Efficient Reductions for k-Nearest Neighbor Search

Joint work with Tobias Christiani and Mikkel Thorup

Rasmus Pagh
BARC and IT University of Copenhagen

Workshop on Sublinear Algorithms and Nearest Neighbor Search

Simons Institute for Theory of Computing
November 29, 2018
Nearest neighbor search

\[ P = \{x_1, \ldots, x_n\} \]
Nearest neighbor search

\[ P = \{ x_1, \ldots, x_n \} \]
Nearest neighbor search

\[ P = \{ x_1, \ldots, x_n \} \]
Nearest neighbor search

\[ P = \{ x_1, \ldots, x_n \} \]
\(k\)-nearest neighbor search

\[Q = \{x_1, \ldots, x_n\}\]
Some application areas
Some application areas

- Association rule mining
- Automation
- Bio-chemistry (finding motifs)
- Bio-informatics (homology search)
- Clustering
- Computer vision and pattern recognition
- Databases
- Data cleaning
- Data stream computation
- Data privacy
- First story detection (with application to Twitter)

- Identifying trends in time series
- Linear algebra
- Motion planning for robots
- Near-duplicate detection
- News personalization (collaborative filtering)
- Privacy preserving data mining
- Search engines for 3D models
- Sensor networks
- …
Hardness of NN search

• [Williams ’04], [Alman & Williams ’15]:
  NN search on $P \subseteq \{0,1\}^d$ in time $n^{0.99} 2^{o(d)}$ with preprocessing time poly$(n) 2^{o(d)}$ \implies
  $k$-SAT w. $n$ variables can be solved in time $c^n$, $c < 2$
Hardness of NN search

• [Williams ’04], [Alman & Williams ’15]:
  NN search on $P \subseteq \{0,1\}^d$ in time $n^{0.99} 2^{o(d)}$ with preprocessing time $\text{poly}(n) 2^{o(d)} \implies$
  $k$-SAT w. $n$ variables can be solved in time $c^n$, $c < 2$

Under strong exponential time hypothesis, this is not possible!
Hardness of NN search

• [Williams ’04], [Alman & Williams ’15]:
  NN search on $P \subseteq \{0,1\}^d$ in time $n^{0.99} 2^{o(d)}$ with preprocessing time $\text{poly}(n) 2^{o(d)} \Rightarrow$
  $k$-SAT w. $n$ variables can be solved in time $c^n$, $c < 2$

Under strong exponential time hypothesis, this is not possible!

In practice: “Curse of dimensionality” makes NN search slow in high dimension.
Approximate nearest neighbor

Approximate nearest neighbors: towards removing the curse of dimensionality
P Indyk, R Motwani
Proceedings of the thirtieth annual ACM symposium on Theory of computing
Approximate nearest neighbor

Approximate nearest neighbors: towards removing the curse of dimensionality
P Indyk, R Motwani
Proceedings of the thirtieth annual ACM symposium on Theory of computing …
Approximate nearest neighbor

Approximate nearest neighbors: towards removing the curse of dimensionality
P Indyk, R Motwani
Proceedings of the thirtieth annual ACM symposium on Theory of computing …
Approximate nearest neighbor

Approximate nearest neighbors: towards removing the curse of dimensionality
P Indyk, R Motwani
Proceedings of the thirtieth annual ACM symposium on Theory of computing …
Approximate nearest neighbor

Approximate nearest neighbors: towards removing the curse of dimensionality
P Indyk, R Motwani
Proceedings of the thirtieth annual ACM symposium on Theory of computing …

Time $n^{\rho(c)}$ and space $n^{1+\rho(c)}$, $\rho(c) < 1$ for $c > 1$

c-approximate NN
(return any one)
Approximate NN in practice

Recall-Queries per second (1/s) tradeoff - up and to the right is better

[Graph showing recall vs. queries per second for different libraries and algorithms]
Approximate NN in practice

Recall = fraction of nearest neighbors found for set of queries
Approximate NN in practice

Recall = fraction of nearest neighbors found for set of queries

Achieving a given recall is an empirical task
NN recall guarantee?

