Statistical Machine Learning – Homework

Prof. Dr. Jan Peters
Daniel Palenicek, An Thai Le, Theo Gruner, Maximilian Tölle & Théo Vincent

TECHNISCHE UNIVERSITÄT DARMSTADT

Sommersemester 2023 – Due date: 26.06.2023, 23:59
Sheet 3

Each (sub)task in this homework is worth 1 point. For example, you can get up to 5 points in Task 3.1. The points you achieve in this homework will count towards the exam bonus points.

In this exercise, all tasks are provided in jupyter notebooks. You can either host the jupyter notebooks on your own pc or you can work on them in a google colab. If you decide to work with colab, make sure to copy the provided notebooks before editing. Otherwise, your changes won't be saved!

To submit the homework, print the jupyter notebook as a pdf and concatenate all pdfs in the provided latex template at the end of the file with

```
\includepdf[pages=-]{task_submission/<your-colab.pdf>}
```

Then submit your homework as a single pdf. Find all the tasks at

Group 125: Aoxiong Zhang-24920, Daina Wang-2755971, Qianhui Cui-2299712

Regression

- https://colab.research.google.com/drive/1ArDZN8IwCycHBqeLb2fWKjVQibBePaZk?usp=sharing or
- as a jupyter notebook *linear_regression.ipynb*

Exercise 3.1

Imports

These are all the imports you will need for exercise 3.1. All exercises should be implemented using only the libraries below.

In [1]:
```
import math
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Linear Regression

In this exercise, you will work on linear regression with polynomial features where we model the function $f(\mathbf{x})$ as

$$f(\mathbf{x}) = \mathbf{w}^{\intercal}\phi(\mathbf{x}).$$

The true model is a polynomial of degree 3

$$f(x) = 0.5 + (2x - 0.5)^2 + x^3$$

We further introduce noise into the system by adding a noise term ε_i which is sampled from a Gaussian distribution

$$y = f(x) + \varepsilon_i, \varepsilon_i \sim \mathcal{N}(\varepsilon; 0, \sigma^2).$$

In [2]:
```
"""The true polynomial that generated the data D
  Args:
    x: Input data
  Returns:
    Polynomial evaluated at x
  """
def f(x):
    return x ** 3 + (2 * x - .5) ** 2 + .5

"""Generate the datasets. Note that we don't want to extrapolate,
  and such, the eval data should always lie inside of the train data.
  Args:
    n: Number of datapoints to sample. n has to be atleast 2.
    minval: Lower boundary for samples x
    maxval: Upper boundary for samples x
    variance: Variance or squared standard deviation of the model
    train: Flag deciding whether we sample training or evaluation data
    seed: Random seed
  Returns:
    Tuple of randomly generated data x and the according y
  """
def generate_data(n, minval, maxval, variance=1., train=False, seed=0):
  # Set numpy random number generator
    rng = np.random.default_rng(seed)
  # Sample data along the x-axis
    if train:
        # We first sample uniformly on the x-Axis
        x = rng.uniform(minval, maxval, size=(n - 2,))
        # We will sample on datapoint beyond the min and max values to
        # guarantee that we do not extrapolate during the evaluation
        margin = (maxval - minval) / 100
        min_x = rng.uniform(minval - margin, minval, (1,))
        max_x = rng.uniform(maxval, maxval + margin, (1,))
        x = np.concatenate((x, min_x, max_x))
    else:
        x = rng.uniform(minval, maxval, size=(n,))
    eps = rng.standard_normal(n)
    return x, f(x) + variance * eps
```

Linear Least Squares Regression

In this exercise we will study linear least squares regression with polynomial features. In particular, we want to evaluate the influence of the polynomial degree k that we assume a priori.

Exercise 3.1.1

To carry out regression, we first need to define the basis functions $\phi(\mathbf{x})$. In this task we would like to use polynomial features of degree k.

Please work through the code and fill in the the `# TODO` s.

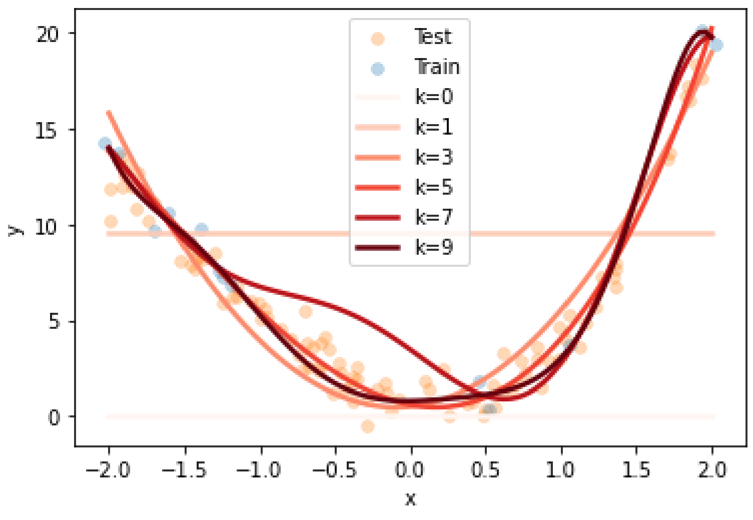
In [3]:
```
"""
  Calculates polynomial features function of degree n.
  The feature function includes all exponents from 0 to n.
  Args:
    x: Input of size (N, D)
    degree: Polynomial degree
  Returns:
    Polynomial features evaluated at x of dim (degree, N)
  """
def polynomial_features(x, degree):
#     N, D = x.shape
    N = x.shape[0]
    features = np.zeros((degree, N))
    for d in range(degree):
        features[d] = np.power(x, d)
    return features

"""
  Fit the weights with the closed-form solution of ridge regression.
  Args:
    x: Input of size (N, D)
    y: Output of size (N,)
    lam: Regularization parameter lambda
    degree: Polynomial degree
  Returns:
    Optimal weights
  """
def fit_w(x, y, lam, degree):
    X = polynomial_features(x, degree)
    A = np.dot(X, X.T)
    lam_I = lam * np.eye(degree)
    b = np.dot(X, y)
    w = np.linalg.solve(A+lam_I, b)
    return w


"""
  Calculate the generalized linear regression estimate given x,
  the feature function, and weights w.
  Args:
    x: input of size (N, D)
    w: Weights of size (M)
    degree: Polynomial degree
  Returns:
    Generalized linear regression estimate
  """
def predict(x, w, degree):
    X = polynomial_features(x, degree)
    y_pred = np.dot(X.T, w)
    return y_pred


"""
  Calculates the mean squared error (MSE) between x and y
  Args:
    x: Data x of size (N,)
    y: Data y of size (N,)
  Returns:
    MSE
  """
def calc_mse(x, y):
    N = len(x)
    mse = np.sum((x - y) ** 2) / N
    return mse
```

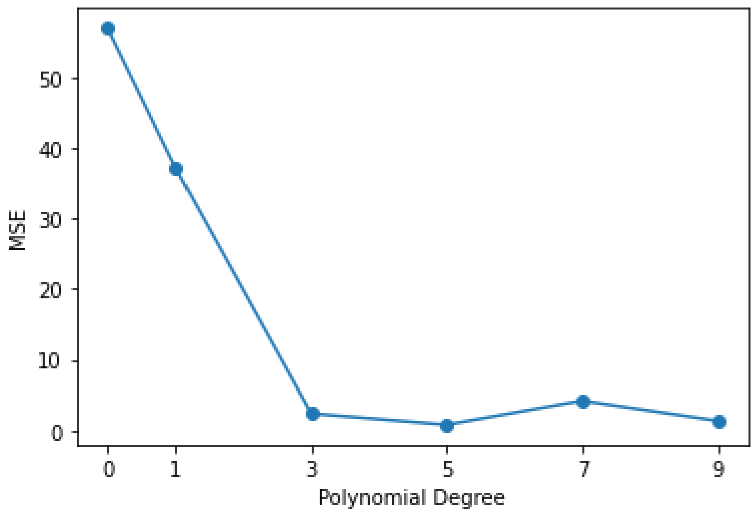
Here you can try out your code by simply running the following cell. This cell will carry out your ridge regression implementation from above for $\lambda = 0$ in which case we are provided with the linear least squares solution.

We evaluate the regression task on six different polynomial sizes $k = \{0, 1, 3, 5, 7, 9\}$ based on your implementation of the MSE.

In [4]:
```
%matplotlib inline

# Settings
n_train = 15
n_test = 100
minval = -2.
maxval = 2



train_data = generate_data(n_train, minval, maxval, train=True, seed=1001)
test_data = generate_data(n_test, minval, maxval, train=False, seed=1002)

"""Plotting functionality for the prediction of linear regression
  for K different polynomial degrees.
  Args:
    x: Data of size (K, N). The first dimension denotes the different
      polynomial degrees that has been experimented with
    y: Data of size (K, N)
  """
def plot_linear_regression(x, y, labels, eval_quantity):
    K = x.shape[0]
```

```
#     colors = mpl.colormaps['Reds'].resampled(K+1)(range(1, K+1))
    colors = plt.cm.Reds(np.linspace(0, 1, K))
    fig = plt.figure()
    plt.scatter(test_data[0], test_data[1], color="tab:orange", linewidths=0.5, lal
    plt.scatter(train_data[0], train_data[1], color="tab:blue", linewidths=0.5, lal
    for i in range(K):
        plt.plot(x[i], y[i], label=f"{eval_quantity}={labels[i]}", color=colors[i]
    plt.xlabel("x")
    plt.ylabel("y")
    plt.legend()

"""Plotting functionality of the MSE for K different polynomial degrees."""
def plot_mse(mse, labels):
    fig = plt.figure()
    plt.plot(labels, mse)
    plt.scatter(labels, mse)
    plt.xticks(labels)
    plt.ylabel("MSE")
    plt.xlabel("Polynomial Degree")

