a04-lvm-output

December 6, 2024

1 Machine Learning

Arne Huckemann (ahuckema), Elise Wolf (eliwolf)

[2]:
```
import numpy as np
import scipy
import scipy.stats
import matplotlib.pyplot as plt
from IPython import get_ipython
from numpy.linalg import svd
from util import nextplot, plot_xy
from sklearn.cluster import KMeans

# setup plotting
import psutil
inTerminal = not "IPKernelApp" in get_ipython().config
inJupyterNb = any(filter(lambda x: x.endswith("jupyter-notebook"), psutil.
 ↪Process().parent().cmdline()))
inJupyterLab = any(filter(lambda x: x.endswith("jupyter-lab"), psutil.Process().
 ↪parent().cmdline()))
if not inJupyterLab:
    from IPython import get_ipython
    get_ipython().run_line_magic("matplotlib", "" if inTerminal else "notebook"␣
 ↪if inJupyterNb else "widget")
```

2 1 Probabilistic PCA

2.1 1a) Toy data

[3]:
```
# You do not need to modify this method.
def ppca_gen(N=10000, D=2, L=2, sigma2=0.5, mu=None, lambda_=None, Q=None,␣
 ↪seed=None):
    """Generate data from a given PPCA model.

    Unless specified otherwise, uses a fixed mean, fixed eigenvalues (variances␣
 ↪along
```

```
    principal components), and a random orthogonal eigenvectors (principal␣
↪components).

    """

    # determine model parameters (from arguments or default)
    rng = np.random.RandomState(seed)
    if mu is None:
        mu = np.arange(D) + 1.0
    if Q is None:
        Q = scipy.stats.ortho_group.rvs(D, random_state=rng)
    if lambda_ is None:
        lambda_ = np.arange(D, 0, -1) * 2

    # weight matrix is determined from first L eigenvectors and eigenvalues of
    # covariance matrix
    Q_L = Q[:, :L]
    lambda_L = lambda_[:L]
    W = Q_L * np.sqrt(lambda_L)  # scales columns

    # generate data
    Z = rng.standard_normal(size=(N, L))  # latent variables
    Eps = rng.standard_normal(size=(N, D)) * np.sqrt(sigma2)  # noise
    X = Z @ W.transpose() + mu + Eps  # data points

    # all done
    return dict(
        N=N, D=D, L=L, X=X, Z=Z, mu=mu, Q_L=Q_L, lambda_L=lambda_L, W=W, Eps=Eps
    )
```

This function simulates data that follows the PPCA model.

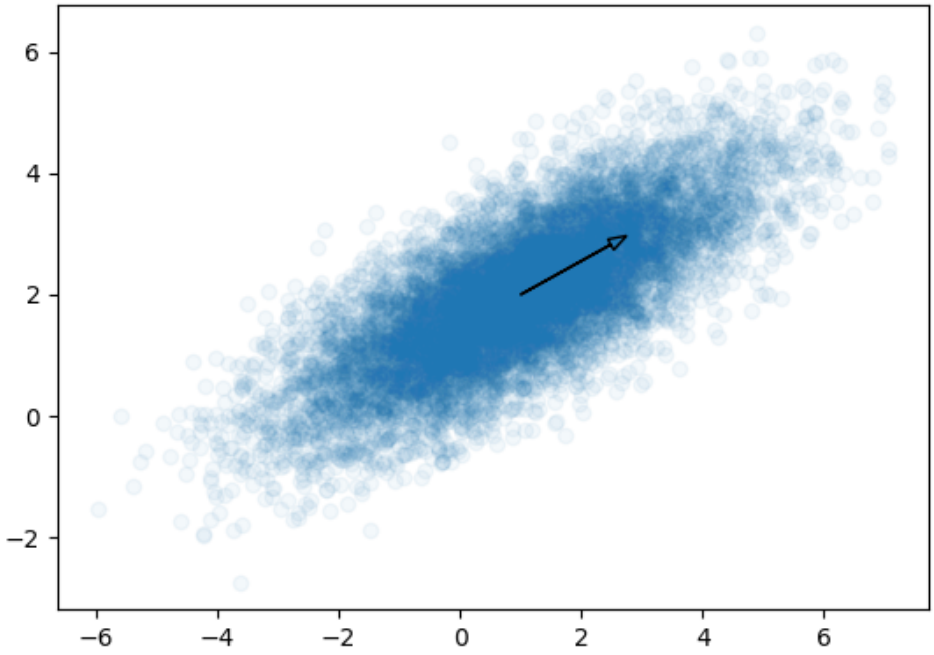
- N: Number of data points to generate.
- D: Dimensionality of the observed data.
- L: Number of principal components (latent variables).
- sigma2: Variance of the noise.
- mu: Mean of the data. If not provided, it defaults to a fixed mean.
- lambda_: Eigenvalues (variances along principal components). If not provided, it defaults to a fixed set of eigenvalues.
- Q: Orthogonal matrix of eigenvectors (principal components). If not provided, it generates a random orthogonal matrix.

Returns additionally - W: Weight matrix calculated by scaling the columns of Q_L by the square root of lambda_L. - Eps: Noise generated from a standard normal distribution of size N x D, scaled by the square root of sigma2. - X: Observed data points calculated as the product of Z and the transpose of W, added to the mean mu and noise Eps.

[4]:
```
# You do not need to modify this method.
def ppca_plot_2d(data, X="X", mu="mu", W="W", alpha=0.05, axis=None, **kwargs):
    """Plot 2D PPCA data along with its weight vectors."""
    if not axis:
        nextplot()
        axis = plt.gca()
    X = data[X] if isinstance(X, str) else X
    plot_xy(X[:, 0], X[:, 1], alpha=alpha, axis=axis, **kwargs)

    # additional plot elements: mean and components
    if mu is not None:
        mu = data[mu] if isinstance(mu, str) else mu
        if W is not None:
            W = data[W] if isinstance(W, str) else W
            head_width = np.linalg.norm(W[:, 0]) / 10.0
            for j in range(W.shape[1]):
                axis.arrow(
                    mu[0],
                    mu[1],
                    W[0, j],
                    W[1, j],
                    length_includes_head=True,
                    head_width=head_width,
                )
```

[5]:
```
# Generate and plot a toy dataset
toy_ppca = ppca_gen(L=1, sigma2=0.5, seed=0)
ppca_plot_2d(toy_ppca)
print(np.sum(toy_ppca["X"] ** 3))  # must be 273244.3990646409
print(toy_ppca['W'])
print(toy_ppca['mu'])
```

```
273244.39906464086
[[1.74885766]
 [0.97030762]]
[1. 2.]
```



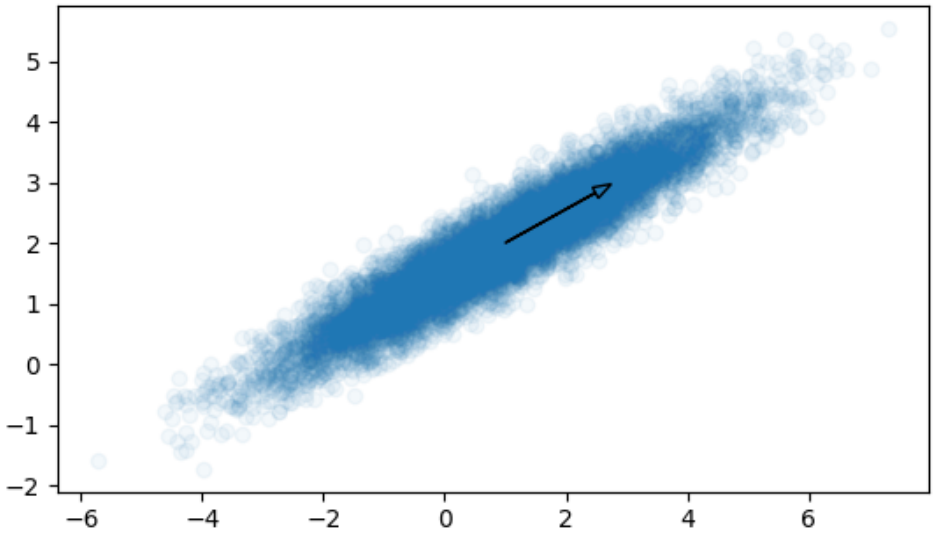
This function `ppca_gen` generates a 2-dimensional scatter plot of 10000 data points that are diagonally distributed along its weight vector [1.74885766, 0.97030762] for the first principal component. The data points are mainly between -2 and 4 on the x-axis and 0 and 4 on the y-axis. The noise is set through the variance 0.5 and the mean of the data is [1, 2].

The diagonal distribution of the points along the vector reflects the underlying structure captured by the probabilistic PCA model.

2.1.1 Impact of Noise

[6]:
```
# Generate and plot a toy dataset
toy_ppca_2 = ppca_gen(L=1, sigma2=0.1, seed=0)
print(toy_ppca_2 ['W'])
print(toy_ppca_2 ['mu'])
ppca_plot_2d(toy_ppca_2)
```

```
[[1.74885766]
 [0.97030762]]
[1. 2.]
```



This function `ppca_gen` generates a 2-dimensional scatter plot of 10000 data points that are diagonally distributed along its weight vector [1.74885766, 0.97030762] for the first principal component. The noise is set through the variance 0.1 and the mean of the data is [1, 2].

