Efficient Database Search via Tensor Distribution Bucketing

Mihir Mongia^(✉), Benjamin Soudry, Arash Gholami Davoodi,
and Hosein Mohimani

Carnegie Mellon University School of Computer Science, Pittsburgh, USA
`mmongia@andrew.cmu.edu.com`, `{bsoudry,gholamid,hoseinm}@andrew.cmu.edu`

Abstract. In mass spectrometry-based proteomics, one needs to search billions of mass spectra against the human proteome with billions of amino acids, where many of the amino acids go through post-translational modifications. In order to account for novel modifications, we need to search all the spectra against all the peptides using a joint probabilistic model that can be learned from training data. Assuming M spectra and N possible peptides, currently the state of the art search methods have runtime of $O(MN)$. Here, we propose a novel bucketing method that sends pairs with high likelihood under the joint probabilistic model to the same bucket with higher probability than those pairs with low likelihood. We demonstrate that the runtime of this method grows sub-linearly with the data size, and our results show that our method is orders of magnitude faster than methods from the locality sensitive hashing literature.

Keywords: Database search · Sublinear · Joint distribution

1 Introduction

One of the fundamental challenges in mass-spectrometry based proteomics is to identify proteins present in a cell culture by searching their mass spectrometry fingerprint against all the peptides in a proteomic database/reference. When the number of mass spectra and the size of the reference proteome increase, this search becomes very slow, especially in cases where post-translational modifications are allowed.

Given a peptide sequence, the existing methods construct a binary-valued spectra from each peptide, where we have ones at the positions peaks are present and zeros otherwise. Then a probabilistic model is trained to learn the joint probability distribution $P(spec, pep)$ between the predicted spectra and the discretized mass spectra (Fig. 1) [10]. Given a spectra $spec$ and a set of peptides $Pep = \{pep_1, pep_2 \dots pep_N\}$, the goal is to find the peptide(s) $pep \in Pep$ that maximize $P(spec|pep)$. This task motivates the following problem.

Electronic supplementary material The online version of this chapter (https://doi.org/10.1007/978-3-030-47436-2_26) contains supplementary material, which is available to authorized users.

Fig. 1. Predicted spectra of three peptide FAG, KLT, and AMR are shown. Their mass spectra are shown in the bottom. The joint probability distribution between the predicted and mass spectra of the peptides, can be learned.

Database Search Problem In Probabilistic Settings (DPPS). Consider the following maximum likelihood problem that generalizes the problem of matching peptides to spectra. Let $\mathcal{A} = \{a_1, a_2, \dots, a_m\}$ and $\mathcal{B} = \{b_1, b_2, \dots, b_n\}$ be discrete alphabets where $m, n \in \mathbb{N}$. Let $\mathcal{P}$ be a joint distribution on the alphabets $\mathcal{A}$ and $\mathcal{B}$ such that $\sum_{i=1}^m \sum_{j=1}^n \mathcal{P}(a_i, b_j) = 1$. Let $S \in \mathbb{N}$ and $\mathbb{P}(X, Y) = \prod_{s=1}^S \mathcal{P}(x_s, y_s)$ where $X = (x_1, x_2, \dots, x_S)$, $Y = (y_1, y_2, \dots, y_S)$ and $x_s \in \mathcal{A}$, $y_s \in \mathcal{B}$ for $1 \leq s \leq S$. Given a data point $Y \in \mathcal{B}^S$ and a set of classes¹ $\{X^1, \dots, X^N\} \subseteq \mathcal{A}^S$ our goal is to accurately and efficiently predict the class X^t , $1 \leq t \leq N$, that generated Y .

Note in reference to mass-spectrometry based proteomics, the classes $\{X^1, \dots, X^N\}$ model the set of peptides $Pep = \{pep_1, pep_2 \dots pep_N\}$ and Y models a spectra $spec$. We address the DPPS problem by solving the following optimization problem:

$$\arg \max_{X \in \{X^1, X^2, X^3 \dots X^N\}} \mathbb{P}(Y|X) \quad (1)$$

A naive way to solve this optimization problem is to compute $\mathbb{P}(Y|X^t)$ for each $1 \leq t \leq N$, and find the maximum among them. The complexity of this approach grows linearly with N , and thus is prohibitively slow for practical applications as N grows large.

