Lab 7: Approximate Nearest Neighbor in High Dimension

Monday, October 29
CompSci 531, Fall 2018
Outline

• Nearest Neighbor Problem in Low Dimension

• High Dimension Application: Classifying Articles in the Bag of Words Model

• Locality Sensitive Hashing
Nearest Neighbor Problem

• Given $n$ points $P$, a similarity measure $S()$, and a query point $q$, find $x$ in $P$ that maximizes $S(x, q)$. Call this $\text{NN}_S(q)$.

• Equivalently, given...distance measure $D()$...that minimizes $D(x, q)$. 

![Diagram of Nearest Neighbor Problem]

- $P$ represents the set of points.
- $q$ is the query point.
- $x$, the point maximizing $S(x, q)$, is illustrated as the closest point to $q$. 
- $S(x, q)$ denotes the similarity measure between $x$ and $q$. 

...End of Image
Similarity Measures for Geometric Data

• Suppose that our data is geometric: each point \( x \) in \( P \) is a point in \( d \)-dimensional Euclidean space.

• Euclidean:

\[
S(x, y) = -\sqrt[d]{\sum_{i=1}^{d} (x_i - y_i)^2}
\]

• Cosine:

\[
S(x, y) = \cos \theta_{xy} = \frac{x \cdot y}{\|x\|_2 \|y\|_2} = \frac{\sum_{i=1}^{d} x_i y_i}{\sqrt{(\sum_{i=1}^{d} x_i^2)(\sum_{i=1}^{d} y_i^2)}}
\]
Nearest Neighbor Problem

• Of course, we can always trivially answer $\text{NN}_S(q)$ in $O(n)$ time by scanning over all of $P$. So why is this interesting?

• Consider a live application where you want to answer these queries in time that scales sublinearly with $n$. Can we preprocess the data so that this is possible?

• Example: suppose $P$ is one dimensional. Can you preprocess to get $\log(n)$ query time?
Nearest Neighbor Problem

• So there is hope! What if our points are 2-dimensional?

• Solution: kd-trees. This is a form of hierarchical clustering. We won’t go through the full construction today, but here is the idea in pictures.
kd-Tree

{A, B, C, D, E, F}

{A, B, C}

{D, E, F}
kd-Tree
kd-Tree
Curse of Dimensionality

• We will leave as an exercise figuring out how to use the kd-tree to get an $O(\log(n))$ algorithm for the nearest neighbor problem.

• This solution suffers from the *curse of dimensionality*. That is to say, it does not scale well with the dimensionality of the data.

• Instead, we want a nearest neighbor algorithm that still runs quickly in high dimension, perhaps at the cost of accuracy.

• Why would we care about this?
Outline

• Nearest Neighbor Problem in Low Dimension

• High Dimension Application: Classifying Articles in the Bag of Words Model

• Locality Sensitive Hashing
Application: Classifying Articles, Bag of Words

• Suppose we have n news articles, of m basic types (e.g., politics, sports, etc). As input, we are told the type of each of these n articles.

• We want to build a classifier from news articles to basic types, i.e., a function that given a new article, predicts what type it is.

• Option 1: Natural Language Processing.
  • (Not in this course)
Application: Classifying Articles, Bag of Words

- Option 2: We will use nearest neighbor classification in the bag of words model.

- Suppose there are $d$ “important” words used across all $n$ articles (so not including articles, prepositions, etc.)

- Represent each article as a vector $x \in \mathbb{R}^d$ where that $x_i$ is the number of times that word $i$ appears in that article.
  - We are simplifying our data by entire ignoring the order in which the words occur.
  - Note that $d$ is large, likely on the order of 100,000!
Application: Classifying Articles, Bag of Words

• Now, our classifier using the nearest neighbor problem is incredibly simple.

• Let $S(x, y) = \cos \theta_{xy} = \frac{x \cdot y}{\|x\|_2 \|y\|_2}$; we will use cosine similarity.

• To classify a new article with bag of words representation $y$, let $x = \text{NN}_S(y)$. Output the type of $x$.

