Homework 1: Error Decomposition & Polynomial Regression

Due: Wednesday, February 2, 2022 at 11:59pm

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g.LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better. The last application is optional.

General considerations (10 Points)

For the first part of this assignment we will consider a synthetic prediction problem to develop our intuition about the error decomposition. Consider the random variables $x \in \mathcal{X} = [0, 1]$ distributed uniformely ($x \sim \text{Unif}([0, 1])$) and $y \in \mathcal{Y} = \mathbb{R}$ defined as a polynomial of degree 2 of x: there exists $(a_0, a_1, a_2) \in \mathbb{R}^3$ such that the values of x and y are linked as $y = g(x) = a_0 + a_1 x + a_2 x^2$. Note that this relation fixes the joint distribution $P_{\mathcal{X}\times\mathcal{Y}}$.

From the knowledge of a sample $\{x_i, y_i\}_{i=1}^N$, we would like to predict the relation between x and y, that is find a function f to make predictions $\hat{y} = f(x)$. We note $\mathcal{H}_d$, the set of polynomial functions on $\mathbb{R}$ of degree d: $\mathcal{H}_d = \left\{f : x \to b_0 + bx + \cdots b_d x^d; b_k \in \mathbb{R} \forall k \in \{0, \cdots d\}\right\}$. We will consider the hypothesis classes $\mathcal{H}_d$ varying d. We will minimize the squared loss $\ell(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$ to solve the regression problem.

1. (2 Points) Recall the definition of the expected risk $R(f)$ of a predictor f. While this cannot be computed in general note that here we defined $P_{\mathcal{X}\times\mathcal{Y}}$. Which function f^* is an obvious Bayes predictor? Make sure to explain why the risk $R(f^*)$ is minimum at f^*.

   In this case, a polynomial with degree 2, taking the form $f^* = y = a_0 + a_1 x + a_2 x^2$ would be the obvious choice, as in our problem statement we have defined our function as such. Since we could fit the parameters a_0, a_1, a_2 perfectly to match those in which the function is defined, we would have no loss, as our predictions match the ground truth perfectly. The expected risk would be minimized as

   $$R(f^*) = E(\frac{1}{2}(\hat{y} - y)^2) = 0 \text{ as } \hat{y}_i = y_i$$

2. (2 Points) Using $\mathcal{H}_2$ as your hypothesis class, which function $f^*_{\mathcal{H}_2}$ is a risk minimizer in $\mathcal{H}_2$? Recall the definition of the approximation error. What is the approximation error achieved by $f^*_{\mathcal{H}_2}$?

   Since we are using $\mathcal{H}_2$ as our hypothesis class, this includes all the polynomials of degree two, and thus the Bayes predictor we identified in problem 1. Since the Bayes predictor, a polynomial of degree 2 taking the form $f^* = y = a_0 + a_1 x + a_2 x^2$, is within our hypothesis space, that would be the function we'd use, and thus $f_F = f^*$. Using the definition of Approximation Error:

   $$\textit{Approximation Error} = R(f_F) - R(f^*) = R(f^*) - R(f^*) = 0$$

   Thus, the resulting approximation error would be 0.

3. (2 Points) Considering now $\mathcal{H}_d$, with $d > 2$. Justify an inequality between $R(f^*_{\mathcal{H}_2})$ and $R(f^*_{\mathcal{H}_d})$. Which function $f^*_{\mathcal{H}_d}$ is a risk minimizer in $\mathcal{H}_d$? What is the approximation error achieved by $f^*_{\mathcal{H}_d}$?

   Update from campus wire: the question is worded unclearly, and in our new hypothesis space all polynomials of degree 2 are included. In other words, our new hypothesis space, $R(f^*_{\mathcal{H}_d})$, only includes polynomials of degree d or lower. Since our Bayes predictor is still within our hypothesis space in all cases, we can still achieve minimal risk minimizer in $R(f^*_{\mathcal{H}_d})$ by using a polynomial of degree two, lets call it b, and setting our weights $a_0 = b_0, a_1 = b_1, a_2 = b_2$. Therefore, $R(f^*_{\mathcal{H}_2}) = R(f^*_{\mathcal{H}_d})$ in all cases.

   Approximation error would be equal to 0, as our Bayes predictor is within our hypothesis space:

$$Approximation\ Error = R(f_F) - R(f^*) = R(f^*) - R(f^*) = 0$$

4. (4 Points) For this question we assume $a_0 = 0$. Considering $\mathcal{H} = \{f : x \to b_1 x; b_1 \in \mathbb{R}\}$, which function $f^*_{\mathcal{H}}$ is a risk minimizer in $\mathcal{H}$? What is the approximation error achieved by $f^*_{\mathcal{H}}$? In particular what is the approximation error achieved if furthermore $a_2 = 0$ in the definition of true underlying relation $g(x)$ above?

   Our prediction $\hat{y}$ now takes the form $\hat{y} = a_1 x$. Applying the loss function to our prediction to calculate risk, we will want to calculate:

$$R(\hat{f}) = E(l(\hat{y}, y)) = E(\frac{1}{2}(b_1 x - a_1 x - a_2 x^2))^2$$

   Since our x is uniform from 0 to 1, we can calculate its expectation by integrating over its domain. Let $h(x)$ be the value of our ground truth function a at point x, and $f(x)$ be the probability density at that point. Since our x is uniform between 0 and 1, $f(x) = 1$ for $0 \leq x \leq 1$ and 0 otherwise.

$$\begin{aligned} E(\frac{1}{2}(b_1 x - a_1 x - a_2 x^2))^2 &= \int_0^1 h(x) f(x) dx \\ &= \int_0^1 \frac{1}{2}(b_1 x - a_1 x - a_2 x^2)^2 dx \\ &= \frac{1}{2}\int_0^1 ((b_1 - a_1)x - a_2 x^2)^2 dx \\ &= (\frac{1}{2}(\frac{1}{3}(b_1 - a_1)^2 x^3) + (\frac{1}{2}(b_1 - a_1)^2 a_2^2 x^4) + \frac{1}{5}a_2^2 x^5)\ |_0^1 \\ &= \frac{1}{6}(b_1 - a_1)^2 + \frac{1}{6}(b_1 - a_1)^2 a_2^2 + \frac{1}{10}a_2^2 \end{aligned} \tag{1}$$

   Taking the derivative with respect to b_1 and setting to 0 to minimize our function:

$$\begin{aligned} \frac{d}{dx}R(f^*_{\mathcal{H}}) &= 0 \\ \frac{1}{3}(b_1 - a_1) - \frac{1}{4}(a_2) &= 0 \\ b_1 &= 3 \times (\frac{1}{4}a_2 + \frac{1}{3}a_1) \\ b_1 &= \frac{3}{4}a_2 + a_1 \end{aligned} \tag{2}$$

We've found our optimal weight for b_1, and now our risk minimizer in $\mathcal{H}$ is equal to $\mathcal{H} = (a_1 + \frac{3}{4}a_2)x$. Now to calculate our expected loss via the risk function:

$$\begin{aligned}\frac{1}{6}(b_1 - a_1)^2 + \frac{1}{6}(b_1 - a_1)^2 a_2^2 + \frac{1}{10}a_2^2 &= \frac{1}{6}((a_1 + \frac{3}{4}a_2) - a_1)^2 + \frac{1}{6}((a_1 + \frac{3}{4}a_2) - a_1)^2 a_2^2 + \frac{1}{10}a_2^2 \\ &= \frac{3}{32}a_2^2 - \frac{3}{16}a_2^2 + \frac{1}{10}a_2^2 = \frac{1}{160}a_2^2\end{aligned} \tag{3}$$

Using the definition of approximation error:

$$\textit{Approximation Error} = R(f^*_{\mathcal{H}}) - R(f^*) = R(f^*_{\mathcal{H}}) - 0 = R(f^*_{\mathcal{H}})$$

We have shown that approximation error will be equal to the following: $R(f^*_{\mathcal{H}}) = \frac{1}{160}a_2^2$. If it were the case that $a_2 = 0$ then approximation error would also be 0 as $\frac{1}{160}0^2 = 0$.

Polynomial regression as linear least squares (5 Points)

In practice, $P_{\mathcal{X}\times\mathcal{Y}}$ is usually unknown and we use the empirical risk minimizer (ERM). We will reformulate the problem as a d-dimensional linear regression problem. First note that functions in $\mathcal{H}_d$ are parametrized by a vector $\boldsymbol{b} = [b_0, b_1, \cdots b_d]^\top$, we will use the notation $f_{\boldsymbol{b}}$. Similarly we will note $\boldsymbol{a} \in \mathbb{R}^3$ the vector parametrizing $g(x) = f_{\boldsymbol{a}}(x)$. We will also gather data points from the training sample in the following matrix and vector:

$$X = \begin{bmatrix} 1 & x_1 & \cdots & x_1^d \\ 1 & x_2 & \cdots & x_2^d \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_N & \cdots & x_N \end{bmatrix}, \quad \boldsymbol{y} = [y_0, y_1, \cdots y_N]^\top. \tag{4}$$

These notations allow us to take advantage of the very effective linear algebra formalism. X is called the design matrix.

5. (2 Points) Show that the empirical risk minimizer (ERM) $\hat{\boldsymbol{b}}$ is given by the following minimization $\hat{\boldsymbol{b}} = \arg\min_b \|Xb - \boldsymbol{y}\|_2^2$.

