Efficient K-nearest-neighbor for discrete feature spaces with separable distance function

Marco Cusumano-Towner

March 20, 2012

1 Problem

Suppose we have a discrete features space $F = F_1 \times \cdots \times F_M$, where the $F_m$ are discrete sets. Suppose our training examples are nonredundant (a given point in feature space only appears at most once in the training set). Then the set of training examples is $S \subseteq F$. Suppose $S$ is generated randomly by choosing each element of the whole feature space $F$ to be included in our training set $S$ with probability $p$. Let $x^{(i)} = (x^{(i)}_1, \ldots, x^{(i)}_M)$ denote an example point in feature space. Suppose our distance function is separable across the feature sets $F_1, \ldots, F_M$:

$$d(x^{(i)}, x^{(j)}) = \sum_{m=1}^{M} d_m(x^{(i)}_m, x^{(j)}_m)$$

This is the case for $\ell_1$ or squared $\ell_2$ norms, for example. Given a test example, $y \in F$ with, we want to find the $K$ elements $s \in S$ that have the smallest $d(y, x^{(s)})$.

2 Naive solution

If we enumerated all elements $s \in S$, and computed the distance $d(y, x^{(s)})$ for each one, we would have $|S|$ distance computations and $|S|$ operations to get the $K$ smallest elements using a selection algorithm—$O(|S|)$ operations to find the $K$-th smallest element, and $O(|S|)$ operations to scan through the list and extract elements smaller than it. Since $E|S| = p \prod_{m=1}^{M} |F_m|$, we have on average $O(p \prod_{m=1}^{M} |F_m|)$ operations.

3 Algorithm

We first introduce the general problem of finding the $K$-smallest-elements using prior comparisons and an algorithm for this problem. We then show that our $K$-nearest-neighbor problem can be solved using an efficient application of this algorithm.
3.1 General K-smallest-elements using prior information

Consider the following simple algorithm for finding the smallest $K$ elements in an ordered set $Q$, given prior information consisting of some statements of the form $a \leq b$ for $a, b \in Q$. We assume that no cycles of the form $a \leq b \leq \ldots \leq a$ exist in this prior information. The prior information takes the form of a partial ordering that can be represented as a directed-acyclic-graph $G = (Q, E)$, where a directed edge $(a, b) \in E$ indicates that $a \leq b$.

Consider the elements $P \subseteq Q$ that have no incoming edges. Each element in the complement $Q \setminus P$ is known to be $\leq$ than some element in $P$ (this follows from the absence of cycles, and the transitivity of $\leq$). Therefore, a minimum element of $Q$ is guaranteed to lie in $P$, and in particular, a minimum element of $P$ is guaranteed to be a minimum element of $Q$.

The algorithm, (Algorithm 1) undergoes $K$ iterations and finds a next-smallest element $u$ at each iteration, adding this element to the list $J$. At each iteration, $u$ is the smallest element in $P$. We remove $u$ from $P$ and we remove the corresponding node from the graph. If there are nodes that now have no incoming edges due to this modification of the graph (possibly the children of $u$), we add them to $P$, so that $P$ maintains the property described above. If $P$ is implemented as a min-heap, then insertion or deletion of a node involves executing $O(\log |P|)$ comparisons, and finding the smallest element is $O(1)$. If there is no prior information given, then all $|Q|$ nodes are added to $P$ in the initialization, taking $O(|Q|)$ operations, and the $K$ extractions of the minimum elements takes $O(K \log |Q|)$ for a total running time of $O(|Q| + K \log |Q|)$.

**Algorithm 1** General K-smallest-elements using prior information

```plaintext
J ← { }. Initialize $P$ with all nodes in $G$ with no incoming edges. for $t = 1 \rightarrow K$ do
  $u$ ← get-min($P$)  $\triangleright$ $O(1)$
  remove($P, u$)  $\triangleright$ $O(\log |P|)$
  $J$ ← ($J, u$)
  for $v \in V$ with $(u, v) \in E$ do  $\triangleright$ for each child $v$ of $u$
    if $\lnot(\exists w \neq u$ with $(w, v) \in E)$ then  $\triangleright$ if $u$ is the only parent of $v$
      insert($P, v$)  $\triangleright$ $O(\log |P|)$
    end if
  end for
  remove($G, u$)
end for
return $J$
```
3.2 K-smallest-elements of a sum

