would calculate the above value in $\mathcal{O}(N^2 M)$. However, as we will see later, we can do much better.

Efficient Implementation We can design a much more efficient algorithm to calculate this value. The idea is to first sort all $x_{i,j}$ pairs by their similarity to t, $s_{i,j}$, from the smallest to the largest, and then scan them in this order. In this way, we can incrementally maintain the $\alpha_{i,j}$ vector during the scan.

Let (i, j) be the current candidate value being scanned, and (i', j') be the candidate value right before (i, j) in the sort order, we have

$$\alpha_{i,j}[n] = \begin{cases} \alpha_{i',j'}[n] + 1 & \text{if } n = i', \\ \alpha_{i',j'}[n]. \end{cases} \tag{1}$$

Therefore, we are able to compute, for each (i, j), its

$$\prod_{n \in [N], n \neq i} \alpha_{i,j}[n] \tag{2}$$

in $\mathcal{O}(1)$ time, *without pre-computing the whole* α. This will give us an algorithm with complexity $\mathcal{O}(MN \log MN)$!

Algorithm 1 Algorithm **SS** for Answering Q2 with K-NN.

Input: $\mathcal{D}$, incomplete dataset; t, target data point.
Output: r, integer vector, s.t. $r[y] = Q2(\mathcal{D}, t, y), \forall y \in \mathcal{Y}$.

```
1: s ← kernel(D, t);
2: α ← zeros(|D|);
3: r ← zeros(|Y|);
4: for all (i, j) ∈ argsort(s) do
5:     α[i] ← α[i] + 1;     // (See Equation (1))
6:     for all l ∈ Y do
7:         for all k ∈ [K] do
8:             Compute C_l^{i,j}(k, N).
9:     for all possible valid tally vectors γ ∈ Γ do
10:        y_p ← argmax(γ);
11:        Compute Support(i, j, γ) = 𝕀[γ_{y_i} ≥ 1] · ∏_{l∈Y} C_l^{i,j}(γ_l, N);
12:        r[y_p] ← r[y_p] + Support(i, j, γ);
13: return r;
```

Example 6. *In Figure 6 we depict exactly this algorithm. We iterate over the candidates $x_{i,j}$ in an order of increasing similarity with the test example t (Step 1). In each iteration we try to compute the number of possible worlds supporting $x_{i,j}$ to be the top-1 data point (Step 2). We update the tally vector α according to Equation 1 (Step 3) and multiply its elements according to Equation 2 (Step 4) to obtain the boundary cont. Since $K = 1$, the label support for the label $l = y_i$ is trivially equal to the boundary count and zero for $l \neq y_i$ (Step 5). We can see that the label 0 is supported by 2 possible worlds when $x_3 = x_{3,1}$ and 4 possible worlds when $x_3 = x_{3,2}$. On the other hand, label 1 has non-zero support only when $x_1 = x_{1,2}$. Finally, the number of possible worlds that will predict label l is obtained by summing up all the label supports in each iteration where $l = y_i$ (Step 6). For label 0 this number is $2 + 4 = 6$, and for label 1 it is $0 + 0 + 0 + 2 = 2$.*

3.1.3 $K \geq 1$, $|\mathcal{Y}| \geq 2$

In the general case, the algorithm follows a similar intuition as the case of $K = 1$ and $|\mathcal{Y}| = 2$. We enumerate each possible candidate value $x_{i,j}$. For each candidate value, we enumerate all possible values of the label tally vector; for each such vector, we compute its support. Let Γ be the set of all possible label tally vectors, we have

$$Q2(\mathcal{D}, t, l) = \sum_{i \in [N]} \sum_{j \in [M]} \sum_{\gamma \in \Gamma} \mathbb{I}[l = \arg\max(\gamma)] \cdot Support(i, j, \gamma).$$

We know that there are $|\Gamma| = \mathcal{O}\left(\binom{|\mathcal{Y}|+K-1}{K}\right)$ many possible configurations of the label tally vector, and for each of them, we can compute the support $Support(i, j, \gamma)$ in $\mathcal{O}(NM|\mathcal{Y}|)$ time. As a result, a naive implementation of the above algorithm would take $\mathcal{O}(N^2M^2|\mathcal{Y}|\binom{|\mathcal{Y}|+K-1}{K})$ time.

Efficient Implementation We can implement the above procedure in a more efficient way, as illustrated in Algorithm 1. Similar to the case of $K = 1$, we iterate over all values $x_{i,j}$ in the order of increasing similarity (line 4). This way, we are able to maintain, efficiently, the similarity tally vector $\alpha_{i,j}$ (line 5). We then pre-compute the result of $K|\mathcal{Y}|$ many dynamic programming procedures (lines 6-8), which will be used to compute the support for each possible tally vector later. We iterate over all valid label tally vectors, where a valid tally vector $\gamma \in \Gamma$ contains all integer vectors whose entries sum up to K (line 9). For each tally vector, we get its prediction y_p (line 10). We then calculate its support (line 11) and add it to the number of possible worlds with y_p as the prediction (line 12).

(Complexity) We analyze the complexity of Algorithm 1:

- The sorting procedure requires $\mathcal{O}(N \cdot M \log N \cdot M)$ steps as it sorts all elements of S.
- The outer loop iterates over $\mathcal{O}(N \cdot M)$ elements.

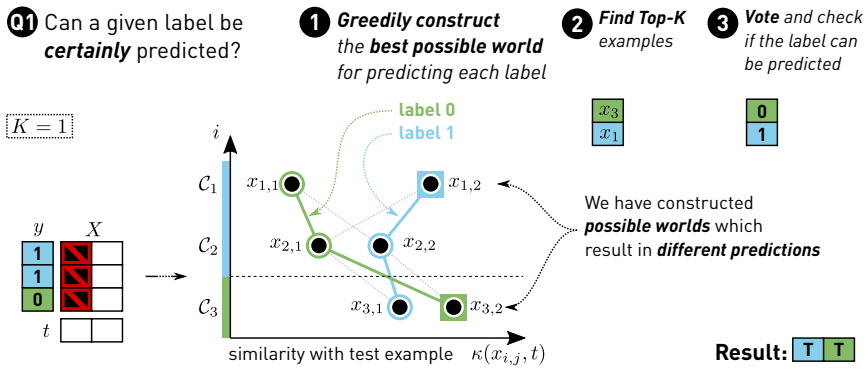

Figure 7: Illustration of MM when $K = 1$ for $Q1$.

- In each inner iteration, we need to compute $|\mathcal{Y}|$ sets of dynamic programs, each of which has a combined state space of size $N \cdot K$.
- Furthermore, in each iteration, we iterate over all possible label assignments, which requires $\mathcal{O}\left(\binom{|\mathcal{Y}|+K-1}{K}\right)$ operations.
- For each label assignment, we need $\mathcal{O}(|\mathcal{Y}|)$ multiplications.

The time complexity is therefore the sum of $\mathcal{O}(N \cdot M \log(N \cdot M))$ and $\mathcal{O}(N \cdot M \cdot (N \cdot K + |\mathcal{Y}| + \binom{|\mathcal{Y}|+K-1}{K}))$.

Further Optimizations We can make this even faster by observing that: (1) all the states relevant for each iteration of the outer loop are stored in α, and (2) between two iterations, only one element of α is updated. We can take advantage of these observations to reduce the cost of computing the dynamic program by employing divide-and-conquer. We recursively divide the elements of α into two subsets and maintain the DP result for each subset. The joint result for the two subsets is obtained by a simple sum-of-products formula with $O(K)$ complexity. We can see that this enables us to maintain a binary tree structure of DP results and in each iteration we need to update $O(\log N)$ elements. This enables us to compute the dynamic program in $O(K \log N)$ instead of $O(KN)$ time, which renders the overall complexity as $\mathcal{O}(N \cdot M \cdot (\log(N \cdot M) + K^2 \cdot \log N))$. We leave the details for the appendix.

3.2 MM Algorithm

One can do significantly better for Q_1 in certain cases. Instead of using the SS algorithm, we can develop an algorithm that deals with the binary classification case ($|\mathcal{Y}| = 2$) with time complexity $\mathcal{O}(N \cdot M + (N \log K + K))$.

This algorithm, illustrated in Figure 7, relies on a key observation that for each label l, we can greedily construct a possible world that has the best chance of predicting label l. We call this possible world the l-extreme world and construct it by selecting from each candidate set $\mathcal{C}_i$ either the candidate most similar to the test example t when $y_i = l$, or the candidate least similar to t when $y_i \neq l$. We can show that the l-extreme world predicts label l if and only if there exists a possible world that predicts label l. This means we can use it as a condition for checking the possibility of predicting label l. Since the construction of the l-extreme world can be done in $\mathcal{O}(N \cdot M)$ time, this leads us to a efficient algorithm for $Q1$.

We first describe the key idea behind the MM algorithm, and then describe the MM algorithm which is listed in Algorithm 2.

Key Idea For *binary classification* ($|\mathcal{Y}| = 2$), we have the following observation — given a possible world $D = \{(x_{i,j_{i,D}}, y_i)\}$ that produces prediction $l \in \mathcal{Y}$ with a top-K set $Top(K, D, t)$, consider a different possible

Algorithm 2 Algorithm **MM** for answering Q1 with K-NN.

Input: $\mathcal{D}$, incomplete dataset; t, target data point.
Output: r, Boolean vector, s.t. $r[y] = Q1(\mathcal{D}, t, y), \forall y \in \mathcal{Y}$.

$S \leftarrow \texttt{kernel}(\mathcal{D}, x_t)$
for all $i \in 1, ..., |\mathcal{D}|$ **do**
  $s_i^{\min} \leftarrow \min\{S_{i,j}\}_{j=1}^{M}$, $s_i^{\max} \leftarrow \max\{S_{i,j}\}_{j=1}^{M}$;
for all $l \in \mathcal{Y}$ **do**
  $s \leftarrow \texttt{zeros}(|\mathcal{D}|)$;
  for all $i \in 1, ..., |\mathcal{D}|$ **do**
    $s[i] \leftarrow s_i^{\max}$ **if** $(y_i = l)$ **else** $s_i^{\min}$;
  $I_K \leftarrow \texttt{argmax\_k}(s, K)$;
  $v \leftarrow \texttt{vote}(\{y_i : i \in I_K\})$;
  if $\texttt{argmax}(v) = l$ **then**
    $r[l] \leftarrow \texttt{true}$;
  else
    $r[l] \leftarrow \texttt{false}$;
return r;

world, which we call the *l-extreme world* of D as $E_{l,D}$. In $E_{l,D}$, we replace, for all candidate sets with $y_i = l$, the $x_{i,j_{i,D}}$ candidate in D with the candidate in the candidate set $\mathcal{C}_i$ that is most similar to the test example

$$j_{i,E_{l,D}} = \arg\max_j \kappa(x_{i,j}, t)$$

and replace, for all candidate sets with $y_i \neq l$, the $x_{i,j_{i,D}}$ candidate in D with the candidate in the candidate set $\mathcal{C}_i$ that is least similar to the test example

$$j_{i,E_{l,D}} = \arg\min_j \kappa(x_{i,j}, t).$$

We have

$$D \text{ predicts } l \implies E_{l,D} \text{ predicts } l.$$

To see why, note that (1) replacing all candidate values for candidate set whose label $y_i \neq l$ by something less similar to the test example t will only make it more likely to predict l; (2) replacing all candidate values for candidate set whose label $y_i = l$ by something more similar to t will only make it more likely to predict l.

