Motivating the Necessity of Feature Scaling in Machine Learning

Julius Kasiske

October 2022

1. Introduction

Before training machine learning models and after the data has been split into training, validation and testing sets, the data should be scaled in many cases. We do it after splitting the data because scaling can cause outliers to be replaced with central tendency measures, e.g. the mean, but we don't want our validation data to be used for computing that mean as we do not want validation data to influence our model training. After all, that is why we split in the first place. Why scaling data in general is necessary will be motivated below.

2. The Problem of Skewed Distance Metrics

The base problem that many machine learning algorithms have with unscaled data is that most of them rely on measurements of distance between data points. For instance, k-Means Clustering computes distances between observations in order to classify them into clusters. The kNN-Algorithm has tangent functionality.

The most popular distance measure used in these and other algorithms is the Euclidean distance. For any given vector $\vec{v} = (v_1, \ldots, v_n)$, it is defined as follows.

$$||\vec{v}|| = \sqrt{v_1^2 + v_2^2 \ldots + v_n^2} \tag{1}$$

Thus, if all v_i for all $i \in [1, n]$ are equal, they contribute equally to the Euclidean distance of $\vec{v}$. For instance:

$$||\vec{1}_3|| = \sqrt{1^2 + 1^2 + 1^2} \tag{2}$$

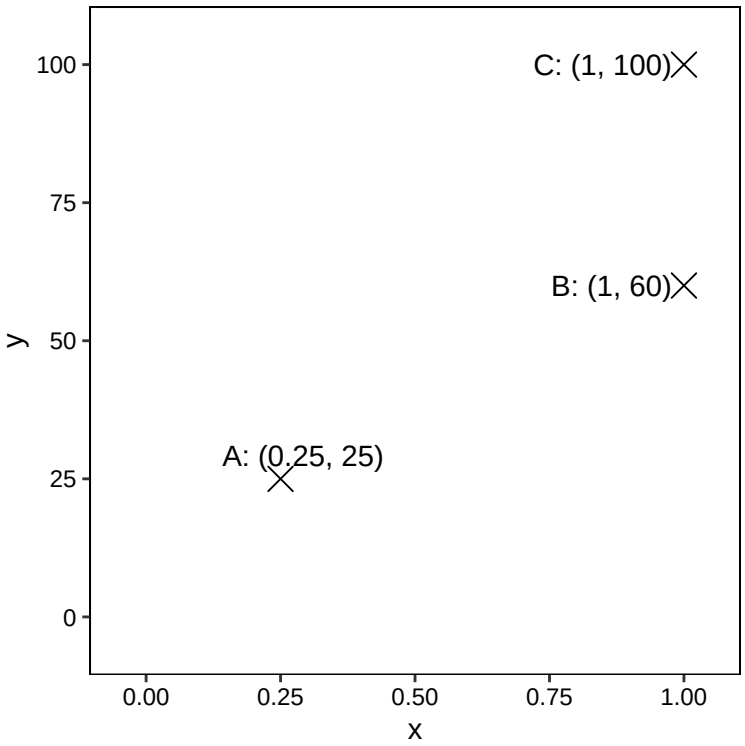
Here, all three dimensions contribute one third of the distance.

If however one of the measurements, e.g. v_1 is scaled by a factor of 100, then we get the following:

$$||(100, 1, 1)|| = \sqrt{100^2 + 1^2 + 1^2} \tag{3}$$

All of a sudden, v_1 contributes almost all of the distance to the final result, thus causing the metric to put little weight on v_2 and v_3. These phenomena cause skewed results when making decisions based on these resulting distances, which is the reason why we scale all variables before applying algorithms that rely on distances.

Consider the following visualized example. Three points are measured in two dimensions, both are supposed to range from 0 to 1. However, because the y-dimension is not scaled, its range is from 0 to 100 instead. The points have the following coordinates:



Imagine our goal now would be to measure whether the point B is closer to A or to C. In clustering for example, that information would be essential to assign B to one of the two clusters. By eye-balling, we would come to the conclusion that B is closer to C than it is to A. And that conclusion would be true if both axes had the same scale, e.g. from 0 to 1. But since the y-axis is not scaled, the distances $||\vec{AB}||$ and $||\vec{BC}||$ compute as follows:

$$||\vec{AB}|| = \sqrt{(1-0.25)^2 + (60-25)^2} = \sqrt{0.75^2 + 35^2} = 35.008 \tag{4}$$

$$||\vec{BC}|| = \sqrt{(1-1)^2 + (100-60)^2} = \sqrt{0^2 + 40^2} = 40 \tag{5}$$

We observe that our Euclidean distance disagrees with our eye-balled conclusion, saying that B is closer to A than it is to C.

That is because distance measures like the Euclidean metric are overly sensitive to large values and thus sacrafice the accuracy that we seek by understanding the meaning behind the data. A popular example is to measure body height in centimeters instead of in meters.