Consider vectors \( v_m \in \mathbb{R}^{L_m} \) for \( m = 1, \ldots, M \). We seek to find \( K \) tuples of indices of the form \((i_1, \ldots, i_M)\) with \( i_m \in \{1, \ldots, L_m\} \) for each \( m \) that have the \( K \) smallest values of

\[
v_{1i_1} + v_{2i_2} + \cdots + v_{Mi_M} = \sum_{m=1}^{M} v_{mi_m}
\]

Suppose for each \( m = 1, \ldots, M \), we sort the elements of \( v_m \) and keep the index vectors \( a_m \) (such that \( a_{m1} \) is the index of the smallest element in \( v_m \)). This requires \( O(\sum_{m=1}^{M} L_m \log L_m) \) operations. This gives us partial ordering information of the form used in the above algorithm. In particular, we have a DAG \( G = (Q, E) \) where the vertices are positions in the \( M \)-dimensional array \( Q = \{1, \ldots, L_1\} \times \cdots \times \{1, \ldots, L_M\} \) with an edge \((i, j)\) precisely when for exactly one \( m \) we have \( j_m = i_m + 1 \), and for all \( r \neq m \), \( i_r = j_r \). An example graph for \( M = 2 \) is shown below. Under this partial ordering, an element \((i_1, \ldots, i_M)\) is less than another element \((j_1, \ldots, j_M)\) if \( i_m \leq j_m \) for all \( m \), and greater than \((j_1, \ldots, j_M)\) if \( i_m \geq j_m \), and not comparable otherwise.

We apply Algorithm 1 to the special case for this graph structure and result with Algorithm 2. Note that we do not actually construct the graph \( G = (V, E) \) or enumerate all of its elements. Instead, we use a dynamic \( M \)-dimensional boolean array \( R \) that indicates whether a particular index vector \( i \) has been removed yet.

3.2.1 Complexity

Since we can’t add more than \( M^2 \) nodes to \( P \) per iteration, We have the simple bound \( |P| \leq M^2 K \) (although this could probably be improved significantly). The running time of Algorithm 2 is then \( O(K \log(M^2 K) + KM \log(M^2 K)) = O(MK \log MK) \). Combined with the sorts at the beginning the running time for the \( K \)-smallest-elements of a sum is \( O(\sum_{m=1}^{M} L_m \log L_m + MK \log MK) \).
**Algorithm 2** K-smallest-elements of a sum

\[
\begin{align*}
J & \leftarrow \{\} \\
P & = \{(1, \ldots , 1)\} & \triangleright (1, \ldots , 1) \text{ is the only node with no incoming edges, } O(1) \\
R & = [\ ] \\
\end{align*}
\]

for \( t = 1 \rightarrow K \) do

\[
\begin{align*}
i^* & \leftarrow \text{get-min}(P) & \triangleright i^* \text{ is a vector of indices e.g. } i^* = (1, 4, 2, 3), \ O(1) \\
\text{remove}(P, i^*) & & \triangleright O(\log |P|) \\
J & \leftarrow (J, i^*) & \triangleright O(1) \text{ amortized} \\
R(i^*) & \leftarrow \text{true.} & \triangleright \text{Remove } i^* \text{ from the graph, } O(1) \text{ amortized} \\
\end{align*}
\]

for \( r = 1 \rightarrow M \) do

\[
\begin{align*}
J & \leftarrow \text{true} \\
j & \leftarrow i^* + e_r & \triangleright j \text{ is the child of } i^* \text{ along dimension } r \\
\text{for } s = 1 \rightarrow M \text{ do} & \quad \triangleright \text{For each parent of } j \text{ (one parent per dimension)} \\
& \quad \text{if } j_s > 1 \text{ then} \\
& \quad \quad k & \leftarrow j - e_s & \triangleright k \text{ is a parent of } j \\
& \quad \quad \text{if } \neg R(k) \text{ then} & \triangleright \text{If } k \text{ is still in the graph, then don’t add } j \\
& \quad \quad \text{add} & \leftarrow \text{false} \\
& \quad \text{end if} & \triangleright O(\log |P|) \\
& \quad \text{end if} & \triangleright \text{For each child of } i^* \text{ (one child per dimension)} \\
& \text{end for} \\
& \text{if add then} \\
& \quad \text{insert}(P, j) \\
& \text{end if} \\
\end{align*}
\]

end for

end for
3.3 Extension to valid subset

Suppose we are only allowed to choose elements from a specified subset of $Q$. We can modify Algorithm 1 to only add an element to $J$ if it is valid. We then modify the for loop over $1, \ldots, K$ to a while loop that terminates when $|J| = K$. For the special case of Algorithm 2, we have $S \subseteq \{1, \ldots, L_1\} \times \cdots \times \{1, \ldots, L_M\}$, and we only add the $i^*$ to $J$ if $i^*$ is valid. If each element of $Q$ has independent probability $p$ of being selected for $S$, then on average this modified algorithm will require $(1/p)$ times more iterations than the original, and the running time is $O\left(\sum_{m=1}^{M} L_m \log L_m + p^{-1} MK \log MK\right)$.