Another powerful observation is that for all possible worlds D they all have the *same* l-extreme worlds $E_{l,D}$ since the construction of the latter only relies on the most and least similar items in each candidate sets. We can then write E_l as the l-extreme world for all possible world D. We now have

$$\exists D.\ D \text{ predicts } l \implies E_l \text{ predicts } l,$$

and, trivially

$$E_l \text{ predicts } l \implies \exists D.\ D \text{ predicts } l,$$

by simply taking $D = E_l$. As a result,

$$\exists D.\ D \text{ predicts } l \Leftrightarrow E_l \text{ predicts } l.$$

One can use this observation to check whether $Q1(\mathcal{D}, t, l)$ evaluates to true: this is equivalent to checking whether there exists any possible world D that predicts a label $l' \neq l$. To achieve this, we can simply check the l'- extreme world $E_{l'}$.

Proof in Appendix This idea might look simple and natural, however, a formal proof is actually quite engaged (e.g., without a formal proof, it is not immediately clear why this algorithm cannot handle cases in which $|\mathcal{Y}| > 2$). We leave the full, formal proof to the appendix of this paper.

Efficient Algorithm The above intuition gives us a very efficient algorithm to answer the query $Q1(\mathcal{D}, t, l)$, as illustrated in Algorithm 2. We first calculate the similarity matrix (line 1), compute the extreme similarities that we use later (lines 2-3), and then try to construct the l-extreme world for each $l \in \mathcal{Y}$ (lines 4-7). We then calculate the top-K set of the l-extreme world (line 8) and tally the labels to get the prediction in the l-extreme world (line 9).

To answer the query $Q1(\mathcal{D}, t, l)$ (lines 10-13), we check all l'-extreme worlds where $l' \neq l$ to see if any of these l'-extreme worlds predicts their corresponding l'. If yes, then $Q1(\mathcal{D}, t, l) = \texttt{false}$; otherwise, $Q1(\mathcal{D}, t, l) = \texttt{true}$.

(Complexity) We analyze the complexity of Algorithm 2 as follows:

- We first precompute the similarity matrix, as well as the minimum and maximum similarities, both of which can be done in $\mathcal{O}(N \cdot M)$ time.
- The outer loop is executed $|\mathcal{Y}|$ times.
- The optimal world construction loop (lines 6-7) is executed N times. In each iteration we retrieve the precomputed maximal or minimal values.
- The `argmax_k` function implemented as a heap requires $\mathcal{O}(N \log K)$ steps.
- The `vote` function requires $\mathcal{O}(K)$ steps. The `argmax` takes $\mathcal{O}(|\mathcal{Y}|)$, although these two steps can be implemented jointly and run in $\mathcal{O}(K)$ time.

The time complexity is therefore $\mathcal{O}\left(N \cdot M + |\mathcal{Y}| \cdot (N \log K + K)\right)$.

4 Application: Data Cleaning for ML

In this section, we show how to use the proposed CP framework to design an effective data cleaning solution, called CPClean, for the important application of data cleaning for ML. We assume as input a dirty training set $\mathcal{D}_{train}$ with unknown ground truth D^* among all possible worlds $\mathcal{I}_{\mathcal{D}}$. Our goal is to select a version D from $\mathcal{I}_{\mathcal{D}}$, such that the classifier trained on $\mathcal{A}_D$ has the same validation accuracy as the classifier trained on the ground truth world $\mathcal{A}_{D^*}$.

Cleaning Model. Given a dirty dataset $\mathcal{D} = \{(\mathcal{C}_i, y_i)\}_{i \in [N]}$, in this paper, we focus on the scenario in which the candidate set $\mathcal{C}_i$ for each data example is created by automatic data cleaning algorithms or a predefined noise model. For each uncertain data example $\mathcal{C}_i$, we can ask a human to provide its true value $x_i^* \in \mathcal{C}_i$. Our goal is to find a good strategy to prioritize which dirty examples to be cleaned. That is, a cleaning strategy of T steps can be defined as

$$\pi \in [N]^T,$$

which means that in the first iteration, we clean the example π_1 (by querying human to obtain the ground truth value of $\mathcal{C}_{\pi_1}$; in the second iteration, we clean the example π_2; and so on. Applying a cleaning strategy π will generate a *partially cleaned* dataset $\mathcal{D}_\pi$ in which all cleaned candidate sets $\mathcal{C}_{\pi_i}$ are replaced by $\{x^*_{\pi_i}\}$.

Formal Cleaning Problem Formulation. The question we need to address is *"What is a successful cleaning strategy?"* Given a validation set D_{val}, the view of CPClean is that a successful cleaning strategy π should be the one that produces a partially cleaned dataset $\mathcal{D}_\pi$ in which all validation examples $t \in D_{val}$ can be certainly predicted. In this case, picking any possible world defined by $\mathcal{D}_\pi$, i.e., $\mathcal{I}_{\mathcal{D}_\pi}$, will give us a dataset that has the same accuracy, on the validation set, as the ground truth world D^*. This can be defined precisely as follows.

We treat each candidate set $\mathcal{C}_i$ as a random variable $\mathbf{c}_i$, taking values in $\{x_{i,1}, ..., x_{i,M}\}$. We write $\mathbf{D} = \{(\mathbf{c}_i, y_i)\}_{i \in [N]}$. Given a cleaning strategy π we can define the conditional entropy of the classifier prediction on the validation set as

$$\mathcal{H}(\mathcal{A}_{\mathbf{D}}(D_{val})|\mathbf{c}_{\pi_1},...,\mathbf{c}_{\pi_T}) \coloneqq \frac{1}{|D_{val}|}\sum_{t\in D_{val}} \mathcal{H}(\mathcal{A}_{\mathbf{D}}(t)|\mathbf{c}_{\pi_1},...,\mathbf{c}_{\pi_T}). \tag{3}$$

Naturally, this gives us a principled objective for finding a "good" cleaning strategy that minimizes the human cleaning effort:

$$\begin{aligned} \min_{\pi} \quad & \dim(\pi) \\ s.t., \quad & \mathcal{H}(\mathcal{A}_{\mathbf{D}}(D_{val})|\mathbf{c}_{\pi_1}=x^*_{\pi_1},...,\mathbf{c}_{\pi_T}=x^*_{\pi_T})=0. \end{aligned}$$

If we are able to find a cleaning strategy in which

$$\mathcal{H}(\mathcal{A}_{\mathbf{D}}(D_{val})|\mathbf{c}_{\pi_1}=x^*_{\pi_1},...,\mathbf{c}_{\pi_T}=x^*_{\pi_T})=0,$$

we know that this strategy would produce a partially cleaned dataset $\mathcal{D}_\pi$ on which all validation examples can be CP'ed. Note that we can use the query $Q2$ to compute this conditional entropy:

$$\mathcal{H}(\mathcal{A}_{\mathbf{D}}(t)|\mathbf{c}_{\pi_1}=x^*_{\pi_1},...,\mathbf{c}_{\pi_T}=x^*_{\pi_T}) \;= -\sum_{l\in\mathcal{Y}} \frac{Q2(\mathcal{D}_\pi,t,y)}{|\mathcal{D}_\pi|}\log\frac{Q2(\mathcal{D}_\pi,t,y)}{|\mathcal{D}_\pi|}$$

Connections to ActiveClean. The idea of prioritizing human cleaning effort for downstream ML models is not new — ActiveClean [14] explores an idea with a similar goal. However, there are some important differences between our framework and ActiveClean. The most crucial one is that our framework relies on *consistency* of predictions instead of the *gradient*, and therefore, we do not need labels for the validation set and our algorithm can be used in ML models that cannot be trained by gradient-based methods. The KNN classifier is one such example. Since both frameworks essentially measure some notion of "local sensitivity," it is interesting future work to understand how to combine them.

4.1 The CPClean Algorithm

Finding the solution to the above objective is, not surprisingly, NP-hard [15]. In this paper, we take the view of sequential information maximization introduced by [10] and adapt the respective greedy algorithm for this problem. We first describe the algorithm, and then review the theoretical analysis of its behavior.

Principle: Sequential Information Maximization. Our goal is to find a cleaning strategy that *minimizes* the *conditional entropy* as fast as possible. An equivalent view of this is to find a cleaning strategy that *maximizes* the *mutual information* as fast as possible. While we use the view of minimizing conditional entropy in implementing the CPClean algorithm, the equivalent view of maximizing mutual information will be useful in analyzing theoretical guarantees about CPClean.

Given the current T-step cleaning strategy $\pi_1,...,\pi_T$, our goal is to greedily find the *next* data example to clean $\pi_{T+1}\in[N]$ that minimizes the entropy conditioned on the partial observation as fast as possible:

$$\pi_{T+1} = \arg\min_{i\in[N]} \mathcal{H}(\mathcal{A}_{\mathbf{D}}(D_{val})|\mathbf{c}_{\pi_1}=x^*_{\pi_1},...,\mathbf{c}_{\pi_T}=x^*_{\pi_T},\mathbf{c}_i=x^*_i).$$

Practical Estimation. The question thus becomes how to estimate

$$\mathcal{H}(\mathcal{A}_{\mathbf{D}}(D_{val})|\mathbf{c}_{\pi_1}=x^*_{\pi_1},...,\mathbf{c}_{\pi_T}=x^*_{\pi_T},\mathbf{c}_i=x^*_i)?$$

The challenge is that when we are trying to decide which example to clean, we do not know the ground truth for item i, x^*_i. As a result, we need to assume some priors on how likely each candidate value $x_{i,j}$ is the ground truth x^*_i. In practice, we find that a uniform prior already works well; this leads to the following expected value:

$$\frac{1}{M}\sum_{j\in[M]} \mathcal{H}(\mathcal{A}_{\mathbf{D}}(D_{val})|\mathbf{c}_{\pi_1}=x^*_{\pi_1},...,\mathbf{c}_{\pi_T}=x^*_{\pi_T},\mathbf{c}_i=x_{i,j}). \tag{4}$$

Algorithm 3 Algorithm CPClean.

