Unsupervised Learning

Madiba Hudson-Quansah

Contents

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Chapter 1

The Curse of Dimensionality

High dimensionality makes models generalize worse to lower dimensions and makes it harder to find the best model. This is known as the curse of dimensionality. In short the more dimensions the training set has the greater the risk of overfitting it.

Chapter 2

Dimensionality Reduction

2.1 Projection

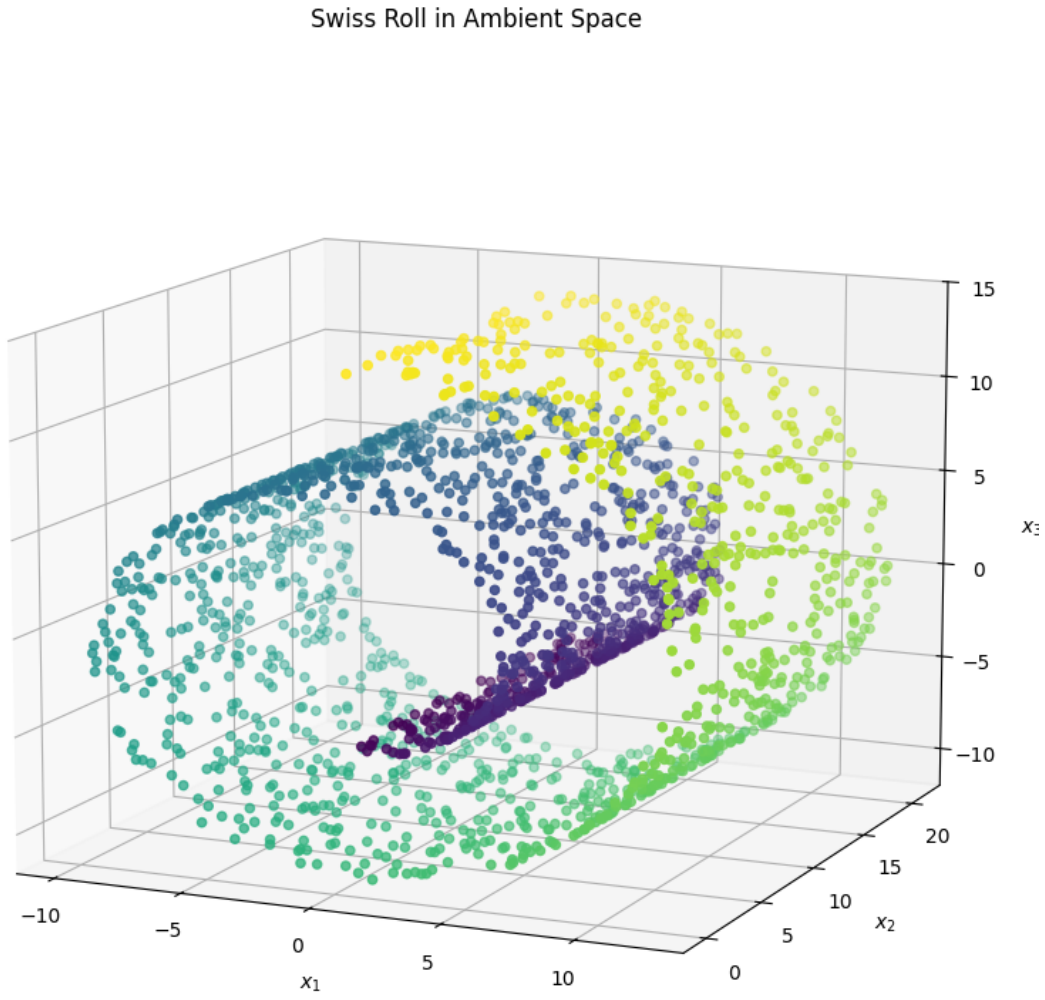
In most real-world problems training instances are not spread out uniformly across all dimensions. Many features are almost constant, while others are highly correlated. As a result, all training instances lie within a much lower-dimensional subspace of the high-dimensional space. Using this fact we can reduce the dimensionality of the training set by projecting it onto a lower-dimensional subspace for example from 3D to 2D, when most data points lie close to a plane,

2.2 Manifold Learning

However in some cases data points may not lie close to a single plane but instead subspaces may twist and turn such as in the famous Swiss roll.

[2]:
```
from sklearn.datasets import make_swiss_roll


def set_xyz_axes(ax, axes):
    ax.xaxis.set_rotate_label(False)
    ax.yaxis.set_rotate_label(False)
    ax.zaxis.set_rotate_label(False)
    ax.set_xlabel("$x_1$", labelpad=8, rotation=0)
    ax.set_ylabel("$x_2$", labelpad=8, rotation=0)
    ax.set_zlabel("$x_3$", labelpad=8, rotation=0)
    ax.set_xlim(axes[0:2])
    ax.set_ylim(axes[2:4])
    ax.set_zlim(axes[4:6])


plt.figure(figsize=(10, 10))
swiss, col = make_swiss_roll(n_samples=2000, random_state=42)
axes = [-11.5, 14, -2, 23, -12, 15]
fig = plt.gcf()
ax = fig.add_subplot(111, projection="3d")
fig.add_axes(ax)
ax.scatter(swiss[:, 0], swiss[:, 1], swiss[:, 2], c=col)
ax.view_init(azim=-66, elev=12)
```