3.4 Application to K-nearest-neighbor

For a training example $y = (y_1, \ldots, y_M)$, we compute the distances $d(y_m, x_m^{(s)})$ for every $m$ and $s \in F_1 \times \cdots \times F_M$, and place them into vectors $v_m$. We then apply the modified version of Algorithm 2 and solve the problem in $O\left(\sum_{m=1}^{M} |F_m| \log |F_m| + p^{-1} MK \log MK\right)$. This is compared to $O\left(p \prod_{m=1}^{M} |F_m|\right)$ for the brute force method. If $p$ is not very small, then this method is a major improvement.
Clearly we have $|P| \leq 2K$ at all iterations, and so the running time is $O(K \log K)$. The running time for the full algorithm, including the initial sorts is $O(n \log n + m \log m + K \log K)$. The memory is $O(K^2)$ for a simple implementation of $A$ and $O(K)$ for $P$ and $J$. If we restrict ourselves to considering only a subset $S$ of the pairs, the expected number of iterations is $K/p$ for $p \gg 0$ where $p$ is the fraction of $m \times n$ pairs that are in $S$, if $S$ is chosen randomly. This algorithm can be scaled up to more than 2 combinations.

Instead, suppose we first sort $x$ and $y$ separately (least-to-greatest), to get sorted versions $\hat{x}$ and $\hat{y}$ with indices $a, b$. This is $O(m \log m + n \log n)$. Let $(i, j)$ now denote indices into the sorted $\hat{x}$ and $\hat{y}$ (we can go back to the original unsorted indices with $a_i$ and $b_j$). The two independent sorts provide a partial ordering over all pairs $(i, j)$. In particular, for two pairs $(i, j)$ and $(s, t)$ we have:

$$i \leq s \text{ and } j \leq t \implies \hat{x}_i + \hat{y}_j \leq \hat{x}_s + \hat{y}_t$$

This can be represented by a two-dimensional grid structure DAG:

A comparison operation between nodes $(i, j)$ and $(s, t)$ involves computing $\hat{x}_i + \hat{y}_j$ and comparing to $\hat{x}_s + \hat{y}_t$. When the general algorithm above is applied to this DAG, it becomes:

### 3.5 Algorithm

Use a dynamic two-dimensional array $A$ to allow for $O(1)$ lookup of whether a pair has been added to $L$ (or equivalently, removed from $G$).

0. Initialize $P = \{(1, 1)\}, L = \{}$

For $t = 1 \ldots K$:

1. $(i^*, j^*) := \text{get-min}(P)$. $O(1)$.

2. remove($P, (i^*, j^*)$). $O(\log |P|)$.

3. add($L, (i^*, j^*)$). $O(1)$.

4. $A(i^*, j^*) = \text{true}$. $O(1)$ amortized.

5. If $A(i^* + 1, j^* - 1)$ or $j^* = 1$, then insert($P, (i^* + 1, j^*)$). $O(\log |P|)$.

6. If $A(i^* - 1, j^* + 1)$ or $i^* = 1$, then insert($P, (i^*, j^* + 1)$). $O(\log |P|)$.

Return $L$

Clearly we have $|P| \leq 2K$ at all iterations, and so the running time is $O(K \log K)$. The running time for the full algorithm, including the initial sorts is $O(n \log n + m \log m + K \log K)$. The memory is $O(K^2)$ for a simple implementation of $A$ and $O(K)$ for $P$ and $L$. If we restrict ourselves to considering only a subset $S$ of the pairs, the expected number of iterations is $K/p$ for $p \gg 0$ where $p$ is the fraction of $m \times n$ pairs that are in $S$, if $S$ is chosen randomly. This algorithm can be scaled up to more than 2 combinations.
Suppose we don’t care about ties. Let $A^t$ denote a set of the $t$ smallest pairs. Note that $A^1 = \{(1,1)\}$. Viewing $Z$ as a grid with $i$ increasing top-to-bottom and $j$ increasing left-to-right, the $A^t$ cover a closed region in the upper-left corner.

The algorithm idea is to keep track of a frontier set of pairs $P^t$ that are incomparable with each other based on the independent sorts alone, but is guaranteed to contain the $t+1$'th smallest pair.