Input: $\mathcal{D}$, incomplete training set; D_{val}, validation set.
Output: D, a dataset in $\mathcal{I}_{\mathcal{D}}$ s.t. $\mathcal{A}_D$ and $\mathcal{A}_{D^*}$ have same validation accuracy

1: $\pi \leftarrow []$
2: **for** $T = 0$ to $N - 1$ **do**
3: **if** D_{val} all CP'ed **then**
4: **break**
5: $min\_entropy \leftarrow \infty$
6: **for all** $i \in [N] \backslash \pi$ **do**
7: $entropy = \frac{1}{M} \sum_{j \in [M]} \mathcal{H}(\mathcal{A}_{\mathbf{D}}(D_{val}) | \mathbf{c}_{\pi_1} = x^*_{\pi_1}, ..., \mathbf{c}_{\pi_T} = x^*_{\pi_T}, \mathbf{c}_i = x_{i,j})$
8: **if** $entropy < min\_entropy$ **then**
9: $\pi_{T+1} \leftarrow i$, $min\_entropy \leftarrow entropy$
10: $\mathbf{x}^*_{\pi_{T+1}} \leftarrow$ obtain the ground truth of $\mathcal{C}_{\pi_{T+1}}$ by human
11: **return** Any world $D \in \mathcal{I}_{\mathcal{D}_\pi}$

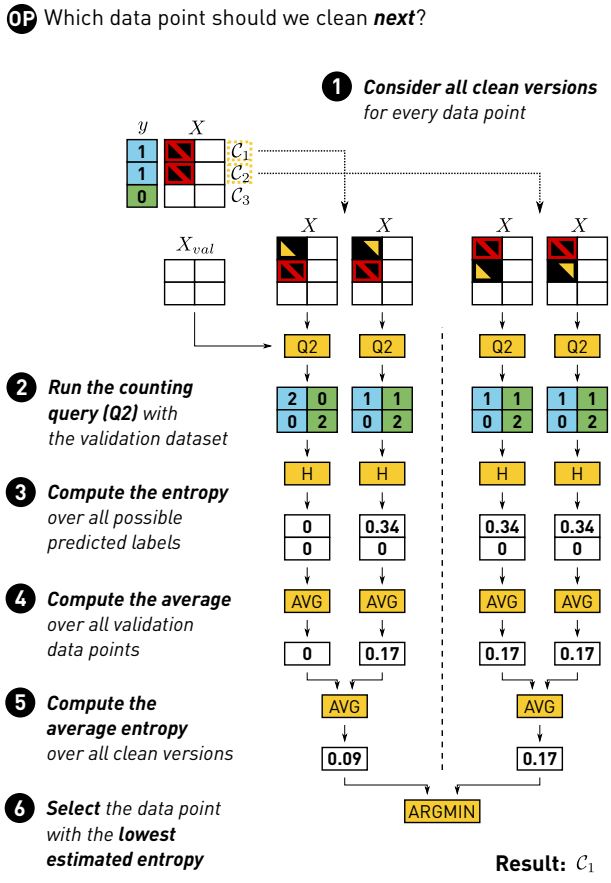

Figure 8: CPClean via sequential info. maximization.

The above term can thus be calculated by invoking the $Q2$ query.

CPClean. The pseudocode for CPClean is shown in Algorithm 3. The algorithm starts with an empty cleaning strategy (line 1). In each iteration, given the current cleaning strategy $\pi_1, ..., \pi_T$, we compute the expected value of entropy conditioned on cleaning one extra training example (lines 6-7). We select the next example to clean π_{T+1} that minimizes the entropy (lines 8-9). We then ask a human to clean the selected example

(line 10). The greedy algorithm terminates when all validation examples become CP'ed (line 3). Finally, we return any world D among all possible partially cleaned worlds $\mathcal{I}_{\mathcal{D}_\pi}$ (line 12). Since all the validation examples are CP'ed with $\mathcal{I}_{\mathcal{D}_\pi}$, classifier trained on any world in $\mathcal{I}_{\mathcal{D}_\pi}$, including the unknown ground truth world D^*, has the same validation accuracy. Therefore, $\mathcal{A}_D$ has the same validation accuracy as $\mathcal{A}_{D^*}$.

Example 7. *Figure 8 shows an example of how* CPClean *selects the next data example to clean in each iteration via sequential information maximization. Assume there are two dirty examples, $\mathcal{C}_1$ and $\mathcal{C}_2$, in the training set and each example has two candidate repairs. Therefore, there are four possible clean versions after cleaning the next data point, based on which data point is selected to be cleaned and which candidate repair is the ground truth. For example, the first table at step 1 shows the clean version after cleaning $\mathcal{C}_1$ if $x_{1,1}$ is the ground truth. Assume that we have two validation examples. We run the counting query (Q2) on each possible version w.r.t. each validation example as shown in step 2. Then we can compute the entropy of predictions on validation examples as shown in step 3 and 4. The results show that if $\mathcal{C}_1$ is selected to be cleaned, the entropy may become 0 or 0.17 depending on which candidate repair is the ground truth. We assume that each of the two candidate repairs has 50% chance to be the ground truth. Therefore, the expected entropy after cleaning $\mathcal{C}_1$ is $(0 + 0.17)/2 = 0.09$ (step 5). Similarly, we compute the expected entropy after cleaning $\mathcal{C}_2$ as 0.17. Since $\mathcal{C}_1$ has a lower expected entropy, we select $\mathcal{C}_1$ to clean.*

Complexity of CPClean. In each iteration of Algorithm 3, we need to (1) automatically select a tuple; and (2) ask human to clean the selected tuple. To select a tuple, we need to first check whether $|D_{val}|$ are all CP'ed (line 3), which invokes the $Q1$ query $O(|D_{val}|)$ times. If not all D_{val} are CP'ed, we need to compute expected value of entropy $O(N)$ times (line 6). Computing the expected value of entropy (line 7) needs to invoke the $Q2$ query $O(M|D_{val}|)$ times. Therefore, when the downstream ML model is KNN, using our SS algorithm for $Q1$ and $Q2$, the complexity for selecting a tuple at each iteration is $O(N^2M^2|D_{val}| \times (\log(MN) + K \log N))$. The quadratic complexity in tuple selection is acceptable in practice, since human involvement is generally considered to be the most time consuming part in practical data cleaning [13].

Theoretical Guarantee. The theoretical analysis of this algorithm, while resembling that of [10], is non-trivial. We provide the main theoretical analysis here and leave the proof to the appendix.

Corollary 1. *Let the optimal cleaning policy that minimizes the cleaning effort while consistently classifying the test examples be denoted by $D_{Opt} \subseteq D_{train}$ with limited cardinality t, such that*

$$D_{Opt} = \underset{D_\pi \subseteq \mathcal{D}_{train},\ |D_\pi| \leq t}{\arg\max} I(\mathcal{A}_{\mathbf{D}}(D_{val}); D_\pi).$$

The sequential information maximization strategy follows a near optimal strategy where the information gathering satisfies

$$I(\mathcal{A}_{\mathbf{D}}(D_{val}); \mathbf{c}_{\pi_1}, ..., \mathbf{c}_{\pi_T}) \geq I(\mathcal{A}_{\mathbf{D}}(D_{val}); D_{Opt})(1 - \exp(-T/\theta t'))$$

where

$$\theta = \left(\max_{v \in \mathcal{D}_{train}} I(\mathcal{A}_{\mathbf{D}}(D_{val}); v) \right)^{-1}$$
$$t' = t \min\{\log |\mathcal{Y}|, \log M\}, \quad \mathcal{Y} : \textit{label space}, \ M : |\mathcal{C}_i|.$$

The above result, similarly as in [10], suggests that data cleaning is guaranteed to achieve near-optimal information gathering up to a logarithmic factor $\min(\log |\mathcal{Y}|, \log M)$ when leveraging the sequential information strategy.

5 Experiments

We now conduct an extensive set of experiments to compare CPClean with other data cleaning approaches in the context of K-nearest neighbor classifiers.

Dataset	Error Type	#Examples	#Features	Missing rate
BabyProduct [16]	real	3042	7	11.8%
Supreme [17]	synthetic	3052	7	20%
Bank [18]	synthetic	3192	8	20%
Puma [18]	synthetic	8192	8	20%

Table 1: Datasets characteristics

5.1 Experimental Setup

Hardware and Platform. All our experiments were performed on a machine with a 2.20GHz Intel Xeon(R) Gold 5120 CPU.

Datasets. One main challenge of evaluating data cleaning solutions is the lack of datasets with ground truth, and hence most data cleaning work resort to synthetic error injection. This is especially true in the context of incomplete information: a dataset with missing values is not likely to come with ground truth. In this work, besides three datasets with synthetic errors, we manage to find one dataset with *real* missing values, where we are able to obtain the *ground truth* via manual Googling. We summarize all datasets in Table 1.

The BabyProduct dataset contains various baby products of different categories (e.g., bedding, strollers). Since the dataset was scraped from websites using Python scripts [16], many records have missing values, presumably due to extractor errors. We designed a classification task to predict whether a given baby product has a high price or low price based on other attributes (e.g. weight, brand, dimension, etc), and we selected a subset of product categories whose price difference is not so high so as to make the classification task more difficult. For records with missing brand attribute, we then perform a Google search using the product title to obtain the product brand. For example, one record titled "*Just Born Safe Sleep Collection Crib Bedding in Grey*" is missing the product brand, and a search reveals that the brand is "Just Born."

We also use three datasets (Supreme, Bank, Puma), originally with no missing values, to inject synthetic missing values. Our goal is to inject missing values in the most realistic way possible and also to ensure that the missing values can have a large impact on classification accuracy. We follow the popular "Missing Not At Random" assumption [19], where the probability of missing may be higher for more sensitive/important attributes. For example, high income people are more likely to not report their income in a survey. We first assess the relative importance of each feature in a classification task (by measuring the accuracy loss after removing a feature), and use the relative feature importance as the relative probability of a feature missing. We can then inject missing values into a dataset for any given missing rate (we use 20% in our experiment).

