ITERATIVE SKYLINE COMPUTATION THROUGH NOISY COMPARISONS

Abishrant Panday
abishrantpanday@college.harvard.edu

Joyce Tian
joycetian@college.harvard.edu

December 10, 2019

1 Introduction

Given a set of points S in $\mathbb{R}^d$, the *skyline* is defined as the subset $K \subseteq S$ such that no point in K is Pareto-dominated by any point in S and that all points in K Pareto-dominate all points in $S \setminus K$. Here, we define a point p as Pareto-dominating a point q if, for all dimensions $i \in [d]$, each coordinate satisfies $q_i \leq p_i$.

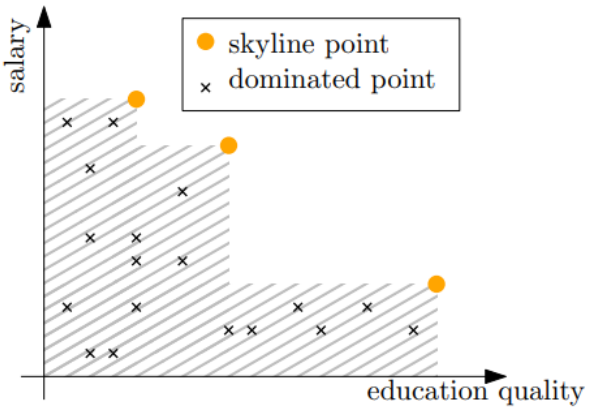

Figure 1: Example of a set of dimension 2 from [2] where the skyline points can be seen to dominate all points coordinatewise.

Given exact data, the problem of computing the skyline reduces to one of finding the vectors in S which are maximal. We, however, chose to study the question of skyline computation when the underlying data is uncertain: specifically, when two points p, q in S can only be compared through noisy coordinate-wise comparisons. Within this paradigm, we construct a *comparison oracle*, which we assume knows the base truth of each comparison but outputs incorrect answers with some probability $\delta < \frac{1}{2}$. Thus, given queries between the i^{th} coordinates of two points, the oracle will determine the strict ordering $q_i < p_i$ and then return a correct outcome for each query independently with probability δ.

Our project thus provides a synthesis of two recent theoretical papers with respect to their contributions on iteratively computing noisy skylines: "Skyline Queries with Noisy Comparisons" by Groz, Milo [1] and "Skyline Computation with Noisy Comparisons" by Mallmann-Trenn, Mathieu, Verdugo [2]. We then contribute to the theory of skyline

computation by showing the limitations of a proof in [2] and using those insights for improved implementation. To compare, we implement three algorithms: **SkylineHighDim**, **Skysample**, both implemented using the framework of **SkylineComputation**, along with a brute-force control algorithm. We add implementation-level improvements, modify the algorithms for real-world convergence, and stress-test on generated data. The implementation can be found at:

`https://github.com/Saffr0n1/Iterative-Noisy-Skyline-Comparisons.`

Section 2 provides preliminary definitions, section 3 gives motivations and a simplified model, and section 4 references predecessor noisy comparison models. Section 5 provides the theoretical foundations of iterative algorithms for noisy skyline and within it, **Theorem VI** in section 5.3.2 provides theoretical contribution with an implementation focus. Finally, section 6 consists of original implementation and model testing, along with results and their interpretations.

2 Preliminary Definitions

We first present the definitions that are used in the theoretical and algorithmic discussion of all subsequent models:
Sample Set: The sample set, X, is the set of all points, for which we wish to return the skyline. For ease of reference, we will denote $|X| = n$.
Dimension: The dimension, d is defined by the fact that X is a subset of $\mathbb{R}^d$; d is the number of attributes $|v|$ of any point $v \in X$, and can be thought of as the number of relevant categories we wish to examine rankings on.
Skyline Cardinality: The cardinality k of the skyline, which is the unknown number of points that comprise the skyline $K \subseteq X$.
Error Tolerance: The error tolerance, δ, represents the maximum error of our algorithm to return the correct skyline.
Preference: We define the actual preference ranking between $a, b \in X$ on dimension i as $a \prec_i b$ if $a[i] < b[i]$, $a \sim_i b$ if $a[i] = b[i]$, and $a \succ_i b$ if $a[i] > b[i]$. A soft preference ranking like $a \preceq_i b$ implies $a[i] \leq b[i]$, and similarly with $a \succeq_i b$.
Comparison Oracle Error Tolerance: The error tolerance, $p < \frac{1}{2}$, represents the maximum error of an oracle comparison's response to a boolean query. We will specifically query oracles in the form $x \succeq_i y$ for $x, y \in X$, $1 \leq i \leq d$. We will assume a stricter bound $p < \frac{1}{6}$ such that the bounds for some proofs hold easily.
Pareto Dominance: A point v is considered Pareto dominant over another point v' if it satisfies $v' \succeq_i v$ for all $1 \leq i \leq d$; v is strictly Pareto dominant over v' if it satisfies $v' \succ_i v$ for at least one i value.
Pareto Frontier: The Pareto frontier, or skyline, is the set of points $C \subseteq X$ such that all $c \in C$ are Pareto dominant over all $x \in X \setminus C$ and for all pairs of points $c \in C, x \in X$, x is not Pareto-dominant over c.

In addition, we define and motivate the following analysis-specific parameters:

 # Rounds: The number of rounds are the number of times comparisons must be executed and processed in parallel. The motivation for examining this metric is to ascertain the value of using a given algorithm in crowdsourcing scenarios; the batching of questions in these models provides some measure on the time required to complete the scenario in real-world.
Oracle Complexity: The oracle complexity of an algorithm is a measure of the number of calls to the oracle that are required within the computation. In our noisy skyline setup, which can be used to model scenarios such as crowdsourcing information, it is assumed that each call to the oracle has a cost, which is usually monetary in tools such as Amazon Mechanical Turk, which globally exports many user-classification tasks. Thus, we will look to minimize the oracle complexity while retaining a high probability of returning the correct skyline.
Computational Complexity: The computational complexity is our traditional measure of an algorithm's processing time with respect to the parameters it utilizes. The computational complexity generally determines an upper bound for oracle complexity as well.

