The k-means Algorithm

From data to insight

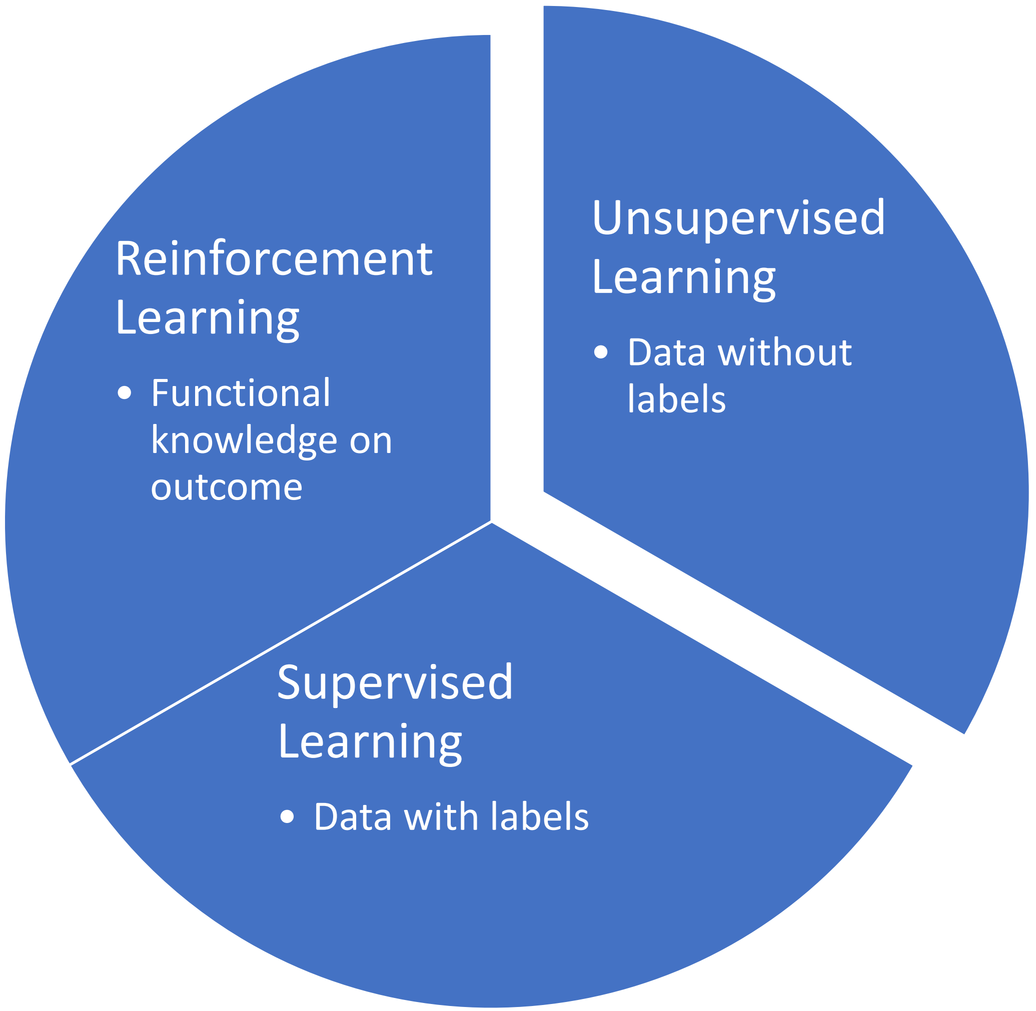
Dr. Jochen Gerhard

Overview

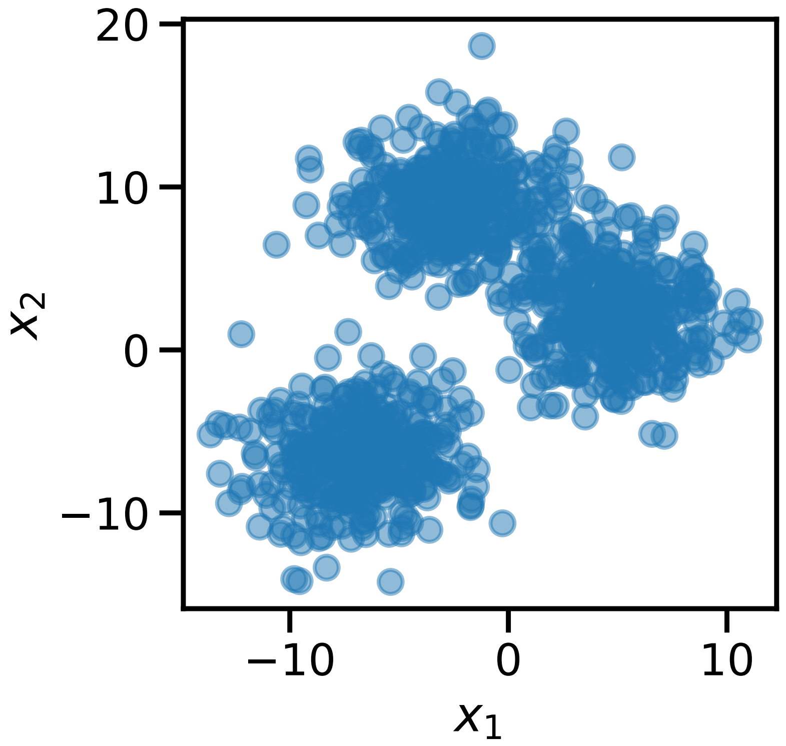
- Why k-means and what is it for?
- How does it work?
- Where to pay attention when using it?
- Where to look next?