Model. We use a KNN classifier with K=3 and use Euclidean distance as the similarity function. For each dataset, we randomly select 1,000 examples as the validation set and 1,000 examples as the test set. The remaining examples are used as the training set.

Cleaning Algorithms Compared. We compare the following approaches for handling missing values in the training data.

- *Ground Truth*: This method uses the ground-truth version of the dirty data, and shows the performance upper-bound.
- *Default Cleaning*: This is the default and most commonly used way for cleaning missing values in practice, namely, missing cells in a numerical column are filled in using the mean value of the column, and those in a categorical column are filled using the most frequent value of that column.
- *CPClean*: This is our proposal, which needs a candidate repair set $\mathcal{C}_i$ for each example with missing values. For missing cells in numerical columns, we consider five candidate repairs: the minimum value, the 25-th percentile, the mean value, the 75-th percentile and the maximum value of the column. For missing cells in categorical columns, we also consider five candidate repairs: the top 4 most frequent categories and a dummy category named "other category". If a record i has multiple missing values, then the Cartesian product of all candidate repairs for all missing cells forms $\mathcal{C}_i$. We simulate human cleaning by picking the candidate repair that is closest to the ground truth.

- *HoloClean*: This is the state-of-the-art probabilistic data cleaning method [11]. As a weakly supervised machine learning system, it leverages multiple signals (e.g. quality rules, value correlations, reference data) to build a probabilistic model for imputing and cleaning data. Note that the focus of HoloClean is to find the most likely fix for a missing cell in a dataset without considering how the dataset is used by downstream classification tasks.
- *BoostClean*: This is the state-of-the-art automatic data cleaning method for ML [7]. At a high level, it selects, from a predefined set of cleaning methods, the one that has the maximum validation accuracy on the validation set. To ensure fair comparison, we use the same cleaning method as in CPClean, i.e., the predefined cleaning methods include cleaning a numerical column with missing values using 25-th percentile, the mean value, etc. We also use the same validation set as in CPClean.
- *RandomClean*: While CPClean uses the idea of sequential information maximization to select which examples to clean, *RandomClean* simply selects an example randomly to clean.

Performance Measures. Besides the cleaning effort spent, we are mainly concerned with the test accuracy of models trained on datasets cleaned by different cleaning methods. Instead of reporting exact test accuracies for all methods, we only report them for *Ground Truth* and *Default Cleaning*, which represents the upper bound and the lower bound, respectively. For other methods, we report the percentage of closed gap defined as:

$$\textit{gap closed by } X = \frac{\text{accuracy(X) - accuracy(}\textit{Default Cleaning}\text{)}}{\text{accuracy(}\textit{Ground Truth}\text{) - accuracy(}\textit{Default Cleaning}\text{)}}.$$

5.2 Experimental Results

Dataset	*Ground Truth*	*Default Cleaning*	*BoostClean*	*HoloClean*	*CPClean*			
	Test Accuracy	**Test Accuracy**	**Gap Closed**	**Gap Closed**	**Gap Closed**	**Examples Cleaned**	**Gap Closed**	**Examples Cleaned**
BabyProduct	0.668	0.589	1%	1%	99%	64%	72%	20%
Supreme	0.968	0.877	12%	-4%	100%	15%	100%	20%
Bank	0.643	0.558	20%	11%	102%	93%	52%	20%
Puma	0.794	0.747	28%	-64%	102%	63%	40%	20%

Table 2: End-to-End Performance Comparison

Model Accuracy Comparison. Table 2 shows the end-to-end performance of our method and other automatic cleaning methods. We can see that the missing values exhibit different degress of impact on these datasets (the gap between *Ground Truth* and *Default Cleaning*). We can also observe that *HoloClean*, the state-of-the-art standalone data cleaning approach performs poorly — the gap closed can even be negative. This suggests that performing data cleaning on a data without considering how it is used later may not necessarily improve downstream model performance. On the other hand, we observe that *BoostClean* shows a consistently positive impact on model performance by using the validation set to pick the most useful cleaning method. In all cases, *CPClean* is able to close 100% of gap without manual cleaning of all dirty data. In fact, on Supreme, *CPClean* only requires the manual cleaning of 15% of missing records to close 100% gap. We can also see from Table 2 that, by cleaning only 20% of all dirty data, i.e., terminating the cleaning process at 20% mark even if not all validation examples are CP'ed, *CPClean* is able to close 66% gap on average.

Early Termination. If users have a limited cleaning budget, they may choose to terminate *CPClean* early. To study the effectiveness of *CPClean* in prioritizing cleaning effort, we compare it with *RandomClean* that randomly picks an example to clean at each iteration. The results for *RandomClean* are the average of 20 runs.

The red lines in Figure 9 show the percentage of CP'ed examples in the validation set as more and more examples are cleaned. As we can see, *CPClean* (solid red line) dramatically outperforms the *RandomClean* (dashed red line) both in terms of the number of training examples cleaned so that all validation examples

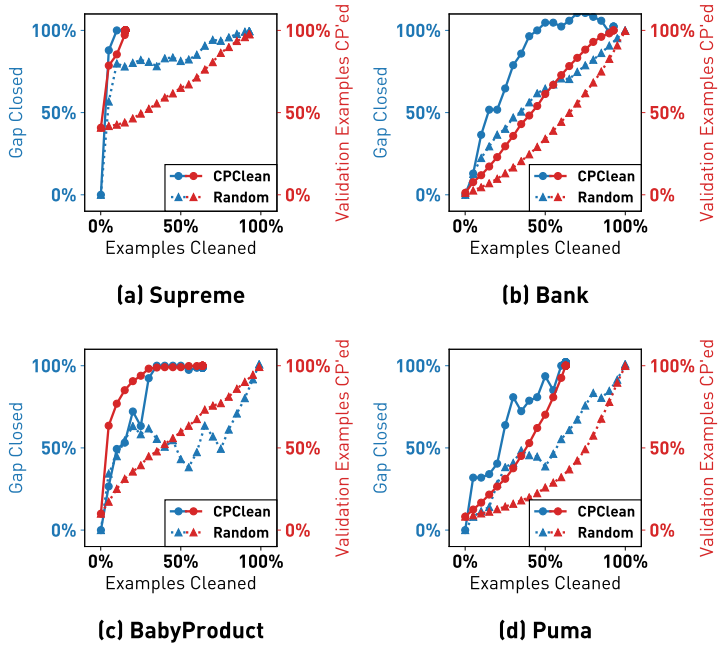

Figure 9: Comparison with Random Cleaning

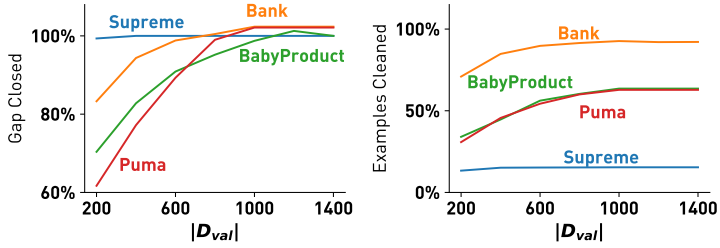

Figure 10: Varying size of D_{val}.

are CP'ed and in terms of the rate of convergence. For example, for Supreme, *CPClean* requires the cleaning of 15% examples while *RandomClean* requires cleaning almost all training examples.

The blue lines in Figure 9 show the percentage of gap closed for the test set accuracy. Again, we can observe that *CPClean* significantly outperforms *RandomClean*. For example, with 50% of data cleaned in Bank, *RandomClean* only closes about 65% of the gap, whereas *CPClean* closes almost 100% of the gap.

Size of the Validation Set D_{val}. We vary the size of D_{val} to understand how it affects the result. As shown in Figure 10, as the size of validation set increases, both the test accuracy gap closed and the cleaning effort spent first increase and then become steady. This is because, when the validation set is small, it is easier to make all validation examples CP'ed (hence the smaller cleaning effort). However, a small validation set may not be representative of some unseen test set, and hence may not close the accuracy gap on test set. In all cases, we observe that 1K validation set is sufficiently large and further increasing it does not improve the performance.

6 Related Work

Relational Query over Incomplete Information. This work is heavily inspired by the database literature of handling incomplete information [1], consistent query answering [3,20,21], and probabilistic databases [2]. While these work targets SQL analytics, our proposed consistent prediction query targets ML analytics.

Learning over Incomplete Data. The statistics and ML community have also studied the problem of learning over incomplete data. Many studies operate under certain missingness assumption (e.g., missing completeness at random) and reason about the performance of downstream classifiers in terms of asymptotic properties and in terms of different imputation strategies [22]. In this work, we focus more on the algorithmic aspect of this problem and try to understand how to enable more efficient manual cleaning of the data. Another flavor of work aims at developing ML models that are robust to certain types of noises, and multiple imputation [23] is such a method that is most relevant to us. Our CP framework can be seen as an extreme case of multiple imputation (i.e, by trying all possible imputations) with efficient implementation (in KNN), which also enables novel manual cleaning for ML use cases.

Recently, Khosravi et al. [24] explored a similar semantics as ours, but for Logistic Regression models. In this paper, we focus on efficient algorithms for nearest neighbor classifiers.

Data Cleaning and Analytics-Driven Cleaning. The research on data cleaning (DL) has been thriving for many years. Many data cleaning works focus on performing standalone cleaning without considering how cleaned data is used by downstream analytics. We refer readers to a recent survey on this topic [13].

As data cleaning itself is an expensive process that usually needs human involvement eventually (e.g., to confirm suggested repairs), the DB community is starting to work on analytics-driven cleaning methods. SampleClean [25] targets the problem of answering SQL aggregate queries when the input data is dirty by cleaning a sample of the dirty dataset, and at the same time, providing statistical guarantees on the query results. ActiveClean [8] is an example of cleaning data intelligently for convex ML models that are trained using gradient descent methods. As discussed before, while both ActiveClean and our proposal assume the use of a human cleaning oracle, they are incomparable as they are targeting different ML models. BoostClean [7] automatically selects from a predefined space of cleaning algorithms, using a hold-out validation set via statistical boosting. We show that our proposal significantly outperforms BoostClean under the same space of candidate repairs.

7 Conclusion

In this work, we focused on the problem of understanding the impact of incomplete information on training downstream ML models. We present a formal study of this impact by extending the notion of *Certain Answers for Codd tables*, which has been explored by the database research community for decades, into the field of machine learning, by introducing the notion of *Certain Predictions* (CP). We developed efficient algorithms to analyze the impact via CP primitives, in the context of nearest neighor classifiers. As an application, we further proposed a novel "DC for ML" framework built on top of CP primitives that often significantly outperforms existing techniques in accuracy, with mild manual cleaning effort.

