lab3

February 6, 2024

1 Lab 3 - Intro to Regression - Modeling COVID19

The goals of this lab are to:

- Examine and practice using linear regression for predictions

- Describe gradient descent
- Use learning curves to visualize and diagnose how well a machine learning algorthm is working.
- Using polynomial regression to model COVID19
- Discuss the systematic uncetainties arising from machine learning models.

Authors: Bernie Boscoe & Tuan Do

Last Update: Tuan Do

1.1 Question 0

(3 pts)

Put the number you get from the mid-quarter class survey in the cell below

6563

Please note I had to fill this out twice as I didn't see the number or know I needed it the first time. The second time I put "-1" as the answer to the first question so that it can be filtered out of the survey data.

2 Linear Regression

Linear regression- one of the simplest models. For this section, we'll be leaning on HOML Chapter 4, because it has a lot of good material, in particular, the two differing ways to train a lin reg model:
1. "Closed form" equation that directly computes the model parameters that best fit the model to the training set (minimizing the cost function)
2. Using an iterative optimization approach called Gradient Descent that minimizes the cost function over the training set, converging to the same parameters as the first method. (This might be new to you.)
Linear Regression model prediction is commonly written as:

$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n$
$\hat{y}$ is the predicted value

n is the number of features
x_i is the i^{th} feature value
θ_j is the j^{th} model parameter (including the bias term θ_0)
the feature weights $\theta_1, \theta_2, \dots, \theta_n$.

The vectorized form is as follows:

$\hat{y} = h_\theta(x) = \theta \cdot x$
Where θ is the model's parameter vector, containing the bias term θ_0
the feature weights θ_1 to θ_n
x is the instance's feature vector, containing x_0 to x_n
with x_0 always equal to 1
$\theta \cdot x$ is the dot product of the vectors
θ and x
h_θ is the hypothesis function, using the model parameters θ.

Important: vectors are often represented as column vectors, so if θ and x are column vectors then the prediction is $\hat{y} = \theta^T x$, where $\theta^T x$ is the matrix multiplication of the two.

We need to find the value of θ that minimizes our error.

$$MSE(X, h_\theta) = \frac{1}{m}\sum_{i=1}^{m}(\theta^T x^{(i)} - y^{(i)})^2$$

2.1 Regression Evaluation Metrics

Here are three common evaluation metrics for regression problems. There are many, and these are given to you as the most simple versions (note they are in slightly different notation common in machine learning):

Mean Absolute Error (MAE) is the mean of the absolute value of the errors:

$$\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Mean Squared Error (MSE) is the mean of the squared errors:

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Root Mean Squared Error (RMSE) is the square root of the mean of the squared errors:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Comparing these metrics:

- **MAE** is the easiest to understand, because it's the average error.

- **MSE** is more popular than MAE, because MSE "punishes" larger errors, which tends to be useful in the real world.
- **RMSE** is even more popular than MSE, because RMSE is interpretable in the "y" units.

All of these are **cost functions**, because we want to minimize them. Important- there is confusion as to the difference between the cost function and loss function. Generally, the loss function is a single instance of a cost function, and the cost function is the average over the parameters. But you will see abuse of these definitions for sure in the ML space with cost and loss being interchangable. Just make sure you define what function you use in your research!

For a warm-up, let's first find the value of θ that minimizes the MSE cost function using the closed-form solution called the least squares equation:

$$\hat{\theta} = (X^T X)^{-1} X^T y$$

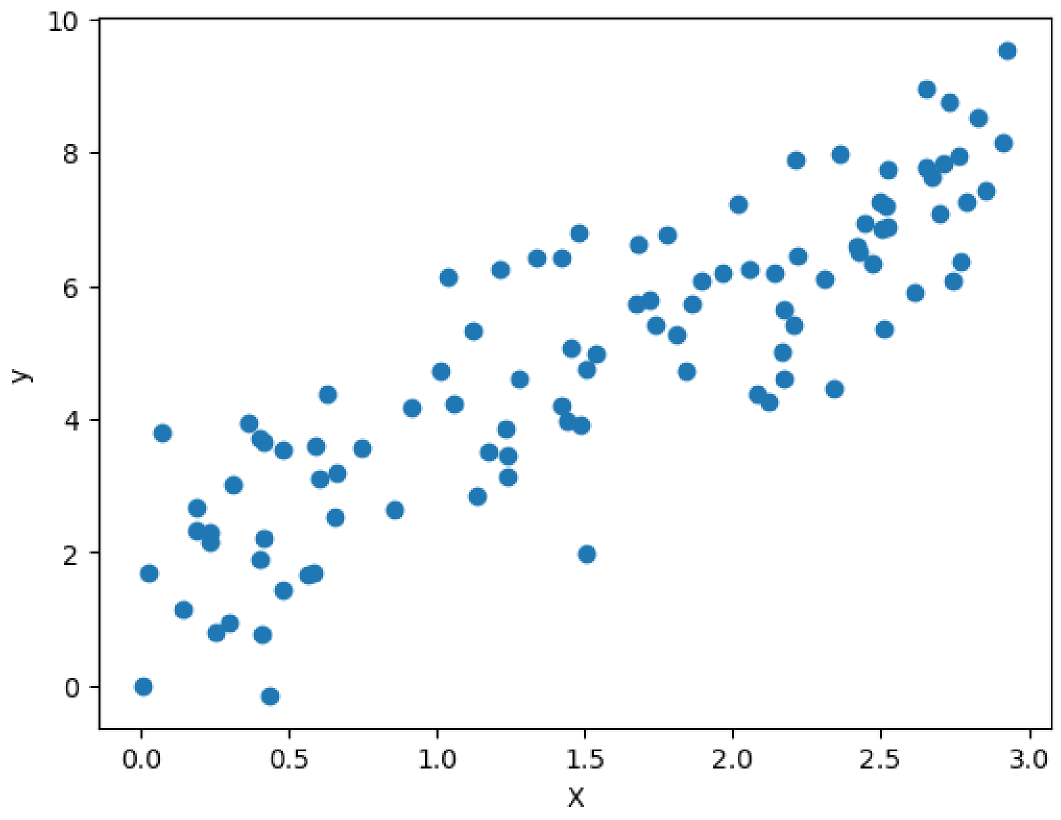
where capital X is the entire training set (so, a matrix of vectors of x's, or instances). Using this formula we do not use gradient descent. See https://www.geeksforgeeks.org/ml-normal-equation-in-linear-regression/#:~:text=Normal%20Equation%20is%20an%20analytical,a%20dataset%20with%20small%20features for an explanation of how the MSE cost function from above is transformed into matrix notation.

From a mathematical perspective, you might see that to use the least squares equation you would need to compute an inverse and sometimes that is impossible. Another consideration in optimization and regression problems is computational complexity, which affects computing speed, and so on. Scikit-learn's LinearRegression class uses SVD (Singular Value Decomposition) that decomposes the training set X in a way that is more efficient than computing the normal equation- which gets us to an important conversation to have: When using tools such as scikit-learn, how much do you need to know about the actual mathematics built into the class? Generally speaking, a machine learning practicioner in say, finance, might not care about what is under the hood and instead cares about speed and accuracy and not the math that gets them there. In the case of a researcher in the physical sciences, it is likely a different case what metrics the researcher cares about! Now we go to a simple introduction of these concepts. Let's import:

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn.metrics import accuracy_score
from sklearn import datasets, model_selection
import pandas as pd
import seaborn as sns # reminder: you may use any plotting tool you like!
%matplotlib inline
```

[2]:
```
# Warmup. let's  generate some random linearish data.
X = 3* np.random.rand(100,1)
y = 2 + 2*X + np.random.randn(100,1)
```

[3]:
```
plt.scatter(X,y)
plt.xlabel('X')
plt.ylabel('y')
plt.show()
```



2.2 Question 1

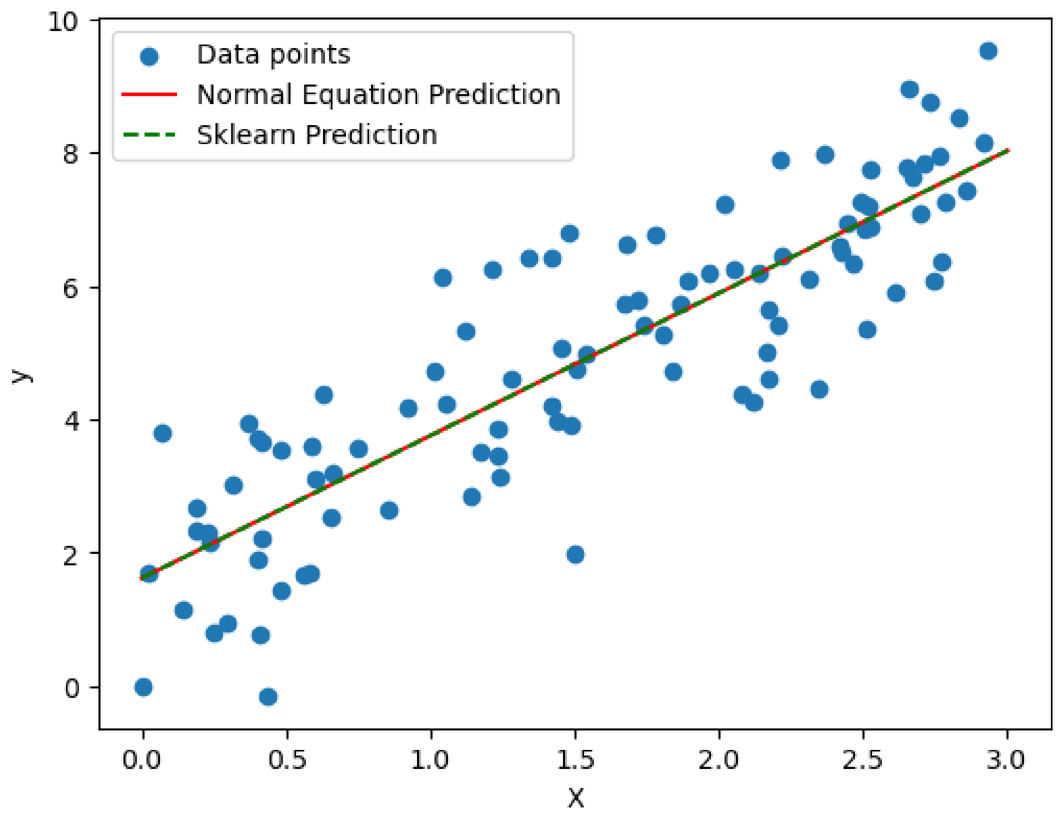
(6 pts)

1. Using the normal equation and numpy, compute $\hat{\theta}$
2. plot your regression line.
3. Use the sklearn LinearRegression model, repeat and fit line.

[4]:
```
from sklearn.linear_model import LinearRegression

# Adding x0 = 1 to each instance for the normal equation
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance

# Solve the normal equation
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)

# Fitting the model using sklearn for comparison
lin_reg = LinearRegression()
lin_reg.fit(X, y)
```

```python
y_intercept_sklearn = lin_reg.intercept_
y_slope_sklearn = lin_reg.coef_

# Print the results
print("Normal Equation: Intercept, Slope:", theta_best[0], theta_best[1])
print("Sklearn LinearRegression: Intercept, Slope:", y_intercept_sklearn,␣
 ↪y_slope_sklearn)

# Plotting the data and the model predictions
plt.scatter(X, y, label='Data points')
X_new = np.array([[0], [3]])  # Choose points to plot the line
y_predict_normal = X_new * theta_best[1] + theta_best[0]
y_predict_sklearn = lin_reg.predict(X_new)
plt.plot(X_new, y_predict_normal, "r-", label="Normal Equation Prediction")
plt.plot(X_new, y_predict_sklearn, "g--", label="Sklearn Prediction")
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()
```