   The empirical risk minimizer being $\hat{\boldsymbol{b}} = \arg\min_b \|Xb - \boldsymbol{y}\|_2^2$ follows directly from the definition of our loss function, as it is merely restated in the language of linear algebra.

   Let us first remember an important Lin Alg. concept: the Euclidean Norm. The Euclidean norm is defined as:

   $$||x||_2 = \sqrt{\langle x, x\rangle} = \sqrt{\sum_{i=1}^{n} x_i^2} \text{ where } x \in \mathbb{R}^n$$

   Squaring the norm simply removes the square root term outside of the summation. Using the definition of our empirical risk minimizer and its loss function:

   $$\begin{aligned} R(\hat{f}) &= E(l(\hat{y}, y)) \\ &= E((\frac{1}{2}(\hat{y} - y)^2) \\ &= \frac{1}{2}E((\hat{y} - y)^2) \\ &= \frac{1}{2 * N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2 \end{aligned} \tag{5}$$

The data set of our prediction, $\hat{y}$, can be expressed as Xb, where $X \in \mathbb{R}^{N\times(d+1)}$ is our design matrix of our $d+1$ polynomial terms, $N \in \mathbb{R}$ is the number of observations, and $b \in \mathbb{R}^{d+1}$ is the vector that parameterizes our design matrix. We can use this representation in our formula above.

$$\begin{aligned}\frac{1}{2*N}\sum_{i=1}^{N}(\hat{y}-y_i)^2 &= \frac{1}{2*N}\sum_{i=1}^{N}(Xb_i-y_i)^2 \\ &= \frac{1}{2*N}\langle Xb-y, Xb-y\rangle \\ &= \frac{1}{2N}||Xb-y||_2^2\end{aligned} \tag{6}$$

Taking the argmin with respect to b yields the optimal coefficient weights for our polynomial terms, and thus the vector that parameterizes our polynomials to minimize loss, $\hat{b}$. We can drop the fraction in front of our equation as it will not affect the answer of $\hat{b}$. This yields the answer to our problem:

$$\hat{\boldsymbol{b}} = \arg\min_b \|Xb-\boldsymbol{y}\|_2^2$$

6. (3 Points) If $N > d$ and X is full rank, show that $\hat{\boldsymbol{b}} = (X^\top X)^{-1}X^\top \boldsymbol{y}$. (Hint: you should take the gradients of the loss above with respect to $\boldsymbol{b}$ [1]). Why do we need to use the conditions $N > d$ and X full rank ?

   Firstly, we must use the condition that $N > d$ as when we generate a polynomial, it has $d+1$ terms. For example, when we consider a polynomial of degree 2, it has 3 terms: $\{a_0, a_1, a_2\}$. Due to Rank-Nullity, the Rank of a matrix (the dimension of its image) is defined as: $Rank(A) \leq min(m,n)$ where $A \in \mathbb{R}^{n\times m}$. Since our design matrix will have $d+1$ columns (n in this case), at minimum, for our our matrix to be full rank it must have at least $d+1$ rows, (m in this case) justifying the strict inequality $N > d$.

   Also, credit to Rank-Nullity, we also know that $Rank(A^T) = Rank(A)$, and that $Rank(AB) \leq min(m,n,r,k)$ where $A \in \mathbb{R}^{n\times m}$ $B \in \mathbb{R}^{r\times k}$. We also know that Therefore, for A^TA to be an invertible matrix, A must be full rank, as it therefore will not have a null space. We need the matrix to be full rank to be able to compute $(X^TX)^{-1}$.

   To arrive at our closed form solution for $\hat{b}$ we consider the following:

$$\begin{aligned}\hat{\boldsymbol{b}} &= \arg\min_b \|Xb-\boldsymbol{y}\|_2^2 \\ &= \arg\min_b \ \langle Xb-y, Xb-y\rangle \\ &= \arg\min_b \ ||Xb||^2 + ||y||^2 - 2\langle Xb, y\rangle \\ &= \arg\min_b \ b^TX^TXb + y^Ty - b^TX^Ty\end{aligned} \tag{7}$$

   Since X is full rank, the expression we arrive to is strongly, (and thus strictly), convex. To

[1] You can check the linear algebra review here if needed `http://cs229.stanford.edu/section/cs229-linalg.pdf`

compute the minimum, we can differentiation both sides with respect to $\hat{b}$ and set to 0.

$$\begin{aligned} 2X^TXb - 2Xy^T &= 0 \\ X^TXb &= X^Ty \\ b &= (X^TX)^{-1}X^Ty \\ \hat{\boldsymbol{b}} &= (X^\top X)^{-1}X^\top \boldsymbol{y} \ \square \end{aligned} \tag{8}$$

We have shown that the b that minimizes $||Xb - y||^2$ is $\hat{b} = (X^TX)^{-1}X^Ty$, assuming X when we assume X is full rank.

Hands on (7 Points)

Open the source code file *hw1_code_source.py* from the *.zip* folder. Using the function `get_a` get a value for $\boldsymbol{a}$, and draw a sample `x_train, y_train` of size $N = 10$ and a sample `x_test, y_test` of size $N_{\text{test}} = 1000$ using the function `draw_sample`.

7. (2 Points) Write a function called `least_square_estimator` taking as input a design matrix $X \in \mathbb{R}^{N\times(d+1)}$ and the corresponding vector $\boldsymbol{y} \in \mathbb{R}^N$ returning $\hat{\boldsymbol{b}} \in \mathbb{R}^{(d+1)}$. Your function should handle any value of N and d, and in particular return an error if $N \leq d$. (Drawing x at random from the uniform distribution makes it almost certain that any design matrix X with $d \geq 1$ we generate is full rank).

```
    #Define our least squares estimator function
def least_squares_estimator(X, y):
    """
    Inputs:
    X: (np.matrix of size N x (deg_true +1))
    y: (np.array) of size deg_true + 1 x 1

    Returns:
    b_hat: (np.array) of size N x (deg_true + 1)
    """
    #Make sure N > d
    if X.shape[0] < X.shape[1]:
        raise ValueError("You must have at least as many rows as
        ↪  columns!")
    else:
        #Compute the solution for b using the closed form linear algebra
        ↪  solution
        b_hat = np.linalg.inv(X.T@X) @ X.T @ y
        return b_hat
```

8. (1 Points) Recall the definition of the empical risk $\hat{R}(\hat{f})$ on a sample $\{x_i, y_i\}_{i=1}^N$ for a prediction function $\hat{f}$. Write a function `empirical_risk` to compute the empirical risk of $f_{\boldsymbol{b}}$ taking as input a design matrix $X \in \mathbb{R}^{N\times(d+1)}$, a vector $\boldsymbol{y} \in \mathbb{R}^N$ and the vector $\boldsymbol{b} \in \mathbb{R}^{(d+1)}$ parametrizing the predictor.

```
def empirical_risk(X,y,b_hat):
    """
    Inputs:
    X: (np.matrix of size N x (deg_true +1))
    y: (np.array) of size deg_true + 1 x 1
    b_hat: (np.array) of size N x (deg_true + 1)
    Returns:
    emp_risk: (float)
    """
    #Get # of observations
    N = X.shape[0]
    #Calculate Predictions
    y_hat = X @ b_hat
    #Calculate squared errors and then empirical risk
    sum_of_squared_errors = sum((y_hat-y)**2)
    emp_risk = sum_of_squared_errors / N
    emp_risk = emp_risk / 2 #because we have 1/2 in our loss function
    return emp_risk
```

9. (3 Points) Use your code to estimate $\hat{\boldsymbol{b}}$ from `x_train, y_train` using $d = 5$. Compare $\hat{\boldsymbol{b}}$ and $\boldsymbol{a}$. Make a single plot (Plot 1) of the plan (x, y) displaying the points in the training set, values of the true underlying function $g(x)$ in $[0, 1]$ and values of the estimated function $f_{\hat{\boldsymbol{b}}}(x)$ in $[0, 1]$. Make sure to include a legend to your plot .

10. (1 Points) Now you can adjust d. What is the minimum value for which we get a "perfect fit"? How does this result relates with your conclusions on the approximation error above?

```
#Iterate through values 1-> 10, which will be
#the degree of our polynomial used to predict y
for i in range(1,10):

    #Get new design matrices for polynomial i
    X_train = get_design_mat(x_train, i)
    X_test = get_design_mat(x_test, i)

    #Calculate best coefficient for each term
    new_b_hat = least_squares_estimator(X_train, y_train)

    #Calculate empirical risk for polynomial degree i
    #Use the test set that we generated
    current_risk = empirical_risk(X_test,y_test,new_b_hat)
    print("Empirical Risk for Polynomial Degree", i, "is:",current_risk)
    Empirical Risk for Polynomial Degree 1 is: 0.010310614710875313
"""
Empirical Risk for Polynomial Degree 2 is: 7.118907606224661e-29
Empirical Risk for Polynomial Degree 3 is: 1.9513479548738786e-27
Empirical Risk for Polynomial Degree 4 is: 7.540817539818515e-23
Empirical Risk for Polynomial Degree 5 is: 1.0413645751929928e-18
```

```
Empirical Risk for Polynomial Degree 6 is: 3.854115911861267e-15
Empirical Risk for Polynomial Degree 7 is: 1.8067338309741379e-09
Empirical Risk for Polynomial Degree 8 is: 2.2386507999205597e-05
Empirical Risk for Polynomial Degree 9 is: 0.9117631203828602"""
```

The lowest d that estimates g(x) perfectly is d=2, which makes perfect sense as our ground truth function is a polynomial of degree 2. This means our least squares estimate is calculating the perfect a_0, a_1, and a_2 weights for the polynomial.