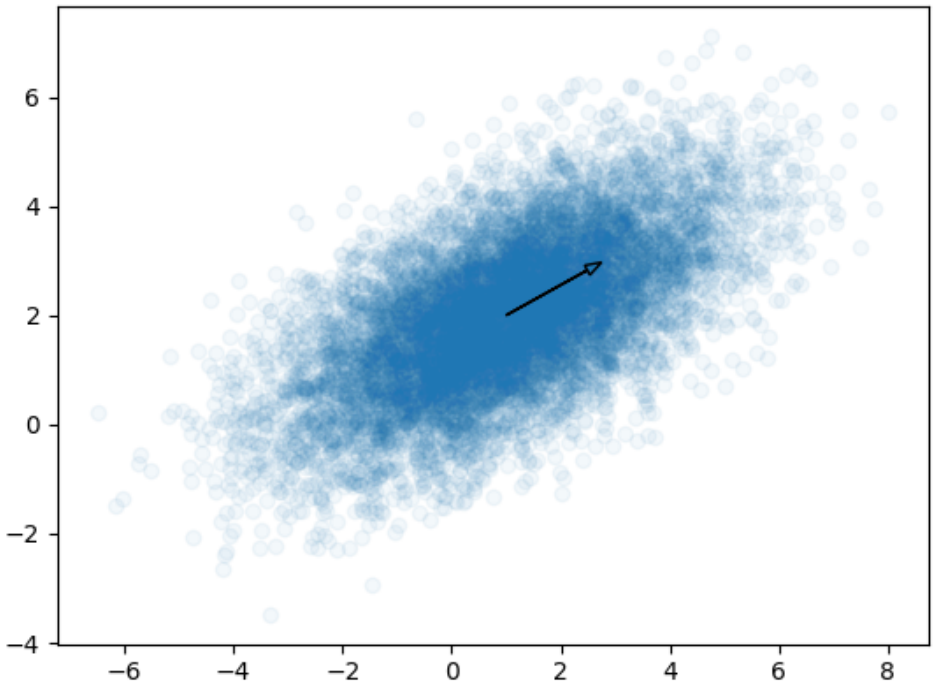
The difference in the noise points variance influenced the plot by altering the spread of the data points around the principal component. With a lower noise variance ($\sigma^2 = 0.1$), the data points are more tightly clustered along the weight vector, resulting in a narrower distribution. This compression is evident as the points mainly lie between -2 and 6 on the x-axis and 0 and 4 on the y-axis. The reduced noise variance introduces less randomness, making the points adhere more closely to the underlying structure captured by the probabilistic PCA model. The diagonal distribution of the points along the vector reflects the underlying structure captured by the probabilistic PCA model.

The variance of the noise points in a probabilistic PCA model influences the spread of the data points around the principal component. Since variance measures the spread of the data points. A higher variance means the data points are more spread out, while a lower variance means they are more tightly clustered around the mean. The points are more aligned along the principal component, making the diagonal distribution clearer.

[7]:
```
# Generate and plot a toy dataset
toy_ppca_3 = ppca_gen(L=1, sigma2=1, seed=0)
print(toy_ppca_3 ['W'])
```

```
print(toy_ppca_3 ['mu'])
ppca_plot_2d(toy_ppca_3)
```

```
[[1.74885766]
 [0.97030762]]
[1. 2.]
```



This function `ppca_gen` generates a 2-dimensional scatter plot of 10000 data points that are diagonally distributed along its weight vector [1.74885766, 0.97030762] for the first principal component. The noise is set through the variance 1.0 and the mean of the data is [1, 2].

The difference in the noise points variance influenced the plot by altering the spread of the data points around the principal component. With a higher noise variance ($\sigma^2 = 1.0$), the data points are much more spread out and less compressed regarding the orthogonal direction of the weight vector. This is evident as the points mainly lie between -2 and 4 on the x-axis and 0 and 4 on the y-axis. Compared to the first plot, the data points are concentrated within the same x and y axis parts but are more spread out and not densely near the weight vector anymore. The increased noise variance introduces more randomness, making the points deviate further from the underlying structure captured by the probabilistic PCA model.

The variance of the noise points in a probabilistic PCA model influences the spread of the data points around the principal component. Since variance measures the spread of the data points, a

higher variance means the data points are more spread out, while a lower variance means they are more tightly clustered around the mean. The points are less aligned along the principal component, making the diagonal distribution less clear.

In summary, different variance leads to: - For $\sigma^2 = 0.1$, the data points are closely packed along the diagonal vector. - For $\sigma^2 = 0.5$, the data points are more spread out but still follow the diagonal trend. - For $\sigma^2 = 1.0$, the data points are even more dispersed, showing a broader distribution around the diagonal vector.

These observations illustrate how the noise variance impacts the data distribution in probabilistic PCA.

2.2 1b) Maximum Likelihood Estimation

[8]:
```python
def ppca_mle(X, L):
    """Computes the ML estimates of PPCA model parameters.

    Returns a dictionary with keys `mu`, `W`, and `sigma2` and the corresponding ML
    estimates as values.

    """
    N, D = X.shape

    # Compute the ML estimates of the PPCA model parameters: mu_mle, sigma2_mle (based
    # on mu_mle), and W_mle (based on mu_mle and sigma2_mle). In your code, only use
    # standard matrix/vector operations and svd(...).
    # YOUR CODE HERE

    # calculate mean of the data and center data
    mu_mle = np.mean(X, axis=0)
    X_centered = X - mu_mle

    # SVD
    U, S, Vt = np.linalg.svd(X_centered, full_matrices=False)
    V = Vt.T
    eigenvalues = (S**2) / (N)  # eigenvalues of the covariance matrix


    # calculate W and sigma2
    Q_L = V[:, :L]
    Lambda_L = np.diag(eigenvalues[:L])
    sigma2_mle = np.mean(eigenvalues[L:]) if L < D else 0  # for L=D, there are no left dimensions
    W_mle = Q_L @ np.sqrt(Lambda_L - sigma2_mle * np.eye(L))
```

```
    return dict(mu=mu_mle, W=W_mle, sigma2=sigma2_mle)
```

[9]:
```
# Test your solution. This should produce:
# {'mu': array([0.96935329, 1.98309575]),
#  'W': array([[-1.72988776], [-0.95974566]]),
#  'sigma2': 0.4838656103694303}
ppca_mle(toy_ppca["X"], 1)
```

[9]:
```
{'mu': array([0.96935329, 1.98309575]),
 'W': array([[-1.72988776],
        [-0.95974566]]),
 'sigma2': np.float64(0.48386561036943065)}
```

[10]:
```
# Test your solution. This should produce:
# {'mu': array([0.96935329, 1.98309575]),
 # 'W': array([[-1.83371058,  0.33746522], [-1.0173468 , -0.60826214]]),
 # 'sigma2': 0.0}
ppca_mle(toy_ppca["X"], 2)
```

[10]:
```
{'mu': array([0.96935329, 1.98309575]),
 'W': array([[-1.83371058,  0.33746522],
        [-1.0173468 , -0.60826214]]),
 'sigma2': 0}
```

When $ L = D $ (in this case $ L=2 $ for $ D=2 $), there are no remaining dimensions between L and D to capture the variance that is not yet respected in the first L eeigenvalues. This means that the variance of the discarded dimensions $ ^2_{MLE} $ is zero. This reflects that the model captures all the variance of the data in the latent dimensions and therefore doesn't have to capture the remaining variance in $ ^2_{MLE} $.

Formally, the variance $ ^2_{MLE} $ is calculated as:

$$\sigma^2_{MLE} = \frac{1}{D-L} \sum_{i=L+1}^{D} \lambda_i$$

where $ _i $ are the eigenvalues of the covariance matrix of the centered data. When $ L = D $, the sum is over an empty set, and thus:

$$\sigma^2_{MLE} = 0$$

This indicates that all the variance in the data is captured by the principal components, and there is no remaining variance to be attributed to noise.

2.3 1c) Negative Log-Likelihood

[11]:
```python
def ppca_nll(X, model):
    """Compute the negative log-likelihood for the given data.

    Model is a dictionary containing keys "mu", "sigma2" and "W" (as produced by
    `ppca_mle` above).

    """
    N, D = X.shape

    # YOUR CODE HERE

    # Extrahiere Modellparameter
    mu = model["mu"]
    W = model["W"]
    sigma2 = model["sigma2"]

    # Berechne C und C^-1
    C = W @ W.T + sigma2 * np.eye(D)
    C_inv = np.linalg.inv(C)

    # Berechne log|C|
    log_det_C = np.log(np.linalg.det(C))

    # Empirische Kovarianzmatrix
    X_centered = X - mu
    S = (1 / N) * (X_centered.T @ X_centered)

    # Berechnung der negativen Log-Likelihood
    nll = 0.5 * N * (D * np.log(2 * np.pi) + log_det_C + np.trace(C_inv @ S))
    return nll
```

To calculate the Negative Log-Likelihood (NLL), we have to define our **Covariance Matrix $C $**

$$C = WW^T + \sigma^2 I$$

where $ I $ is the identity matrix.