References

[1] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases: The Logical Level*, 1st ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] D. Suciu, D. Olteanu, C. Ré, and C. Koch, "Probabilistic databases," *Synthesis Lectures on Data Management*, vol. 3, no. 2, pp. 1–180, 2011. [Online]. Available: https://doi.org/10.2200/S00362ED1V01Y201105DTM016

[3] M. Arenas, L. Bertossi, and J. Chomicki, "Consistent query answers in inconsistent databases," in *Proc. 18th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, 1999, pp. 68–79.

[4] P. K. Agarwal, A. Efrat, S. Sankararaman, and W. Zhang, "Nearest-neighbor searching under uncertainty," in *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. PODS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 225–236. [Online]. Available: https://doi.org/10.1145/2213556.2213588

[5] P. K. Agarwal, B. Aronov, S. Har-Peled, J. M. Phillips, K. Yi, and W. Zhang, "Nearest-neighbor searching under uncertainty ii," *ACM Trans. Algorithms*, vol. 13, no. 1, Oct. 2016. [Online]. Available: https://doi.org/10.1145/2955098

[6] H.-P. Kriegel, P. Kunath, and M. Renz, "Probabilistic nearest-neighbor query on uncertain objects," in *Proceedings of the 12th International Conference on Database Systems for Advanced Applications*, ser. DASFAA'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 337–348.

[7] S. Krishnan, M. J. Franklin, K. Goldberg, and E. Wu, "Boostclean: Automated error detection and repair for machine learning," *arXiv preprint arXiv:1711.01299*, 2017.

[8] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg, "Activeclean: Interactive data cleaning for statistical modeling," *Proc. VLDB Endowment*, vol. 9, no. 12, pp. 948–959, 2016.

[9] P. Li, X. Rao, J. Blase, Y. Zhang, X. Chu, and C. Zhang, "Cleanml: A benchmark for joint data cleaning and machine learning [experiments and analysis]," *arXiv preprint arXiv:1904.09483*, 2019.

[10] Y. Chen, H. Hassani, S., A. Karbasi, and A. Krause, "Sequential information maximization: When is greedy near-optimal?" in *Conference on Learning Theory*, 2015.

[11] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré, "Holoclean: Holistic data repairs with probabilistic inference," *arXiv preprint arXiv:1702.00820*, 2017.

[12] X. Chu, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, N. Tang, and Y. Ye, "KATARA: A data cleaning system powered by knowledge bases and crowdsourcing," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2015, pp. 1247–1261.

[13] I. F. Ilyas and X. Chu, *Data Cleaning*. ACM, 2019. [Online]. Available: https://doi.org/10.1145/3310205

[14] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg, "Activeclean: Interactive data cleaning for statistical modeling," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 948–959, 2016.

[15] C. Wa Ko, J. Lee, and M. Queyranne, "An exact algorithm for maximum entropy sampling," *Oper. Res.*, vol. 43, no. 4, pp. 684–691, 1995. [Online]. Available: https://doi.org/10.1287/opre.43.4.684

[16] S. Das, A. Doan, P. S. G. C., C. Gokhale, P. Konda, Y. Govind, and D. Paulsen, "The magellan data repository," https://sites.google.com/site/anhaidgroup/projects/data.

[17] J. S. Simonoff, *Analyzing categorical data*. Springer Science & Business Media, 2013.

[18] C. E. Rasmussen, R. M. Neal, G. E. Hinton, D. van Camp, M. Revow, Z. Ghahramani, R. Kustra, and R. Tibshirani, "The delve manual," *URL http://www. cs. toronto. edu/˜ delve*, 1996.

[19] D. B. Rubin, "Inference and missing data," *Biometrika*, vol. 63, no. 3, pp. 581–592, 1976.

[20] A. Lopatenko and L. E. Bertossi, "Complexity of consistent query answering in databases under cardinality-based and incremental repair semantics," in *Proc. 11th Int. Conf. on Database Theory*, 2007, pp. 179–193.

[21] L. E. Bertossi, *Database Repairing and Consistent Query Answering*. Morgan & Claypool Publishers, 2011.

[22] P. J. García-Laencina, J.-L. Sancho-Gómez, and A. R. Figueiras-Vidal, "Pattern classification with missing data: A review," *Neural Comput. Appl.*, vol. 19, no. 2, p. 263–282, Mar. 2010. [Online]. Available: https://doi.org/10.1007/s00521-009-0295-6

[23] D. B. Rubin, "Multiple imputation after 18+ years," *Journal of the American Statistical Association*, vol. 91, no. 434, pp. 473–489, 1996. [Online]. Available: http://www.jstor.org/stable/2291635

[24] P. Khosravi, Y. Liang, Y. Choi, and G. V. den Broeck, "What to expect of classifiers? reasoning about logistic regression with missing features," *CoRR*, vol. abs/1903.01620, 2019. [Online]. Available: http://arxiv.org/abs/1903.01620

[25] J. Wang, S. Krishnan, M. J. Franklin, K. Goldberg, T. Kraska, and T. Milo, "A sample-and-clean framework for fast and accurate query processing on dirty data," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2014, pp. 469–480.

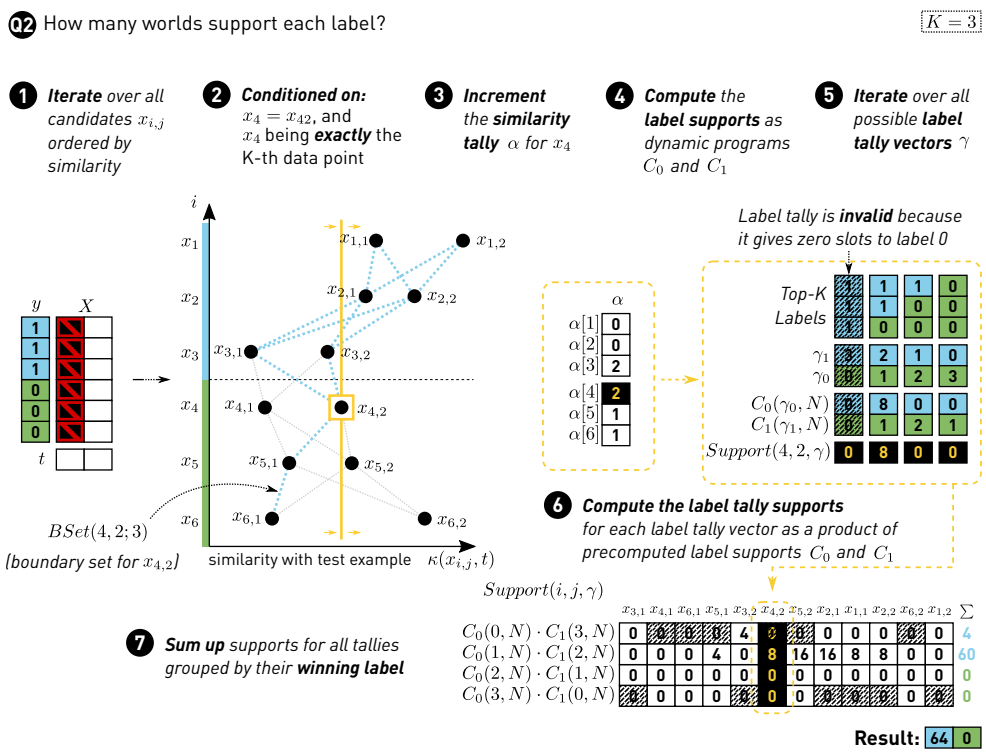

Figure A.1: Illustration of SS when $K = 3$ for $Q2$.

A The SS Algorithm for Q2

Algorithm Outline. The SS algorithm answers the counting query according to the following expression:

$$Q2(\mathcal{D}, t, l) = \sum_{i \in [N]} \sum_{j \in [M]} \sum_{\gamma \in \Gamma} \mathbb{I}[l = \arg\max(\gamma)] \cdot Support(i, j, \gamma) \tag{A.1}$$

Here, we iterate over all $i = 1, ..., N$ candidate sets $\mathcal{C}_i$, all their $j = 1, ..., M$ candidates $x_{i,j} \in \mathcal{C}_i$, and all possible tally vectors $\gamma \in \Gamma$. In each iteration we check if the winning label induced by tally vector γ is the label l which we are querying for. If yes, we include the label tally support into the count. An example of this iteration is depicted in the lower right table in Figure B.1.

The *label tally support* $Support(i, j, \gamma)$ is the number of possible worlds from the boundary set $BSet(i, j; K)$ where the tally of the labels in the top-K is exactly γ. The boundary set $BSet(i, j; K)$ is the set of possible worlds where the value of example x_i is taken to be $x_i = x_{i,j}$ *and* x_i is exactly the K-th most similar example to the test example t. Given a label tally γ, its label tally support is computed as:

$$Support(i, j, \gamma) = \prod\nolimits_{l \in \mathcal{Y}} C_l^{i,j}(\gamma_l, N) \tag{A.2}$$

Here, $C_l^{i,j}(\gamma_l, N)$ is the *label support* for label l, which is the number of possible worlds from the *boundary set* $BSet(i, j; K)$ having exactly γ_l examples in the top-K. Computing the label support is performed by using the following recursive structure:

$$C_l^{i,j}(c, n) = \begin{cases} C_l^{i,j}(c, n-1), & \text{if } y_n \neq l, \\ C_l^{i,j}(c-1, n-1), & \text{if } x_n = x_i, \text{otherwise} \\ \alpha_{i,j}[n] \cdot C_l^{i,j}(c, n-1) + (M - \alpha_{i,j}[n]) \cdot C_l^{i,j}(c-1, n-1). \end{cases}$$

This recursion can be computed as a dynamic program, defined over $c \in \{0...K\}$ and $n \in \{1...N\}$. Its boundary conditions are $C_l^{i,j}(-1, n) = 0$ and $C_l^{i,j}(c, 0) = 1$. To compute the support, it uses similarity tallies α, defined as such:

$$\alpha_{i,j}[n] = \sum_{m=1}^{M} \mathbb{I}[\kappa(x_{n,m}, t) \leq \kappa(x_{i,j}, t)].$$

A.1 Proof of Correctness

Theorem A.1. *The SS algorithm correctly answers $Q2(\mathcal{D}, t, l)$.*

Proof. The SS algorithm aims to solve a counting problem by using a technique of partitioning a set and then computing the sizes of the relevant partitions. To prove its correctness we need to: (1) argue that the partitioning procedure is valid and produces disjoint subsets of the original set; and (2) argue that the size of the subset is computed correctly.