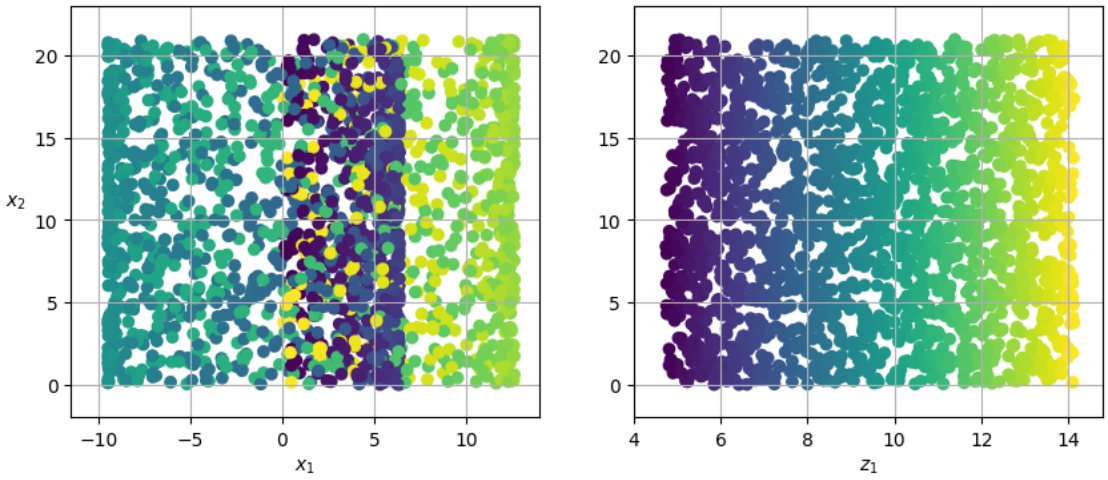
```
ax.set_title("Swiss Roll in Ambient Space")
set_xyz_axes(ax, axes)
```



Simply projecting onto a plane (dropping x_3) would squash different layers of the Swiss roll together. Instead we can unroll the Swiss roll to obtain a 2D dataset. This is an example of manifold learning.

[3]:
```
plt.figure(figsize=(10, 4))


plt.subplot(121)
plt.scatter(swiss[:, 0], swiss[:, 1], c=col)
plt.axis(axes[:4])
plt.xlabel("$x_1$")
plt.ylabel("$x_2$", labelpad=10, rotation=0)
```

```
plt.grid(True)

plt.subplot(122)
plt.scatter(col, swiss[:, 1], c=col)
plt.axis([4, 14.8, axes[2], axes[3]])
plt.xlabel("$z_1$")
plt.grid(True)
```



The swiss roll is an example of a 2D manifold. A 2D manifold is a 2D shape that can be bent and twisted in a higher dimensional space. More generally a d-dimensional manifold is a part of an n-dimensional space (where $d < n$) that locally resembles a d-dimensional hyperplane. In the case of the Swiss roll, $d = 2$ and $n = 3$. Where a hyperplane is a flat subspace with dimension $n - 1$.

2.3 Principle Component Analysis (PCA)

PCA identifies the hyperplane that lies closest to the data, and then projects the data onto it.

2.3.1 Preserving the Variance

Before projecting the data onto a lower dimensional hyperplane, we first need to find the right hyperplane. In finding the right hyperplane we want to preserve as much variance as possible. The hyperplane that preserves the maximum variance is the one that minimizes the mean squared distance between the original data points and their projections onto that plane. This is the principle of PCA.

2.3.2 Principal Components

PCA identifies the axis that accounts for the largest amount of variance in the training set. It then finds a second axis orthogonal to the first that accounts for the largest amount of remaining variance, then another, and so on until there are as many axes as dimensions in the dataset. The i^{th} axis is called the i^{th} principal component (PC) of the data

In finding the the principle component of a training set we use a standard matrix factorization technique called Singular Value Decomposition (SVG) that can decompose the training set matrix X into the matrix multiplication of three matrices $U\Sigma V^T$, where V contains the unit vectors that define all the principal components that we are

looking for. Therefore

$$V = \hat{\mathbf{c}_1}\ \hat{\mathbf{c}_2}\ \ldots\ \hat{\mathbf{c}_n}$$

[4]:
```
swissCentered = swiss - swiss.mean(axis=0)
U, s, Vt = np.linalg.svd(swissCentered)
c1 = Vt[0]
c2 = Vt[1]
c1, c2
```

[4]:
```
(array([ 0.49582458, -0.08887116,  0.86386336]),
 array([-0.86839309, -0.05895388,  0.49235951]))
```

2.3.3 Projecting Down to d Dimensions

Once we have identified all the principal components, we can reduce the dimensionality of the dataset down to d dimensions by projecting it onto the hyperplane defined by the first d principal components. Selecting this hyperplane ensures that the projection will preserve as much variance as possible.

To project the training set onto the hyperplane and obtain the reduced dataset X_d we compute the matrix multiplication of the training set matrix X by the matrix W_d defined as the matrix containing the first d columns of V.

