Bias-Variance Tradeoff

CSI 4106 - Fall 2025

Marcel Turcotte
Version: Oct 10, 2025 12:42

Preamble

Summary

In this lecture, we explore how **model complexity** influences **bias**, **variance**, and **generalization** by examining **underfitting** and **overfitting** through **learning curves** across various models, including linear, polynomial, tree-based, KNN, and deep networks.

Learning Outcomes

- **Grasp** how model complexity affects bias, variance, and generalization.
- **Analyze** learning curves to diagnose underfitting and overfitting.

Model Complexity

Rationale

Optimizing **model performance** critically depends on the careful selection and tuning of **hyperparameters**.

These hyperparameters play a pivotal role in regulating the **complexity** of machine learning models.

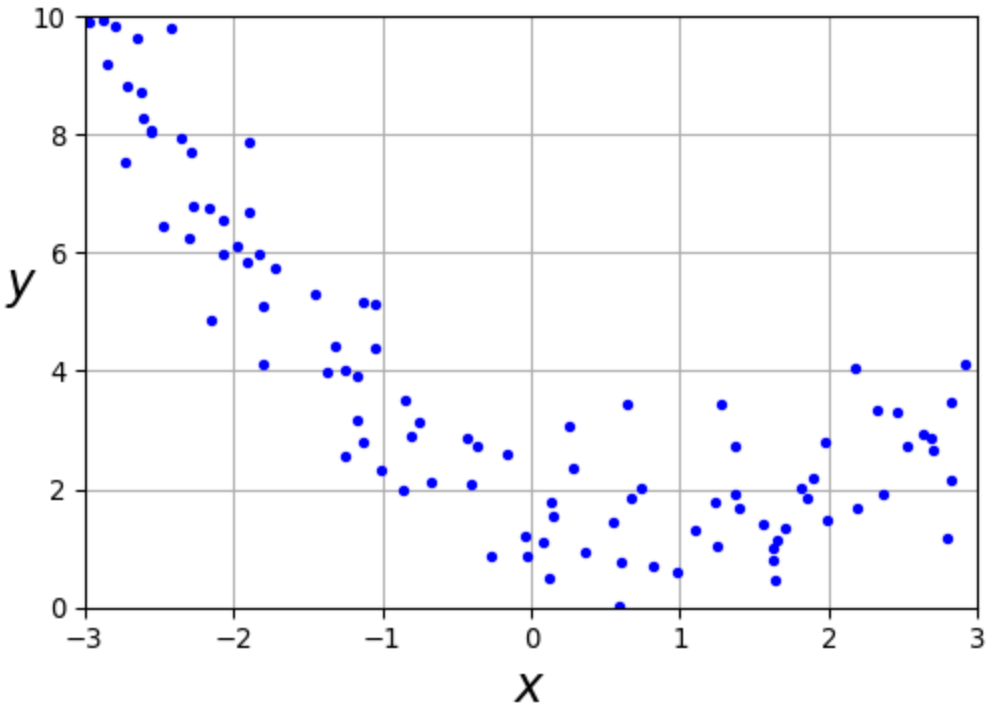
Definition

Model complexity in refers to the capacity of a model to capture intricate patterns in the data.

It is determined by the **number of parameters** or the **structure** of the model.

Exploration

In [2]:
```
import numpy as np

np.random.seed(42)

X = 6 * np.random.rand(100, 1) - 3
y = 0.5 * X ** 2 - X + 2 + np.random.randn(100, 1)

import matplotlib as mpl
import matplotlib.pyplot as plt

plt.figure(figsize=(6,4))

plt.plot(X, y, "b.")
plt.xlabel("$x$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, 0, 10])
plt.grid(True)
plt.show()
```



Attribution: Géron (2022), Chapter 4

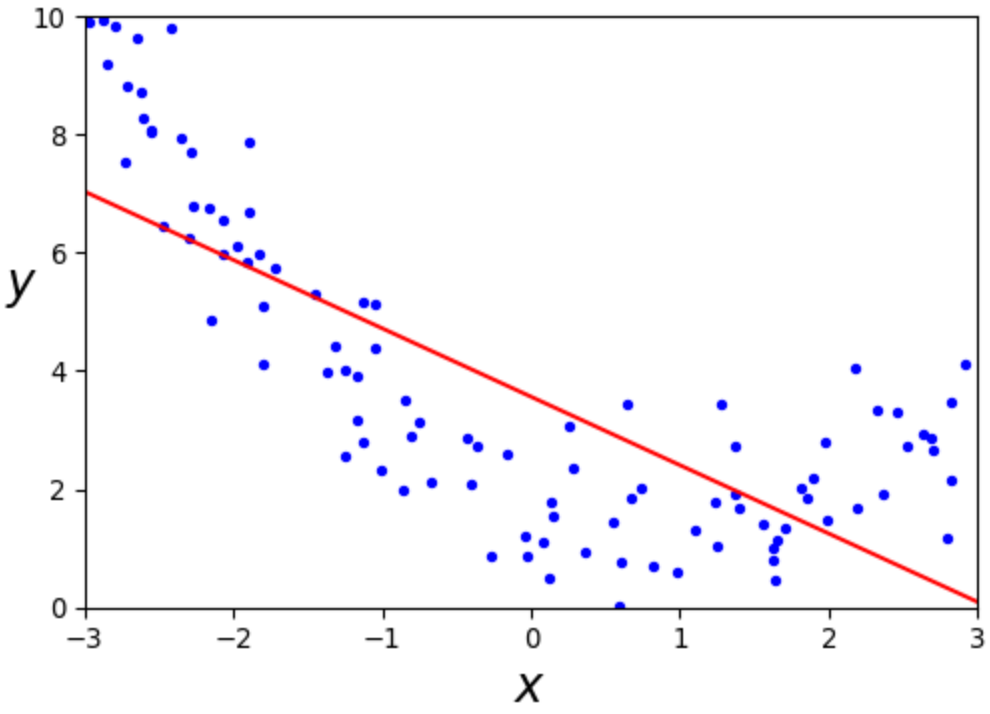
In machine learning experiments, specifying the seed of the random number generator is crucial for ensuring reproducibility. By setting a fixed seed, programmers can guarantee that the same sequence of random numbers will be generated each time the experiment is run. This consistency is vital for several reasons:

1. **Reproducibility**: It allows other programmers to replicate the experiment with the exact same conditions, facilitating verification and validation of results.

2. **Comparative Analysis**: It enables consistent comparison between different models or algorithms under the same initial conditions, ensuring that observed differences are due to the models themselves rather than variations in the random initialization.
3. **Debugging**: It aids in debugging by providing a stable environment where issues can be consistently reproduced and investigated.

Linear Regression

In [3]:
```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(X, y)

X_new = np.array([[-3], [3]])
y_pred = lin_reg.predict(X_new)

plt.figure(figsize=(6,4))

plt.plot(X, y, "b.")
plt.plot(X_new, y_pred, "r-")
plt.xlabel("$x$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, 0, 10])
plt.show()
```



A **linear model** inadequately represents this dataset.

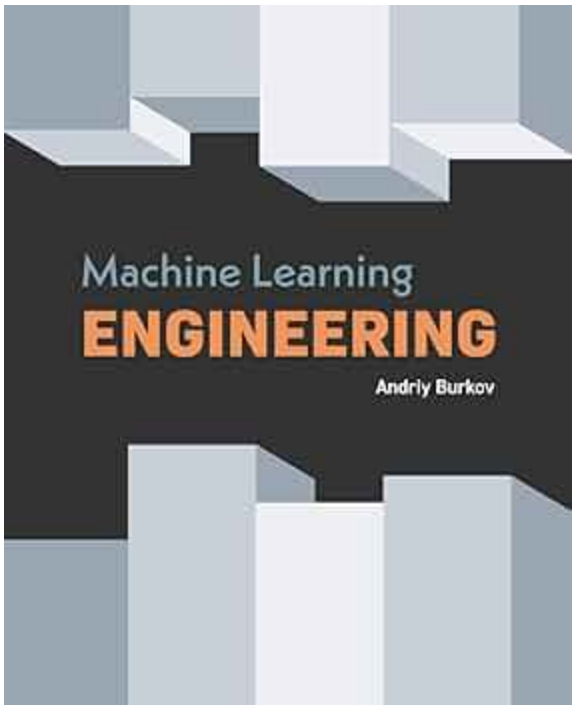
Adjusting the slope invariably leads to a scenario where residuals (errors) are minimized for certain data points while remaining significant for others.

Definition

Feature engineering is the process of **creating**, **transforming**, and **selecting** variables (attributes) from raw data to improve the performance of machine learning models.

Here, our focus is on **creating** new attributes from raw data.

Machine Learning Engineering



- **Machine Learning Engineering** by Andriy Burkov (A. Burkov 2020).
- Covers data collection, storage, preprocessing, feature engineering, model testing and debugging, deployment, retirement, and maintenance.
- From the author of **The Hundred-Page Machine Learning Book** (Andriy Burkov 2019).
- Available under a "read first, buy later" model.

`PolynomialFeatures`

In [4]:
```
from sklearn.preprocessing import PolynomialFeatures

poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
```

...

In [5]:
```
X[0]
```

```
array([-0.75275929])
```

...

In [6]:
```
X_poly[0]
```

```
array([-0.75275929,  0.56664654])
```

...

> ** sklearn.preprocessing.PolynomialFeatures**
>
> Generate a new feature matrix consisting of all polynomial combinations of the features with degree less than or equal to the specified degree. For example, if an input sample is two dimensional and of the form $[a, b]$, the degree-2 polynomial features are $[1, a, b, a^2, ab, b^2]$.

`PolynomialFeatures`

Given two features a and b, `PolynomialFeatures` with `degree=3` would add a^2, a^3, b^2, b^3, as well as, ab, a^2b, ab^2!

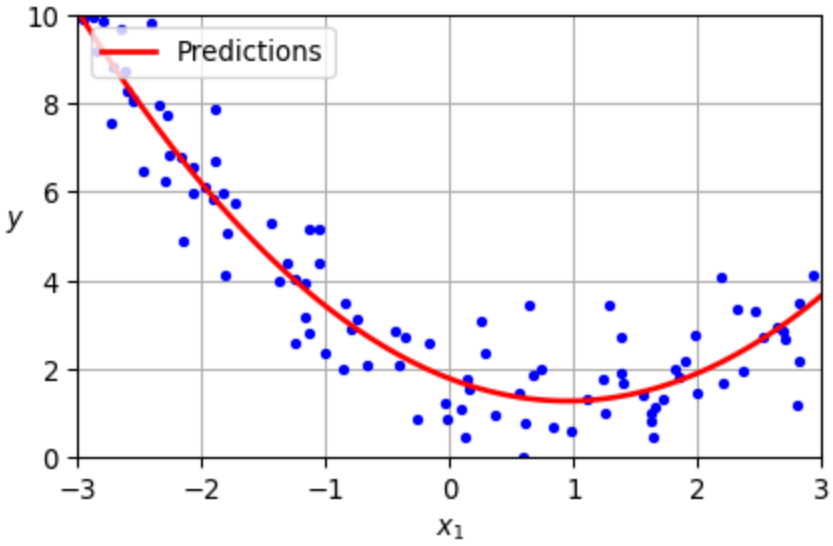
...

> **Warning**
>
> `PolynomialFeatures(degree=d)` adds $\frac{(D+d)!}{d!D!}$ features, where D is the original number of features.

Additionally, you have the option to engineer new features of your own.

Polynomial Regression

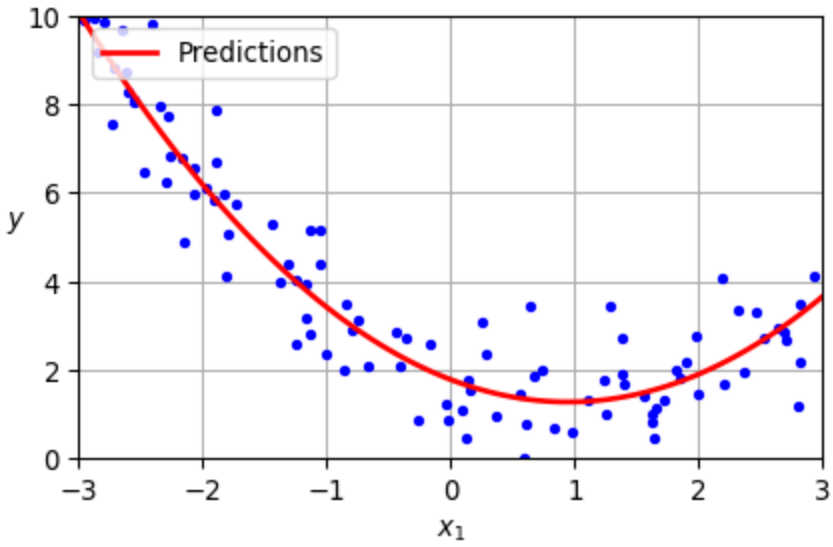
In [7]:
```
lin_reg = LinearRegression()
lin_reg = lin_reg.fit(X_poly, y)

X_new = np.linspace(-3, 3, 100).reshape(100, 1)
X_new_poly = poly_features.transform(X_new)
y_new = lin_reg.predict(X_new_poly)

plt.figure(figsize=(5, 3))
plt.plot(X, y, "b.")
plt.plot(X_new, y_new, "r-", linewidth=2, label="Predictions")
plt.xlabel("$x_1$")
plt.ylabel("$y$", rotation=0)
plt.legend(loc="upper left")
plt.axis([-3, 3, 0, 10])
plt.grid()
plt.show()
```



`LinearRegression` on `PolynomialFeatures`

In [8]:
```
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
```

Polynomial Regression



The data was generated according to the following equation, with the inclusion of Gaussian noise.

$$ y = 0.5 x^2 - 1.0 x + 2.0 $$

Presented below is the learned model.

$$ \hat{y} = 0.56 x^2 + (-1.06) x + 1.78 $$

In [10]:
```
lin_reg.coef_, lin_reg.intercept_
```

```
(array([[-1.06633107,  0.56456263]]), array([1.78134581]))
```

1. **Linearity is about the coefficients, not the raw inputs**

- Logistic regression is linear in its **parameters** (weights).
- The decision boundary comes from:

$$ \hat{y_i} = \sigma(\theta_0 + \theta_1 z_i^{(1)} + \theta_2 z_i^{(2)} + \dots + \theta_{D'} z_i^{(D')}) $$

where each $z_i^{(j)}$ is a feature.