To prove the validity of the partitioning method, we start off by reviewing how a brute-force approach would answer the same query:

$$Q2(\mathcal{D}, t, l) = \sum_{D \in \mathcal{D}} \mathbb{I}[\mathcal{A}_D(t) = l]$$

When we partition the sum over all possible worlds into boundary sets $BSet(i, j; K)$ for each $i \in 1...N$ and $j \in 1...M$, we obtain the following expression:

$$Q2(\mathcal{D}, t, l) = \sum_{i \in [N]} \sum_{j \in [M]} \sum_{D \in BSet(i,j;K)} \mathbb{I}[\mathcal{A}_D(t) = l]$$

As we mentioned, a boundary set is the set of the possible worlds where $x_i = x_{i,j}$ and x_i is the K-th most similar data example to t. Since every possible world selects just one candidate per candidate set, for every $i \in \{1...N\}$, the possible world where $x_i = x_{i,j}$ is always different from the possible world where $x_i = x_{i,j'}$, for every $j, j' \in \{1...M\}$ such that $j \neq j'$. Furthermore, every possible world induces a fixed ordering of data examples based on their similarity to t. Therefore, any possible worlds where x_i occupies the K-th position in that ordering is different from the possible world where it occupies any other position. Thus, we can conclude that all boundary sets $BSet(i, j, K)$ are distinct for all distinct i and j.

Given that we are dealing with the K-NN algorithm, since each possible world D induces a fixed set of top-K examples, consequently it induces a fixed top-K label tally γ^D. Since only one label tally of all the possible ones will be correct one, we can rewrite the inner sum as:

$$Q2(\mathcal{D}, t, l) = \sum_{i \in [N]} \sum_{j \in [M]} \sum_{D \in BSet(i,j;K)} \sum_{\gamma \in \Gamma} \mathbb{I}[\gamma^D = \gamma] \mathbb{I}[l = \arg\max \gamma]$$

Since in the above expression, the γ is independent from D, we can reorganize the sums as such:

$$Q2(\mathcal{D}, t, l) = \sum_{i \in [N]} \sum_{j \in [M]} \sum_{\gamma \in \Gamma} \mathbb{I}[l = \arg\max \gamma] \sum_{D \in BSet(i,j;K)} \mathbb{I}[\gamma^D = \gamma]$$

We can notice that the innermost sum is equivalent to the definition of a label tally support, which means we can replace it as such:

$$Q2(\mathcal{D}, t, l) = \sum_{i \in [N]} \sum_{j \in [M]} \sum_{\gamma \in \Gamma} \mathbb{I}[l = \arg\max(\gamma)] \cdot Support(i, j, \gamma)$$

Assuming that the label tally support $Support(i, j, \gamma)$ is computed correctly, as shown in Section 3.1.1, we can conclude that both the partitioning and the partition size computation problems are solved correctly, hence proving our original claim.

□

A.2 Optimization Using Divide and Conquer

Algorithm Outline. This version of the algorithm is almost identical to the original SS algorithm described previously, except for the way it computes the label support. Namely, in the original algorithm we were using the dynamic program $C_l^{i,j}(c,n)$ to return the number of possible worlds in the boundary set $BSet(i,j;K)$ that support having exactly c examples in the top-K. Here, the parameter $n \in \{1...N\}$ denoted that we were only considering the subset of candidate sets $\mathcal{C}_i$ where $i \in \{1...n\}$.

If we observe Algorithm 1, we can see that the dynamic program $C_l^{i,j}(c,n)$ is re-computed in every iteration of the outer loop. However, at the same time we can see that the similarity tally α, which is used to compute the dynamic program, gets only one of its elements updated. To take advantage of that, we apply a divide-and-conquer strategy and redefine the recurrence relation as a tree structure:

$$T_l^{i,j}(c,a,b) = \sum_{k=0}^{c} T_l^{i,j}(k,a,m) \cdot T_l^{i,j}(c-k,a+1,b), \quad \text{where } m = \lfloor \frac{a+b}{2} \rfloor. \tag{A.3}$$

To efficiently maintain the dynamic program $T_l^{i,j}$ across iterations over (i,j), we organize it in a binary tree structure. Each node, denoted as $n_{a,b}$, contains a list of values of $T_l^{i,j}(c,a,b)$ for all $c \in \{0...K\}$. Its two children are $n_{a,m}$ and $n_{m+1,b}$ where $m = \lfloor \frac{a+b}{2} \rfloor$. The leaves are nodes $n_{a,a}$ with both indices equal, which get evaluated according to the following base conditions:

1. $T_l^{i,j}(c,a,a) = 1$, if $y_a \neq l$;
   Rationale: Skip examples with label different from l.
2. $T_l^{i,j}(0,i,i) = 0$ and $T_l^{i,j}(1,i,i) = 1$;
   Rationale: The i-th example must be in the top-K, unless it got skipped.
3. $T_l^{i,j}(0,a,a) = \alpha[a]$;
   Rationale: If the a-th example is in the top-K, there are $\alpha[a]$ candidates to choose from.
4. $T_l^{i,j}(1,a,a) = M - \alpha[a]$;
   Rationale: If the a-th example is not in the top-K, there are $M - \alpha[a]$ candidates to choose from.
5. $T_l^{i,j}(c,a,a) = 0$, if $c \notin \{0,1\}$
   Rationale: Invalid case because an example can either be ($c = 1$) or not be ($c = 0$) in the top-K.

The leaf nodes $n_{a,a}$ of this tree correspond to label support coming from individual data examples. The internal nodes $n_{a,b}$ correspond to the label support computed over all leaves in their respective sub-trees. This corresponds to data examples with index $i \in \{a...b\}$. The root node $n_{1,N}$ contains the label support computed over all data examples.

Since between any two consecutive iterations of (i,j) in Algorithm 1 we only update the i-th element of the similarity tally α, we can notice that out of all leaves in our binary tree, only $n_{i,i}$ gets updated. This impacts only $\mathcal{O}(N)$ internal nodes which are direct ancestors to that leaf. If we update only those nodes, we can avoid recomputing the entire dynamic program. The full algorithm is listed in Algorithm A.1.

Complexity. We analyze the complexity of Algorithm A.1:

- The sorting procedure requires $\mathcal{O}(N \cdot M \log N \cdot M)$ steps as it sorts all elements of S.
- The tree initialization can be performed eagerly in $\mathcal{O}(KN)$ time, or lazily in constant amortized time.
- The outer loop iterates over $\mathcal{O}(N \cdot M)$ elements.

Algorithm A.1 Algorithm **SS-DC** for Q2 with K-NN.

Input: $\mathcal{D}$, incomplete dataset; t, target data point.
Output: r, integer vector, s.t. $r[y] = Q2(\mathcal{D}, t, y), \forall y \in \mathcal{Y}$.

1: $s \leftarrow \texttt{kernel}(\mathcal{D}, t)$;
2: $\alpha \leftarrow$ *zeros*$(|\mathcal{D}|)$;
3: $r \leftarrow$ *zeros*$(|\mathcal{Y}|)$;
4: **for all** $l \in \mathcal{Y}$ **do**
5: **for all** $k \in [K]$ **do**
6: Initialize the tree $T_l(k, 1, N)$.
7: **for all** $(i, j) \in \texttt{argsort}(s)$ **do**
8: $\alpha[i] \leftarrow \alpha[i] + 1$;
9: Update the leaf node $T_{y_i}(c, i, i)$ and its ancestors for all $c \in \{1...K\}$.
10: **for all** possible valid tally vectors $\gamma \in \Gamma$ **do**
11: $y_p \leftarrow \texttt{argmax}(\gamma)$;
12: Compute $Support(i, j, \gamma) = \mathbb{I}[\gamma_{y_i} \geq 1] \cdot \prod_{l \in \mathcal{Y}} T_l(\gamma_l, 1, N)$;
13: $r[y_p] \leftarrow r[y_p] + Support(i, j, \gamma)$;
14: **return** r;

- In each inner iteration, we update $\mathcal{O}(\log N)$ nodes. Each node maintains support values for all $c \in \{1...K\}$ and each one takes $\mathcal{O}(K)$ to recompute. Therefore, the tree update can be performed in $\mathcal{O}(K^2 \log N)$ time.
- Furthermore, in each iteration, we iterate over all possible label assignments, which requires $\mathcal{O}\left(\binom{|\mathcal{Y}|+K-1}{K}\right)$ operations.
- For each label assignment, we need $\mathcal{O}(|\mathcal{Y}|)$ multiplications.

This renders the final complexity to be the sum of $\mathcal{O}(N \cdot M \cdot (\log(N \cdot M) + K^2 \cdot \log N))$ and $\mathcal{O}(N \cdot M \cdot (|\mathcal{Y}| + \binom{|\mathcal{Y}|+K-1}{K}))$. When $|\mathcal{Y}|$ and K are relatively small constants, this reduces to $\mathcal{O}(N \cdot M \cdot \log(N \cdot M))$.

A.3 Polynomial Time Solution for $|\mathcal{Y}| \geq 1$

We have seen that the previously described version of the SS algorithm gives an efficient polynomial solution for $Q2$, but only for a relatively small number of classes $|\mathcal{Y}|$. When $|\mathcal{Y}| \gg 1$, the $\mathcal{O}(\binom{|\mathcal{Y}|+K-1}{K})$ factor of the complexity starts to dominate. For a very large number of classes (which is the case for example in the popular ImageNet dataset), running this algorithm becomes practically infeasible. In this section we present a solution for $Q2$ which is polynomial in $|\mathcal{Y}|$.

The main source of computational complexity in Algorithm A.1 is in the for-loop starting at line 10. Here we iterate over all possible tally vectors γ, and for each one we compute the label tally support $Support(i, j, \gamma)$ (line 12) and add it to the resulting sum (line 13) which is selected according to the winning label with the largest tally in γ (line 11).