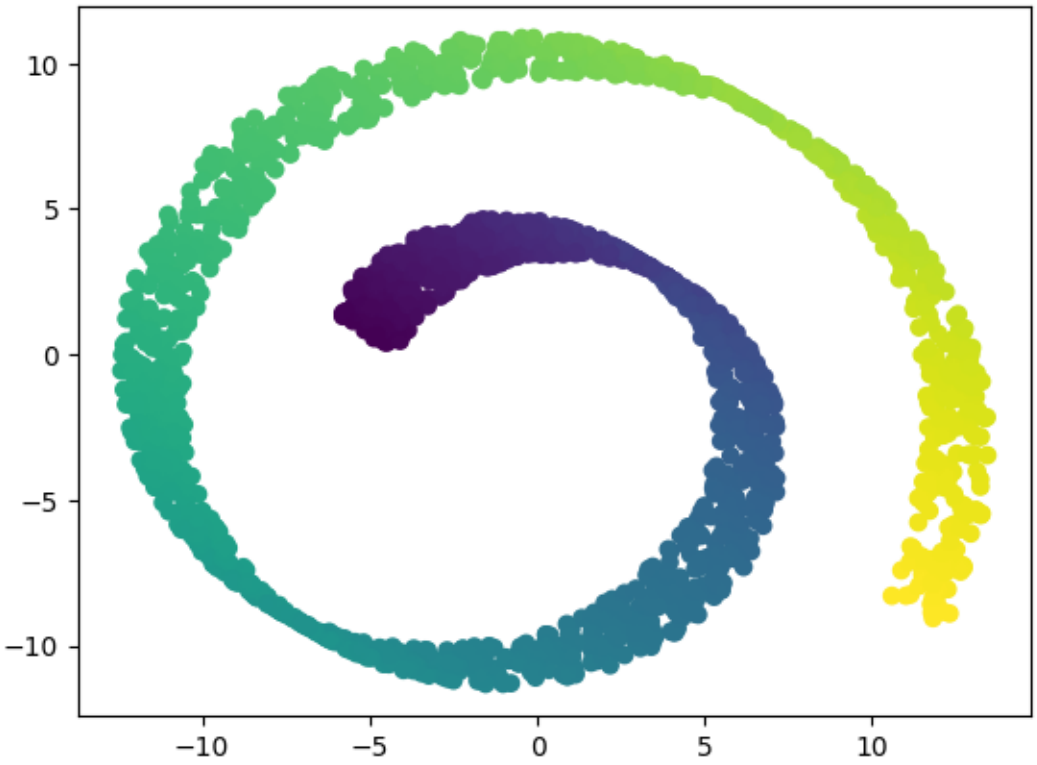
$$X_d = XW_d$$

[5]:
```
W2 = Vt[:2].T
X2D = swissCentered @ W2
X2D[:, 1] = X2D[:, 1] * -1
X2D
```

[5]:
```
array([[ 4.45646186, -8.53342194],
       [13.03828194, -2.78484238],
       [-6.6526694 ,  9.22135169],
       …,
       [-2.9130364 ,  3.01718609],
       [-3.64863529,  2.8203337 ],
       [ 6.93844902, -2.43952201]])
```

[6]:
```
plt.scatter(X2D[:, 0], X2D[:, 1], c=col)
```

[6]:
```
<matplotlib.collections.PathCollection at 0x76ec22f5e960>
```



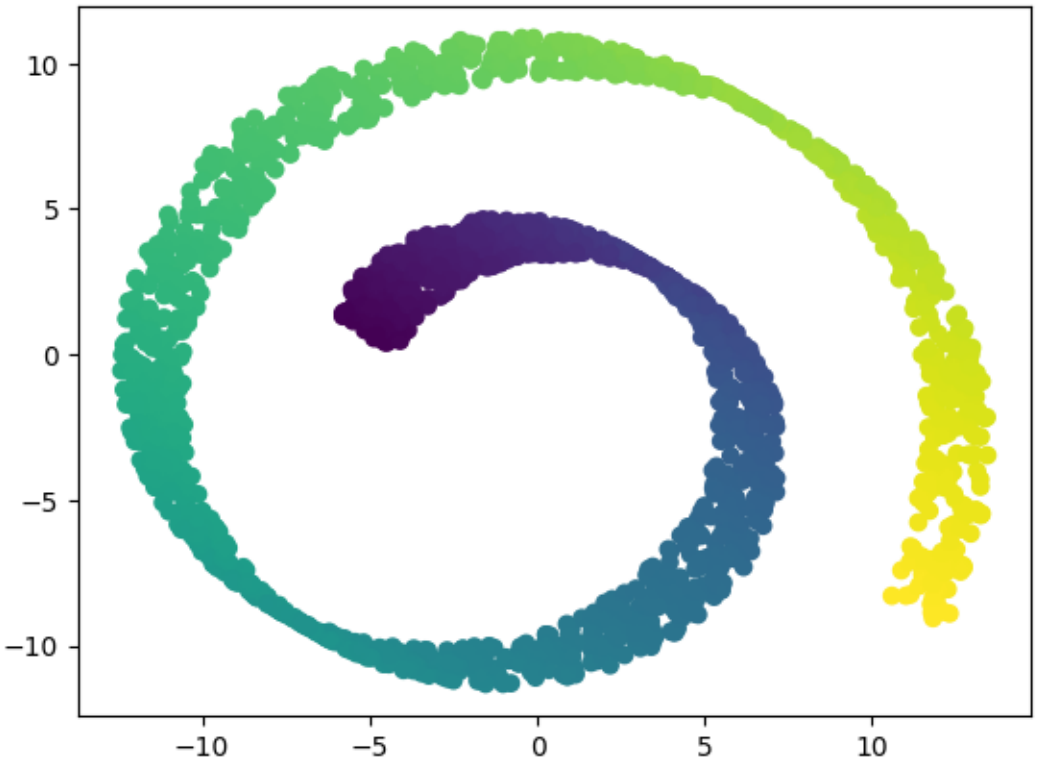
2.3.3.1 Using Scikit-Learn

[7]:
```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X2D = pca.fit_transform(swiss)
X2D
```

[7]:
```
array([[ 4.45646186, -8.53342194],
       [13.03828194, -2.78484238],
       [-6.6526694 ,  9.22135169],
       …,
       [-2.9130364 ,  3.01718609],
       [-3.64863529,  2.8203337 ],
       [ 6.93844902, -2.43952201]])
```

[8]:
```
plt.scatter(X2D[:, 0], X2D[:, 1], c=col)
```

[8]:
```
<matplotlib.collections.PathCollection at 0x76ec21acd700>
```



[9]:
```
pca.components_
```

[9]:
```
array([[ 0.49582458, -0.08887116,  0.86386336],
       [ 0.86839309,  0.05895388, -0.49235951]])
```

The explained variance ratio of each PC indicates the proportion of the dataset that lies along each PC.

[10]:
```
pca.explained_variance_ratio_
```

[10]:
```
array([0.40287395, 0.31287487])
```

2.3.4 Choosing the Right Number of Dimensions

Instead of arbitrarily choosing dimensions it is instead simpler to choose the number of dimensions that add up to a sufficiently large portion of the variance, like 95%. Unless you are reducing dimensionality for data visualization, in which case you will want to reduce the dimensionality down to 2 or 3.