2 Related Work

As stated above, the issue with the naive way of solving **DPPS** is that brute force calculation of $\mathbb{P}(Y|X)$ for every $X \in \{X^1, \dots, X^N\}$ is a slow algorithm. Another domain where brute force calculation is necessary is nearest neighbor search. In nearest neighbor search, there is a data point $Y \in \mathbb{R}^S$ and a set of points $\{X^1, X^2, X^3 \dots X^N\} \in \mathbb{R}^S$. The goal is to quickly solve the following minimization problem:

¹ $X^1, X^2, X^3 \dots X^N$ are explained by the marginal distribution $\mathbb{P}_x$.

$$\arg \min_{X \in \{X^1, X^2, X^3, \dots, X^N\}} \|Y - X\|_2$$

where $\|Y - X\|_2$ stands for the Euclidean norm. This problem is equivalent to (1) in the special case where the probability distribution $\mathbb{P}$ is continuous and $P(y_s|x_s) \sim \mathcal{N}(x_s, \sigma)$, $1 \leq s \leq S$.

For high-dimensional data, the exact nearest neighbor search problem grows linearly with the number of data points [9]. Therefore, researchers consider the approximate nearest neighbor (ANN) search problem. In the ANN-search problem, the objective is to find $X \in \{X^1, X^2, \dots, X^N\}$ such that

$$\|Y - X\|_2 \leq c\|Y - X^*\|_2 \quad (2)$$

where

$$X^* = \arg \min_{X \in \{X^1, X^2, \dots, X^N\}} \|Y - X\|_2 \quad (3)$$

and $c > 1$ is referred to as the “approximation factor”. A common algorithm for solving this problem is *locality sensitive hashing* [2, 6, 7, 9] (Algorithm 1). This algorithm takes as input hashes h that satisfy the following constraints for some $R > 0$, and $0 < P_2 \leq P_1 \leq 1$:

• If $\|Y - X\|_2 \leq R$, then $h(X) = h(Y)$ with probability at least P_1 (4)

• If $\|Y - X\|_2 \geq cR$, then $h(X) \neq h(Y)$ with probability at most P_2 (5)

where hashes h satisfying (4) and (5) are called (R, cR, P_1, P_2) - sensitive hashes. As stated in Gionis et al. [7], “The key idea is to hash the points using several hash functions so as to ensure that, for each function, the probability of collision is much higher for objects which are close to each other than for those which are far apart. Then, one can determine near neighbors by hashing the query point and retrieving elements stored in buckets containing that point.”

Algorithm 1. Locality Sensitive Hashing

Inputs: A set of points $\mathcal{X} = \{X^1, X^2, \dots, X^N\} \subseteq \mathbb{R}^S$, a point $Y \in \mathbb{R}^S$, $h_{i,j}: \mathbb{R}^S \mapsto \{1, 2, \dots, z\}$ where $h_{i,j}$ are (R, cR, P_1, P_2) - sensitive, $1 \leq i \leq r$ and $1 \leq j \leq b$, $z \in \mathbb{N}$.

Outputs: A class $X \in \mathcal{X}$ that is the most similar to Y .

Preprocessing:

for $j = 1$ to b do

for $X \in \mathcal{X}$ do

$h_j(X) = (h_{1,j}(X), h_{2,j}(X), \dots, h_{r,j}(X))$

Store $\{h_j(x) | x \in \mathcal{X}\}$ in a hash table $\mathbf{H}_j$

Query:

for $j = 1$ to b do

Form $h_j(Y) = (h_{1,j}(Y), h_{2,j}(Y), \dots, h_{r,j}(Y))$.

Search $h_j(Y)$ in $\mathbf{H}_j$, i.e. find all $X \in \mathcal{X}$ that satisfy $h_j(Y) = h_j(X)$.

Among all the positives (all the $X \in \mathcal{X}$ that satisfy the line above), report the X that has the smallest Euclidean distance from Y .