# Evaluate regression for different polynomial degrees
degrees = [0, 1, 3, 5, 7, 9]
xs, ys, mse = [], [], []
for degree in degrees:
    w = fit_w(train_data[0], train_data[1], lam=0., # Edge case resulting in linear
              degree=degree)
    # Predict the test data
    y_test = predict(test_data[0], w, degree)
    mse.append(calc_mse(y_test, test_data[1]))
    # Run regression over the whole interval
    xs.append(np.linspace(minval, maxval, 100))
    ys.append(predict(xs[-1], w, degree))
xs = np.stack(xs)
ys = np.stack(ys)

plot_linear_regression(xs, ys, labels=degrees, eval_quantity="k")
plot_mse(mse, degrees)
```





Exercise 3.1.2

Please describe your results below in a few lines thereby answering which model you would choose and which phenomenon we see for small and large polynomial degrees.

I will choose the model k=5, From the plot, we can see that the MSE decreases as the polynomial degree increases initially but then starts to increases after k=5. k>5, overfitting k<5, uderfitting

---

Bias Variance Tradeoff

Next up, we will compare the model performance of **ridge regression** based on the penalty parameter λ. For that we will evaluate the expected squared error of the true model against our predictions. As we have shown in the lecture, this leads to the bias-variance decomposition

$$L_{\hat{f}}(\mathbf{x}_q) = \mathbb{E}_{\mathcal{D},\varepsilon}\left[\left(y(\mathbf{x}_q) - \hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)^2\right] = \sigma^2 + \text{bias}^2\left[\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right] + \text{var}\left[\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right]$$

Here $\hat{f}_{\mathcal{D}}$ denotes the function estimator trained on the data $\mathcal{D} = \{(y_i, \mathbf{x_i}) \mid i = 1, \dots, N\}$. We have left the two following identities open in the lecture which are required to arrive at the above equation

$$\begin{aligned}\mathbb{E}_{\mathcal{D},\varepsilon}\left[\varepsilon\left(f(\mathbf{x}_q) - \hat{f}_{\mathcal{D}}(\mathbf{x_q})\right)\right] &= 0 \\ \mathbb{E}_{\mathcal{D}}\left[\left(f(\mathbf{x}_q) - \hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)^2\right] &= \left(f(\mathbf{x}_q) - \bar{\hat{f}}(\mathbf{x}_q)\right)^2 + \mathbb{E}_{\mathcal{D}}\left[\left(\bar{\hat{f}}(\mathbf{x}_q) - \hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)^2\right]\end{aligned}$$

Here, the notation is simplified by adding the variable $\bar{\hat{f}}(\mathbf{x}_q) = \mathbb{E}_{\mathcal{D}}\left[\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right]$.

Exercise 3.1.3

Please show the two identities

1. $\mathbb{E}_{\mathcal{D},\varepsilon}\left[\varepsilon\left(f(\mathbf{x}_q) - \hat{f}_{\mathcal{D}}(\mathbf{x_q})\right)\right] = 0$

2. $\mathbb{E}_{\mathcal{D}}\left[\left(f(\mathbf{x}_q)-\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)^2\right]=\left(f(\mathbf{x}_q)-\bar{\hat{f}}(\mathbf{x}_q)\right)^2+\mathbb{E}_{\mathcal{D}}\left[\left(\bar{\hat{f}}(\mathbf{x}_q)-\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)^2\right]$

`TODO: Your answer here`

Identity 1:

$$\mathbb{E}_{\mathcal{D},\varepsilon}\left[\varepsilon\left(f(\mathbf{x}_q)-\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)\right]=0$$

To prove this identity, we can expand the expectation:

$$\mathbb{E}_{\mathcal{D},\varepsilon}\left[\varepsilon\left(f(\mathbf{x}_q)-\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)\right]=\int\int\varepsilon\left(f(\mathbf{x}_q)-\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)p(\mathcal{D})p(\varepsilon)\mathrm{d}\mathcal{D}\mathrm{d}\varepsilon$$

We can rearrange the terms in the integral:

$$\mathbb{E}_{\mathcal{D},\varepsilon}\left[\varepsilon\left(f(\mathbf{x}_q)-\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)\right]=\int\varepsilon\left(\int\left(f(\mathbf{x}_q)-\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)p(\mathcal{D})\mathrm{d}\mathcal{D}\right)p(\varepsilon)\mathrm{d}\varepsilon$$

Now, let's focus on the inner integral:

$$\int\left(f(\mathbf{x}_q)-\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)p(\mathcal{D})\mathrm{d}\mathcal{D}$$

This integral represents the expected difference between the true function $f(\mathbf{x}_q)$ and the estimated function $\hat{f}_{\mathcal{D}}(\mathbf{x}_q)$, averaged over all possible training datasets $\mathcal{D}$. By definition, the expected value of this difference is zero. Therefore, the inner integral evaluates to zero, resulting in:

$$\mathbb{E}_{\mathcal{D},\varepsilon}\left[\varepsilon\left(f(\mathbf{x}_q)-\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)\right]=\int\varepsilon\cdot 0\cdot p(\varepsilon)\mathrm{d}\varepsilon\qquad=0$$

This proves Identity 1.

Identity 2:

$$\mathbb{E}_{\mathcal{D}}\left[\left(f(\mathbf{x}_q)-\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)^2\right]=\left(f(\mathbf{x}_q)-\bar{\hat{f}}(\mathbf{x}_q)\right)^2+\mathbb{E}_{\mathcal{D}}\left[\left(\bar{\hat{f}}(\mathbf{x}_q)-\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)^2\right]$$

To prove this identity, let's expand the left-hand side expectation:

$$\mathbb{E}_{\mathcal{D}}\left[\left(f(\mathbf{x}_q)-\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)^2\right]=\int\left(f(\mathbf{x}_q)-\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)^2p(\mathcal{D})\mathrm{d}\mathcal{D}$$

Now, let's focus on the term $\left(f(\mathbf{x}_q)-\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)^2$ inside the integral. We can rewrite this term as follows:

$$\left(f(\mathbf{x}_q)-\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)^2=\left(f(\mathbf{x}_q)-\bar{\hat{f}}(\mathbf{x}_q)+\bar{\hat{f}}(\mathbf{x}_q)-\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)^2$$

Using the binomial expansion, we have:

$$\left(f(\mathbf{x}_q)-\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)^2=\left(f(\mathbf{x}_q)-\bar{\hat{f}}(\mathbf{x}_q)\right)^2+2\left(f(\mathbf{x}_q)-\bar{\hat{f}}(\mathbf{x}_q)\right)\left(\bar{\hat{f}}(\mathbf{x}_q)-\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)+\left(\bar{\hat{f}}\right.$$

Taking the expectation with respect to $\mathcal{D}$, we get:

$$\mathbb{E}_{\mathcal{D}}\left[\left(f(\mathbf{x}_q) - \hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)^2\right] = \mathbb{E}_{\mathcal{D}}\left[\left(f(\mathbf{x}_q) - \bar{\hat{f}}(\mathbf{x}_q)\right)^2\right] + 2\mathbb{E}_{\mathcal{D}}\left[\left(f(\mathbf{x}_q) - \bar{\hat{f}}(\mathbf{x}_q)\right)\left(\bar{\hat{f}}(\mathbf{x}_q) -$$

The first and third terms on the right-hand side are the variances of the expected value of the estimated function and the variance of the estimated function, respectively, which can be denoted as follows:

$$\mathbb{E}_{\mathcal{D}}\left[\left(f(\mathbf{x}_q) - \bar{\hat{f}}(\mathbf{x}_q)\right)^2\right] = \left(f(\mathbf{x}_q) - \bar{\hat{f}}(\mathbf{x}_q)\right)^2$$
$$\mathbb{E}_{\mathcal{D}}\left[\left(\bar{\hat{f}}(\mathbf{x}_q) - \hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)^2\right] = \mathrm{Var}\left[\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right]$$

Now, let's focus on the second term:

$$2\mathbb{E}_{\mathcal{D}}\left[\left(f(\mathbf{x}_q) - \bar{\hat{f}}(\mathbf{x}_q)\right)\left(\bar{\hat{f}}(\mathbf{x}_q) - \hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)\right]$$

We can expand this term as follows:

$$2\mathbb{E}_{\mathcal{D}}\left[\left(f(\mathbf{x}_q) - \bar{\hat{f}}(\mathbf{x}_q)\right)\left(\bar{\hat{f}}(\mathbf{x}_q) - \hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)\right] = 2\mathbb{E}_{\mathcal{D}}\left[\bar{\hat{f}}(\mathbf{x}_q)\left(f(\mathbf{x}_q) - \hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)\right] - 2\mathbb{E}_{\mathcal{D}}\left[\bar{\hat{f}}$$

The first term can be written as:

$$2\mathbb{E}_{\mathcal{D}}\left[\bar{\hat{f}}(\mathbf{x}_q)\left(f(\mathbf{x}_q) - \hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)\right] = 2\bar{\hat{f}}(\mathbf{x}_q)\mathbb{E}_{\mathcal{D}}\left[\left(f(\mathbf{x}_q) - \hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)\right] \qquad = 2\bar{\hat{f}}(\mathbf{x}_q)\cdot 0 \ =$$

For the second term, we have:

$$2\mathbb{E}_{\mathcal{D}}\left[\bar{\hat{f}}(\mathbf{x}_q)\left(\bar{\hat{f}}(\mathbf{x}_q) - \hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)\right] = 2\bar{\hat{f}}(\mathbf{x}_q)\mathbb{E}_{\mathcal{D}}\left[\bar{\hat{f}}(\mathbf{x}_q) - \hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right] \qquad = 2\bar{\hat{f}}(\mathbf{x}_q)\left(\bar{\hat{f}}(\mathbf{x}_q)\right.$$

Therefore, the second term also evaluates to zero. Putting it all together, we have:

$$\mathbb{E}_{\mathcal{D}}\left[\left(f(\mathbf{x}_q) - \hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right)^2\right] = \left(f(\mathbf{x}_q) - \bar{\hat{f}}(\mathbf{x}_q)\right)^2 + \mathrm{Var}\left[\hat{f}_{\mathcal{D}}(\mathbf{x}_q)\right]$$

This proves Identity 2.

The bias-variance tradeoff is typically a purely theoretical concept as it requires the evaluation of $f(x)$. In this task we assume that $f(x)$ is known and thus, an approximation of the bias and variance is possible. We approximatie the bias and variance by its sample means

$$\mathrm{Bias}\,\mathrm{bias}^2[\hat{f}_{\mathcal{D}}] \approx \frac{1}{N}\sum_{i=1}^{N}\left(f(x_i) - \bar{\hat{f}}(x_i)\right)^2,$$

$$\mathrm{Var}\,\mathrm{var}\left[\hat{f}_{\mathcal{D}}\right] \approx \frac{1}{NM}\sum_{i=1}^{N}\sum_{j=1}^{M}\left(\hat{f}_{\mathcal{D}_j}(x_i) - \bar{\hat{f}}(x_i)\right)^2$$

Here, $\bar{\hat{f}}(x_i)$ is the average prediction of the maximum likelihood over the data distribution $p(\mathcal{D})$ which we approximate given M datasets $\mathcal{D}_j$

$$\bar{\hat{f}}(x_i) \approx \frac{1}{M}\sum_{j=1}^{M}\left(\hat{f}_{\mathcal{D}_j}(x_i)\right).$$

To approximate the bias and variance, we first evaluate the maximum likelihood estimate $f_{\mathcal{D}_j}$ for each dataset $\mathcal{D}_j$. Afterwards we can approximate the two terms.

Exercise 3.1.4

In this exercise we implement the average prediction $\bar{\hat{f}}(x_i)$, $\text{Bias}\ \text{bias}^2[\hat{f}_{\mathcal{D}}]$, and $\text{Var}\ \text{var}\left[\hat{f}_{\mathcal{D}}\right]$ as introduced above.

Please work through the code and fill in the the `# TODO` s.

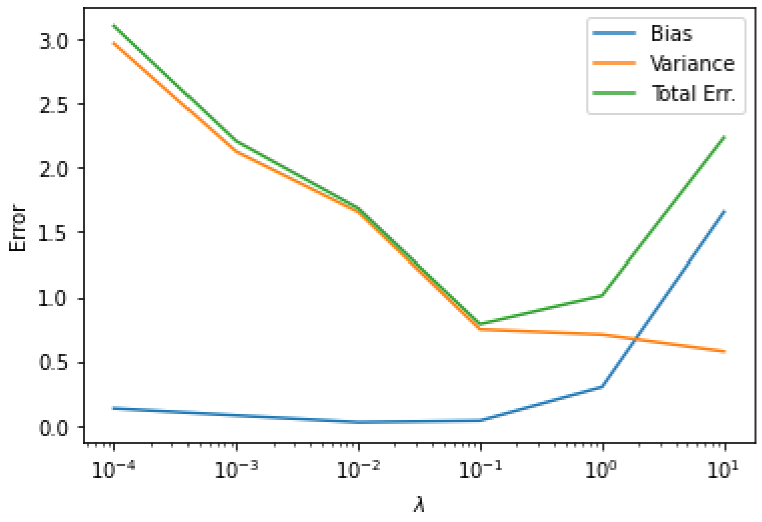
```
In [17]: def avg_prediction(x, ws, degree=3):
             """
           Approximation of the average prediction using the M function estimations
           Args:
             x: input data of size (N,)
             ws: The weights obtained from ridge regression of size (M, degree)
             degree: The polynomial degree
           Returns:
             The average prediction as a scalar
           """
             N = len(x)
             M = ws.shape[0]
             predictions = np.zeros((M, N))