K-means reveals structure

- Take a data set with numerical features as input
- No additional information (target, output) given.
- K-means shows where the data points „stick together“
  - These areas of data sticking together are called „clusters“



We start with data and many questions…



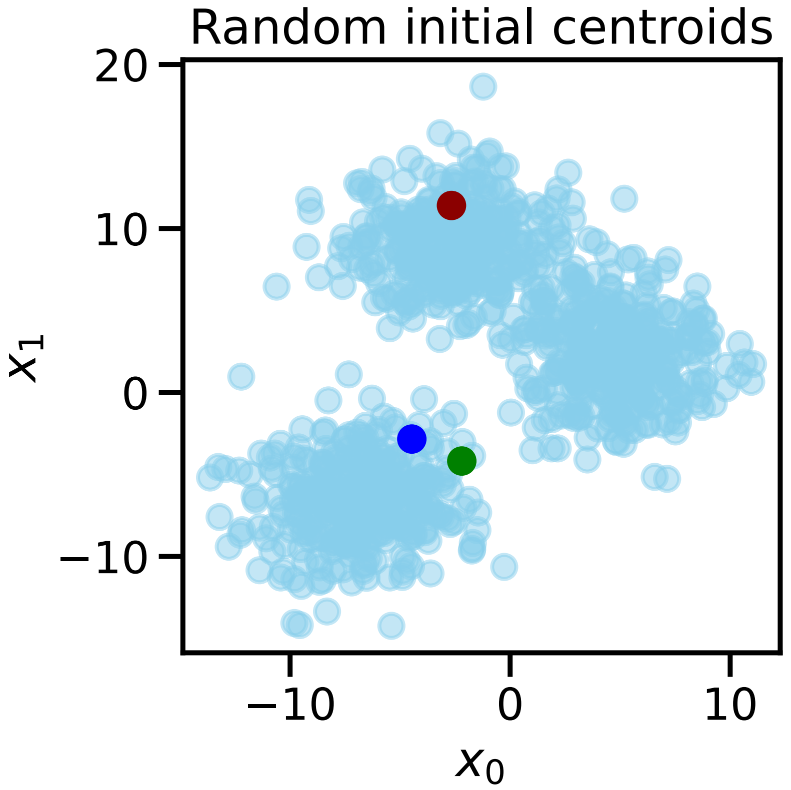
- We are given data in two dimensions:
    - This could be anything like temperature and humidity up to square meters and rent price.
- Without any additional information, what insight can we gather.
- Eyeballing in 2d reveals data lumps together…
- Generally: Number of features is the dimension of our space

The Algorithm (full)

```
centroids = initialise_centroids(points, k)
WHILE ((i < maxiter) & (delta_loss > eps)):
        forall point in points:
                forall centroid in centroids:
                        calculate_distance(point, centroid)
                my_centroid = select_nearest_centroid(point)
                belonging_points[my_centroid].append(point)
        forall centroid in centroids:
                update_centroid(belonging_points[centroid])
        delta_loss = calculate_delta_loss(points, centroids)
        i++
```

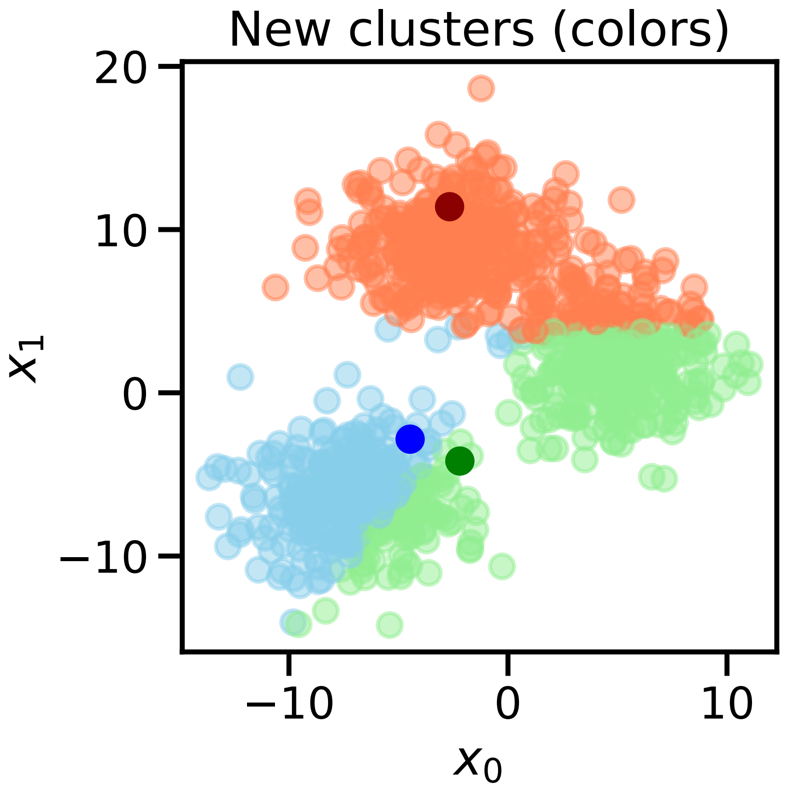
The Algorithm (initialise)

```
centroids = initialise_centroids(points, k)
```



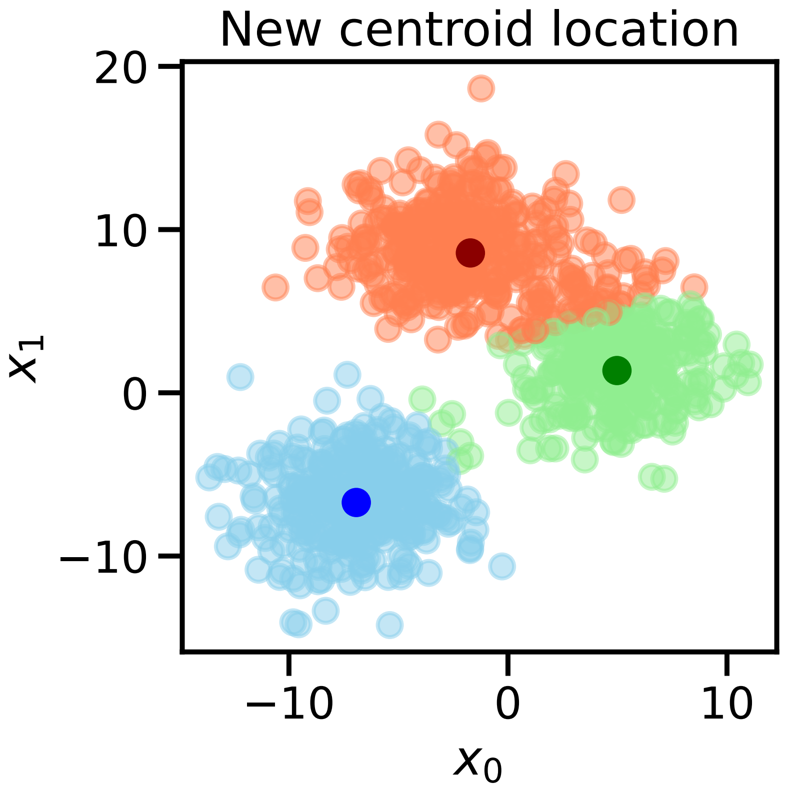
The Algorithm (assign point to centroid)

```
centroids = initialise_centroids(points, k)
WHILE ((i < maxiter) & (delta_loss > eps)):
        forall point in points:
                forall centroid in centroids:
                        calculate_distance(point, centroid)
                my_centroid = select_nearest_centroid(point)
                belonging_points[my_centroid].append(point)
```



