Function Compression

Ali Khudiyev

Abstract. There is always a bunch of data points among the all points that can represent the whole data set with some predefined accuracy. Such data points may have the potential to reduce the complexity of a function that can represent the whole data set as well as reducing the the amount of required storage or memory to save the whole set. In the other words, there is always a set of linear equations which can represent more complex function by keeping maximum error level in control. This paper covers the essentials and important details in order to find better set of linear functions to compress the original function efficiently and the solution to the problem is given by using Dijkstra algorithm.

Introduction

To be able to compress the discrete points of a given function we are using simply linear functions which are easy to compute and can be perfectly fit onto any function. It is also not less obvious that we are looking for much accuracy while having less linear functions or in the other words, it is better for our purpose to get less number of linear functions while maintaining as much accuracy as possible. The problem becomes easier if represented as a graph. To transform our problem into a graph we need to look at each of those linear functions which are going to represent our whole function, as two points and one edge that connects them in order to create a linear function $y = ax + b$.

Once the problem can be represented as a graph we are going to use Dijkstra's algorithm to solve it since this algorithm finds the shortest path on a positively weighted graph. However, there is still a problem of choosing proper weights for each and every edge in order to find the right set of linear functions. This leads us to ask the question of what decides the weights? To answer this important question it is better to visualize some random data points first:

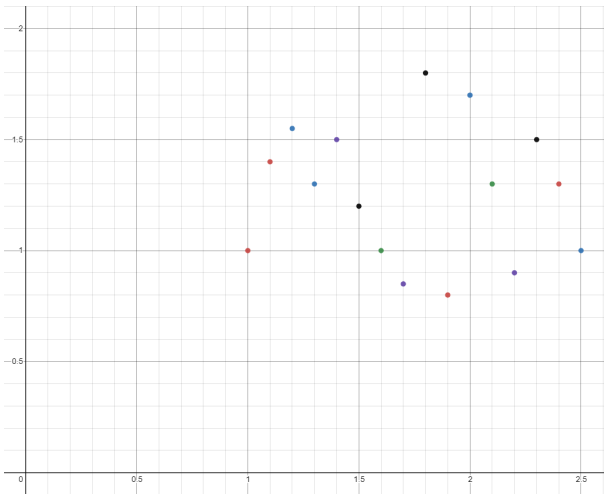


Figure 1: Data points

As it seems from the figure 1 there are lots of ways to compose linear functions to represent the whole data set. However, we do not want to have less accuracy while representing the set, and in fact, we can set the accuracy to be some number between 0 and 1 in order to measure how accurate our compression is(i.e. we do not want to have a compressed point p' which has the significant high difference with the corresponding original point p in the set). Secondly, we want to compress the set as much as possible. This is to say, we measure our compression with the help of two main metrics: *accuracy* and *the number of linear functions*.

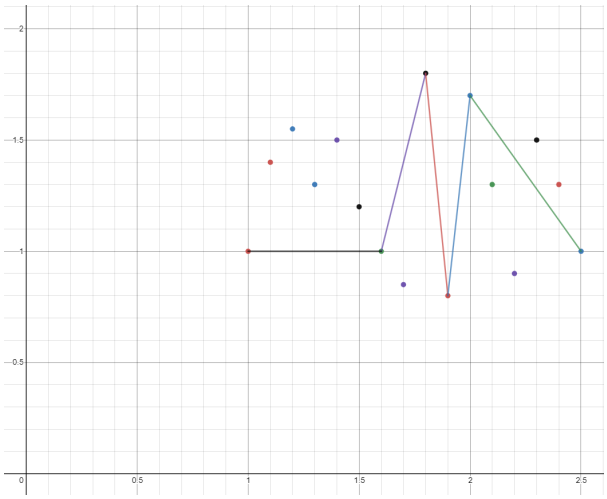


Figure 2: Bad compression

Figure 2 is an example of bad compression. It is not good because $d_c = 5/7$ or approximately 71.43% compression degree while maintaining the maximum error of 0.55.