             for i in range(M):
                 w = ws[i]
                 predictions[i] = predict(x, w, degree)

             avg_pred = np.mean(predictions, axis=0)
             return np.mean(avg_pred)

         def calc_bias(x_q, ws, degree):
             """
             Estimate the bias.
           Args:
             x_q: Queries x of size (N,)
             ws: The weights obtained from ridge regression of size (M, degree)
             degree: The polynomial degree
           Returns:
             Bias
           """
             N = len(x_q)
             M = len(ws)  # Use len(ws) instead of ws.shape[0]
             ws_array = np.array(ws)  # Convert ws to a NumPy array
             predictions = np.zeros((M, N))

             for i in range(M):
                 w = ws_array[i]
                 predictions[i] = predict(x_q, w, degree)

             avg_pred = np.mean(predictions, axis=0)
             true_values = f(x_q)
```

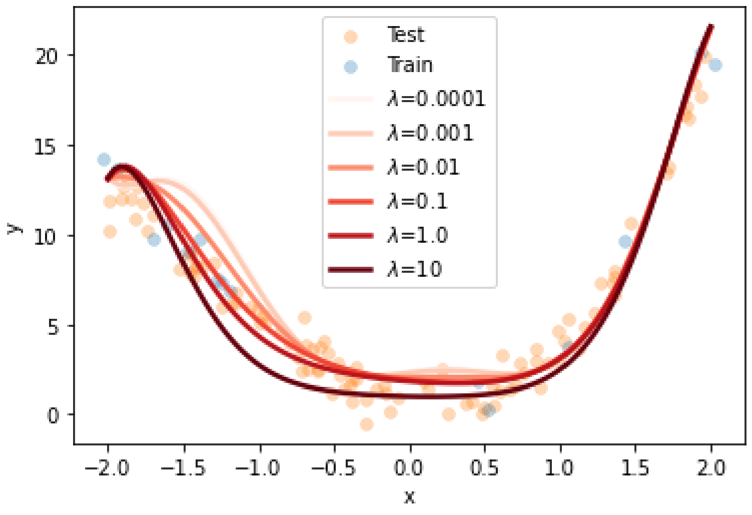
```
    bias = np.mean((avg_pred - true_values) ** 2)
    return bias

def calc_variance(x_q, ws, degree):
    """
  Estimate the model variance
  Args:
    x_q: Queries x of size (N,)
    ws: The weights obtained from ridge regression of size (M, degree)
    degree: The polynomial degree
  Returns:
    Model variance
  """
    N = len(x_q)
    M = len(ws)  # Use len(ws) instead of ws.shape[0]
    ws_array = np.array(ws)  # Convert ws to a NumPy array
    predictions = np.zeros((M, N))

    for i in range(M):
        w = ws_array[i]
        predictions[i] = predict(x_q, w, degree)

    avg_pred = np.mean(predictions, axis=0)
    variance = np.mean((predictions - avg_pred) ** 2)
    return variance
```

You can test your implementation by running the below coding snippet. It estimate the bias and variance for $M = 25$ datasets with each dataset containing $N = 20$ datapoints.

In [18]:
```
%matplotlib inline

# Settings
n = 20
m = 25
degree = 9
train_datasets = []
seed = 3001
for i in range(m):
    train_datasets.append(generate_data(n_train, minval, maxval, train=True, seed=
    seed += 1
eval_points = np.linspace(minval, maxval, n)

# Estimate the bias and variance
biases = []
vars = []
xs, ys = [], []
lambdas = [0.0001, 0.001, 0.01, 0.1, 1., 10]
for l in lambdas:
    w_maps = []
    for data in train_datasets:
        w = fit_w(data[0], data[1], l, 9)
        w_maps.append(w)
    bias = calc_bias(eval_points, w_maps, degree)
    biases.append(bias)
    var = calc_variance(eval_points, w_maps, degree)
    vars.append(var)
    xs.append(np.linspace(minval, maxval, 100))
    ys.append(predict(xs[-1], w_maps[0], degree))

biases = np.array(biases)
vars = np.array(vars)
xs = np.stack(xs)
```

```python
ys = np.stack(ys)

# Plot the bias and variance for different lambas
plt.figure()
plt.plot(lambdas, biases, label="Bias")
plt.plot(lambdas, vars, label="Variance")
plt.plot(lambdas, biases + vars, label="Total Err.")
plt.xscale("log")
plt.xlabel(r"$\lambda$")
plt.ylabel("Error")
plt.legend()

# Calculate predictions
plot_linear_regression(xs, ys, labels=lambdas, eval_quantity=r"$\lambda$")
```





Exercise 3.1.5

Please explain the results in a few sentences. In particular, provide an explanation if the bias and variance behave as expected. For which regularization parameter λ would you decide?

I will choose $\lambda = 0.1$.\ Bias: The bias initially decreases as λ increases but then starts to increase again.\ Variance: The variance decreases as λ increases.\ Total Error: The total error, which is the sum of bias and variance, exhibits a U-shaped curve as a function of lambda. It initially decreases as lambda increases, reaching a minimum, and then starts to increase again. This behavior suggests a trade-off between bias and variance.

Gradient Descent

In the lecture we have seen that the closed form solution of linear regression requires us to take the inverse $(\mathbf{\Phi}\mathbf{\Phi}^\intercal)^{-1}$. For high dimensional features, the inverse can be a high computational burden. For these reasons, gradient descent provides an alternative to approximate the weight vector.

Exercise 3.1.6

Please implement gradient descent optimization to find the regression weights $\mathbf{w}$. We will use the loss from linear least squares with polynomial features of degree $k = 3$

$$\mathcal{J}(\mathbf{w}) = ||\mathbf{\Phi}^\intercal\mathbf{w} - \mathbf{y}||^2.$$

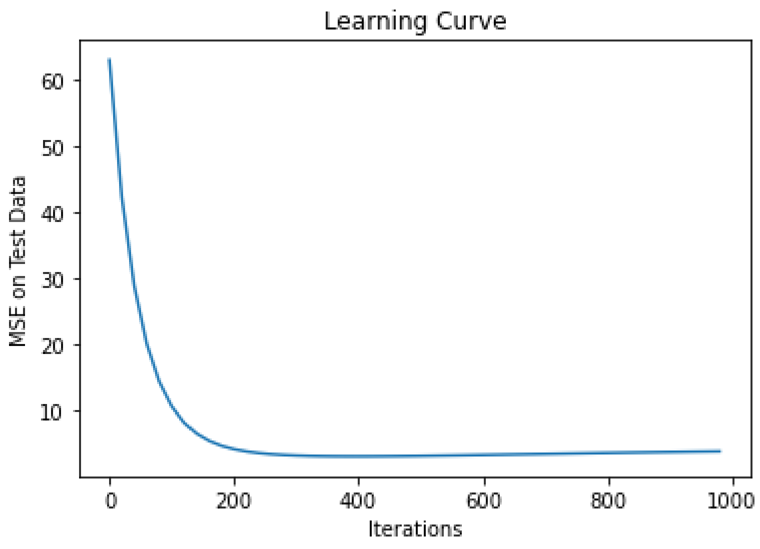
The number of gradient updates is fixed to $n_{\text{iter}} = 1000$. The learning rate can be freely chosen, but a good initial value is lr=0.0001. Please update the gradient by using all the training data points n_{train}, i.e., no mini-batches.

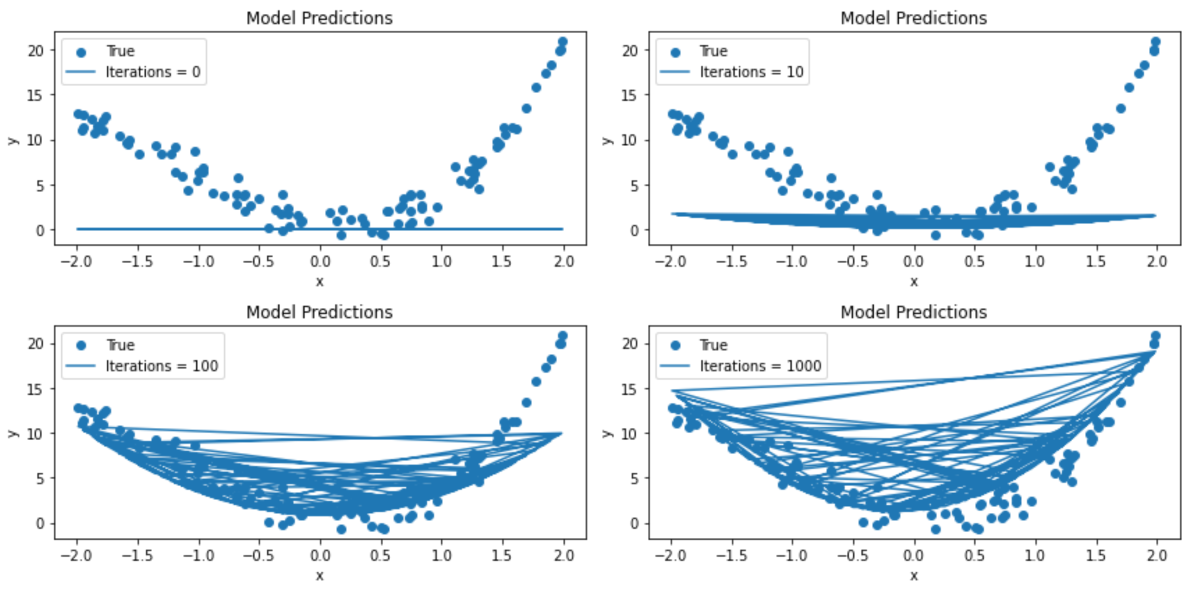
We expect you to provide a plot of the learning curve, i.e., a plot of the MSE on the test data against the iterations. You can evaluate your model after $n_{\text{eval}} = 20$ gradient updates. We further would like to see the model prediction after $n = 0, 10, 100, 1000$ gradient updates/iterations.

In this task we expect you to provide the full code. Note that you are allowed to use all functions defined above.

In [22]:
```python
%matplotlib inline