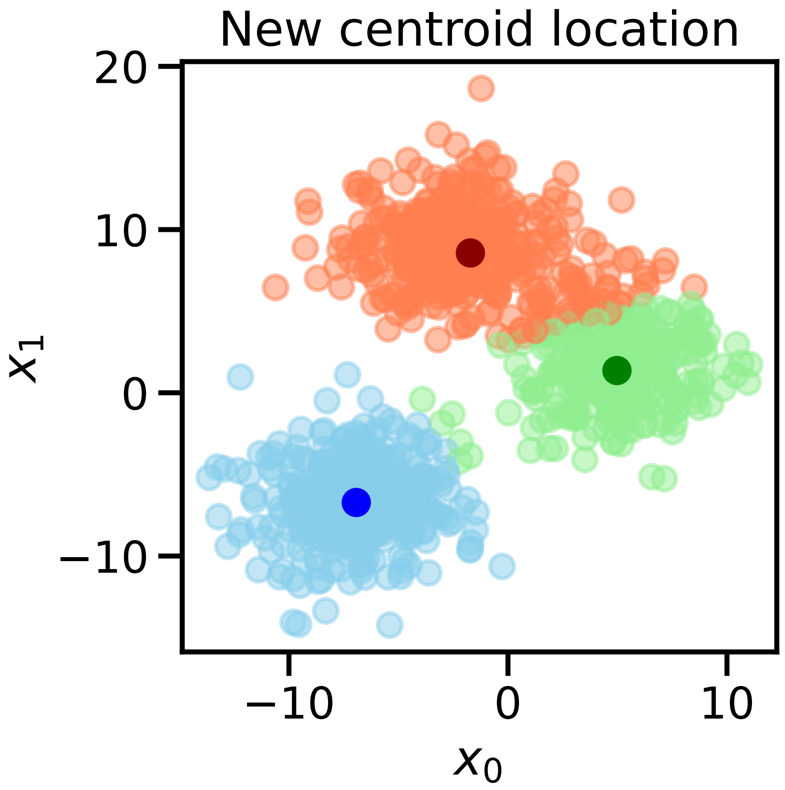
The Algorithm (assign point to centroid)

```
centroids = initialise_centroids(points, k)
WHILE ((i < maxiter) & (delta_loss > eps)):
        forall point in points:
                forall centroid in centroids:
                        calculate_distance(point, centroid)
                my_centroid = select_nearest_centroid(point)
                belonging_points[my_centroid].append(point)
        forall centroid in centroids:
                update_centroid(belonging_points[centroid])
```

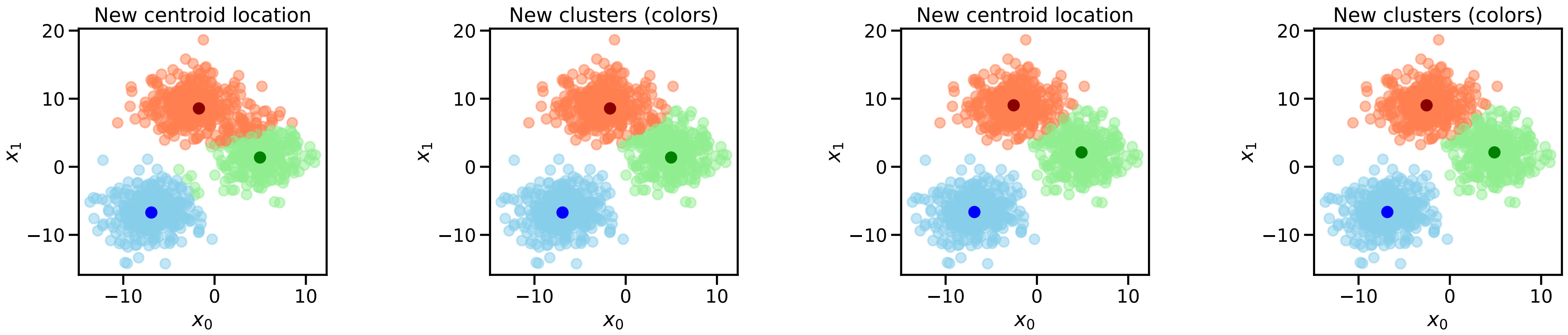


The Algorithm (prepare next iteration)

```
centroids = initialise_centroids(points, k)
WHILE ((i < maxiter) & (delta_loss > eps)):
        forall point in points:
                forall centroid in centroids:
                        calculate_distance(point, centroid)
                my_centroid = select_nearest_centroid(point)
                belonging_points[my_centroid].append(point)
        forall centroid in centroids:
                update_centroid(belonging_points[centroid])
        delta_loss = calculate_delta_loss(points, centroids)
        i++
```



Next steps…


New centroid location
New clusters (colors)
New centroid location
New clusters (colors)
20
10
0
−10
−10
0
10
x_0
x_1

Have a look into the python example

```python
i = 0
old_loss = np.inf
delta_loss = np.inf

while ((i < maxiter) & (delta_loss > eps)):

    # Calculate distance of all points to the all centroids
    for j, c in enumerate(centroids):
        distances[:, j] = get_distances(c, X)

    # Determine cluster membership of each point
    # by picking the closest centroid
    clusters = np.argmin(distances, axis=1)

    # Update centroid location using the newly
    # assigned data point cluster
    for c in range(k):
        centroids[c] = np.mean(X[clusters == c], 0)

    # For loss criterion calculate sum of squared distances to cluster centroid

    loss = calculate_loss(centroids, clusters)
    delta_loss = np.abs(old_loss - loss)
    old_loss = loss
    i = i+1
```

Calculations: distance and loss

- In classic k-means algorithm we use the euclidian distance
- For each point x and cluster with centroid C in n-dimensions:

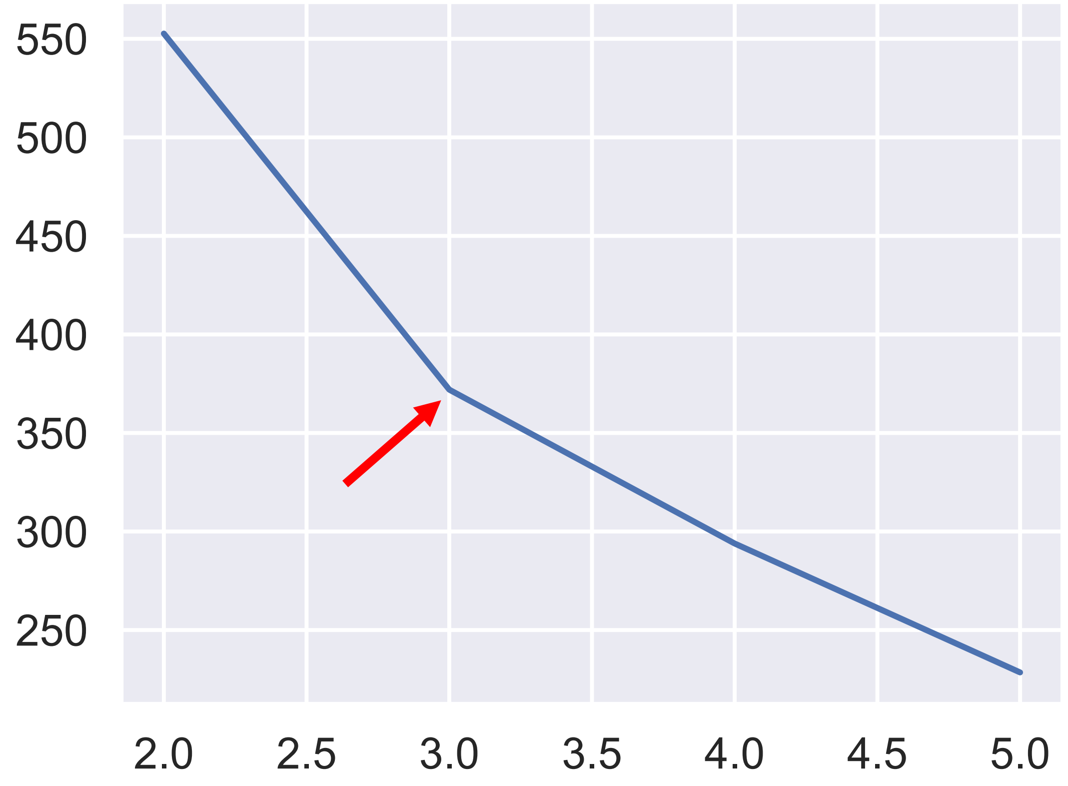
$$d(x, C) = \sqrt{\sum_{i=1}^{n} (x_i - C_i)^2}$$

- The loss term is often called „inertia“ or „WCSS“ (within-cluster-sum-of-squares)
- It is the same (up to factor) as the sum over all clusters with centroids C_i of the variances within each cluster:

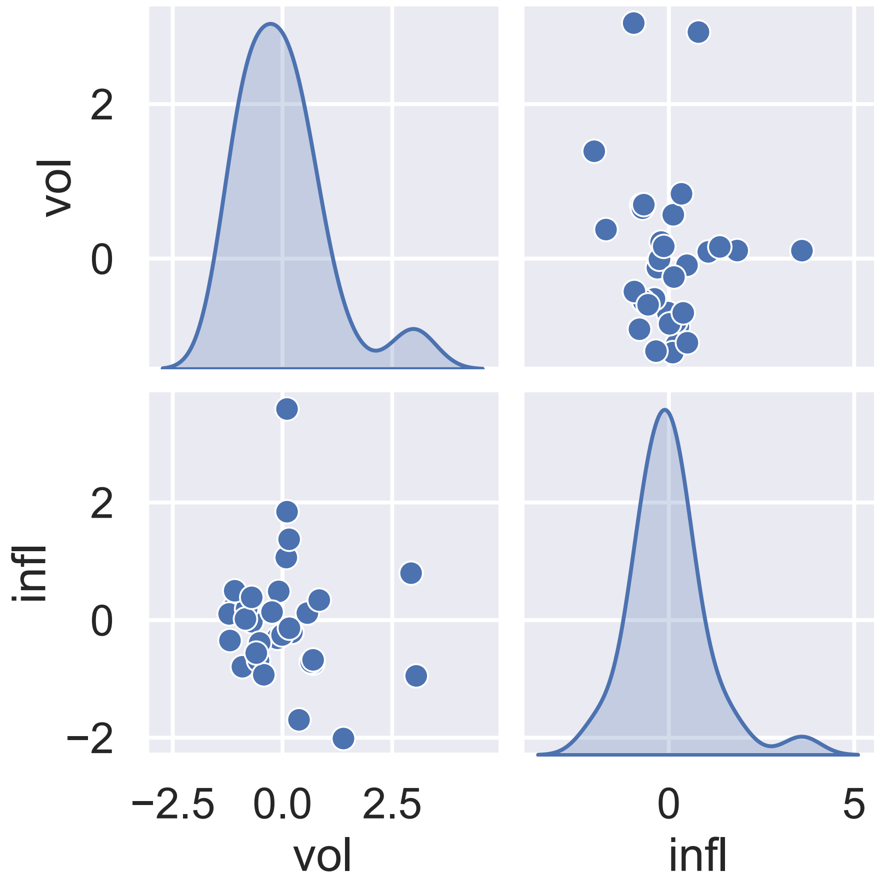
$$\sum_{C} \sum_{p \in C} d(p, C)^2$$

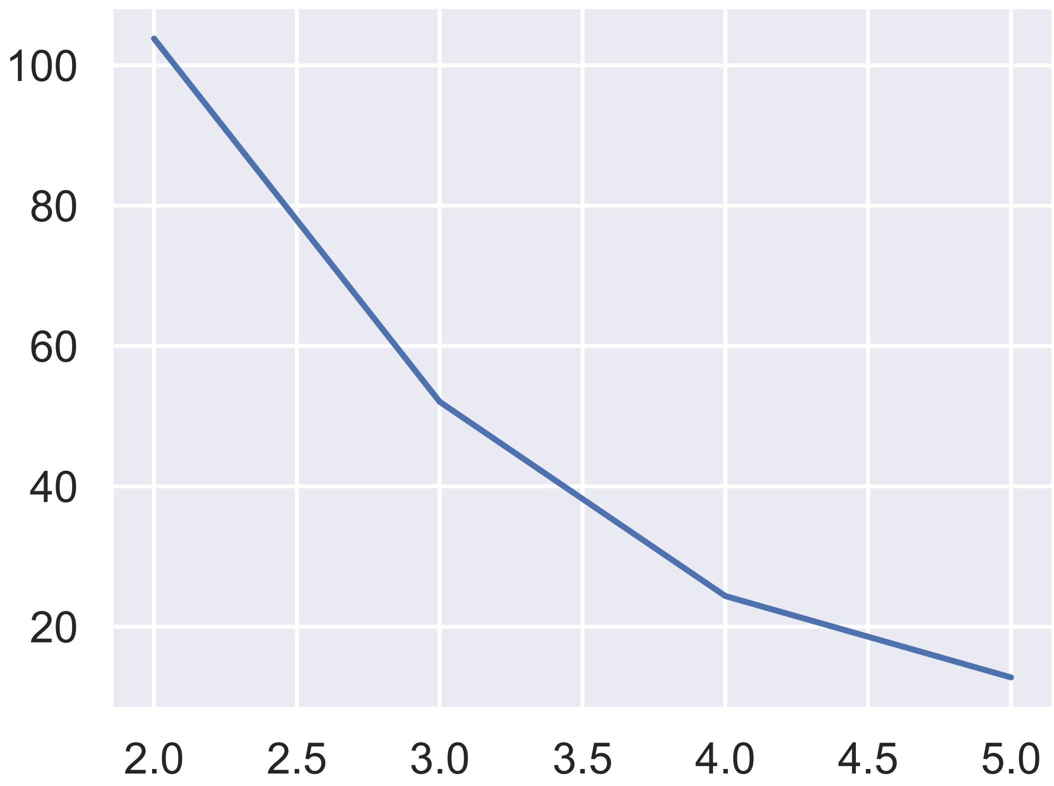
The one parameter: number of clusters k

- Has to be set before starting the loop.
- Can be derived from additional (business, science) insights.
- If totally unknown, there's the elbow-heuristic:
  - Try for consecutive k, save the loss after convergence
  - Plot losses depending on k
  - Pick one that is „the elbow"