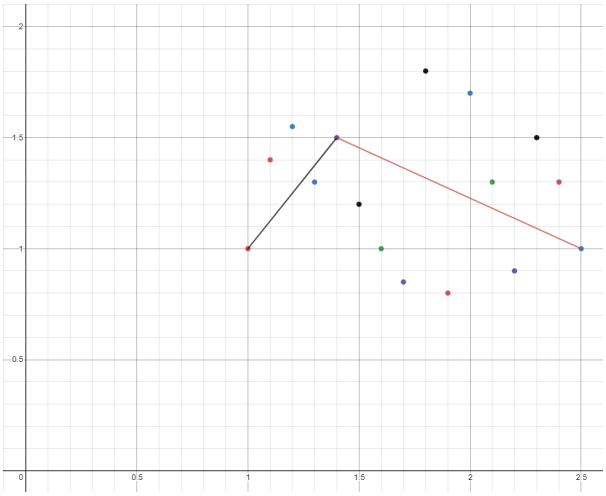


Figure 3: Good compression

Opposite to the previous figure, this figure 3 is an example of good compression since its compression degree is approximately 92.86% ($\sim$ 21.4% better) while maintaining the maximum error of 0.555.

Metrics

Distance between two points

Distance between two points is an euclidean distance between them.

Error of a point

Error of a point is is to measure the error or the vertical distance between the y value of that point and the corresponding output of the linear function whose domain contains that point.

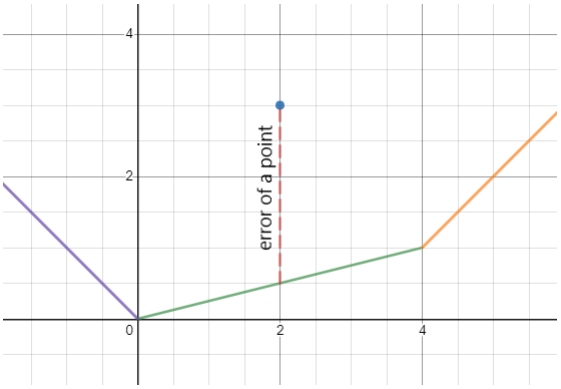


Figure 4: Error of a point

Weight of a line

Weight of a line is set according to the two parameters: *distance between the endpoints of that line* and *the accuracy of that line*.

Accuracy of a point

Accuracy of a point shows the degree of its error being how much closer to either 0 or predefined maximum error value. If the error of a point is 0 then its accuracy is 1, if the error is equal to the predefined maximum error value then its accuracy is 0, and if the error bigger than the predefined maximum error then the accuracy is a negative number.

Accuracy of a line

Accuracy of a line is the mean accuracy of all points located in the domain of that line.

Accuracy of a compression

Accuracy of a compression is the mean accuracy of all lines used in that compression.

Compression degree

Compression degree is to measure the quality of compression. We define it as a decreasing linear function in the interval of $[2, \#V]$ starting from 1 and ending with 0. This is due to our logic that we want the compression degree to be at maximum 1 when only the list of compressed points consists of only two points and we want it to be 0 when the list contains all data points or in the other words, the number of data points is #V. In between 2 and #V we want the compression degree to decrease gradually.

Why $d_C = 1$ when the list of compressed points contains only 2 data points? -Because we say that the best compression can only happen when the number of chosen points is two and obviously, it cannot be less than two since we need at least two points to represent the whole data.

Why $d_C = 0$ when the list of compressed points contains all the data points in the set? -Because this is the maximum number of points that we can represent our data set with 100% accuracy, so, the compression degree would be 0 since we have used all the points of a data set to represent itself.

Note: #C is the number of points used in the list of compressed points and #V is the number of all data points.