# Settings
n_train = 15
n_test = 100
minval = -2.
maxval = 2
degree = 3


train_data = generate_data(n_train, minval, maxval, train=True, seed=4001)
test_data = generate_data(n_test, minval, maxval, train=False, seed=4002)
```

```
# Gradient Descent Optimization
def gradient_descent(x_train, y_train, lr, n_iter, n_eval, degree):
    # Initialize weights
    w = np.zeros(degree)

    # Track MSE on test data during training
    mse_test = []

    # Perform gradient updates
    for i in range(n_iter):
        # Compute predictions
        y_pred = predict(x_train, w, degree)

        # Compute gradients
        grad = np.dot(polynomial_features(x_train, degree), (y_pred - y_train))

        # Update weights
        w -= lr * grad

        # Evaluate model every n_eval iterations
        if i % n_eval == 0:
            # Calculate MSE on test data
            y_test_pred = predict(test_data[0], w, degree)
            mse = calc_mse(test_data[1], y_test_pred)
            mse_test.append(mse)

    return w, mse_test

# Perform gradient descent optimization
learning_rate = 0.0001
num_iterations = 1000
eval_interval = 20
w_opt, mse_test = gradient_descent(train_data[0], train_data[1], learning_rate, nu

# Plot learning curve
plt.plot(range(0, num_iterations, eval_interval), mse_test)
plt.xlabel("Iterations")
plt.ylabel("MSE on Test Data")
plt.title("Learning Curve")
plt.show()

# Model predictions after different numbers of iterations
n_updates = [0, 10, 100, 1000]
predictions = []
for n in n_updates:
    w_pred = gradient_descent(train_data[0], train_data[1], learning_rate, n, 1, d
    predictions.append(predict(test_data[0], w_pred, degree))

# Plot model predictions
plt.figure(figsize=(12, 6))
for i, n in enumerate(n_updates):
    plt.subplot(2, 2, i+1)
    plt.scatter(test_data[0], test_data[1], label="True")
    plt.plot(test_data[0], predictions[i], label=f"Iterations = {n}")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.legend()
    plt.title("Model Predictions")

plt.tight_layout()
plt.show()
```


Learning Curve
MSE on Test Data
Iterations
0
200
400
600
800
1000
10
20
30
40
50
60


Model Predictions
True
Iterations = 0
Iterations = 10
Iterations = 100
Iterations = 1000
x
y

Homework 3/4
LastName, FirstName: ______________________________ EnrollmentID: ⌴⌴⌴⌴⌴⌴⌴

Gaussian Processes

- https://colab.research.google.com/drive/1nUkGI9co8pY3BgISzbxWNuJkGdNBzDVa?usp=sharing or
- in the attached jupyter notebook *gp_with_numpy.ipynb*

Exercise 3.2

Utility function for nicely plotting GPs

In [3]:
```
import numpy as np
import matplotlib.pyplot as plt

from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D

def plot_gp(mu, cov, X, X_train=None, Y_train=None, samples=[]):
    '''
    This function plots 95% confidence interval of GP given its parameters. It
    also plots data points and function samples if provided.

    Args:
        mu: mean values (n, d).
        cov: kernel matrix (n, n).
        X_train: training data points (m, d).
        Y_train: training data labels (m, 1).
        samples: list of function samples ([n, d]).
    '''
    X = X.ravel()
    mu = mu.ravel()
    uncertainty = 1.96 * np.sqrt(np.diag(cov))

    plt.fill_between(X, mu + uncertainty, mu - uncertainty, alpha=0.1)
    plt.plot(X, mu, label='Mean')
    for i, sample in enumerate(samples):
        plt.plot(X, sample, lw=1, ls='--', label=f'Sample {i+1}')
    if X_train is not None:
        plt.plot(X_train, Y_train, 'rx')
    plt.legend()
```

In [4]:
```
np.random.seed(11111)  # fixed seed for grading
```

Gaussian Process Implementation with NumPy

In this task, we will use the squared exponential kernel, also known as Gaussian kernel or RBF kernel:

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \exp(-\frac{1}{2l^2}(\mathbf{x}_i - \mathbf{x}_j)^T(\mathbf{x}_i - \mathbf{x}_j))$$

The length parameter l controls the smoothness of the function and σ_f the vertical variation. For simplicity, we use the same length parameter l for all input dimensions (isotropic kernel).

Exercise 3.2.1

Implement the isotropic Gaussian kernel in the function below.

In [5]:
```
import numpy as np

def kernel(X1, X2, l=1.0, sigma_f=1.0):
    '''
    Isotropic squared exponential kernel. Computes
    a covariance matrix from points in X1 and X2.

    Args:
        X1: Array of m points (m, d).
        X2: Array of n points (n, d).

    Returns:
        Covariance matrix (m, n).
    '''
    sq_dist = np.sum((X1[:, np.newaxis] - X2) ** 2, axis=-1)
    k = sigma_f ** 2 * np.exp(-0.5 * sq_dist / (l ** 2))
    return k
```

There are many other kernels for Gaussian processes. For example, please see the scikit-learn documentation for some kernel examples.

Prior

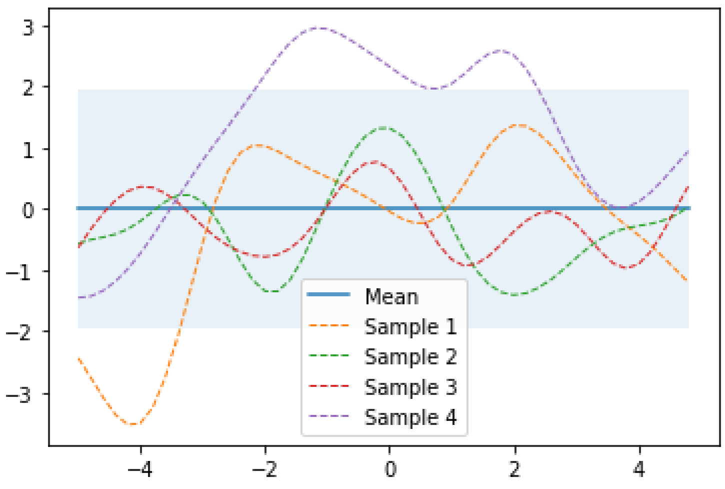
Let us first define a prior over functions with mean zero and a covariance matrix computed with kernel parameters $l = 1$ and $\sigma_f = 1$. Now, to see the random functions that can be sampled from this GP, we draw random samples from the corresponding multivariate normal.

Exercise 3.2.2

Draw four random samples and plots it together with the zero mean and the 95% confidence interval (computed from the diagonal of the covariance matrix).

Hint: use `np.random.multivariate_normal`.

In [6]:
```
%matplotlib inline

# Finite number of points
X = np.arange(-5, 5, 0.2).reshape(-1, 1)

l=1.0
sigma_f=1.0

cov = kernel(X, X, l, sigma_f)

# Sample four random functions from the GP
num_samples = 4
samples = np.random.multivariate_normal(np.zeros(X.shape[0]), cov, num_samples)