• Thus, if we can efficiently solve the nearest neighbor problem in high dimension in sublinear time, we can do efficient classification of high dimensional data.
Outline

• Nearest Neighbor Problem in Low-Dimension

• High-Dimension Application: Classifying Articles in the Bag of Words Model

• Locality Sensitive Hashing
Locality Sensitive Hash in General

• Recall the standard universal hashing assumption: for any \( x \neq y \), \( \Pr[h(x) = h(y)] \leq \frac{1}{n} \), for a hash table of size \( n \). Such hash functions try to obscure how similar \( x \) and \( y \) are.

• Could we define a hash function with the opposite sort of property? One for which the probability of a collision depends on how similar \( x \) and \( y \) are?

• If so, maybe we can approximately solve the nearest neighbor problem by hashing with multiple trials, as we have seen in the count min sketch!
Locality Sensitive Hash for Cosine Similarity

• For cosine similarity, recall that $S(x, y) = \cos \theta_{xy} = \frac{x \cdot y}{\|x\|_2 \|y\|_2}$, which varies between -1 and 1.

• We want a locality sensitive hash function $h$ such that $\Pr[h(x) = h(y)] \approx \frac{1 + S(x, y)}{2}$, where the randomness will (as usual) come from the random draw of $h$ from a family.

• Solution: Draw a random unit vector $r \in \mathbb{R}^d$, say by taking $r_i \sim N(0,1)$ for every coordinate and normalizing. Now let $h_r(x) = \text{Sign}(x \cdot r)$, that is, +1 if the inner product is nonnegative, and -1 otherwise.
Locality Sensitive Hash for Cosine Similarity

$$\Pr[h_r(x) = h_r(y)] = \Pr[\text{Sign}(x \cdot r) = \text{Sign}(y \cdot r)]$$

$$= 1 - \frac{\theta_{xy}}{\pi} = 1 - \frac{\cos^{-1} S(x, y)}{\pi} \approx \frac{1 + S(x, y)}{2}$$
Algorithm for Nearest Neighbor Problem

At a high level, we want to use our locality sensitive hash $h()$ to

1. hash all of our data
2. answer nearest neighbor queries by hashing the query point and only searching over the colliding data points.

- **Problem.** The hash we developed only maps to -1 or +1, so this could still require us to search over roughly half of the points at each step.

- **Solution.** Create a new hash function $H()$ by drawing $k$ independent hash functions $h_1, \ldots, h_k$ and letting $H(x) = (h_1(x), \ldots, h_k(x))$. 
Algorithm for Nearest Neighbor Problem

- **Solution.** Create a new hash function \( H() \) by drawing \( k \) independent hash functions \( h_1(), \ldots, h_k() \) and letting \( H(x) = (h_1(x), \ldots, h_k(x)) \).
  - Recall that each \( h_i() \) is defined by drawing a random unit vector in \( \mathbb{R}^d \).
  - Also note that the hash function \( H() \) maps to \( 2^k \) possible buckets, since it is a length \( k \) bit string.

- Now, \( \Pr[H(x) = H(y)] = \left(1 - \frac{\theta_{xy}}{\pi}\right)^k \), so we can substantially cut down the number of other points we have to scan over.

- **Problem.** Since we look at fewer points, our error is likely to increase.

- **Solution.** Draw \( l \) independent hash functions \( H_1(), \ldots, H_l() \), and search over collisions on any of these
Algorithm for Nearest Neighbor Problem

Altogether then, here is our algorithm. We have n articles, each represented as a vector \( x \in \mathbb{R}^d \). We have parameters \( k \) and \( l \).

• There are \( l \) hash tables, \( T_1, \ldots, T_l \), each with \( 2^k \) buckets
• To define the hash functions, draw \( l \) matrices \( M_1, \ldots, M_l \), each of dimension \( k \times d \), where every entry is drawn independently from a standard normal distribution \( N(0,1) \).
  • Normalize every row of every matrix to be a unit vector.
• The \( i \)'th hash of some \( x \) is \( \text{Sign}(M_i x) \). Store \( x \) in \( T_i \).
  • Note that this is a vector of \( k \) values in \{ -1, +1 \}, which you will have to map to the \( 2^k \) table \( T_i \) somehow.
Example

Suppose we have \( l = k = 2 \), and we are tracking \( d = 5 \) words in our bag of words model (this is a toy example).

