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

def generate_data(n, minval, maxval, variance=1., train=False, seed=0):
  """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
  """
  # 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.

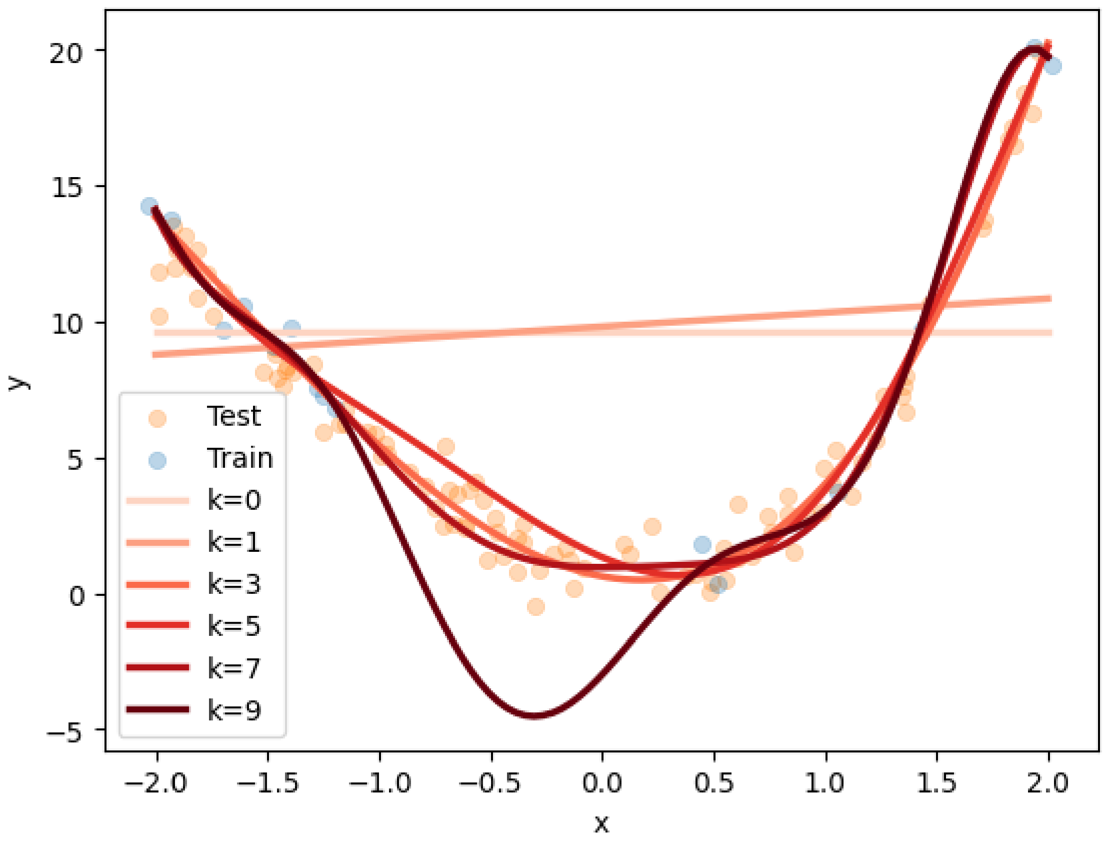
In [3]:
```
def polynomial_features(x, degree):
  """
  Calculates polynomial features function of degree n.
  The feature function includes all exponents from 0 to n.
  Args:
    x: Input of size (N, D)
    degree: Polynomial degree
  Returns:
    Polynomial features evaluated at x of dim (degree, N)
  """
  # TODO: Your code here
  N = x.shape[0]
  result = np.zeros((degree+1, N))
  temp = x.T
  for i in range(degree+1):
    result[i] = temp**i
  return result


def fit_w(x, y, lam, degree):
  """
  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
  """
  # TODO: Your code here
  feature = polynomial_features(x, degree)
  weight = np.linalg.inv((feature @ feature.T) + (lam * np.identity(degree+1)))
  weight = weight @ feature
  weight = weight @ y[:, None]
  return weight.flatten()


def predict(x, w, degree):
  """
  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
  """
  # TODO: Your code here
  # print(w.shape)
  # print(w)
  # print(w[None,:].shape)
  # print("------")
  return ((w[None, :]) @ polynomial_features(x, degree)).flatten()

def calc_mse(x, y):
  """
  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
  """
  # TODO: Your code here
  return np.mean((y-x)**2)
```