```
Normal Equation: Intercept, Slope: [1.61520198] [2.13615551]
Sklearn LinearRegression: Intercept, Slope: [1.61520198] [[2.13615551]]
```



2.3 Question 2

(8 pts)

Next we are going to use the "machine learning way."
1. Using the same datasets X and y,import sklearn train_test_split and split X,y into X_train, X_test, y_train, y_test with a test_size of 0.3 and random_state = 42 .
2. Train/Fit model to training set
3. Create predictions from X_test
4. Create a scatterplot of the real test values versus the predicted values.

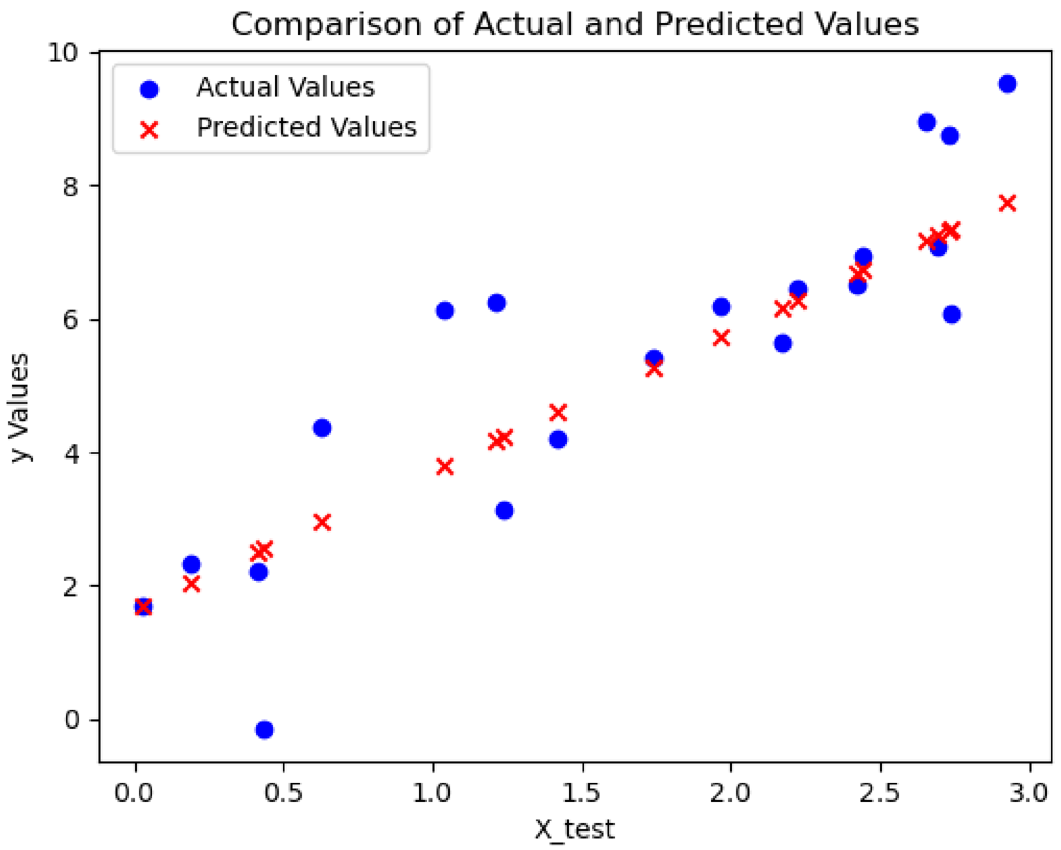
[5]:
```
from sklearn.model_selection import train_test_split
# Split up data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
```

[6]:
```
# Create a linear regression model and fit it to the training data
lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)
```

[6]:
```
LinearRegression()
```

[7]:
```
# Make predictions on the test set
y_pred = lin_reg.predict(X_test)
```

[8]:
```
# Plotting both actual and predicted values
plt.scatter(X_test, y_test, color='blue', label='Actual Values')
plt.scatter(X_test, y_pred, color='red', marker='x', label='Predicted Values')

# Labeling the axes and the plot
plt.xlabel("X_test")
plt.ylabel("y Values")
plt.title("Comparison of Actual and Predicted Values")
plt.legend()
plt.show()
```



2.4 Question 3

(4 pts)

1. Calculate the Mean Absolute Error, the Mean Squared Error, and the Root Mean Squared Error.
2. Next, You should have gotten a very good model with a good fit. Let's quickly explore the residuals to make sure everything was okay with our data. Plot a histogram of the residuals and make sure it looks normally distributed. Use either seaborn distplot, or just plt.hist().

[9]:
```
from sklearn.metrics import mean_absolute_error, mean_squared_error
# Assuming I'm not meant to define manually since we imported metrics