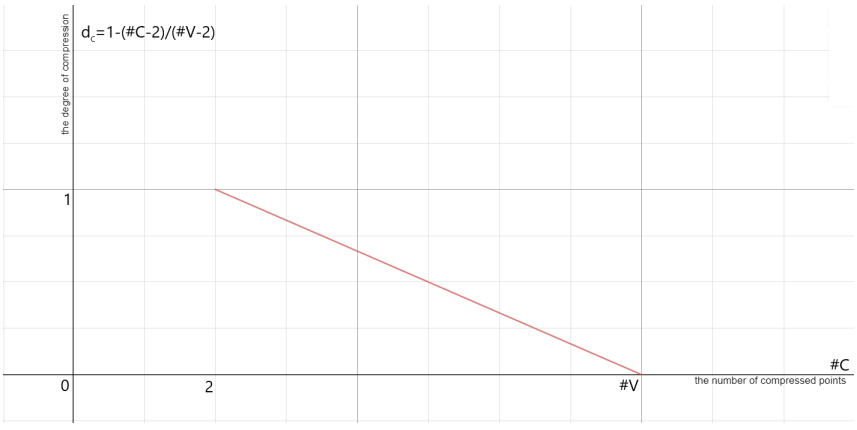


Figure 5: The linear function of compression degree

Distance between two points	$d(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
Error of a point p	$e(p) = \lvert l(p) - f(p) \rvert$
Weight of a line	$w(l) = [d(p_1, p_2)(e_{\max} - \max(e_p))]^{-1}$
Accuracy of a point	$\rho(p) = 1 - \frac{e(p)}{e_{\max}}$
Accuracy of a line l	$\alpha(l) = \frac{1}{\#l} \sum_{p \in l} \rho(p)$
Overall accuracy of compression C	$A_C = \frac{1}{\#C} \sum_{l \in C} \alpha(l)$
Compression degree	$d_C = 1 - \frac{\#C - 2}{\#V - 2}$

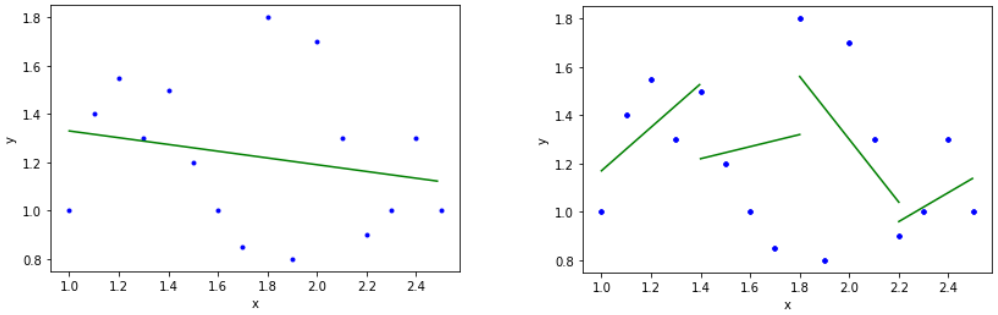
Table 1: Formulas for metrics

Data Generator

In order to generate data **some high degree polynomial** was used as an example. For generating 2-D data it is known that we are strictly prohibited to have more than one y-value for any x-value - this constraint is observed in the provided solution for generating data.

Linear Regression

It could be easily observed that linear function is not the best fitting function for polynomial function all the time(especially when the degree of the polynomial is bigger than 1):



(a) Linear

(b) Multi step (step=0.4)

Figure 6: Regressions

In the figure 6 (a) there is only one step which is $[1, 2.5]$ while in (b) there are four steps with the equal step value of 0.4 (which means the steps are $[1, 1.4]$, $[1.4, 1.8]$, $[1.8, 2.2]$, $[2.2, 2.6]$ respectively). As it seems from (b) this four linear functions are somehow similar to the one show in figure 3 since they tend to give the least error while representing their

parts (however, in linear regression method there is no maximum error value(s) unlike what our method requires).