We draw 2 random 2 by 5 matrices where every row is a unit vector:

\[
M_1 = \begin{bmatrix} 0.70 & -0.27 & -0.04 & 0.65 & -0.14 \\ 0.71 & -0.38 & -0.26 & 0.36 & -0.39 \end{bmatrix} \quad M_2 = \begin{bmatrix} -0.11 & 0.07 & -0.63 & -0.73 & 0.23 \\ -0.64 & 0.68 & -0.22 & -0.19 & 0.22 \end{bmatrix}
\]

To hash the inputs \( x = (5, 1, 0, 2, 0) \), we compute: \( M_1x = (4.53, 3.89) \) and \( M_2x = (-1.95, -2.89) \). We take the sign to get the hash values \((1, 1)\) and \((-1, -1)\). Then we store \( x \) the corresponding tables.
Algorithm for Nearest Neighbor Problem

• To compute the nearest neighbor of an article, also represented in the bag of words model as some $y \in \mathbb{R}^d$:
  • Scan over all $x$ hashed to the same bucket in at least one of the $l$ hash tables.
  • Among all such, $x$, return the one with maximum similarity to $y$.

• If we then want to solve the classification problem using nearest neighbor classification, simply classify the query point $y$ as the same class as the $x$ returned as the nearest neighbor.

• **Question.** How do we decide how to set $l$ and $k$?
Reasoning About the Parameters

• The greater the value of $k$, the lower the probability that a collision happens on any given hash table.

• So as we increase $k$, we expect to have to scan over fewer points looking for a nearest neighbor.

• = faster query time, but less accurate results.

• The greater $l$ is, the more independent hashes we compute for each data point.

• Since we compare any points that collide on at least one hash, as we increase $l$, we expect to increase the probability that we find a good nearest neighbor.

• = more accurate results, but slower query time.
Formal Guarantees

• You may have noticed that we have been extremely loose with our guarantees, for example:
  • What is the big-O runtime?
  • What is our approximation or probability of correctness?

• We can formulate the problem more formally as follows. The approximate nearest neighbor problem asks a query $NN(y, r, c)$, with $r \geq 0, c \geq 1$. We want to give an algorithm that, with constant probability (say 1/3):
  • If there is an $x^*$ with $S(x^*, y) \geq r$, returns some $x$ such that $S(x, y) \geq r/c$.
  • If there is no $x^*$ with $S(x^*, y) \geq r/c$, reports failure
  • Else, reports failure or returns some $x$ such that $S(x, y) \geq r/c$. 
Formal Guarantees

• This parameterized version of the problem is easier to work with in theory, although what we have already described in the more practical version.

• Using the techniques we have already seen, one can prove that it is sufficient to set $k \approx \frac{1}{cr} \log(n)$ and $l = n^{1/c}$ to solve this problem with probability at least $1/3$.

• To get an error probability of, say, 1%, just run the algorithm $\lceil \log_3 100 \rceil = 5$ times and take the best (most similar) result.
Formal Guarantees

• One can show that the expected total number of similarity comparisons you have to make during a query with these parameters is $O \left( \frac{n^{1/c}}{cr} \log(n) \right)$.

• So, for example, if we take $r=1$ and $c=2$, we get an expected query time of $O(\sqrt{n} \log(n))$. That’s a lot better than $O(n)$!

• See [MunagalaLectureNotes](#) for these details (also linked under optional reading for this lab).
Practical Guarantees

• That said, you might be left wondering: how well does nearest neighbor classification work in practice for our news article classification problem?

• Lucky for you, you will implement and test this on just such a data set in lab homework 3.
Summary

• We described the nearest neighbor problem in computational geometry; where the key idea is to trade off space and preprocessing to get sublinear query time.

• For low dimensional data, kd-trees are very effective solutions. The curse of dimensionality makes them impractical in high dimension.

• We care about high dimensional data for applications like classification of documents in the bag of words models.

• We can use locality sensitive hashing to approximately solve the nearest neighbor problem for high dimensional data.