Assignment 4 - Introduction to Data Science

Ninell Oldenburg

March 18, 2022

Exercise 1 - Plotting cell shapes

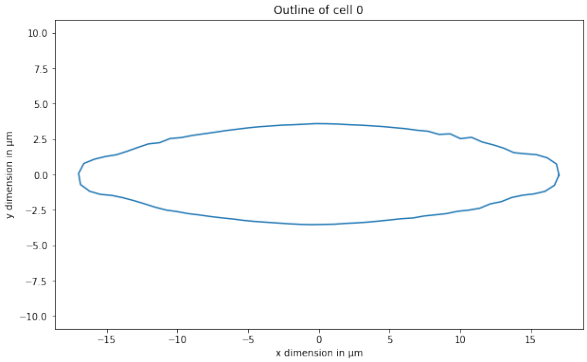

Figure 1: Outline of first cell in data set

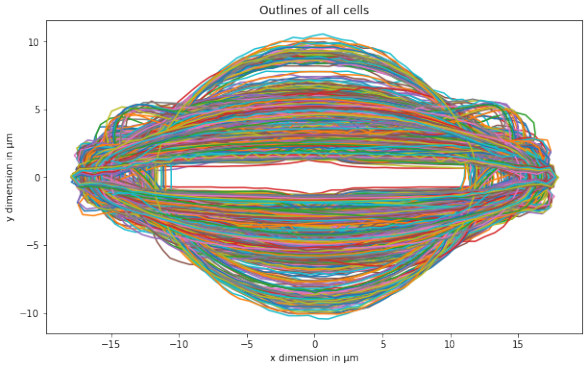

Figure 2: Outlines of all cells in data set

Description In Figure 2 we clearly see two, if we're really detailed even three larger groups of similar cell shapes: either roundish (e.g. like the yellow plotted cell) , stretched along the x-dimension (e.g. as in Figure 1, and the third one with sort of corners at $y = 5$ and either $x = \pm 12.5$.

Exercise 2 - Visualizing Variance

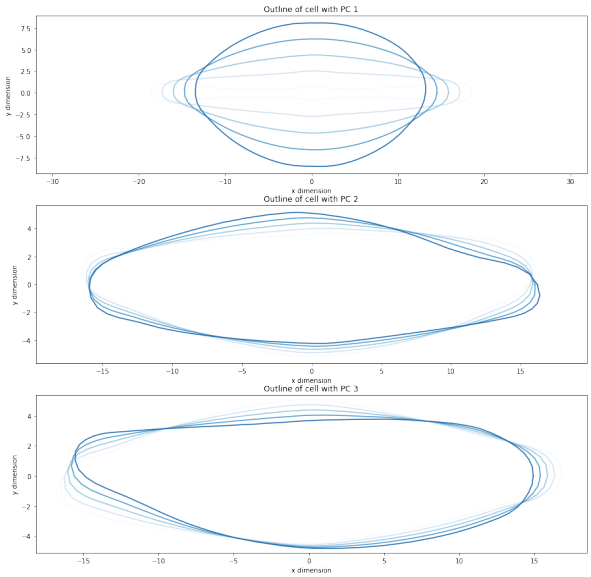

Figure 3: Spatial Variance of the cells by first three PCs

Description The PCs report the variance over a given data set over different dimensions. By multiplying the PCs as suggested by the assignment, we make this variance visible. So what does this mean for our cells?
m is a constant, so it will be the same for each of the three plots. What we does however change are the variance and the values of the eigenvectors. So what we're trying to visualize here is how much each of the PCs that correspond to the changing variance/eigenvectors (in descending order) affect the change from the mean. So as larger the index of the respective PC gets, the smaller the variance get, that is, the less it differs from the mean, because the variance around the eigenvectors is getting smaller (values for eigenvectors for PC1 are larger than the values for the eigenvectors for the following PCs). In the previous assignment, we were ask to calculate exactly how much of this variance can be captured for each of the components. This exercise is the visual addition to this task.
To be specific for each PC, the first PC seems to address the change that we've observed in exercise 1. The spatial variance seems to be stretched along the x-dimension. The more it stretches along the y-axis, the less it stretches along the x-axis, that is, the more round it gets.
However, the second and third PC seem to address the smaller variance over the corner shapes/edge shapes.