In presence of noise (13 Points)

Now we will modify the true underlying $P_{\mathcal{X}\times\mathcal{Y}}$, adding some noise in $y = g(x)+\epsilon$, with $\epsilon \sim \mathcal{N}(0,1)$ a standard normal random variable independent from x. We will call training error e_t the empirical risk on the train set and generalization error e_g the empirical risk on the test set.

11. (6 Points) Plot e_t and e_g as a function of N for $d < N < 1000$ for $d = 2$, $d = 5$ and $d = 10$ (Plot 2). You may want to use a logarithmic scale in the plot. Include also plots similar to Plot 1 for 2 or 3 different values of N for each value of d.

12. (4 Points) Recall the definition of the estimation error. Using the test set, (which we intentionally chose large so as to take advantage of the law of large numbers) give an empirical estimator of the estimation error. For the same values of N and d above plot the estimation error as a function of N (Plot 3).

    Approximation Error $= R(f_F) - R(f^*)$ **Estimation Error** $= R(\hat{f}) - R(f_F)$ Excess Risk, and thus estimation error, approaches 0 as N becomes large. Specifically, estimation error decreases to 0 due to the law of large numbers, optimization error is already 0 as we are using the closed form solution of Least Squares Estimation rather than gradient descent, and approximation error also converges to 0 as our polynomials fit better and better to the ground truth polynomial, while simultaneously not being able to fit the random noise with variance 1.

    Therefore, the only error that remains is Bayes risk. Since our Bayes predictor, f^*, is a polynomial function of degree 2 with some noise added, is outside our hypothesis space. Bayes risk is equal to $\frac{1}{2}$, which makes perfect sense as our noise term is a random variable that is 0 mean and has variance 1 and our loss function is defined as the sum of squared errors divided by 2.

13. (2 Points) The generalization error gives in practice an information related to the estimation error. Comment on the results of (Plot 2 and 3). What is the effect of increasing N? What is the effect of increasing d?

    What is the effect of increasing N?

    As the higher dimension polynomials ($d > 2$) are fit with larger N observations of training data, the weights on the larger coefficients (a_i where $i \in \{3, 4, 5, \ldots, d\}$ approach 0, while the weights of the first three coefficients (a_0, a_1, a_2) approach the ground truth function weights parameterized by a. The loss approaches 1, which is the same loss we would expect from our Bayes predictor, as our ground truth function contains noise with unit variance, and our loss is measure my MSE. When we calculate estimation error, it quickly approaches 0 for all of the polynomials as we get closer to 100 observations.

    What are the effects of increasing d?

    As we increase d, generalization error increases substantially. This is because the higher degree polynomials are liable to over-fit our random sample data with over-complicated

polynomial curves, as well as modeling the noise and not capturing the true shape of the graph. As mentioned above, the generalization error converges to 0 as N becomes sufficiently large.

14. (1 Points) Besides from the approximation and estimation there is a last source of error we have not discussed here. Can you comment on the optimization error of the algorithm we are implementing?

    Optimization Error

    There has been no optimization error in the algorithm we are implementing, as we are using the closed form linear algebra solution for least squares on a fairly small data set. If we had a large matrix, say 300m rows by 300m columns, then we would not be able to employ our current method as it would be too computationally expensive. We'd have to resort to methods like gradient descent, and then we would most likely have approximation error. That being said, in the methods employed on this homework, we have no optimization erorr.

Application to Ozone data (optional) (2 Points)

You can now use the code we developed on the synthetic example on a real world dataset. Using the command `np.loadtxt('ozone_wind.data')` load the data in the *.zip*. The first column corresponds to ozone measurements and the second to wind measurements. You can try polynomial fits of the ozone values as a function of the wind values.

15. (2 Points) Reporting plots, discuss the again in this context the results when varying N (subsampling the training data) and d.

    This time around, the generalization error of each polynomial does not converge to a certain value, which is very interesting. A polynomial of degree 2 performed the best for both empirical and generalization risk, across varying amounts of training observation data used to fit our model $N \in \{20, 40, 60\}$. As we don't know the Bayes function, it is impossible to comment on the extent that approximation risk is present in our current total Risk, as the ozone and wind may or not may be explained by a polynomial relationship. Estimation risk is present in our model as the generalization error does not converge. If we had more data points, we could eliminate estimation risk as N would increase the generalization error of our various polynomial functions would converge to the same value. Optimization risk should not be present in our excess risk decomposition, as we are still using the closed form linear algebra solution for ordinary least squares. If our N observations grew untractably large, perhaps we'd have to resort to gradient descent methods, in which case, optimization error would be present.

In [1]:
```
#Import Libraries
import numpy as np
import matplotlib.pyplot as plt
destination = "/Plots"
%matplotlib inline
```

In [2]:
```
#Given functions
def get_a(deg_true):
    """
    Inputs:
    deg_true: (int) degree of the polynomial g

    Returns:
    a: (np array of size (deg_true + 1)) coefficients of polynomial g
    """
    return 5 * np.random.randn(deg_true + 1)

def get_design_mat(x, deg):
    """
    Inputs:
    x: (np.array of size N)
    deg: (int) max degree used to generate the design matrix

    Returns:
    X: (np.array of size N x (deg_true + 1)) design matrix
    """
    X = np.array([x ** i for i in range(deg + 1)]).T
    return X

def draw_sample(deg_true, a, N):
    """
    Inputs:
    deg_true: (int) degree of the polynomial g
    a: (np.array of size deg_true) parameter of g
    N: (int) size of sample to draw

    Returns:
    x: (np.array of size N)
    y: (np.array of size N)
    """
    x = np.sort(np.random.rand(N))
    X = get_design_mat(x, deg_true)
    y = X @ a
    return x, y

def draw_sample_with_noise(deg_true, a, N):
    """
    Inputs:
```

```
    deg_true: (int) degree of the polynomial g
    a: (np.array of size deg_true) parameter of g
    N: (int) size of sample to draw

    Returns:
    x: (np.array of size N)
    y: (np.array of size N)
    """
    x = np.random.rand(N)
    X = get_design_mat(x, deg_true)
    y = X @ a + np.random.randn(N)
    return x, y
```

Problem 7

In [3]:
```
#Define the polynomial size
p = 2

#Get a value for a
a = get_a(p)

#Generate training and test data
x_train, y_train = draw_sample(p,a,10)
x_test, y_test = draw_sample(p,a,1000)
```

In [4]:
```
#Define our least squares estimator function
def least_squares_estimator(X, y):
    """
    Inputs:
    X: (np.matrix of size N x (deg_true +1))
    y: (np.array) of size deg_true + 1 x 1

    Returns:
    b_hat: (np.array) of size N x (deg_true + 1)
    """
    #Make sure N > d
    if X.shape[0] < X.shape[1]:
        raise ValueError("You must have at least as many rows as colum
    else:
        #Compute the solution for b using the closed form linear algeb
        b_hat = np.linalg.inv(X.T@X) @ X.T @ y
        return b_hat
```

Problem 8

In [5]:
```
def empirical_risk(X,y,b_hat):
    """
    Inputs:
    X: (np.matrix of size N x (deg_true +1))
    y: (np.array) of size deg_true + 1 x 1
    b_hat: (np.array) of size N x (deg_true + 1)
    Returns:
    emp_risk: (float)
    """
    #Get # of observations
    N = X.shape[0]
    #Calculate Predictions
    y_hat = X @ b_hat
    #Calculate squared errors and then empirical risk
    sum_of_squared_errors = sum((y_hat-y)**2)
    emp_risk = sum_of_squared_errors / N
    emp_risk = emp_risk / 2 #because we have 1/2 in our loss function
    return emp_risk
```

Problem 9

In [6]:
```
#Generate design matrices
X_train = get_design_mat(x_train, p)
X_test = get_design_mat(x_test, p)
```

In [7]:
```
#Calculate b_hat
b_hat = least_squares_estimator(X_train,y_train)
```

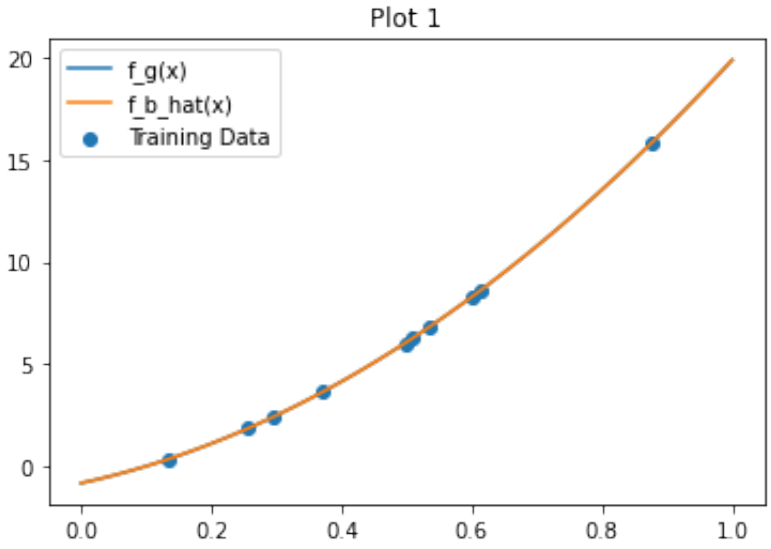
In [8]:
```
#Compare a and b_hat values
for i in range(len(b_hat)):
    print('Values at Index', i, 'For Vectors b_Hat and a',a[i],b_hat[i
    print("Difference rounded to 5 decimal places:", np.round(a[i]-b_h
```

```
Values at Index 0 For Vectors b_Hat and a -0.825424518251156 -0.82542
45182511442
Difference rounded to 5 decimal places: -0.0
Values at Index 1 For Vectors b_Hat and a 6.90016660753996 6.90016660
7539923
Difference rounded to 5 decimal places: 0.0
Values at Index 2 For Vectors b_Hat and a 13.797972489732542 13.79797
2489732707
Difference rounded to 5 decimal places: -0.0
```