### 3.6 Algorithm

$P^0 = \{(1,1)\}$, $A^0 = \{\}$

For $t = 1 \ldots K$:

1. $(i^*, j^*) := \text{argmin}_{(i,j) \in P^{t-1}} \hat{x}_i + \hat{y}_j \ (|P^{t-1}| \text{ operations})$
2. $A^t := A^{t-1} \cup \{(i^*, j^*)\}$
3. $P^t := P^{t-1} \setminus \{(i^*, j^*)\}$
4. If $(i^* + 1, j^* - 1) \in A^{t-1}$ or $i^* = 1$, then $P_t = P_t \cup \{(i^* + 1, j^*)\}$
5. If $(i^* - 1, j^* + 1) \in A^{t-1}$ or $j^* = 1$, then $P_t = P_t \cup \{(i^*, j^* + 1)\}$

Return $A^K$

### 3.7 Proof of correctness

Proof by loop-invariants. Let $Z = \{1 \ldots m\} \times \{1 \ldots n\}$. We have the following loop invariants for all $t = 1 \ldots K$:

1. The elements of $P^t$ form a frontier that is guaranteed to contain a $t-1$'th smallest element: $(i,j) \in Z \setminus (A^t \cup P^t) \implies \exists (r,s) \in P^t: r \leq i \wedge s \leq j \implies x_r + y_s \leq x_i + y_j$
2. The elements of $A^t$ are the $t$'th smallest elements:
   $(i,j) \in A^t \implies \forall (r,s) \in Z \setminus A^t: \hat{x}_i + \hat{y}_j \leq \hat{x}_r + \hat{y}_s$

Base cases:

1. $(1,1) \in P^0$ has $1 \leq i \wedge 1 \leq j$ for all $(i,j) \in Z \supset (Z \setminus (A^0 \cup P^0))$.
2. Vacuously true since $A^0 = \{\}$.

Induction step:

1. Since $(i^*, j^*)$ is the only element in $P^{t-1}$ that is not in $P^t$, is suffices to show that
   $i^* \leq r \wedge j^* \leq s \implies \exists (p,q) \in P^t: p \leq r \wedge q \leq s$. We break into cases:
   - If $i > i^*$, then we consider $(i^* + 1, j^*)$. If $(i^* + 1, j^*)$ was added to $P^t$, we are done, this serves as the required element in $P^t$. If $(i^* + 1, j^*)$ was not added, then $j^* > 1$ and $(i^* + 1, j^* - 1) \not\in A^{t-1} \implies (i^* + 1, j^* - 1) \in P^{t-1} \setminus A^{t-1}$ or $(i^* + 1, j^* - 1) \in P^{t-1}$.
Z \ (A^{t-1} \cup P^{t-1})). If (i^* + 1, j^* - 1) ∈ P^t, we are done since it is also in P^t. If (i^* + 1, j^* - 1) ∈ Z \ (A^{t-1} \cup P^{t-1}) then by (2) of the previous iteration, we have ∃(p, q) ∈ P^t : p ≤ i^* + 1 ∧ q ≤ j^* - 1. Since this (p, q) cannot be (i^*, j^*), it must still be in P^t, and we are done.

If j > j^* the same argument can be applied with i’s and j’s switched, using whether or not (i^*, j^* + 1) was added to P^t.

Otherwise, i = i^* and j = j^* and (i^*, j^*) ∉ Z \ (A^t \cup P^t).

2. We have Z \ A^t = (Z \ A^{t-1}) \ {(i^*, j^*)} and A^t = A^{t-1} \ {(i^*, j^*)}. Therefore by invariant (2) at the previous iteration it suffices to show that ∀(r, s) ∈ Z \ A^t : x_{i^*} + y_{j^*} ≤ x_r + y_s. By the definition of (i^*, j^*) we have x_{i^*} + y_{j^*} ≤ x_r + y_s for all (r, s) ∈ P^{t-1}. To extend this to all (r, s) ∈ P^t, we only need to additionally consider (r, s) = (i^* + 1, j^*) and (r, s) = (i^*, j^* + 1), which have i^* + 1 ≥ i^* ∧ j^* ≥ i^* \implies \hat{x}_{i^*} + \hat{y}_{j^*} ≤ \hat{x}_{i^*+1} + \hat{y}_{j^*}, and i^* ≥ i^* ∧ j^* + 1 ≥ i^* \implies \hat{x}_{i^*} + \hat{y}_{j^*} ≤ \hat{x}_{i^*} + \hat{y}_{j^*+1}. We then use (1) of the current iteration to extend this to all (r, s) ∈ Z \ (A^t \cup P^t). Since Z \ A^t = P^t \cup (Z \ (A^t \cup P^t)) (because A^t and P^t are disjoint), we are done.