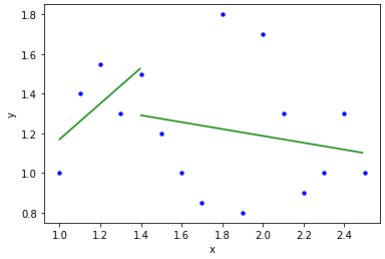


Figure 7: 2-step linear regression

In the figure 7 the steps are $[1, 1.4]$ and $[1.4, 2.5]$ respectively. It is even more obvious from this figure that the linear regression is related to our method but it does not give us what we want: the control over the error and the least number of points which can represent the whole data while maintaining the predefined maximum error level. Since linear regression does not have such parameter controls it is not able to do the process we want automatically, instead, we should give it the number of steps and the step values. Even in that case, it does not check the maximum error to know whether choosing a particular point is what we need to do or not.

Implementation

Algorithm 1 Create Graph

Require: Data **data**
  Create an empty *Graph*
  Initialize nodes with 3 attributes names x, y and *weight*
  while node $\in$ V **do**
    Initialize x of node with the corresponding x value of the point
    Initialize y of node with the corresponding y value of the point
    Initialize *weight* of node with the value returned by **weight function**(1)
  end while
  return Graph

Algorithm 2 Dijkstra

Require: Graph, u(starting node), v(ending node, *optional*)

```
Create vertex set Q
while v ∈ Graph do
    dist[v] ← ∞
    prev[v] ← UNDEFINED
    Add v to Q
end while
dist[source] ← 0
while Q is not empty do
    u ← vertex in Q with min dist[u]
    Remove u from Q
    while v ∈ neighbour of u do
        alt ← dist[u]+length(u,v)
        if alt < dist[v] then
            dist[v] ← alt
            prev[v] ← u
        end if
    end while
end while
return dist[], prev[]
```

Algorithm 3 Compress

Require: Graph, Dijkstra, u and v (vertices) V

```
Use Dijkstra to find the the list of shortest paths P from u
Nodes ≔ list(v)
current ≔ P[v]
Append current to Nodes
while current ≠ u do
    current ← P[current]
    Append current to Nodes
end while
return Nodes
```

With the help of the algorithms shown above we can implement our idea just by putting them in the right order. Therefore, the final view of the program would look like this:

Algorithm 4 The Complete Algorithm

Require: Data, Create Graph, Dijkstra, Compress

```
Create Graph with Data
Use Dijkstra to Compress Graph
Print the final nodes that represent Data
Print some useful metrics
Plot both Data and compressed data points
```

Other applications

As a conclusion we see that the compression of any mathematical function can be done by using either Dijkstra or Bellman-Ford's algorithm with a proper weight initializing. However, this is not the only problem that this method can solve, in fact, the method can be extended towards the fields of compression of multi-variable functions, artificial neural networks or compression of data sets as well as data clustering. To give a bit more insights why we think this is possible there is a list below covering the general idea of implementation of this method in different fields.

- **Multi-variable Function Compression**

  Instead of connecting 2D points with 1D line, imagine connecting n dimensional points with n-1 dimensional 'line' which is called a hyper-plane. So, for any n dimensional points we can do the same thing with one less dimensional hyper-planes.

- **Compression of Data Sets**

  We can apply this method to compress data sets since data sets are data points of some function. For example, if there is a valid data set with n features then it has at least one n-variable function that fits the data set perfectly, therefore, we can compress the data set itself with some predefined maximum error value or some accuracy.

- **Data Clustering**

  If we know the labels of data points or in the other words, it is a 'supervised clustering' by using this method we can cluster the points belonging to the same labels effectively according to their distances between themselves. Therefore, at the end we would be able to have such clusters that represent the data well in the terms of well-organized groups with some labels.