Exercise 3 - Critical Thinking

a)

i) **Centering** This shouldn't affect our PCA as we're not changing the variance between the values, so we're neither loosing any information nor creating any bias. The co-variance matrix will still be the same. Even more, centering is even recommend before a PCA in order to not misguide PC1 through the origin.

ii) **Standardization** The benefit of this is that we can compare our values better according to our different variables. If we don't do the standardization, we could end up entailing the variables with larger on-paper variance in their units are more important than the smaller-unit-variables.

iii) **Whitening** If we apply whitening to our data before we do PCA, we would erase the variance and we would end up loosing a lot, if not all the information. So that's why we shouldn't do it.

b)

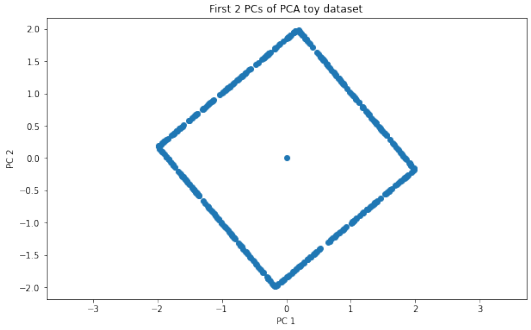

Figure 4: First 2 PCs of complete PCA toy data set

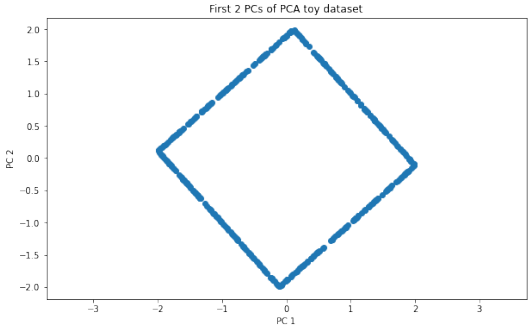

Figure 5: First 2 PCs of PCA toy data set without last 2 data points

Explanation By removing the last two points, we seem to remove the visual center of the toy data. The form of the rest of the data is still a square, yet slightly rotated, which means that there might have been more points on one of the sides, as the center itself probably didn't affect the rotation.

Exercise 4 - Clustering II

Description of the software. The multidimensional scaling function is taken from my assignment 3 and takes as input a data set and the number of desired output dimensions. It works through performing a PCA on the data, slicing the eigenvectors in respect to the input value for the dimensions, and outputting the dot product, this is the projection, of the data and its PC eigenvectors.

The PCA function is also taken from assignment 3, takes as input a data set and outputs the eigenvalues, eigenvectors and the mean of the data. I start by first centring the data to stabilize the output PC1, compute the covariance matrix, and output the performed the eigen-decomposition as well as the mean (I decided to just output it at this point as it will often be used from the calling functions).

For the clustering part, I've used the `KMeans` clustering method provided by the `sklearn.cluster` package. Here is how it works:

1. **Create a `KMeans` Class**. Here, we pass the parameters that we want to specialize our `KMeans` cluster with. In general, all of the parameters, e.g. the number of target clusters, the initial cluster centers, the maximum number of iterations to find cluster center, etc. have a default value, so we don't *have* to specify these. In our case, however, we tell the algorithm that we want to have 2 clusters, that it should start at `random_state=42`, we want it to use the `"full"` algorithm (so force it to not optimize on the more memory intense algorithm if the data clusters end up being well defined), that we only want to have one iteration with different center seeds (which makes sense as we are initializing the centers ourselves and increasing that number wouldn't change the output), and finally, pass the two center seeds that we've defined before as suggested by the assignment.

   The algorithm then returns a K-means class that is tuned with these parameters and is ready to fit our data.

2. **Fit our data**. This is done in several stages.

   (a) **Choose initial cluster centers**. That can be done in several ways. 1) Centers are being initialized by the calling function (via the `init=` parameter). That's how I did it in the last assignment. In sake of robustness and because we already give the function a `random_state` here (explained in the following), we leave it out for our purposes. The other way is the 2) the random or semi-random initialization where the centers are either being assigned completely random or semi-random by also selecting a `random_state` to make the results reproducible. The number of initial cluster centers corresponds of course always to the number of cluster centers we want to have and pass through the calling function (`n_clusters` parameter).

(b) **Assign data points to closest centers**. Assign every data point (by which I mean vectors, not every single point!) to their closest center. So in our case every of the 13 points in our 1000 vectors is being assigned to their respective closest points at the respective index. In our case there are only two options.