- If you define $z_i^{(j)}$ to be a polynomial function of the original variables (e.g. $z_i^{(3)} = (x_i^{(1)})^2$), the model is still *linear in the $z_i^{(j)}$*.

2. **PolynomialFeatures only changes the feature space**

- `PolynomialFeatures` maps input $(x_i^{(1)}, x_i^{(2)})$ to a new vector: $(x_i^{(1)}, x_i^{(2)}, (x_i^{(1)})^2, x_i^{(1)}x_i^{(2)}, (x_i^{(2)})^2, \ldots)$.
- Logistic regression then finds a linear decision boundary in this **expanded feature space**.
- When you map it back to the original $(x_i^{(1)}, x_i^{(2)})$ space, that "linear" decision surface can appear curved or wiggly, but mathematically it's still a hyperplane in the transformed space.

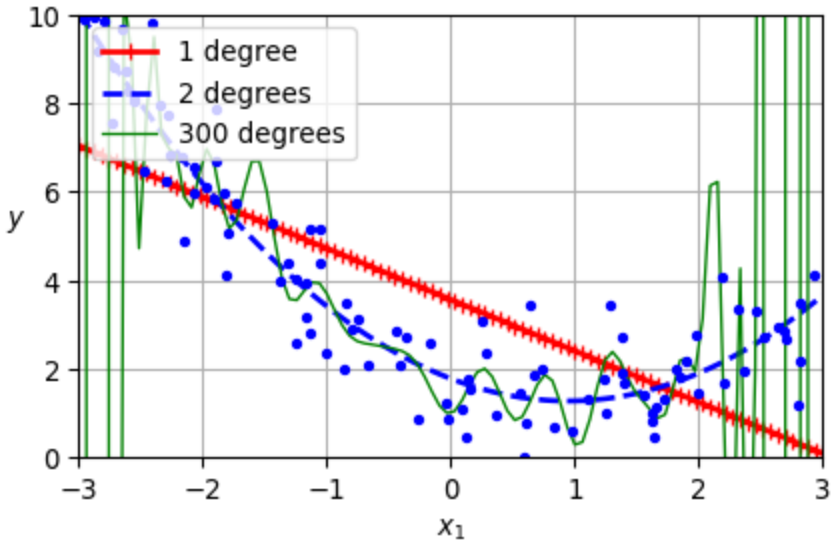
Key message:

"Logistic regression always finds a linear boundary in the features you give it. If you feed it polynomials, the linear boundary in that space looks nonlinear in the original space."

Model Complexity

```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

plt.figure(figsize=(5, 3))

for style, width, degree in (("r-+", 2, 1), ("b--", 2, 2), ("g-",
1, 300)):
    polybig_features = PolynomialFeatures(degree=degree,
include_bias=False)
    std_scaler = StandardScaler()
    lin_reg = LinearRegression()
    polynomial_regression = make_pipeline(polybig_features,
std_scaler, lin_reg)
    polynomial_regression.fit(X, y)
```

```
    y_newbig = polynomial_regression.predict(X_new)
    label = f"{degree} degree{'s' if degree > 1 else ''}"
    plt.plot(X_new, y_newbig, style, label=label, linewidth=width)

plt.plot(X, y, "b.", linewidth=3)
plt.legend(loc="upper left")
plt.xlabel("$x_1$")
plt.ylabel("$y$", rotation=0)
plt.axis([-3, 3, 0, 10])
plt.grid()
plt.show()
```



A **low loss value** on the **training set** does not necessarily indicate a “better” model.

Attribution: 04_training_linear_models.ipynb

In this example, the linear regression model has high mean squared error (loss) on the training set (red line). This suggests that the model makes numerous errors even on the training data.

Conversely, the polynomial model with degree=300 exhibits a low mean squared error (loss) on the training set (green line), implying that it makes few errors on the training data.

However, the degree=300 polynomial model is likely to perform poorly on future predictions. The green curve extends beyond the boundaries of the image on the y-axis. For instance, for input values in the range of 2 to 3, the model predicts values exceeding 10 (as well as negative values), whereas the expected values should lie within the range of 2 to 4.

This illustrative example may seem simplistic since the data is generated from a quadratic equation and involves only a single attribute, making visualization straightforward. However, it serves to highlight a key point relevant to more complex

models, such as deep neural networks. As the number of parameters increases, the model's capacity to fit the training data also increases, which can lead to overfitting if not properly managed.

Under- and Over- Fitting

- **Underfitting:**
  - Your model is **too simple** (here, linear).
  - Uninformative features.
  - **Poor** performance on both **training** and **test** data.
- **Overfitting:**
  - Your model is **too complex** (tall decision tree, deep and wide neural networks, etc.).
  - **Too many features** given the number of examples available.
  - **Excellent** performance on the **training set**, but **poor** performance on the **test set**.

A model is said to **underfit** when it fails to capture the underlying structure of the data, it performs poorly both on the training set and on unseen test data. Here are the main ways this can happen:

Model capacity too low

- **Too simple hypothesis class**: e.g., fitting a straight line (linear regression) to data with a quadratic pattern.
- **Shallow architecture**: using a single-layer perceptron for a highly non-linear classification task.
- **Too few features**: not providing the model with enough descriptive variables to capture the signal.

Too much regularization

- **Excessive L2 (ridge) or L1 (lasso) penalty** shrinks coefficients nearly to zero.
- **Dropout too high** in neural networks.
- **Early stopping too soon** before the model learns the patterns.

Optimization issues

- **Learning rate too small:** model barely learns, stuck with high loss.
- **Learning rate too large:** optimizer oscillates, never converges properly.
- **Too few training epochs/iterations:** model doesn't get enough updates to fit the data.

- **Poor initialization** (especially in non-convex problems like deep nets) can hinder convergence.

Data-related causes

- **Label noise dominates signal**: if the dataset is very noisy, even a good model looks like it underfits.
- **Insufficient data volume**: the model cannot generalize well because the training set doesn't capture the true structure.
- **Feature scaling issues**: e.g., logistic regression or neural nets trained on raw, unscaled features → poor convergence and fit.
- **Wrong feature engineering**: missing interaction terms, polynomial terms, embeddings, etc.

Algorithmic mismatches

- **Wrong loss function**: e.g., using MSE for highly imbalanced classification instead of log-loss.
- **Incompatible model choice**: e.g., k-NN with `k` set too large, predicts majority class almost always.
- **Too strong bias assumption**: e.g., Naive Bayes assuming feature independence when it is strongly violated.

Summary:

Underfitting happens whenever the model is **too constrained** (low capacity, too much regularization, too little training) or when the **training process fails** (optimization/data problems). You usually spot it when **training loss remains high** even after long training.

A model **overfits** when it learns not only the underlying structure of the training data but also the **noise and idiosyncrasies**, leading to poor generalization on unseen data. Here are the main ways this can happen:

Model capacity too high

- Model is too flexible (e.g., very deep neural net on a tiny dataset).
- Too many features relative to number of samples → high variance.
- Adding high-degree polynomial features without sufficient data.

Too little regularization

- Weak or no L1/L2 penalty allows weights to grow large.
- Dropout rate too low or disabled.

Training too long

- Model continues to minimize training loss far beyond the point where validation loss bottoms out.
- No early stopping → memorization of noise/outliers.

Data issues

- Training set is too small, model memorizes instead of generalizing.
- High label noise, model fits the errors.
- Leakage of test information into training (e.g., improper preprocessing, scaling, or feature construction).

Evaluation mismatch

- Hyperparameters tuned directly on the test set, model adapts to that test set rather than the underlying distribution.
- Cross-validation folds not stratified, misleading generalization signals.

Algorithmic choices

- k-NN with `k=1` (memorizes training points).
- Decision tree with no max depth or pruning.
- Neural network with too many parameters relative to dataset size.

Summary:

Overfitting happens when the model has **too much freedom**, **too little constraint**, or **trains too long** on a limited/noisy dataset. You recognize it when **training loss is very low but validation/test loss is much higher**.

Learning Curves

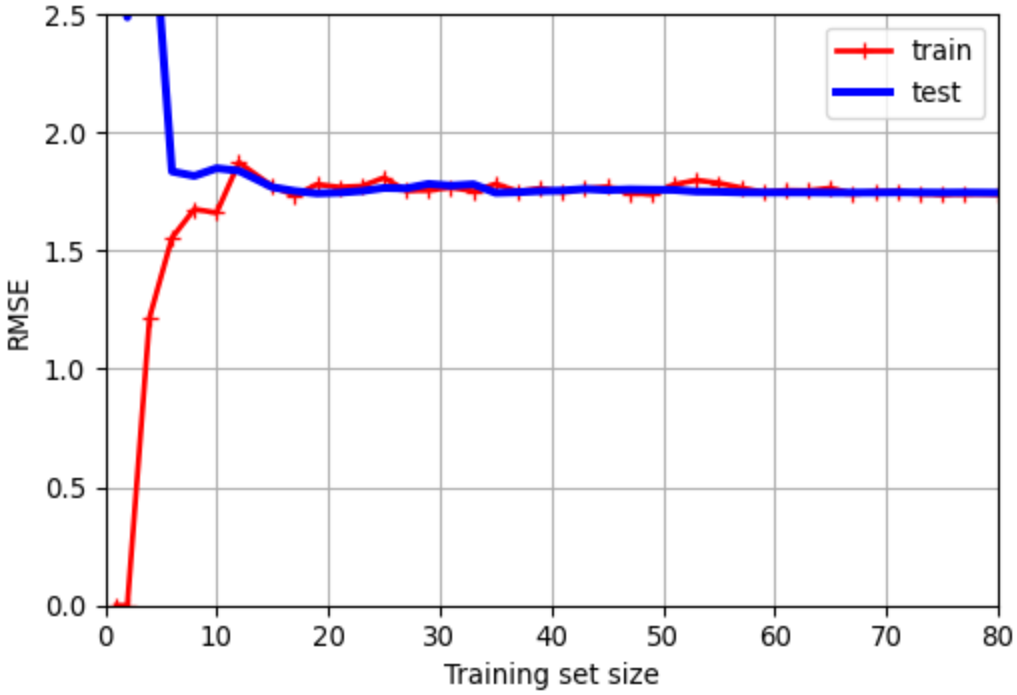
- One way to **assess our models** is to visualize the **learning curves**:
  - A **learning curve** shows the performance of our model, here using RMSE, on both the **training set** and the **test set**.
  - Multiple **measurements** are obtained by repeatedly training the model on **larger** and **larger subsets** of the data.

See: sklearn.model_selection.learning_curve.

Learning Curve – Underfitting

```
from sklearn.model_selection import learning_curve

train_sizes, train_scores, valid_scores = learning_curve(
    LinearRegression(), X, y, train_sizes=np.linspace(0.01, 1.0,
```

```
40), cv=5,
    scoring="neg_root_mean_squared_error")

train_errors = -train_scores.mean(axis=1)
valid_errors = -valid_scores.mean(axis=1)

plt.figure(figsize=(6, 4))  # extra code – not needed, just
formatting
plt.plot(train_sizes, train_errors, "r-+", linewidth=2,
label="train")
plt.plot(train_sizes, valid_errors, "b-", linewidth=3,
label="test")

# extra code – beautifies
plt.xlabel("Training set size")
plt.ylabel("RMSE")
plt.grid()
plt.legend(loc="upper right")
plt.axis([0, 80, 0, 2.5])
plt.show()
```



- Polynomial with **degree=1**.
- **Poor** performance on both **training** and **test** data.

Source code: 04_training_linear_models.ipynb.

This graph illustrates the learning curve for a linear regression model applied to data generated from a quadratic equation, which serves as our ongoing example.

The horizontal axis represents the size of the training set. Initially, the linear regression model is trained on a very small dataset, consisting of just one or a few examples, and the Root Mean Square Error (RMSE) is plotted for both the training and test sets. The size of the training set is then incrementally increased, a new model is trained, and the performance is recorded. This procedure continues until the entire dataset is utilized.