# Plot the GP with random samples using the plot_gp function
plot_gp(np.zeros(X.shape[0]), cov, X, samples=samples)
```



Prediction from training data

In the following tasks, we will learn the `sine` function with GP and do regression tasks for data generated from both noise-free case

$$\mathbf{y} = f(\mathbf{x}) = \sin(\mathbf{x})$$

and noisy case

$$\mathbf{y} = f(\mathbf{x}) = \sin(\mathbf{x}) + \epsilon,\ \epsilon \sim \mathcal{N}(\epsilon; \mathbf{0}, \mathbf{I}).$$

In the lecture, we have seen shown how to compute the sufficient statistics, i.e., the mean and the covariance of the posterior predictive distribution for a new data point. Now, in this task, we generalize the posterior predictive distribution for multiple new data points.

A GP prior $p(\mathbf{f}|\mathbf{X})$ can be converted into a GP posterior $p(\mathbf{f}|\mathbf{X}, \mathbf{y})$ after having observed some data $\mathbf{y}$. The posterior can then be used to make predictions $\mathbf{f}_*$ given new input $\mathbf{X}_*$:

$$\begin{aligned} p(\mathbf{f}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{y}) &= \int p(\mathbf{f}_*|\mathbf{X}_*, \mathbf{f}) p(\mathbf{f}|\mathbf{X}, \mathbf{y})\ d\mathbf{f} \\ &= \mathcal{N}(\mathbf{f}_*|\boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*) \end{aligned}$$

By definition of the GP, the joint distribution of observed data $\mathbf{y}$ and predictions $\mathbf{f}_*$ is

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{f}_* \end{pmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{pmatrix} \mathbf{K}_y & \mathbf{K}_* \\ \mathbf{K}_*^T & \mathbf{K}_{**} \end{pmatrix}\right)$$

With N training data and N_* new input data, $\mathbf{K}_y = \kappa(\mathbf{X}, \mathbf{X}) + \sigma_y^2\mathbf{I} = \mathbf{K} + \sigma_y^2\mathbf{I}$ is $N \times N$, $\mathbf{K}_* = \kappa(\mathbf{X}, \mathbf{X}_*)$ is $N \times N_*$ and $\mathbf{K}_{**} = \kappa(\mathbf{X}_*, \mathbf{X}_*)$ is $N_* \times N_*$. σ_y^2 is the noise term in the diagonal of $\mathbf{K}_\mathbf{y}$. It is set to zero if training targets are noise-free and to a value greater than zero if training observations are noisy. The mean is set to $\mathbf{0}$ for notational simplicity. The sufficient statistics of the posterior predictive distribution, $\boldsymbol{\mu}_*$ and $\boldsymbol{\Sigma}_*$, can be computed by

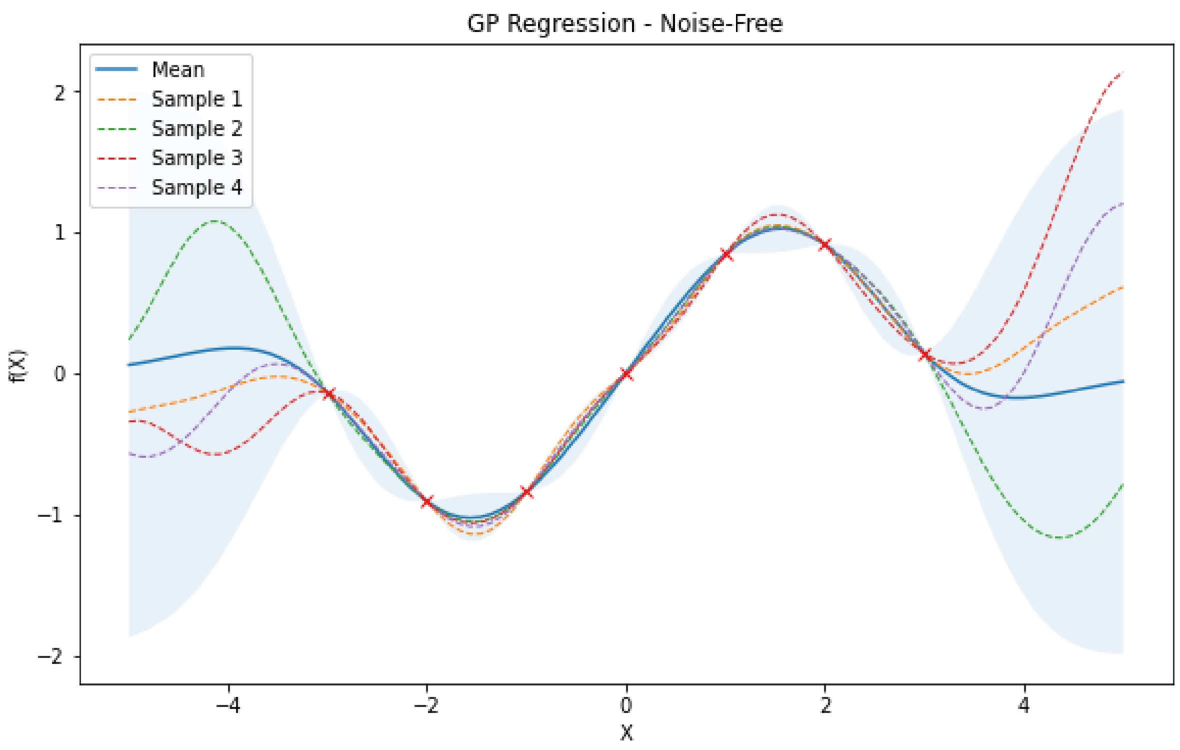
$$\begin{aligned} \boldsymbol{\mu}_* &= \mathbf{K}_*^T\mathbf{K}_y^{-1}\mathbf{y} \\ \boldsymbol{\Sigma}_* &= \mathbf{K}_{**} - \mathbf{K}_*^T\mathbf{K}_y^{-1}\mathbf{K}_* \end{aligned}$$

Exercise 3.2.3

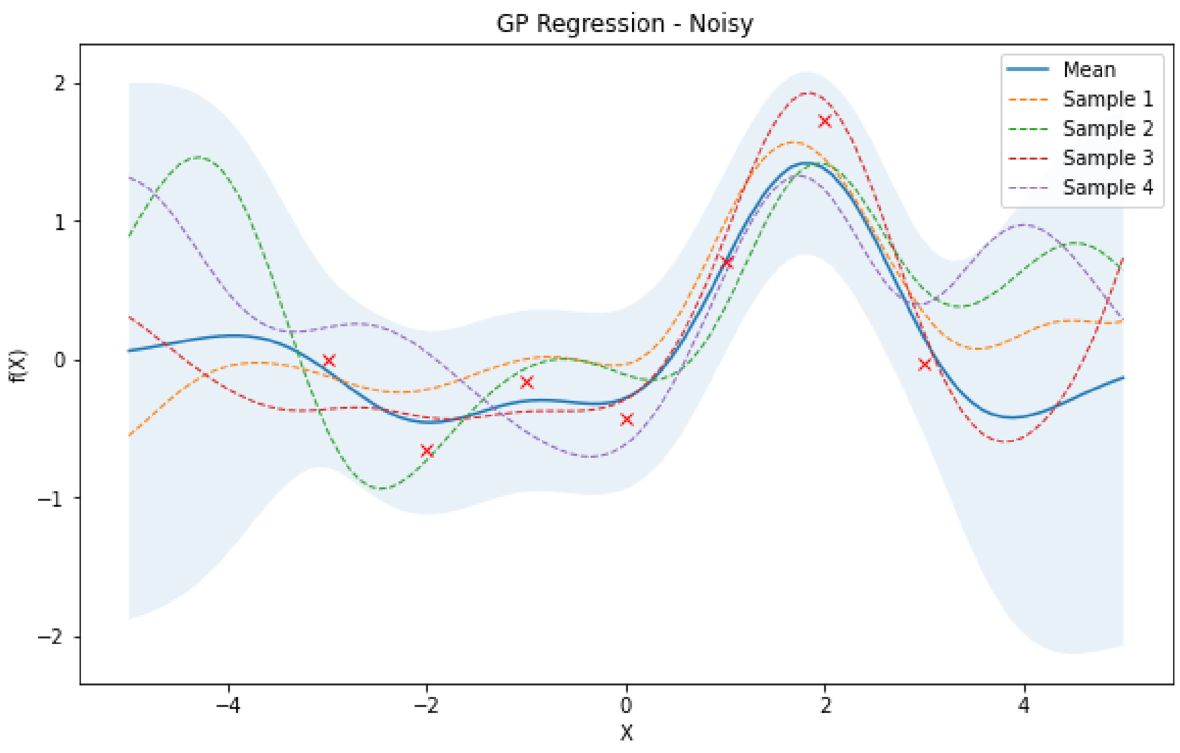
Implement the below function that computes the predictive mean and variances. Apply them to both noise-free and noisy training data `X_train` and `Y_train`. Draw four samples from the predictive posterior and plot them along with the mean, confidence interval and training data. Why don't the samples go through the training data points in noisy case?

In [7]:
```
from numpy.linalg import inv

def posterior_predictive(X_s, X_train, Y_train, l=1.0, sigma_f=1.0, sigma_y=1e-8):
    '''
    Computes the suffifient statistics of the GP posterior predictive distribution
    from m training data X_train and Y_train and n new inputs X_s.

    Args:
        X_s: New input locations (n, d).
        X_train: Training locations (m, d).
        Y_train: Training targets (m, 1).
        l: Kernel length parameter.
        sigma_f: Kernel vertical variation parameter.
        sigma_y: Noise parameter.