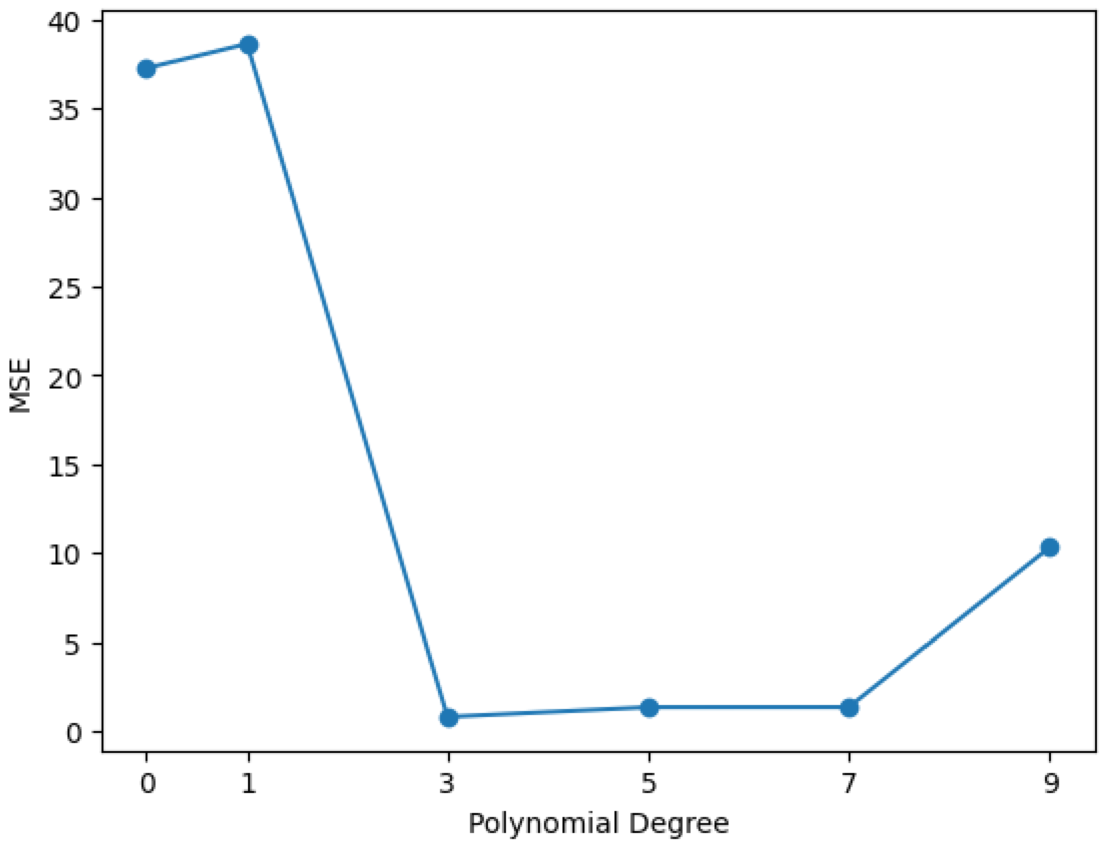
Here you can try out your code by simply running the following cell. This cell will carry out your ridge regression implementation from above for $\lambda = 0$ in which case we are provided with the linear least squares solution.

