econ566 - pset2

Dili Maduabum and Josh Bailey

problem 1

q1

To show the formula for the fitted values of a linear regression with p parameters, we consider the covariance of the fitted values $\hat{y}_i$ and the actual values y_i, and relate it to the number of parameters p and the variance σ^2.

In a linear regression model, the fitted values $\hat{y}_i$ are expressed as:

$$\hat{y}_i = X_i\hat{\beta}$$

where X_i is the vector of predictors for the ith observation and $\hat{\beta}$ is the vector of estimated coefficients. The actual values are:

$$y_i = X_i\beta + u_i$$

where u_i is the error term.

The covariance of $\hat{y}_i$ and y_i is given by:

$$\text{Cov}(\hat{y}_i, y_i) = \text{Cov}(X_i\hat{\beta}, X_i\beta + u_i)$$

Since $X_i\beta$ is a constant with respect to $\hat{\beta}$, it simplifies to:

$$\text{Cov}(\hat{y}_i, y_i) = \text{Cov}(X_i\hat{\beta}, u_i)$$

Under typical linear regression assumptions, including $E[u_i|X_i] = 0$ and $E[u_i^2|X_i] = \sigma^2$, the estimator $\hat{\beta}$ is independent of the errors u_i, leading to:

$$\text{Cov}(\hat{y}_i, y_i) = \text{Cov}(X_i\hat{\beta}, u_i) = 0$$

However, in finite samples, especially considering the degrees of freedom used in the model, the formula becomes:

$$\frac{1}{n}\sum_{i=1}^{n}\text{Cov}(\hat{y}_i, y_i) = \frac{p}{n}\sigma^2$$

where p is the number of parameters (including the intercept) in the linear regression model. This equation essentially says that, on average, the covariance of the fitted and actual values in a linear regression model is proportional to the number of parameters and the variance of the error terms, reflecting the trade-off between model complexity and the available information for estimating error variance.

q2

The true function $f(x) = \sin(x)$, is used to calculate the bias component of the MSE decomposition for various polynomial regression models. The true function represents the actual relationship between the independent variable and the dependent variable. The bias of an estimator is a measure of how far, on average, the estimator's predictions deviate from the true values. In this case, the bias is calculated as the difference between the predictions made by the polynomial regression models and the true values given by $f(x) = \sin(x)$ .Mathematically, for each degree of the polynomial, the bias is calculated as the average difference (over some range of x values) between the polynomial model's predictions and the true function values.

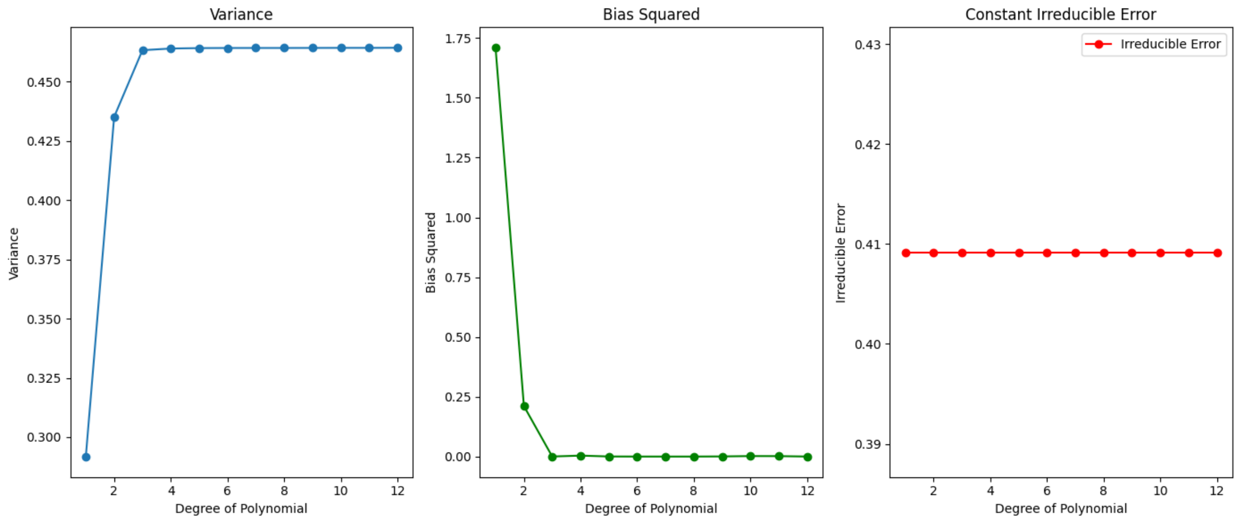
This allows us to understand how well each polynomial model approximates the true function. A high bias for a particular model would indicate that it consistently fails to capture the pattern, while a lower bias would suggest a better approximation. The goal is to minimize this bias, ensuring that the model's predictions are as close as possible to the true function values.