    Returns:
        Posterior mean vector (n, d) and covariance matrix (n, n).
    '''
    K = kernel(X_train, X_train, l, sigma_f) + sigma_y**2 * np.eye(X_train.shape[0
    K_s = kernel(X_train, X_s, l, sigma_f)
    K_ss = kernel(X_s, X_s, l, sigma_f)
    K_inv = inv(K)

    # Compute posterior mean
    mean_s = K_s.T.dot(K_inv).dot(Y_train)

    # Compute posterior covariance
    cov_s = K_ss - K_s.T.dot(K_inv).dot(K_s)

    return mean_s, cov_s
```

In [8]:
```
# Noise free training data
X_train = np.arange(-3, 4, 1).reshape(-1, 1)
Y_train = np.sin(X_train)

# Define the test data
X_test = np.linspace(-5, 5, 100).reshape(-1, 1)

# Compute the posterior mean and covariance for the noise-free case
mean_s, cov_s = posterior_predictive(X_test, X_train, Y_train, l=1.0, sigma_f=1.0,

# Sample four functions from the predictive posterior
num_samples = 4
samples = np.random.multivariate_normal(mean_s.ravel(), cov_s, num_samples)

# Plot the results
plt.figure(figsize=(10, 6))
plot_gp(mean_s, cov_s, X_test, X_train, Y_train, samples)
plt.xlabel('X')
plt.ylabel('f(X)')
plt.title('GP Regression - Noise-Free')
plt.show()
```



In [9]:
```
noise = 0.4

# Noisy training data
X_train = np.arange(-3, 4, 1).reshape(-1, 1)
Y_train = np.sin(X_train) + noise * np.random.randn(*X_train.shape)

# Define the test data
X_test = np.linspace(-5, 5, 100).reshape(-1, 1)

# Compute the posterior mean and covariance for the noisy case
mean_s, cov_s = posterior_predictive(X_test, X_train, Y_train, l=1.0, sigma_f=1.0,

# Sample four functions from the predictive posterior
num_samples = 4
samples = np.random.multivariate_normal(mean_s.ravel(), cov_s, num_samples)

# Plot the results
plt.figure(figsize=(10, 6))
plot_gp(mean_s, cov_s, X_test, X_train, Y_train, samples)
plt.xlabel('X')
plt.ylabel('f(X)')
plt.title('GP Regression - Noisy')
plt.show()
```

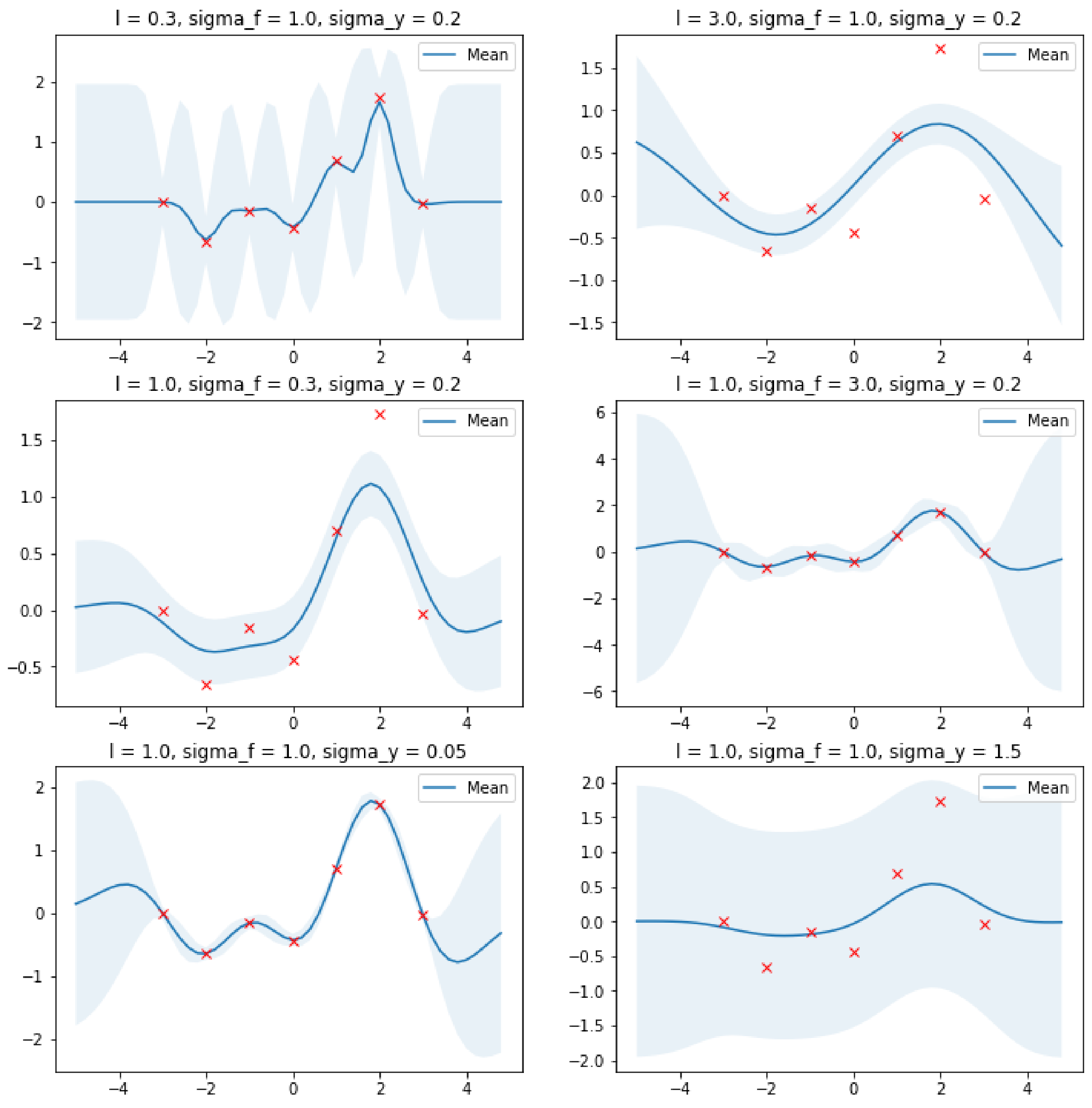


In the noisy case, the observed training data points have associated noise, which means that the target values are not exact observations of the true underlying function. The noise introduces unvertainty in the teaining data, and this uncertainty is reflected in the posterior predictive distribution.

Effect of kernel parameters and noise parameters

The following task shows the effect of kernel parameters l and σ_f as well as the noise parameter σ_y. Higher l values lead to smoother functions and therefore to coarser approximations of the training data. Lower l values make functions more wiggly with wide confidence intervals between training data points. σ_f controls the vertical variation of functions drawn from the GP. This can be seen by the wide confidence intervals outside the training data region in the right figure of the second row. σ_y represents the amount of noise in the training data. Higher σ_y values make more coarse approximations which avoids overfitting to noisy data.

In [10]:
```
import matplotlib.pyplot as plt

params = [
    (0.3, 1.0, 0.2),
    (3.0, 1.0, 0.2),
    (1.0, 0.3, 0.2),
    (1.0, 3.0, 0.2),
    (1.0, 1.0, 0.05),
    (1.0, 1.0, 1.5),
]

plt.figure(figsize=(12, 5))

for i, (l, sigma_f, sigma_y) in enumerate(params):
    mu_s, cov_s = posterior_predictive(X, X_train, Y_train, l=l,
                                       sigma_f=sigma_f,
                                       sigma_y=sigma_y)
    plt.subplot(3, 2, i + 1)
    plt.subplots_adjust(top=2)
    plt.title(f'l = {l}, sigma_f = {sigma_f}, sigma_y = {sigma_y}')
    plot_gp(mu_s, cov_s, X, X_train=X_train, Y_train=Y_train)
```



Optimal values for these parameters can be estimated by maximizing the log marginal likelihood

$$\log p(\mathbf{y}|\mathbf{X}) = \log \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{K}_y) = -\frac{1}{2}\mathbf{y}^T\mathbf{K}_y^{-1}\mathbf{y} - \frac{1}{2}\log|\,\mathbf{K}_y\,| - \frac{N}{2}\log(2\pi)$$

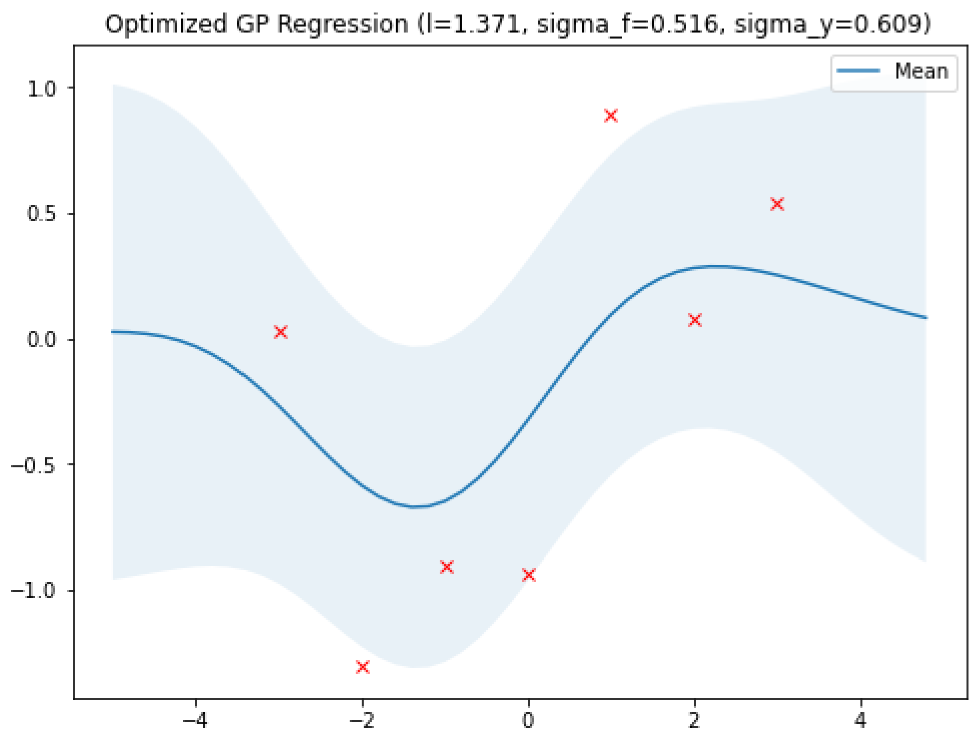
In this task, we will minimize the negative log marginal likelihood w.r.t. parameters l and σ_f, σ_y is set to the known noise level of the data. If the noise level is unknown, σ_y can be estimated as well along with the other parameters.

Exercise 3.2.4

Implement the log likehood objective below. Then, given the noisy training data above, optimize for the parameters $\theta = [l, \sigma_f, \sigma_y]$ w.r.t. the log likelihood objective. Finally, compute the predictive mean and variances with the optimized parameters and plot the result GP together with the training data.

Hint: see `scipy.optimize.minimize` documentation.

In [11]:
```
from scipy.optimize import minimize

def nll_fn(X_train, Y_train, noise):
    '''
    Returns a function that computes the negative log marginal
    likelihood for training data X_train and Y_train and given
    noise level.

    Args:
        X_train: training locations (m, d).
        Y_train: training targets (m, 1).
        noise: known noise level of Y_train.