3 Motivation

In general, the problem of determining a skyline given a set of points is motivated by the fact that points in the skyline can be used to determine the "best"—based on some measure of comparing values at each coordinate–items within the set over many dimensions. As an example, if we are considering the problem of ranking different universities, we might consider the dimensions of student happiness, endowment, faculty size, and future student income. The skyline over these dimensions will then give a measure of how these universities compare to each other on the basis of these factors.

3.1 Noiseless Skyline Model

The first model of the general skyline problem thus consists of the *noiseless skyline*, where we have a set of points $S \in \mathbb{R}^d$ and can compare any two $p, q \in S$ by querying an oracle that is always correct. If we define $|S| = n, |K| = k$, we propose the following naïve algorithm with computational complexity $O(knd)$, which we will use as a model for more complex noisy algorithms:

- Define S_i as the set of i greatest skyline points in lexicographic order and R_i as the set of points not dominated by S_i
- Note then that $S_0 = \{\}$ and $R_0 = S$, the whole set.
- for $i \in \{0, 1, \ldots, k\}$:
- Take the largest $r \in R_i$ based on lexicographic order, which has computational complexity $O(dn)$. Construct $S_{i+1} = S_i + \{r\}$. To construct R_{i+1}, iterate through R and remove all items dominated by r, also in $O(dn)$ time.

3.2 Adding Noise to Model Real Data

By adding noise to the queries, the noisy skyline model reflects scenarios where the comparison of attributes between different items is not deterministic or simple. In particular, this paradigm models an important source of data labeling and categorization in large-scale analysis: crowdsourcing. This scenario thus motivates the definition of a non-deterministic oracle, due to the addition of uncertainty in the way a general crowd can compare any two features, and also highlights the importance of oracle complexity, defined above, since obtaining the correct answer to a comparison often requires multiple people's answers—and querying the crowd is generally of higher cost than computational considerations.

Furthermore, we find many applications of such crowdsourcing within fields like machine learning, which require large, accurately-labeled datasets. As an explicit example, one can consider the creation of the ImageNet database–a large image database constructed on the WordNet synset structure [5]. The paper, which utilizes Amazon Mechanical Turk, combats the problem of user uncertainty by presampling some examples from each category and using their labels to generate a confidence table that can then be referenced. These user labeling tasks can be reformulated in our notion of binary oracle queries in a labeling-with-options manner. Thus, computation of skylines with noisy comparisons, which seeks to minimize the oracle complexity, may prove useful in such tasks, since querying the oracle comes with financial cost.

4 Basic Noisy Comparisons

Before determining the theoretical bounds for noisy skyline models, we introduce other operators which have previously been studied and their complexity bounds as determined in [3]. However, we first note a naïve algorithm for noisy comparisons:

Let algorithm A return the skyline in a noiseless setting with computational complexity $f(n)$. Then, for an error tolerance δ, we can construct an algorithm A' in the noisy case by calling each comparison in $A \log\left(\frac{f(n)}{\delta}\right)$ times. The majority vote is then used for the decision. Thus, each noiseless comparison is now represented by a comparison that has tolerance $\frac{\delta}{f(n)}$. By the union bound we thus get an overall error of δ for the converted algorithm A'. For the specific problems below, we provide better bounds.

We consider the following noisy comparison problems and their computational complexity bounds, as determined in [3]:

- **OR:** Given n Boolean inputs, we decide if one of them is true. $\Theta(n \log \frac{1}{\delta})$
- **MAX:** Given n items, we return the largest one. $\Theta(n \log \frac{1}{\delta})$
- **SORT:** Given n items, we return their sorted order. $\Theta(nlog\frac{n}{\delta})$
- **BINARY SEARCH:** Given an ordered list S with n items and an item v, we return the successor of v in S. $\Theta(log\frac{n}{\delta})$

5 Theoretical Discussion of Skyline Computation

In order to motivate our implementation-level heuristics, experimental bounds, and implementation improvements, along with a divergence between theoretical and empirical findings, we first discuss the theory of two skyline computation algorithms presented in the papers.

5.1 Dominance Testing

Before we consider the actual algorithms, we prove bounds for the oracle complexity of dominance testing.

Theorem I Let $C \subseteq X$ and choose a point $p \in X$. We can check that $p \preceq C$ with error probability δ in oracle complexity $O(d|C| \log \frac{1}{\delta})$.

Proof We will derive this complexity by considering dominance tests as compositions of **OR** queries and then utilize the bounds from section 4. In particular, given $q \in C$, we can define the dominance test between p, q by:

$$p \preceq q = \bigwedge_{i=1}^{d} p_i \leq q_i \tag{1}$$

We then utilize the **OR** algorithm to check each test in $O(d)$ with an error probability of $\delta = \frac{1}{3}$. Now, we can construct the dominance test between p, C via:

$$p \preceq C = \bigvee_{q \in C} \bigwedge_{i=1}^{d} p_i \leq q_i \tag{2}$$

We let each dominance test $p \preceq q$ define an oracle for the **OR** algorithm. We can thus rewrite:

$$p \preceq C = \bigvee_{q \in C} p \preceq q \tag{3}$$

This can now be checked in $O(d|C| \log \frac{1}{\delta})$ time, once again by the bound for **OR**.

Theorem II Let $C \subseteq X$ and choose a point $p \in X$. If we know the order of C on each dimension, checking $v \preceq C$ can be done with error probability δ in oracle complexity $O\left(d \log \frac{d|C|}{\delta}\right)$.

Proof We are given the order of C along each dimension $i \in [d]$. Thus, by direct application of **BINARY SEARCH** from section 4 we compute, for each i, the successor of v_i on the i^{th} dimension of entries in C with error probability $\frac{\delta}{d}$ in $O(log\frac{d|C|}{\delta})$. Note, however, that once this has been determined, although the computation $v \preceq C$ does not require further calls to the oracle, it may still require significant computation.

5.2 Control Algorithm

We define the control algorithm **FullSort**, which functions by sorting the data along each dimension with a given error probability and then utilizing any noiseless skyline algorithm to compute the final set based on the orderings.

```
1 FullSort(X, δ):
2   sort X along each dimension
3   compute skyline with noiseless algorithm (assuming orderings are correct)
```

We note that we are interested in determining the oracle complexity of this function, which is impacted only by step 2. After ordering the points in X per dimension, the noiseless skyline computation then requires no additional calls to the oracle.