[11]:
```
from sklearn.datasets import fetch_openml

mnist = fetch_openml("mnist_784", as_frame=False, data_home="../datasets",␣
↪cache=True)
```

[12]:
```
split = 60_000
XTrain, yTrain = mnist.data[:split], mnist.target[:split]
XTest, yTest = mnist.data[split:], mnist.target[split:]
```

[13]:
```
pca = PCA()
pca.fit(XTrain)
cumsum = np.cumsum(pca.explained_variance_ratio_)
```

[14]:
```
d = np.argmax(cumsum > 0.95) + 1
pca = PCA(n_components=d)
pcs = pca.fit_transform(XTrain)
```

[15]:
```
pca = PCA(n_components=0.95)
XRed = pca.fit_transform(XTrain)
pca.n_components_
```

[15]:
```
np.int64(154)
```

For using dimensionality reduction as a preprocessing step in a supervised learning task, then you can tune the number of dimensions like any other hyperparamter, as shown in the code below

[16]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.pipeline import make_pipeline

randSkogPipe = make_pipeline(
    PCA(random_state=42), RandomForestClassifier(random_state=42)
)

paramDist = {
    "pca__n_components": np.arange(10, 80),
    "randomforestclassifier__n_estimators": np.arange(50, 500),
}

rndSr = RandomizedSearchCV(
    randSkogPipe,
    param_distributions=paramDist,
    n_iter=10,
    cv=3,
    n_jobs=4,
    random_state=42,
)
rndSr.fit(XTrain[:1000], yTrain[:1000])
```

[16]:
```
RandomizedSearchCV(cv=3,
                   estimator=Pipeline(steps=[('pca', PCA(random_state=42)),
                                             ('randomforestclassifier',
RandomForestClassifier(random_state=42))]),
                   n_jobs=4,
                   param_distributions={'pca__n_components': array([10, 11,
 ↪12,
13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
       27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
       44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 5…
       414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426,
       427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439,
       440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452,
```

```
        453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465,
        466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478,
        479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491,
        492, 493, 494, 495, 496, 497, 498, 499])},
                   random_state=42)
```

[17]:
```
rndSr.best_params_
```

[17]:
```
{'randomforestclassifier__n_estimators': np.int64(475),
 'pca__n_components': np.int64(57)}
```

2.3.5 PCA for Compression

After dimensionality reduction the training set takes up much less space. For example, try applying PCA to the MNIST dataset while preserving 95% of its variance. You should find that each instance will have just over 150 features, instead of the original 784 features. So while most of the variance is preserved, the dataset is now less than 20% of its original size. It is also possible to decompress the reduced dataset back to 784 dimensions by using the inverse transformation of the PCA projection, although due to the variance loss this will not give back the exact same dataset. The mean squared distance between the original data and the decompressed data is called the reconstruction error.

$$X_{\text{recovered}} = X_d W_d$$

[18]:
```
XRec = pca.inverse_transform(XRed)
XRec
```

[18]:
```
array([[0., 0., 0., …, 0., 0., 0.],
       [0., 0., 0., …, 0., 0., 0.],
       [0., 0., 0., …, 0., 0., 0.],
       …,
       [0., 0., 0., …, 0., 0., 0.],
       [0., 0., 0., …, 0., 0., 0.],
       [0., 0., 0., …, 0., 0., 0.]])
```

2.3.6 Randomized PCA

Using a different stochastic algorithm that finds an approximation of the first d principal components much faster than the previous algorithms. This can be done with Sci-kit learn by setting the `svd_solver` parameter to `randomized`. This is exponentially faster than solving the full SVD when d is much smaller than n.

[19]:
```
from sklearn.metrics import root_mean_squared_error

rndPCA = PCA(n_components=154, svd_solver="randomized", random_state=42)
XRedRand = rndPCA.fit_transform(XTrain)

root_mean_squared_error(XRed, XRedRand)
```

[19]:
```
np.float64(12.066353048571338)
```

By default, `svd_solver` is set to `auto` which uses the randomized method if $\max(m, n) > 500$ and `n_components` is an integer smaller than 80% of $\min(m, n)$, else it solves the full SVD. This is why the root mean squared error is zero

2.3.7 Incremental PCA

One problem with all the implementations of PCA covered so far is that they require the whole training set to fit in memory in order to run the algorithm, Incremental PCA (IPCA) algorithms allow you to split the training set into mini-batches and feed these in one mini-batch at a time. This is useful for large training sets and for applying PCA online.

[20]:
```
from sklearn.decomposition import IncrementalPCA

n_batches = 100
incPCA = IncrementalPCA(n_components=154)
for XB in np.array_split(XTrain, n_batches):
    incPCA.partial_fit(XB)

XRed = incPCA.transform(XTrain)
```

Chapter 3

Unsupervised Learning Techniques

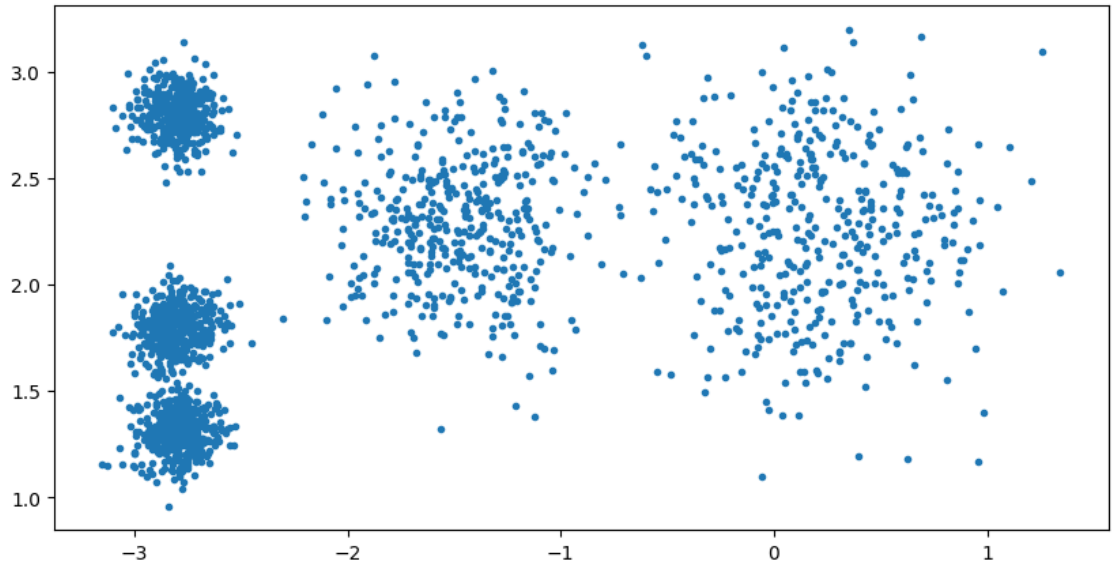
3.1 Clustering Algorithms: k-means and DBScan

Clustering refers to grouping objects based on similar qualities or characteristics / groups of similar instances.

3.1.1 k-means

[21]:
```
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