Then we have to calulate the **Log-Determinant of** C

$$\log|C| = \log(\text{np.linalg.det}(C))$$

The **Empirical Covariance Matrix $ S $** is defined as

$$S = \frac{1}{N}\sum_{n=1}^{N}(x_n - \mu)(x_n - \mu)^T = \frac{1}{N}X_{\text{centered}}^T X_{\text{centered}}$$

Then our **Negative Log-Likelihood (NLL)** is given through

$$\text{NLL} = \frac{N}{2}\left(D\log(2\pi) + \log|C| + \text{trace}(C^{-1}S)\right)$$

as in the book "Probabilistic Machine Learning: An Introduction". Online version. November 23, 2024

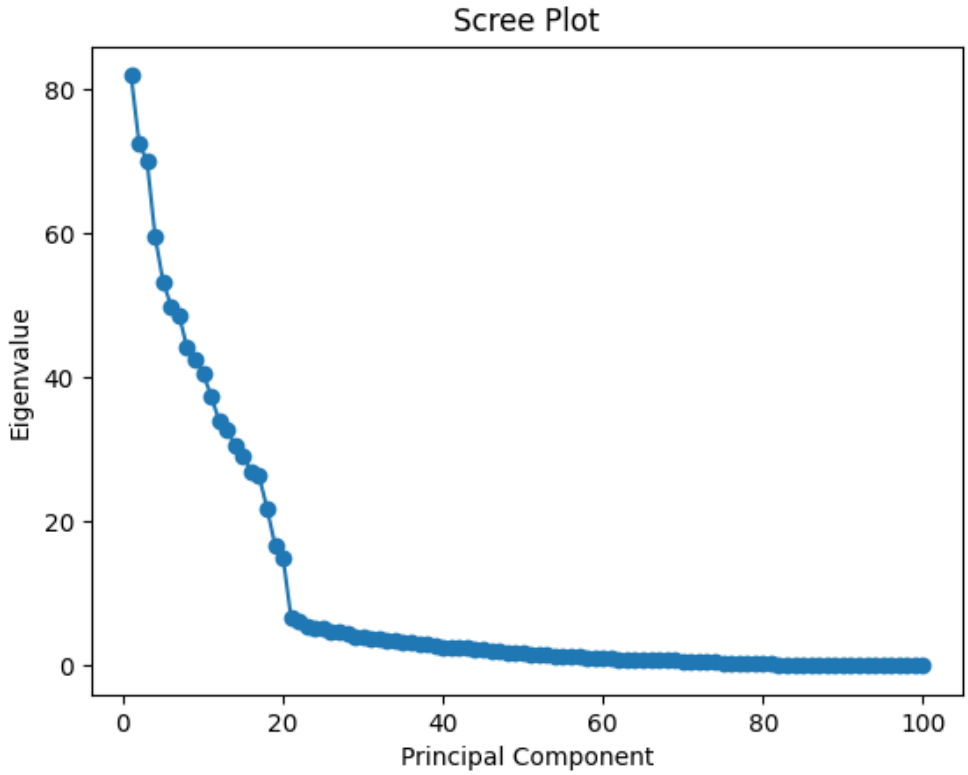
[12]:
```
# Test your solution. This should produce: 32154.198760474777
ppca_nll(toy_ppca["X"], ppca_mle(toy_ppca["X"], 1))
```

[12]:
```
np.float64(32154.198760474763)
```

2.4 1d) Discover the Secret!

[13]:
```
# Load the secret data
X = np.loadtxt("data/secret_ppca.csv", delimiter=",")
```

[14]:
```
# Determine a suitable choice of L using a scree plot.
# Compute the eigenvalues of the covariance matrix of the training data
cov_matrix = np.cov(X, rowvar=False)
eigenvalues, _ = np.linalg.eigh(cov_matrix)

# Sort eigenvalues in descending order
eigenvalues = np.flip(np.sort(eigenvalues))

# Plot the scree plot
plt.figure()
plt.plot(range(1, len(eigenvalues) + 1), eigenvalues, marker='o')
plt.xlabel('Principal Component')
plt.ylabel('Eigenvalue')
plt.title('Scree Plot')
plt.show()
```

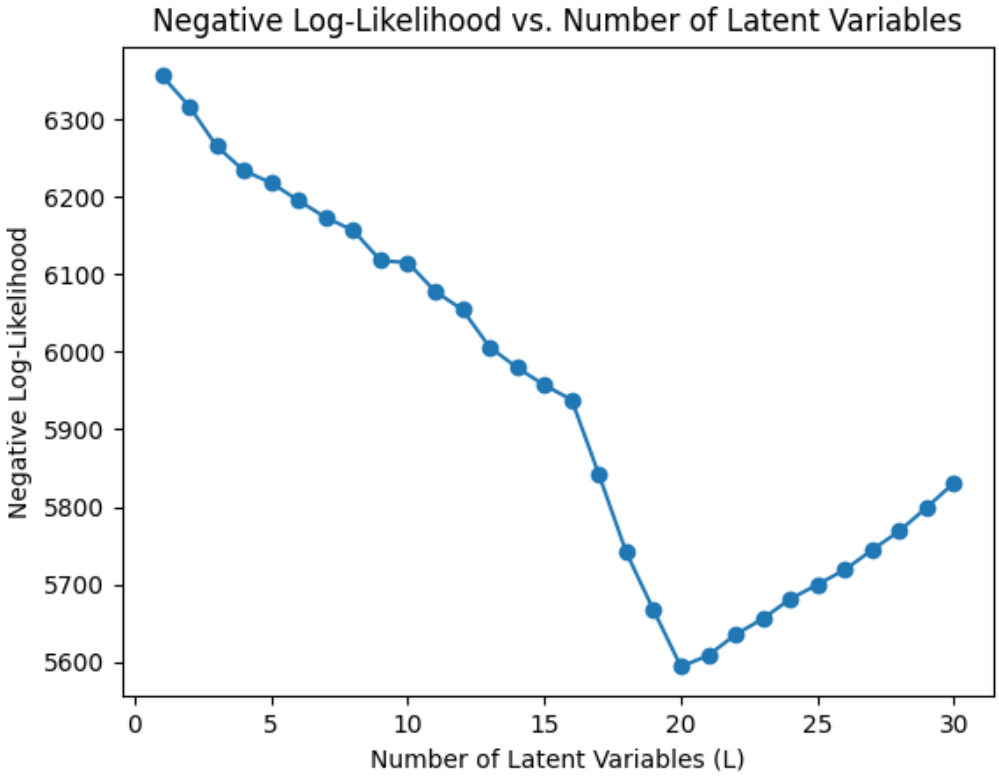


Between 20 and 21, the largest kink is observed, suggesting that the number of used latent variables is likely 20.

[15]:
```
# Determine a suitable choice of L using validation data.
split = len(X) * 3 // 4
X_train = X[:split,]
X_valid = X[split:,]
```

[16]:
```
# YOUR CODE HERE
# Check the assumption on the number of latent variables using validation data
nlls = []
L_values = range(1, 31)  # Check for L values from 1 to 30

for L in L_values:
    model = ppca_mle(X_train, L)
    nll = ppca_nll(X_valid, model)
    nlls.append(nll)

# Plot the negative log-likelihoods for different L values
plt.figure()
plt.plot(L_values, nlls, marker='o')
```

```
plt.xlabel('Number of Latent Variables (L)')
plt.ylabel('Negative Log-Likelihood')
plt.title('Negative Log-Likelihood vs. Number of Latent Variables')
plt.show()

# Determine the optimal number of latent variables
optimal_L = L_values[np.argmin(nlls)]
print(f"The optimal number of latent variables (L) is: {optimal_L}")
```




```
The optimal number of latent variables (L) is: 20
```

2.4.1 Evaluation of the Optimal Number of Latent Variables (L)

In this section, we aim to determine the optimal number of latent variables (L) for our probabilistic PCA model. We approach this task using two methods: 1. Studying the scree plot. 2. Using validation data.

(i) Studying the Scree Plot The scree plot is a graphical representation of the eigenvalues of the covariance matrix of the data. It helps us identify the number of principal components that capture the most variance in the data. The plot typically shows a sharp decline in eigenvalues, followed by a plateau. The point where the decline slows down is often considered the optimal

number of latent variables.

In our scree plot, we observed a significant kink between the 20th and 21st eigenvalues. This suggests that the first 20 principal components capture most of the variance in the data, and adding more components beyond this point yields diminishing returns.

(ii) Using Validation Data To validate our choice of L, we split the data into training and validation sets. We then computed the negative log-likelihood (NLL) for different values of L on the validation set. The NLL measures how well the model fits the data, with lower values indicating a better fit.

We plotted the NLL against the number of latent variables (L) and observed that the NLL reached its minimum at L = 20. This indicates that the model with 20 latent variables provides the best fit to the validation data.

Comparison of Results Both methods—studying the scree plot and using validation data—converged on the same conclusion: the optimal number of latent variables (L) is 20. This agreement between the two methods strengthens our confidence in the result.

Conclusion Based on the scree plot and validation data, we conclude that the optimal number of latent variables (L) for our probabilistic PCA model is 20. This choice balances the trade-off between model complexity and the ability to capture the underlying structure of the data.