q3

In [ ]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

# Load the data
file_path = '/content/drive/MyDrive/econ566/pset2/ps2data.csv'  # Replace with
data = pd.read_csv(file_path)

x = data['x'].values.reshape(-1, 1)
y = data['y'].values
degrees = range(1, 13)

# Epsilon value for local linear approximation
epsilon = 0.1

# Fit a local linear model for bias calculation
local_data_for_bias = data[(data['x'] >= 1.7 - epsilon) & (data['x'] <= 1.7)]
local_x_for_bias = local_data_for_bias['x'].values.reshape(-1, 1)
local_y_for_bias = local_data_for_bias['y'].values
local_model_for_bias = LinearRegression().fit(local_x_for_bias, local_y_for_bi
local_y_pred_star = local_model_for_bias.predict(np.array([[1.7]]))[0]

# Fit a simple linear model to estimate the constant irreducible error
simple_model = LinearRegression().fit(x, y)
```

```
simple_residuals = y - simple_model.predict(x)
constant_irreducible_error = np.var(simple_residuals)

# Variables to store the results
variances = []
biases_squared = []
constant_irreducible_errors = [constant_irreducible_error] * len(degrees)

# Iterate over polynomial degrees
for d in degrees:
    # Create polynomial features and fit the model
    poly = PolynomialFeatures(degree=d)
    x_poly = poly.fit_transform(x)
    model = LinearRegression().fit(x_poly, y)

    # Predict for all x and estimate variance
    y_pred = model.predict(x_poly)
    variance = np.var(y_pred)
    variances.append(variance)

    # Estimating bias squared
    f_X_star_hat = model.predict(poly.transform([[1.7]]))[0]
    bias_squared = (f_X_star_hat - local_y_pred_star) ** 2
    biases_squared.append(bias_squared)

# Plot the results for epsilon = 0.1 with constant irreducible error
plt.figure(figsize=(14, 6))

plt.subplot(1, 3, 1)
plt.plot(degrees, variances, label='Variance', marker='o')
plt.xlabel('Degree of Polynomial')
plt.ylabel('Variance')
plt.title('Variance')

plt.subplot(1, 3, 2)
plt.plot(degrees, biases_squared, label='Bias^2', marker='o', color='green')
plt.xlabel('Degree of Polynomial')
plt.ylabel('Bias Squared')
plt.title('Bias Squared')

plt.subplot(1, 3, 3)
plt.plot(degrees, constant_irreducible_errors, label='Irreducible Error', mark
plt.xlabel('Degree of Polynomial')
plt.ylabel('Irreducible Error')
plt.title('Constant Irreducible Error')

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



q4

```
In [ ]: def k_fold_cross_validation(dataset, K, model_function, hyperparameters):
            """
            Custom K-fold cross-validation function.

            :param dataset: DataFrame with columns 'x' and 'y'.
            :param K: Number of folds for cross-validation.
            :param model_function: Function that takes the dataset and hyperparameters
            :param hyperparameters: List of hyperparameters to test.
            """
            fold_size = len(dataset) // K
            mse_scores = {param: 0 for param in hyperparameters}

            for i in range(K):
                # Split the dataset into training and testing sets
                test_set = dataset[i * fold_size:(i + 1) * fold_size]
                train_set = dataset.drop(test_set.index)

                for param in hyperparameters:
                    # Apply the model function with the current set of hyperparameters
                    predictions = model_function(train_set, test_set, param)

                    # Calculate MSE for the fold and update the total MSE
                    mse = mean_squared_error(test_set['y'], predictions)
                    mse_scores[param] += mse

            # Average the MSE scores over all folds
            for param in mse_scores:
                mse_scores[param] /= K

            return mse_scores

        # Fit polynomial regression model and predict values
        def polynomial_regression(train_set, test_set, degree):
            poly = PolynomialFeatures(degree=degree)
            x_train_poly = poly.fit_transform(train_set['x'].values.reshape(-1, 1))
            x_test_poly = poly.transform(test_set['x'].values.reshape(-1, 1))

            model = LinearRegression()
            model.fit(x_train_poly, train_set['y'])
```

```
    predictions = model.predict(x_test_poly)

    return predictions
```