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 [41]:
```
%matplotlib inline

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

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

def plot_linear_regression(x, y, labels, eval_quantity):
  """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)
  """
  K = x.shape[0]
  colors = mpl.colormaps['Reds'].resampled(K+1)(range(1, K+1))
  fig = plt.figure()
  plt.scatter(test_data[0], test_data[1], color="tab:orange", linewidths=0.5, la
  plt.scatter(train_data[0], train_data[1], color="tab:blue", linewidths=0.5, la
  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()

def plot_mse(mse, labels):
  """Plotting functionality of the MSE for K different polynomial degrees."""
  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]
#degrees = [3,5,7]
xs, ys, mse = [], [], []
for degree in degrees:
  w = fit_w(
      train_data[0], train_data[1],
      lam=0., # Edge case resulting in linear least squares regression
      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.

`TODO: Your answer here`

In the two figures above, we see that

```
* For small K=0,1, the prediction model is too simple, therefore can
not capture the complexity, patterns and variabilities of the data
(both training and testing data). This leads to underfit phenomenon
* For too large K=9, the prediction model is more flexible and complex.
It can capture more nonlinearities in the training data. However, in
this case, our model overfits the training data too much. Therefore it
was exposed to noise and random patterns. Hence the performance on the
test data was poor (bad generalization)
* So in our concrete case, we would choose K=3,5,7 since these models
don't underfit or overfit too much. They can capture the essence of our
data and generalize well to unseen data in the future.
```

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

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

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`

3.1.3.a

$$
\begin{aligned}
&E_{D,\varepsilon}\left[\varepsilon\left(f\left(x_q\right)-\hat{f}_D\left(x_q\right)\right)\right]\\
=&E_{\varepsilon}[\varepsilon]E_D\left[f\left(x_q\right)-\hat{f}_D\left(x_q\right)\right]\\
=&0
\end{aligned}
$$

3.1.3.b

$$
\begin{aligned}
&E_D\left[\left(f\left(x_q\right)-\hat{f}_D\left(x_q\right)\right)^2\right]\\
=&E_D\left[\left(f\left(x_q\right)-\bar{\hat{f}}\left(x_q\right)+\bar{\hat{f}}\left(x_q\right)-\hat{f}_D\left(x_q\right)\right)^2\right]\\
=&E_D\left[\left(f\left(x_q\right)-\bar{\hat{f}}\left(x_q\right)\right)^2\right]+E_D\left[\left(\bar{\hat{f}}\left(x_q\right)-\hat{f}_D\left(x_q\right)\right)^2\right]+E_D\Big[2\Big(f\left(x_q\right)-\bar{\hat{f}}\left(x_q\right)\Big)\\
=&\left(f\left(x_q\right)-\bar{\hat{f}}\left(x_q\right)\right)^2+E_D\left[\left(\left(\bar{\hat{f}}\left(x_q\right)-\hat{f}_D\left(x_q\right)\right)^2\right)\right]+E_D\Big[2\Big(f\left(x_q\right)-\bar{\hat{f}}\left(x_q\right)\Big)\Big(
\end{aligned}
$$

We evaluate the third term and show why it is equal to 0

$$
\begin{aligned}
&E_D\left[2\left(f\left(x_q\right)-\bar{\hat{f}}\left(x_q\right)\right)\left(\bar{\hat{f}}\left(x_q\right)-\hat{f}_D\left(x_q\right)\right)\right]\\