The key observation is that, for l to be the winning label, one only needs to ensure that no other label has a larger label tally. In other words, label l will be predicted whenever $\gamma_l > \gamma_{l'}$ for all $l \neq l'$, regardless of the actual tallies of all l'. Therefore, we found that we can group all the label tally vectors according to this predicate. To achieve this, we define the following recurrence:

$$D_{Y,c}(j, k) = \sum_{n=0}^{\min\{c,k\}} T_{Y_j}(n, 1, N) \cdot D_{Y,c}(j + 1, k - l). \tag{A.4}$$

Here Y is the list of all labels in $\mathcal{Y} \setminus \{l\}$ and $T^{i,j}_{Y_j}(n, 1, N)$ is the label support for label Y_j, as described in the previous section. The semantics of $D_{Y,c}(j, k)$ is the number of possible worlds where the top-K contains

at most k examples with labels $l' \in Y_{0...j}$ and no label has tally above c. We can see that $D_{Y,c}(j,k)$ can also be computed as a dynamic program with base conditions $D_{Y,c}(|Y|,0) = 1$ and $D_{Y,c}(|Y|,k) = 0$ for $k > 0$.

Algorithm A.2 Algorithm **SS-DC-MC** for Q2 with K-NN.

Input: $\mathcal{D}$, incomplete dataset; t, target data point.
Output: r, integer vector, s.t. $r[y] = Q2(\mathcal{D}, t, y), \forall y \in \mathcal{Y}$.

1: $s \leftarrow \texttt{kernel}(\mathcal{D}, t)$;
2: $\alpha \leftarrow$ *zeros*$(|\mathcal{D}|)$;
3: $r \leftarrow$ *zeros*$(|\mathcal{Y}|)$;
4: **for all** $l \in \mathcal{Y}$ **do**
5: **for all** $k \in [K]$ **do**
6: Initialize the tree $T_l(k, 1, N)$.
7: **for all** $(i, j) \in \texttt{argsort}(s)$ **do**
8: $\alpha[i] \leftarrow \alpha[i] + 1$;
9: Update the leaf node $T_{y_i}(c, i, i)$ and its ancestors for all $c \in \{1...K\}$.
10: **for all** $l \in \mathcal{Y}$ **do**
11: $Y \leftarrow [\mathcal{Y} \setminus \{l\}]$;
12: **for all** $c \in \{1...K\}$ **do**
13: Compute $D_{Y,c}(|\mathcal{Y}| - 1, K - c - 1)$ using dynamic programming;
14: $r[y_p] \leftarrow r[y_p] + T_l(c, 1, N) \cdot D_{Y,c}(|\mathcal{Y}| - 1, K - c - 1)$;
15: **return** r;

In terms of performance, the complexity of Algorithm A.2, compared to Algorithm A.1 has one more major source of time complexity, which is the computation of the dynamic program $D_{Y,c}$ which takes $\mathcal{O}(|\mathcal{Y}| \cdot K^2)$ time. Since the for loops in lines 10 and 12 take $O(|\mathcal{Y}|)$ and $O(K)$ time respectively, the overall complexity of the algorithm becomes $O(M \cdot N \cdot (\log(M \cdot N) + K^2 \log N + |\mathcal{Y}|^2 \cdot K^3)$.

B The MM Algorithm for Q1

Algorithm Outline. We are given an incomplete dataset $\mathcal{D} = \{(\mathcal{C}_i, y_i) : i = 1, ..., N\}$, a test data point $t \in \mathcal{X}$ and a class label $l \in \mathcal{Y}$. The MM algorithm answers the checking query $Q1(\mathcal{D}, t, l)$ for K-NN with similarity kernel κ by constructing the l-expreme possible world $E_{l,\mathcal{D}}$ defined as:

$$E_{l,\mathcal{D}} = \{(M_i, y_i) : (\mathcal{C}_i, y_i) \in \mathcal{D}\}$$
$$M_i = \begin{cases} \arg\max_{x_{i,j} \in \mathcal{C}_i} \kappa(x_{i,j}, t), & \text{if } y_i = l, \\ \arg\min_{x_{i,j} \in \mathcal{C}_i} \kappa(x_{i,j}, t), & \text{otherwise} \end{cases} \tag{B.1}$$

The answer to $Q1(\mathcal{D}, t, l)$ is obtained by checking if: (1) K-NN trained over $E_{l,\mathcal{D}}$ predicts l, and (2) for all other labels $l' \in \mathcal{Y} \setminus \{l\}$, K-NN trained over $E_{l',\mathcal{D}}$ does not predict l. Figure B.1 depicts this algorithm for an example scenario.

Example B.1. *In Figure B.1 we can see an example scenario illustrating the MM algorithm for $K = 3$. On the left, we have an incomplete dataset with $N = 6$ examples, each with $M = 4$ candidates. We construct l-extreme worlds for both $l = 0$ and $l = 1$, by picking the candidate with maximal similarity when $y_i = l$ and the candidate with minimal similarity when $y_i \neq l$. We can see visually that any other choice of candidate could not reduce the odds of l being predicted. In this scenario, we can see that both l-extreme worlds predict label 1, which means that we can conclude that label 1 can be* certainly predicted.

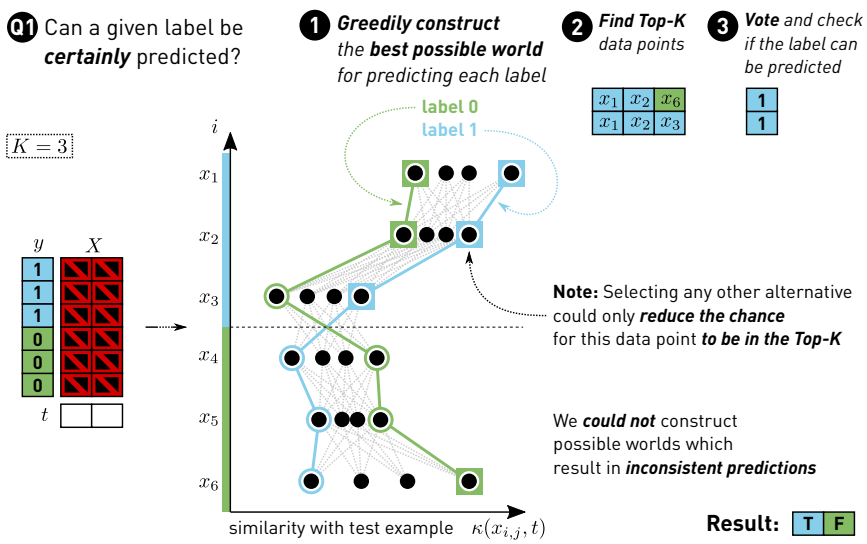

Figure B.1: Illustration of MM when $K = 3$ for $Q1$.

B.1 Proof of Correctness

Lemma B.1. *Let $D^{(1)}, D^{(2)} \in \mathcal{I}_{\mathcal{D}}$ be two possible worlds generated from an incomplete dataset $\mathcal{D}$. Given a test example $t \in \mathcal{X}$ and label $l \in \mathcal{Y}$ where $|\mathcal{Y}| = 2$, let $R_{t,l}$ be a partial ordering relation defined as such:*

$$R_{t,l}\left(D^{(1)}, D^{(2)}\right) := \bigwedge_{i=1}^{N} \left(y_i = l \ \wedge \ \kappa(x_i^{(1)}, t) \leq \kappa(x_i^{(2)}, t)\right) \vee \left(y_i \neq l \ \wedge \ \kappa(x_i^{(1)}, t) \geq \kappa(x_i^{(2)}, t)\right)$$

Then, the following relationship holds:

$$R_{t,l}\left(D^{(1)}, D^{(2)}\right) \implies \left((\mathcal{A}_{D^{(1)}}(t) = l) \implies (\mathcal{A}_{D^{(2)}}(t) = l)\right)$$

Proof. We will prove this by contradiction. Consider the case when $\mathcal{A}_{D^{(1)}}(t) = l$ and $\mathcal{A}_{D^{(2)}}(t) \neq l$, that is, possible world $D^{(1)}$ predicts label l but possible world $D^{(2)}$ predicts some other label $l' \neq l$. That means that in the top-K induced by $D^{(2)}$ has to be at least one more data point with label l' than in the top-K induced by $D^{(1)}$. Is it possible for the premise to be true?

The similarities $\kappa(x_i^{(1)}, t)$ and $\kappa(x_i^{(2)}, t)$ cannot all be equal because that would represent equal possible worlds and that would trivially contradict with the premise since the labels predicted by equal possible worlds cannot differ. Therefore, at least one of the $i = 1, ..., N$ inequalities has to be strict. There, we distinguish three possible cases with respect to the class label y_i of the i-th example:

1. $y_i = l$: This means that the similarity of a data point coming from $D^{(2)}$ is higher than the one coming from $D^{(1)}$. However, this could only elevate that data point in the similarity-based ordering and could only increase the number of data points with label l in the top-K. Since $\mathcal{A}_{D^{(1)}}(t) = l$, the prediction of $D^{(2)}$ could not be different, which is **contradicts** the premise.

2. $y_i = l' \neq l$ and $|\mathcal{Y}| = 2$: We have a data point with a label different from l with lowered similarity, which means it can only drop in the ordering. This can not cause an increase of data points with label l' in the top-K, which again **contradicts** the premise.

3. $y_i = l' \neq l$ and $|\mathcal{Y}| \geq 2$: Here we again have a data point with label l' with lowered similarity. However, if that data point were to drop out of the top-K, it would have been possible for a data point with a third label $l'' \notin \{l, l'\}$ to enter the top-K and potentially tip the voting balance in factor of this third

label l'' (assuming there are enough instances of that label in the top-K already). This would **not contradict** the premise. However, since the lemma is defined for $|\mathcal{Y}| = 2$, this third case can actually never occur.

Finally, for $|\mathcal{Y}| = 2$, we can conclude that our proof by contradiction is complete.

□

Lemma B.2. *Let $E_{l,\mathcal{D}}$ be the l-extreme world defined in Equation B.1. Then, the K-NN algorithm trained over $E_{l,\mathcal{D}}$ will predict label l if and only if there exists a possible world $D \in \mathcal{I}_{\mathcal{D}}$ that will predict label l.*

Proof. We consider the following two cases:

1. $\mathcal{A}_{E_{l,\mathcal{D}}}(t) = l$: Since $E_{l,\mathcal{D}} \in \mathcal{I}_{\mathcal{D}}$, the successful prediction of label l represents a trivial proof of the existence of a possible world that predicts l.