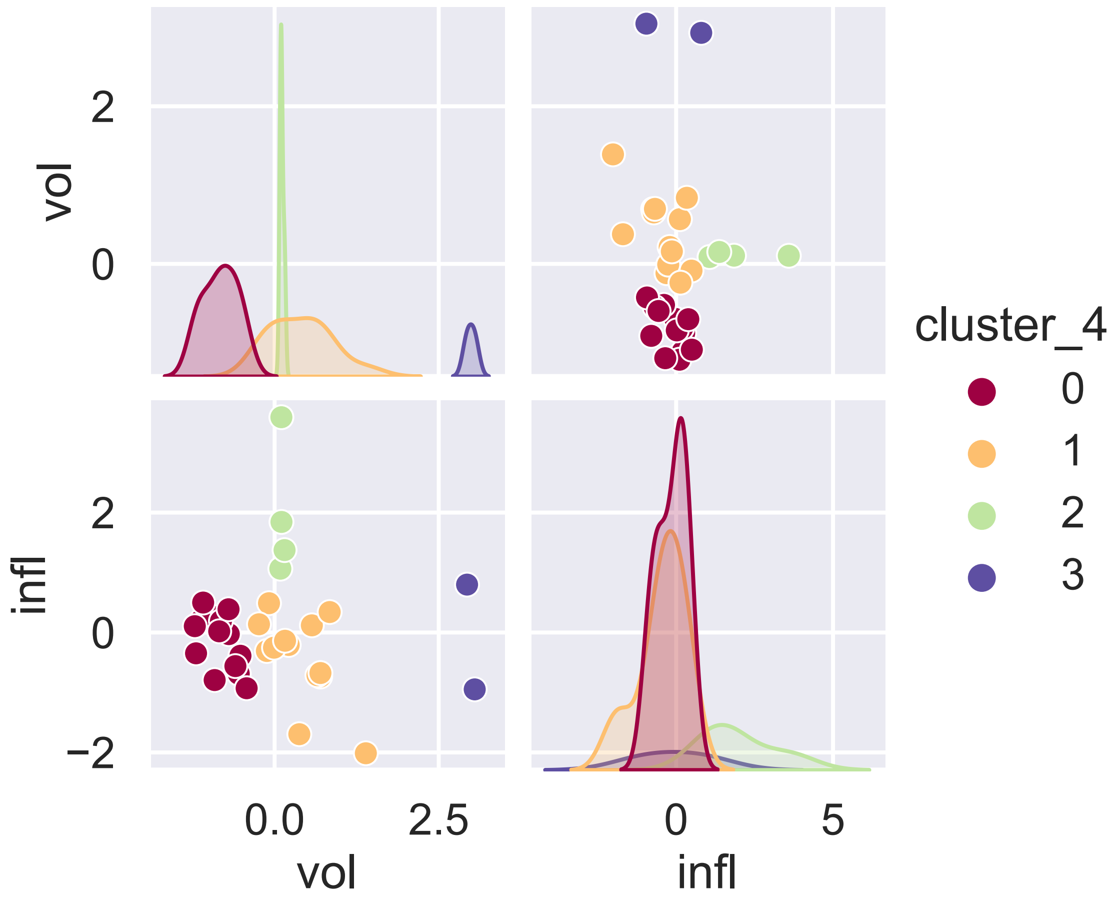
Preparing your Hedge Fond



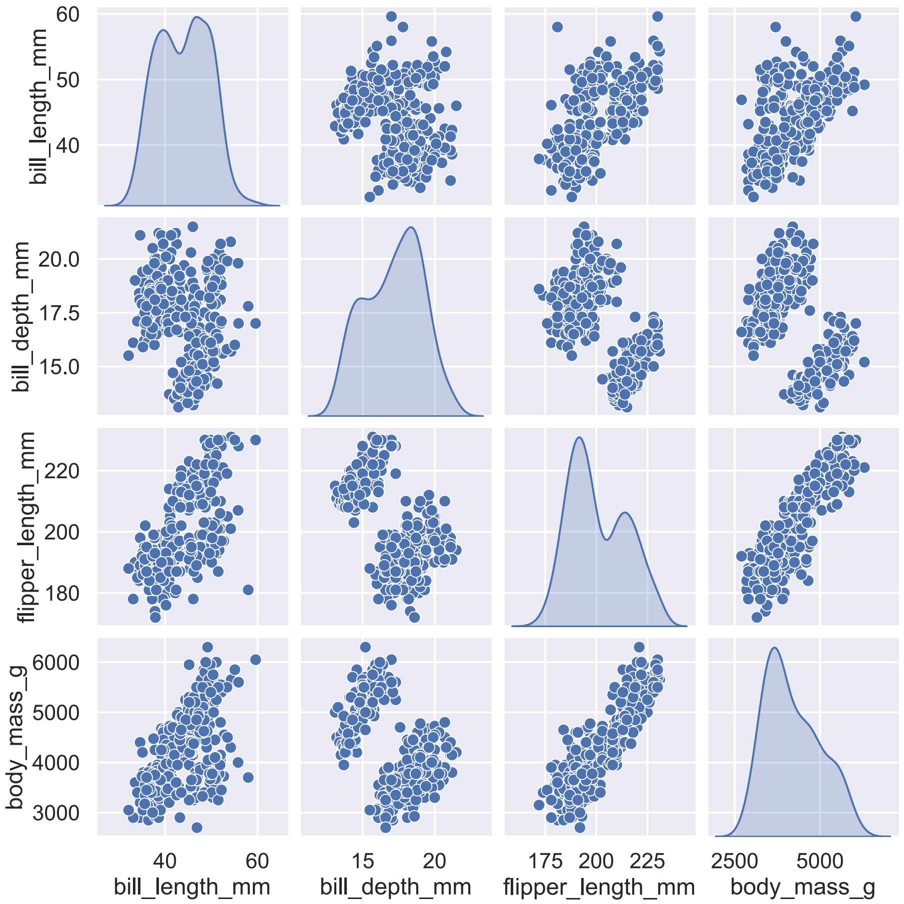


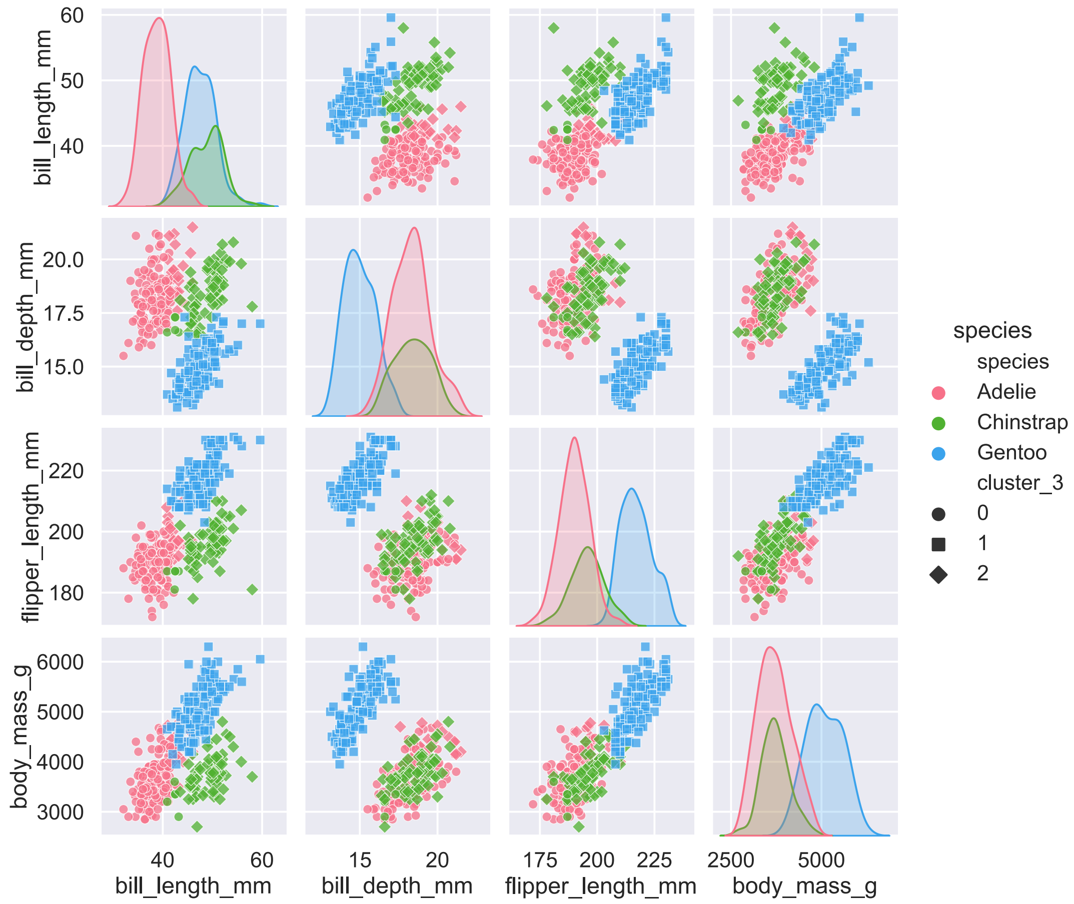
We choose k=4 based on our business Idea

- The elbow heuristic is useful, but if you have an idea how many clusters would be useful, go for it!
- Taking macro economic variables for regime prediction is often part of trading strategies…