Key observations from the graph include:

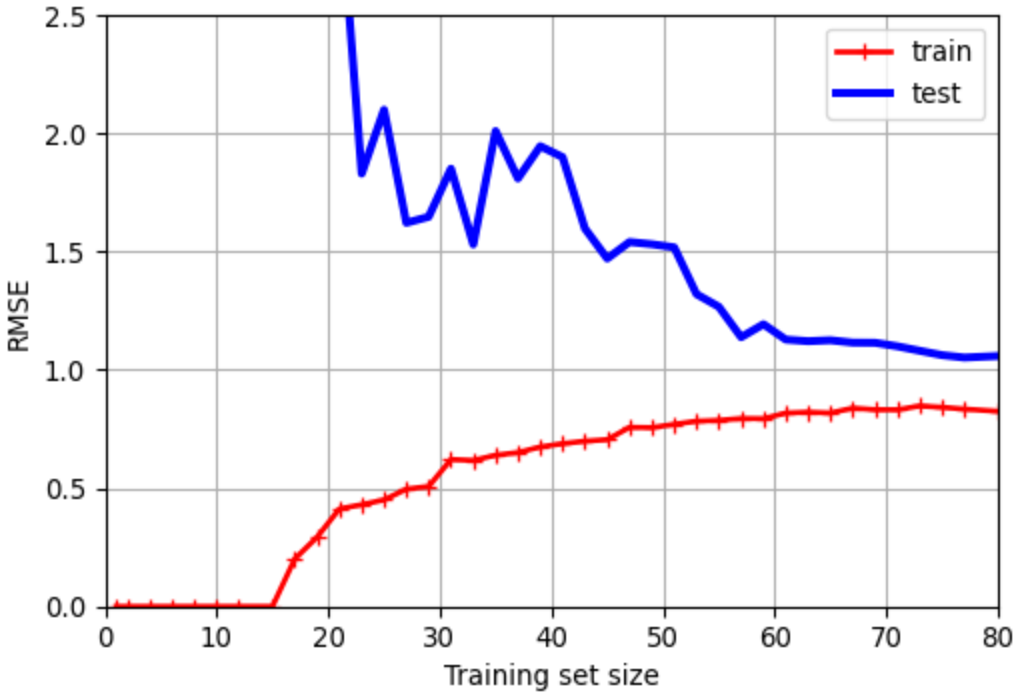
- With only one or two examples, the model perfectly fits the training set, resulting in low RMSE for the training data.
- As the size of the training set increases, the model struggles to fit the training data due to the quadratic nature of the data generation process. Consequently, the RMSE for the training set rises and stabilizes at a higher level.
- For small training sets, the model performs poorly on the test set due to inadequate generalization, resulting in high RMSE.
- As the training set size grows, the test set performance improves, indicated by decreasing RMSE, until it reaches a point where further increases in training set size do not yield significant improvements.

These learning curves are indicative of a model that is underfitting. Both the training and test set RMSE curves plateau at relatively high values and remain close to each other, as noted by Géron (2022).

Learning Curve – Overfitting

```
from sklearn.pipeline import make_pipeline

polynomial_regression = make_pipeline(
    PolynomialFeatures(degree=14, include_bias=False),
    LinearRegression())

train_sizes, train_scores, valid_scores = learning_curve(
    polynomial_regression, X, y, train_sizes=np.linspace(0.01, 1.0,
40), cv=5,
    scoring="neg_root_mean_squared_error")
# extra code – generates and saves Figure 4–16

train_errors = -train_scores.mean(axis=1)
valid_errors = -valid_scores.mean(axis=1)

plt.figure(figsize=(6, 4))
plt.plot(train_sizes, train_errors, "r-+", linewidth=2,
label="train")
plt.plot(train_sizes, valid_errors, "b-", linewidth=3,
label="test")
```

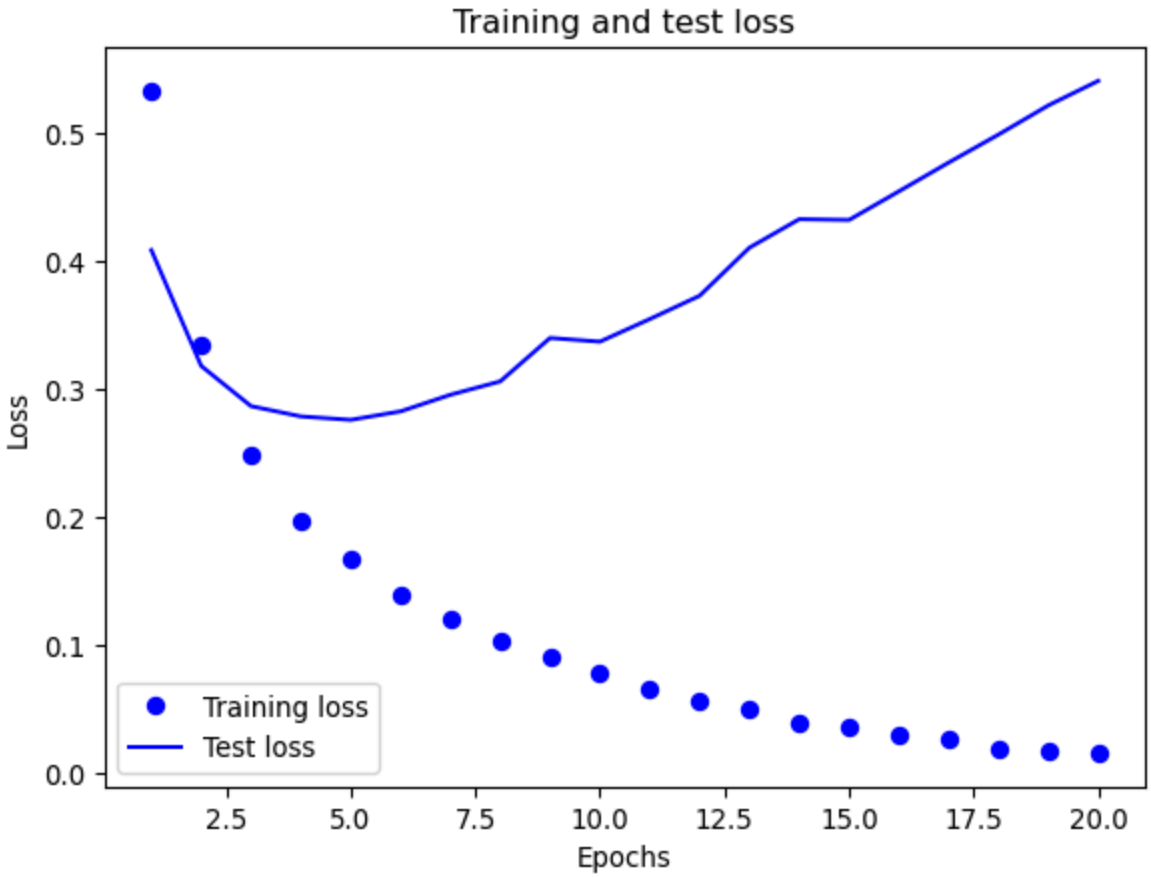
```
plt.legend(loc="upper right")
plt.xlabel("Training set size")
plt.ylabel("RMSE")
plt.grid()
plt.axis([0, 80, 0, 2.5])
plt.show()
```



- Polynomial with **degree=14**.
- **Excellent** performance on the **training set**, but **poor** performance on the **test set**.
- For a training set of up to 14 data points, the polynomial fits the training data perfectly, resulting in an RMSE of zero.
- The error on the **training data** in this instance is significantly lower.
- A notable gap between the two curves indicates that the model performs substantially better on the training data compared to the test data.

Overfitting - Deep Net - Loss

In [18]:



Example from Chollet (2017) Chapter 3 (chapter04_getting-started-with-neural-networks.ipynb, ipynb from 2021 edition).

Neural networks will be covered in detail later in our course. The graph presented here illustrates the variation in the loss function as a deep learning model undergoes training.

This example utilizes the IMDB movie review sentiment classification dataset available in Keras. The dataset comprises 25,000 movie reviews from IMDB, each labeled with a sentiment (positive or negative).

The model consists of three dense layers with sizes 16, 16, and 1, respectively. It includes a total of 160,305 trainable parameters.

The network is trained using mini-batch stochastic gradient descent with a batch size of 512. The horizontal axis represents the number of epochs, where each epoch indicates that the model has seen the entire training set once. During each epoch, the stochastic gradient descent algorithm updates the model parameters iteratively using mini-batches of 512 examples.

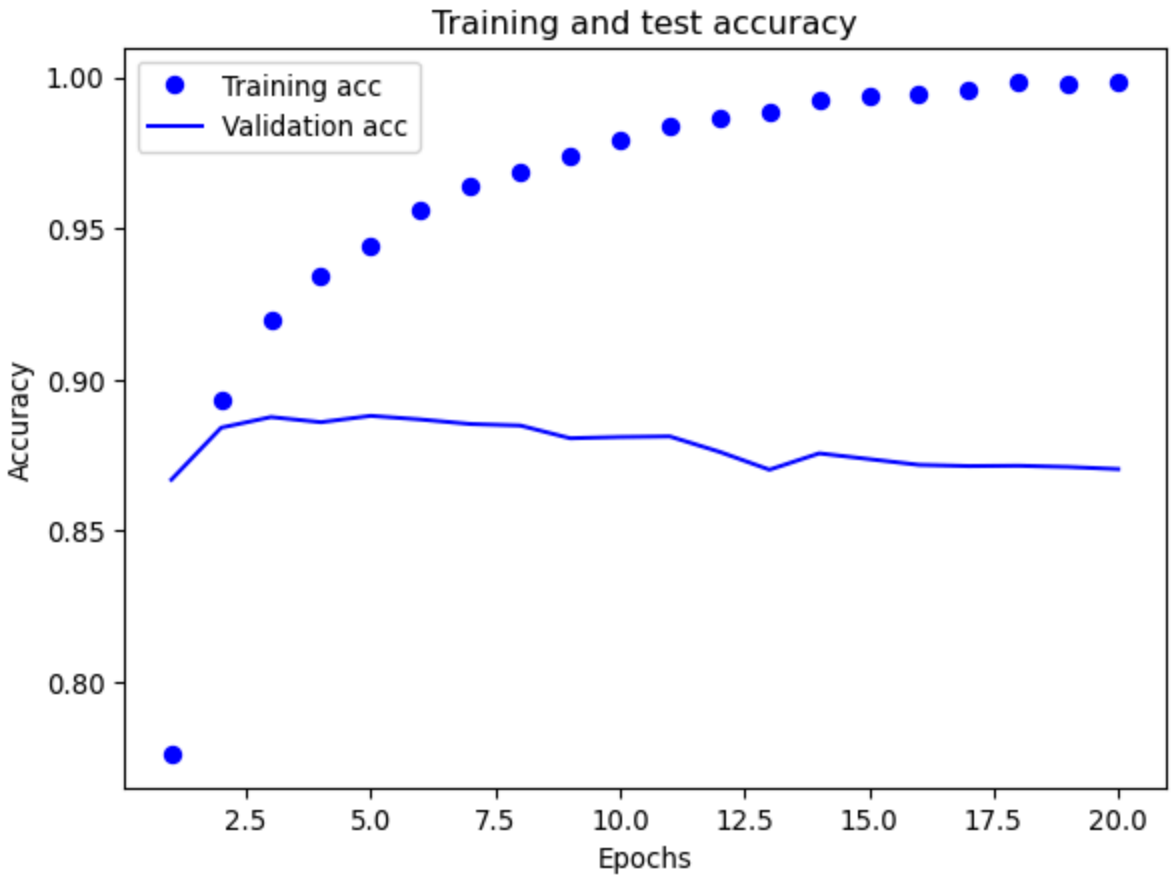
I selected this example to illustrate that a neural network with with sufficient capacity (number of parameters) can minimize training errors almost to zero, as reducing training error is the primary objective of optimization. However, the graph clearly demonstrates that beyond a certain point, the learned patterns become specific to the training set

rather than general principles. Generalization, rather than mere memorization, is the ultimate goal of machine learning.

Overfitting occurs when a model learns the details and noise in the training data to an extent that it negatively impacts the model's performance on new data. This can result in a decision boundary that fits the training data too tightly, capturing noise and irrelevant details rather than general patterns.

Overfitting - Deep Net - Accuracy

In [19]:



This graph similarly illustrates the variation in accuracy for both the training and test sets as the model undergoes training.

Bias-Variance Tradeoff

Bias

- **Bias** refers to the error introduced by approximating a real-world problem, which may be complex, using a simplified model.

- It represents the **difference** between the **average prediction** of the model and the **true outcome**.
- **High bias** can cause an algorithm to **miss important patterns**, leading to **underfitting**.

Bias

$$ \text{Bias}(\hat{f}) = \mathbb{E}[\hat{f}(x)] - f(x) $$ where:

- $\hat{f}(x)$ is the prediction made by the model,
- $f(x)$ is the true function,
- $\mathbb{E}[\hat{f}(x)]$ is the expected prediction over different datasets.