3 2 Gaussian Mixture Models

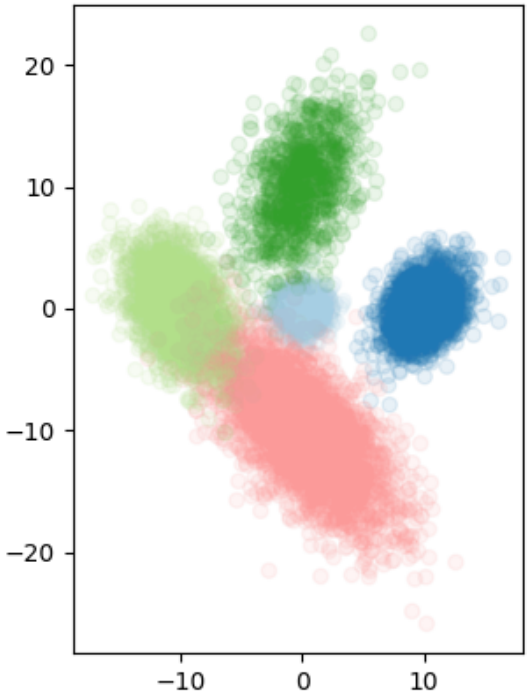
3.1 2a) Toy data

[17]:
```
# You do not need to modify this function.
def gmm_gen(N, mu, pi, Sigma=None, seed=None):
    """Generate data from a given GMM model.

    `N` is the number of data points to generate. `mu` and `Sigma` are lists␣
↪with `K`
    elements holding the mean and covariance matrix of each mixture component.␣
↪`pi` is a
    `K`-dimensional probability vector of component sizes.

    If `Sigma` is unspecified, a default (random) choice is taken.
    """
    K = len(pi)
    D = len(mu[0])
    rng = np.random.RandomState(seed)
    if Sigma is None:
        Sigma = [
            Q.transpose() @ np.diag([(k + 1) ** 2, k + 1]) @ Q
            for k, Q in enumerate([scipy.stats.ortho_group.rvs(2,␣
↪random_state=rng) for k in range(K)])
        ]
```

```
    components = rng.choice(range(K), p=pi, size=N)
    X = np.zeros([N, D])
    for k in range(K):
        indexes = components == k
        N_k = np.sum(indexes.astype(np.int_))
        if N_k == 0:
            continue

        dist = scipy.stats.multivariate_normal(mean=mu[k], cov=Sigma[k],␣
 ↪seed=rng)
        X[indexes, :] = dist.rvs(size=N_k)

    return dict(X=X, components=components, mu=mu, Sigma=Sigma, pi=pi)
```

[18]:
```
# Generate a toy dataset and plot it.
toy_gmm = gmm_gen(
    10000,
    [
        np.array([0, 0]),
        np.array([10, 0]),
        np.array([-10, 0]),
        np.array([0, 10]),
        np.array([0, -10]),
    ],
    np.array([0.1, 0.2, 0.25, 0.1, 0.35]),
    seed=4,
)

print(np.sum(toy_gmm["X"] ** 3))  # must be -4380876.753061278

plot_xy(toy_gmm["X"][:, 0], toy_gmm["X"][:, 1], toy_gmm["components"], alpha=0.
 ↪1)
```

```
-4380876.753061278
```



3.1.1 Plot Description and Analysis

The dataset displayed in the plot was generated using a Gaussian Mixture Model (GMM). The GMM consists of five components, each defined by a mean vector, a covariance matrix, and a mixture proportion. The code creates these clusters by sampling points from multivariate normal distributions.

3.1.2 GMM Parameters:

1. **Number of points**: The dataset contains 10,000 points.
2. **Means of the clusters**:
   - Cluster 1: [0, 0]
   - Cluster 2: [10, 0]
   - Cluster 3: [-10, 0]
   - Cluster 4: [0, 10]
   - Cluster 5: [0, -10]
3. **Proportions of each cluster ( )**:
   - Cluster 1: 0.1
   - Cluster 2: 0.2
   - Cluster 3: 0.25
   - Cluster 4: 0.1

- Cluster 5: 0.35

4. **Covariance matrices**: If not specified (as in this case), random covariance matrices are generated. This randomness introduces different spreads and orientations for each cluster.

3.1.3 Key Observations from the Plot

The plot reveals five distinct clusters, each with unique characteristics in size, orientation, and density. Below is a detailed description of each cluster:

1. **Pink Cluster**:
   - **Location**: The cluster is situated roughly between -10 and 10 on the x-axis and -2 to -20 on the y-axis.
   - **Shape and Orientation**: It is elongated diagonally from the top-left to the bottom-right.
   - **Density**: This is the largest and densest cluster, representing 35% of the total data points (as per the GMM's).
2. **Light Green Cluster**:
   - **Location**: Found between -15 and -5 on the x-axis and -7 to 7 on the y-axis.
   - **Shape and Orientation**: Its structure aligns with the direction of the pink cluster, appearing as an extension.
   - **Density**: Slightly less dense compared to the pink cluster, representing 25% of the data points.
3. **Light Blue Cluster**:
   - **Location**: Centered near the origin ([0, 0]) with a radius of approximately 3.
   - **Shape and Orientation**: It is circular and isotropic, unlike the other clusters.
   - **Density**: This is the smallest cluster, corresponding to only 10% of the data points.
4. **Dark Blue Cluster**:
   - **Location**: Positioned between 5 and 15 on the x-axis and -5 to 5 on the y-axis.
   - **Shape and Orientation**: This cluster is rotated differently from others and oriented diagonally from the bottom-left to the top-right. It is slightly more compressed compared to the light green cluster.
   - **Density**: Represents 20% of the dataset.
5. **Dark Green Cluster**:
   - **Location**: The core lies between -5 and 5 on the x-axis and 5 to 15 on the y-axis.
   - **Shape and Orientation**: The cluster is vertically elongated but slightly rotated from the bottom-left to the top-right.
   - **Density**: This cluster is the least dense and most spread-out among all, corresponding to 10% of the data points.

3.1.4 Connection Between Code and Plot

1. **Cluster Locations (Means)**:
   - The means defined in the code (`mu`) directly determine the central positions of the clusters in the plot:
     - `[0, 0]` corresponds to the **light blue circular cluster** at the origin.
     - `[10, 0]` corresponds to the **dark blue diagonal cluster** on the right side of the plot.
     - `[-10, 0]` corresponds to the **light green diagonal cluster** on the left side of the plot.

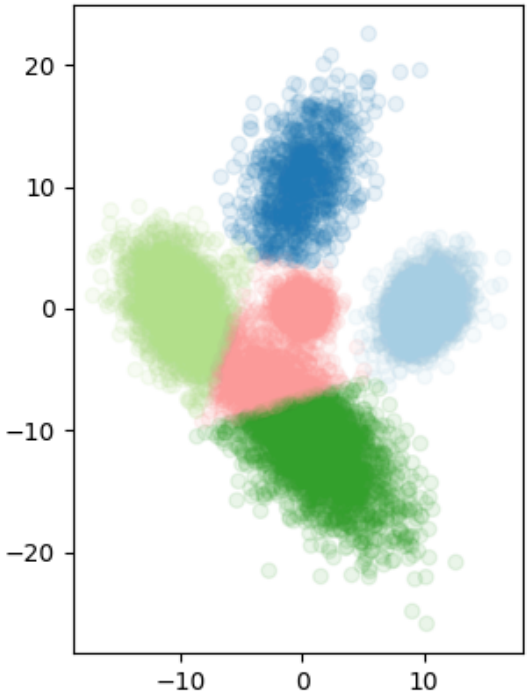
- `[0, 10]` corresponds to the **dark green vertically elongated cluster** near the top.
- `[0, -10]` corresponds to the **pink elongated cluster** near the bottom.

2. **Cluster Sizes (Proportions,)**:
   - The proportions (`pi`) in the code influence the relative sizes (densities) of the clusters:
     - The **pink cluster** is the largest, aligning with the highest proportion (`0.35`).
     - The **light green cluster** is the second-largest, matching its proportion (`0.25`).
     - The **dark blue cluster** is intermediate in size, reflecting its proportion (`0.2`).
     - The **dark green cluster** and **light blue cluster** are smaller, consistent with their lower proportions (`0.1` each).
3. **Cluster Shapes and Orientations (Covariance Matrices, Σ)**:
   - The covariance matrices (`Sigma`) define the shapes and orientations of each cluster:
     - The **pink cluster's diagonal orientation** and large spread are explained by its covariance structure, which generates a top-left to bottom-right elongation.
     - The **light green cluster's diagonal alignment**, extending in a similar direction, arises from a covariance matrix with similar orientation but less spread.
     - The **dark blue cluster's diagonal alignment**, running bottom-left to top-right, reflects a rotated and slightly compressed covariance structure.
     - The **dark green cluster's vertical elongation**, with slight rotation, is due to a covariance matrix emphasizing vertical spread.
     - The **light blue cluster's circular shape** matches an isotropic covariance structure.
4. **Density and Distribution**:
   - The random sampling process (`N_k` points per cluster based on proportions and random seed) ensures the clusters' sizes in the plot mirror the proportions provided in the code. The visible density differences, especially the sparsity of the dark green cluster, directly result from its low proportion and high spread.