The locality sensitive hashing algorithm takes a query point Y and aims to find the point that is the most similar to it in a database. The algorithm does this by first applying r hash functions to the query point in each band j , $1 \leq j \leq b$. Then in each band, the algorithm considers all points X in the database, that have been hashed to the same values as Y in all of the r hash functions.

Currently, LSH is limited to a number of distance metrics including manhattan distance (L_1), and euclidean distance (L_2). In order to use LSH for other similarity measures, one needs to transform them to L_1, L_2 metrics for which standard hashes are known. While it is possible to transform the **DPPS** problem to an approximate nearest neighbor search problem with standard metrics, in this paper we show such transformations result in algorithms with suboptimal complexity. In this paper, we design *buckets* for the **DPPS** problem that significantly outperform standard LSH algorithms.

We address the **DPPS** problem by defining pairs of relations (one for each alphabet) that are sensitive to the specific joint distribution that pairs of data points belong. Another distinctive feature of these hash relations, which we refer to as buckets, is that elements in the domain can be mapped to more than one element in the range. We refer to this framework as *distribution sensitive bucketing*. In distribution sensitive bucketing, the buckets $U^x: \mathcal{A}^S \mapsto 2^{\{1,2,\dots,Z\}}$, $U^y: \mathcal{B}^S \mapsto 2^{\{1,2,\dots,Z\}}$ satisfy the following constraints for some $0 \leq \beta \leq \alpha \leq 1$:

$$\bullet P(U^x(X) \cap U^y(Y) \neq \emptyset | (X, Y) \sim \mathbb{P}) = \alpha \quad (6)$$

$$\bullet P(U^x(X) \cap U^y(Y) \neq \emptyset | X \sim \mathbb{P}_X, Y \sim \mathbb{P}_Y, X \text{ and } Y \text{ are independent}) = \beta \quad (7)$$

$$\bullet \mathbb{E}|U^x(X)| = \delta_x \quad (8)$$

$$\bullet \mathbb{E}|U^y(X)| = \delta_y \quad (9)$$

$$\bullet |U^x(X) \cap U^y(Y)| \leq 1 \quad (10)$$

Here, $Z \in \mathbb{N}$ and for some set R , we define 2^R to be the set of all subsets of R . We refer to the buckets that satisfy (6), (7), (8), (9), and (10) as $(\mathcal{P}, \mathcal{Q}, \alpha, \beta, \delta_x, \delta_y)$ - sensitive buckets. For $(\mathcal{P}, \mathcal{Q}, \alpha, \beta, \delta_x, \delta_y)$ - sensitive buckets, it is the case that (i) the probability of a jointly generated pair being mapped to the same value is α , (ii) the probability of random pairs X, Y being mapped to the same value is β , and (iii) the complexity of assigning points to the buckets is proportional to δ_x and δ_y . Thus intuitively, we would like to maximize α while minimizing β, δ_x , and δ_y .

The rest of the document will proceed as follows. In Sect. 3, we assume an oracle has given us a family of $(\mathcal{P}, \mathcal{Q}, \alpha, \beta, \delta_x, \delta_y)$ - sensitive buckets, and we design an algorithm to solve (1) based on this family. In Sect. 4, we provide a way to construct these buckets. In Sect. 5 we derive the overall complexity of the algorithm presented in Sect. 3 and in Sect. 6, we propose an algorithm for constructing optimal buckets. Finally, in Sect. 7, we detail our experiments on simulated and real mass spectra.

3 Distribution Sensitive Bucketing Algorithm

In this section we introduce an algorithm for solving (1) when an oracle has given a family of distribution sensitive buckets and we refer to this algorithm as Distribution Sensitive Bucketing.

In contrast to the locality sensitive hashing algorithm that attempts to find the pairs of data points that are very similar to each other, the goal of distribution sensitive bucketing is to find pairs of data points that are jointly generated from a known joint probability distribution. Algorithm 2 describes a procedure to solve (1) using a family of distribution sensitive buckets. Here we use r rows and b bands, and in each band we check whether the query Y has a collision with a data point X in each of the r rows.