Variance

- **Variance** measures the model's **sensitivity to fluctuations in the training data**.
- **High variance** indicates that the model is **capturing noise** as if it were a true pattern, leading to **overfitting**.
- It reflects how **much the predictions of the model would vary** if **different training data** were used.

Variance

$$ \text{Variance}(\hat{f}) = \mathbb{E}[(\hat{f}(x) - \mathbb{E}[\hat{f}(x)])^2] $$

where:

- $\hat{f}(x)$ is the prediction made by the model,
- $\mathbb{E}[\hat{f}(x)]$ is the expected prediction over different datasets.

Remarks

- **Statistical learning** makes assumptions about the model, data distribution, and noise to analytically derive expected values.
- In practical applications, empirical techniques like **cross-validation** and **bootstrapping** are employed to estimate bias and variance.

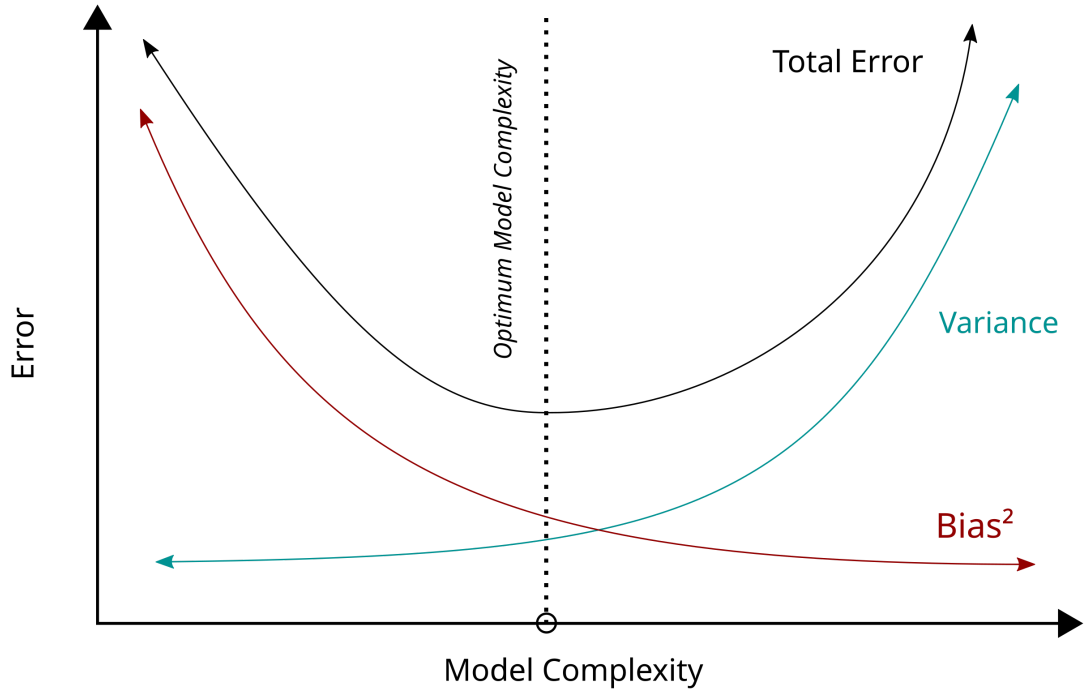
Bais-Variance Tradeoff

$$ \text{Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error} $$

- Model selection aims to **minimize bias**, which arises from overly *simplistic models*, and **variance**, which results from overly *complex models prone to overfitting*.
- Ideally, with **infinite data**, both **bias** and **variance** could be **reduced to zero**.
- However, in practice, **data is typically noisy**, and some irreducible error persists due to unaccounted factors beyond the model's scope.

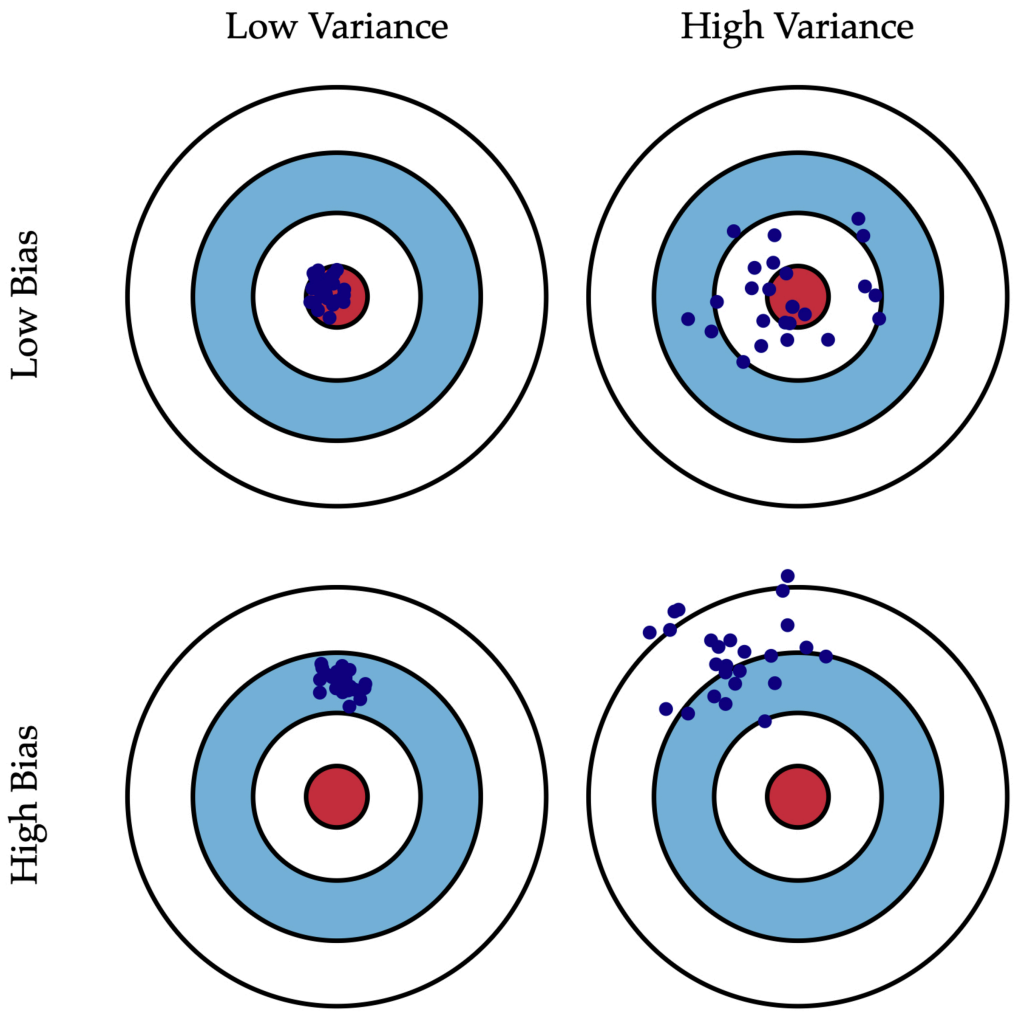
Strive for a model complexity that captures essential patterns without tipping into overfitting.

Bais-Variance Tradeoff



Attribution: Bigbossfarin, CC0, via Wikimedia Commons

Bais-Variance Tradeoff

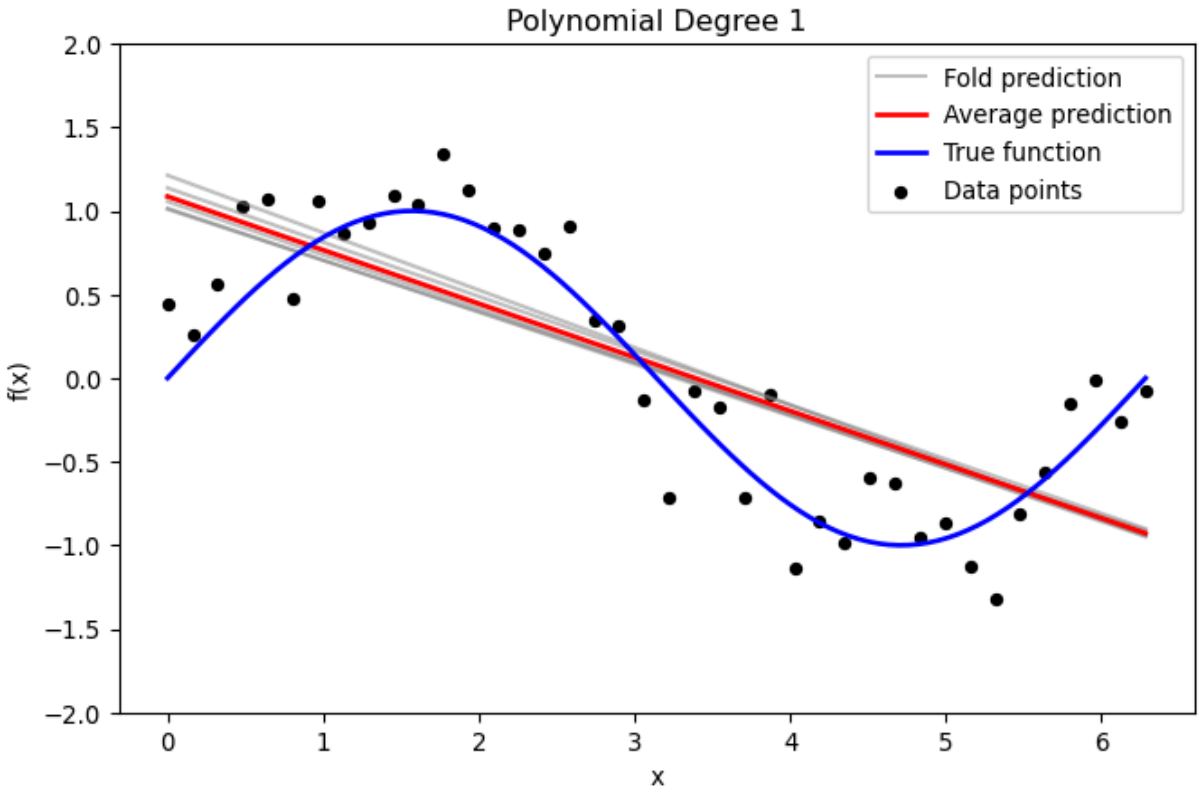


Attribution: Understanding the Bias-Variance Tradeoff by Scott Fortmann-Roe, June 2012.

High Bias, Low Variance

In [20]:
```
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error

def true_function(x):
    return np.sin(x)

def plot_fold_predictions(degree, X, y, X_grid, y_true_grid, n_splits=5, rar
    """
    For a given polynomial degree, perform KFold cross-validation,
    plot the individual fold predictions along with the average prediction
    and the true function (with y-axis limited to [-2, 2]),
    and return predictions and errors.
    """
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=random_state)
    fold_predictions = []  # To store predictions on the evaluation grid for
    fold_errors = []       # To store test errors for each fold
```

