PSet 02

January 31, 2024

```
[1]: import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     import warnings
     from sklearn.linear_model import LinearRegression
     warnings.filterwarnings("ignore")
```

Mario Garcia and Stephanie Hu

1 Problem 1

1.1 Part 1.2

In part d of problem 1 in the last problem set, the true function f is used to calculate the square-bias, which is used in the MSE decomposition.

1.2 Part 1.3

1.2.1 Subpart A

Without knowing the function f, we can estimate the square-bias as the average squared difference between the predicted value and the true value at the specific data point. We can estimate the true value by using the linear approximation of an y value associated with the x values in the interval $(\tilde{x} - \epsilon, \tilde{x})$ where $\tilde{x}$ is the x value closest to 1.7. I will pick $\epsilon = 0.03$, which further controls the bias-variance tradeoff because a larger ϵ will be "trained" on more points and thus have a lower variance but higher bias.

1.2.2 Subpart B

```
[2]: df = pd.read_csv("ps2data.csv", index_col=0)
```

```
[3]: def polynomial_regression(x, y, x_test, degree):
         coefs = np.polyfit(x, y, degree)
         p = np.poly1d(coefs)
         y_fitted = p(x_test)
         return y_fitted
```

```
[4]: def approximate(df, x, epsilon):
         interval = df[df["x"] > df.iloc[-1]["x"] - epsilon]
```

```
    return LinearRegression().fit(interval["x"].values.reshape(-1,1),␣
↪interval["y"]).predict(np.array([[x]]))[0]
```

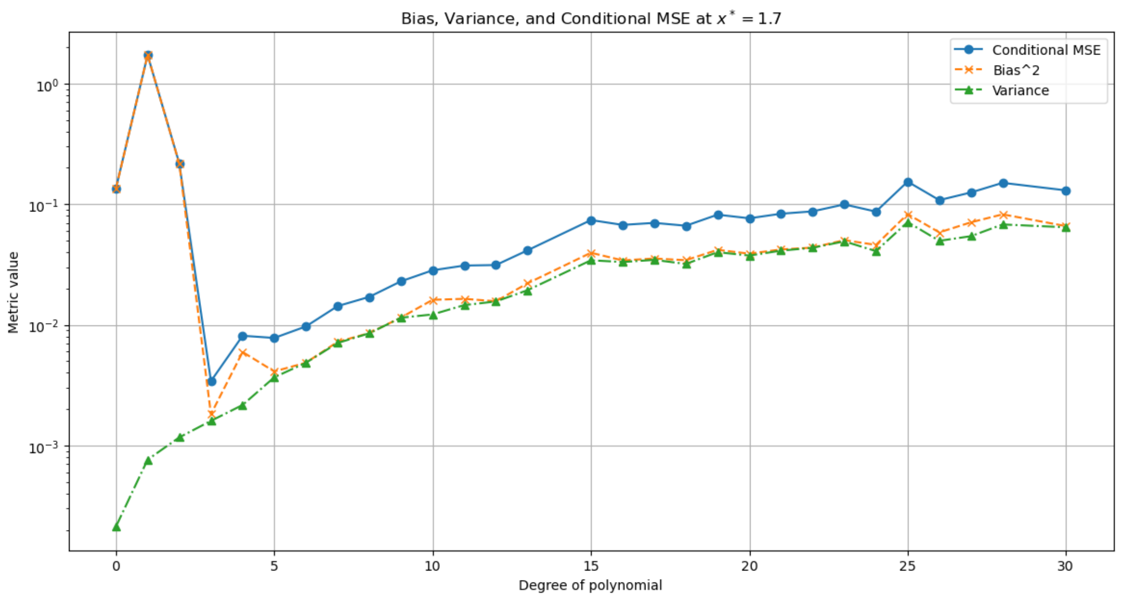
[30]:
```
n_simulations = 100
degrees = np.linspace(0, 30, 29, dtype=int)
# Estimate "true" y using hint
epsilon = 0.1
x_true = 1.7
y_true = approximate(df, x_true, epsilon)
n = len(df)

# Compute Bias and Variance
biases = []
variances = []
conditional_MSEs = []

for d in degrees:
    simulation_predictions = []
    for _ in range(n_simulations):
        sample = df.sample(n, replace=True)
        y_pred = polynomial_regression(sample["x"], sample["y"], x_true, d)
        simulation_predictions.append(y_pred)

    # Bias^2
    bias_sq = np.mean([(pred - y_true) ** 2 for pred in simulation_predictions])
    biases.append(bias_sq)
    # Variance
    variance = np.var(simulation_predictions)
    variances.append(variance)
    # Conditional MSE
    mse = bias_sq + variance
    conditional_MSEs.append(mse)

# Plotting
plt.figure(figsize=(14, 7))
plt.plot(degrees, conditional_MSEs, marker='o', linestyle='-',␣
↪label='Conditional MSE')
plt.plot(degrees, biases, marker='x', linestyle='--', label='Bias^2')
plt.plot(degrees, variances, marker='^', linestyle='-.', label='Variance')
plt.title('Bias, Variance, and Conditional MSE at $x^* = 1.7$')
plt.xlabel('Degree of polynomial')
plt.ylabel('Metric value')
plt.yscale('log')
plt.legend()
plt.grid(True)
plt.show()
```