Algorithm 2. Distribution Sensitive Bucketing

Inputs: A set of points $\mathcal{X} = \{X_1, X_2, \dots, X_n\} \subseteq \mathcal{A}^S$, a point $Y \in \mathcal{B}^S$, family of $(\mathcal{P}, \mathcal{Q}, ha, \beta, \delta_x, \delta_y)$ -sensitive buckets $U_{i,j}^x: \mathcal{A}^S \mapsto 2^{\{1,2,\dots,Z\}}$, $U_{i,j}^y: \mathcal{B}^S \mapsto 2^{\{1,2,\dots,Z\}}$, $b, r \in \mathbb{N}$

Outputs: A point $X \in \mathcal{X}$ that maximizes $\mathbb{P}(Y|X)$.

Preprocessing:

for $j = 1$ to b do

for $x \in \mathcal{X}$ do

Form $\mathbf{U}_j^x(X) = (U_{1,j}^x(X) \times U_{2,j}^x(X) \times \dots \times U_{r,j}^x(X))$

Store $\bigcup_{X \in \mathcal{X}} \mathbf{U}_j^x(x = X)$ in a hash table $\mathbf{H}_j$

Query:

for $j = 1$ to b do

Form $\mathbf{U}_j^y(Y) = (U_{1,j}^y(Y) \times U_{2,j}^y(Y) \times \dots \times U_{r,j}^y(Y))$

Search each member of $\mathbf{U}_j^y(Y)$ in $\mathbf{H}_j$ to find all $X \in \mathcal{X}$ that satisfy $\mathbf{U}_j^y(Y) \cap \mathbf{U}_j^x(X) \neq \emptyset$.

Among all the positives (all the $X \in \mathcal{X}$ that satisfy the line above, report the X that maximizes $\mathbb{P}(Y|X)$.

4 Constructing Distribution Sensitive Buckets

In the previous section, we assumed an oracle has given us a set of distribution sensitive buckets, and we designed an algorithm to solve (1) using this family of buckets. In this section, we present an approach for constructing distribution sensitive buckets.

Let A and B be arbitrary binary tensors of dimensions $m^k \times Z$ and $n^k \times Z$, respectively. To construct distribution sensitive buckets, for each pair of buckets we choose positions $1 \leq S_1 \leq S_2 \leq \dots \leq S_k \leq S$ randomly, and we define buckets $U_{S_1, S_2, \dots, S_k}^x$ and $U_{S_1, S_2, \dots, S_k}^y$ as follows:

$$U_{S_1, S_2, \dots, S_k}^x(X) = \left\{ z \in \{1, \dots, Z\} \mid A_{X_{S_1}, X_{S_2}, \dots, X_{S_k}, z} = 1 \right\} \quad (11)$$

$$U_{S_1, S_2, \dots, S_k}^y(Y) = \left\{ z \in \{1, \dots, Z\} \mid B_{Y_{S_1}, Y_{S_2}, \dots, Y_{S_k}, z} = 1 \right\} \quad (12)$$

As there is a straightforward way to convert a tensor to a matrix, in the rest of this paper we treat A and B as m^k by Z and n^k by Z binary matrices. Furthermore, for any matrix M and $u, v \in \mathbb{N}$ we use the notation $M[u, v]$ to refer to the entry belonging to the u th row and v th column of M . In the rest of this section, we derive α , β , δ_x , and δ_y for the proposed family of buckets.

Remark 1. Denote two buckets U^x and U^y and their corresponding matrices as A and B , respectively. These buckets satisfy (10) if for any row i of A and row j of B there is at most one column c where both $A[i, c] = 1$ and $B[j, c] = 1$. A and B satisfying this constraint are called **non-intersecting**.

Theorem 1. *Given a pair of bucket U^x and U^y and a corresponding pair of matrices A and B that satisfy the non-intersecting constraint, it can be shown that U^x and U^y are $(\mathcal{P}, \mathcal{Q}, \alpha, \beta, \delta_x, \delta_y)$ -sensitive where*

$$\alpha = \text{Trace}(A^T \mathcal{P}^k B) \quad (13)$$

$$\beta = \text{Trace}(A^T \mathcal{Q}^k B) \quad (14)$$

$$\delta_x = P_x^k A \mathbb{I}_z \quad (15)$$

$$\delta_y = P_y^k B \mathbb{I}_z \quad (16)$$

and $\mathbb{I}_z$ denotes a vector ones of dimension z

Proof. For a proof of Theorem 1, see Supplementary Section 1.