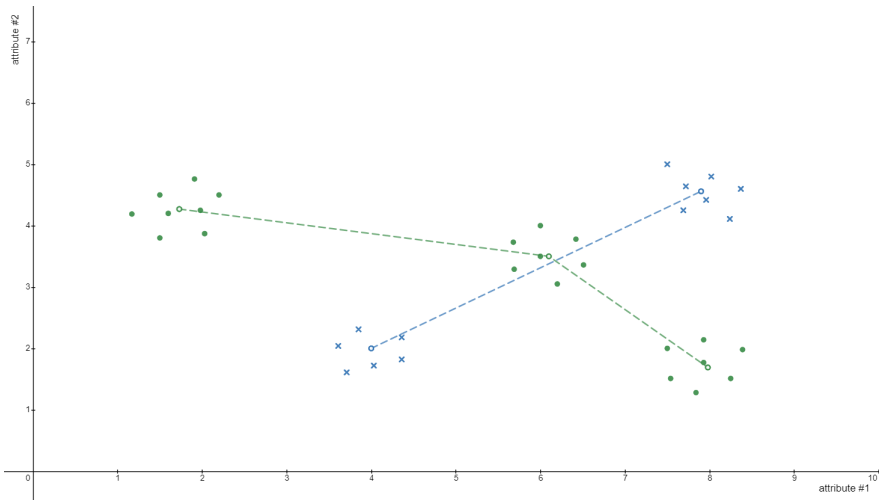


Figure 8: Supervised Data Clustering

Here a weight $w_{p,p'}$ between a point p and p' is set according to the labels, distances and errors of the points p and p'. The general formula would look like this:

$$w_{p,p'} = \begin{cases} \infty, & \text{if } \alpha(p,p') \text{ is less than predefined minimum accuracy} \\ \infty, & \text{else if } p \text{ and } p' \text{ doesn't have the same label} \\ d(p,p')^{-1}, & \text{otherwise} \end{cases}$$

This proper initialization of weights would lead the program to cluster the data points resulting with something like shown in the figure 8.

- **Artificial Neural Networks**

  Data clustering would do a lot in artificial neural networks by the means of finding the appropriate number of perceptrons or units in each hidden layer. Since the number of linear functions that can separate the data according to their labels is equal to the number of needed perceptrons it would be better if we clustered the data properly. After the clustering process there are only left the points which represents their group or in the other words, the centroids of each group. Since the number of points that we should work on has been decreased significantly and also the selected points are centroids of each data cluster it is much easier to try to draw linear functions which separates these different labeled data points from each other.

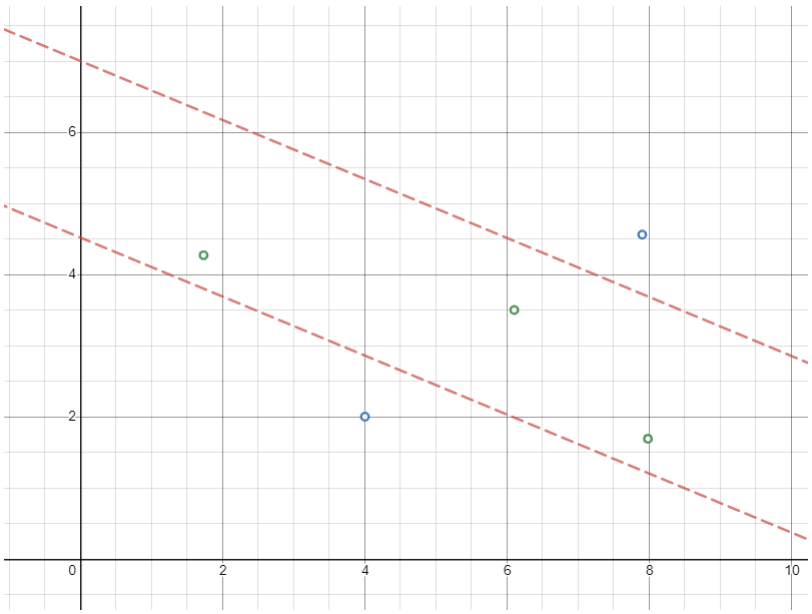


Figure 9: The number of lines also represents the number of needed perceptrons

By having these centroids as the result of our method it would ease the job significantly by resulting with these 5 points which refers to the central points of their clusters, and therefore, it would be easy to find such two lines that separate the different labeled points. However, some further exploration and understanding is required to improve the separation process as it might not give the desired results all the times.