blob_centers = np.array(
    [[0.2, 2.3], [-1.5, 2.3], [-2.8, 1.8], [-2.8, 2.8], [-2.8, 1.3]]
)
blob_std = np.array([0.4, 0.3, 0.1, 0.1, 0.1])
X, y = make_blobs(
    n_samples=2000, centers=blob_centers, cluster_std=blob_std, random_state=7
)

plt.figure(figsize=(10, 5))
plt.scatter(X[:, 0], X[:, 1], marker=".")
```

[21]:
```
<matplotlib.collections.PathCollection at 0x76ec20976000>
```



[22]:
```
k = 5
km = KMeans(n_clusters=k, n_init=10, random_state=42)
preds = km.fit_predict(X)
preds
```

[22]:
```
array([0, 0, 4, …, 3, 1, 0], dtype=int32)
```

[23]:
```
centers = km.cluster_centers_
centers
```

[23]:
```
array([[-2.80214068,  1.55162671],
       [ 0.08703534,  2.58438091],
       [-1.46869323,  2.28214236],
       [-2.79290307,  2.79641063],
       [ 0.31332823,  1.96822352]])
```

[24]:
```
XNew = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
km.predict(XNew)
```

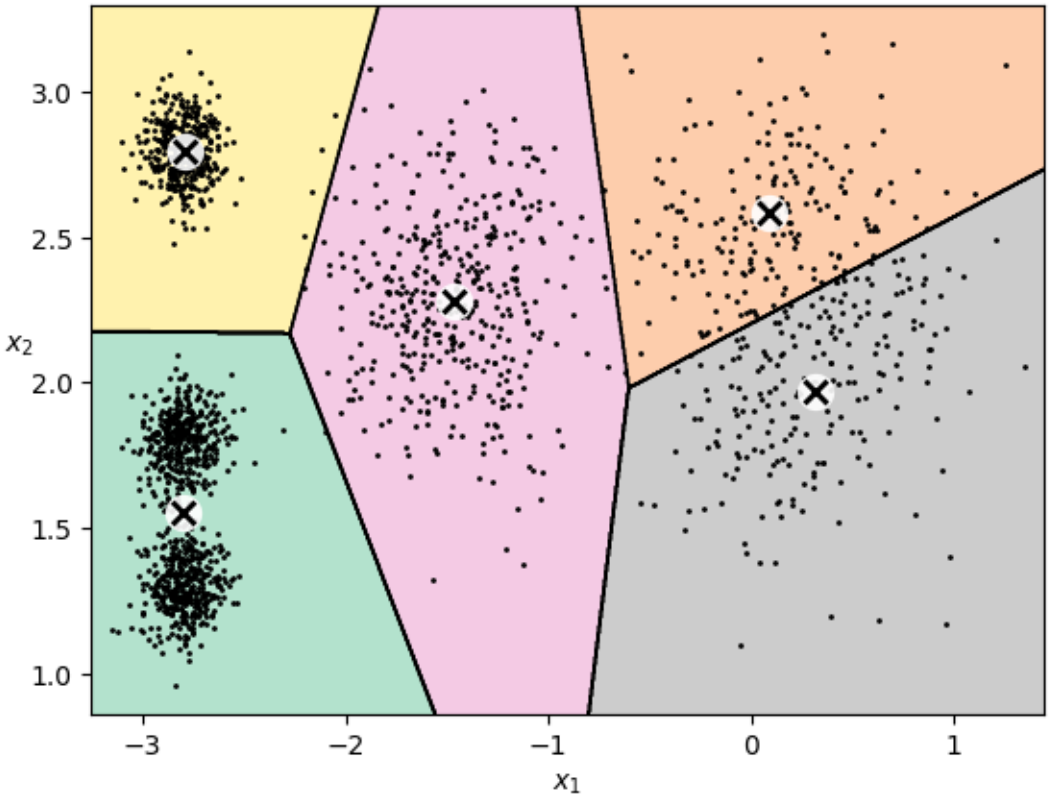
[24]:
```
array([4, 4, 3, 3], dtype=int32)
```

[25]:
```
def plotData(X):
    plt.plot(X[:, 0], X[:, 1], "k.", markersize=2)


def plotCentroids(centroids, weights=None, circle_color="w", cross_color="k"):
    if weights is not None:
        centroids = centroids[weights > weights.max() / 10]
    plt.scatter(
        centroids[:, 0],
        centroids[:, 1],
```

```
        marker="o",
        s=35,
        linewidths=8,
        color=circle_color,
        zorder=10,
        alpha=0.9,
    )
    plt.scatter(
        centroids[:, 0],
        centroids[:, 1],
        marker="x",
        s=2,
        linewidths=12,
        color=cross_color,
        zorder=11,
        alpha=1,
    )


def plotDecBounds(
    clusterer,
    X,
    resolution=1000,
    show_centroids=True,
    show_xlabels=True,
    show_ylabels=True,
):
    mins = X.min(axis=0) - 0.1
    maxs = X.max(axis=0) + 0.1
    xx, yy = np.meshgrid(
        np.linspace(mins[0], maxs[0], resolution),
        np.linspace(mins[1], maxs[1], resolution),
    )
    Z = clusterer.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.contourf(Z, extent=(mins[0], maxs[0], mins[1], maxs[1]),␣
 ↪cmap="Pastel2")
    plt.contour(
        Z, extent=(mins[0], maxs[0], mins[1], maxs[1]), linewidths=1,␣
 ↪colors="k"
    )
    plotData(X)
    if show_centroids:
        plotCentroids(clusterer.cluster_centers_)

