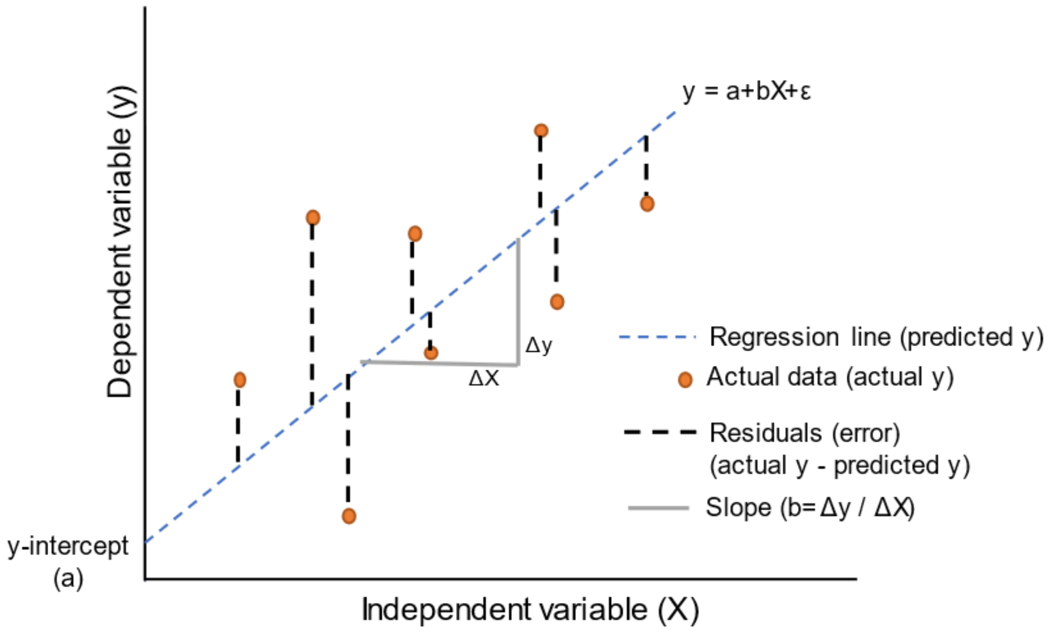
Linear regression

Linear regression is a supervised machine learning algorithm that models the linear relationship between independent (X) variables and dependent variable (y).



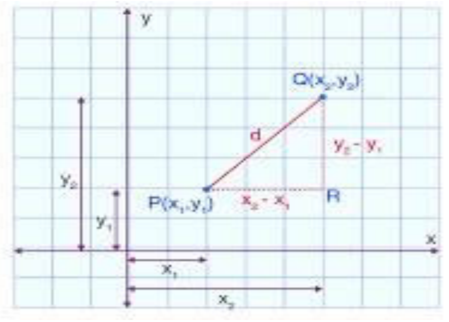
a -> intercept (how low or high the line is)

b -> gradient (slope/angle of the line)

ϵ -> residual (error term) is a difference between the actual value y and the predicted value of y

Our model builds different prediction/regression lines, for each line it sums up the residuals and tries to find the line with the minimal error.

Euclidian distance



ED is a distance between 2 points in a multidimensional space:

distance between points

2D: $\sqrt{(x_1-x_0)^2+(y_1-y_0)^2}$

3D: $\sqrt{(x_1-x_0)^2+(y_1-y_0)^2+(z_1-z_0)^2}$

4D: $\sqrt{(x_1-x_0)^2+(y_1-y_0)^2+(z_1-z_0)^2+(a_1-a_0)^2}$

nD: $\sqrt{(x_1-x_0)^2+(y_1-y_0)^2+(z_1-z_0)^2+(a_1-a_0)^2+\cdots}$

Every axis presents one parameter/feature, e.g. age, income, years of experience, number of children, etc.