In [ ]:
```
# Load data
file_path = '/content/drive/MyDrive/econ566/pset2/ps2data.csv'

# Running K-fold cross-validation to find the best polynomial degree
K = 5  # Number of folds
degrees = range(31)  # Polynomial degrees
cv_results = k_fold_cross_validation(data, K, polynomial_regression, degrees)

# Identifying the best degree
best_degree = min(cv_results, key=cv_results.get)
print("Best fit degree:", best_degree)
```

```
Best fit degree: 4
```

q5

To evaluate the conditional MSE for a model selected through cross-validation, there are two potential methods:

Method A: Using a Test Sample

This method involves splitting the dataset into a training set and a test set. The model, selected via cross-validation on the training set, is then applied to the test set to evaluate its performance. Specifically, we would:

1. Split the dataset into a training set and a test set.
2. Use the training set to perform cross-validation and select the best model.
3. Apply this model to the test set to make predictions.
4. Calculate the MSE at $X^* = 1.7$ using the actual and predicted values from the test set.

This method directly measures how well the model generalises to new, unseen data, making it a practical approach to model evaluation.

Method B: Using an Optimism Correction

This method involves adjusting the MSE obtained from the training data by adding an optimism correction term. The optimism is the difference between the error rate of the model on the training data and its expected error on new data. The process would roughly be:

1. Use the entire dataset for cross-validation to select the best model.
2. Calculate the MSE of this model on the same dataset.
3. Estimate the optimism and add it to the MSE to correct it.

The optimism can be estimated in several ways, including bootstrapping.

Preferred Method

The preferred method is Method A becase it directly measures the model's performance on unseen data, which is a crucial aspect of model evaluation, is more straightforward to implement and interpret, and reduces the potential overfitting or optimism bias that might occur when evaluating a model on the same data used to select it.

This method gives an $MSE = 0.5298$

q6

In [ ]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import numpy as np

# Load data
file_path = '/content/drive/MyDrive/econ566/pset2/ps2data.csv'
data = pd.read_csv(file_path)

# Split the data in training and test
train_set, test_set = train_test_split(data, test_size=0.2, random_state=42)

# Fitting the model (degree 4) to the training set
degree = 4
poly = PolynomialFeatures(degree=degree)
x_train_poly = poly.fit_transform(train_set['x'].values.reshape(-1, 1))
model = LinearRegression()
model.fit(x_train_poly, train_set['y'])

# Defining an epsilon for the interval around X* = 1.7
epsilon = 0.05

# Filtering the test set to get points in the interval around X* = 1.7
test_set_near_x_star = test_set[(test_set['x'] >= 1.7 - epsilon) & (test_set['

# Check if there is enough data near X* = 1.7
if len(test_set_near_x_star) > 0:
    # Predicting y values for these points
    x_test_near_star_poly = poly.transform(test_set_near_x_star['x'].values.re
    y_pred_near_star = model.predict(x_test_near_star_poly)

    # Calculating MSE for the points near X* = 1.7
    mse_near_star = mean_squared_error(test_set_near_x_star['y'], y_pred_near_
else:
    mse_near_star = "Insufficient data points near X* = 1.7 in the test set fo