=&2E_D\left[\left(f\left(x_q\right)-\bar{\hat{f}}\left(x_q\right)\right)\left(\bar{\hat{f}}\left(x_q\right)-\hat{f}_D\left(x_q\right)\right)\right]\\
=&2\left(f\left(x_q\right)-\bar{\hat{f}}\left(x_q\right)\right)E_D\left[\bar{\hat{f}}\left(x_q\right)-\hat{f}_D\left(x_q\right)\right]\\
=&2\left(f\left(x_q\right)-\hat{f}\left(x_q\right)\right)\left(E_D\left[\bar{\hat{f}}\left(x_q\right)\right]-E_D\left[\hat{f}_D\left(x_q\right)\right]\right)\\
=&0 \text{ since } E_D\left[\hat{f}_D\left(x_q\right)\right]=\bar{\hat{f}}\left(x_q\right)
\end{aligned}
$$

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

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

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

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 $\mathcal{D}_j$

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

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

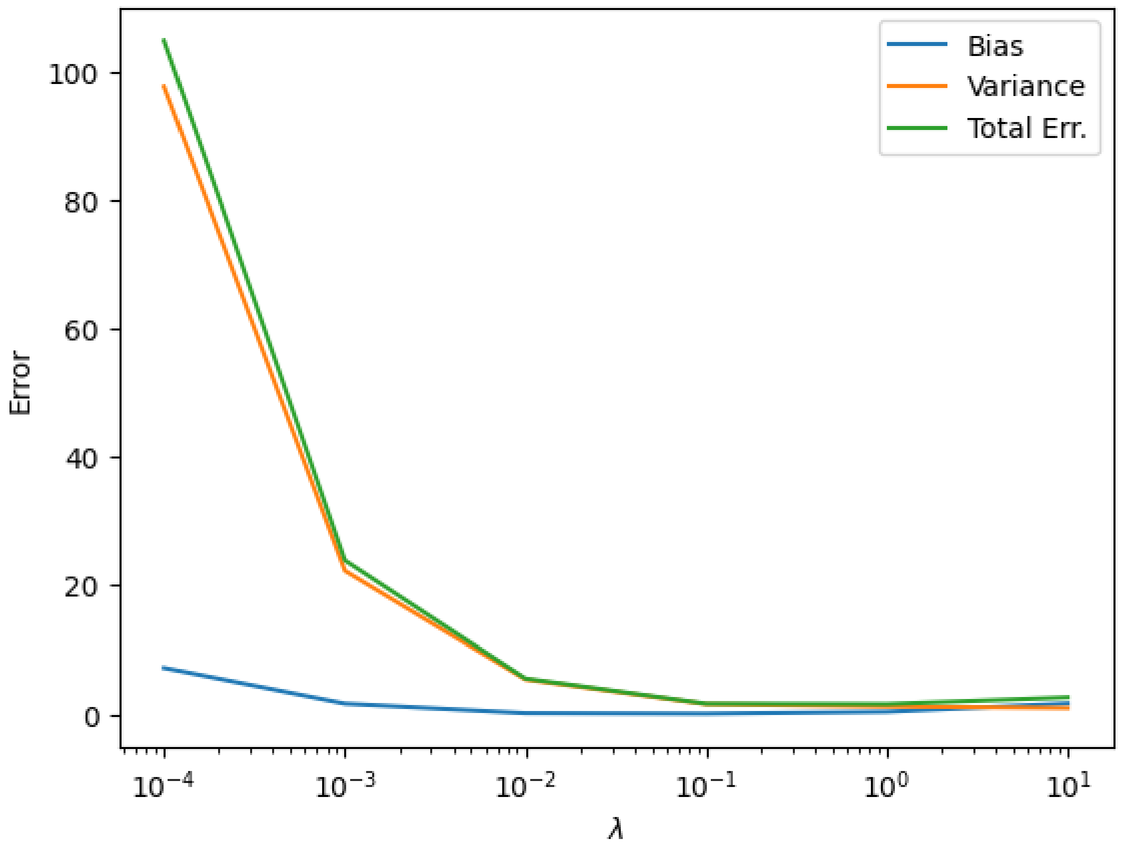
In this exercise we implement the average prediction $\bar{\hat{f}}(x_i)$, $\mathrm{Bias}\,\mathrm{bias}^2[\hat{f}_{\mathcal{D}}]$, and $\mathrm{Var}\,\mathrm{var}\left[\hat{f}_{\mathcal{D}}\right]$ as introduced above.

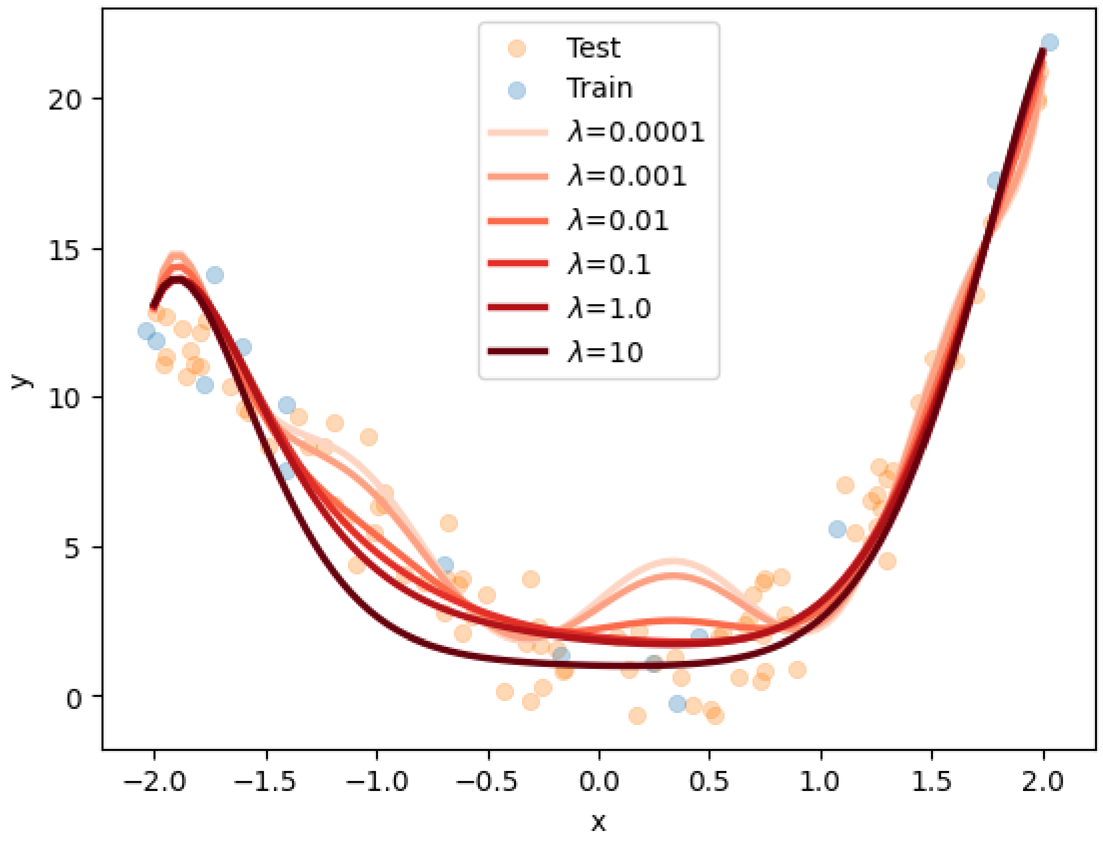
Please work through the code and fill in the the `# TODO` s.

In [49]:
```python
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 matrix (N,)
  """
  # TODO: Your code here
  poly_x = polynomial_features(x, degree)
  prediction_mat = (ws @ poly_x).T
  return np.mean(prediction_mat, axis = 1)