Oracle Complexity: Each dimension is sorted in (1) with error probability $\frac{\delta}{4}$. To do this, **SORT** is utilized from section 4, which gives us the oracle complexity of

$$O\left((nd)\log\frac{(nd)}{\delta}\right) \tag{4}$$

Then, we can take the union bound to get that the orderings are correct with error probability δ. After this, computing the skyline, line 3 can be done with an algorithm such as that from section 3.1. **FullSort** thus provides a baseline model for determining a skyline in the noisy comparison model which we will now build upon.

5.3 Iterative Skyline Recovery

The main algorithms we will consider from both a theoretical and implementation standpoint determine the skyline given an input set X by computing each point on the skyline, one by one. The implementation of each algorithm as outlined in the paper can be found in the appendix (section 8).

5.3.1 Skysample

We first consider **Skysample**, which is reproduced in the appendix from [1], section 8.1. This function computes iteratively each maximum point (in lexicographic order) among the set of points that are not dominated by the already found skyline points, and hence are potential additions to the skyline. In addition, it is of note that **Skysample** can't output exactly the whole skyline, since $|k|$ is unknown immediately from the dataset. Instead, an estimated $k'' \geq k$ is used for the computation. We now prove bounds for the oracle and computational complexity of **Skysample**.

Theorem III **Skysample** computes the first $min\{|k|, |k''|\}$ points in $O(kdn\log\frac{dk''}{\delta})$ oracle complexity.

Proof First, we consider the oracle for lexicographic comparisons. We denote by S_i the set of i items in the skyline of S, K with the highest lexicographic rank. This oracle can be run in $O(d)$ time with an error probability $\delta = \frac{1}{3}$ using **Newman's OR** algorithm.[1] The algorithm itself then utilizes this oracle, along with the dominance-testing oracle from section 5.1 to iteratively compute S_{i+1} from S_i. In particular, we utilize **Theorem II**, from section 5.1, to see that we obtain a new skyline item in $O(dn\log\frac{dk''}{\delta})$ with error probability $\frac{\delta}{k''}$. Thus, we see that for all the skyline items, we

[1] Trust preserving algorithms are ones whose error tolerance is determined by those of input oracles. Newman showed that **OR** can be computed in O(n) with a trust-preserving algorithm.

sum

$$O\left(\sum_{i<k} dn \log \frac{dk''}{\delta}\right) = O\left(kdn \log \frac{dk''}{\delta}\right) \tag{5}$$

We union-bound the error probabilities and thus have that the oracle complexity is $O = \left(kdn \log \frac{dk''}{\delta}\right)$ with error probability δ.

Theorem IV **Skysample** computes the first $min\{|k|, |k''|\}$ points in $O(k^2 dn \log \frac{k''}{\delta})$ computational complexity.

Proof With the same setup as above, we now use **Theorem I** from section 5.1 to see that we get the new skyline in $O = \left(idn \log \frac{k''}{\delta}\right)$ with error probability $\frac{\delta}{k''}$. Thus, we see that for all the skyline items, we sum

$$O\left(\sum_{i<k} idn \log \frac{k''}{\delta}\right) = O\left(\frac{(k-1)k}{2} dn \log \frac{k''}{\delta}\right) = O\left(k^2 dn \log \frac{k''}{\delta}\right) \tag{6}$$

We union-bound the error probabilities and thus have that the oracle complexity is $O = \left(k^2 dn \log \frac{k''}{\delta}\right)$ with error probability δ.

5.3.2 SkylineHighDim

In Mallmann-Trenn, Mathieu, and Verdugo [2], the authors construct **SkylineHighDim** along the same iterative paradigm as the **Skysample** of Groz and Milo [1]. However, instead of a single function that iteratively computes the maximum not-dominated point, the algorithm separates the two tasks. As illustrated in Appendix 8.2, **SkylineHighDim** first chooses a random point p that is not dominated by points in the current skyline and then computes a lexicographic maximum within the points that dominate p. This is performed by calling the **MaxLex** function, whose higher-level implementation from [2] is presented in 8.3, stores a counter for each point $q \in X$, wherein each iteration consists of a pairwise comparison between the $q_1, q_2 \in X$ whose counters have the largest values, and a consequent comparison between $q' = \max_{>_{lex}}\{q_1, q_2\}$ and p. The counter for q' is increased if q' dominates p and decreased if it doesn't dominate p; the point in the pair which is less lexicographically $q \neq q'$ is similarly decremented. The values by which the counter increments or decrements are chosen so that the counter of the desired output mimics random walk biased up with non-uniform step size and the counters of all other points perform random walks biased downwards. The points with the largest counters are then routinely compared. From an algorithmic standpoint, lines 3,4 of **MaxLex** call for an arbitrary argmax function, whose implementation we vary in order to affect convergence results.

We now theoretically analyze **SkylineHighDim** and **MaxLex**, presenting proofs of their correctness and bounds and showing the areas where we will focus on improving heuristics and implementing algorithm-level changes. Specifically, we note problems within the provided proof for algorithm **MaxLex**, which then leads us to implementation improvements, especially with regards to convergence.

Theorem V Let $k \geq |Skyline(X)|$. Then, **SkylineHighDim**(k, X, δ) outputs the correct skyline with probability bounded by $1 - \delta$ and with expected oracle complexity $O = \left(k^2 d \log \frac{kn}{\delta} + ndk \log \frac{k}{\delta}\right)$.

Proof We will first show that **SkylineHighDim** returns $min\{k, |Skyline(X)|\}$ points with error tolerance δ. To do this, we consider a separation of the algorithm into two 'phases,' as denoted in 8.2, and determine the sources of error from each at any step i.