Black-box reduction: Choose $c$ small enough to distinguish nearest neighbor from other points.
NN recall guarantee?

Black-box reduction: Choose $c$ small enough to distinguish nearest neighbor from other points.
NN recall guarantee?

Black-box reduction: Choose $c$ small enough to distinguish nearest neighbor from other points.

$c$-approximate NN = nearest neighbor
Black-box reduction does not distinguish easy and hard cases.

NN recall guarantee?
LSH with multi-probing

• Building block: Linear space data structure (a hash table) that allows us to retrieve $x_i$ with probability $p_i$.
  • Let $p_1 =$ retrieval probability of nearest neighbor
LSH with multi-probing

• Building block: Linear space data structure (a hash table) that allows us to retrieve $x_i$ with probability $p_i$.

  • Let $p_1 = \text{retrieval probability of nearest neighbor}$
  
  • Expected cost $O \left( K + T + \sum_i p_i \right)$

[Panigrahy '06, Lv et al. '07]  

Parameters $K, T$
LSH with multi-probing

- Building block: Linear space data structure (a hash table) that allows us to retrieve $x_i$ with probability $p_i$.
  - Let $p_1 =$ retrieval probability of nearest neighbor
  - Expected cost $O\left(K + T + \sum_i p_i\right)$
  - Repeat independently $L$ times; fails to find NN with probability $(1 - p_1)^L$

[Panigrahy ’06, Lv et al. ‘07]
LSH with multi-probing

- Building block: Linear space data structure (a hash table) that allows us to retrieve $x_i$ with probability $p_i$.
  - Let $p_1 = \text{retrieval probability of nearest neighbor}$
  - Expected cost $O \left( K + T + \sum_i p_i \right)$
  - Repeat independently $L$ times; fails to find NN with probability $(1 - p_1)^L$

[Panigrahy ’06, Lv et al. ‘07]

Need $L \approx \ln(1/\delta)/p_1$ for expected recall $1-\delta$
Choosing parameters

All in all, we have three parameters:

- K, the number of hash functions (space partitions) per hash table.
- L, the number of hash tables (each of which has K hash functions).
- T, the number of probes (total number of buckets probed across all hash tables).

Usually, it is a good idea to choose L first based on the available memory. Then, we have a trade-off between K and T: the larger K is, the more probes we need to achieve a given probability of success, and vice versa. The best way to choose K and T is usually the following parameter search: Try increasing values of K, and for each value of K, find the right number of probes T so that we get the desired accuracy on a set of sample queries. Varying the parameter T does not require rebuilding the hash table (as opposed to K and L). Moreover, we can search over T using a binary search. Usually, this means that we can find the optimal parameter setting fairly quickly.
Choosing parameters

All in all, we have three parameters:

- \( K \), the number of hash functions (space partitions) per hash table.
- \( L \), the number of hash tables (each of which has \( K \) hash functions).
- \( T \), the number of probes (total number of buckets probed across all hash tables).

Usually, it is a good idea to choose \( L \) first based on the available memory. Then, we have a trade-off between \( K \) and \( T \): the larger \( K \) is, the more probes we need to achieve a given probability of success, and vice versa. The best way to choose \( K \) and \( T \) is usually the following parameter search: Try increasing values of \( K \), and for each value of \( K \), find the right number of probes \( T \) so that we get the desired accuracy on a set of sample queries. Varying the parameter \( T \) does not require rebuilding the hash table (as opposed to \( K \) and \( L \)). Moreover, we can search over \( T \) using a binary search. Usually, this means that we can find the optimal parameter setting fairly quickly.
Choosing parameters

All in all, we have three parameters:

- $K$, the number of hash functions (space partitions) per hash table.
- $L$, the number of hash tables (each of which has $K$ hash functions).
- $T$, the number of probes (total number of buckets probed across all hash tables).