The GMM generates a diverse dataset with clusters varying in size, shape, orientation, and density, as illustrated in the plot. The largest cluster (pink) dominates the visual spread, while the smallest (light blue) remains compact and isotropic. The light green and dark blue clusters add intermediate levels of density and unique directional characteristics. Finally, the dark green cluster stands out as the most spread and least dense, further diversifying the visual complexity of the data.

3.2 2b) K-Means

[19]:
```
# Fit 5 clusters using k-means.
kmeans = KMeans(5).fit(toy_gmm["X"])
plot_xy(toy_gmm["X"][:, 0], toy_gmm["X"][:, 1], kmeans.labels_, alpha=0.1)
```



4 Comparison of Gaussian Mixture Model (GMM) and K-Means Clustering

4.1 GMM Plot and K-Means Plot: Visual Comparison and Analysis

The GMM and K-Means clustering results are compared to evaluate how well K-Means reproduces the original GMM structure. While both methods generate five clusters, there are notable differences in the cluster shapes, sizes, and boundaries due to the fundamental differences in the clustering algorithms.

4.1.1 Key Observations:

1. **Clusters That Are Similar**:
   - The **dark blue cluster** and the **light green cluster** remain almost unchanged between the two plots. This similarity arises because these clusters have distinct and well-separated means and structures in the original GMM, which K-Means handles effectively.
2. **Clusters with Significant Differences**:
   - The **light blue cluster** in the K-Means plot now extends from the origin to the bottom-left corner. This extension merges parts of the original **pink cluster** and **light green cluster** into the new light blue cluster.

- The **pink cluster** in the K-Means plot is smaller, as part of its data points have been reassigned to the **light blue cluster**.
- The boundary between the original **pink cluster** and **light blue cluster** is now more horizontal. This change occurs because K-Means produces straight-line decision boundaries that are orthogonal to the feature space, unlike the flexible covariance-driven boundaries in GMM.

3. **Boundary Adjustments**:
   - The most noticeable change is the **diagonal decision boundary** between the original **pink cluster** and **light blue cluster**, cutting from the top-right to the bottom-left. This creates a sharp separation that includes overlapping points from both clusters into the new light blue cluster.
   - The **original light green cluster** boundary remains nearly intact, as its distinct separation and strong density prevent significant reassignment.

4.1.2 Reasoning About Decision Boundaries:

The differences in decision boundaries result from the fundamental mechanisms of K-Means clustering: - **K-Means** uses Euclidean distance to assign points to the nearest centroid, leading to **linear boundaries** between clusters. - **GMM** assigns points based on a probabilistic model, allowing **elliptical or non-linear boundaries** that align more naturally with the data's underlying distribution.

In the K-Means plot: - The **light blue cluster's extension** is a result of the centroid for this cluster being pulled toward the overlapping regions. Consequently, points previously belonging to the **pink** and **light green clusters** are reassigned based on their proximity to this shifted centroid. - The **smaller pink cluster** reflects the more compact reassignment caused by K-Means' inability to model the elongated shape of the original GMM cluster.

4.2 Connection Between Code and K-Means Results

1. **Centroids Initialization**:
   - The GMM means (`mu`) act as the starting point for the centroids in K-Means. However, as K-Means iterates, centroids shift to minimize intra-cluster variance. This shift explains why the light blue cluster absorbs points from other clusters—it moves toward the denser overlapping regions.
2. **Cluster Sizes**:
   - The proportions (`pi`) in the GMM influence initial cluster densities, but K-Means does not explicitly account for proportions. Instead, cluster sizes in K-Means are determined by the density and proximity of points to each centroid.
3. **Shape and Boundaries**:
   - The covariance matrices (`Sigma`) in GMM allow for flexible, elliptical boundaries, whereas K-Means imposes straight-line boundaries. This rigidity in K-Means leads to artificial splits, such as the diagonal and horizontal separations observed in the plot.

4.2.1 Conclusion:

While K-Means captures the general structure of the GMM clusters, it struggles with clusters that overlap significantly or have elongated, non-circular shapes. The differences in the light blue and

pink clusters highlight how K-Means' linear decision boundaries reshape the original clusters, often diminishing accuracy in favor of simplicity.

4.3 2c) Fit a GMM

[20]:
```
def gmm_e(X, model, return_F=False):
    """Perform the E step of EM for a GMM (MLE estimate).

    `model` is a dictionary holding model parameters (keys `mu`, `Sigma`, and `pi`
    defined as in `gmm_gen`).

    Returns a NxK matrix of cluster membership probabilities. If `return_F` is true,
    also returns an NxK matrix holding the density of each data point (row) for each
    component (column).

    """
    mu, Sigma, pi = model["mu"], model["Sigma"], model["pi"]
    N, D = X.shape
    K = len(pi)

    # YOUR CODE HERE

    F = np.zeros((N, K))

    for k in range(K):
        dist = scipy.stats.multivariate_normal(mean=mu[k], cov=Sigma[k])
        F[:, k] = dist.pdf(X)

    W = F * pi
    W /= W.sum(axis=1, keepdims=True)

    # all done
    if return_F:
        return W, F
    else:
        return W
```

[21]:
```
# Test your solution. This should produce:
# (array([[9.99999999e-01, 8.65221693e-10, 1.59675131e-23, 1.14015011e-41, 2.78010004e-65],
#        [1.00000000e+00, 3.75078862e-12, 1.55035521e-23, 1.28102693e-34, 1.86750812e-46],
#        [9.99931809e-01, 6.81161224e-05, 7.23302032e-08, 2.17176125e-09, 1.62736835e-10]]),
```

```
 # array([[1.71811600e-08, 5.94620494e-18, 1.82893598e-31, 9.79455071e-50, 1.
 ↪59217812e-73],
        # [1.44159783e-15, 2.16285148e-27, 1.48999246e-38, 9.23362817e-50, 8.
 ↪97398547e-62],
        # [1.85095927e-09, 5.04355064e-14, 8.92595932e-17, 2.01005787e-18, 1.
 ↪00413265e-19]]))
dummy_model = dict(
    mu=[np.array([k, k + 1]) for k in range(5)],
    Sigma=[np.array([[3, 1], [1, 2]]) / (k + 1) for k in range(5)],
    pi=np.array([0.1, 0.25, 0.15, 0.2, 0.3]),
)
gmm_e(toy_gmm["X"][:3,], dummy_model, return_F=True)
```

[21]:
```
(array([[9.99999999e-01, 8.65221693e-10, 1.59675131e-23, 1.14015011e-41,
         2.78010004e-65],
        [1.00000000e+00, 3.75078862e-12, 1.55035521e-23, 1.28102693e-34,
         1.86750812e-46],
        [9.99931809e-01, 6.81161224e-05, 7.23302032e-08, 2.17176125e-09,
         1.62736835e-10]]),
 array([[1.71811600e-08, 5.94620494e-18, 1.82893598e-31, 9.79455071e-50,
         1.59217812e-73],
        [1.44159783e-15, 2.16285148e-27, 1.48999246e-38, 9.23362817e-50,
         8.97398547e-62],
        [1.85095927e-09, 5.04355064e-14, 8.92595932e-17, 2.01005787e-18,
         1.00413265e-19]]))
```

[22]:
```
def gmm_m(X, W):
    """Perform the M step of EM for a GMM (MLE estimate).

    `W` is the NxK cluster membership matrix computed in the E step. Returns a␣
 ↪new model
    (dictionary with keys `mu`, `Sigma`, and `pi` defined as in `gmm_gen`).

    """
    N, D = X.shape
    K = W.shape[1]

    # YOUR CODE HERE

    # Compute the mixture weights
    pi = np.mean(W, axis=0)

    # Compute the means
    mu = np.dot(W.T, X) / np.sum(W, axis=0)[:, np.newaxis]

    # Compute the covariances
    Sigma = []
```

```
    for k in range(K):
        X_centered = X - mu[k]
        W_diag = np.diag(W[:, k])
        Sigma_k = np.dot(X_centered.T, np.dot(W_diag, X_centered)) / np.sum(W[:
 ↪, k])
        Sigma.append(Sigma_k)

    return dict(mu=mu, Sigma=Sigma, pi=pi)
```

[23]:
```
# Test your solution. This should produce:
# {'mu': [array([ 6.70641574, -0.47971125]),
#   array([8.2353509 , 2.52134815]),
#   array([-3.0476848 , -1.70722161])],
#  'Sigma': [array([[88.09488652, 11.08635139],
#          [11.08635139,  1.39516823]]),
#   array([[45.82761873, 11.38773232],
#          [11.38773232,  6.87352224]]),
#   array([[98.6662729 , 12.41671355],
#          [12.41671355,  1.56258842]])],
#  'pi': array([0.13333333, 0.53333333, 0.33333333])}
gmm_m(toy_gmm["X"][:3,], np.array([[0.1, 0.2, 0.7], [0.3, 0.4, 0.3], [0.0, 1.0,␣
 ↪0.0]]))
```