- **Phase 1:** The possible error in this phase occurs on line 6 in the pseudocode, where **SetDominates** (a helper function that determines, given a set S, point p, and two error parameters, whether $\exists q \in S$ such that q

dominates p) can incorrectly classify a skyline point as dominated, with error probability at most $\frac{\delta}{4k}$ or a dominated point as not dominated, which happens with probability at most $\frac{|X|\delta}{4k|X|} = \frac{\delta}{4k}$.[2]

- **Phase 2:** In this phase, the only possible error occurs in line 11, where **MaxLex** returns an incorrect computation. This happens with probability at most $\frac{\delta}{2k}$, which we will analyze in the proof of the following theorem.

Thus, we see that we can sum over all i and union-bound, which tells us that the probability of the function being correct is at least

$$1 - \sum_i \left(\frac{\delta}{4k} + \frac{\delta}{4k} + \frac{\delta}{2k} \right) = 1 - k \left(\frac{\delta}{k} \right) = 1 - \delta \tag{7}$$

Now, we analyze the oracle complexity of **SkylineHighDim**. To do this, we note that each point $p \in X$, when passed to a call of **SetDominates**, will call at most two oracle comparisons. We then use the complexity result outlined in the footnotes to get an expected query complexity $O(|S_i| d \log \frac{4k|X|}{\delta})$. We now note $O(|S_i|) = O(k)$ since S_k is the minimum of k and the skyline, both of which are $O(k)$. Moreover, $O(\log \frac{4k|X|}{\delta}) = O(\log \frac{k|X|}{\delta})$. We thus have that the oracle complexity for each skyline point is

$$O\left(|S_i| d \log \frac{4k|X|}{\delta} \right) = O\left(kd \log \frac{k|X|}{\delta} \right) \tag{8}$$

Then, for each non-skyline point the expected query complexity at each step i is $O(d \log \frac{|S_i|k}{\delta})$. This expression can be written

$$O(d \log \frac{|S_i|k}{\delta}) = O(d \log \frac{k}{\delta} \log |S_i|) = O(d \log \frac{k}{\delta}) \tag{9}$$

There, we obtained the second equality by noting that $O(|S_i|) = O(k)$ since $|S_i| \leq k$. Finally, we note that in the worst case, the number of non-skyline points the algorithm would have to query to the oracle is $O(|X|)$. Thus, we have the oracle complexity for non-skyline points as $O(|X| d \log \frac{k}{\delta})$.
Finally, we conclude by summing the complexities over $i \in \{1, \ldots, k\}$ and combining them to get

$$O\left(k^2 d \log \frac{k|X|}{\delta} + k|X| d \log \frac{k}{\delta} \right) = O\left(k^2 d \log \frac{kn}{\delta} + ndk \log \frac{k}{\delta} \right) \tag{10}$$

Theorem VI There is an algorithm for finding the lexicographical maximum of the values which dominate p in X, **MaxLex**(p, X, δ), that is correct with maximum error probability δ and expected query complexity $O(|X| d \log(1/\delta))$.

Note We identify an error in the given proof for this theorem and utilize that insight to generate implementation improvements.

Incorrect Proof from [2] We first consider a discrete time Markov chain of a random walk $(Z_{\tau(t)})_{\tau(t) \geq 0}$ which begins at $Z_0 = \log(1/\delta)$, incrementing by $+1$ with probability p and -1 with probability $1 - p$ until it hits 0 or b, its absorbing states. For $p \neq \frac{1}{2}$, b, and $T = \min\{t \geq 0 \mid Z_t \in \{0, b\}\}$, [4] found that

$$P(Z_T = 0) = \frac{\left(\frac{1-p}{p}\right)^s - \left(\frac{1-p}{p}\right)^b}{1 - \left(\frac{1-p}{p}\right)^b} \tag{11}$$

For $p \leq \frac{2}{3}$, $\left(\frac{1-p}{p}\right) \leq \frac{1}{2}$, this yields:

$$P(Z_T = 0) \leq \frac{2^{-\log(1/\delta)} - 2^{-b}}{1 - 2^{-b}} \to 2^{-\log(1/\delta)} = \delta \tag{12}$$

[2]This comes from a lemma in [2] that **Dominates**, which has an analogous structure to **SetDominates** with respect to oracle calls, has expected query complexity O(d).

As such, we find that $(Z_t) > 0$ with probability $1 - \delta$ for $p \geq \frac{2}{3}$.
We will attempt to bound the potential error within the **MaxLex** algorithm, as described in [2], using the probability of the absorbing state in $(Z_t)_{t\geq 0}$ above. Let p^* denote the lexicographical maximum of the values which dominate p in X, and let q_1^i and q_2^i denote the two values chosen to be compared with each other and potentially with p in time-step i. We consider the discrete-time Markov chain $(Y_{2t})_{t\geq 0}$ for time-steps t, such that it reflects the value of the counter of p^*, $c(p^*)$, every 2 time-steps. As such, given that p^* is chosen to be considered, we denote I_{inc}^t as the indicator event that p^* is incremented positively by $\frac{1}{2}$, at time-step t; $\bar{I}_{inc}^t$ is thus the indicator event that p^* is decremented by 1, at time-step t, where the increment values are specified in **MaxLex**. We then examine $P(Y_{2(t+1)} = i \mid Y_{2t} = i - 1)$, the probability of being checked successfully as the lexical maximum and being dominant over p in two successive time-steps. The error probability for this, should p^* be chosen twice, is bounded by the error for lexical comparison, **Lex**, and the error for dominance checking, **Dominance**, should the lexical comparison succeed; both of these sub-algorithms' errors are strictly bound by $\frac{1}{16}$ by construction, and thus the upper bound for the probability that p^* is not incremented in one time-step is $\frac{1}{16} + \frac{15}{16} \cdot \frac{1}{16} < \frac{1}{6}$. Thus,

$$P(Y_{2(t+1)} = i \mid Y_{2t} = i - 1) \geq \left(1 - \frac{1}{6}\right)^2 > \frac{2}{3} \geq P(Z_T = 0) \tag{13}$$

In order for p^* to not be outputted, $c(p^*) \leq -2$ at some timestep t, and as such the probability for this event is encapsulated in the event that $Y_{2t} \leq 0$. This seems to yield the upper bound for **MaxLex**'s error as:

$$P(p^* \text{ not outputted by } \textbf{MaxLex}) \leq P(Y_{2t} \leq 0) \leq P(Z_t \leq 0) = P(Z_t = 0) \leq \delta \tag{14}$$