Usually, it is a good idea to choose $L$ first based on the available memory. Then, we have a trade-off between $K$ and $T$: the larger $K$ is, the more probes we need to achieve a given probability of success, and vice versa. The best way to choose $K$ and $T$ is usually the following parameter search: Try increasing values of $K$, and for each value of $K$, find the right number of probes $T$ so that we get the desired accuracy on a set of sample queries. Varying the parameter $T$ does not require rebuilding the hash table (compared to $K$ and $L$). Moreover, we can search over $T$ using a binary search. Usually, this means that we can find the optimal parameter setting fairly quickly.
Choosing parameters

All in all, we have three parameters:

- K, the number of hash functions (space partitions) per hash table.
- L, the number of hash tables (each of which has K hash functions).
- T, the number of probes (total number of buckets probed across all hash tables).

Usually, it is a good idea to choose L first based on the available memory. Then, we have a trade-off between K and T: the larger K is, the more probes we need to achieve a given probability of success, and vice versa. To find the optimal parameter setting, we can try the following parameter search:

Try increasing values of K, then L, and finally T so that we get the desired accuracy on a per-query basis without rebuilding the hash tables. In most cases, we do not require T to be larger than L. Usually, this means that we can find the optimal parameter setting fairly quickly.

Achieving a given recall using LSH methods is an empirical task.
Adaptive stopping

Idea:

- Suppose that after searching \( t \) hash tables the nearest neighbor retrieved so far is \( x^* \).
- Let \( p^* \) denote the probability that \( x^* \) is retrieved in a hash table.
- If \( t > \frac{\ln(1/\delta)}{p^*} \) then stop and return \( x^* \).
Adaptive stopping

Idea:

• Suppose that after searching $t$ hash tables the nearest neighbor retrieved so far is $x^*$. 

• Let $p^*$ denote the probability that $x^*$ is retrieved in a hash table. 

• If $t > \frac{\ln(1/\delta)}{p^*}$ then stop and return $x^*$. 

Yields expected recall $1-\delta$ if the nearest neighbor has the largest retrieval probability
Adaptive stopping

Idea:

- Suppose that after searching $t$ hash tables the nearest neighbor retrieved so far is $x^*$. 
- Let $p^*$ denote the probability that $x^*$ is retrieved in a hash table. 
- If $t > \ln(1/\delta)/p^*$ then stop and return $x^*$. 

Yields expected recall $1-\delta$ if the nearest neighbor has the largest retrieval probability

Issue: Need way of computing $p^*$. 

[Dong et al. ’08]
Abstract view

$q$
Abstract view

$q$
Abstract view

\[ \beta \bullet \]

[\bullet \eta]
Abstract view

$\beta$

$q$
Abstract view
Abstract view
Abstract view

\[ \beta, q \]
Abstract view
Abstract view
How can we know if $\beta$ is likely to be the nearest neighbor?
How can we know if $\beta$ is likely to be the nearest neighbor?
Confirmation sampling

• Notation: Repetition $i$ produces one nearest neighbor candidate $y_i$, independently for each $i$.
• Let $\beta_i$ be the nearest neighbor of $q$ in $Y_i = \{y_1, \ldots, y_i\}$. 
Confirmation sampling

• Notation: Repetition $i$ produces one nearest neighbor candidate $y_i$, independently for each $i$.

• Let $\beta_i$ be the nearest neighbor of $q$ in $Y_i = \{y_1, \ldots, y_i\}$.

• Stop after repetition $i$ if $\beta_i$ appears $t+1$ times in $Y_i$. 
Confirmation sampling

- Notation: Repetition $i$ produces one nearest neighbor candidate $y_i$, independently for each $i$.
- Let $\beta_i$ be the nearest neighbor of $q$ in $Y_i = \{y_1, \ldots, y_i\}$.
- Stop after repetition $i$ if $\beta_i$ appears $t+1$ times in $Y_i$.
Confirmation sampling

• Notation: Repetition $i$ produces one nearest neighbor candidate $y_i$, independently for each $i$.