    if show_xlabels:
        plt.xlabel("$x_1$")
    else:
        plt.tick_params(labelbottom=False)
    if show_ylabels:
        plt.ylabel("$x_2$", rotation=0)
    else:
```

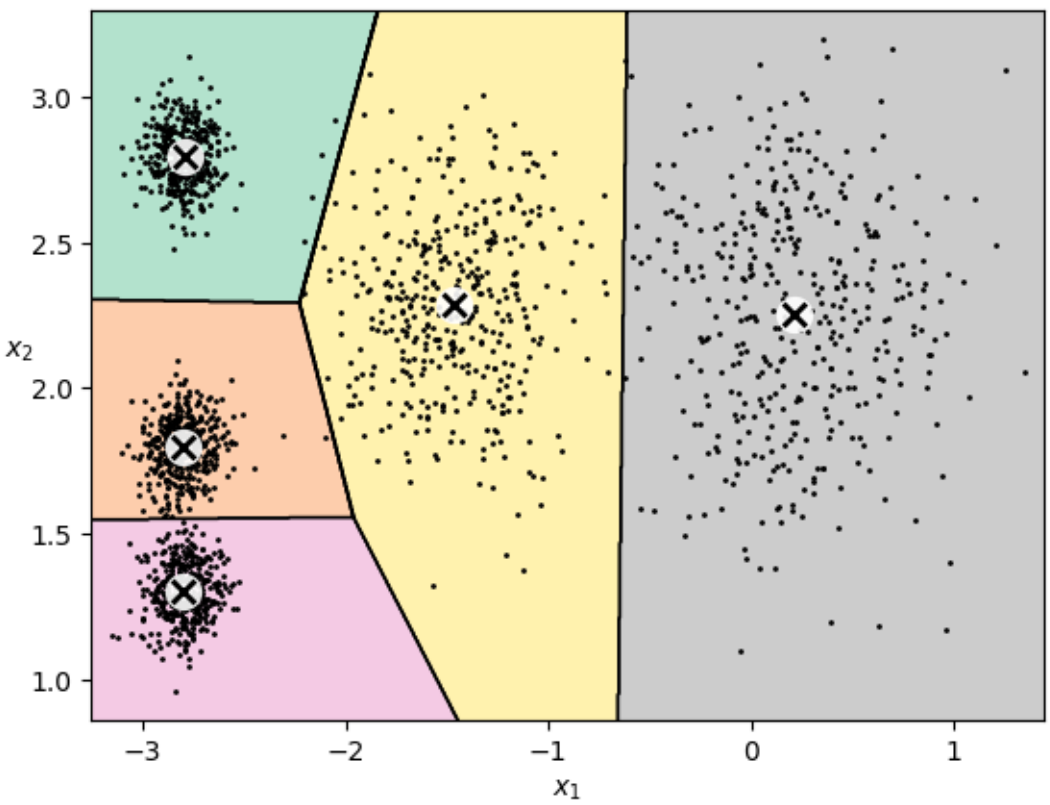
```
        plt.tick_params(labelleft=False)
```

[26]:
```
plotDecBounds(km, X)
```



The K-Means algorithm clusters some groups correctly by does not always find the optimal solution. The algorithm is sensitive to the initial random placement of the centroids. The algorithm is also sensitive to the number of clusters k that you specify. If you already have the most optimum centroids these can be passed to the KMeans transformer as the `init` parameter.

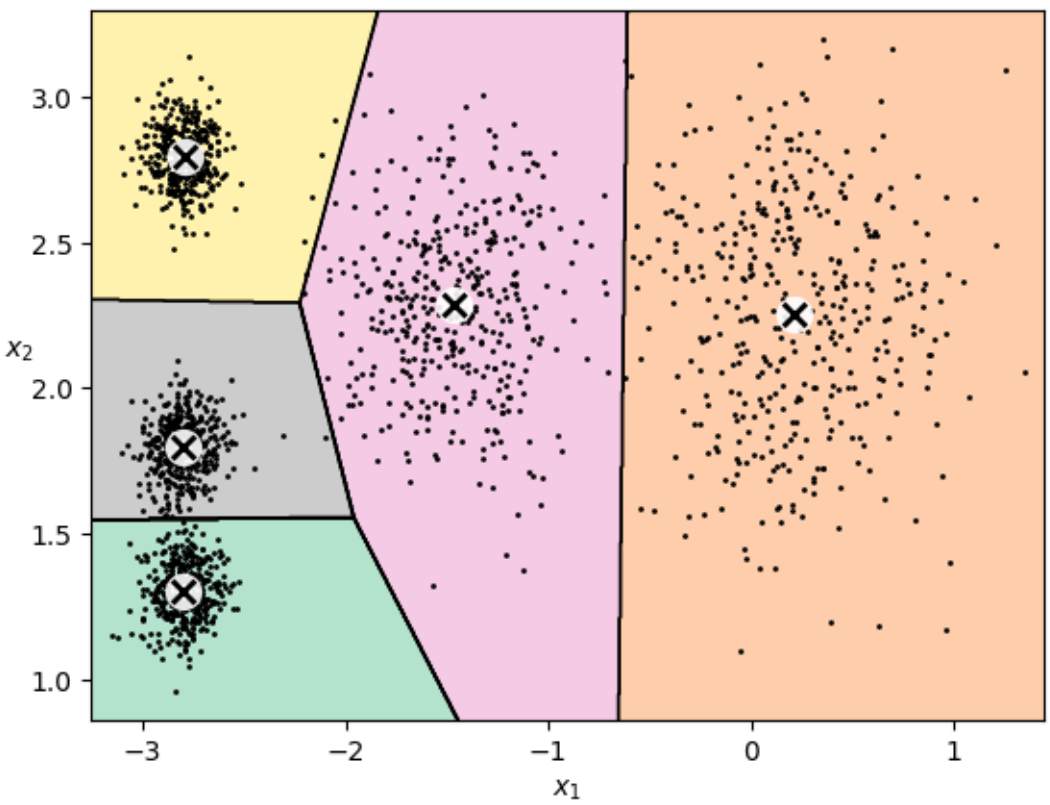
[27]:
```
goodInit = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
km = KMeans(n_clusters=k, init=goodInit, n_init=1, random_state=42)
km.fit(X)
plotDecBounds(km, X)
```



Another solution is to run the algorithm multiple times with different random initializations and keep the best solution, as this is controlled by a hyperparameter `n_init` this can be done by setting a higher `n_init`, using the inertia score as a metric to determine the best solution.

Inertia is the sum of the squared distances between the instances and their closest centroids.

[28]:
```