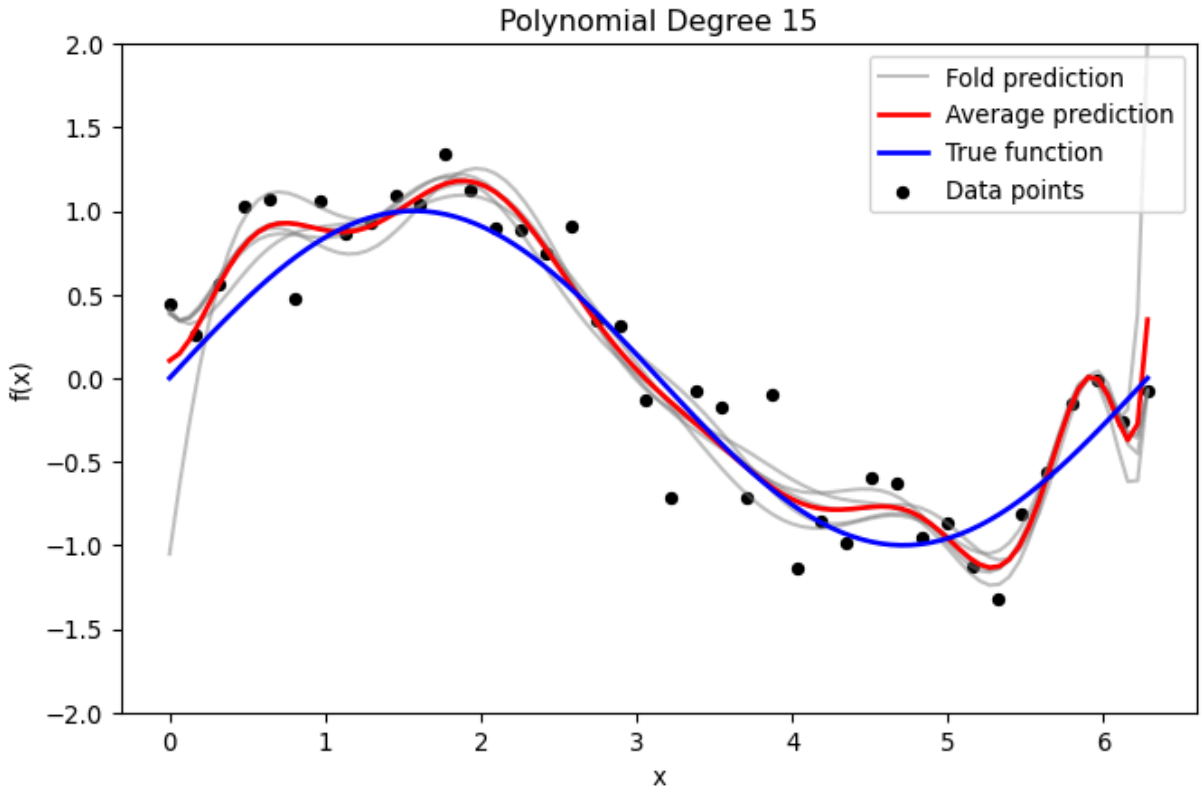
```
    for train_index, test_index in kf.split(X):
        poly = PolynomialFeatures(degree=degree)
        X_train_poly = poly.fit_transform(X[train_index])
        X_test_poly = poly.transform(X[test_index])
        X_grid_poly = poly.transform(X_grid)

        model = LinearRegression()
        model.fit(X_train_poly, y[train_index])

        # Predictions on the dense grid for bias-variance analysis
        y_pred_grid = model.predict(X_grid_poly)
        fold_predictions.append(y_pred_grid)

        # Test error on held-out data
        y_pred_test = model.predict(X_test_poly)
        fold_errors.append(mean_squared_error(y[test_index], y_pred_test))

    fold_predictions = np.array(fold_predictions)
    avg_prediction = np.mean(fold_predictions, axis=0)

    # Plot individual fold predictions with y-axis limited to [-2, 2]
    plt.figure(figsize=(8, 5))
    for i in range(n_splits):
        plt.plot(X_grid, fold_predictions[i], color='gray', alpha=0.5,
                 label='Fold prediction' if i == 0 else "")
    plt.plot(X_grid, avg_prediction, color='red', linewidth=2, label='Averag
    plt.plot(X_grid, y_true_grid, color='blue', linewidth=2, label='True fur
    plt.scatter(X, y, color='black', s=20, label='Data points')
    plt.ylim(-2, 2)
    plt.title(f'Polynomial Degree {degree}')
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.legend()
    plt.show()

    return fold_predictions, avg_prediction, fold_errors

# --- Data Generation with Increased Noise and Reduced Sample Size ---
np.random.seed(0)
n_samples = 40  # Reduced sample size increases model sensitivity to trainir
X = np.linspace(0, 2 * np.pi, n_samples).reshape(-1, 1)
noise_std = 0.25  # Increased noise level amplifies prediction variability
y = true_function(X).ravel() + np.random.normal(0, noise_std, size=n_samples

# Create a dense evaluation grid and compute the true function values
X_grid = np.linspace(0, 2 * np.pi, 100).reshape(-1, 1)
y_true_grid = true_function(X_grid).ravel()

# --- Plot Individual Fold Predictions for Selected Degrees ---
_ = plot_fold_predictions(1, X, y, X_grid, y_true_grid, n_splits=5)
```



- **High Bias**: The model lacks complexity and, consequently, fails to capture the underlying patterns in the data, resulting in a high average error.
- **Low Variance**: The model exhibits insensitivity to variations in the dataset, which manifests as low variance.

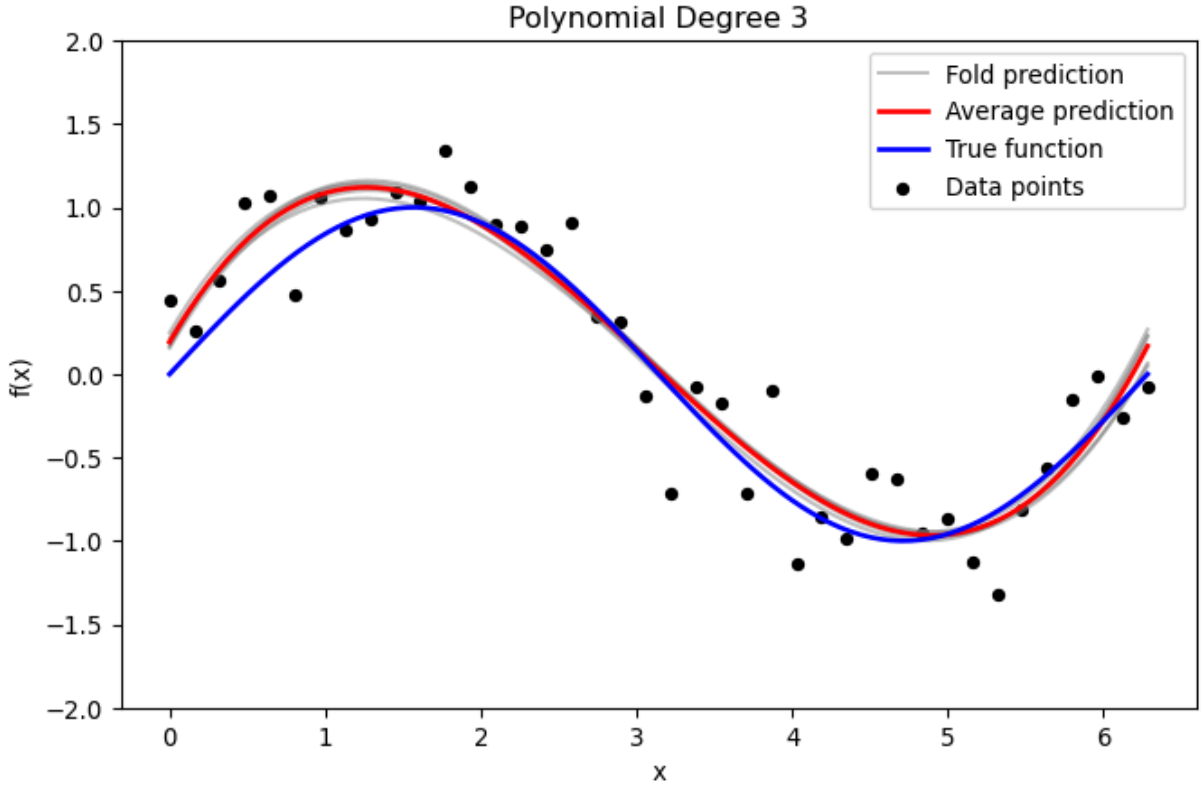
Low Bias, High Variance

In [21]:
```
_ = plot_fold_predictions(15, X, y, X_grid, y_true_grid, n_splits=5)
```



Just Right

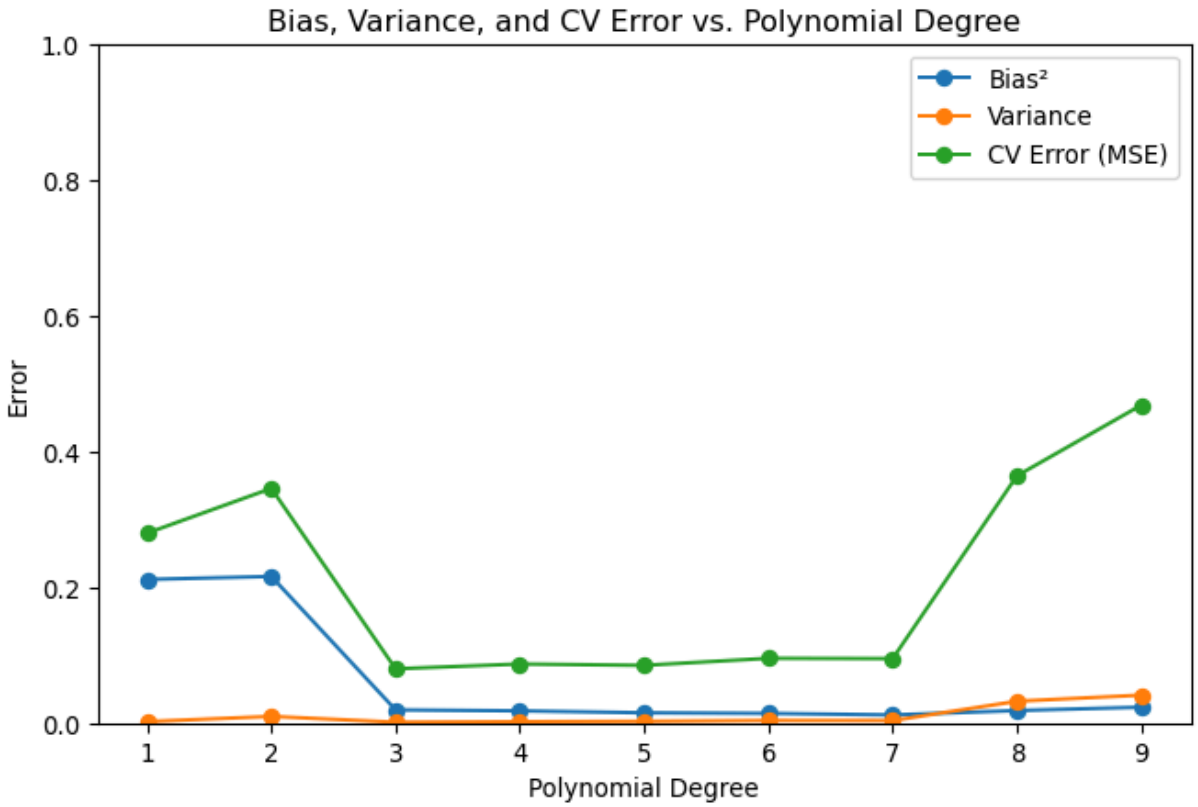
```
In [22]: _ = plot_fold_predictions(3, X, y, X_grid, y_true_grid, n_splits=5)
```



Bias, Variance, and CV Error

In [23]:

```
# --- Compute Bias², Variance, and CV Error Across Degrees 1 to 10 ---
degrees = range(1, 10)
bias_list = []
variance_list = []
cv_error_list = []

for degree in degrees:
    kf = KFold(n_splits=5, shuffle=True, random_state=42)
    fold_predictions = []
    fold_errors = []

    for train_index, test_index in kf.split(X):
        poly = PolynomialFeatures(degree=degree)
        X_train_poly = poly.fit_transform(X[train_index])
        X_test_poly = poly.transform(X[test_index])
        X_grid_poly = poly.transform(X_grid)

        model = LinearRegression()
        model.fit(X_train_poly, y[train_index])

        y_pred_grid = model.predict(X_grid_poly)
        fold_predictions.append(y_pred_grid)

        y_pred_test = model.predict(X_test_poly)
        fold_errors.append(mean_squared_error(y[test_index], y_pred_test))

    fold_predictions = np.array(fold_predictions)
    mean_prediction = np.mean(fold_predictions, axis=0)

    # Bias²: Average squared difference between the average prediction and t
    bias_sq = np.mean((mean_prediction - y_true_grid)**2)
    # Variance: Average variance of the predictions across the evaluation gr
    variance = np.mean(np.var(fold_predictions, axis=0))
    # CV Error: Mean of the MSE on held-out test sets
    cv_error = np.mean(fold_errors)

    bias_list.append(bias_sq)
    variance_list.append(variance)
    cv_error_list.append(cv_error)

# --- Plot Bias², Variance, and CV Error vs. Polynomial Degree ---
plt.figure(figsize=(8, 5))
plt.plot(degrees, bias_list, marker='o', label='Bias²')
plt.plot(degrees, variance_list, marker='o', label='Variance')
plt.plot(degrees, cv_error_list, marker='o', label='CV Error (MSE)')
plt.title('Bias, Variance, and CV Error vs. Polynomial Degree')
plt.xlabel('Polynomial Degree')
plt.ylabel('Error')
plt.ylim(0, 1)
plt.legend()
plt.show()
```

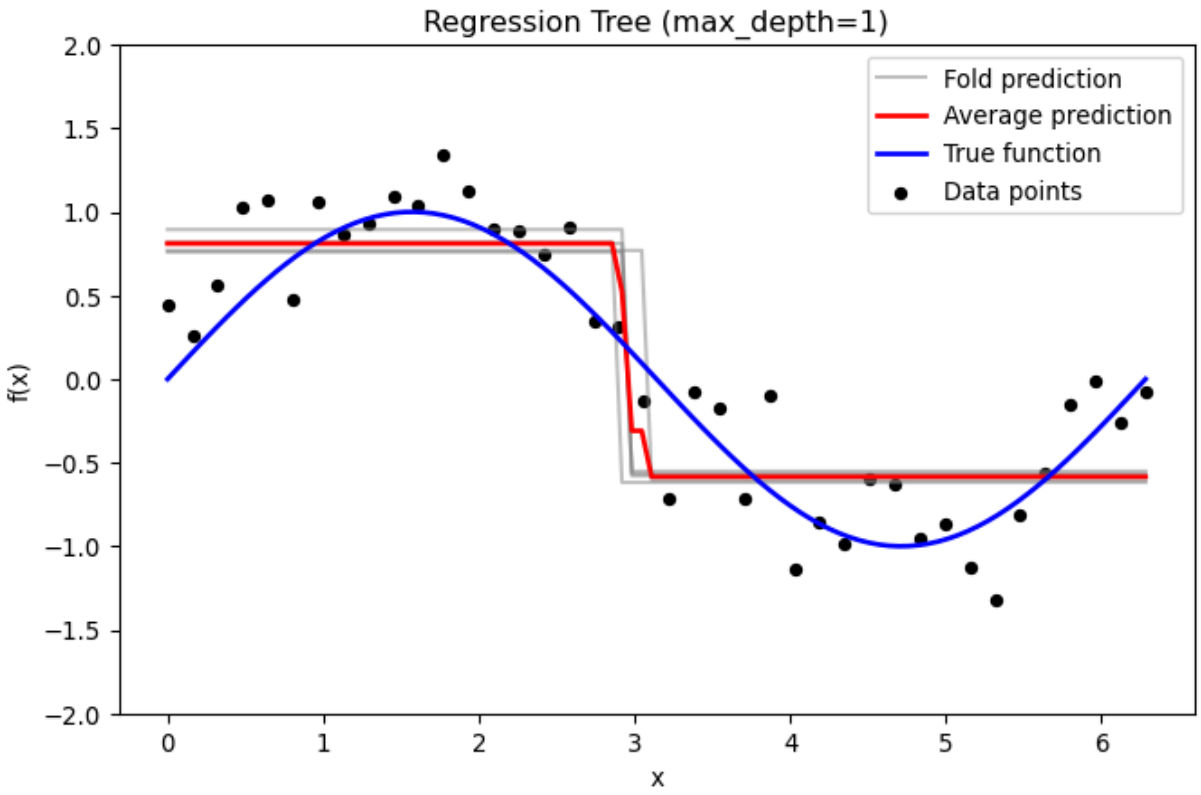


Regression Tree

In [24]:

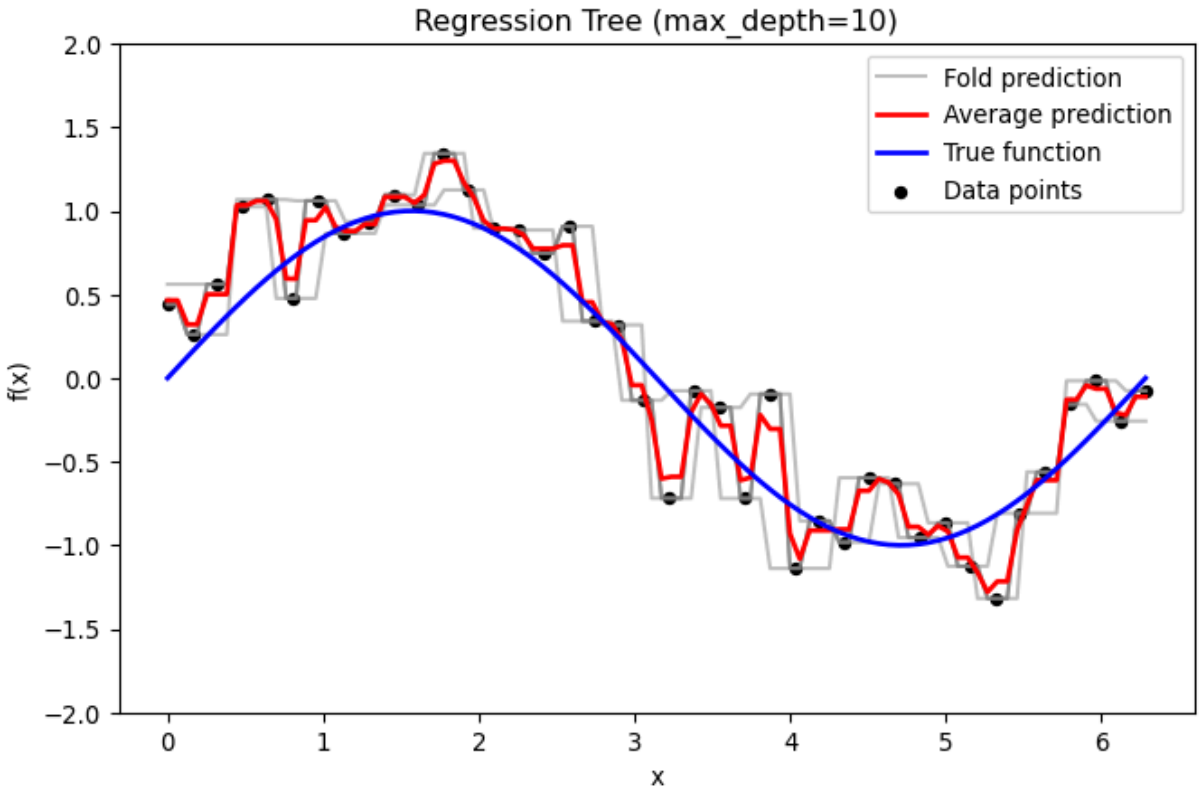
```
from sklearn.tree import DecisionTreeRegressor

def plot_tree_fold_predictions(max_depth, X, y, X_grid, y_true_grid, n_split
    """
    For a given tree max_depth, perform KFold cross-validation with a Decisi
    plot the individual fold predictions along with the average prediction a
    The y-axis is limited to [-2, 2] for clarity.
    """
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=random_state)
    fold_predictions = []  # Store predictions on the evaluation grid for ea
    fold_errors = []       # Store test errors for each fold

    for train_index, test_index in kf.split(X):
        X_train = X[train_index]
        X_test = X[test_index]

        model = DecisionTreeRegressor(max_depth=max_depth, random_state=ranc
        model.fit(X_train, y[train_index])

        # Prediction on a dense evaluation grid
        y_pred_grid = model.predict(X_grid)
        fold_predictions.append(y_pred_grid)

        # Test error on held-out data
        y_pred_test = model.predict(X_test)
        fold_errors.append(mean_squared_error(y[test_index], y_pred_test))
```

```
    fold_predictions = np.array(fold_predictions)
    avg_prediction = np.mean(fold_predictions, axis=0)

    plt.figure(figsize=(8, 5))
    for i in range(n_splits):
        plt.plot(X_grid, fold_predictions[i], color='gray', alpha=0.5,
                 label='Fold prediction' if i == 0 else "")
    plt.plot(X_grid, avg_prediction, color='red', linewidth=2, label='Averag
    plt.plot(X_grid, y_true_grid, color='blue', linewidth=2, label='True fur
    plt.scatter(X, y, color='black', s=20, label='Data points')
    plt.ylim(-2, 2)
    plt.title(f'Regression Tree (max_depth={max_depth})')
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.legend()
    plt.show()

    return fold_predictions, avg_prediction, fold_errors

# --- Plot Individual Fold Predictions for Selected Tree Depths ---
_ = plot_tree_fold_predictions(1, X, y, X_grid, y_true_grid, n_splits=5)
```



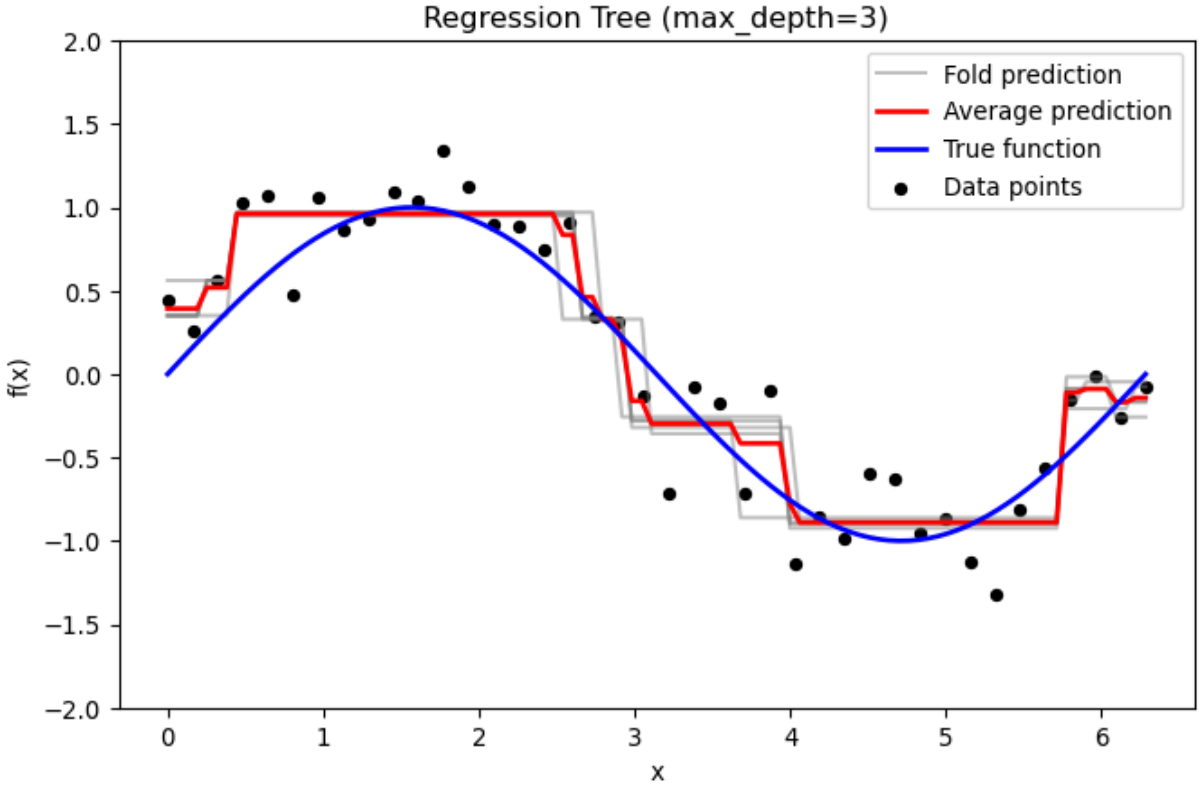
Regression Tree

In [25]:
```
_ = plot_tree_fold_predictions(10, X, y, X_grid, y_true_grid, n_splits=5)
```



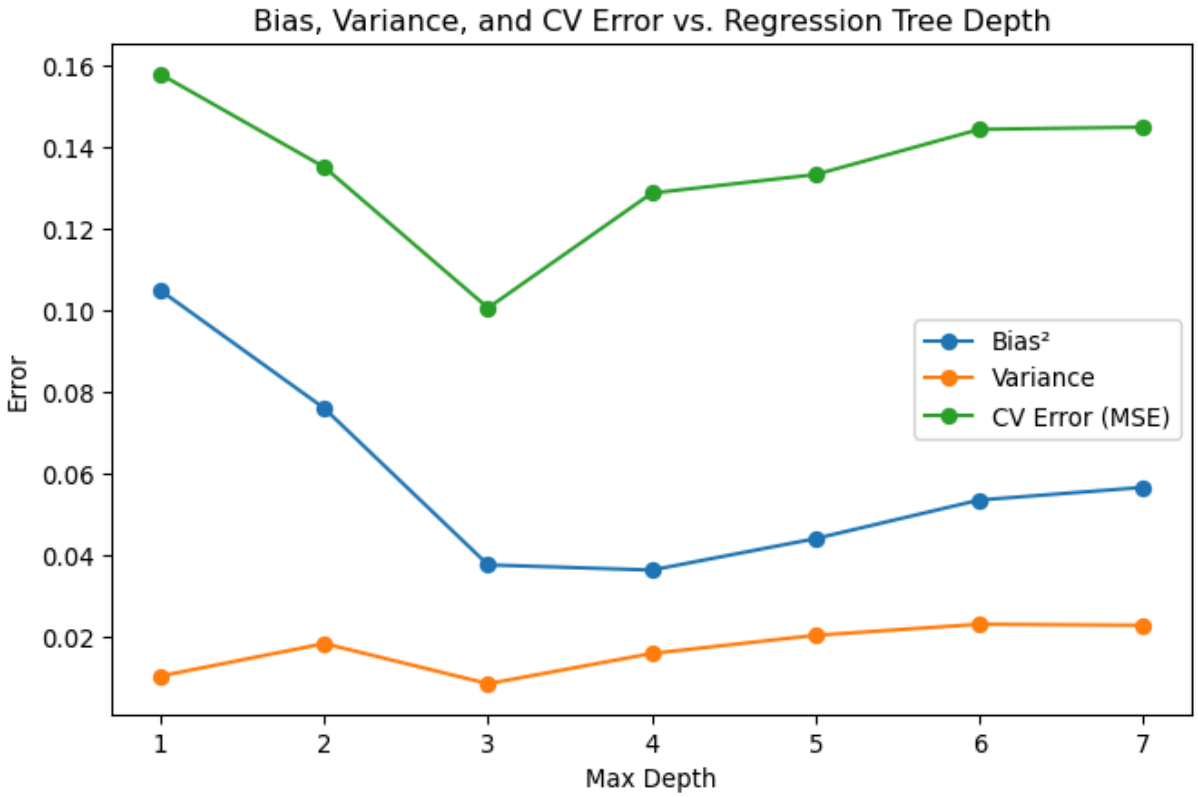
Regression Tree

In [26]:
```
_ = plot_tree_fold_predictions(3, X, y, X_grid, y_true_grid, n_splits=5)
```