Given this, we then note that **MaxLex**'s increments are constructed such that at the end of each time-step i, the non q_1^i and q_2^i values are all within 1 of each other, and that the sum of the counters is at most $|X|\log(1/\delta) - i/2$, as there is a net decrement of at least $-\frac{1}{2}$ per time-step. If so, then at time-step $i \geq 4(|X|+3)(\log(1/\delta)+3)$, the sum of the counters will be at most $-2|X|\log(1/\delta) - 18|X| < -2|X|(\log(1/\delta) + 1)$; since $\log(1/\delta) > 1$, this means that every counter which does not map to q_1^i or q_2^i must have value < -2. Specifically, $c(q_1^i) + c(q_2^i) \leq 2\log(1/\delta) + 2(|X| + 3)(\log(1/\delta) + 3)$. As such, after $6(|X| + 3)(\log(1/\delta) + 3)$ more iterations, we note that at least one of $c(q_1)$ or $c(q_2)$ must be ≤ -1, which is when the algorithm terminates. Thus, we bound the number of timesteps above by $10(|X| + 3)(\log(1/\delta) + 3)$, wherein each time-step takes expected $O(d)$ time to compute **Lex** and **Dominance**. As such, the overall expected computational complexity is $O(|X|d\log(1/\delta)$.

However, the above analysis of $(Y_t)_{t\geq 0}$ assumes that $p^* = q_1$ or $p^* = q_2$ in the round examined, as otherwise $c(p^*)$ is not chosen and thus will not be able to increment positively in two consecutive rounds; thus, we note that should p^* be incorrectly incremented and tie with multiple other counters in value, the probability of being chosen as q_1 or q_2 may decrease on the order of $\frac{1}{n}$ (with the worst-case scenario of all n counters being tied in value) if $argmax$ randomly chooses among tied counters, or a worst-case of 0 if one or more tied-value counters are closer to the front of the dataset than p^*. [2] fails to account for the multi-tie situation decreasing probability of being chosen and thus the probability space for incrementing/decrementing, which leads to $P(Y_{2(t+1)} = i \mid Y_{2t} = i - 1)$ being potentially less than $\frac{2}{3}$ and thus failing to yield an error of at most δ.

5.4 Skyline Computation and SkyHighDim-Search

Finally, we note that both aforementioned algorithms operate on inputs that include X and δ but not k or $|k|$, the actual skyline. The motivation of this approach is to find all the skyline items in an efficient way while having an unknown skyline cardinality. Thus, they have to run on values $k'' \geq k$. However, once either of the previous algorithms are implemented, in order to actually compute the skyline, $Skyline(X)$, both **Skyline Computation** and **SkyHighDim-Search**, of which only the former's code is presented in Appendix 8.4 since the approaches are very similar, guess an upper bound for $|Skyline(X)|$ by increasing k'' until the size of the skyline computed with k'' is less than the size of k''.

Theorem VII The algorithm **Skyline Computation** computes with oracle complexity $O\left(dkn \log \frac{dk}{\delta}\right)$ and computational complexity $O\left(dk^2 n \log \frac{k}{\delta}\right)$, with error tolerance δ.

Proof To see that the error tolerance is δ, we note that at any step i, the probability of returning the incorrect result is $\frac{\delta}{2^i}$ by definition of the algorithm. Thus, the error probability is bounded by

$$\sum_{i \leq \log \log k} \frac{\delta}{2^i} \leq \delta \tag{15}$$

To calculate the oracle and computational complexities, we use the result of **Theorems III, IV** in section 8.1 to get that the oracle complexity is

$$O\left(dn \sum_{i=1}^{\lfloor \log \log k \rfloor} k_i \log \frac{dk_i 2^i}{\delta}\right) + O\left(dkn \log \frac{dk}{\delta}\right) = O\left(dkn \log \frac{dk}{\delta}\right) \tag{16}$$

and the computational complexity is

$$O\left(dn \sum_{i=1}^{\lfloor \log \log k \rfloor} k_i^2 \log \frac{k_i 2^i}{\delta}\right) + O\left(dk^2 n \log \frac{k}{\delta}\right) = O\left(dk^2 n \log \frac{k}{\delta}\right) \tag{17}$$

6 Implementation and Results

6.1 MaxLex Implementation

Because the bound for **MaxLex**(p, X, δ) using the probability of a random walk's absorption state does not necessarily hold, we alter the algorithm's counter updates from their specified decrement of 1 decrement and increment of 0.5. We implement this by changing all constant update values to derive from a vector u passed in as argument, where the default update values correspond to $u_0 = (1, 0.5, 1, -2)$. Specifically, we found that increasing the increment value significantly reduced error probabilities by favoring objects with Pareto dominance over p and thus differentiating potential p^* object from the overall set X. Further, a slightly stronger emphasis on decrementing in the case of not Pareto dominating p (e.g. decrementing by 1.1 rather than 1) also tended to help with error reduction, which may suggest that the latter half of the algorithm's, which measures a potential p^* object's dominance over p, may be more important than the comparison of the two objects with the highest-valued counters (particularly in the case of ties).

These increments/decrements do not seem to show consistent improvement over all n, k, or d values, and we postulate that the optimal update values likely depend on these parameters in some way. Holistically, we note that larger increment/decrement values seem to yield more significant error improvements for larger n and larger k; this intuitively follows from the proposed worst-case scenario of an n-way tie in counter values, wherein the probability of p^* being chosen to be compared is $\frac{2}{n-1}$, which decreases rapidly with n and is intuitively more likely to occur should k be large and thus most objects in X be a skyline item. Specifically, lexicographical ordering may be less informative in this scenario than dominance, which may filter out more of the X values. In Table 1, we show a particularly strong case for the update value importance in **MaxLex** within a relatively small dataset S which has many skyline items which are not Pareto-dominant over $(1, 99)$ but are lexicographically greater than it. In this case, it is clear that the changing of update values to favor the dominance condition vastly improves error rates from being close to 1 to nearly 0, with little change in mean oracle queries and thus similar complexity.