3. Problems in Gradient Descent

3.1 Gradient Descent with Distance Measures

To illustrate the problem above, lets look at one algorithm that is often under the hood of Linear Regression, the Gradient Descent Algorithm. While it comes in many shapes, lets go over the basic idea step by step. We look at Gradient Descent in the context of Least Squares Regression. Thus, the objective function Q might look as follows:

$$\min Q(\beta, \alpha) = \frac{1}{n} \sum_{i=1}^{n} [y_i - (\alpha + \beta x_i)]^2 \tag{6}$$

Beware that y_i and x_i are not function parameters to be adjusted for minimization but rather constants in the function term that are just generalized in the depiction of the sum.

We are all familiar with the basic process of finding minima of functions. We find domain elements for which the functions gradient is $\vec{0}$. In a second step we then classify each candidate point as a minimum, maximum or saddle point. Gradient Descent implements that rational for our objective function $Q(\beta, \alpha)$ as follows:

1. The algorithm is initialized by choosing more or less a random point in the domain $(\beta^{(0)}, \alpha^{(0)})$.
2. For the (k)-th iteration, check if the gradient is $\vec{0}$ or close to $\vec{0}$. If true, the algorithm would terminate, because we have found a minimum or a point close enough to it.

$$||\nabla Q(\beta^{(k-1)}, \alpha^{(k-1)})|| < \varepsilon \tag{7}$$

3. Find the normalized search direction ν.

$$\nu = -\frac{\nabla Q(\beta^{(k-1)}, \alpha^{(k-1)})}{||\nabla Q(\beta^{(k-1)}, \alpha^{(k-1)})||} \tag{8}$$

4. Find the new point (β^k, α^k) whose function value is lower than $Q(\beta^{(k-1)}, \alpha^{(k-1)})$ by starting at $(\beta^{(k-1)}, \alpha^{(k-1)})$ and going along the search direction ν using an adequate step size λ. The new point from which to reiterate off of is thus defined as follows:

$$\begin{bmatrix} \beta^{(k)} \\ \alpha^{(k)} \end{bmatrix} = \begin{bmatrix} \beta^{(k-1)} \\ \alpha^{(k-1)} \end{bmatrix} + \lambda \cdot \nu \tag{9}$$

The above equation raises the question of how to choose λ. While methods like the exact line search exist, often times, λ is chosen by heuristics that check if an actual descent is made using a certain base value. After choosing some base value for λ, these heuristics could look as follows:

$$\begin{aligned} &while \quad Q(\beta^{(k)}, \alpha^{(k)}) \geq Q(\beta^{(k-1)}, \alpha^{(k-1)}) : \\ &\lambda = \lambda \cdot \frac{1}{2} \end{aligned}$$

The attentive reader might have realized that both step 2 and step 3 make use of the Euclidean norm.

3.2 Gradient Descent without Distance Measures

We can, however, try to omit using distance measures in Gradient Descent, which, however, creates new unwanted problems. Lets go through them step by step.

In step 2 and equation (7), we only use the Euclidean norm to check if the gradient is close enough to $\vec{0}$. We could try to circumnavigate that problem by defining the condition to be satisfied if $\nabla Q(\beta^{(k-1)}, \alpha^{(k-1)}) = \vec{0}$, but that will numerically rarely ever be the case. The runtime of the algorithm would thus be greatly extended, limiting its efficacy. If we left the definition of the termination condition as is (i.e. $||\nabla Q(\beta^{(k-1)}, \alpha^{(k-1)})|| < \varepsilon$) and we had unscaled data whose features vary greatly in their ranges, we have two options:

1. Choose ε to be rather high, s.t. changes in the derivative of coordinates with large value ranges can be detected quicker. Then, on-par runtime of the algorithm can be preserved. The downside is that the condition would severely neglect inaccuracies in directions, whose coordinate (feature in ML lingo) has a small value range, leading to performance losses of the algorithm.

2. Choose ε to be rather low, s.t. we can avoid said performance losses. However, that would bring about the downside that the runtime would be severely extended.

If we left the termination condition in step 2 as is, our Gradient Descent would either be severely slower or less performant.

We essentially run into similar issues with the use of distance measures in step 3 (equation (8)). If we omit the normalization of the search direction, we have to adjust our step size λ, which takes time and leads to inefficiencies.

4. Conclusions

We have talked about the core issue that arises when avoiding feature scaling in Machine Learning: Skewed distance measures. We further discussed what that problem can lead to in the case of Least Squares Regression, one of the most common Machine Learning techniques in existence. There are, however, many other techniques that run into similar issues with distance metrics with unscaled data. Most of them usually come back to two core drawbacks: Loss of efficiency or runtime and loss of accuracy or performance. Beware that models like Decision Trees and Random Forests usually perform well even without scaling, even though scaling will generally still not hurt model performance. That is why this summary ends with one core message: After splitting the data and before building the models, scale your data! More on how that can be done will follow.