Algorithms to Calculate the Manhattan (L1) Distance for Vertical Data Represented in pTrees.

Mohammad K. Hossain, Arjun G. Roy, Arijit Chatterjee, William Perrizo
Department of Computer Science
North Dakota State University
Fargo, ND 58105, USA

ABSTRACT.

In data mining applications different types of distance metrics are used to measure the closeness of two data points. Among these metrics Manhattan (L1), Euclidean (L2) and Max (L∞) distances are used very frequently in various algorithms. In pTree vertical data representation Max distance can be efficiently implemented using only bitwise operations across the pTrees without any horizontal access of the data points. But many clustering and classification algorithms require computing L1 and L2 distances in order to increase their accuracy. In this paper we have shown how Manhattan or L1 distance can be calculated for vertical data represented in pTrees. Similar to the Max distance this algorithm also uses only bitwise operations across various pTrees without performing any horizontal scan of the data points. As a result the algorithm works very fast on huge volume of data represented by pTrees comparing with traditional horizontal data representation. Also these algorithms enable various data mining algorithms that use pTrees to improve their accuracy without sacrificing any significant speed.

1. INTRODUCTION

A distance metric measures the closeness between two data points giving the result in some numerical value [1]. Less the distance is, closer the points are and vice versa. Given two points X and Y in the n-dimensional space, d(X,Y) measures the distance between and X and Y where function d maps any two points into a positive real number such that d(X, Y) > 0, if (X ≠ Y) and d(X, Y) = 0, if (X = Y). A general form of this distance is the weighted Minkowski distance. Considering a point, X, in n-dimensional space as a vector \(<x_1, x_2, x_3, ..., x_n>\), the weighted Minkowski distance,

\[ d_p(x,y) = \left( \sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p} \]

where, \( p \) is a positive integer, \( x_i \) and \( y_i \) are the \( i^{th} \) components of \( X \) and \( Y \) respectively.

\( w_i \) (\( i \leq 0 \)) is the weight associated to the \( i^{th} \) dimension or \( i^{th} \) feature.

Considering the weights \( w_i = 1 \) for all \( i \) the distance function becomes

\[ d_p(x,y) = \left( \sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p} \] which is also known as the \( L_p \) distance.

Manhattan Distance: When \( p = 1 \), the Minkowski distance is called the Manhattan distance or \( L_1 \) distance which is:

\[ d_1(x,y) = \sum_{i=1}^{n} |x_i - y_i| \]

Euclidian Distance: If \( p = 2 \), the Minkowski distance is known as the Euclidian distance or \( L_2 \) distance, which is:

\[ d_2(x,y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \]

---

1 We acknowledge partial financial support for this research from a Department of Energy Award (award # DE-FG52- 08NA28921).
Max Distance: When \( p = \infty \), the summation, in the Minkowski distance is known as Max distance or \( L_\infty \) distance which is:

\[
d_{\infty}(X, Y) = \max_i |x_i - y_i|
\]

... ... ... ... (4)

Figure 1: Three types of distances shown in a two-dimensional space between two points \( X(2, 1) \) and \( Y(6, 4) \).

From the figure 1, it is clear that \( d_1 \geq d_2 \geq d_\infty \) for any two points \( X \) and \( Y \).

The algorithm discussed in this paper assumes that the data are represented in vertical form using a special data structure called predicate tree or pTree which is a lossless, compressed and data mining-ready data structure. In the next section of this paper, we shall discuss pTree and its construction. The algorithm uses 2’s complement arithmetic to perform addition and subtraction required to calculate the distance. Next subsequent sections we shall briefly discuss 2’s complement arithmetic and different bitwise operation required by the algorithm to compute the \( L_1 \) distance using pTree. Then the algorithm will be discussed with its various aspects.

2. VERTICAL DATA REPRESENTATION USING P-TREE

Predicate tree or pTree is the vertical data representation that represents the data column-by-column rather than row-by-row (which is relational data representation). It was initially developed for mining spatial data [2][4]. Since then it has been used for mining many other types of data [3][5]. The creation of pTree is typically started by converting a relational table of horizontal records to a set of vertical, compressed P-trees by decomposing each attribute in the table into separate bit vectors (e.g., one for each bit position of a numeric attribute or one bitmap for each category in a categorical attribute). Such vertical partitioning guarantees that the information is not lost.