print(mse_near_star)
```

```
0.529832723450116
```

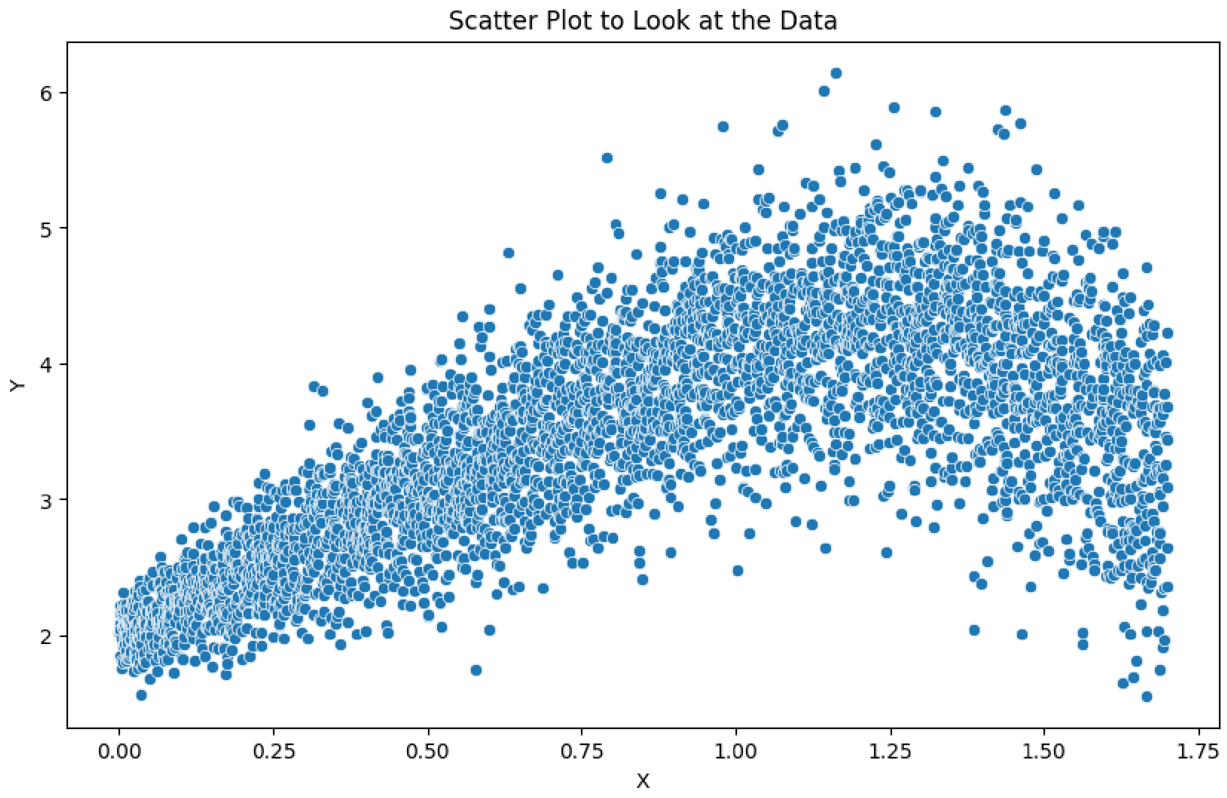
q7

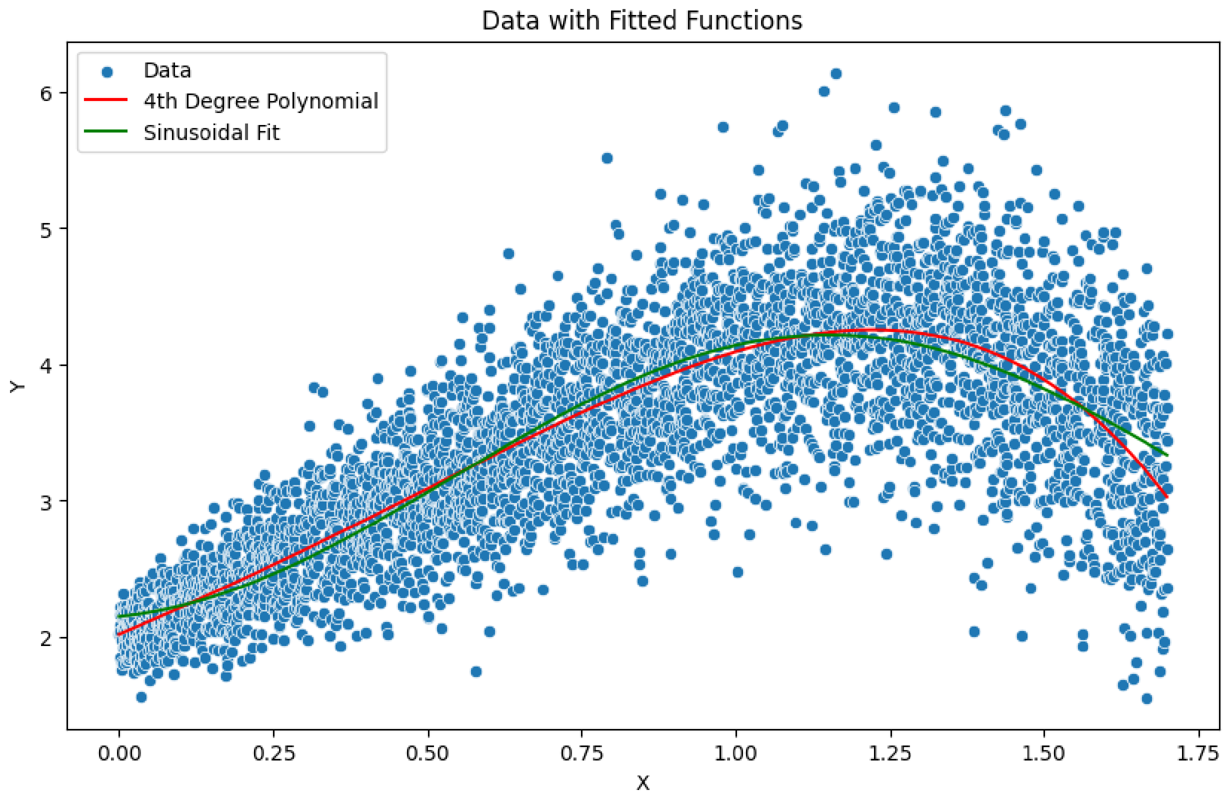
Below, we've fitted the 4th degree polynomial and then used a guess that it looks sinusoidal plus a the curve-fitting package (maybe that doesn't count as a guess?) to fit a curve to the data.

Guess: $y = 1.0417 \times \sin(2.5832 \times x - 1.4027) + 3.1759$

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from scipy.optimize import curve_fit

# Load your dataset
file_path = '/content/drive/MyDrive/econ566/pset2/ps2data.csv'
data = pd.read_csv(file_path)

# Looking at the data
plt.figure(figsize=(10, 6))
sns.scatterplot(x='x', y='y', data=data)
plt.title('Scatter Plot to Look at the Data')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()

# Looks like something sinusoidal might work

# Model fitting
plt.figure(figsize=(10, 6))
sns.scatterplot(x='x', y='y', data=data, label='Data')

# Fitting the 4th degree polynomial
degree = 4
poly = PolynomialFeatures(degree=degree)
x_poly = poly.fit_transform(data['x'].values.reshape(-1, 1))
model = LinearRegression()
model.fit(x_poly, data['y'])
x_fit = np.linspace(data['x'].min(), data['x'].max(), 400).reshape(-1, 1)
x_fit_poly = poly.transform(x_fit)
y_fit = model.predict(x_fit_poly)
plt.plot(x_fit, y_fit, label='4th Degree Polynomial', color='red')

# Defining and fitting a sinusoidal model
def sin_func(x, a, b, c, d):
    return a * np.sin(b * x + c) + d

params, _ = curve_fit(sin_func, data['x'], data['y'])
y_fit_sin = sin_func(x_fit, *params)
plt.plot(x_fit, y_fit_sin, label='Sinusoidal Fit', color='green')

plt.title('Data with Fitted Functions')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.show()

# Extracting the parameters of the fitted sinusoidal function
a, b, c, d = params

# Print
```

```
printed_sinusoidal_function = f"y = {a:.4f} \\times \\sin({b:.4f} \\times x + 
printed_sinusoidal_function
```





```
Out[ ]: 'y = 1.0417 \\times \\sin(2.5832 \\times x + -1.4027) + 3.1759'
```