km = KMeans(n_clusters=k, n_init=40, random_state=42)
km.fit(X)
plotDecBounds(km, X)
```



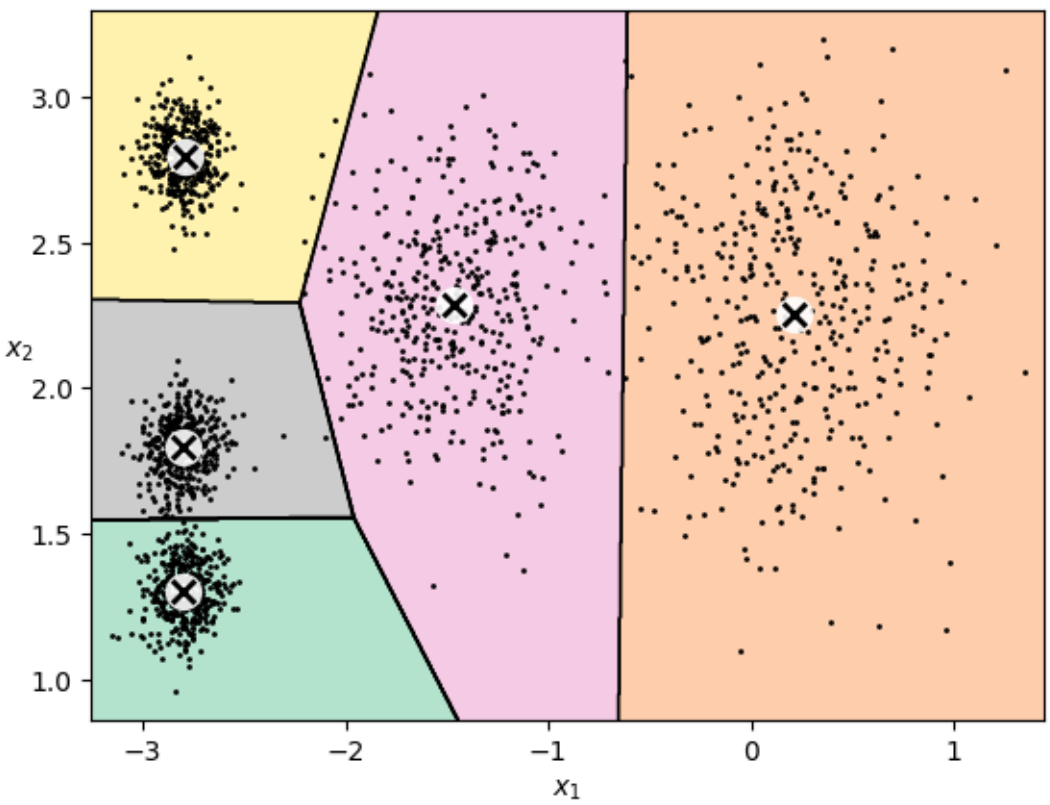
[29]:
```
km.inertia_
```

[29]:
```
211.5985372581684
```

3.1.1.1 Accelerated k-means and mini-batch k-means

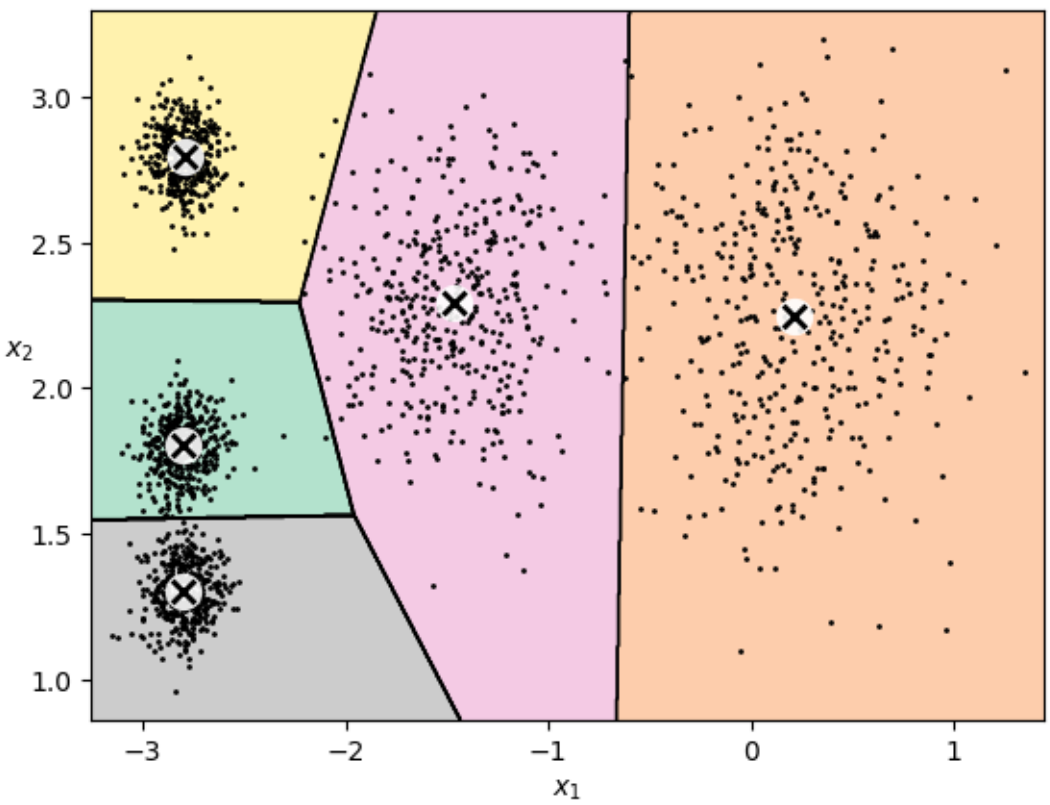
On some large datasets with many clusters, the algorithm can be accelerated by avoiding many unnecessary distance calculations, using triangle inequality (i.e. a straight line is always the shortest distance between two points) and by keeping track of lower and upper bounds distances between instances and centroids. However this approach does not always accelerate training depending on the dataset. This approach was proposed by Charles Elkan and can be used by setting the hyperparameter `algorithm` to `elkan`.

[30]:
```
km = KMeans(n_clusters=k, n_init=40, random_state=42, algorithm="elkan")
km.fit(X)
plotDecBounds(km, X)
```



Another variant of the k-means algorithm, mini-batch kmeans, uses mini-batches of the whole dataset each iteration, moving the centroids just slightly each iteration. This speeds up the algorithm significantly and allows clustering large dataset that would not fit into memory. This increase in speed comes with the drawback of slightly increased inertia.

[31]:
```
from sklearn.cluster import MiniBatchKMeans