Short version for any dimension:

$$distance = \sqrt{\sum_{i=0}^{n}(x_i - y_i)^2}$$

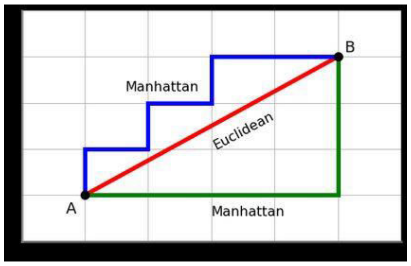
Euclidean distance works great with **low-dimensional data**. KNN and HDBSCAN show great results out of the box if Euclidean distance is used on low-dimensional data. **Normalize/scale** the data before using this distance measure.

From "A Few Useful Things to Know about Machine Learning" by Pedro Domingos at the University of Washington: “In high dimensions, the ratio between the nearest and farthest points approaches 1, the points essentially become uniformly distant from each other. “

For nearest neighbours search, "closer" points are more relevant than "farther" points, but if all points are uniformly distant from each other, the distinction is meaningless.

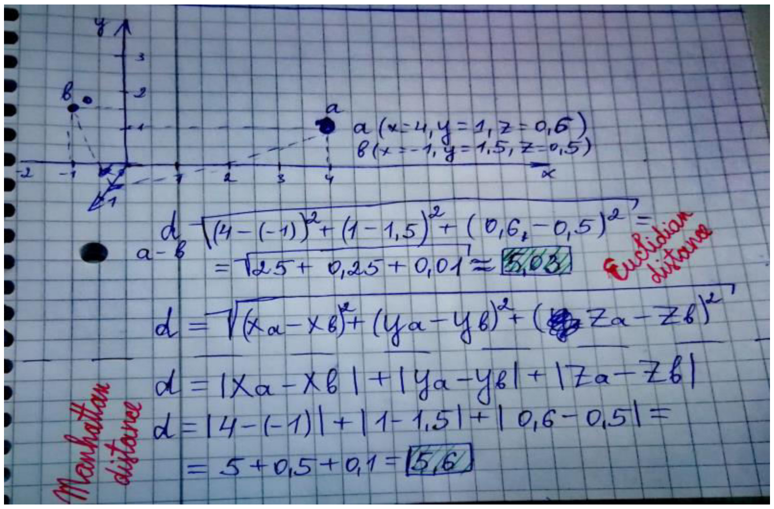
Manhattan distance

MD is a distance between 2 points in a space:



$$\text{Mdist} = |x2 - x1| + |y2 - y1|$$

Manhattan distance is calculated as **the sum of the absolute differences between the two vectors**.



$a(x=4, y=1, z=0{,}6)$
$b(x=-1, y=1{,}5, z=0{,}5)$

$$d_{a-b} = \sqrt{(4-(-1))^2 + (1-1{,}5)^2 + (0{,}6-0{,}5)^2} = \sqrt{25 + 0{,}25 + 0{,}01} \approx \boxed{5{,}03}$$

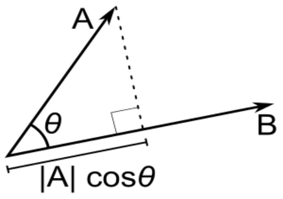
$$d = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2 + (z_a - z_b)^2}$$

$$d = |x_a - x_b| + |y_a - y_b| + |z_a - z_b|$$

$$d = |4-(-1)| + |1-1{,}5| + |0{,}6-0{,}5| = 5 + 0{,}5 + 0{,}1 = \boxed{5{,}6}$$

Cosine similarity

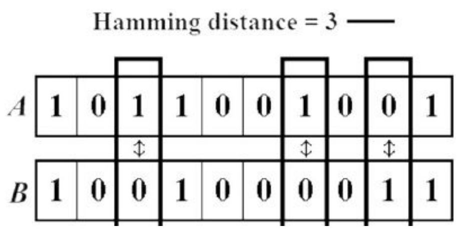
The cosine similarity is simply the cosine of the angle between two vectors. The greater the angle the smaller the similarity (cos 0° = 1, cos 90° = 0). If θ is 10°, cos 10° = 0,985, so the similarity is approximately 99%, if θ is 85°, cos 85° = 0,087, so the similarity is app. 9%.