# Calculate Mean Absolute Error
mae = mean_absolute_error(y_test, y_pred)

# Calculate Mean Squared Error
mse = mean_squared_error(y_test, y_pred)

# Calculate Root Mean Squared Error
rmse = np.sqrt(mse)
```

```python
# Print the errors
print(f"Mean Absolute Error (MAE): {mae:.2f}")
print(f"Mean Squared Error (MSE): {mse:.2f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.2f}")
```

```
Mean Absolute Error (MAE): 0.94
Mean Squared Error (MSE): 1.57
Root Mean Squared Error (RMSE): 1.25
```

3 Gradient Descent

For now, sticking with our toy dataset, we will optimize with Gradient Descent. You wouldn't ordinarily use gradient descent with linear regression because you can just solve directly with the closed form least squares equation. (For example you might ask: Is there an analytical (closed-form) solution to Logistic Regression similar to the Normal Equation for Linear Regression? Unfortunately, there is no closed-form solution for maximizing the log-likelihood (or minimizing the inverse, the logistic cost function); at least it has not been found, yet).
But, for demonstration purposes we will use gradient descent on a linear regression example. In this way, using gradient descent which is iterative, we can tweak various hyperparameters to minimize a cost function. Hyperparameters include the size of the steps taken to find the minimum, called the learning rate. If the learning rate is too small, the algorithm will have to iterate many times before converging, which is slow. Conversely, if the learning rate is too high, you might end up not converging to the global minimum. When using gradient descent, you should ensure features have a similar scale to speed up convergence. Oftentimes in ML literature, you will see the learning rate set to a default parameter. Of course, you can test out various learning rates and see what happens, part of our exercise task.

There are a variety of gradient descent algorithms, and the 'best' change often, we will use the batch gradient descent which you would likely not use in real life, but it provides an understanding of the algorithm's inner workings. The cost function is commonly written in this simple form:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

(when you see it in this form $h_\theta = \theta^T x$)
and taking the partial derivatives of the cost function,

$$\frac{\partial}{\partial\theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

Cost functions are 'nice' because their first derivatives are convex.

HOML uses the matrix form of the cost function and its partial derivatives (shown here: https://rpubs.com/dnuttle/ml-linear-cost-func-derivative), but here we will break it down into a simpler form so you can see the steps easier. We use the classic y = mx + b function for a line, and loss is the error in the predicted value of m and b (ordinary least squares). We will initialize m and b to 0, and they will be updated iteratively as the error becomes smaller. so then our cost function now becomes