1.3 Part 1.4

```
[8]: def cross_validate(df, K, func, hyperparameters):
         n = len(df)
         fold_size = n // K
         performance = []

         for hp in hyperparameters:
             total_mse = 0

             for fold in range(K):

                 # Split training and validation set
                 valid_start = fold * fold_size
                 valid_end = (fold + 1) * fold_size
                 validation_set = df.iloc[valid_start:valid_end]
                 training_set = pd.concat([df.iloc[:valid_start], df.iloc[valid_end:
      ↪]])

                 # Train and evaluate model
                 model_pred = func(training_set["x"], training_set["y"],␣
      ↪validation_set["x"], hp)
                 mse = np.mean((model_pred - validation_set["y"]) ** 2)
                 total_mse += mse

             avg_mse = total_mse / K
             performance.append(avg_mse)
```

```
    return performance
```

[9]:
```
performances = cross_validate(df, 10, polynomial_regression, degrees)
best = np.argmin(performances)
best
```

[9]:
```
4
```

The best degree of a polynomial regression for this function using cross-validation would be $d = 4$.

1.4 Part 1.5

Based on the hint in the announcement, we could take two approaches to estimate the MSE for $X^* = 1.7$. 1) **Test sample approach:** We do essentially what we did in part 1.3. We generate bootstrap samples to calculate square-bias and variance and use a linear approximation with a small ϵ to approximate the "true" Y^*. MSE is just the sum of the square-bias and variance. 2) **Optimism correction approach:** In class, we have shown that we can get the conditional MSE from the in-sample MSE by correcting it by a term called the "optimism" of the model, which is the difference between the error with the training set and the validation set. This correction makes sense because the in-sample MSE will systematically underestimate the true MSE since the bias term in-sample is essentially zero.

We prefer the second approach because it avoids having to use linear approximation to estimate the true Y^*, which significantly affects the bias calculation.

1.5 Part 1.6

[28]:
```
def mse_cross(df, K, func, degree):
    n = len(df)
    fold_size = n // K
    performance = []

    train_mse = 0
    validate_mse = 0
    optimism = [None] * K

    for fold in range(K):

        # Split training and validation set
        valid_start = fold * fold_size
        valid_end = (fold + 1) * fold_size
        validation_set = df.iloc[valid_start:valid_end]
        training_set = pd.concat([df.iloc[:valid_start], df.iloc[valid_end:]])

        # Train and evaluate model on training set
        model_pred = func(training_set["x"], training_set["y"],␣
 ↪training_set["x"], degree)
        mse = np.mean((model_pred - training_set["y"]) ** 2)
        train_mse += mse

        # Train and evaluate model on validation set
        model_pred = func(training_set["x"], training_set["y"],␣
↪validation_set["x"], degree)
        mse = np.mean((model_pred - validation_set["y"]) ** 2)
        validate_mse += mse

        opt = validate_mse - train_mse
        optimism[fold] = opt

    avg_opt = np.mean(optimism)

    in_sample_pred = func(df["x"], df["y"], df["x"], degree)
    in_sample_mse = np.mean((in_sample_pred - df["y"]) ** 2)

    return in_sample_mse - avg_opt
```