We further noted that [2] did not specify the formalism for choosing $\text{argmax}_{q \in S}(c(q))$ in the event of a tie between multiple counter values within the **MaxLex** algorithm (lines 3,4 in 8.3). We thus experimented with the output of **MaxLex** when using an **ArgmaxFirst** formalism, which simply chooses the first counter in the list with

maximum value, and when using an **ArgmaxRandom** formalism, which chooses at random one counter from those with maximum value. As shown in Table 1, **ArgmaxRandom** significantly outperforms **ArgmaxFirst**, particularly in the case with changed update values, where it nearly halved the error rate. This suggests that random tie-breaking is more useful in helping differentiate q^*, which makes sense as it allows for more symmetry in the solution space.

Furthermore, inspection of several smaller cases showed that **MaxLex** sometimes terminated before all but one counter had values less than -2, which was the condition needed for the error bound in **Theorem VI**. As such, we also examined the impact of adding an additional condition within **MaxLex** that all $c(q) \leq -2$ for $q \in X \setminus q_1$ before exiting the while-loop, in order to explicitly check the condition needed for the other counter values to complete the proof of p^* error. This condition seemed to improve mean error rates, particularly in the case of the augmented update values; the latter finding may simply be due to the higher volatility surrounding the update values leading to the need for stricter conditions on convergence to p^*. However, we also note that the mean and maximum number of oracle queries significantly rise upon calling for this condition, which signals that the increased iterations needed to satisfy this condition may not be worth the marginal decrease in error.

MaxLex Statistics							
Argmax Formalism	Update Values	Additional Condition	Mean Error Rate	Mean Oracle Queries	Maximum Oracle Queries	Mean Probability of Incrementing p^*	Median Probability of Incrementing p^*
First	$(1, 0.5, 1, -2)$	No	0.9978	281.01	380	0	0
Random	$(1, 0.5, 1, -2)$	No	0.8525	291.5	413	0.0476	0.0625
Random	$(1, 0.5, 1, -2)$	Yes	0.8203	350.5	491	0.0491	0.0526
First	$(1, 4, 3, -2)$	No	0.0206	165.04	288	0.242	0.25
Random	$(1, 4, 3, -2)$	No	0.0131	156.3	330	0.224	0.25
Random	$(1, 4, 3, -2)$	Yes	0.0049	283.169	425	0.2651	0.2667

Table 1: Statistics for various implementations of **MaxLex** run $10,000$ times on the dataset $S = \{(1, 1), (2, 2), (3, 5), (7, 7), (1, 99), (99, 1), (97, 5)\}$ for value $p = (1, 99)$, which is meant to return $(1, 99)$.

The implementation of function **MaxLex**, along with **ArgmaxFirst** and **ArgmaxRandom** can be analyzed:

```
def argmax_lex(a):
    return max(enumerate(a), key=lambda a:a[1])[0]
def argmax_rand(a):
    b = np.array(a)
    return np.random.choice(np.flatnonzero(b == b.max()))
def MaxLex(p, S, delta, deltaMain, use_argmax_lex = True, use_update = (1, 0.5, 1, -2),
     expected=None, use_cond = False):
    if len(S) == 1:
        return S[0], 0
    c = []
    for i in range(0,len(S)):
        c.append(math.log(1/delta))
    compl = False
    num_calls = 0
    if use_argmax_lex:
        argmax = lambda x: argmax_lex(x)
    else:
        argmax = lambda x: argmax_rand(x)
    rounds = 0
    if expected:
```

```
    ind = S.index(expected)
    prev = c[ind]
num_increased = 0
while not compl:
    q1Star = argmax(c)
    q1 = S[q1Star]

    cStar = c[:q1Star] + c[q1Star + 1:]
    q2Star = argmax(cStar)
    q2Star = q2Star + 1 if q2Star >= q1Star else q2Star
    q2 = S[q2Star]

    cond1, calls1 = Lex(q1,q2,delta)
    num_calls += calls1
    if cond1:
        x = q1
        xStar = q1Star
        y = q2Star
    else:
        x = q2
        xStar = q2Star
        y = q1Star

    c[y] = c[y] - use_update[0]

    cond2, calls2 = Dominates(x, p, deltaMain)
    num_calls += calls2
    if cond2:
        c[xStar] = c[xStar] + use_update[1]
    else:
        c[xStar] = c[xStar] - use_update[2]

    cond = (c[q2Star] <= use_update[3])
    if len(c) > 2 and use_cond:
        remaining = c[:min(q1Star, q2Star)] + c[min(q1Star, q2Star)+1:max(q1Star, q2Star)] +
            c[max(q1Star, q2Star)+1:]
        cond = cond and np.all([x <= -2 for x in remaining])
    if cond:
        compl = True
    rounds += 1
    if expected:
        curr = c[ind]
        if curr == prev + use_update[1]:
            num_increased += 1
        else:
            num_increased = num_increased
        prev = curr
print(c, S)
print(num_increased/rounds)
return S[argmax(c)], num_calls
```