Bias, Variance, and CV Error

In [27]:
```
# --- Compute Bias², Variance, and CV Error vs. Tree Depth ---
max_depths = range(1, 8)
bias_list = []
variance_list = []
cv_error_list = []

for depth in max_depths:
    kf = KFold(n_splits=5, shuffle=True, random_state=42)
    fold_predictions = []
    fold_errors = []

    for train_index, test_index in kf.split(X):
        X_train = X[train_index]
        X_test = X[test_index]

        model = DecisionTreeRegressor(max_depth=depth, random_state=42)
        model.fit(X_train, y[train_index])

        y_pred_grid = model.predict(X_grid)
        fold_predictions.append(y_pred_grid)

        y_pred_test = model.predict(X_test)
        fold_errors.append(mean_squared_error(y[test_index], y_pred_test))

    fold_predictions = np.array(fold_predictions)
    mean_prediction = np.mean(fold_predictions, axis=0)

    # Bias²: Mean squared difference between the average prediction and the
    bias_sq = np.mean((mean_prediction - y_true_grid)**2)
    # Variance: Average variance of predictions across the evaluation grid
    variance = np.mean(np.var(fold_predictions, axis=0))
    # CV Error: Average test error over folds
    cv_error = np.mean(fold_errors)

    bias_list.append(bias_sq)
    variance_list.append(variance)
    cv_error_list.append(cv_error)

plt.figure(figsize=(8, 5))
plt.plot(max_depths, bias_list, marker='o', label='Bias²')
plt.plot(max_depths, variance_list, marker='o', label='Variance')
plt.plot(max_depths, cv_error_list, marker='o', label='CV Error (MSE)')
plt.title('Bias, Variance, and CV Error vs. Regression Tree Depth')
plt.xlabel('Max Depth')
plt.ylabel('Error')
# plt.ylim(0, 1)
plt.legend()
plt.show()
```

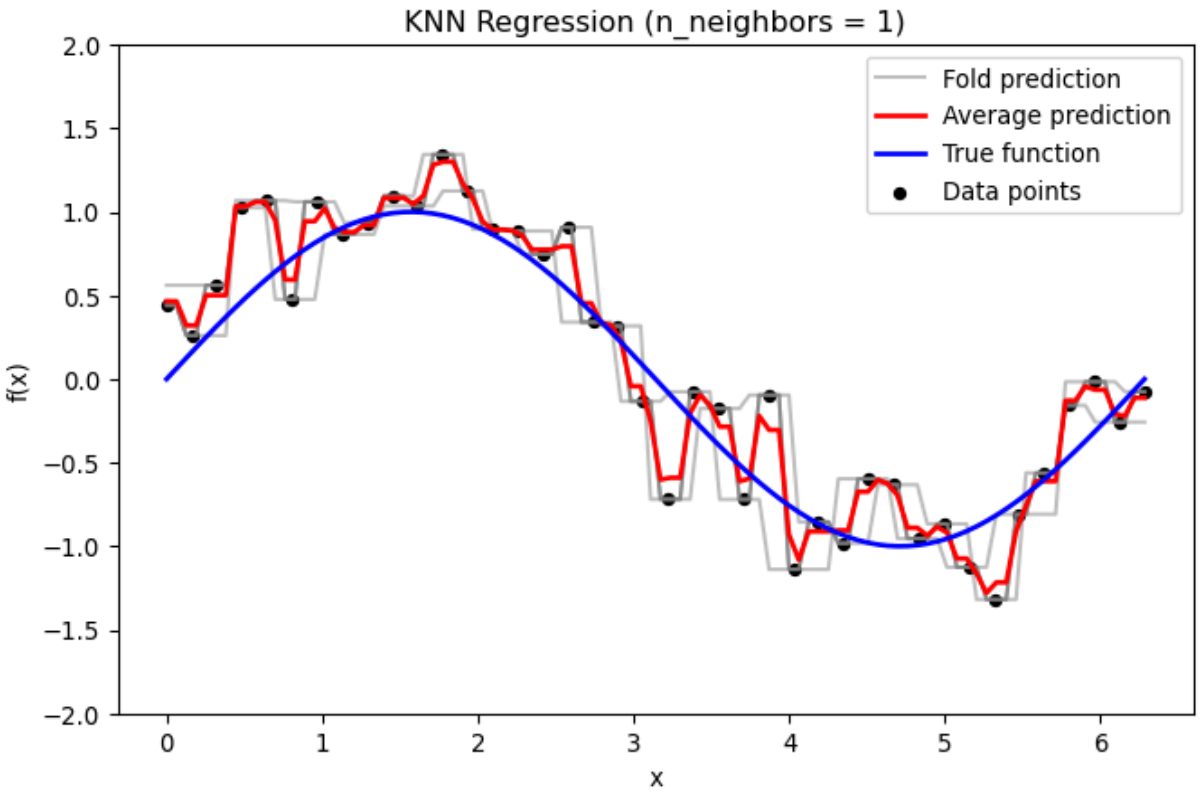


KNN Regression

In [28]:

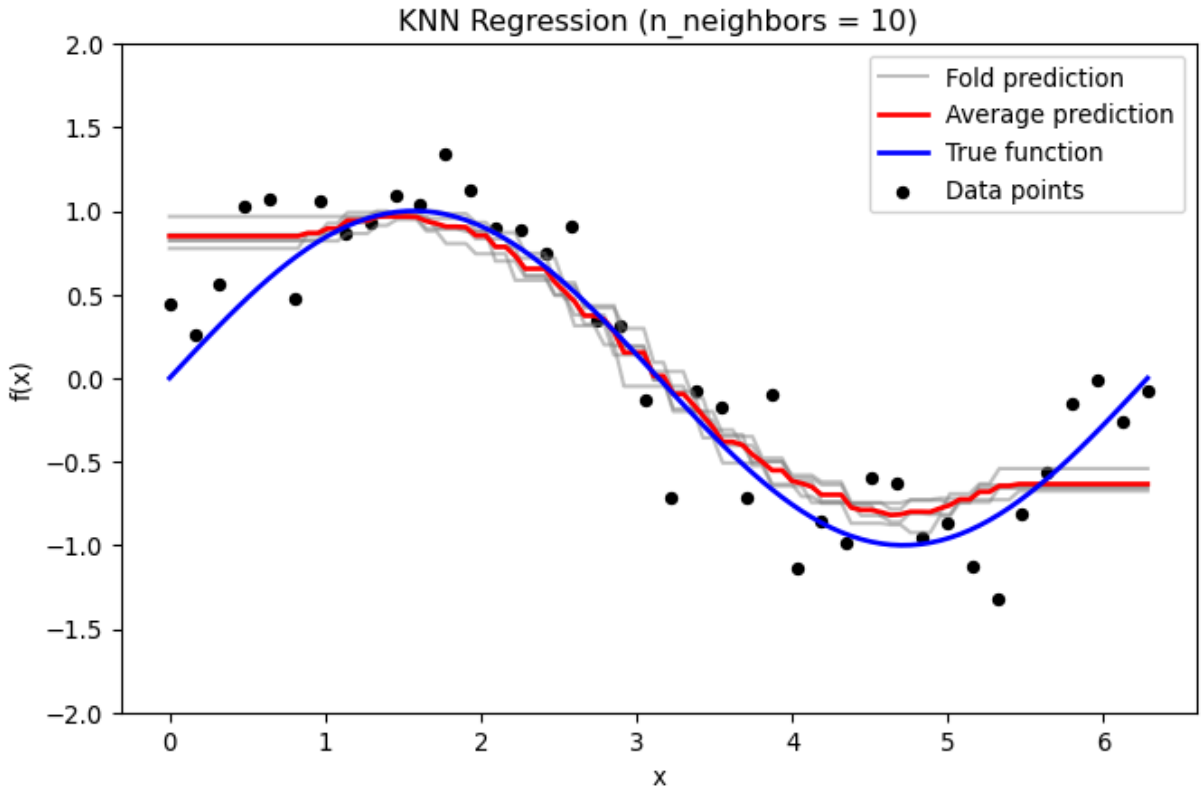
```
from sklearn.neighbors import KNeighborsRegressor

def plot_knn_fold_predictions(n_neighbors, X, y, X_grid, y_true_grid, n_spli
    """
    For a given number of neighbors, perform KFold cross-validation using KN
    plot the predictions from each fold along with the average prediction ar
    The y-axis is limited to [-2, 2] for clarity.
    """
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=random_state)
    fold_predictions = []  # Store predictions on the evaluation grid for ea
    fold_errors = []       # Store test errors for each fold

    for train_index, test_index in kf.split(X):
        X_train = X[train_index]
        X_test = X[test_index]

        model = KNeighborsRegressor(n_neighbors=n_neighbors)
        model.fit(X_train, y[train_index])

        # Prediction on a dense evaluation grid for bias-variance analysis
        y_pred_grid = model.predict(X_grid)
        fold_predictions.append(y_pred_grid)

        # Test error on held-out data
        y_pred_test = model.predict(X_test)
        fold_errors.append(mean_squared_error(y[test_index], y_pred_test))
```

```
    fold_predictions = np.array(fold_predictions)
    avg_prediction = np.mean(fold_predictions, axis=0)

    # Plot individual fold predictions
    plt.figure(figsize=(8, 5))
    for i in range(n_splits):
        plt.plot(X_grid, fold_predictions[i], color='gray', alpha=0.5,
                 label='Fold prediction' if i == 0 else "")
    plt.plot(X_grid, avg_prediction, color='red', linewidth=2, label='Averag
    plt.plot(X_grid, y_true_grid, color='blue', linewidth=2, label='True fur
    plt.scatter(X, y, color='black', s=20, label='Data points')
    plt.ylim(-2, 2)
    plt.title(f'KNN Regression (n_neighbors = {n_neighbors})')
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.legend()
    plt.show()

    return fold_predictions, avg_prediction, fold_errors

# --- Plot Individual Fold Predictions for Selected Values of k ---
_ = plot_knn_fold_predictions(1, X, y, X_grid, y_true_grid, n_splits=5)
```



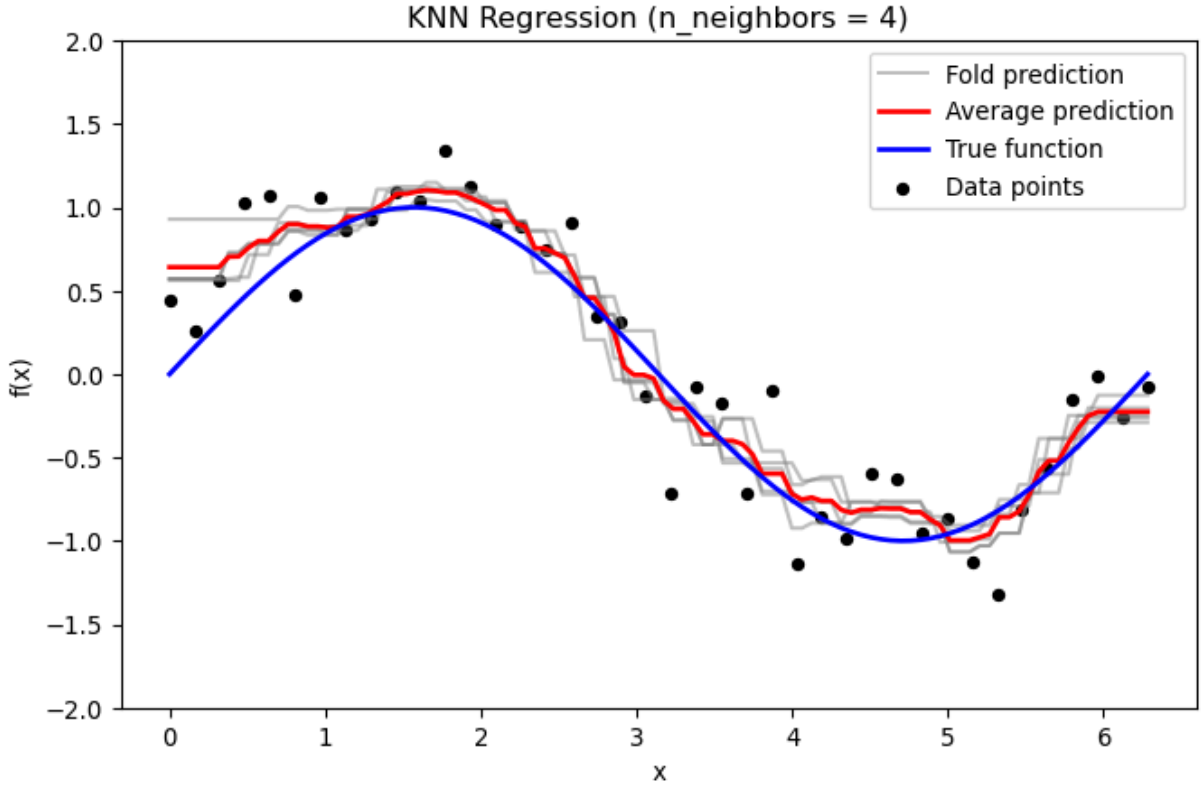
KNN Regression

In [29]:
```
_ = plot_knn_fold_predictions(10, X, y, X_grid, y_true_grid, n_splits=5)
```