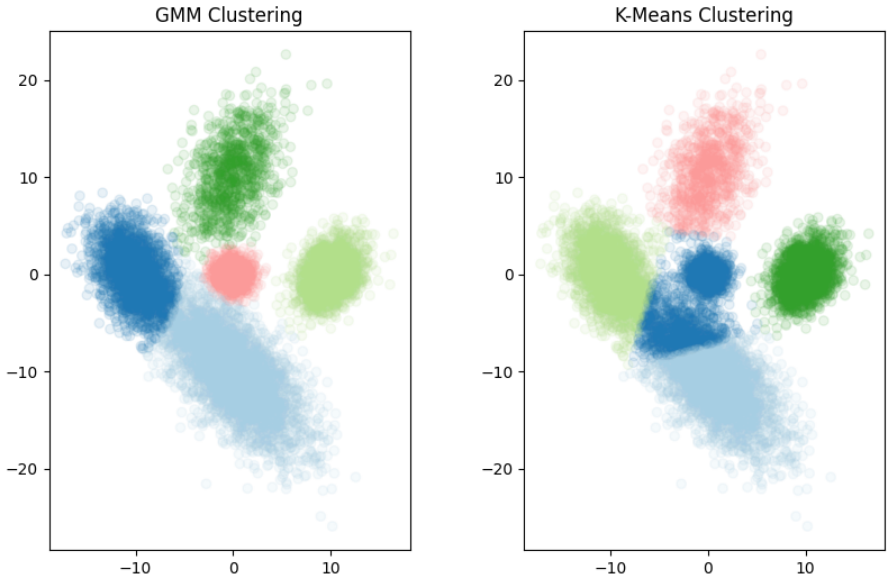
[23]:
```
{'mu': array([[ 6.70641574, -0.47971125],
        [ 8.2353509 ,  2.52134815],
        [-3.0476848 , -1.70722161]]),
 'Sigma': [array([[88.09488652, 11.08635139],
         [11.08635139,  1.39516823]]),
  array([[45.82761873, 11.38773232],
         [11.38773232,  6.87352224]]),
  array([[98.6662729 , 12.41671355],
         [12.41671355,  1.56258842]])],
 'pi': array([0.13333333, 0.53333333, 0.33333333])}
```

[24]:
```
# you do not need to modify this method
def gmm_fit(X, K, max_iter=100, mu0=None, Sigma0=None, pi0=None, gmm_m=gmm_m):
    """Fit a GMM model using EM.

    `K` refers to the number of mixture components to fit. `mu0`, `Sigma0`, and␣
 ↪`pi0`
    are initial parameters (automatically set when unspecified).

    """
    N, D = X.shape

    if mu0 is None:
        mu0 = [np.random.randn(D) for k in range(K)]
```

```
    if Sigma0 is None:
        Sigma0 = [np.eye(D) * 10 for k in range(K)]
    if pi0 is None:
        pi0 = np.ones(K) / K

    model = dict(mu=mu0, Sigma=Sigma0, pi=pi0)
    for it in range(max_iter):
        W = gmm_e(X, model)
        model = gmm_m(X, W)

    return model
```

4.4 2d+2e) Experiment with GMMs for the toy data

[ ]:
```
# Fit on toy data and color each point by most likely component. Also try␣
 ↪fitting with 4
# or 6 components.
toy_gmm_fit = gmm_fit(toy_gmm["X"], 5)

# Predict cluster assignments based on the highest posterior probability
W = gmm_e(toy_gmm["X"], toy_gmm_fit)
cluster_assignments = np.argmax(W, axis=1)

# Assign each data point to its most likely component
W, F = gmm_e(toy_gmm["X"], toy_gmm_fit, return_F=True)
most_likely_component = np.argmax(W, axis=1)

# Create subplots
fig, axes = plt.subplots(1, 2, figsize=(10, 6))

# Plot the dataset with the GMM clustering
axes[0].set_title('GMM Clustering')
plot_xy(toy_gmm["X"][:, 0], toy_gmm["X"][:, 1], most_likely_component, alpha=0.
 ↪1, axis=axes[0])

# Fit 5 clusters using k-means
kmeans = KMeans(5).fit(toy_gmm["X"])
axes[1].set_title('K-Means Clustering')
plot_xy(toy_gmm["X"][:, 0], toy_gmm["X"][:, 1], kmeans.labels_, alpha=0.1,␣
 ↪axis=axes[1])

plt.show()
```



4.4.1 Comparison Between GMM Clustering and K-Means Clustering with k=5 Clusters

4.4.2 GMM Clustering: Description and Key Observations

The GMM clustering results with `k=5` clusters reveal a distinct improvement in capturing the underlying data distribution compared to the K-Means approach. GMM uses a probabilistic model with elliptical decision boundaries, allowing more flexible assignments of data points.

4.4.3 Connection to the EM Algorithm in GMM Clustering

The clustering results from GMM are a direct outcome of the **Expectation-Maximization (EM) algorithm**, which iteratively optimizes the parameters of the Gaussian components to best fit the data. Below is an explanation of how each step in the EM algorithm contributes to the clustering structure observed in the GMM plot:

1. **Initialization**:
   - The initial means (`mu`), covariances (`Sigma`), and proportions (`pi`) are set, often based on heuristics or random initialization. These initial parameters define the starting points of the Gaussian components and directly influence the first set of cluster assignments.
2. **E-Step (Expectation)**:
   - In this step, the algorithm computes the **responsibility matrix**, which represents the probability of each point belonging to each Gaussian component. This step allows for **soft assignments** of points to clusters, meaning a single point can belong partially to multiple clusters.
   - **Observed Effect**: In the plot, this is why edge points, which are probabilistically split between clusters, create smooth and natural boundaries (e.g., the curved boundary

between Clusters 4 and 5).

3. **M-Step (Maximization)**:
   - The parameters (`mu`, `Sigma`, and `pi`) of each Gaussian component are updated to maximize the likelihood of the data given the current responsibilities.
     - The means (`mu`) shift toward the weighted average of the points, influenced by the responsibilities.
     - The covariances (`Sigma`) adapt to fit the spread and orientation of the data points assigned to each cluster.
     - The proportions (`pi`) are updated to reflect the relative size of each cluster.
   - **Observed Effect**: The adjusted cluster shapes and sizes (e.g., the larger Cluster 5 and the reduced Cluster 1) arise from this optimization step, ensuring each Gaussian component fits the data as accurately as possible.
4. **Iterative Refinement**:
   - The EM algorithm alternates between the E-step and M-step until convergence, ensuring the parameters stabilize and the clustering reflects the data's underlying structure.
   - **Observed Effect**: The final boundaries are highly refined, as the iterative process enables the Gaussian components to adapt to even subtle patterns in the data, such as the curved boundary for Cluster 4 and the inclusion of border points in Cluster 3.
5. **Probabilistic Nature of Assignments**:
   - Unlike K-Means, which deterministically assigns each point to the nearest cluster, the EM algorithm allows for probabilistic assignments. This mechanism is evident in the **smooth transitions** between clusters in the GMM plot, avoiding the abrupt boundaries seen in K-Means clustering.

In summary, the EM algorithm provides a robust framework for GMM clustering by iteratively optimizing the Gaussian parameters to capture the true structure of the data. The results observed in the plot—curved boundaries, adjusted cluster sizes, and smoother transitions—are a direct outcome of the probabilistic and iterative nature of the EM algorithm.

1. **Cluster 1 (Centered Around the Origin)**:
   - **Appearance**: This cluster closely resembles its counterpart in the K-Means plot but with **clearer boundaries** in its dense central region. The **size is reduced** compared to the K-Means result.
   - **Reasoning**: GMM probabilistically assigns edge points (which were included in this cluster under K-Means) to other nearby clusters. The probabilistic nature allows these points to contribute partially to other clusters, leading to a **cleaner boundary** and **smaller core size** for this cluster.
2. **Cluster 2 (Right of the Origin)**:
   - **Appearance**: This cluster remains **unchanged** compared to the K-Means result.
   - **Reasoning**: Its distinct separation and well-defined structure in the data mean that the Gaussian component aligns closely with the K-Means centroid. Both methods agree on the boundary and assignment.
3. **Cluster 3 (Top Region)**:
   - **Appearance**: This cluster stretches from -5 to 5 on the x-axis and 5 to 15 on the y-axis. It appears **larger** in the GMM plot compared to K-Means.
   - **Reasoning**: GMM incorporates **border points** that were previously part of Cluster 1 in K-Means. The elliptical boundary adapts to include these points, leading to a more natural assignment and slight expansion of the cluster.
4. **Cluster 4 (Left Region)**:

- **Appearance**: This cluster retains its general position as an extension of the fifth cluster (bottom-left region) but exhibits **curved boundaries** and loses its border with Cluster 1. It now only shares a boundary with Cluster 5.
- **Reasoning**: The curved boundary arises from the **elliptical nature of Gaussian components** in GMM, replacing the linear boundary seen in K-Means. The absence of a boundary with Cluster 1 reflects the reassignment of edge points to more appropriate clusters.

5. **Cluster 5 (Bottom-Left Region)**:
   - **Appearance**: This cluster has **increased in size** compared to its counterpart in K-Means. It now includes points from Cluster 1 that were previously part of the diagonal decision boundary.
   - **Reasoning**: GMM's probabilistic approach leads to a more flexible boundary, allowing points near the bottom-left diagonal region to be reassigned here. The **Gaussian density model** better captures the true spread of these points, leading to an expanded cluster.