5 Complexity Analysis

In Sect. 3, we provided Algorithm 2 to solve (1) given a family of Distribution Sensitive Buckets, and in the previous section we constructed a family of Distribution Sensitive Buckets based on $m^k \times Z$ and $n^k \times Z$ matrices, A and B . In this section, we analyze the expected complexity of Query portion of Algorithm 2 given the generative process defined in in DPPS. We first analyze the complexity given a specific instance of Y and $\mathcal{X} = \{X^1, X^2, X^3 \dots X^N\}$, and then derive the expected complexity. In Algorithm 2, the computational work for each query can be broken into (i) searching members of $\mathbf{U}_j(Y)$ in a hash table containing $\bigcup_{X \in \mathcal{X}} \mathbf{U}_j(\mathbf{X})$ in order to find positives and (ii) computing $\mathbb{P}(Y|X)$ for all the positives.

Since searching a hash table is $O(1)$ complexity, the computational work of (i) is equivalent to $|\mathbf{U}_j(Y)|$. Computing $\mathbf{U}_j(Y)$ requires forming the cartesian product $U_{1,j}(Y) \times U_{2,j}(Y) \times \dots \times U_{r,j}(Y)$. Thus, the total size of $\mathbf{U}_j^y$ over b bands can be upper bounded by

$$\sum_{j=1}^b \prod_{i=1}^r |U_{i,j}(Y)| \quad (17)$$

The computational work of (ii) is equal to the number of positives. The number of positives can be calculated in the following way:

$$\sum_{j=1}^b \sum_{x \in \mathcal{X}} \prod_{i=1}^r |U_{i,j}(X) \cap U_{i,j}(Y)| \quad (18)$$

The expectation of the sum of (17) and (18) given the generative process in DPPS is then given by

$$\mathbb{E} \left[\sum_{j=1}^b \prod_{i=1}^r |U_{i,j}(Y)| + \sum_{j=1}^b \sum_{x \in \mathcal{X}} \prod_{i=1}^r |U_{i,j}(X) \cap U_{i,j}(Y)| \mid X \sim \mathbb{P}_x, Y \sim \mathbb{P}_y \right] = bN\beta^r + b\delta_y^r \quad (19)$$

Note that here we assumed that almost all of the pairs are independently generated. This assumption is due to the fact that only one $X \in \mathcal{X}$ is responsible for generating Y . Now the question is how do we select b ? The probability of a jointly generated X, Y pair being called a positive (i.e. $\mathbf{U}_j^x(X) = \mathbf{U}_j^y(Y)$ for at least one j), which we refer to as the *True Positive Rate*, can be calculated in the following way:

$$1 - \prod_{j=1}^b \left(1 - \prod_{i=1}^{i=r} P(U_{i,j}^x \cap U_{i,j}^y(Y) \neq \emptyset) \right) = 1 - (1 - \alpha^r)^b \geq 1 - (e^{-\alpha^r})^b \quad (20)$$

We usually want to maintain a true positive rate of nearly 1, e.g. *True Positive Rate* $\geq 1 - \epsilon$ where ϵ is a small number. This can be realized by setting $(e^{-\alpha^r})^b \leq \epsilon$, i.e.

$$b \geq \frac{-\ln \epsilon}{\alpha^r}, \quad (21)$$

Therefore, the overall expected complexity given the generative process in DPPS, can be upper bounded by the following expression² and $\mathcal{X}$ is already preprocessed:

$$\frac{-\ln \epsilon}{\alpha^r} N\beta^r + \frac{-\ln \epsilon}{\alpha^r} \delta_y^r \quad (22)$$

6 Designing Distribution Sensitive Buckets

In Sects. 4 and 5, we described how to construct a family of distributive sensitive buckets using matrices A and B, and we derived the complexity of Algorithm 2 based on these matrices. In this section we present an approach to find the optimal matrices based on integer linear programming.