    Returns:
        Minimization objective.
    '''
    def nll(theta):
        l, sigma_f = np.exp(theta[:2])
        sigma_y = np.exp(theta[2])
        K = kernel(X_train, X_train, l=l, sigma_f=sigma_f) + sigma_y**2 * np.eye(l
        K_inv = np.linalg.inv(K)
        ll = 0.5 * (np.log(np.linalg.det(K)) + Y_train.T @ K_inv @ Y_train + len(X
        return ll.flatten()

    return nll

# Noisy training data
X_train = np.arange(-3, 4, 1).reshape(-1, 1)
Y_train = np.sin(X_train) + noise * np.random.randn(*X_train.shape)

# Define the negative log marginal likelihood objective function
nll = nll_fn(X_train, Y_train, noise)

# Initial parameter guess
theta_initial = np.array([0, 0, np.log(noise)])

# use scipy.optimize.minimize to minimize the parameters theta
result = minimize(nll, theta_initial, bounds=((None, None), (None, None), (None, N

# Obtained optimized parameters
l_opt, sigma_f_opt, sigma_y_opt = np.exp(result.x)

# compute the prosterior predictive statistics with optimized kernel parameters and
mu_s_opt, cov_s_opt = posterior_predictive(X, X_train, Y_train, l=l_opt, sigma_f=s

# Plot GP regression result with optimized parameters
plt.figure(figsize=(8, 6))
plt.title(f"Optimized GP Regression (l={l_opt:.3f}, sigma_f={sigma_f_opt:.3f}, sig
plot_gp(mu_s_opt, cov_s_opt, X, X_train=X_train, Y_train=Y_train)
plt.show()
```


Optimized GP Regression (l=1.371, sigma_f=0.516, sigma_y=0.609)
Mean
1.0
0.5
0.0
−0.5
−1.0
−4
−2
0
2
4

Homework 3/4
LastName, FirstName: ______________________ EnrollmentID: ␣␣␣␣␣␣␣

Expectation Maximization

- https://colab.research.google.com/drive/1T0wyDvAR8XySHRh336k8QmZnxpEXWi2i?usp=sharing or
- in the attached jupyter notebook *em_image_segmentation.ipynb*

Exercise 3.3

Image Segmentation via Expectation Maximization

In this task, we apply EM algorithm to segment images. The intuition is to assign every pixels of the image to a class/cluster based on their colors (color image has 3 channels). The class/cluster can be thought as latent variable of the pixels, in which the pixels have similar colors.

Necessary imports

In [117…

```
import numpy as np
import scipy.stats
import math
import cv2
from scipy import ndimage
# Try Kmeans from diffrerent packages
from sklearn import cluster
from scipy.cluster.vq import kmeans2
import matplotlib.pyplot as plt
from scipy import ndimage
%matplotlib inline

sample_image = 'flower.jpg'
```

Image processings

In [118…

```
# input 3d array >> output 2d array
'''
  Flatten 3-dim image [w, h, c] into 2-dim array [w x h, c]

  Args:
    img_3d: Image array [w, h, c]

  Returns:
    img_2d: flatten image [w x h, c]
  '''
def flatten_img(img_3d):
    x, y, z = img_3d.shape
    img_2d = img_3d.reshape(x*y, z)
    img_2d = np.array(img_2d, dtype=float)
    return img_2d

'''
  Recover 2-dim image [w x h, c] into original 3-dim array [w, h, c]

  Args:
    img_2d: [w, h, c]
    w, h, c: width, height and channels of image to recover

  Returns:
    img_3d: recovered image [w, h, c]
  '''
```

```
def recover_img(img_2d, w, h, c):
    recover_img = img_2d.reshape(w, h, c)
    return recover_img
```

EM implementation with KMeans Clustering Initialization

Exercise 3.3.1

Implement the EM algorithm with KMeans Clustering initialization. Feel free to reuse your EM implementation from previous exercise here.

In [119…

```
# EM Algorithm
def initialize_parameters(img_2d, num_clusters):
    # Use K-Means clustering for initialization
    centers, labels = kmeans2(img_2d, num_clusters)

    # Calculate the covariance matrix for each cluster
    covariances = []
    for i in range(num_clusters):
        cluster_points = img_2d[labels == i]
        covariance = np.cov(cluster_points.T)
        covariances.append(covariance)

    # Calculate the prior probabilities (pi) for each cluster
    priors = np.ones(num_clusters) / num_clusters
    return centers, covariances, priors



def expectation(img_2d, centers, covariances, priors):
    num_pixels = img_2d.shape[0]
    num_clusters = centers.shape[0]
    responsibilities = np.zeros((num_pixels, num_clusters))

    for i in range(num_clusters):
        # Calculate the probability density function (PDF) for each pixel in each c
        pdf = scipy.stats.multivariate_normal.pdf(img_2d, mean=centers[i], cov=cova
        responsibilities[:, i] = priors[i] * pdf

    # Normalize the responsibilities
    sum_responsibilities = np.sum(responsibilities, axis=1)
    responsibilities /= sum_responsibilities[:, np.newaxis]

    return responsibilities

def maximization(img_2d, responsibilities):
    num_clusters = responsibilities.shape[1]
    num_pixels, num_channels = img_2d.shape

    centers = np.dot(responsibilities.T, img_2d) / np.sum(responsibilities, axis=0
    covariances = []
    priors = np.mean(responsibilities, axis=0)

    for i in range(num_clusters):
        # Update the covariances of each cluster
        centered_data = img_2d - centers[i]
        covariance = np.dot((responsibilities[:, i] * centered_data.T), centered_da
        covariances.append(covariance)
    return centers, covariances, priors
```

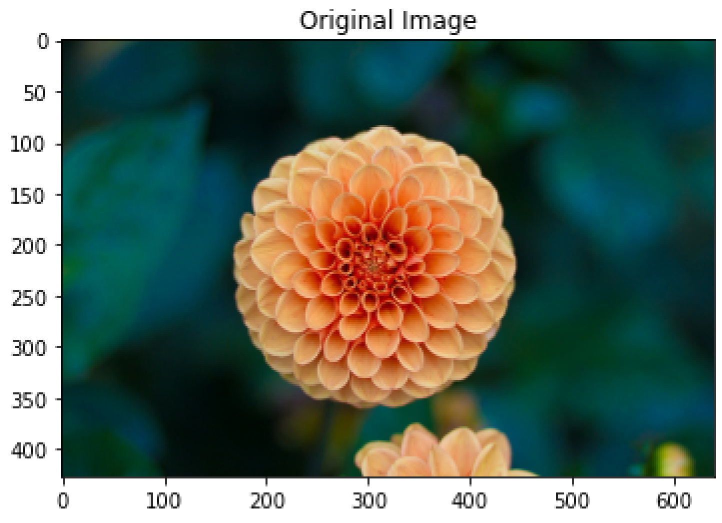
Image Segmentation with EM

We experiment the implemented EM algorithm to segment the sample flower image from `sklearn`.

In [120…

```
from sklearn.datasets import load_sample_image

orig_img = load_sample_image(sample_image)
W, H, C = orig_img.shape
print('Size of sample image: ', orig_img.shape)
plt.imshow(orig_img)
plt.title('Original Image');
```

```
Size of sample image:  (427, 640, 3)
```

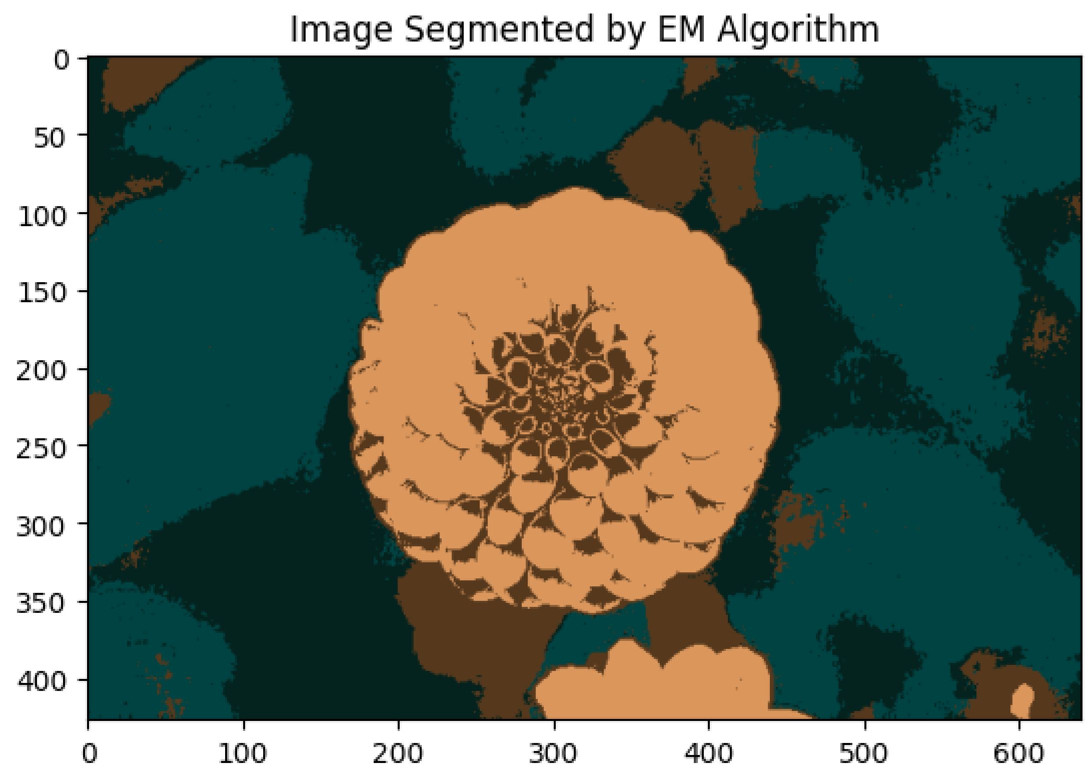


Exercise 3.3.2

Apply your EM implementation with four components (`k=4`) to segment the image and visualize the segments with the mean color of each segment.

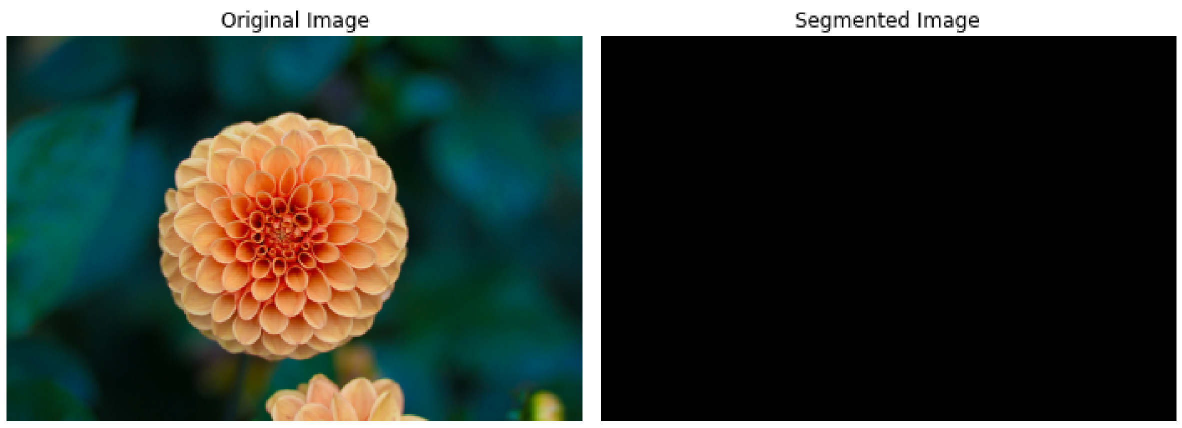
Hint: You should flatten the image first and use the argmax operation on the responsibility vector to classify the segment of the pixels.

Try to match the following segmented image as closely as possible



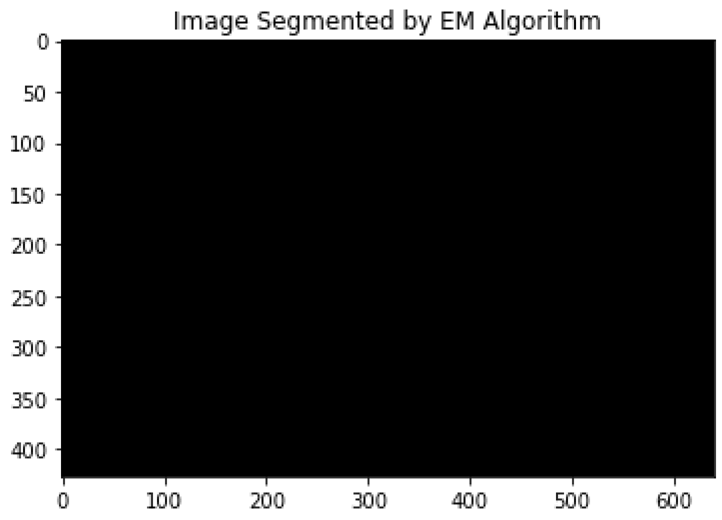
In [121…