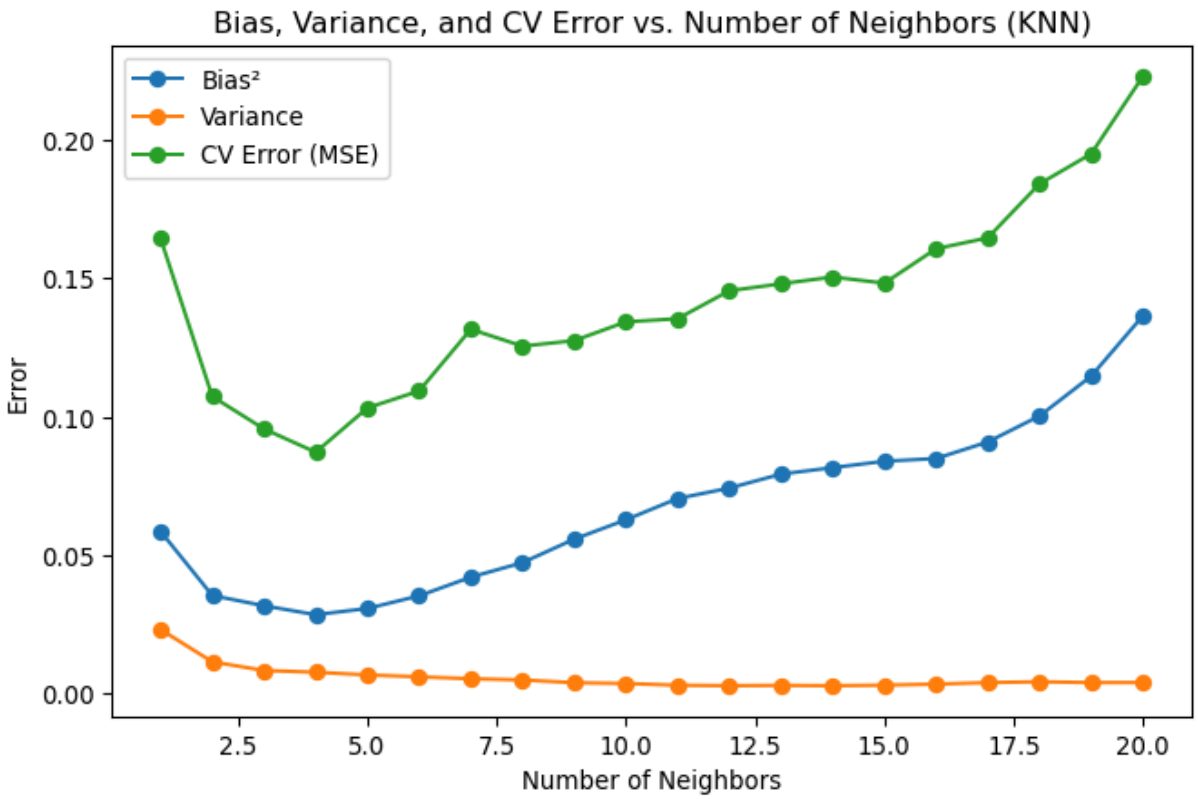
KNN Regression

In [30]:
```
_ = plot_knn_fold_predictions(4, X, y, X_grid, y_true_grid, n_splits=5)
```



Bias, Variance, and CV Error

In [31]:
```python
# --- Compute Bias², Variance, and CV Error vs. Number of Neighbors ---
neighbors_range = range(1, 21)  # Vary k from 1 to 20
bias_list = []
variance_list = []
cv_error_list = []

for k in neighbors_range:
    kf = KFold(n_splits=5, shuffle=True, random_state=42)
    fold_predictions = []
    fold_errors = []

    for train_index, test_index in kf.split(X):
        X_train = X[train_index]
        X_test = X[test_index]

        model = KNeighborsRegressor(n_neighbors=k)
        model.fit(X_train, y[train_index])

        y_pred_grid = model.predict(X_grid)
        fold_predictions.append(y_pred_grid)

        y_pred_test = model.predict(X_test)
        fold_errors.append(mean_squared_error(y[test_index], y_pred_test))

    fold_predictions = np.array(fold_predictions)
    mean_prediction = np.mean(fold_predictions, axis=0)

    # Bias²: Mean squared difference between the average prediction and the
    bias_sq = np.mean((mean_prediction - y_true_grid)**2)
    # Variance: Average variance of predictions across the evaluation grid
    variance = np.mean(np.var(fold_predictions, axis=0))
    # CV Error: Average MSE on the held-out test sets
    cv_error = np.mean(fold_errors)

    bias_list.append(bias_sq)
    variance_list.append(variance)
    cv_error_list.append(cv_error)

# --- Plot Bias², Variance, and CV Error vs. Number of Neighbors ---
plt.figure(figsize=(8, 5))
plt.plot(neighbors_range, bias_list, marker='o', label='Bias²')
plt.plot(neighbors_range, variance_list, marker='o', label='Variance')
plt.plot(neighbors_range, cv_error_list, marker='o', label='CV Error (MSE)')
plt.title('Bias, Variance, and CV Error vs. Number of Neighbors (KNN)')
plt.xlabel('Number of Neighbors')
plt.ylabel('Error')
# plt.ylim(0, 1)
plt.legend()
plt.show()
```



How do you explain that Bias^2 increases with the value of k?

At first, this might seem counterintuitive, but it's a direct consequence of how KNN works.

With a very low k (e.g., $k = 1$), the model makes predictions based on the single closest neighbor, capturing almost every detail of the training data—this yields low bias but high variance because the predictions are highly sensitive to noise.

In contrast, a higher k forces the regressor to average over many neighbors, which smooths out the predictions. This results in a simpler, less flexible model that might not capture local nuances (high bias) but is less sensitive to fluctuations in the training set (low variance).

Prologue

Summary

- Evaluated model complexity and its impact on performance.
- Illustrated underfitting, overfitting, and the bias–variance tradeoff.
- Demonstrated learning curves and cross-validation across diverse models (linear, polynomial, tree, KNN, deep nets).

Next lecture

- Machine Learning Engineering

References

Ambroise, Christophe, and Geoffrey J. McLachlan. 2002. "Selection bias in gene extraction on the basis of microarray gene-expression data." *Proceedings of the National Academy of Sciences* 99 (10): 6562–66. https://doi.org/10.1073/pnas.102102699.

Burkov, A. 2020. *Machine Learning Engineering*. True Positive Incorporated. https://books.google.ca/books?id=HeXizQEACAAJ.

Burkov, Andriy. 2019. *The Hundred-Page Machine Learning Book*. Andriy Burkov.

Chollet, François. 2017. *Deep Learning with Python*. Manning Publications.

Géron, Aurélien. 2022. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 3rd ed. O'Reilly Media, Inc.

Libbrecht, Maxwell W, and William Stafford Noble. 2015. "Machine learning applications in genetics and genomics." *Nature Reviews Genetics* 16 (6): 321–32. https://doi.org/10.1038/nrg3920.

Statnikov, Alexander, Constantin F. Aliferis, Ioannis Tsamardinos, Douglas Hardin, and Shawn Levy. 2004. "A comprehensive evaluation of multicategory classification methods for microarray gene expression cancer diagnosis." *Bioinformatics* 21 (5): 631–43. https://doi.org/10.1093/bioinformatics/bti033.

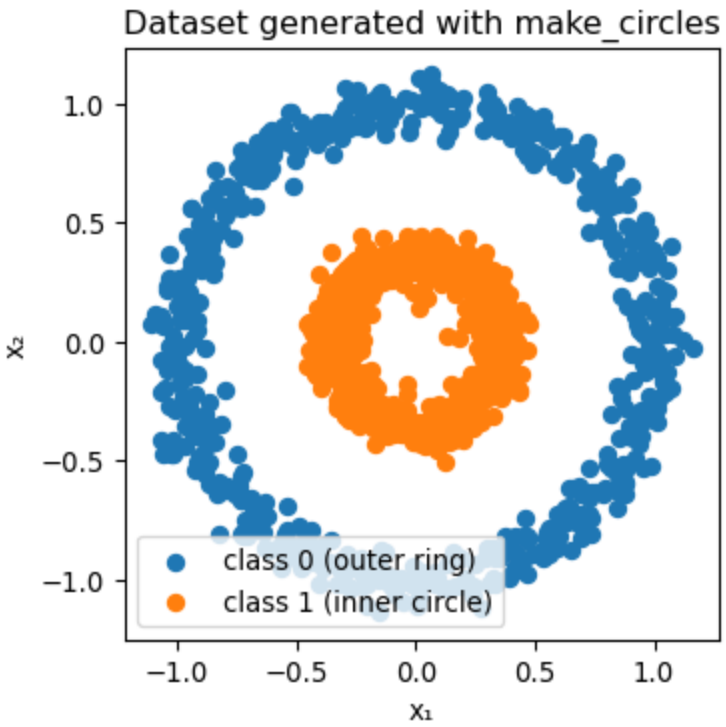
Appendix: Feature Engineering and Classification

Data

Generating a dataset comprising an inner circle (class 1) and an outer ring (class 0).

In [32]:
```
# Generate and plot the "circles" dataset
import matplotlib.pyplot as plt
from sklearn.datasets import make_circles

# Generate synthetic data
X, y = make_circles(n_samples=1200, factor=0.35, noise=0.06, random_state=42

# Separate coordinates for plotting
x1, x2 = X[:, 0], X[:, 1]
```

```
# Plot the two classes
plt.figure(figsize=(4, 4))
plt.scatter(x1[y==0], x2[y==0], color="C0", label="class 0 (outer ring)")
plt.scatter(x1[y==1], x2[y==1], color="C1", label="class 1 (inner circle)")
plt.xlabel("x₁")
plt.ylabel("x₂")
plt.title("Dataset generated with make_circles")
plt.axis("equal") # ensures circles look round
plt.legend()
plt.show()
```



Clearly, this dataset is not linearly separable in (x_1, x_2)!

Feature engineering

When using a linear classifier like `LogisticRegression`, it is not possible to derive parameters that enable accurate classification of the given examples.

We could incorporate two additional features, x_1^2 and x_2^2. This allows for the classification of examples using logistic regression. However, the feature space becomes four-dimensional, complicating direct visualization.

Feature engineering (continued)

In this notebook, we introduce a single feature specifically designed to facilitate visualization. $$ r = x_1^2 + x_2^2, $$ **(r)** represents the **squared distance from the origin** — essentially the **radius squared** in polar coordinates.

Intuition

- Each point in the original 2-D plane has coordinates (x_1, x_2).
- If you express those same coordinates in polar form, you have

$$ x_1 = r^{1/2} \cos\theta, \quad x_2 = r^{1/2} \sin\theta, $$ or more conventionally, $r_{\text{polar}} = \sqrt{x_1^2 + x_2^2}$.

- Here, we define $r = x_1^2 + x_2^2$, i.e., the **square** of that radius.

Using r instead of $\sqrt{r}$ keeps the mapping differentiable and avoids square roots in the model.

Why it's useful

In the "circle vs. ring" dataset:

- Points from the **inner circle** are close to the origin, small r.
- Points from the **outer ring** are farther away, large r.

Thus, the problem that is **non-linear** in (x_1, x_2) becomes **linearly separable** in r:

$$ \text{inner if } r < r^*, \quad \text{outer if } r > r^*. $$

So r is a **feature encoding the radial distance**, allowing a linear model like logistic regression to separate the classes with a single threshold in 1-D.

3D view

In [33]:
```
import numpy as np
from sklearn.datasets import make_circles
from sklearn.linear_model import LogisticRegression
import plotly.graph_objects as go

# --- our new feature ---
r = x1**2 + x2**2

# --- fit logistic on r only and get threshold plane ---

clf = LogisticRegression().fit(r.reshape(-1,1), y)
w = float(clf.coef_[0][0]); b = float(clf.intercept_[0])
r_thresh = -b / w

# --- 3D scatter of (x1, x2, r) ---

scatter = go.Scatter3d(
    x=x1, y=x2, z=r,
    mode="markers",
```

```
    marker=dict(size=3, color=y, colorscale="Viridis", showscale=False),
    hovertemplate="x1=%{x:.3f}<br>x2=%{y:.3f}<br>r=%{z:.3f}<extra></extra>",
    name="points"
)

# --- horizontal plane z = r_thresh ---

gx = np.linspace(x1.min()-0.2, x1.max()+0.2, 50)
gy = np.linspace(x2.min()-0.2, x2.max()+0.2, 50)
GX, GY = np.meshgrid(gx, gy)
GZ = np.full_like(GX, r_thresh)

plane = go.Surface(
    x=GX, y=GY, z=GZ,
    opacity=0.35, showscale=False,
    hoverinfo="skip",
    name="p=0.5 plane"
)

fig = go.Figure(data=[plane, scatter])
fig.update_scenes(
    xaxis_title="x₁", yaxis_title="x₂", zaxis_title="r = x₁² + x₂²",
    aspectmode="cube",
    camera=dict(eye=dict(x=1.6, y=1.6, z=0.9))
)
fig.update_layout(margin=dict(l=0,r=0,b=0,t=30), title=f"Decision plane at r
fig

## Unfotunately, plotly graphical objects can only be visualized in HTML, nc
## TODO: Explore https://plotly.com/python/static-image-export/, perhaps thi
```

Decision plane at r* ≈ 0.545