(c) **Calculate new cluster centers**. Now, that all data points are assigned to either of the clusters, we calculate the mean over all points in the cluster. That's how the initial cluster centers are being drawn more and more to the actual center of the potential clusters.

(d) **Repeat steps (b) and (c)**. We stop when we've either 1) reached the `max_iter` parameter that is 300 per default, or 2) when the cluster center do not change anymore from the one iteration to the next one (that's how we know we've found the best possible mean for the center seeds of this iteration).

(e) **Compare different cluster seeds**. This step is being skipped in our implementation due to us assigning the initial seeds. But usually, if they're picked at random, it makes sense to compare several seeds and see which one is having the best possible outcome. That is, which distribution of variance of the cluster centers has the most equal one over the data. If the variance over all cluster centers is comparably similar, we know that these are the best centers.

3. **Return attributes**. The `KMeans` class that is now tuned and trained on our data has now several attributes that are interesting for us. For this assignment, we're only interested in the `kmeans.cluster_centers_` that gives us, as the name suggests, the centroids of our cluster. In general, however, people like to look at the `kmeans.labels_`, that gives us an array of the size of our data, but instead of the data points/vectors, we get the labels (so the cluster index the point belongs to (in our case either 0 or 1)).

Projection of two cluster centers

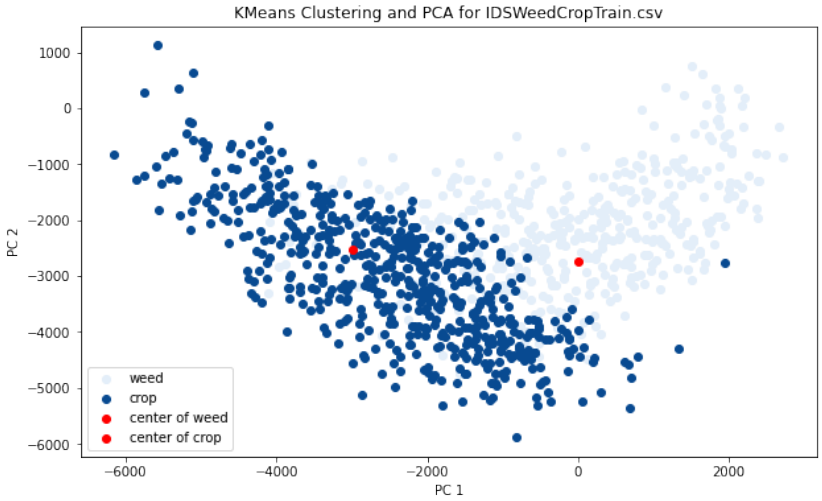

Figure 6: 2D plot with the data points and the cluster centres

```
cluster centers:
[-2993.12258233, -2537.69938273], [5.77619091, -2738.3200362 ]
```

Discussion The two found centers seem to reflect the centers of the two cluster somehow well. Both red dots seem to lie somewhat in the middle of the data, which I think is surprising taking into account that these points are the centers of the cluster when we disregard the actual labels, but rather taking the centers from the clustering approach for the already scaled data. This might tell us, that our data is very good clusterable (if this is a word).

Exercise 5 - Clustering III

Description of the software. The multidimensional scaling function is taken from my assignment 3 and takes as input a data set and the number of desired output dimensions. It works through performing a PCA on the data, slicing the eigenvectors in respect to the input value for the dimensions, and outputting the dot product, this is the projection, of the data and its PC eigenvectors.

The PCA function is also taken from assignment 3, takes as input a data set and outputs the eigenvalues, eigenvectors and the mean of the data. I start by first centring the data to stabilize the output PC1, compute the covariance matrix, and output the performed the eigen-decomposition as well as the mean (I decided to just output it at this point as it will often be used from the calling functions).

For the clustering part, I've used the `KMeans` clustering method provided by the `sklearn.cluster` package. Here is how it works:

1. **Create a `KMeans` Class**. Here, we pass the parameters that we want to specialize our `KMeans` cluster with. In general, all of the parameters, e.g. the number of target clusters, the initial cluster centers, the maximum number of iterations to find cluster center, etc. have a default value, so we don't *have* to specify these. In our case, however, we tell the algorithm that we want to have 2 clusters, that it should start at `random_state=42`, we want it to use the `"full"` algorithm (so force it to not optimize on the more memory intense algorithm if the data clusters end up being well defined), that we only want to have one iteration with different center seeds (which makes sense as we are initializing the centers ourselves and increasing that number wouldn't change the output), and finally, pass the two center seeds that we've defined before as suggested by the assignment.