² Note that δ_x is not present in this expression. This is because we are considering the situation we search Y against $\mathcal{X} = \{X^1, \dots, X^N\}$.

6.1 Integer Linear Programming Method

Algorithm 3 presents a integer linear programming approach for finding matrices A and B that optimize the complexity (22) using (13), (14), (15), and (16). Our approach is based on the assumption that matrix A is identity. The reason behind this assumption is that any matrix B is non-intersecting with the identity matrix.

Algorithm 3. Designing distribution sensitive buckets through integer linear programming

Inputs: $m \times n$ matrix $\mathcal{P}$, $\mathcal{Q}$, k , r , $\Delta_x, \Delta_y, \bigcirc$

$\mathcal{P}_x \leftarrow \mathcal{P}\mathbb{I}_m, \mathcal{P}_y \leftarrow \mathbb{I}_n\mathcal{P}, \mathcal{Q} \leftarrow \mathcal{P}_x\mathcal{P}_y$

Outputs: $A \in \{0, 1\}^{m \times m}$, $B \in \{0, 1\}^{n \times m}$

$A = I_m$ (identity matrix with size m)

for $\delta_x \in \Delta_x$ **do**

for $\delta_y \in \Delta_y$ **do**

for $\beta \in \bigcirc$ **do**

 Maximize $\alpha = \text{Trace}(A^T \mathcal{P}^k B)$ via a integer linear program such that:

- every entry of B is in $\{0, 1\}$.
- $\text{Trace}(A^T \mathcal{Q}^k B) = \beta$
- $P_y^k B \mathbb{1} = \delta_y$
- $P_x^k B \mathbb{1} = \delta_x$

Report B minimizing (22)

We often need to design buckets for larger values of k in order to design more efficient algorithms for solving (1). However, the size of $\mathcal{P}^k$ grows exponentially with k and thus Algorithm 3 will not run efficiently for $k > 10$. Thus, we use Algorithm 4 to filter $\mathcal{P}^k$ to a smaller matrix $\mathcal{P}_\epsilon^k$, which only keeps the rows and columns in $\mathcal{P}^k$ with sum above ϵ , and then pass this matrix, which we denote as $\mathcal{P}_\epsilon^k$, as an input to Algorithm 3.

Algorithm 4. Filtering of kronecker product matrix

Inputs: $\mathcal{P}$, $k \in \mathbb{N}$, $\epsilon \leq 1$

Outputs: $\mathcal{P}_\epsilon^k$

$\mathcal{P}_\epsilon^0 \leftarrow \mathbb{1}$

for $t = 1$ **to** k **do**

$\mathcal{P}_\epsilon^t \leftarrow \mathcal{P}_\epsilon^{t-1} \otimes \mathcal{P}$

 Remove any row/column in $\mathcal{P}_\epsilon^t$ with sum below ϵ

Report $\mathcal{P}_\epsilon^k$.

7 Experiments

In this section we verify the advantage of our Distribution Sensitive Bucketing approach with several experiments. In the first experiment, we compare

the performance of Distribution Sensitive Bucketing to several commonly used methods in the Locality Sensitive Hashing literature on a range of theoretical distributions $\mathcal{P}$. Although these methods are not directly applicable to our problem, we can transform the **DPSS** problem into problems where these methods work. In the second experiment, we apply the Distribution Sensitive Bucketing algorithm along with the same methods from the Locality Sensitive Hashing literature to the problem of peptide identification from mass spectrometry signals. In this problem, given millions of peptides $\mathcal{X} = \{X^1, X^2, X^3 \dots X^N\}$ and a discretized mass spectrum Y , our goal is to find a peptide $X \in \mathcal{X}$ that maximizes $\mathbb{P}(Y|X) = \prod_{s=1}^{s=S} \mathcal{P}(y_s|x_s)$ where $\mathcal{P}$ can be learned from a training data set of known peptide-spectra pairs. We use the probabilistic model introduced in Kim et al. [10], which is trained to score a mass spectra against a peptide sequence accounting for neutral losses and the intensity of peaks (see Supplementary Figure 1).