Re-discovering penguin species

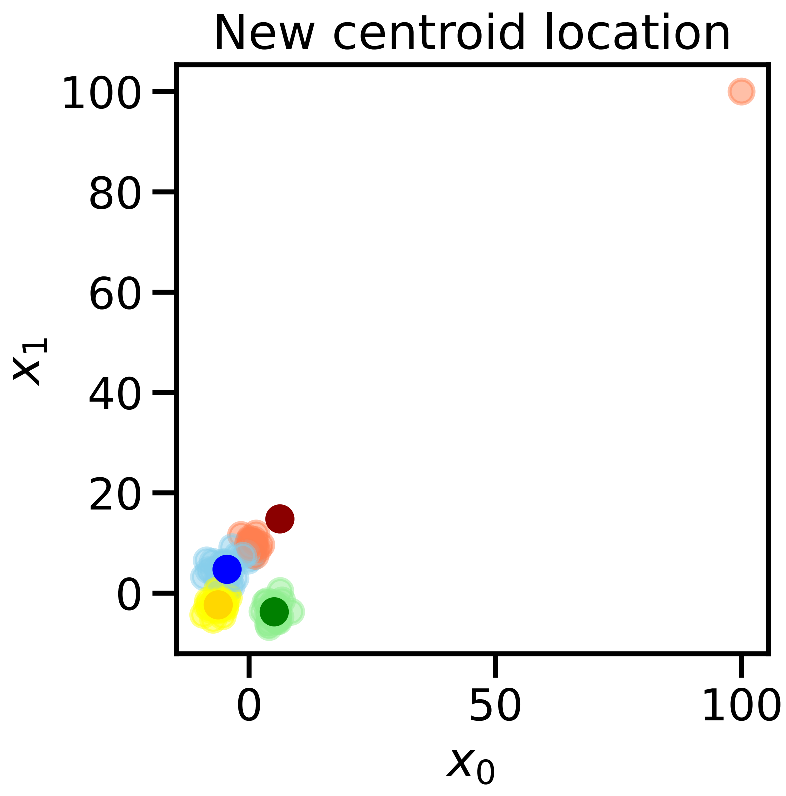


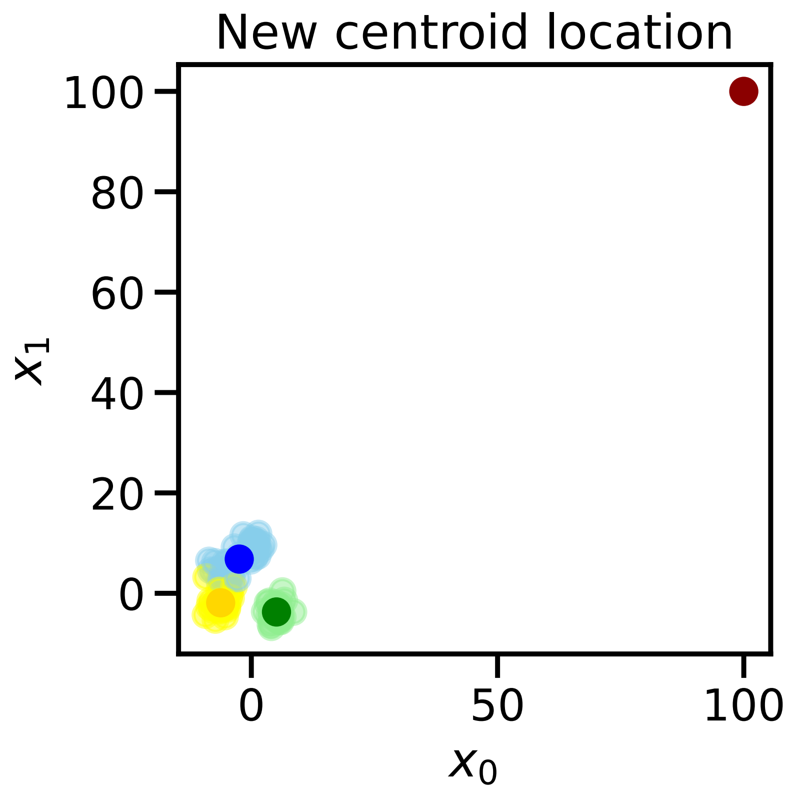


Caveats:

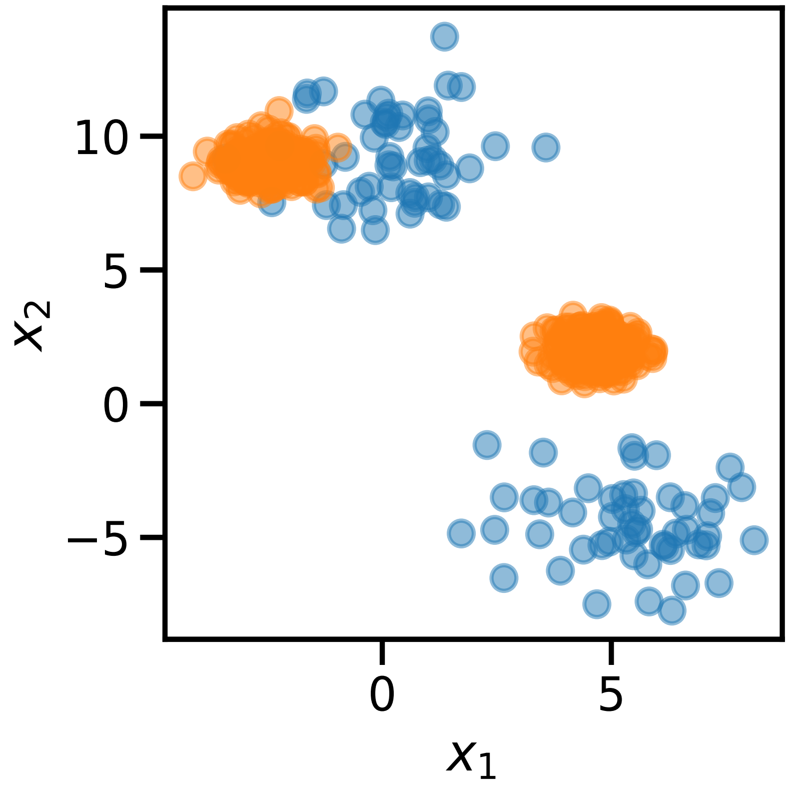
- The algorithm works on features where calculating a mean makes sense – not on categorical data.
- The distance function ignores the scales of dimensions. Make sure the scales are what you want (it‘s a way, to weigh features!) or standardise.
- If you have severe outliers, the algorithm goes astray – Windsorisation (cutting off!) can help.
- Initializing the algorithm with random centroids makes it non deterministic – local minima can be compensated for by re-runs.
- The Algorithm works well, if data are in similar blobs.