4.4.4 Key Differences Between GMM and K-Means Results

1. **Boundary Shape**:
   - GMM uses **elliptical, probabilistic boundaries** defined by covariance matrices (`Sigma`), resulting in smoother and more natural transitions between clusters.
   - K-Means uses **linear boundaries** determined by Euclidean distance, leading to artificial, sharp cuts.
2. **Point Assignments**:
   - GMM assigns points based on their probabilities of belonging to each cluster, allowing for partial contributions. This flexibility avoids hard assignments for edge points and results in more precise boundaries.
   - K-Means assigns points exclusively to the nearest centroid, leading to abrupt transitions and less nuanced boundaries.
3. **Cluster Sizes and Shapes**:
   - GMM clusters better reflect the true density and spread of the data, leading to **size adjustments** (e.g., Cluster 1 shrinking, Cluster 5 expanding) and **shape refinements** (e.g., curved boundaries for Cluster 4).
   - K-Means clusters are constrained by isotropic shapes, leading to uniform spreads and straight-line cuts.

4.4.5 Connection Between Code and GMM Results

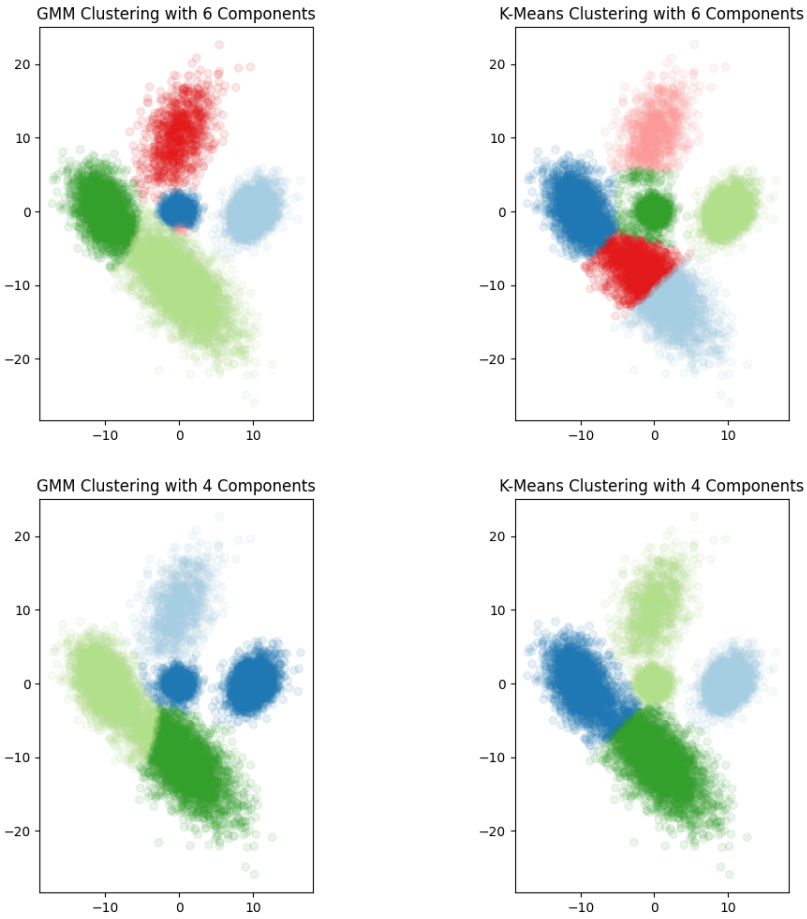
1. **Gaussian Components**:
   - The means (`mu`) define the central points of each cluster in GMM, aligning closely with the cluster centers in the plot.
   - The covariance matrices (`Sigma`) determine the orientation and spread of each cluster, directly influencing the curved, elliptical boundaries visible in the GMM plot.
2. **Cluster Probabilities**:
   - The proportions (`pi`) govern the overall size and weight of each Gaussian component. These proportions ensure the clusters' densities in the plot match the true data distribution.
3. **Point Assignments**:

- GMM probabilistically assigns points to clusters based on their likelihood under each Gaussian component. This mechanism results in the reassignment of edge points (e.g., from Cluster 1 to Clusters 3 and 5), leading to more accurate and natural-looking boundaries.

GMM clustering provides a more flexible and accurate representation of the data compared to K-Means. The elliptical decision boundaries and probabilistic assignments better capture the true structure and relationships between clusters, as evidenced by the cleaner and more natural transitions in the GMM plot.

[35]:
```
# Fit on toy data and color each point by most likely component. Also try␣
 ↪fitting with 4 and 6 components.
toy_gmm_fit_6 = gmm_fit(toy_gmm["X"], 6)
toy_gmm_fit_4 = gmm_fit(toy_gmm["X"], 4)

# Predict cluster assignments based on the highest posterior probability for 6␣
 ↪components
W_6 = gmm_e(toy_gmm["X"], toy_gmm_fit_6)
most_likely_component_6 = np.argmax(W_6, axis=1)

# Predict cluster assignments based on the highest posterior probability for 4␣
 ↪components
W_4 = gmm_e(toy_gmm["X"], toy_gmm_fit_4)
most_likely_component_4 = np.argmax(W_4, axis=1)

# Fit 6 clusters using k-means
kmeans_6 = KMeans(6).fit(toy_gmm["X"])
most_likely_component_kmeans_6 = kmeans_6.labels_

# Fit 4 clusters using k-means
kmeans_4 = KMeans(4).fit(toy_gmm["X"])
most_likely_component_kmeans_4 = kmeans_4.labels_

# Create subplots
fig, axes = plt.subplots(2, 2, figsize=(12, 12))

# Plot the dataset with the GMM clustering for 6 components
axes[0, 0].set_title('GMM Clustering with 6 Components')
plot_xy(toy_gmm["X"][:, 0], toy_gmm["X"][:, 1], most_likely_component_6,␣
 ↪alpha=0.1, axis=axes[0, 0])

# Plot the dataset with the K-Means clustering for 6 components
axes[0, 1].set_title('K-Means Clustering with 6 Components')
plot_xy(toy_gmm["X"][:, 0], toy_gmm["X"][:, 1], most_likely_component_kmeans_6,␣
 ↪alpha=0.1, axis=axes[0, 1])

# Plot the dataset with the GMM clustering for 4 components
```

```
axes[1, 0].set_title('GMM Clustering with 4 Components')
plot_xy(toy_gmm["X"][:, 0], toy_gmm["X"][:, 1], most_likely_component_4,␣
 ↪alpha=0.1, axis=axes[1, 0])

# Plot the dataset with the K-Means clustering for 4 components
axes[1, 1].set_title('K-Means Clustering with 4 Components')
plot_xy(toy_gmm["X"][:, 0], toy_gmm["X"][:, 1], most_likely_component_kmeans_4,␣
 ↪alpha=0.1, axis=axes[1, 1])

plt.show()
```



We have now created the needed algorithm steps of the EM algorithm. Now we want to compare the GMM Clustering with the K means clustering with four and six clusters. Furthermore we want

to say what and why something has changed compared to the cluster with k=5. Therefore I will now only describe the GMM Clustering as well as the k means clustering for k = 4 and k = 6. You have already written an extensive text on the clustering fpr GMM and k means with k = 5 and I want you to refer to it in a seperate section. I then want you to make a comparison between both. What has changes, what is similar and why does the plot look like that. Reference a connection specifically to the GMM Clustering in theory to reason about the following results:

for the GMM with k = 6: there are only five clusters definietely visible. the sixt cluster is only to be guessed since it is so small. the structure of the five clusters look the same as in the GMM with k = 5. Also the decision boundary between the large cluster in the bottom and the one on the left side looks the same. Then there are a few pink points on the "south" of the circle cluster around the origin as well as on the left side of the circle cluster around the origin, directly between the upper and left cluster. But those are really only a few. It looks like the EM Algorithm assigned noise points like these to the sixth cluster.

for the k means with k = 6: There are six equally large clusters visible. Specifically, compared to kmeans with k = 5 the decision boundary of the large cluster in the bottom and the left cluster, which has had included substential parts of the cluster around the origin inbetween them and these diagonal (but linear) decision boundarys, is now interchanged in these parts of the cluster around the origin with the sixth cluster. the decision boundaries are in parallel to each other and split the left cluster and the cluster at the bottom in equal parts. Additionally, since this cluster took parts of the cluster at the origin, which was extended in kmeans k = 5, the cluster around the origin has lost almost half of its size. But it still covers a few noise points of the upper cluster.

for the GMM Algorithm with k = 4: There are four clusters visible. What catches the attention is that the cluster at the bottom and the left cluster have one moved decision boundary compared to the GMM with k = 5 into the south. therefore, now they are equally large and have a decision boundary that is almost vertical, but not really linearly only if you would say it makes sense here. The cluster at the origin and the cluster on the right side in k means, that where both distinct clusters, are now merged into one. Think about a reason on why that might be the case.

for the k means with k = 4: This plot has also four clusters visible. compared to the k means with k = 5 the top and right cluster remain the same while to the top cluster is now also the cluster around the origin added. This cluster around the origin in k = 5 in k means before, was really interesting, since it had two decision boundaries that covered parts of the points on the left and bottom clusters. Now especially these parts that were part of the long data point area going from left to bottom are now reassigned and only one diagonal (linear) decision boundary is visible between the left and the right cluster. the cluster around the origin does not cover parts of it anymore.