6.2 Skysampling and SkylineHighDim Implementation

The **Skysampling** algorithm computes $\max_{<_{lex}}\{p|p \not\preceq S\})$ by calling **MaxLex**$(p, S, \frac{\delta}{k''})$, rather than the Oracle comparison method specified in [1]. This allowed us to better compare the relative efficiency of **Skysampling** and **SkylineHighDim** without the additional differences in lexicographic and dominance computations. We followed the format of [2] in the high-level implementation of **SkylineHighDim**. However, we consolidated both as being called from the general method of sampling for k specified Algorithm 5 in [1], which updates $k_i = k_{i-1}^2$ from $k_0 = 4$ for each i-th iteration, so as to again better compare the relative efficiency of **Skysampling** and **SkylineHighDim**. The algorithms are shown below:

```
def SkylineHighDim(k, X, delta, deltaMain, use_argmax_lex = True, use_update = (1, 0.5, 1, -2)):
    S = []
    C = X.copy()
    num_calls = 0
    for i in range(1,k+1):
        #Finding a point p not dominated by current skyline points
        found = False
        while len(C) > 0 and not found:
            p = C[random.randint(0,len(C) -1)]
            cond1, calls1 = SetDominates(S, p, delta/(4*k), delta/(4*k*len(X)), deltaMain)
            num_calls += calls1
            if not cond1:
                print(p, S, "not dominated")
                found = True
            else:
                print(p, S, "dominated")
                C.remove(p)
                # print(C)
        if len(C) == 0:
            return S, num_calls
        else:
            #Finding a skyline point that dominates p
            pStar, calls2 = MaxLex(p, C, delta/(2*k), deltaMain, use_argmax_lex = use_argmax_lex,
                use_update = use_update)
            num_calls += calls2
            C.remove(pStar)
            print(pStar, C)
            S.append(pStar)
    return S, num_calls
```

```
def skysample(khat, s, delta, error, use_argmax_lex = None, use_update = None):
    assert len(s) > 0
    sky = []
    dims = len(s[0])
    remaining = set(s)
    num_calls = 0
    for i in range(khat):
        # find non-dominated points
        to_remove = []
        for r in remaining:
            comp, calls = is_dominated(r, sky, delta, error)
```

```
            num_calls += calls
            if comp:
                to_remove.append(r)
        for r in to_remove:
            remaining.remove(r)
        if len(remaining) > 0:
            remaining = list(remaining)
            z, calls = MaxLex(remaining[0], remaining, delta/2, error)
            num_calls += calls
            sky.append(z)
            remaining = set(remaining)
            remaining.remove(z)
    return sky, num_calls
```

6.3 FullSort Implementation

The **FullSort** algorithm was implemented in two forms, one which takes general δ for the **FullSortControl** algorithm and one with $\delta = 0$ for the **NoiselessSkylineComputation** algorithm. The latter was used to find the actual skyline C for any input set X with perfect information, and thus provided a base to compare all other algorithm outputs to for accuracy. Meanwhile, the **FullSortControl** algorithm provided a baseline number of oracle complexity, computational complexity, and output accuracy which can be compared to our more complex algorithm using **Skysample** and **SkylineHighDim**.

```
def brute_force(s, delta, error):
    start = time.time()
    n = len(s)
    dims = len(s[0])
    optimal = []
    sorted_i = []
    optimal = []
    num_calls = 0
    for i in range(dims):
        s_i, calls = msort2(s, i, error)
        num_calls += calls
        sorted_i.append(s_i)

    changed = True
    while changed:
        changed = False
        for i in range(dims):
            optima_i = []
            compl = False
            curr = -1
            while not compl and len(sorted_i[i]) > 0:
                dominated, calls = SetDominates(optimal, sorted_i[i][curr], delta/2, delta/2, error)
                num_calls += calls
                if not np.any(dominated):
                    optimal.append(sorted_i[i][curr])
                    changed = True
                    sorted_i[i].pop(curr)
```

```
            else:
                compl = True

    # check internally:
    new_optimal = []
    for i in range(len(optimal)):
        sublist = optimal[:i] + optimal[i + 1:]
        dominated, calls = SetDominates(sublist, optimal[i], delta/2, delta/2, error)
        num_calls += calls
        if not dominated:
            new_optimal.append(optimal[i])
    end = time.time()
    return new_optimal, end - start, num_calls

def msort2(x, dim, error):
    if len(x) < 2:
        return x, 0
    num_calls = 0
    result = []
    mid = int(len(x) / 2)
    y, calls1 = msort2(x[:mid], dim, error)
    z, calls2 = msort2(x[mid:], dim, error)
    num_calls += calls1
    num_calls += calls2
    while (len(y) > 0) and (len(z) > 0):
        comp, calls = BoostProb("oracle", z[0], y[0], dim, error, 1/(16*len(y)), 1/16)
        num_calls += calls
        if not comp:
            result.append(z[0])
            z.pop(0)
        else:
            result.append(y[0])
            y.pop(0)
    result += y
    result += z
    return result, num_calls
```

6.4 Skyline Computation Algorithms Sample Testing and Convergence

We measure the relative accuracy and complexity of each algorithm implemented by the average and maximum values for 0-1 outputted skyline set accuracy, runtime, and oracle queries. For **Skysampling** and **SkylineHighDim**, we further gauge the relative efficiency by measuring the average and median probability of incrementing p^* for each call of **MaxLex** within the algorithms, to best examine the real-world performance of **MaxLex** for various n, k, and d. Finally, we introduce the Hamming distance HD as a metric, which given $A, B \subseteq X$ computes the cardinality of their symmetric difference:

$$HD(A, B) = |(A \cup B) \setminus (A \cap B)| \tag{18}$$

Specifically, we measure the average and maximum Hamming distances of each algorithm as a more nuanced measure of outputted skyline set accuracy which can yield a sense of how close an output is to the actual solution, as found by **NoiselessSkylineComputation**.

Skyline Computation Algorithm Statistics								
Model	n	k	d	Mean Error	Mean Time	Mean Oracle Queries	Mean Probability of Incrementing p^*	Mean Hamming Distance
FSC	7	4	2	0.016667	0.00984	8860.25	NaN	0.016667
FSC	10	8	5	0.000333	0.05836	62800.48	NaN	0.0000333
FSC	100	39	5	1.000000	0.20841	2002561.7	NaN	27.973333
SS	7	4	2	0.156667	0.02873	1896.2	0.114525	0.178333
SS	10	8	5	0.400000	0.012918	12431.5	0.210021	0.406667
SS	100	39	5	1.000000	0.959234	101281.4	0.103598	8.240000
SHD	7	4	2	0.273810	0.015900	4352.2	0.568547	0.226190
SHD	10	8	5	0.260952	0.035679	15033.5	0.539881	0.306667
SHD	100	39	5	0.882222	2.713313	100415.7	0.554667	5.970000

Table 2: Statistics for various implementations of skyline computation algorithms (FSC = **FullSortControl**, SS = **Skysample**, SHD = **SkylineHighDim**) run 1000 times on datasets with the specified size n, skyline set cardinality k, and dimension d.