For example, let \( R \) be a relational table consists of three numeric attributes \( R(A_1, A_2, A_3) \). To convert it into pTree we have to convert the attribute values into binary then take vertical bit-slices of every attribute and store them in separate files. Each bit slice is considered as a pTree, which indicates the predicate if a particular bit position is zero or one. This bit slice may be compressed dividing it into binary trees recursively. Figure 2 depicts the conversion of a numerical attribute, \( A_1 \), into pTrees.

<table>
<thead>
<tr>
<th>( A_1 )</th>
<th>( A_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>7</td>
<td>101</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
</tbody>
</table>

Figure 2. Construction of pTrees from attribute \( A_1 \). The pTrees are built from the top down, stopping any branch as soon as purity (either purely 1-bits or purely 0-bits) is reached (giving the compression).
Different bitwise operations like AND (&), OR (|), XOR(^), NOT (!) are executed on pTrees which works simply as logical operations.

3.2 Subtraction using 2’s complement.

Assume another binary number M. When we add 2’s complement of N with M, mathematically we get the result of M – N in the following way:

\[ M + 2's \text{ complement of N} = M + 2^n - N \]

Thus (M + 2’s complement of N) gives us (M – N) plus \(2^n\) which is the (n+1)th bit also known as the carry bit of this addition operation. So if \(M \geq N\), (M + 2’s complement of N) gives the M – N after discarding the carry. If \(M < N\), (M + 2’s complement of N) = \(2^n - (N - M) = 2's \text{ complement of (N - M)}\). That is, (M + 2’s complement of N) has no carry in this case. It also indicates that the result of (M – N) is negative and is represented in the 2’s complement form. So taking 2’s complement of (M + 2’s complement of N) will give the absolute value of (M – N) using lemma 2.

3.3 Calculation of summation of two integers.

The algorithm presented in this paper utilizes the procedures of adding two numbers represented in binary bits starting from adding two single bit numbers then expanding the process to add two arbitrary n-bit numbers. When adding two single bit numbers (assume a and b) the possible result we shall get is a two bit number where the least significant bit is the sum (s) and most significant bit is the carry (c). The following bitwise operations compute the value of s and c from a and b

\[ s = a \text{ ^ } b \]
\[ c = a \text{ & } b \]

Now assume we have two n-bit numbers A and B which are represented in binary as follows:

\[ A = a_n, a_{n-2} ... a_0 \]
\[ B = b_n, b_{n-2} ... b_0 \]

to add two number of n-bit each, the algorithm starts from least significant bit (which is bit o) and proceed to left to the more significant bits and ends at the most significant bit (which is bit n-1). At any step i of these steps the algorithm adds three bits \(a_i\)
(the $i^{th}$ bit of A), $b_i$ (the $i^{th}$ bit of B) and $c$ (the carry produced as a result of similar operation in step $i-1$) except for $i=0$ where $c=0$). Following operations are executed in step $i$:

\[ s_i = a_i \land b_i \lor c \]

\[ c = (a_i \land b_i) \land (a_i \lor b_i) \land c \]

**4. ALGORITHM TO CALCULATE THE MANHATTAN ($L_1$) DISTANCE**

The algorithm assumes a set of points $(x,y)$ where $x$ and $y$ are $n$-bit values. Both $x$ and $y$ are represented by $n$ pTrees. We calculate the distance from each points from a center point $(p,q)$ where $p$ and $q$ are $n$-bit values. The distances are produced in $(n+1)$ pTrees. Following steps are involved in calculating the distances:

i) Convert the value $p$ and $q$ into their binary.

ii) Convert them into their 2’s complement. Assume the values are $np$ and $nq$ respectively.

iii) Use the AddValueToPtree function of Algorithm 1 to add the value $np$ with $x$ pTrees and $nq$ with $y$ pTrees. Assume the results are found in $Rx$ and $Ry$ pTrees respectively.

iv) Use FindAbsoluteValue function of Algorithm 2 to find the absolute value pTrees of $Rx$ and $Ry$ pTrees. The results are found in $Ax$ and $Ay$ pTrees respectively.

v) Use CalculateSumPtree function of Algorithm 3 to add $Ax$ and $Ay$ pTrees and get the result in $D$ pTrees.