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
  """
  # TODO: Your code here
  avg_prediction_vector = avg_prediction(x_q, ws, degree)
  return np.mean((f(x_q)-avg_prediction_vector)**2)**(1)



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
  """
  # TODO: Your code here
  poly_x = polynomial_features(x_q, degree)
  predictions_M = ws @ poly_x
  return np.mean(np.var(predictions_M, axis = 0))
```

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 [52]:
```
%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)
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?

`TODO: Your answer here` The bias and variance behaves as expected because

- For small lambda, the model is not constrained by regularization,which leads to the fact that the attributes are not much penalized. Larger attributes cause the variance of our model to increase. Small lambda also leads to smaller bias, since the model is allowed to fit data more closely
- For larger lambda, the model is more constrained, so the inverse effect to the effect of small lambda happens
- Overall, we would choose lambda to be around 1.0, since this is where the graph of bias and variance crosses, indicating an optimal model error-wise.

Gradient Descent

In the lecture we have seen that the closed form solution of linear regression requires us to take the inverse $(\mathbf{\Phi}^T\mathbf{\Phi})^{-1}$. 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.

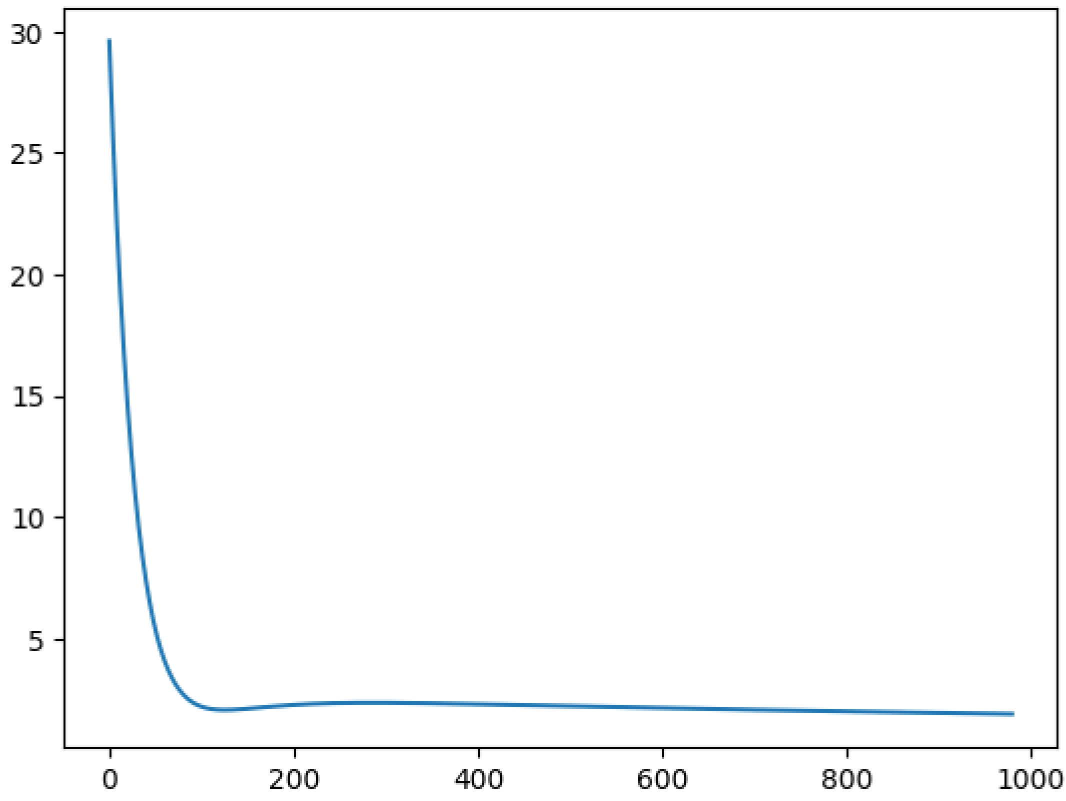
We expect you to provide a plot of the learning curve, i.e., a plot of the MSE on the test data against the iterations. You can evaluate your model after $n_{\text{eval}} = 20$ gradient updates. We further would like to see the model prediction after $n = 0, 10, 100, 1000$ gradient updates/iterations.