• Let $\beta_i$ be the nearest neighbor of $q$ in $Y_i = \{y_1, \ldots, y_i\}$.

• Stop after repetition $i$ if $\beta_i$ appears $t+1$ times in $Y_i$.

Claim: If nearest neighbor is most likely to be a candidate, the probability of returning a different point is at most $2^{-t}$.
Confirmation sampling

\[ t = 1 \]
Confirmation sampling

\[ t = 1 \]

\( q \)
Confirmation sampling

\[ t = 1 \]

\[ y \]

\[ q \]
Confirmation sampling

\[ t = 1 \]

\[ \beta \cdot y \]

\[ q \]
Confirmation sampling

t = 1

\beta \cdot y

q
Confirmation sampling

\[ t = 1 \]

\[ \beta \quad q \quad y \]
Confirmation sampling

\[ t = 1 \]

\[ \beta \cdot y \]

\[ q \]
Confirmation sampling

\[ t = 1 \]

\[ \beta \cdot y_q \]
Confirmation sampling

$t = 1$

$y$

$q$

$\beta$
Confirmation sampling

\[ t = 1 \]

\[ \beta \cdot y \cdot q \]

Confirmation:
Return $\beta$
Why confirmation sampling works

• Suppose $x_1$ is the nearest neighbor.

• For $i > 1$, if $y_i = \beta_{i-1}$ but $y_i \neq x_1$ we say step $i$ is a false confirmation.
Why confirmation sampling works

• Suppose $x_1$ is the nearest neighbor.

• For $i > 1$, if $y_i = \beta_{i-1}$ but $y_i \neq x_1$ we say step $i$ is a false confirmation.

• Consider the first $t$ steps where we either sample $x_1$ or produce a false confirmation:
  
  - If sampling $x_1$ is more likely than sampling $\beta_{i-1} \neq x_1$, the probability of false confirmation is at most $1/2$ in each step.

  - Probability of $t$ false confirmations is at most $2^{-t}$. 

Abstract confirmation sampling

Algorithm 1: CONFIRMATION_SAMPLING(Q, t, \( \prec \))

1. \( \beta \leftarrow \infty \), count \( \leftarrow 0 \)
2. while count \( < t \) do
   3. sample \( X \sim Q \)
   4. if \( X = \beta \) then
      5. count \( \leftarrow \) count + 1
   6. else if \( X \prec \beta \) then
      7. \( \beta \leftarrow X \)
      8. count \( \leftarrow 0 \)
9. return \( \beta \)
Algorithm 1: CONFIRMATION_SAMPLING(Q, t, <)

1 \[ \beta \leftarrow \infty, \text{ count } \leftarrow 0 \]
2 \[ \textbf{while} \text{ count } < t \textbf{ do} \]
3 \[ \text{ sample } X \sim Q \]
4 \[ \text{ if } X = \beta \text{ then} \]
5 \[ \quad \text{ count } \leftarrow \text{ count } + 1 \]
6 \[ \text{ else if } X < \beta \text{ then} \]
7 \[ \quad \beta \leftarrow X \]
8 \[ \text{ count } \leftarrow 0 \]
9 \[ \textbf{return } \beta \]

Theorem 3. Let Q denote a probability distribution with finite support S. For \( x_1 = \min(S) \) and \( X \sim Q \) let \( p_1 = \Pr[X = x_1] \) and let \( p_2 = \max\{\Pr[X = x] \mid x \in S\setminus\{x_1\}\} \) be the largest sampling probability among elements of S other than \( x_1 \). Then:

\[
\Pr[\text{CONFIRMATION_SAMPLING}(Q, t) \neq x_1] \leq (1 - p_1) \left( \frac{p_2}{p_1 + p_2} \right)^t
\]