```
# run EM on image for segmentation
def segment_image(img_2d, centers):
    labels = np.argmax(centers, axis=1)
    segmented_img = np.zeros_like(img_2d)
    for i in range(centers.shape[0]):
        indices = np.where(labels == i)
        mean_color = np.mean(img_2d[indices], axis=0)
        segmented_img[indices] = mean_color
    segmented_img = np.uint8(segmented_img.reshape((img_2d.shape[0], img_2d.shape[
    return segmented_img

# Flatten the image
img_2d = flatten_img(orig_img)

# Set the number of clusters/components
num_clusters = 4

# Initialize the parameters
centers, covariances, priors = initialize_parameters(img_2d, num_clusters)

# Run the EM algorithm
max_iterations = 100
for iteration in range(max_iterations):
    responsibilities = expectation(img_2d, centers, covariances, priors)
    centers, covariances, priors = maximization(img_2d, responsibilities)

# Segment the image
segmented_img = segment_image(img_2d, centers)

# Reshape the segmented image to its original dimensions
segmented_img = recover_img(segmented_img, orig_img.shape[0], orig_img.shape[1], o

# Display the original and segmented images
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(orig_img)
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(segmented_img)
plt.title('Segmented Image')
plt.axis('off')
plt.tight_layout()
plt.show()
```



In [122…

```
# recover image with color segments using recover_img() above.
# Note that converting array type to int might be necessary
# Reshape the segmented image to its original dimensions
em_img = recover_img(segmented_img, orig_img.shape[0], orig_img.shape[1], orig_img

# Convert the image array type to int
em_img = np.uint8(em_img)

plt.imshow(em_img)
plt.title('Image Segmented by EM Algorithm');
```



LastName, FirstName: ______________________ EnrollmentID: ⌴⌴⌴⌴⌴⌴⌴

Linear Dimensionality Reduction

- https://colab.research.google.com/drive/1ELCa-dag1SJbpHTLW2zezmLdUvzHlxe-?usp=sharing or
- in the attached jupyter notebook *pca.ipynb*

Exercise 3.4

Linear Dimensionality Reduction

In this task, we will visualize the main principle axes of the IRIS dataset. The dataset contains 150 datapoints of different irises' petal types (Setosa, Versicolour, and Virginica) which are characterized by the *sepal length, sepal width, petal length, and the petal width*. The task can also be thought of as projecting the high-dimensional dataset into lower dimensions (2D in our case) while retaining most of data information, for data exploratory purposes.

Exercise 3.4.1

Implement the Principle Component Analysis (PCA) class.

In [5]:
```python
import numpy as np

'''
  This class computes the first n eigenvectors from the dataset via fit(), and
  projects the original data to the subspace spanned by its eigenvectors via
  transform().
  '''
class PCA:
    '''
    Args:
        n_components (int): number of principle components. n_components <= d
    '''
    def __init__(self, n_components):
        self.n_components = n_components
        self.components = None  # expected size [n_components, d]
        self.mean = None  # expected size [d]

    '''
    Compute the first n_components of eigenvectors from data, and store them
    in self.components.

    Args:
        X: Array of m points (m, d).
    '''
    def fit(self, X):
        self.mean = np.mean(X, axis=0)
        X = X - self.mean
        cov = np.cov(X.T)
        eigenvalues, eigenvectors = np.linalg.eig(cov)
        indices = np.argsort(eigenvalues)[::-1]
        eigenvalues = eigenvalues[indices]
        eigenvectors = eigenvectors[:, indices]
        self.components = eigenvectors[:, :self.n_components]

    '''
    Project the data into the n_components of eigenvectors.

    Args:
        X: Array of m points (m, d).

    Returns:
        X_projected: X: Array of m points (m, n_components).
    '''
    def transform(self, X):
        X = X - self.mean
        return np.dot(X, self.components)

    def fit_transform(self, X):
        self.fit(X)
        return self.transform(X)
```

Exercise 3.4.2

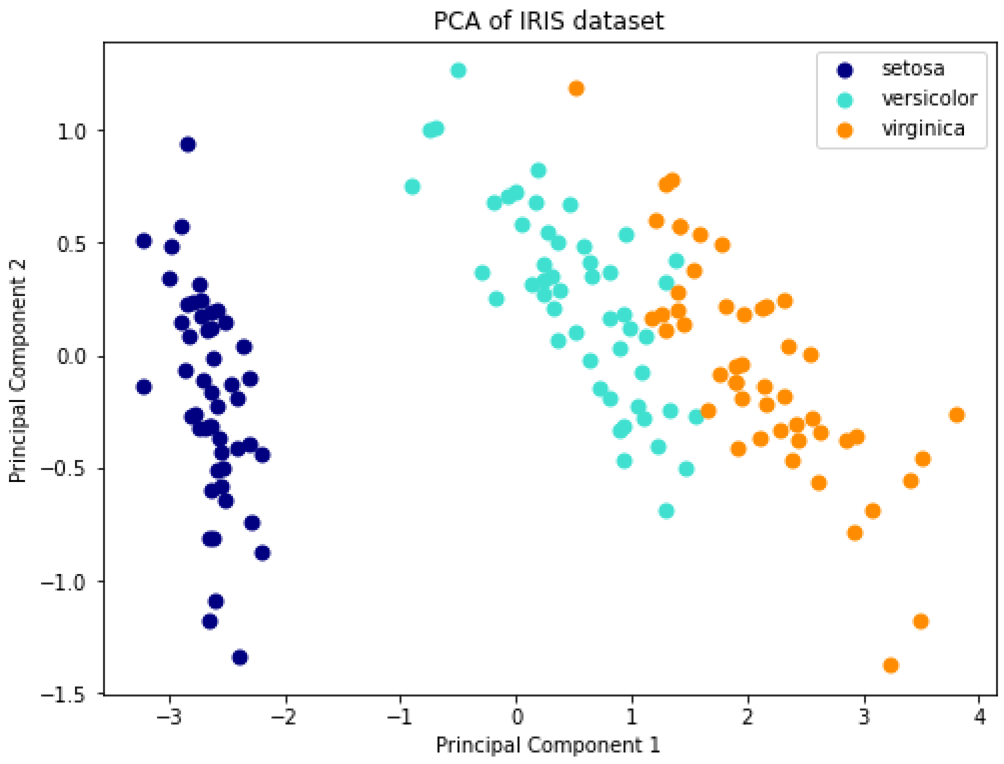
Use the PCA class to visualize the IRIS dataset in the first two principle components. The data points are also needed to be colored according to their distinct classes. Are the classes separable with linear discriminators?

Hint: Use `plt.scatter` to plot the projected data points with colors.

In [6]:

```
# Imports
import matplotlib.pyplot as plt
from sklearn import datasets
# from PCA import PCA

data = datasets.load_iris()
X = data.data
y = data.target

# Project the data onto the 2 primary principal components
pca = PCA(n_components=2)
X_projected = pca.fit_transform(X)

print("Shape of X:", X.shape)
print("Shape of transformed X:", X_projected.shape)

# visualize the projected data
colors = ['navy', 'turquoise', 'darkorange']
target_names = data.target_names

plt.figure(figsize=(8, 6))
for color, i, target_name in zip(colors, [0, 1, 2], target_names):
    plt.scatter(X_projected[y == i, 0], X_projected[y == i, 1], color=color, lw=2,

plt.title('PCA of IRIS dataset')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend(loc='best')
plt.show()
```

```
Shape of X: (150, 4)
Shape of transformed X: (150, 2)
```


PCA of IRIS dataset
setosa
versicolor
virginica
Principal Component 1
Principal Component 2
1.0
0.5
0.0
−0.5
−1.0
−1.5
−3
−2
−1
0
1
2
3
4