In [9]:
```
#Use a helper function to make the graphing easier
def helper_func(x, degree):
    return np.array([x**i for i in range(degree+1)])
```

In [10]:
```
#Plot b_hat predictions vs A
x = np.linspace(0,1,100)
func_g = a@ helper_func(x,p)
b_hat_x = b_hat @ helper_func(x,p)
plt.scatter(x_train,y_train)

#B_hat predictions given
plt.plot(x, b_hat_x)
plt.plot(x, func_g)
plt.legend(labels=['f_g(x)','f_b_hat(x)','Training Data'])
plt.title("Plot 1")
plt.savefig('Plots//Plot_1')
print("The function is well estimated, and therefore not visible on th
```

```
The function is well estimated, and therefore not visible on the grap
h as it is covered up by the original function g(x)
```



Problem 10

In [11]:
```
#Iterate through values 1-> 10, which will be
#the degree of our polynomial used to predict y
for i in range(1,10):

    #Get new design matrices for polynomial i
    X_train = get_design_mat(x_train, i)
    X_test = get_design_mat(x_test, i)

    #Calculate best coefficient for each term
    new_b_hat = least_squares_estimator(X_train, y_train)

    #Calculate empirical risk for polynomial degree i
    #Use the test set that we generated
    current_risk = empirical_risk(X_test,y_test,new_b_hat)
    print("Empirical Risk for Polynomial Degree", i, "is:",current_ris
```

```
Empirical Risk for Polynomial Degree 1 is: 0.6809097664580388
Empirical Risk for Polynomial Degree 2 is: 1.776156084824159e-27
Empirical Risk for Polynomial Degree 3 is: 5.15135024080385e-24
Empirical Risk for Polynomial Degree 4 is: 1.252680372643675e-20
Empirical Risk for Polynomial Degree 5 is: 3.9316957719710045e-16
Empirical Risk for Polynomial Degree 6 is: 3.4111689154669325e-11
Empirical Risk for Polynomial Degree 7 is: 1.2427202507726653e-07
Empirical Risk for Polynomial Degree 8 is: 0.16439015158302786
Empirical Risk for Polynomial Degree 9 is: 17.998925946761307
```

The lowest d that estimates g(x) perfectly is d=2, which makes perfect sense as our ground truth function is a polynomiail of degree 2. This means our least squares estimate is calculating the perfect a_0, a_1, and a_2 weights for the polynomial.

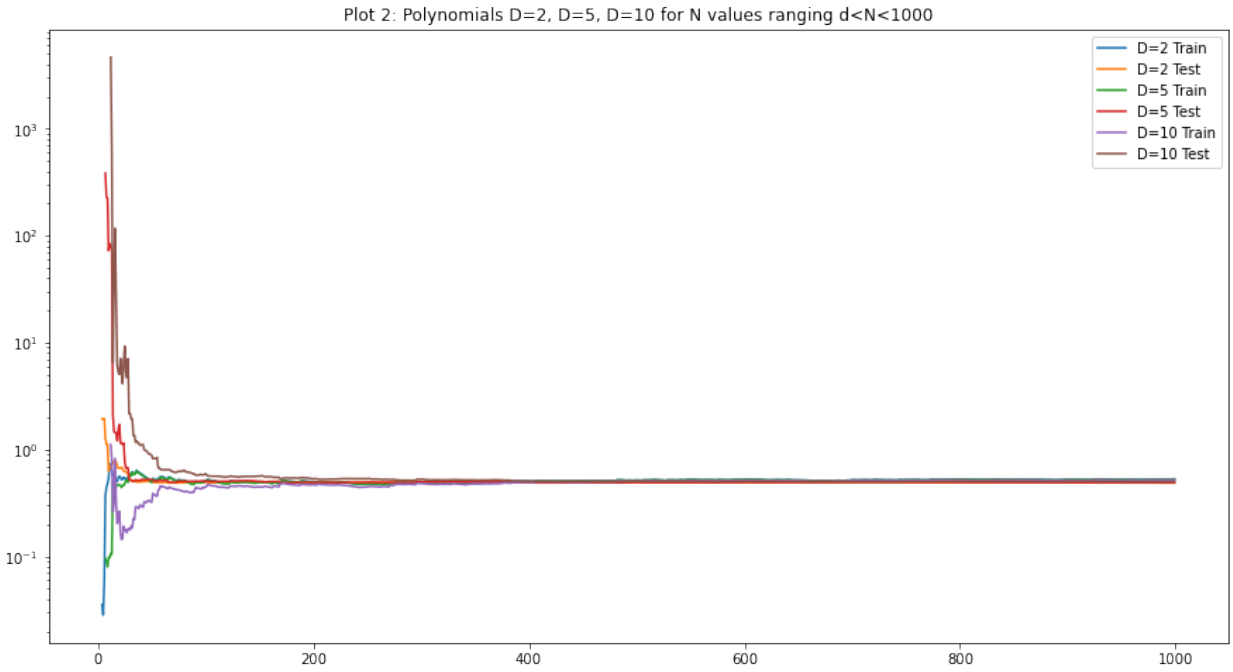
Problem 11

In [12]:
```
def noisy_emp_and_gen_risk(d,x_train,y_train, x_test,y_test):
    """
    Inputs:
    d: (int) degree of polynomial desired
    n: (int) number of samples to be generated in

    Outputs:
    training_error: (float) average sum of squares of loss function or
    generalization_error: (float) average Sum of Squares of loss funct
    """
    #Generate design matrices
    X_train = get_design_mat(x_train, d)
    X_test = get_design_mat(x_test, d)
    #Calculate b_hat
    b_hat = least_squares_estimator(X_train, y_train)
    training_error = empirical_risk(X_train, y_train, b_hat)
    generalization_error = empirical_risk(X_test, y_test, b_hat)

    return training_error, generalization_error, b_hat
```

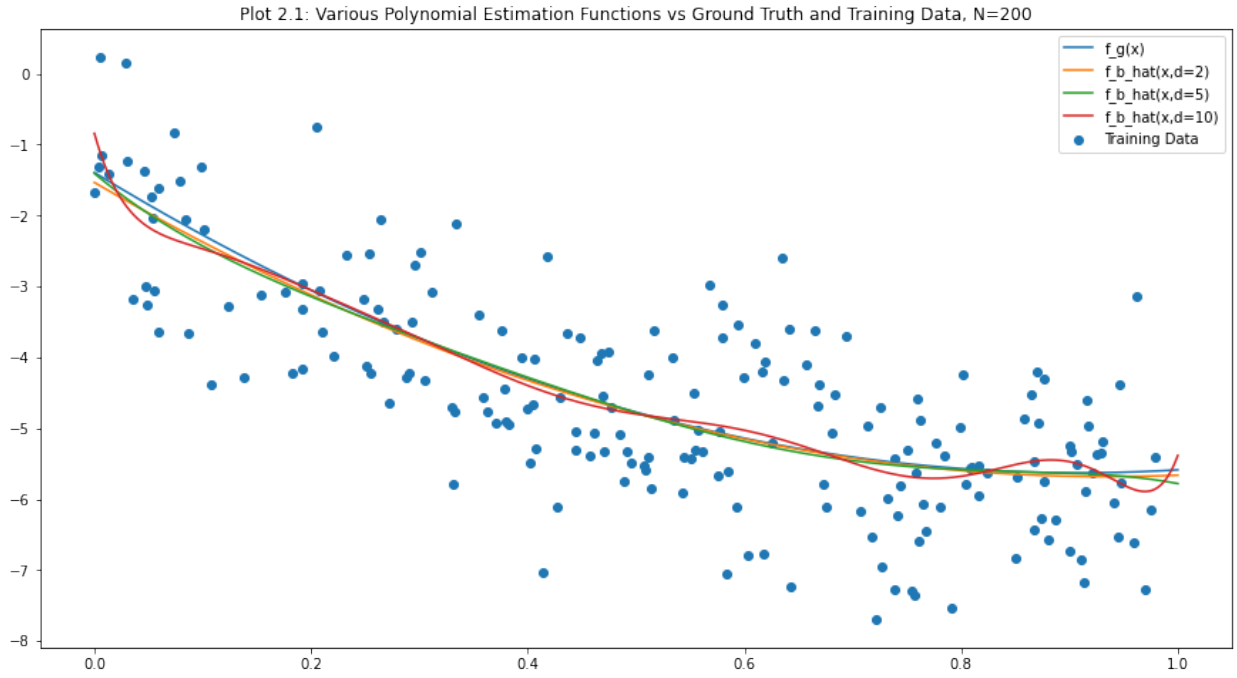
In [13]:
```
def poly_risk_gen(d,n, x_train, y_train, x_test,y_test):
    """
    Inputs:
    d: (int) desired degree of polynomial
    n: (int) number of rows TBD
    x_train: (np.array) training data x values
    y_train: (np.array)
    x_test:
    y_test:

    Ouptut:
    train_error_arr: (np.array) e_t for various n
    test_error_arr: (np.array) e_g for various n
    """
    #Initiliaze np arrays
    train_error_arr = []
    test_error_arr = []

    #Iterate over N
    for i in range(d+1,len(n)):

        #Get relevant subset of data
        train_x_subset = x_train[:i+1]
        train_y_subset = y_train[:i+1]

        #Calculate e_t, e_g, and append to output
        training_error, generalization_error, b_hat = noisy_emp_and_ge
        train_error_arr.append(training_error)
        test_error_arr.append(generalization_error)

    return train_error_arr, test_error_arr, b_hat
```