The expected number of samples made by CONFIRMATION_SAMPLING is bounded by \( (t + 1)/p_1 \).
Application to nearest neighbor

**Theorem 1.** Suppose there is a sequence of independent, randomized data structures $D_1, D_2, \ldots$, such that on query $q$, $D_i$ returns the nearest neighbor of $q$ in $P$ with probability at least $p_q$ and each other point in $P$ with probability at most $p_q$. Let $\delta > 0$ be given. There is an algorithm that depends on $\delta$ but not on $p_q$ that on input $q$ queries data structures $D_1, \ldots, D_{j_q}$, performs $j_q$ distance computations, where $E[j_q] = O(\ln(1/\delta)/p_1)$, and returns the nearest neighbor of $q$ with probability at least $1 - \delta$. 
Application to nearest neighbor

**Theorem 1.** Suppose there is a sequence of independent, randomized data structures $D_1, D_2, \ldots$, such that on query $q$, $D_i$ returns the nearest neighbor of $q$ in $P$ with probability at least $p_q$ and each other point in $P$ with probability at most $p_q$. Let $\delta > 0$ be given. There is an algorithm that depends on $\delta$ but not on $p_q$ that on input $q$ queries data structures $D_1, \ldots, D_{j_q}$, performs $j_q$ distance computations, where $\mathbb{E}[j_q] = O((\ln(1/\delta))/p_1)$, and returns the nearest neighbor of $q$ with probability at least $1 - \delta$.

- Finding the nearest neighbor quickly boils down to minimizing the product of the expected time for querying $D_i$ and $1/p_1$. 
Finding the nearest neighbor quickly boils down to minimizing the product of the expected time for querying $D_i$ and $1/p_1$.

**Theorem 1.** Suppose there is a sequence of independent, randomized data structures $D_1, D_2, \ldots$, such that on query $q$, $D_i$ returns the nearest neighbor of $q$ in $P$ with probability at least $p_q$ and each other point in $P$ with probability at most $p_q$. Let $\delta > 0$ be given. There is an algorithm that depends on $\delta$ but not on $p_q$ that on input $q$ queries data structures $D_1, \ldots, D_{j_q}$, performs $j_q$ distance computations, where $\mathbb{E}[j_q] = O(\ln(1/\delta)/p_1)$, and returns the nearest neighbor of $q$ with probability at least $1 - \delta$.

**Question:** Can this be done optimally without knowledge of the distance to the nearest neighbor?
Partial answer for LSH forest
Partial answer for LSH forest

Tree = recursive space partitioning
Partial answer for LSH forest

Standard query algorithm: Search level where number of points in q’s space partition is O(1) on average.
Partial answer for LSH forest

Tree = recursive space partitioning

Standard query algorithm: Search level where number of points in \( q \)'s space partition is \( O(1) \) on average.

Works if number of trees is high enough, depending on distance to nearest neighbor.
Partial answer for LSH forest

Tree = recursive space partitioning
Partial answer for LSH forest

Tree = recursive space partitioning

Modified query algorithm: Search higher levels until confirmation search says we found the nearest neighbor.
Partial answer for LSH forest

Modified query algorithm: Search higher levels until confirmation search says we found the nearest neighbor.
Partial answer for LSH forest

Modified query algorithm: Search higher levels until confirmation search says we found the nearest neighbor.
Partial answer for LSH forest

Modified query algorithm: Search higher levels until confirmation search says we found the nearest neighbor.

Also consider partially searching a level.
Partial answer for LSH forest

Modified query algorithm: Search higher levels until confirmation search says we found the nearest neighbor.

Also consider partially searching a level.

Tree = recursive space partitioning

Competitive with best way of searching LSH forest for given query
Open question

• Is it possible to achieve space and time that is $O(1)$-competitive with the best LSH scheme, adapted to the query and to the data distribution, for a given expected recall?