Cosine similarity is useful for a multidimensional data.

One main disadvantage of cosine similarity is that the magnitudes of the vectors are not taken into account, but only their directions.

Hamming distance



Hamming distance is the number of values that are different between two vectors. It is used to compare two binary strings of **equal length**. Hamming distance suits well to measure the distance between categorical variables.

Chebyshev distance

The Chebyshev distance **measures distance between two points as the maximum difference over any of their axis values**. In a 2D grid, for instance, if we have two points (x1, y1), and (x2, y2), the Chebyshev distance between is max(|y2 - y1|, |x2 - x1|).

KNN Clustering

Clustering

1. Step in the process of clustering is to ensure that our results will not get skewed due to outliers or different systems of measurement.
   a) check your data for outliers;

```
df.describe()
```

   - Check the difference between the mins and means, maxes and means. If they are drastically different it is a sign of outliers.

   - Plot the data, this will help to detect outliers.

```
df.column.hist(figsize=(20,10))
```

   b) check the scales of your data, if they differ, transform the data by scaling

Scaling

Due to different measurement systems one feature may have a greater influence on distances between data points, and by this skew the further results. This is why it make sense to bring all variables to one scale.

Scaling is the process that aligns our data within one system of measurements.

There are different scalers. Scalers do not change the distribution of the data, they change the range of the data. Range = Max – Min. If our data have outliers, then we are to utilize median based scaling. (Medians are less sensitive to outliers than means.)

ScikitLarn returns data as Numpy arrays, so we need to convert them back to a Data Frame.

MinMax Scaler

newValue = (oldValue-min(column))/(max(column)-min(column))

By default, all columns get scaled between 0 and 1. The lowest value of a column becomes 0, the highest value of a column becomes 1, and all other values get scaled in between.

# Import MinMax Scaler

```
from sklearn.preprocessing import MinMaxScaler
```

# Create DataFrame of MinMax scaled values

```
min_max_df = MinMaxScaler().fit_transform(df)
```
#In brackets we can pass the max and min value of the range we want to transform our data to (*MinMaxScaler*(feature_range=(0,100)))

```
with_age_minmax = pd.DataFrame(min_max_df, columns=df.columns, in
dex=df.index)
```

Robust scaler

Data are scaled between their interquartile ranges. The lowest value equals the first quartile, the highest value equals the third quartile, and all other values get scaled in between. In addition, the data get centred around 0 as the median. The median becomes 0 and, all the values less than the median will be negative, and all values higher than the median will be positive. This scaler is robust/unsensitive to outliers.

newValue = (oldValue-median(column))/IQR(column)

# Import Robust Scaler

```
from sklearn.preprocessing import RobustScaler
```

# Create DataFrame of Robust scaled values

#longer version

```
robust = RobustScaler()
robust.fit(df) # calc --> median and IQR
robust_df = robust.transform(df) #apply the formula
```

#shorter version

```
robust_df = RobustScaler().fit_transform(df)
with_age_robust = pd.DataFrame(robust_df, columns=df.columns, ind
ex=df.index)
```

2. Second step is to calculate the distances between all the variable points.

Calculating distances

```
from sklearn.metrics import pairwise_distances
```

# Euclidean (by default)

```
pd.DataFrame(pairwise_distances(df), index=df.index, columns=df.i
ndex)
```

# Manhattan

```
pd.DataFrame(pairwise_distances(df, metric='manhattan'), index=df
.index, columns=df.index)
```

K-Means proceeds like this:

1. Pick k random points as “cluster center”
2. Assign each data point to its closest cluster center.

3. Recompute cluster centers as the mean of the assigned points.

4. Repeat steps 2 and 3 until converge (converge means the assignment of the points to the clusters doesn't change anymore).

```
from sklearn.cluster import KMeans
from sklearn.cluster import KMeans
km = KMeans(n_clusters = 3)
km.fit(X)
```

K-Nearest Neighbours Algorithm is used to make classifications or predictions about the grouping of an individual data point.