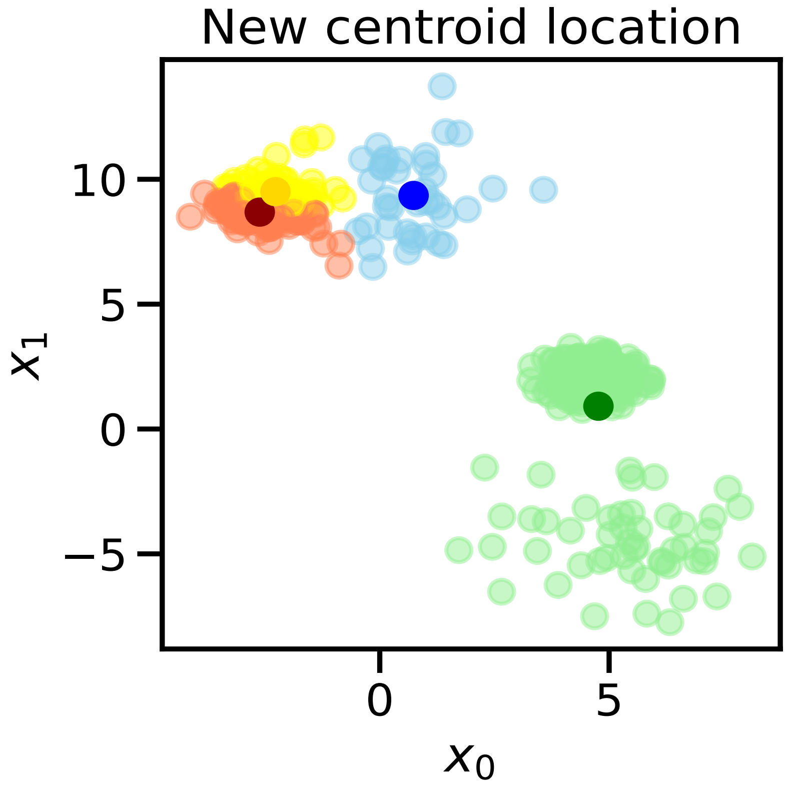
Outliers


New centroid location
100
80
60
40
20
0
0
50
100
x_0
x_1

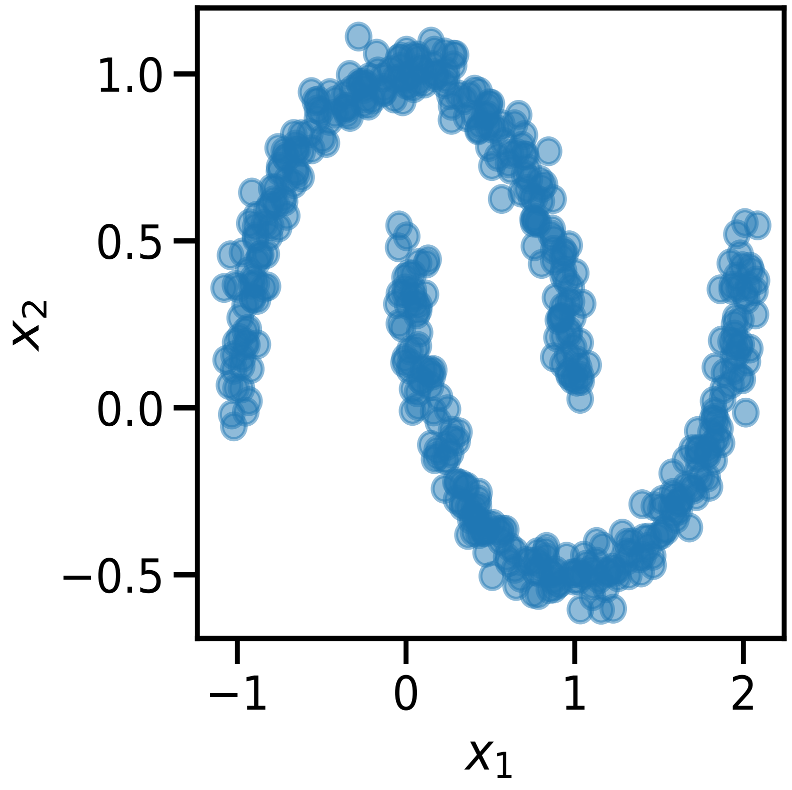

New centroid location
100
80
60
40
20
0
0
50
100
x_0
x_1

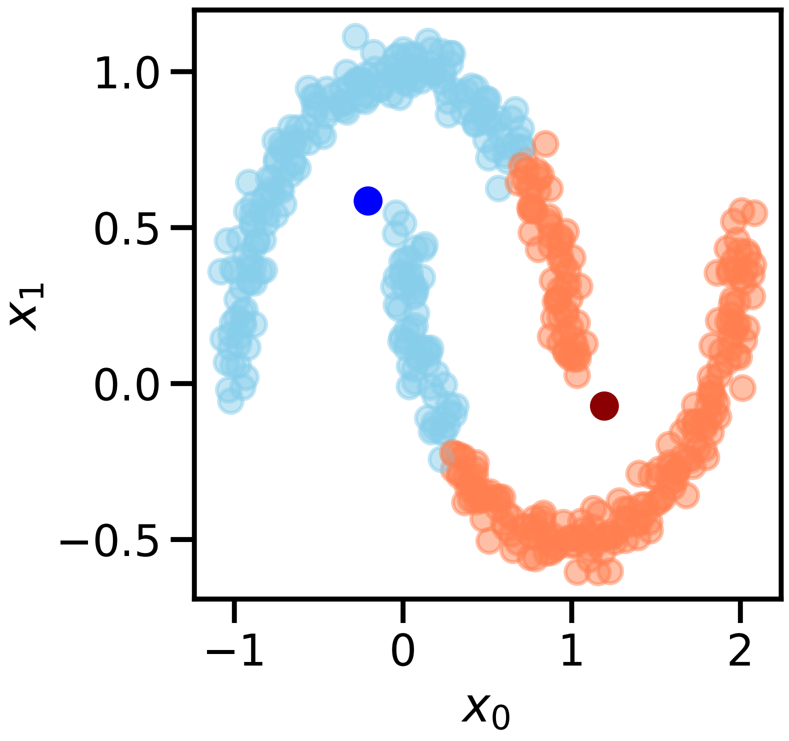
Not the same variance in blobs





Not even blobs...





Performance

- The k-means algorithm offers vast possibilities for parallelisation:
  - Using different initial clusters
  - Using different k (when finding a suitable one)
  - Distance of each point to each cluster
  - Calculation per cluster of new centroid

- Together with the simple math, its suited for GPU computing.

Rule of thumb: If calculations are somehow geometric and independent (game like explosions) they are suitable for GPUs.

Transparency

- The algorithm by itself is very transparent: we can see how it iterates to it‘s solution
- The math behind is well understood.

- The non-determinism needs to be carefully watched.
- In the selection of k and in the scaling can be hidden assumptions

Where to look next?

- Feel free to experiment with the python notebooks:
  - https://github.com/jgerhard/k-means
- If you want to try out real life financial data for your experiments:
  - https://fred.stlouisfed.org/
- On Kaggle you can find bunch of data sets and contests:
  - https://kaggle.com/
- If you want to play with basic data science algorithms, Joel Grus “Data Science from Scratch” can be a nice starter.
- If you prefer the next step, Jake VanderPlas’ “Python Data Science Handbook“ is available freely: Data Science Handbook