In [14]:
```
#Generate noisy data for polynomial of degree 5, with N = 1000 for tra
d = 2
a = get_a(d)
x_train, y_train = draw_sample_with_noise(d,a,1000)
x_test, y_test = draw_sample_with_noise(d,a,1000)

#Generate a helper variable
n = list(range(1,1000))

#Calculate e_g, e_t for various N's of polynomial degree 2
train_error_arr_2, test_error_arr_2, b_hat_2 = poly_risk_gen(2,n,x_tra

#Calculate e_g, e_t for various N's of polynomial degree 5
train_error_arr_5, test_error_arr_5, b_hat5 = poly_risk_gen(5,n,x_trai

#Calculate e_g, e_t for various N's of polynomial degree 10
train_error_arr_10, test_error_arr_10, b_hat10 = poly_risk_gen(10,n,x_
```

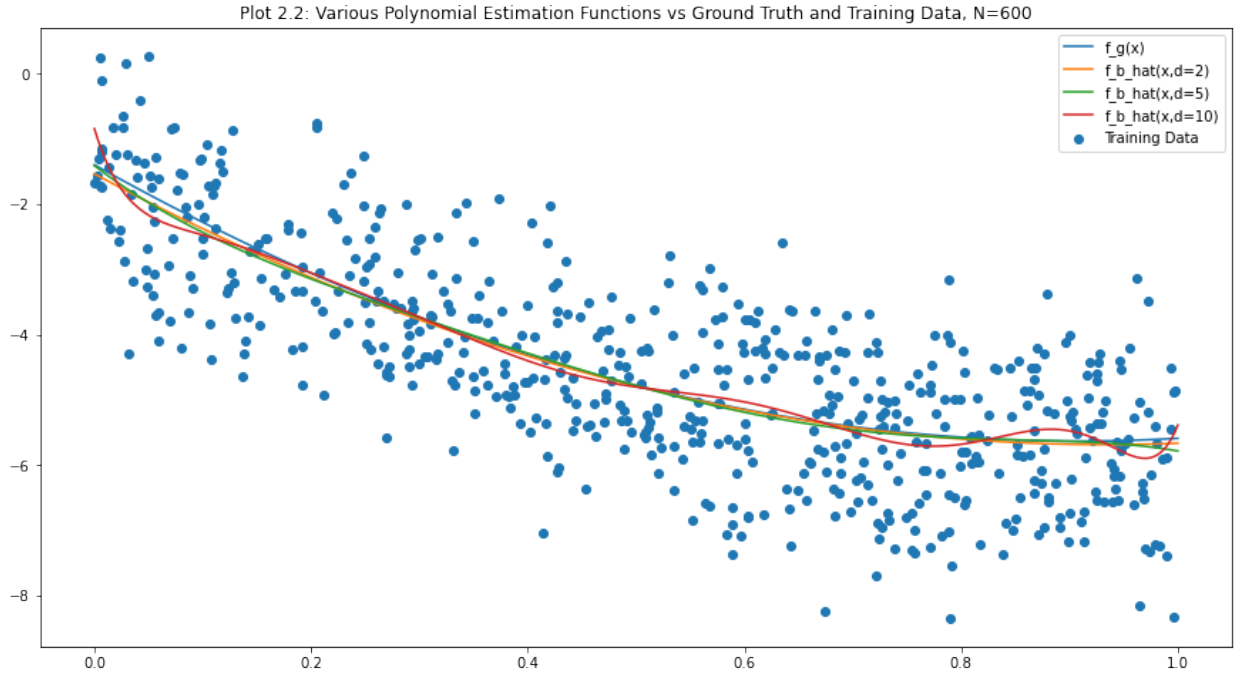
```
#Plot
plt.figure(figsize=(15,8))
plt.plot(n[3:], train_error_arr_2)
plt.plot(n[3:], test_error_arr_2)
plt.plot(n[6:], train_error_arr_5)
plt.plot(n[6:], test_error_arr_5)
plt.plot(n[11:], train_error_arr_10)
plt.plot(n[11:], test_error_arr_10)
plt.legend(labels=['D=2 Train','D=2 Test','D=5 Train', 'D=5 Test', 'D=
plt.title("Plot 2: Polynomials D=2, D=5, D=10 for N values ranging d<N
plt.yscale('log')
plt.savefig('Plots//Plot_2')
```



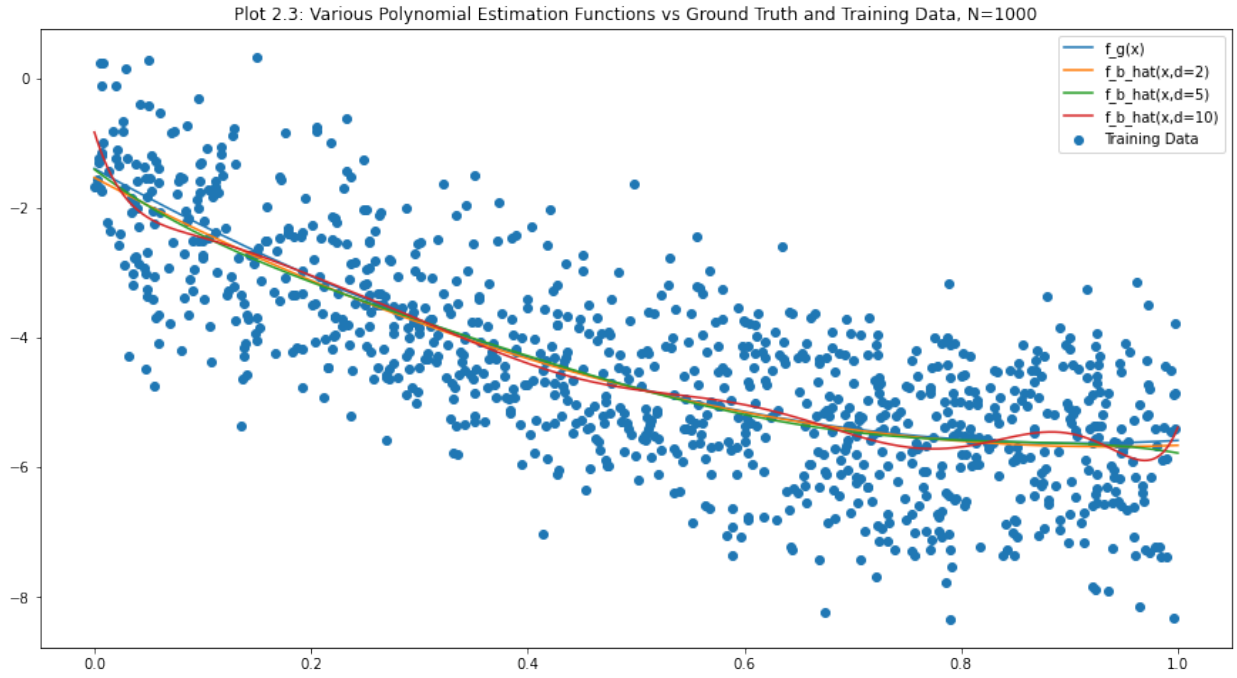
In [15]:
```
#Calculate e_g, e_t for various N's of polynomial degree 2
train_error_arr_2, test_error_arr_2, b_hat_2_200 = poly_risk_gen(2,n,x

#Calculate e_g, e_t for various N's of polynomial degree 5
train_error_arr_5, test_error_arr_5, b_hat_5_200 = poly_risk_gen(5,n,x

#Calculate e_g, e_t for various N's of polynomial degree 10
train_error_arr_10, test_error_arr_10, b_hat_10_200 = poly_risk_gen(10

#Plot b_hat predictions vs A
x = np.linspace(0,1,1000)
func_g = a @ helper_func(x,p)
func_b_hat_2 = b_hat_2_200 @ helper_func(x,2)
func_b_hat_5 = b_hat_5_200 @ helper_func(x,5)
func_b_hat_10 = b_hat_10_200 @ helper_func(x,10)

#B_hat predictions given
plt.figure(figsize=(15,8))
plt.scatter(x_train[:201],y_train[:201])
plt.plot(x, func_g)
plt.plot(x, func_b_hat_2)
plt.plot(x, func_b_hat_5)
plt.plot(x, func_b_hat_10)

plt.legend(labels=['f_g(x)','f_b_hat(x,d=2)','f_b_hat(x,d=5)','f_b_hat
plt.title("Plot 2.1: Various Polynomial Estimation Functions vs Ground
plt.savefig('Plots//Plot_2_1')
```