7.1 Experiment 1. Theoretical Complexity

In this experiment we compared the complexity of 3 algorithms - LSH-Hamming, Inner Product Hash [13], and Distribution Sensitive Bucketing - on a range of probability distributions. LSH-Hamming and Inner Product Search can not be directly applied to the **DPSS** problem as LSH-Hamming and Inner Product Hash only work when $\{X^1, \dots, X^N\}$ and Y both belong to the same alphabet. Nevertheless, through the following transformations, the **DPSS** problem can be transformed to a nearest neighbor search problem with hamming distance and the maximum inner product search problem. To transform the **DPSS** problem to nearest neighbor search with hamming distance, map each element in the alphabets $\mathcal{A} = \{a_1, \dots, a_m\}$ and $\mathcal{B} = \{b_1, \dots, b_n\}$ to either 0 or 1. As a result, for each $X \in \{X^1, \dots, X^N\}$, $X \in \{0, 1\}^S$ and $Y \in \{0, 1\}^S$. To transform **DPSS** to the maximum inner product search problem, first change the objective function to $\log(\mathbb{P}(Y|X)) = \sum_{s=1}^{s=S} \log(\mathcal{P}(y_s|x_s))$. Observe that $\log(\mathcal{P}(y_s|x_s))$ can be expressed as the dot product of a one hot vector of size n (the size of the alphabet $\mathcal{B}$) with a vector $\log(\mathcal{P}(y|x_s)) \in \mathbb{R}^n$. Now we can concatenate all the vectors (one for each $1 \leq s \leq S$) into signals v_Y and w_X of length Sn . The dot product of these two vectors will be $\log(\mathbb{P}(Y|X))$. Thus one can then apply maximum inner product search (MIPS) to the set $\mathcal{X}' = \{w_X | X \in \{X^1, \dots, X^N\}\}$ and query vector v_Y in order to solve the **DPSS** problem.

We benchmark the algorithms using probability distribution $\mathcal{P}(t) = \mathcal{P}_1 t + \mathcal{P}_2(1 - t)$ for different values of $0 \leq t \leq 1$ where $\mathcal{P}_1 = \begin{bmatrix} .25 & .25 \\ .25 & .25 \end{bmatrix}$ and $\mathcal{P}_2 = \begin{bmatrix} .95 & .01 \\ .03 & .01 \end{bmatrix}$. Since LSH-Hamming's performance depends on the particular mappings of the original alphabets to $\{0, 1\}$, we do an exhaustive search over all mappings and use the best mapping for each $\mathcal{P}(t)$.

In Fig. 2, we plot the asymptotic complexity of each of the three algorithms. The asymptotic complexity can be expressed as $O(N^\lambda)$, for some λ . The value of λ is plotted versus t in Fig. 2. As one can see, Distribution Sensitive Bucketing has a lower asymptotic complexity than LSH-Hamming for $0 \leq t \leq .5$ and for $t \geq .5$ Distribution Sensitive Bucketing and LSH-Hamming have the same asymptotic complexity. Inner Product Hash is always worse than Distribution Sensitive Bucketing by a large margin.

7.2 Experiment 2

In this experiment we evaluated the performance of Distribution Sensitive Bucketing on simulated spectra and peptides. We simulated the mass spectra and peptides using the probabilistic model from Kim et al. [10]. For each of the algorithms, we choose the parameters so that they theoretically achieve a 95% true positive rate. In Figure 2 of the Supplementary, we verify experimentally that we indeed achieve a 95% True Positive Rate. Figure 3 shows the number of positive calls for our method in comparison to the brute force method, LSH-Hamming, and Inner Product Hash. Here, we changed the number of peptides from 100 to 100,000 and computed the average number of positives per spectrum averaged over 5,000 spectra. Figure 4 shows the runtime of brute force search, LSH-Hamming, and Inner Product Hash versus Distribution Sensitive Bucketing. Distribution Sensitive Bucketing is 20X faster than brute force while LSH - Hamming is only 2X faster than brute force. Inner Product Hash is always as slow as brute force search.