Invariant (2) at t = K certifies that the algorithm returns the K-smallest pairs.

3.8 Running time

Geometrically, if Z is laid out in a grid, then P^t is exactly the set of ‘corners’ on the boundary of the A^t region, which occupies the upper-left corner. For a given number of corners c the minimum area is a triangle with area \(\frac{c^2 - c}{2}\). Therefore, for a given area \(|A^t| = t\) the number of corners is less than \([1/2 + 1/2\sqrt{8t + 1}]\), which gives \(|P^t| \leq [1/2 + 1/2\sqrt{8t + 1}]\). Assuming the set operations are all \(O(1)\) the algorithm is then \(O(\sum_{t=1}^{K}\sqrt{t}) = O(K^{3/2})\). To see this consider the increase of both functions between \(K\) and \(K + 1\). For the first function the increase is \(\sqrt{K + 1}\). For \(K^{3/2}\), which is convex, the derivative gives that increase is at least \((3/2)\sqrt{K}\). Constant \(C = 4/3\) gives \(\sqrt{K + 1} \leq C(3/2)\sqrt{K}\) for all \(K \geq 1\). A \(O(n)\) implementation of the set operations leads to a \(O(K^2)\) running time.

3.9 Extension to subset case

Consider the case when not all of the pairs \((i, j) \in \{1\ldots m\} \times \{1\ldots n\}\) are valid:

\((i, j)^*_1, \ldots, (i, j)^*_K = k\arg\min_{(i, j) \in S} x_i + y_j\)

with \(S \subseteq Z = \{1\ldots m\} \times \{1\ldots n\}\). Then we use the above algorithm, but we have an additional list \(B^k\) we only add to if the \((i^*, j^*) \in S\). The algorithm then terminates when \(|B^k| = k\). If the elements in \(S\) are sampled randomly from \(Z\) with some high probability \(p\) of each \((i, j) \in Z\) being selected, then the expected running time is proportional to \(1/p\) times the running time when \(S = Z\). The worst-case running time is \(O(mn)\) (if all the elements of \(S\) are clustered near the bottom-right corner of \(Z\)).
4 Application to K-nearest-neighbor

This algorithm is useful for a K-nearest-neighbor problem where the feature space is the cartesion product of two discrete feature spaces $F_1 \times F_2$, with $|F_1| = m$ and $|F_2| = n$. For now, we assume that no two examples are identical. Then the training set can be expressed as a subset $S$ of the features space $F_1 \times F_2$. We also require that distance function $d((i, j), (s, t))$ is separable: $d((i, j), (s, t)) = d_1(i, s) + d_2(j, t)$.

First, as a preprocessing step we compute all the pairwise distances $d_1(i, s)$ for all $i, s \in F_1$ and similarly for $d_2(j, t)$ for all $j, t \in F_2$, we store these in matrices $D_1$ and $D_2$ ($|F_1| \times |F_1|$ and $|F_2| \times |F_2|$, respectively). This takes $O(m^2 + n^2)$.

Next, we sort the columns of $D_1$ and the columns of $D_2$, to give matrices $\hat{D}_1$ and $\hat{D}_2$. This takes $O(m^2 \log m + n^2 \log n)$.

Given a test example $(i, j)$ with $i \in F_1$ and $j \in F_2$, we extract the sorted distances $d_1(i, s)$ for all $s$ and $d_2(j, t)$ for all $t$ (the appropriate column of $\hat{D}_1$ and the appropriate column of $\hat{D}_2$). These vectors of distances are the $\hat{x}$ and the $\hat{y}$ of the algorithm above. Running this algorithm has $O(K^{3/2})$ or $O(K^2)$ running time, depending on the implementation of the set operations.

The total running is then $O(m^2 + n^2 + m^2 \log m + n^2 \log n + N_{test}K^2)$, although in practice the $O(m^2)$ and $O(n^2)$ distance computations have a much larger constant than the $O(m^2 \log m)$ and $O(n^2 \log n)$ sorts, and therefore dominate. Ignoring the shared features and treating each train and test example as independent would result in $N_{test}|S|$ distance evaluations. If $|S| \cdot N_{test} = \Omega(mn)$ (as would be the case with random sampling with some fixed $p$ for cross validation for example), then the running time is at least $\Omega(m^2n^2)$. 