In [16]:
```
#Calculate e_g, e_t for various N's of polynomial degree 2
train_error_arr_2, test_error_arr_2, b_hat_2_600 = poly_risk_gen(2,n,x

#Calculate e_g, e_t for various N's of polynomial degree 5
train_error_arr_5, test_error_arr_5, b_hat_5_600 = poly_risk_gen(5,n,x

#Calculate e_g, e_t for various N's of polynomial degree 10
train_error_arr_10, test_error_arr_10, b_hat_10_600 = poly_risk_gen(10
func_g = a @ helper_func(x,p)
func_b_hat_2 = b_hat_2_600 @ helper_func(x,2)
func_b_hat_5 = b_hat_5_600 @ helper_func(x,5)
func_b_hat_10 = b_hat_10_600 @ helper_func(x,10)

#B_hat predictions given
plt.figure(figsize=(15,8))
plt.scatter(x_train[:601],y_train[:601])
plt.plot(x, func_g)
plt.plot(x, func_b_hat_2)
plt.plot(x, func_b_hat_5)
plt.plot(x, func_b_hat_10)

plt.legend(labels=['f_g(x)','f_b_hat(x,d=2)','f_b_hat(x,d=5)','f_b_hat
plt.title("Plot 2.2: Various Polynomial Estimation Functions vs Ground
plt.savefig('Plots//Plot_2_2')
```



In [17]:
```
#Generate a helper variable
n = list(range(1,1001))

#Calculate e_g, e_t for various N's of polynomial degree 2
train_error_arr_2, test_error_arr_2, b_hat_2_1000 = poly_risk_gen(2,n,

#Calculate e_g, e_t for various N's of polynomial degree 5
train_error_arr_5, test_error_arr_5, b_hat_5_1000 = poly_risk_gen(5,n,

#Calculate e_g, e_t for various N's of polynomial degree 10
train_error_arr_10, test_error_arr_10, b_hat_10_1000 = poly_risk_gen(1
func_g = a @ helper_func(x,p)
func_b_hat_2 = b_hat_2_1000 @ helper_func(x,2)
func_b_hat_5 = b_hat_5_1000 @ helper_func(x,5)
func_b_hat_10 = b_hat_10_1000 @ helper_func(x,10)

#B_hat predictions given N data points
plt.figure(figsize=(15,8))
plt.scatter(x_train[:1001],y_train[:1001])
plt.plot(x, func_g)
plt.plot(x, func_b_hat_2)
plt.plot(x, func_b_hat_5)
plt.plot(x, func_b_hat_10)

plt.legend(labels=['f_g(x)','f_b_hat(x,d=2)','f_b_hat(x,d=5)','f_b_hat
plt.title("Plot 2.3: Various Polynomial Estimation Functions vs Ground
plt.savefig('Plots//Plot_2_3')
```



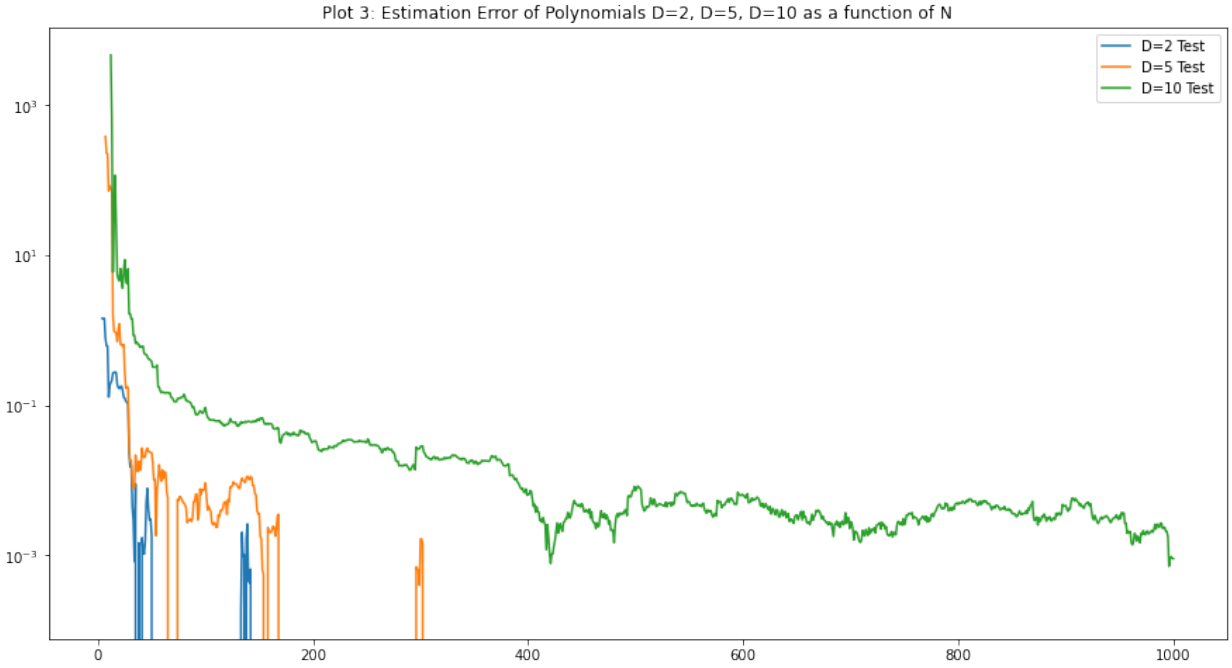
Question 12

Approximation Error = $R(f_F) - R(f^*)$ **Estimation Error** = $R(\hat{f}) - R(f_F)$

Excess Risk, and thus estimation error, approaches 0 as N becomes large. Specifically, estimation error decreases to 0 due to the law of large numbers, optimization error is already 0 as we are using the closed form solution of Least Squares Estimation rather than gradient descent, and approximation error also converges to 0 as our polynomials fit better and better to the ground truth polynomial, while simoltanuesly not being able to fit the random noise with variance 1.

Therefore, the only error that remains is Bayes risk. Since our Bayes predictor, f^*, is a polynomial function of degree 2 with some noise added, is outside our hypothesis space. Bayes risk is equal to $\frac{1}{2}$, which makes perfect sense as our noise term is a random variable that is 0 mean and has variance 1 and our loss function is defined as the sum of squared errors divided by 2.

In [18]:
```
test_error_arr_2 = np.array(test_error_arr_2)
test_error_arr_5 = np.array(test_error_arr_5)
test_error_arr_10 = np.array(test_error_arr_10)

#Plot
plt.figure(figsize=(15,8))
plt.plot(n[3:], test_error_arr_2-.5)
plt.plot(n[6:], test_error_arr_5-.5)
plt.plot(n[11:], test_error_arr_10-.5)
plt.legend(labels=['D=2 Test','D=5 Test', 'D=10 Test'])
plt.title("Plot 3: Estimation Error of Polynomials D=2, D=5, D=10 as a
plt.yscale('log')
plt.savefig('Plots//Plot_3')
```



Question 13

What is the effect of increasing N?

As the higher dimension polynomials ($d > 2$) are fit with larger N observations of training data, the weights on the larger coefficients (a_i where $i \in \{3, 4, 5, \ldots, d\}$ approach 0, while the weights of the first three coefficients (a_0, a_1, a_2) approach the ground truth function weights parameterized by a. The loss approaches 1, which is the same loss we would expect from our Bayes predictor, as our ground truth function contains noise with unit variance, and our loss is measure my MSE. When we calculate estimation error, it quickly approaches 0 for all of the polynomials as we get closer to 100 observations.

What are the effects of increasing d?

As we increase d, generalization error increases substantially. This is because the higher degree polynomials are liable to overfit our random sample data with overcomplicated polynomial curves, as well as modeling the noise and not capturing the true shape of the graph. As mentioned above, the generalization error converges to 0 as N becomes sufficiently large.

Question 14

Optimization Error

There has been no optimization error in the algorithim we are implementing, as we are using the closed form linear algebra solution for least squares on a fairly small data set. If we had a large matrix, say 300m rows by 300m columns, then we would not be able to employ our current method as it would be too computationally expensive. We'd have to resort to methods like gradient descent, and then we would most likely have approximation error. That being said, in the methods employed on this homework, we have no optimization erorr.