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

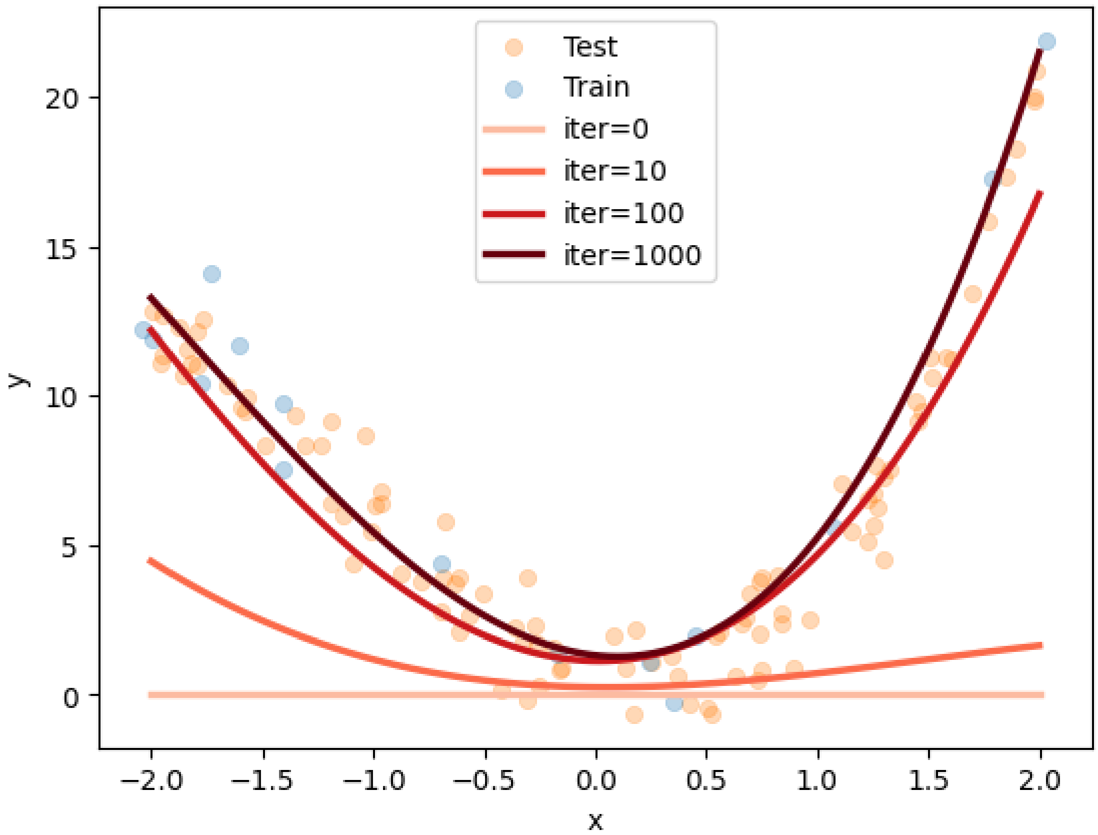
In [51]:
```python
%matplotlib inline

# 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)
```

```python
# TODO: Your code here
#Setting of our solution
X_train, y_train = train_data
X_test, y_test = test_data
iteration_10 = [0,10,100,1000]
eval_val = np.zeros(980)
w_val = []

def gradient_descent(X, y, learning_rate=0.0001, iters=1000):
  #Initialize paramter
  w = np.zeros(degree+1)

  #for-loop to perform gradient_descent optimization
  for i in range(iters):
    if(i in iteration_10):
      w_val.append(w)
    pred_mat = predict(X, w, degree)
    gradient = gradient_calculator(X, y, pred_mat)
    w = w - gradient*learning_rate
    if(i>=20):
      eval_val[i-20]=calc_mse(predict(X_test, w, degree),y_test)
  plt.plot(eval_val)
  w_val.append(w)
  return w

def gradient_calculator(X, y, prediction_mat):
  return 2*(polynomial_features(X, degree)@(prediction_mat-y))

def w_update(w, gradient, learning_rate=0.0001):
  w = w - gradient*learning_rate

X, y = train_data
# print(X.shape)
# print(polynomial_features(X, 3).shape)
# print(predict(X_train, np.zeros(4), 3).shape)
param=gradient_descent(X_train, y_train)

fig = plt.figure()
plt.scatter(X_test, y_test, color="tab:orange", linewidths=0.5, label="Test", al
plt.scatter(X_train, y_train, color="tab:blue", linewidths=0.5, label="Train", a
x_ax, y_ax=[],[]
for w in w_val:
  x_ax.append(np.linspace(minval, maxval, 100))
  y_ax.append(predict(x_ax[-1], w, degree))
x_ax=np.stack(x_ax)
y_ax=np.stack(y_ax)
colors = mpl.colormaps['Reds'].resampled(4+1)(range(1, 4+1))
for i in range(4):
    plt.plot(x_ax[i], y_ax[i], label=f"iter={iteration_10[i]}", color=colors[i],
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
```

```
Out[51]: <matplotlib.legend.Legend at 0x7f6d701d9780>
```



30
25
20
15
10
5
0
200
400
600
800
1000



Test
Train
iter=0
iter=10
iter=100
iter=1000
y
x
20
15
10
5
0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

```
In [ ]:
```