[29]:
```
mse_cross(df, 10, polynomial_regression, 4)
```

[29]:
```
0.6151601801896038
```

1.6 Part 1.7

Our guess: $6 \sin\left(\frac{4}{3}x\right)$

2 Problem 2

To show that the expected optimism shown in class also applies to 0-1 loss when Y is binary, we just have to show the following to be true:

$$L_{0-1} = L_{MSE}$$

We can begin by expanding the definition for MSE as follows:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{N}(y_i^2 - 2y_i\hat{y}_i + \hat{y}_i^2)$$

Now, since y is binary, $y^2 = y$. (When $y = 0$, $y^2 = 0$ as well; same for when $y = 1$, y^2=1.) We can rewrite the MSE as follows:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - 2y_i\hat{y}_i + \hat{y}_i^2)$$

Now let's consider the two potential cases for $y = 0$ and $y = 1$. 1) When $y = 0$: $y_i - 2y_i\hat{y}_i + \hat{y}_i^2 = 0 - 0 + \hat{y}_i^2 = \hat{y}_i^2$ As before, whether $\hat{y}$ is 0 or 1, the MSE will be equal to itself. 2) When $y = 1$: $y_i - 2y_i\hat{y}_i + \hat{y}_i^2 = 1 - 2\hat{y}_i + \hat{y}_i^2$ In the reverse of above, when $\hat{y}$ is 0, the MSE is 1 and vice-versa.

Taking these four combinations of y and $\hat{y}$, we can recognize that when $y = \hat{y}$, the MSE is 0, while if they are not, the MSE is 1. This indicator variable can be written as $(y - \hat{y})^2$ (i.e. sameness is 0, differentness is 1), which is exactly the loss-function for MSE.

We can then follow the steps from the proof done in class to show that the expected optimism is the same.

3 Problem 3

3.1 Part 3.1

[31]:
```
import numpy as np
import pandas as pd

# 1000 individuals
n = 1000

#
# For simplicity, assume the potential outcomes are normally distributed, but␣
↪they could be based on any distribution.
np.random.seed(428)  # for reproducibility
Y0 = np.random.normal(loc=50, scale=10, size=n)  # potential outcome without␣
↪treatment
Y1 = Y0 + np.random.normal(loc=5, scale=2, size=n)  # potential outcome with␣
↪treatment, with some average treatment effect

# Step 3: Define a treatment assignment mechanism that depends on the potential␣
↪outcomes
# For instance, higher potential outcome without treatment might lead to a␣
↪higher probability of receiving treatment.
prob_of_treatment = 1 / (1 + np.exp(-0.1 * (Y1 - Y0)))  # 5, Maybe make it␣
↪depend on Y0 and y1, or make it depend on the difference between
# Y1 - Y0,
D = np.random.binomial(1, p=prob_of_treatment)  # treatment assignment

# Step 4: Calculate the observed outcomes based on the treatment assignment
Y_obs = D * Y1 + (1 - D) * Y0  # observed outcomes

# Step 5: Compute the observed average treatment effect
ATE_obs = np.mean(Y_obs[D == 1]) - np.mean(Y_obs[D == 0])

# The true ATE is the average difference between Y1 and Y0
ATE_true = np.mean(Y1 - Y0)
# ATE should be above 5

# Check if the observed ATE approximates the true ATE well
ATE_obs, ATE_true
```

```
# Shouldnt work well, cause treatment depends on Y1
```

[31]:
```
(6.024379886388864, 5.101488699524822)
```

3.2 Part 3.2

[33]:
```
#Assign treatment independently of the potential outcomes
D_independent = np.random.binomial(1, p=0.5, size=n)  # random treatment␣
 ↪assignment with probability 0.5

# Step 3: Generate the observable outcomes using the new treatment assignment
Y_obs_independent = D_independent * Y1 + (1 - D_independent) * Y0  # observed␣
 ↪outcomes

# Step 4: Calculate the observed average treatment effect using the new␣
 ↪treatment assignment
ATE_obs_independent = np.mean(Y_obs_independent[D_independent == 1]) - np.
 ↪mean(Y_obs_independent[D_independent == 0])

# Step 5: Compare the observed ATE to the true ATE
ATE_obs_independent, ATE_true
```

[33]:
```
(4.793604836328292, 5.101488699524822)
```