Here are the algorithm 1, 2 and 3 used to calculate the distances:

---

**Algorithm 1: Add a Constant value with a data set of n bits using pTree**

AddValueToPtree()

\[
\text{ptree } a[N]; /*input pTrees representing}\n\text{N bit data by } a[N-1], a[N-2] ... a[2], a[1], a[0] */\n\text{bits } v[N]; /* a constant value}\n\text{represented by N bits } v[N-1], v[N-2] ... v[2], v[1], v[0] */\n\text{ptree } s[N+1]; /* output pTrees}\n\text{representing N+1 bit data by}\n\text{s[N], s[N-1] ... s[2],}\n\text{s[1], s[0]}*/\n\]

\[
\text{ptree } carry; /*store the carry bits when}\n\text{a bit is added with a pTree */}\n\text{int } i;\n\]

\[
\text{carry = 0; /*initial carry pTree is zero,}\n\text{i.e. there is no carry*/}\n\text{for (i = 0; i<N; i++)}\n\text{if (val[i] == 0)}\n\text{\quad s[i] = a[i] ^ carry;}\n\text{\quad carry = a[i] & carry;}\n\text{\}\n\text{else}\n\text{\quad s[i] = a[i]’ ^ carry;}\n\text{\quad carry = a[i] | carry}\n\text{\}\n\text{\quad s[N] = carry;}\n\]
Algorithm 2: Find absolute value of a data set of \( n \) bits using pTree

\[
\text{FindAbsoluteValue()}{
\begin{array}{c}
\text{ptree } s[N+1]; \quad /* \text{The value represented by the following } N+1 \text{ pTrees will be converted into absolute value; } s[N], s[N-1] \ldots s[2], s[1], s[0] \text{ Originally the values are } N \text{ bit value represented by } s[N-1] \ldots s[2], s[1], s[0], \text{where } s[N] \text{ determines the sign. When a bit of } s[N] \text{ is 1(or 0), bits of } s[N-1] \ldots s[0] \text{ represent a positive value(or negative value). A negative value is converted into absolute value by changing it to its 2's complement. At the end of the algorithm pTrees } s[N-1] \ldots s[2], s[1], s[0] \text{ hold the absolute value of the data set.} */
\end{array}
\]
\]

\[
\text{ptree carry; } /* \text{store the carry bits when a bit is added with a pTree */}
\]
\]

\[
\text{int i; carry = 0; } /* \text{initial carry}
\]
\]

\[
\text{if } s[N] = (s[N])';
\]
\]

\[
\text{for(i=0; i<N; i++)}{
\]
\]

\[
\text{s[i] = s[i] ^ s[N];}
\]
\]

\[
\text{t1 = a[i] & b[i];}
\]
\]

\[
\text{if } i = 1; \text{ i<N; i++}
\]
\]

\[
\text{s[i] = s[i] ^ carry;}
\]
\]

\[
\text{t2 = sum & carry;}
\]
\]

\[
\text{carry = t1 | t2;}
\]
\]

\[
\text{s[N] = carry;}
\]
\]

5. ANALYSIS OF THE ALGORITHMS

Algorithms 1, 2 and 3 introduced in the previous section use only the bitwise operation AND, OR, NOT and XOR among the various pTrees. The number of these operations depends on the \( n \), the number of bits required to represent the value of each dimension of the data points. Following table shows the number of required deferent bitwise operations with respect to \( n \).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Algorithm 1</th>
<th>Algorithm 2</th>
<th>Algorithm 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR</td>
<td>( n )</td>
<td>( 2n )</td>
<td>( 2n-1 )</td>
</tr>
<tr>
<td>AND</td>
<td>Max. ( n )</td>
<td>Min. ( n )</td>
<td>Total: ( n )</td>
</tr>
<tr>
<td>OR</td>
<td>Max. ( n )</td>
<td>Min. ( n )</td>
<td>( n )</td>
</tr>
<tr>
<td>NOT</td>
<td>Max. ( n )</td>
<td>Min. ( n )</td>
<td>( n )</td>
</tr>
</tbody>
</table>

This number does not depend on the size of the data points we are considering. As a result this method works equally faster when we are dealing with large volume of data. So the algorithm works in a constant time irrespective of the data size. However each individual bitwise operation depends on the size of the pTrees but as the bitwise operations are fast enough [7] comparing with
other arithmetic operations, the algorithm will outperform any horizontal data processing algorithms to calculate the distances on large volume data.

6. CONCLUSION

This algorithm uses the vertical data format represented in pTrees and calculates the $L_1$ distance using only bitwise operations. We showed that it should work very fast keeping the accuracy unchanged. So this algorithm will be particularly useful in classification and clustering algorithm of vertical data format where distances are calculated to find the nearest neighbor or similarity of various data points. As the algorithm uses pTree we hope this will give acceptability of pTree further more in data mining arena.

REFERENCES