4.4.6 Connection Between the EM Algorithm and GMM Clustering for k = 4, 5, 6

The GMM clustering results for different values of `k` are deeply tied to the theoretical properties of the **Expectation-Maximization (EM) algorithm**. Here, we will reason about the observed differences and similarities between the results for `k=4`, `k=5`, and `k=6`, focusing on how the EM algorithm adapts the Gaussian mixture model for different numbers of components:

4.4.7 General Observations Across k Values

1. **Probabilistic Assignments**:

- The EM algorithm computes the responsibility matrix to assign probabilities for each point to belong to each of the `k` clusters. This results in **soft clustering boundaries**, as seen in the curved and smooth decision boundaries in GMM clustering.
- For `k=6`, the tiny sixth cluster that incorporates only noise points demonstrates this process. The EM algorithm assigns these points to the Gaussian component with the highest likelihood, even if the cluster is too small to be visually significant.

2. **Maximization of the Likelihood**:
   - The iterative updates to means (`mu`), covariances (`Sigma`), and weights (`pi`) aim to maximize the overall likelihood of the data under the mixture model. For different values of `k`:
     - With **fewer clusters (`k=4`)**, the model merges clusters (e.g., the origin and right clusters) to simplify the overall structure while maintaining maximum likelihood.
     - With **more clusters (`k=6`)**, the model adds additional Gaussian components to capture finer variations or accommodate outliers, as seen in the sixth cluster.

4.4.8 Specific Reasoning for k = 4, 5, and 6

1. **GMM with k = 4**:
   - **Observed Behavior**:
     - The cluster at the origin (previously a distinct cluster for `k=5`) is now merged with the cluster on the right, forming a single Gaussian component. This new combined cluster spans the origin and extends horizontally, incorporating points that previously belonged to two separate clusters.
     - The boundary between the bottom and left clusters has shifted slightly downward compared to `k=5`. These clusters now appear equally sized, and their decision boundary is approximately vertical but with a subtle curve near the bottom, reflecting the probabilistic assignment.
     - The upper cluster remains unchanged compared to `k=5`, retaining its distinct identity and smooth boundary with neighboring clusters.
   - **Theoretical Explanation**:
     - When `k=4`, the EM algorithm consolidates the clusters to reflect a simplified model. The merging of the origin and right clusters likely occurs because the combined Gaussian better represents the shared density of these points, resulting in a broader coverage for this region. The covariance matrices adjust to ensure the likelihood is maximized while maintaining a reasonable fit for the data distribution.
2. **GMM with k = 5**:
   - **Observed Behavior**:
     - The origin and right clusters are clearly distinct, forming two separate Gaussian components. The cluster at the origin remains circular, while the right cluster retains its elliptical shape with a slight tilt.
     - The boundary between the bottom and left clusters is smooth and curved, following the natural density distribution of the points. These clusters retain their relative sizes from the `k=4` and `k=6` configurations but exhibit slightly more refined boundaries here.
     - The upper cluster is consistent in size and shape with its counterpart in `k=4` and `k=6`, highlighting that this cluster remains well-defined regardless of the number of components.
   - **Theoretical Explanation**:

- At `k=5`, the EM algorithm finds a balance between granularity and model complexity. The origin and right clusters are separated because their densities are distinct enough to warrant individual Gaussians. The covariance adjustments allow the decision boundaries to be smooth and reflect the actual distribution of the data, yielding a model that aligns closely with the natural groupings.

3. **GMM with k = 6**:
   - **Observed Behavior**:
     - The five main clusters from `k=5` remain visibly identical, with no significant changes in their size, shape, or boundaries. The additional sixth cluster is barely noticeable and appears to capture a small set of noise points or outliers.
     - The sixth cluster is most evident in the "south" of the origin cluster and along the left boundary between the origin and left clusters. These points are sparsely distributed and do not significantly alter the structure of the main clusters.
     - The boundary between the bottom and left clusters remains the same as in `k=5`, indicating that the additional Gaussian does not interfere with the structure of the main clusters.
   - **Theoretical Explanation**:
     - With `k=6`, the EM algorithm introduces an additional Gaussian to account for variations not captured by the five-component model. However, since the data does not naturally suggest a sixth cluster, this Gaussian ends up fitting sparse points or noise. The parameters of the main clusters remain largely unchanged because their densities are already well-represented, demonstrating the robustness of the EM algorithm in preserving meaningful structures.

4.4.9 Specific Reasoning for k = 4, 5, and 6

1. **K-Means with k = 4**:
   - **Observed Behavior**:
     - The cluster around the origin, which was distinct in `k=5`, is now merged with the top cluster. This combined cluster is elongated, stretching both horizontally and vertically, and now encompasses all points previously belonging to the origin and top clusters. The decision boundary between this new cluster and the left cluster is diagonal, slanting from the top left toward the bottom right.
     - The bottom and left clusters remain distinct, but their boundaries have shifted compared to `k=5`. The decision boundary between them is now a single diagonal line rather than the two intersecting boundaries seen in `k=5`. This change reflects the redistribution of points from the origin cluster into the other regions.
     - The right cluster remains largely unchanged from `k=5`, retaining its size and shape. Its decision boundaries with the neighboring clusters are sharp and linear, consistent with the rigid structure of K-Means.
   - **Explanation**:
     - In K-Means with `k=4`, the algorithm splits the space into four equally sized and shaped clusters, minimizing the variance within each cluster. The merging of the origin and top clusters into one reflects the need to reduce the number of centroids. The boundaries between clusters are strictly linear, as expected in K-Means, and this leads to some regions appearing artificially partitioned.
2. **K-Means with k = 5**:
   - **Observed Behavior**:

- The cluster at the origin is distinct and circular, but it extends significantly into the regions of the left and bottom clusters. This overlap creates two intersecting linear boundaries between the origin cluster and its neighbors. The decision boundaries are diagonal and straight, segmenting the left and bottom clusters into portions that are assigned to the origin cluster.
- The left and bottom clusters appear stretched and irregular compared to their counterparts in `k=4`. These clusters include long tails of points that extend into regions influenced by the origin cluster.
- The top and right clusters remain well-defined and unchanged from `k=4`. Their boundaries are linear and consistent with the density of the points in their respective regions.

- **Explanation**:
  - K-Means with `k=5` creates a structure where the origin cluster partially encroaches on the left and bottom clusters due to the linear partitioning of space. This results in less natural-looking clusters compared to the GMM approach, as the linear boundaries do not conform to the underlying data distribution.

3. **K-Means with k = 6**:
   - **Observed Behavior**:
     - The addition of a sixth cluster introduces significant changes to the partitioning of space. The origin cluster, which was larger in `k=5`, is now split into two clusters. This results in a much smaller origin cluster, while the new sixth cluster appears in its place, capturing the points that previously belonged to the origin cluster's extended region.
     - The left and bottom clusters are also significantly affected. The new sixth cluster splits these clusters along parallel diagonal lines, creating a series of adjacent regions. The decision boundaries are sharp and evenly spaced, giving the appearance of artificially imposed symmetry.
     - The top and right clusters remain unchanged from `k=4` and `k=5`. Their sharp linear boundaries are unaffected by the introduction of the sixth cluster.
   - **Explanation**:
     - K-Means with `k=6` forces the division of space into six equally weighted clusters, regardless of the natural grouping of the data. The splitting of the origin cluster reflects the algorithm's emphasis on minimizing variance within clusters, even at the cost of creating artificial partitioning. The parallel boundaries between the new clusters demonstrate the inflexibility of K-Means in adapting to irregular data distributions.

4.4.10 Comparison to K-Means Clustering

1. **Boundary Shapes**:
   - K-Means produces **linear decision boundaries**, while GMM generates **smooth, elliptical boundaries** due to the covariance matrices in the Gaussian components. For example, the curved boundary between the bottom and left clusters in GMM contrasts sharply with the linear boundary in K-Means.
2. **Cluster Sizes**:
   - In K-Means, clusters are forced to be of similar size because the algorithm minimizes within-cluster variance without considering density or spread. GMM, on the other hand, adapts cluster sizes and shapes based on the underlying data distribution and Gaussian

densities. This is evident in the unevenly sized clusters in GMM for `k=5` and the small sixth cluster in `k=6`.

3. **Handling of Noise**:
   - GMM assigns noise points probabilistically to the most likely cluster or to small additional clusters (e.g., the sixth cluster for `k=6`). In K-Means, noise points are forced into existing clusters, often distorting their structure.
4. **Changes Across k Values**:
   - K-Means splits and merges clusters in a rigid, linear manner as `k` changes (e.g., the splitting of the origin cluster into multiple parts for `k=6`). GMM adjusts its clusters more fluidly, either by merging Gaussian components (for `k=4`) or adding components for edge cases (for `k=6`).

The differences in GMM clustering for different `k` values highlight the flexibility of the EM algorithm in fitting the data distribution. For fewer clusters (`k=4`), the algorithm merges clusters to maintain overall likelihood, while for more clusters (`k=6`), it introduces new components to capture finer details or outliers. This adaptability leads to natural and meaningful cluster boundaries that align with the theoretical principles of Gaussian mixture modeling.

4.5 2f) Discover the Secret (optional)

[28]:
```
# Load the secret data.
X = np.loadtxt("data/secret_gmm.csv", delimiter=",")
```

[29]:
```
# How many components are hidden in this data?
```