   The algorithm then returns a K-means class that is tuned with these parameters and is ready to fit our data.

2. **Fit our data**. This is done in several stages.

   (a) **Choose initial cluster centers**. That can be done in several ways. 1) Centers are being initialized by the calling function (via the `init=` parameter). That's how I did it in the last assignment. In sake of robustness and because we already give the function a `random_state` here (explained in the following), we leave it out for our purposes. The other way is the 2) the random or semi-random initialization where the centers are either being assigned completely random or semi-random by also selecting a `random_state` to make the results reproducible. The number of initial cluster centers corresponds of course always to the number of cluster centers we want to have and pass through the calling function (`n_clusters` parameter).

   (b) **Assign data points to closest centers**. Assign every data point (by which I mean vectors, not every single point!) to their closest center. So in our case every of the 13 points in our 1000 vectors is being assigned to their respective closest points at the respective index. In our case there are only two options.

   (c) **Calculate new cluster centers**. Now, that all data points are assigned to either of the clusters, we calculate the mean over all points in the cluster. That's how the initial cluster centers are being drawn more and more to the actual center of the potential clusters.

   (d) **Repeat steps (b) and (c)**. We stop when we've either 1) reached the `max_iter` parameter that is 300 per default, or 2) when the cluster center do not change anymore from the one iteration to the next one (that's how we know we've found the best possible mean for the center seeds of this iteration).

   (e) **Compare different cluster seeds**. This step is being skipped in our implementation due to us assigning the initial seeds. But usually, if they're picked at random, it makes sense to compare several seeds and see which one is having the best possible outcome. That is, which distribution of variance of the cluster centers has the most equal one over the data. If the variance over all cluster centers is comparably similar, we know that these are the best centers.

3. **Return attributes**. The `KMeans` class that is now tuned and trained on our data has now several attributes that are interesting for us. For this assignment, we're only interested in the

`kmeans.cluster_centers_` that gives us, as the name suggests, the centroids of our cluster. In general, however, people like to look at the `kmeans.labels_`, that gives us an array of the size of our data, but instead of the data points/vectors, we get the labels (so the cluster index the point belongs to (in our case either 0 or 1)).

Projection of two cluster centers

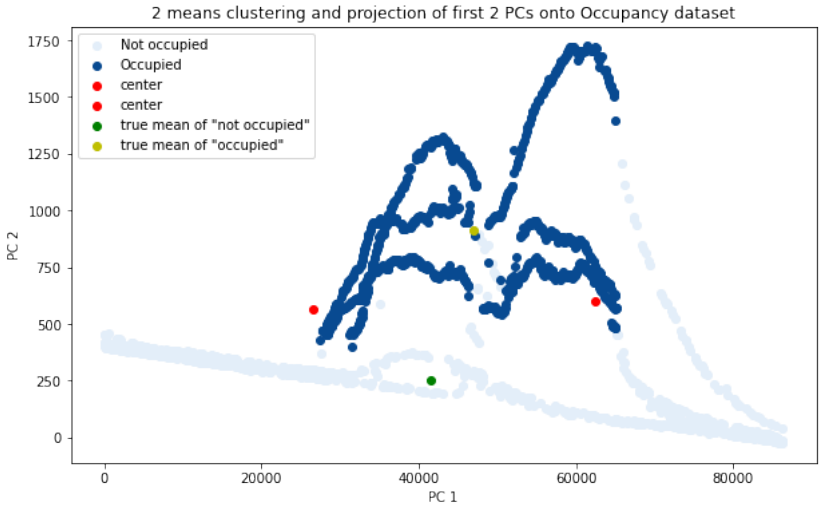

Figure 7: 2-means clustering and projection of first 2 PCs onto Occupancy data set

```
cluster centers:
[26603.53555159, 562.96781155] [62493.68090106, 601.57889495]
```

Discussion The center points seem to do not reflect the visual center points of the data really well. The true labels for the points reflecting the occupied vs. not occupied data seem to cluster more around the upper middle vs. lower range of points (true means in yellow/green). However, the centers of the performed multidimensional scaling reflect a clustering that could take into account the two visual peaks we can observe in the plot. This means, that the occupancy data set's true labels not perfectly cluster as an algorithm would predict it.