Question 15

In [19]:
```
data = np.loadtxt('ozone_wind.data')
print(max(data[:,0]))
```

```
168.74
```

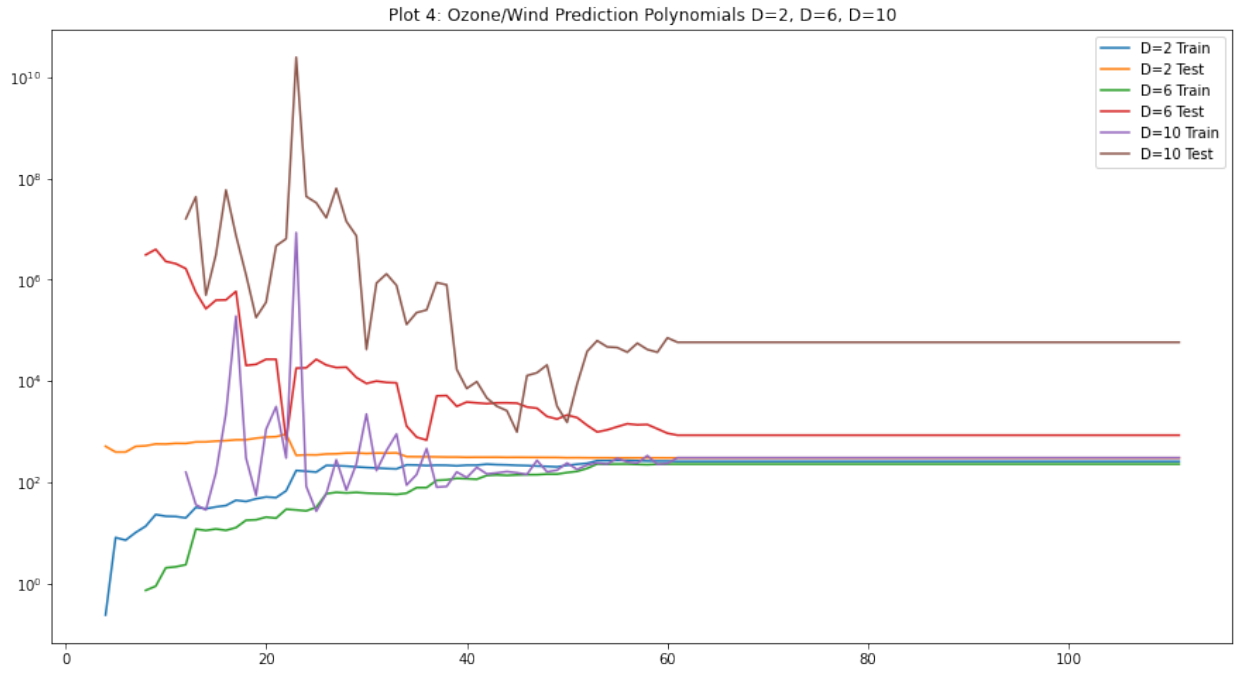
In [20]:
```
#Make 70/30 Train Test Splits
x_train, y_train = data[:,1][:61], data[:,0][:61]
x_test, y_test = data[:,1][61:], data[:,0][61:]
```

In [21]:
```
#Iterate through values 1-> 10, which will be
#the degree of our polynomial used to predict y
for i in range(1,15):

    #Get new design matrices for polynomial i
    X_train = get_design_mat(x_train, i)
    X_test = get_design_mat(x_test, i)

    #Calculate best coefficient for each term
    new_b_hat = least_squares_estimator(X_train, y_train)

    #Calculate empirical risk for polynomial degree i
    #Use the test set that we generated
    current_risk = empirical_risk(X_test,y_test,new_b_hat)
    print("Empirical Risk for Polynomial Degree", i, "is:",current_ris
```

```
Empirical Risk for Polynomial Degree 1 is: 384.7048169669439
Empirical Risk for Polynomial Degree 2 is: 300.2385522885913
Empirical Risk for Polynomial Degree 3 is: 342.7922052960828
Empirical Risk for Polynomial Degree 4 is: 418.22226037488457
Empirical Risk for Polynomial Degree 5 is: 434.10441766132203
Empirical Risk for Polynomial Degree 6 is: 844.1754597489719
Empirical Risk for Polynomial Degree 7 is: 1451.9818719739806
Empirical Risk for Polynomial Degree 8 is: 1351.525170933294
Empirical Risk for Polynomial Degree 9 is: 29187.968041553024
Empirical Risk for Polynomial Degree 10 is: 57896.52413379224
Empirical Risk for Polynomial Degree 11 is: 1230282.2230483317
Empirical Risk for Polynomial Degree 12 is: 263877.9141563437
Empirical Risk for Polynomial Degree 13 is: 24855.36055544052
Empirical Risk for Polynomial Degree 14 is: 325618.839584572
```

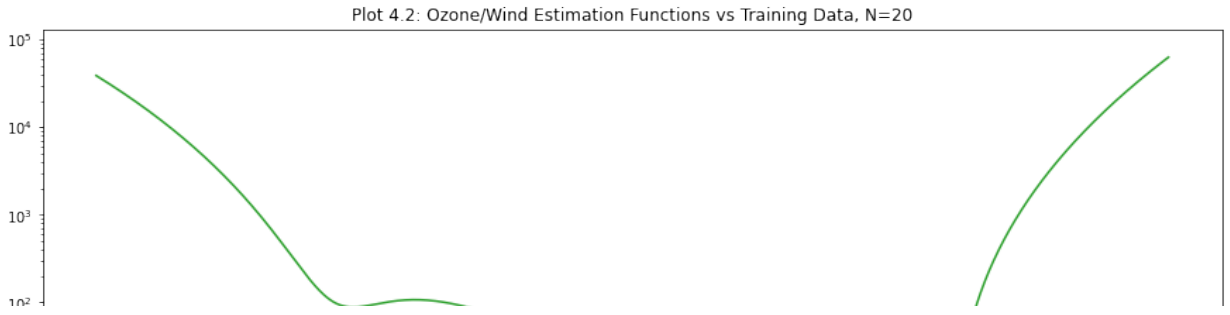
It appears that a polynomial of degree 2 fits the data the best

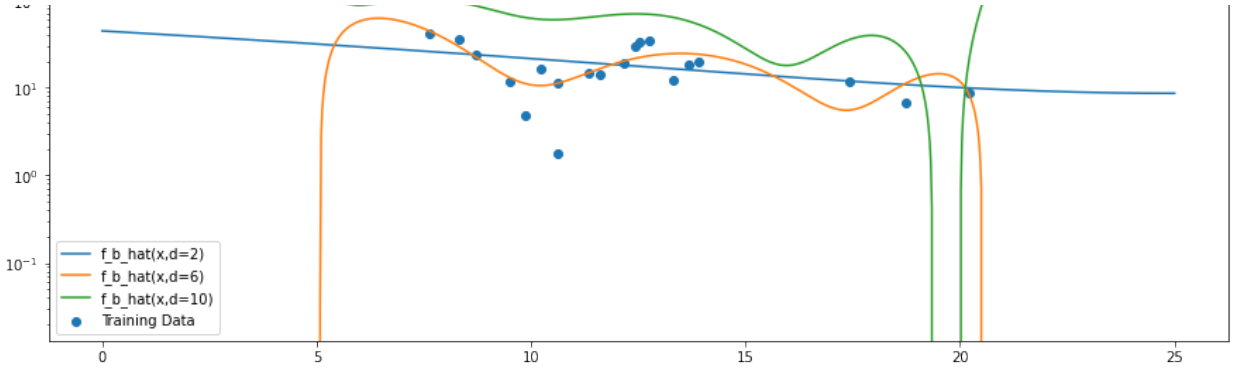
In [22]:
```
#Generate a helper variable
n = list(range(1,112))

#Calculate e_g, e_t for various N's of polynomial degree 2
train_error_arr_2, test_error_arr_2, b_hat_2 = poly_risk_gen(2,n,x_tra

#Calculate e_g, e_t for various N's of polynomial degree 6
train_error_arr_6, test_error_arr_6, b_hat6 = poly_risk_gen(6,n,x_trai

#Calculate e_g, e_t for various N's of polynomial degree 10
train_error_arr_10, test_error_arr_10, b_hat10 = poly_risk_gen(10,n,x_


#Plot
plt.figure(figsize=(15,8))
plt.plot(n[3:], train_error_arr_2)
plt.plot(n[3:], test_error_arr_2)
plt.plot(n[7:], train_error_arr_6)
plt.plot(n[7:], test_error_arr_6)
plt.plot(n[11:], train_error_arr_10)
plt.plot(n[11:], test_error_arr_10)
plt.legend(labels=['D=2 Train','D=2 Test','D=6 Train', 'D=6 Test', 'D=
plt.title("Plot 4: Ozone/Wind Prediction Polynomials D=2, D=6, D=10")
plt.yscale('log')
plt.savefig('Plots//Plot_4_1')
```



In [23]:
```
#Demonstrate how the polynomials generate different generalization err
n = list(range(20))
polynomials = [2,6,10]
```