2. $\mathcal{A}_{E_{l,\mathcal{D}}}(t) \neq l$: We can see that $E_{l,\mathcal{D}}$ is unique because it is constructed by taking from each candidate set $\mathcal{C}_i$ the minimal/maximal element, which itself is always unique (resulting from the problem setup laid out in Section 3.1.1). Consequently, the relation $R_{t,l}(D, E_{l,\mathcal{D}})$ holds for every $D \in \mathcal{I}_{\mathcal{D}}$. Given Lemma B.1, we can say that if there exists any $D \in \mathcal{I}_{\mathcal{D}}$ that will predict l, then it is impossible for $E_{l,\mathcal{D}}$ to not predict l. Conversely, we can conclude that if $E_{l,\mathcal{D}}$ does not predict l, then no other possible world can predict l either.

□

Theorem B.1. *The MM algorithm correctly answers $Q1(\mathcal{D}, t, l)$.*

Proof. The MM algorithm simply constructs the l'-extreme world $E_{l',\mathcal{D}}$ for each label $l' \in \mathcal{Y}$ and runs K-NN over it to check if it will predict l'. Given Lemma B.2, we can conclude that this test is sufficient to check if there exists a possible world that can predict label l'. Then, the algorithm simply checks if l is the only label that can be predicted. We can trivially accept that this always gives the correct answer given that it is an exhaustive approach.

□

C The CPClean Algorithm

C.1 Theoretical Guarantee

We begin with the following supplementary results.

Lemma C.1. *Let D_{Opt} be the optimal set of size t described in Corollary 1. Denote the set of cleaned training data instances of size T by D_π. For $\mathbf{c}_{Opt_i} \in D_{Opt}, i = \{1, 2, \ldots, t\}$, we have*

$$\begin{aligned} &I(\mathcal{A}_{\mathbf{D}}(D_{val}); \mathbf{c}_{Opt_j} | \mathbf{c}_{Opt_{j-1}}, \ldots, \mathbf{c}_{Opt_1}, D_\pi) \\ &\leq \theta \min\{\log|\mathcal{Y}|, \log m\} I(\mathcal{A}_{\mathbf{D}}(D_{val}); \mathbf{c}_{\pi_{t+1}} | D_\pi). \end{aligned} \tag{C.1}$$

where $\theta = \frac{1}{\max_{\mathbf{v} \in D_{train}} I(\mathcal{A}_{\mathbf{D}}(D_{val}); \mathbf{v})}$.

Proof. We first start with the following inequality:

$$\begin{aligned} &I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{c}_{\text{Opt}_j} | \mathbf{c}_{\text{Opt}_{j-1}}, ..., \mathbf{c}_{\text{Opt}_1}, D_\pi) \\ &= \frac{I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{c}_{\text{Opt}_j} | \mathbf{c}_{\text{Opt}_{j-1}}, ..., \mathbf{c}_{\text{Opt}_1}, D_\pi)}{\max_{\mathbf{v}} I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{v} | D_\pi)} \max_{\mathbf{v}} I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{v} | D_\pi) \\ &\leq \frac{\min\{\log|\mathcal{Y}|, \log m\}}{\max_{\mathbf{v}} I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{v} | D_\pi)} I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{c}_{\text{Opt}_j} | D_\pi) \end{aligned} \tag{C.2}$$

where the last inequality follows from

$$\begin{aligned} &I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{c}_{\text{Opt}_j} | \mathbf{c}_{\text{Opt}_{j-1}}, ..., \mathbf{c}_{\text{Opt}_1}, D_\pi) \\ &\quad \leq \min\{\mathcal{H}(\mathcal{A}_{\mathbf{D}}(D_{\text{val}})), \mathcal{H}(\mathbf{c}_{\text{Opt}_j})\} \\ &\quad \leq \min\{\log|\mathcal{Y}|, \log m\}. \end{aligned}$$

Let $\mathbf{v}^* = \arg\max_{\mathbf{v} \in D_{\text{train}}} I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{v})$. We have

$$\begin{aligned} \max_{\mathbf{c}_{\text{Opt}_j}} I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{c}_{\text{Opt}_j} | D_\pi) &\geq I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{v}^* | D_\pi) \\ &\geq I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{v}^*) \end{aligned} \tag{C.3}$$

where the last inequality follows from the independence of $\mathbf{v}_i$'s in D_{train}. That is,

$$\begin{aligned} &I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}), D_\pi; \mathbf{v}^*) \\ &\quad = I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{v}^*) + I(D_\pi^{[l]}; \mathbf{v}^* | \mathcal{A}_{\mathbf{D}}(D_{\text{val}})) \\ &\quad = I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{v}^* | D_\pi) + I(D_\pi^{[l]}; \mathbf{v}^*). \end{aligned}$$

The independence of $\mathbf{v}_i$'s implies that $I(D_\pi; \mathbf{v}^*) = 0$. Hence (C.3) follows.

In the next step, we let $\theta = \frac{1}{I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{v}^*)}$ where $\mathbf{v}^* = \arg\max_{\mathbf{v} \in D_{\text{train}}} I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{v})$. Combining (C.2) and (C.3), we further have

$$\begin{aligned} &I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{c}_{\text{Opt}_j} | \mathbf{c}_{\text{Opt}_{j-1}}, ..., \mathbf{c}_{\text{Opt}_1}, D_\pi) \\ &\quad \leq \theta \min\{\log|\mathcal{Y}|, \log m\} \max_{\mathbf{v}} I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{v} | D_\pi). \end{aligned} \tag{C.4}$$

We remind that our update rule is simply

$$\mathbf{c}_{\pi_{T+1}} \coloneqq \{\arg\max_{\mathbf{v}} I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{v} | D_\pi)\}.$$

Inserting this into (C.4) proves the Lemma. □

We now move to the proof of Corollary 1.

Proof. We start by noting that

$$\mathcal{H}(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}) | D_{\text{Opt}}) \geq \mathcal{H}(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}) | D_{\text{Opt}}, D_\pi).$$

We also note

$$\begin{aligned} \mathcal{I}(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); D_{\text{Opt}}) &= \mathcal{H}(\mathcal{A}_{\mathbf{D}}(D_{\text{val}})) - \mathcal{H}(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}) | D_{\text{Opt}}) \\ \mathcal{I}(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); D_{\text{Opt}}, D_\pi) &= \mathcal{H}(\mathcal{A}_{\mathbf{D}}(D_{\text{val}})) \\ &\quad - \mathcal{H}(h\mathcal{A}_{\mathbf{D}}(D_{\text{val}}) | D_{\text{Opt}}, D_\pi). \end{aligned}$$

Hence

$$I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); D_{\text{Opt}}) \leq I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); D_{\text{Opt}}, D_\pi). \tag{C.5}$$

We further proceed with (C.5) as follows.

$$\begin{aligned} &I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); D_{\text{Opt}}) \\ &\quad \leq I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); D_{\text{Opt}}, D_\pi) \\ &\quad = I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); D_{\text{Opt}}, D_\pi) - I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); D_\pi) \\ &\qquad + I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); D_\pi) \\ &\quad = I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); D_{\text{Opt}} | D_\pi) + I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); D_\pi) \\ &\quad = \sum_{j=1}^{t} I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{c}_{\text{Opt}_j} | \mathbf{c}_{\text{Opt}_{j-1}}, \ldots, \mathbf{c}_{\text{Opt}_1}, D_\pi) \\ &\qquad + I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); D_\pi) \end{aligned} \tag{C.6}$$

where the last equality follows from the telescopic sum with $\mathbf{c}_{\text{Opt}_j} \in D_{\text{Opt}}$.

Using Lemma C.1, (C.6) can be followed by

$$\begin{aligned}
&I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); D_{\text{Opt}}) - I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); D_\pi) \\
&\leq \sum_j I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{c}_{\text{Opt}_j} | \mathbf{c}_{\text{Opt}_{j-1}}, \ldots, \mathbf{c}_{\text{Opt}_1}, D_\pi) \\
&\leq \sum_j \theta \min\{\log|\mathcal{Y}|, \log m\} \max_{\mathbf{c}_{\text{Opt}_j}} I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{c}_{\text{Opt}_j} | D_\pi) \\
&\leq \theta \min\{\log|\mathcal{Y}|, \log m\} \sum_j I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{c}_{\pi_{T+1}} | D_\pi) \\
&\leq t\theta \min\{\log|\mathcal{Y}|, \log m\} I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{c}_{\pi_{T+1}} | D_\pi) \\
&\leq t\theta \min\{\log|\mathcal{Y}|, \log m\} \big(I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{c}_{\pi_{T+1}} \cup D_\pi) \\
&\quad - I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); \mathbf{c}_{\pi_{T+1}} \cup D_\pi) \big).
\end{aligned} \tag{C.7}$$

We further let $\Delta_T = I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); D_{\text{Opt}}) - I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); D_\pi)$. (C.7) becomes:

$$\Delta_T \leq t\theta \min\{\log|\mathcal{Y}|, \log m\}(\Delta_T - \Delta_{T+1}). \tag{C.8}$$

Arranging the terms of (C.8), we have

$$t\theta \min\{\log|\mathcal{Y}|, \log m\} \Delta_{T+1} \leq (t\theta \min\{\log|\mathcal{Y}|, \log m\} - 1)\Delta_T$$

and hence

$$\begin{aligned}
\Delta_{T+1} &\leq \frac{t\theta \min\{\log|\mathcal{Y}|, \log m\} - 1}{t\theta \min\{\log|\mathcal{Y}|, \log m\}} \Delta_T \\
&\leq \ldots \\
&\leq \left(\frac{t \min\{\log|\mathcal{Y}|, \log m\}\theta - 1}{t\theta \min\{\log|\mathcal{Y}|, \log m\}} \right)^T \Delta_0.
\end{aligned} \tag{C.9}$$

Noting

$$\left(\frac{t\theta \min\{\log|\mathcal{Y}|, \log m\} - 1}{t\theta \min\{\log|\mathcal{Y}|, \log m\}} \right)^l \leq \exp(-l/t\theta \min\{\log|\mathcal{Y}|, \log m\})$$

we have

$$\begin{aligned}
\Delta_{T+1} &\leq \exp(-T/t\theta \min\{\log|\mathcal{Y}|, \log m\})\Delta_0 \\
&= \exp(-T/t\theta \min\{\log|\mathcal{Y}|, \log m\}\gamma) I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); D_{\text{Opt}}).
\end{aligned}$$

By the definition of Δ_T, we therefore have

$$\begin{aligned}
&I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); D_\pi) \\
&\quad \geq I(\mathcal{A}_{\mathbf{D}}(D_{\text{val}}); D_{\text{Opt}})(1 - e^{-\frac{T}{t\theta \min\{\log|\mathcal{Y}|, \log m\}}}).
\end{aligned} \tag{C.10}$$

which proves the Corollary 1. □