miniKM = MiniBatchKMeans(n_clusters=k, random_state=42)
miniKM.fit(X)
plotDecBounds(miniKM, X)
```



[32]:
```
miniKM.inertia_
```

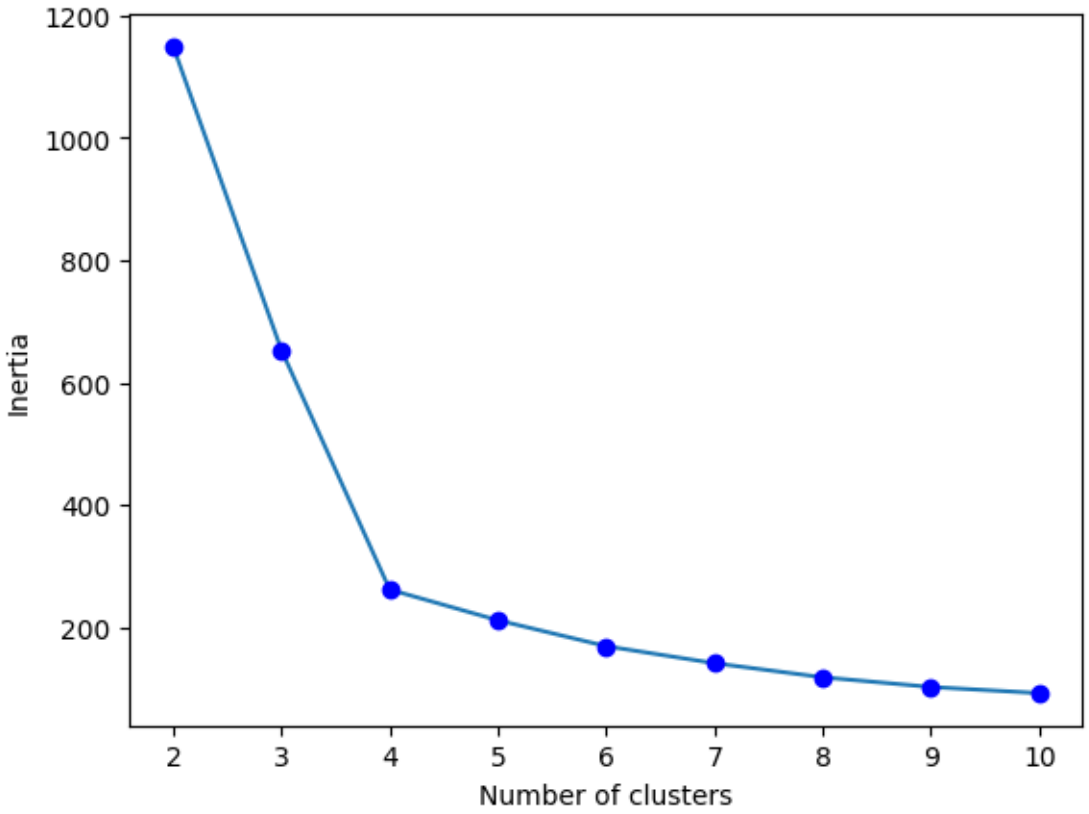
[32]:
```
211.6594510571262
```

3.1.1.2 Finding the optimal number of clusters

The optimal number of clusters is often not clear in unsupervised learning problems, with the total number of possible labels being unknown. One way to find the optimal number of clusters is to use the inertia score as a metric. The inertia score is the sum of the squared distances between the instances and their closest centroids. The inertia score decreases as the number of clusters increases, however the rate of decrease slows down as the number of clusters increases. This is called the elbow method. The optimal number of clusters is the point where the inertia score decreases at a slower rate.

[33]:
```
cSizes = range(2, 11)
kmPerK = [KMeans(n_clusters=k, random_state=42, n_init=40).fit(X) for k in␣
 ↪cSizes]
inertias = [mod.inertia_ for mod in kmPerK]
plt.plot(cSizes, inertias)
plt.plot(cSizes, inertias, "bo")
plt.xlabel("Number of clusters")
plt.ylabel("Inertia")
```

[33]:
```
Text(0, 0.5, 'Inertia')
```



Therefore the optimal number of clusters is the one where the inertia score decreases significantly. This method however is not precise a better but more computationally expensive method is using the **silhouette score**. The Silhouette score is the mean silhouette coefficient over all instances, where an instances silhouette coefficient is

$$\text{Silhouette Coefficient} = \frac{(b-a)}{\max(a,b)}$$

Where a is the mean distance to the other instances in the same cluster / the mean intra-cluster distance, and b is is the nearest-cluster distance, i.e. the mean distance to the instances of the next closest cluster, defined as the one that minimizes b / the minimum mean inter-cluster distance. The Silhouette coef can vary between -1 and $+1$, with values close to $+1$ meaning that the instance is well inside it's own cluster and far from other clusters, while a value close to 0 means that it is close to a cluster boundary and, and finally a value close to -1 means that the instance may have been assigned to the wrong cluster

[34]:
```
from sklearn.metrics import silhouette_score

silhouette_score(X, km.labels_)
```

[34]:
```
np.float64(0.655517642572828)
```

[35]:
```
scores = [silhouette_score(X, mod.labels_) for mod in kmPerK]
plt.figure(figsize=(10, 5))
plt.plot(cSizes, scores)
plt.plot(cSizes, scores, "bo")
plt.xlabel("Number of clusters")
```

```
plt.ylabel("Silhouette Score")
```

[35]: Text(0, 0.5, 'Silhouette Scores')