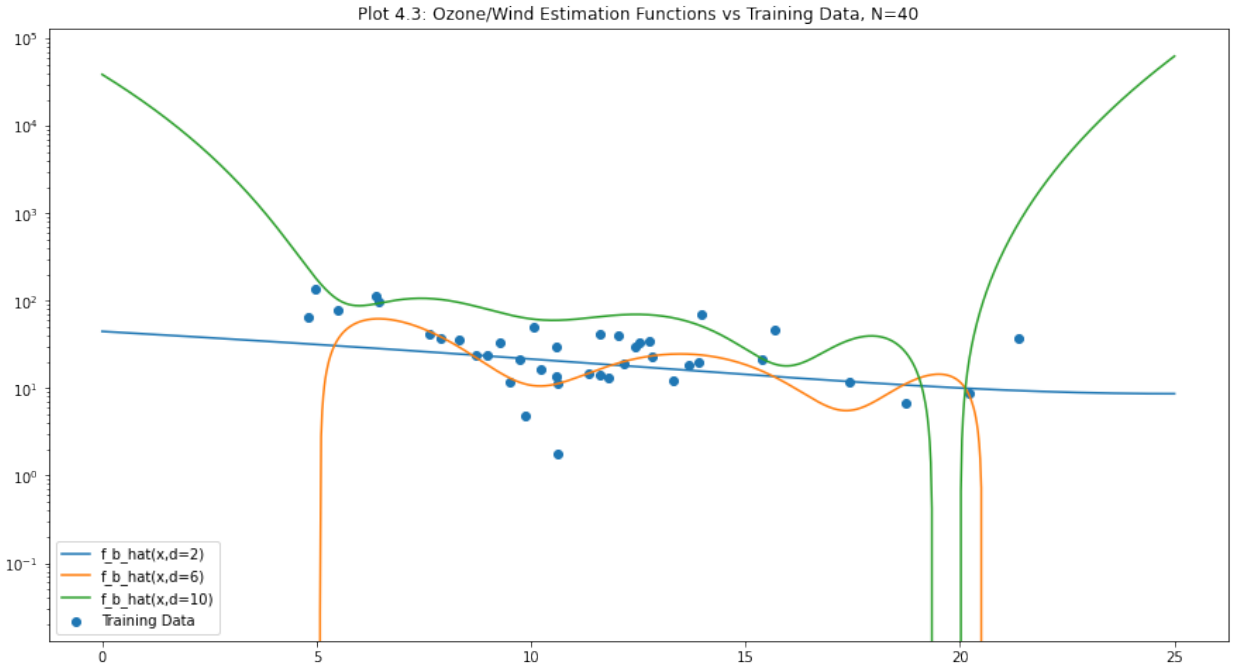
```
#Generate helper storage variables
train_error_dict = {2:[],
                    6:[],
                    10:[]}
test_error_dict = {2:[],
                   6:[],
                   10:[]}
b_hat_dict = {2:[],
                   6:[],
                   10:[]}

#Iterate over the polynomials
for d in polynomials:

    #Get training subsets of N samples
    x_train_subset = x_train[:20]
    y_train_subset = y_train[:20]

    #Calculate and store train /test error
    train_error, test_error, b_hat = poly_risk_gen(d, n, x_train_subse
    b_hat_dict[d].append(b_hat)
    train_error_dict[d].append(train_error)
    test_error_dict[d].append(test_error)

#Plot b_hat predictions vs A
x = np.linspace(0,25,1000)
func_g = a @ helper_func(x,p)
func_b_hat_2 = b_hat_dict[2][0] @ helper_func(x,2)
func_b_hat_6 = b_hat_dict[6][0] @ helper_func(x,6)
func_b_hat_10 = b_hat_dict[10][0] @ helper_func(x,10)

#B_hat predictions given
plt.figure(figsize=(15,8))
plt.scatter(x_train[:20],y_train[:20])
plt.plot(x, func_b_hat_2)
plt.plot(x, func_b_hat_6)
plt.plot(x, func_b_hat_10)
plt.legend(labels=['f_b_hat(x,d=2)','f_b_hat(x,d=6)','f_b_hat(x,d=10)'
plt.title("Plot 4.2: Ozone/Wind Estimation Functions vs Training Data,
plt.yscale('log')
plt.savefig('Plots//Plot_4_2')
```





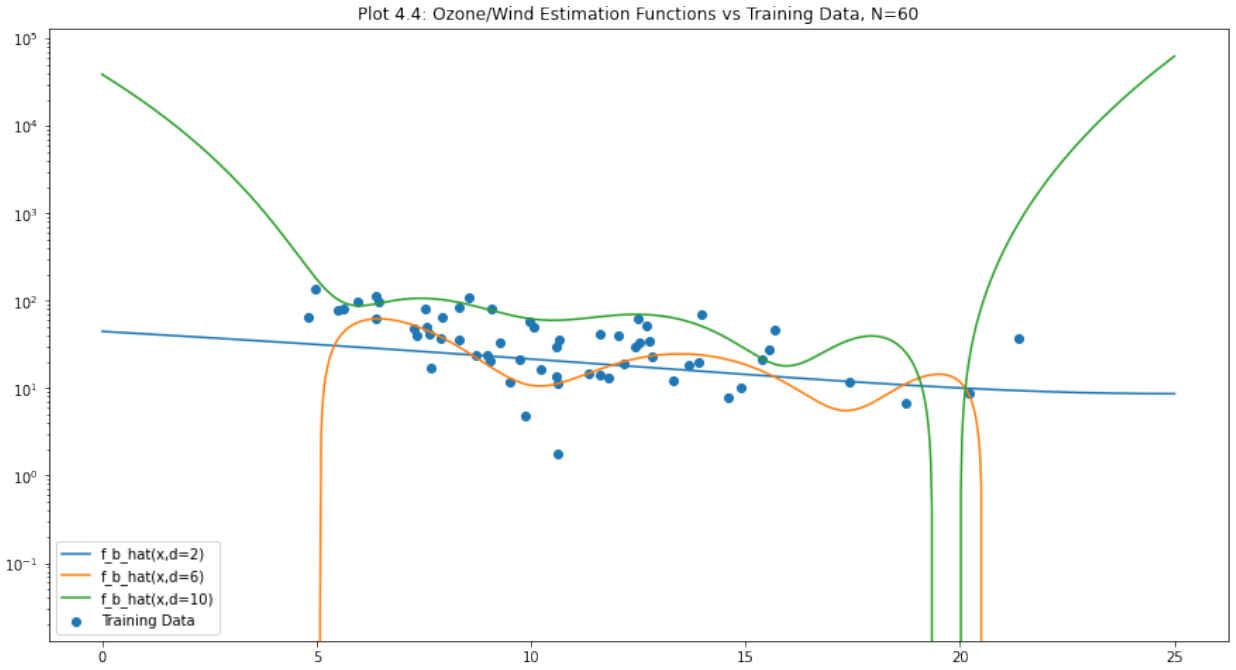
In [24]:
```
#Demonstrate how the polynomials generate different generalization err
n = list(range(40))
polynomials = [2,6,10]

#Generate helper storage variables
train_error_dict = {2:[],
                    6:[],
                    10:[]}
test_error_dict = {2:[],
                   6:[],
                   10:[]}
b_hat_dict = {2:[],
                   6:[],
                   10:[]}

#Iterate over the polynomials
for d in polynomials:

    #Get training subsets of N samples
    x_train_subset = x_train[:20]
    y_train_subset = y_train[:20]

    #Calculate and store train /test error
    train_error, test_error, b_hat = poly_risk_gen(d, n, x_train_subse
    b_hat_dict[d].append(b_hat)
    train_error_dict[d].append(train_error)
    test_error_dict[d].append(test_error)

#Plot b_hat predictions vs A
x = np.linspace(0,25,1000)
func_g = a @ helper_func(x,p)
func_b_hat_2 = b_hat_dict[2][0] @ helper_func(x,2)
func_b_hat_6 = b_hat_dict[6][0] @ helper_func(x,6)
func_b_hat_10 = b_hat_dict[10][0] @ helper_func(x,10)

#B_hat predictions given
plt.figure(figsize=(15,8))
plt.scatter(x_train[:40],y_train[:40])
plt.plot(x, func_b_hat_2)
```

```
plt.plot(x, func_b_hat_6)
plt.plot(x, func_b_hat_10)
plt.legend(labels=['f_b_hat(x,d=2)','f_b_hat(x,d=6)','f_b_hat(x,d=10)'
plt.title("Plot 4.3: Ozone/Wind Estimation Functions vs Training Data,
plt.yscale('log')
plt.savefig('Plots//Plot_4_3')
```



In [25]:
```
#Demonstrate how the polynomials generate different generalization err
n = list(range(60))
polynomials = [2,6,10]

#Generate helper storage variables
train_error_dict = {2:[],
                    6:[],
                    10:[]}
test_error_dict = {2:[],
                   6:[],
                   10:[]}
b_hat_dict = {2:[],
              6:[],
              10:[]}

#Iterate over the polynomials
for d in polynomials:

    #Get training subsets of N samples
    x_train_subset = x_train[:20]
    y_train_subset = y_train[:20]

    #Calculate and store train /test error
```

```
    train_error, test_error, b_hat = poly_risk_gen(d, n, x_train_subse
    b_hat_dict[d].append(b_hat)
    train_error_dict[d].append(train_error)
    test_error_dict[d].append(test_error)

#Plot b_hat predictions vs A
x = np.linspace(0,25,1000)
func_g = a @ helper_func(x,p)
func_b_hat_2 = b_hat_dict[2][0] @ helper_func(x,2)
func_b_hat_6 = b_hat_dict[6][0] @ helper_func(x,6)
func_b_hat_10 = b_hat_dict[10][0] @ helper_func(x,10)

#B_hat predictions given
plt.figure(figsize=(15,8))
plt.scatter(x_train[:60],y_train[:60])
plt.plot(x, func_b_hat_2)
plt.plot(x, func_b_hat_6)
plt.plot(x, func_b_hat_10)
plt.legend(labels=['f_b_hat(x,d=2)','f_b_hat(x,d=6)','f_b_hat(x,d=10)'
plt.title("Plot 4.4: Ozone/Wind Estimation Functions vs Training Data,
plt.yscale('log')
plt.savefig('Plots//Plot_4_4')
```



This time around, the generalization error of each polynomial does not converge to a certain value, which is very interesting. A polynomial of degree 2 performed the best for both empirical and generalization risk, across varying amounts of training observation data used to fit our model $N \in \{20, 40, 60\}$. As we don't know the Bayes function, it is impossible to comment on the extent that approximation risk is present in our current total Risk, as the ozone and wind may or not may be explained by a polynomial relationship. Estimation risk is present in our model as the generalization error does not converge. If we had more data points, we could eliminate estimation risk as N would increase the generalization error of our various polynomial functions would converge to the same value. Optimization risk should not be present in our excess risk decomposition, as we are still using the closed form linear algebra solution for ordinary least squares. If our N observations grew untractably large, perhaps we'd have to resort to gradient descent methods, in which case, optimization error would be present.