Table 2 depicts various statistics for the **FullSortControl**, **Skysample**, and **SkylineHighDim** algorithms (where the latter use the **ArgmaxRandom** formalism and update values $(1, 4, 3, -2)$) which aim to highlight the relative efficiencies and accuracies of each algorithm. **FullSortControl** has the highest observed accuracy over the smaller $n = 4, 7$ datasets, which suggests a high accuracy on a small scale; however, for $n = 100$, it performs the worst in terms of mean Hamming distance, suggesting that the errors from sorting incorrectly build up much more quickly than within the other two algorithms. Regardless, **FullSortControl** is extremely fast relative to the other algorithms, which speaks to the simplicity of the code design. However, the sorting algorithm nearly doubles the mean oracle queries compared to both **Skysample** and **SkylineHighDim**, which indicates that Examining this, we note that**FullSortControl** may therefore be of use for smaller computations which allow for oracles/votes to be tallied in one large batch; should there be a need for extended rounds of oracle voting, the other computation algorithms may be more useful. We note that while **Skysample** used many fewer queries than **SkylineHighDim** for the $n = 7$ dataset, the oracle queries became comparable for $n = 7$ and $n = 100$, suggesting that their bounds are asymptotically equivalent and supporting the findings of [1] and [2]. We further note that while **SkylineHighDim** has a much higher mean probability of incrementing p^*, there does not seem to be a particularly strong relationship between that probability and the mean error, as the error seems to scale with n and k while the probability of incrementing remains essentially constant. Further, we note that while **SkylineHighDim** is the only algorithm which did not completely diverge in error for the $n = 100$ dataset, and consistently has small mean Hamming distance, showing that its erroneous outputs are not far from the actual answers, its mean computation time was almost thrice that of **Skysample** and nearly ten times that of **FullSortControl**, signalling that after applying the changes which are necessary for minimizing the error of **MaxLex**, the **SkylineHighDim** algorithm may scale worse with n and k than predicted in [2]. We note that the divergence of **Skysample** and **SkylineHighDim** was lessened through tuning of the update parameters for each; however, we show the standardized version above to highlight the relative efficiencies.

7 Discussion

In this project, we provide a synthesis of iterative methods for skyline computation with noisy comparisons as published in [1] and [2]. We then show deficiencies within a proof involving **MaxLex** in [2] and, with that insight, implement algorithms with better convergence. Finally, we expand upon the provided algorithms with implementations, small-dataset stress testing and introduce Hamming Distance as a more nuanced version of accuracy. Further research is needed on exactly what bounds may be placed on **MaxLex** and how one may best tune the update parameter to best fit

a given dataset; in particular, the impact of the ratio between k and n, which generally represents the "crowdedness" of a dataset, would be interesting to examine.

References

[1] Benoit Groz and Tova Milo. Skyline queries with noisy comparisons. In *Proceedings of the 34th ACM Sigmod-Sigact-Sigai Symposium on Principles of Database Systems, 2015*, pages 185-198. ACM.

[2] Frederik Mallmann-Trenn, Claire Mathieu, and Victor Verdugo. Skyline Computation with Noisy Comparisons *arXiv preprint arXiv:1710.02058*, 2017

[3] Uriel Feige, Prabhakar Raghavan, David Peleg, and Eli Upfal. Computing With Noisy Information. In *SIAM Journal on Computing, 23(5), 1994*, pages 1001 - 1018.

[4] William Feller. An Introduction to Probability Theory and Its Applications. Vol. 1. *Wiley, 1968*, page 17.

[5] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition, 2009*, pages 248-255, IEEE.

8 Appendix

The actual implementation code can be found at `https://github.com/Saffr0n1/Iterative-Noisy-Skyline-Comparisons`.

8.1 Skysample

We present the pseudocode for **Skysample** as published in [1].

---

1 `Skysample(S, k'',`δ`):`
2 $S_0 = \emptyset$
3 `for i in (0,..,k*-1):`
4 `Compute z =` $\max_{<_{lex}}\{p|p \not\preceq S\}$ `with error tolerance` $\frac{\delta}{k''}$
5 `if` $z = \emptyset$ `return` S_i
6 `else` $S_{i+1} = S_i \bigcup z$
7 `return` S_k

We also define the oracle for $<_{lex}$ (p, p', δ) as

---

1 `Find` $l_1 = min\{i|p_i < p'_i\}$ `with error probability` $\frac{\delta}{2}$
2 `Find` $l_2 = min\{i|p_i > p'_i\}$ `with error probability` $\frac{\delta}{2}$
3 `if (`$l_2 =$ `Null or` $l_1 \preceq l_2$`) return true`
4 `else return false`

Here, the oracle and function are shown in separate blocks for visual clarity, although in the paper, they are subparts of the same overall **Skysample** function.

8.2 SkylineHighDim

We present the pesudocode for **SkylineHighDim** as published in [2].

---

1 `SkylineHighDim(k, X,`δ`):`
2 $S_0 = \emptyset$
3 `C = X`

```
3  for i in (0,..,k):
4  'Phase 1: finding p which is not dominated by current skyline'
5    while C not empty:
6      pick arbitrary p ∈ C
7      if not SetDominates(S_i, p, δ/4k, δ/4k|X|):
8        break
9      C = C \ {p}
10   'Phase 2: computing lexicographic maximum of the values which Pareto dominate p'
11   p'' = MaxLex(C, p, δ/2k)
12   S_i = S_{i-1} ⋃ {p''}
13 return S_k
```

8.3 MaxLex

We present the pseudocode for **MaxLex** as published in [2].

```
1 MaxLex(p, X,δ):
2   c(q) = log(1/δ) for all q ∈ X
3   q_1 = argmax_{q∈S} c(q)
4   q_2 = argmax_{q∈S\{q_1}} c(q)
5   while c(q_2) > −2:
6       if Lex(q_1, q_2):
7           x = q_1, y = q_2
8       else:
9           y = q_1, x = q_2
10      c(y) = c(y) − 1
11      if Dominates(x, p):
12          c = c + 1/2
13      else:
14          c = c − 1
15   return argmax_{q∈S} c(q)
```

8.4 Skyline Computation

Both skyline computation algorithms function on the same principle so only the algorithm from [1] is shown for space considerations.

```
1 Skyline_computation(S, δ)
2 i = 1; k_i = 4; compl = false
3 while compl= false:
4   R = SkySample(S, k_i, δ/2^i)
5   if |R| < k_i:
6     compl = true
7   else:
8     i = i+1
9     k_i = 2^{2^i}
10 return R
```