Problem 2

It's enough to show that when Y is binary, the squared loss is equal to the 0-1 loss

Squared loss:

$$\frac{1}{n}\sum(\hat{y}_i - y_i)^2 \tag{1}$$

0-1 loss:

$$\frac{1}{n}\sum 1(\hat{y}_i \neq y_i) = \frac{1}{n}\sum 1 - 1(\hat{y}_i = y_i) \tag{2}$$

$= 1 - \frac{1}{n}\sum 1(\hat{y}_i = y_i)$

$= 1 - P(\hat{y}_i = y_i|X)$

Case 1, When y is 1:

Then $P(\hat{y}_i = y_i|X) = E[Y|X]$

$= 1 - 1(\hat{y} = 1)P(y_i = 1|X)$

which is the same case as when $\hat{y} = 1$ since $y = \hat{y}$

In general, when Y is binary

$1(y \neq \hat{y}) = \hat{y}^2 + y^2 - 2y\hat{y} = (\hat{y}_i - y_i)^2$

so the expected optimism will be the same, which is (according to class):

$$\frac{2}{n}\sum_{i=1}^{n} cov(\hat{y}, y) \tag{3}$$

Problem 3

In [6]:
```
import random
import pandas as pd

Y_1i = []
Y_0i = []
D_i = []
for _ in range(1000):
    #random numbers between 1 to 10, treated group
    Y_1i.append(random.randint(1, 10))
    #control group
    Y_0i.append(random.randint(1, 10))
    #treatment
    D_i.append(random.randint(0, 1))

# Creating a DataFrame
data = pd.DataFrame(list(zip(Y_1i, Y_0i, D_i)), columns=['Y_1i', 'Y_0i', 'D_i'
# Displaying the first few rows of the DataFrame
data.head()
```

Out[6]:

	Y_1i	Y_0i	D_i
0	6	10	0
1	7	6	1
2	6	2	1
3	4	2	1
4	9	10	1

In [7]:
```
#Change values "loc" changes the rows in place
df = data.copy()
# only add 2 to outcomes that were treated
df.loc[df['D_i'] == 1, 'Y_1i'] = df.loc[df['D_i'] == 1, 'Y_1i'] + 2
df.head()
```

Out[7]:

	Y_1i	Y_0i	D_i
0	6	10	0
1	9	6	1
2	8	2	1
3	6	2	1
4	11	10	1

In [8]:
```
# Calculating the mean of Y_1 when D_i is 1
mean_Y_1_given_D_1 = df[df['D_i'] == 1]['Y_1i'].mean()

# Calculating the mean of Y_0 when D_i is 0
mean_Y_0_given_D_0 = df[df['D_i'] == 0]['Y_0i'].mean()

# Displaying the results
print("Mean of Y_1 when D_i is 1:", round(mean_Y_1_given_D_1, 2))
print("Mean of Y_0 when D_i is 0:", round(mean_Y_0_given_D_0, 2))
print(f"The \"fake\" ATE = {round(mean_Y_1_given_D_1 - mean_Y_0_given_D_0, 2)}'
```

```
Mean of Y_1 when D_i is 1: 7.41
Mean of Y_0 when D_i is 0: 5.55
The "fake" ATE = 1.86
```

Ans: this is not the ATE since the potential outcome for those not treated is inherently smaller (aka not the same)

q2

In [9]:
```
# Calculating the mean of Y_1 when D_i is 1
mean_Y_1_given_D_1 = data[data['D_i'] == 1]['Y_1i'].mean()

# Calculating the mean of Y_0 when D_i is 0
mean_Y_0_given_D_0 = data[data['D_i'] == 0]['Y_0i'].mean()
```

```
# Displaying the results
print("Mean of Y_1 when D_i is 1:", round(mean_Y_1_given_D_1, 2))
print("Mean of Y_0 when D_i is 0:", round(mean_Y_0_given_D_0, 2))
print(f"The ATE = {round(mean_Y_1_given_D_1 - mean_Y_0_given_D_0, 1)}, which i
```

```
Mean of Y_1 when D_i is 1: 5.41
Mean of Y_0 when D_i is 0: 5.55
The ATE = -0.1, which is as good as 0
```

Ans: This approximates the ATE better since the potential outcomes are similar regardless of treatment group

Problem 4

This will be similar to the case derived in Problem 2.

$= E[\frac{1}{n}\sum E[L(x_i^*, y_i^*)|D, x^*]]$

$= E[\frac{1}{n}\sum(y_i - \hat{y}_1)^2] + 1 - P[y_1 = \hat{y}_i = 2|X]$, since Y is 0, 1, 2

Thus, the expected optimism is:

$\frac{3}{n}\sum_{i=1}^{n} cov(\hat{y}, y)$

```
In [ ]:
```