7.3 Experiment 3 - Mass Spectrometry Database Search in Proteomics

We applied Distribution Sensitive Bucketing, LSH-Hamming, Inner Product Hash, and brute force search to the problem of mass spectrometry database search in proteomics. Here we search a dataset of 93,587 spectra against the human proteome sequence. We tune the parameters of Distribution Sensitive Bucketing, LSH-Hamming, and Inner Product Hash on a smaller test data set to get a 95% True Positive rate and then apply the algorithms on the larger data set. For distribution sensitive bucketing, this resulted in a 91% true positive rate and 50X decrease in the number of positives in comparison to brute force search. Distribution Sensitive Bucketing also led to 30X reduction in time compared to brute force search. LSH-Hamming resulted in a 2X reduction in positives and a 2X reduction in computation time while achieving a True Positive Rate of 93%. Inner Product Hash did not improve on brute force search.

Fig. 2. For each algorithm we plot the theoretical asymptotic complexity of search for different values of t , which corresponds to different mixtures of distributions $\mathcal{P}_1$ and $\mathcal{P}_2$. By theoretical asymptotic complexity we mean the value λ for which the complexity of the algorithm is $O(N^\lambda)$

Fig. 3. The empirical number of positive calls by brute force search, Distribution Sensitive Bucketing, LSH-Hamming, and Inner Product Hash. Note in this case that Brute Force and Inner Product Hash perform the same. Distribution Sensitive Bucketing has 20X less positive calls than brute force search and 10X less positive calls than LSH-Hamming.

Fig. 4. The run time of Distribution Sensitive Bucketing (Algorithm 2), in comparison to brute force, LSH-Hamming, and Inner Product Hash. Distribution Sensitive Bucketing has a 20X reduction in time compared brute force. LSH-Hamming has a reduction of 2X compared to brute force.

8 Conclusion

In this paper we introduce a problem from computational biology which requires computing a joint likelihood of all pairs of data points coming from two separate large data sets. In order to speed up this brute force procedure, we develop a novel bucketing method. We show theoretically and experimentally our method is superior to methods from the locality sensitive literature.

Acknowledgements. This work was supported by a research fellowship from the Alfred P. Sloan Foundation and a National Institutes of Health New Innovator Award (DP2GM137413).

References

1. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In: 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006), pp. 459–468. IEEE (2006)
2. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM* **51**(1), 117 (2008)
3. Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.Y.: An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM (JACM)* **45**(6), 891–923 (1998)

4. Charikar, M.S.: Similarity estimation techniques from rounding algorithms. In: Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing, pp. 380–388. ACM (2002)
5. Christiani, T., Pagh, R.: Set similarity search beyond MinHash. In: Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, pp. 1094–1107. ACM (2017)
6. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: Proceedings of the Twentieth Annual Symposium on Computational Geometry, pp. 253–262. ACM (2004)
7. Gionis, A., Indyk, P., Motwani, R., et al.: Similarity search in high dimensions via hashing. *VLDB* **99**, 518–529 (1999)
8. Indyk, P.: Approximate algorithms for high-dimensional geometric problems. In: Invited Talk at DIMACS Workshop on Computational Geometry (2001)
9. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the 30th Annual ACM Symposium on Theory of Computing, pp. 604–613. ACM (1998)
10. Kim, S., Pevzner, P.A.: MS-GF+ makes progress towards a universal database search tool for proteomics. *Nat. Commun.* **5**, 5277 (2014)
11. Kong, A.T., Leprevost, F.V., Avtonomov, D.M., Mellacheruvu, D., Nesvizhskii, A.I.: MSFragger: ultrafast and comprehensive peptide identification in mass spectrometry-based proteomics. *Nat. Methods* **14**(5), 513 (2017)
12. Leskovec, J., Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets. Cambridge University Press, Cambridge (2014)
13. Neyshabur, B., Srebro, N.: On symmetric and asymmetric LSHS for inner product search. arXiv preprint [arXiv:1410.5518](https://arxiv.org/abs/1410.5518) (2014)
14. Shrivastava, A., Li, P.: In defense of MinHash over SimHash. In: Artificial Intelligence and Statistics, pp. 886–894 (2014)