1. **Preparation**

a) Cleaning the data

- delete duplicates

- deal the missing values

- drop or convert (e.g. to index) categorical columns

b) Scaling the data (if the ranges of the data differ, the analysis may get skewed towards the metrics with a bigger range)

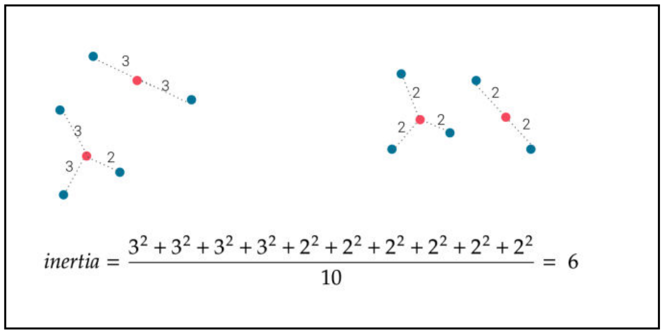
MinMaxScaler –> minimal value in a set becomes 0, the maximum – 1, all other values are spread between 0 and 1.

Standard Scaler –> the mean of the dataset becomes 0, all other data are spread respectively to whether they are bigger or smaller than the mean.

Robust Scaler –> subtracts the median and then dividing by the interquartile range (75% value — 25% value).

2. **Select K-value** – the number of clusters:

a) Inertia – the mean of all distances between all points and their centroids being squared. In other words, we sum up all the distances (between a point and its centroid) being squared and divided by the number of points.



$$inertia = \frac{3^2+3^2+3^2+3^2+2^2+2^2+2^2+2^2+2^2+2^2}{10} = 6$$

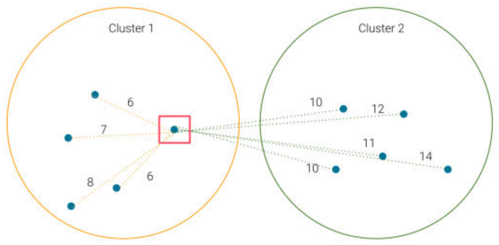
Lower inertia = better clustering (our clusters are cohesive(сплоченные)). More clusters = lower inertia. When all points of a cluster are on one other, the inertia = 1.

b) The Silhouette coefficient – for every point in a dataset we calculate:

a is the mean distance to the other instances in the same cluster

b is the mean distance to the instances of the next closest cluster

With these two values, we can compute the silhouette coefficient for each and every observation of the dataset as *(b-a) / max(a, b)*.



$$a = \frac{6+7+8+6}{4} = 6.75$$

$$b = \frac{10+12+11+14+10}{5} = 11.4$$

$$silhouette = \frac{(11.4 - 6.75)}{11.4} = 0.41$$

If we repeat this process for every observation to get all 10 silhouette scores, and then take the mean of all 10 values, we will have the silhouette score for a KMeans with a K of 2. The silhouette coefficient can vary between -1 and +1: a coefficient close to +1 means that the instance is well inside its own cluster and far from other clusters, while a coefficient close to 0 means that it is close to a cluster boundary, and finally a coefficient close to -1 means that the instance may have been assigned to the wrong cluster.

<u>Elbow</u> – the point where the decline stops being sharp and becomes smooth and gradual. The point of elbow is the optimal number of clusters.

When there is no elbow, but a slope, then increase the number of clusters.

3. Select random distinct data points as <u>**centroids**</u> of initial clusters.
4. Measure the distance between a point and the centroids and assign the point to the closest cluster.
5. Find the mean of each cluster, it becomes a new centroid of the cluster.
6. Recluster on the base of the means-centroids.
7. Repeat steps 4-6 until convergence.

   Convergence – (совпадение) a situation, when means do not change their position.
8. Analyse every cluster.