$$E = \frac{1}{n}\sum_{i=1}^{n}(y_i - (mx_i + b)^2$$

. Notice we changed m to n because m is now our slope. And what happened to the 1/2 in the other cost function? Well, we are showing you what you'll see out there in the real world- some are multiplied by 1/2 so when you take the partial derivatives they cancel, at any rate its just a constant so it doesn't matter, as P&A folks know all too well. Now, we need the partial derivatives with respect to m and b:

$$D_m = \frac{-2}{n}\sum_{i=1}^{n} x_i(y_i - \hat{y_i}))$$

which becomes in our code: D_m = (-2/n) * sum(X * (y - Y_pred)) # Derivative wrt m. Likewise the partial derivative wrt b is:

$$D_b = \frac{-2}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})$$

(see https://ml-cheatsheet.readthedocs.io/en/latest/linear_regression.html for how to compute each partial derivative) I'll leave it to you as an exercise to translate into code.

3.1 Question 4

(6 pts)

To create a gradient step algorithm, we set m and b to 0 (this starts our prediction where?)
We also need to set a learning rate, let's try L = 0.0001 In our loop, 1. set Y_pred = m$X + b$*, the current predicted value of Y 2. get derivative wrt m, set it to D_m 3. get derivative wrt b, set it to D_b 4. update m by m - L*D_m 5. update c 6. print out m, b and in a cell below plot r on the scatter plot.

For the slope ((m)):

$$\frac{\partial}{\partial m} = \frac{-2}{N}\sum(y_i - (mx_i + b))x_i$$

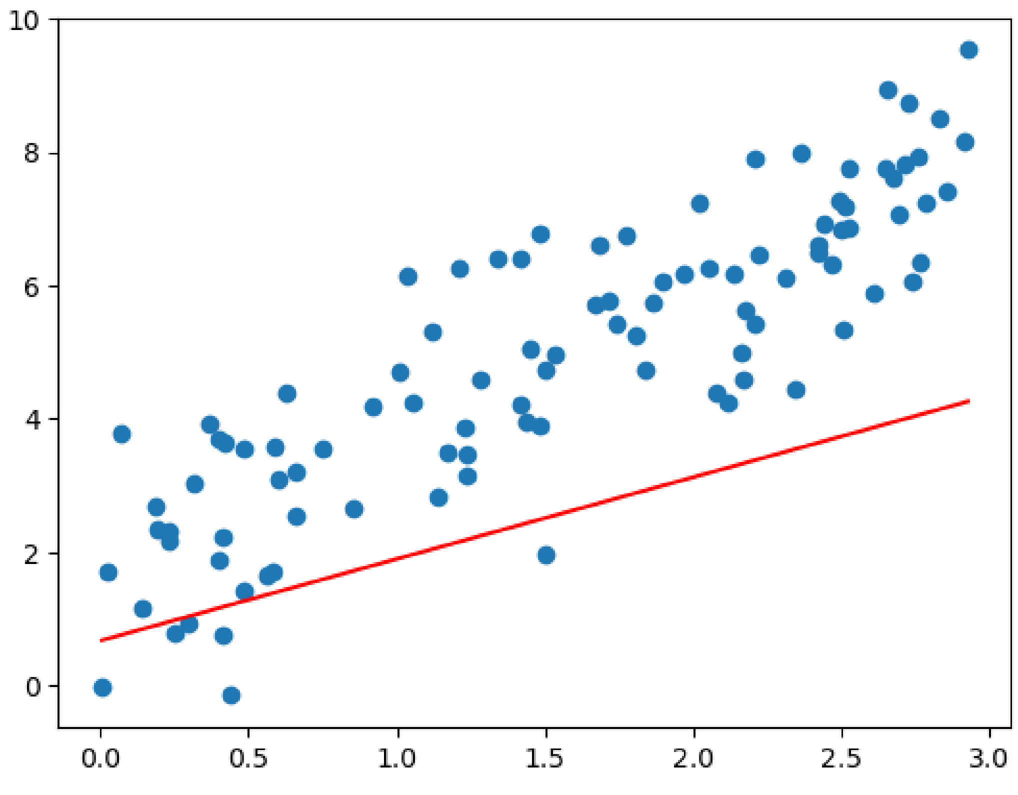
For the intercept ((b)):

$$\frac{\partial}{\partial b} = \frac{-2}{N}\sum(y_i - (mx_i + b))$$

[10]:
```
m = 0
b = 0
L = 0.0001  # The learning Rate
epochs = 1000  # The number of iterations to perform gradient descent
n = float(len(X)) # Number of elements in X

# Gradient Descent
for i in range(epochs):
    Y_pred = m * X_train + b  # The current predicted value of Y
    D_m = (-2/len(X_train)) * sum(X_train * (y_train - Y_pred))  # Derivative␣
 ↪wrt m
    D_b = (-2/len(X_train)) * sum(y_train - Y_pred)  # Derivative wrt b
    m = m - L * D_m  # Update m
    b = b - L * D_b  # Update b

# Print out the final values of m and b
print(f"Final slope (m): {m}")
print(f"Final intercept (b): {b}")
```

```
Final slope (m): [1.23165807]
Final intercept (b): [0.67375888]
```

[11]:
```
plt.scatter(X, y)
plt.plot([min(X), max(X)], [min(Y_pred), max(Y_pred)], color='red')  #␣
 ↪regression line
plt.show()
# wait, what happened? Check your learning rate.
```



3.2 Question 5

(3 pts)

1. Can you find a better learning rate for your regression task?

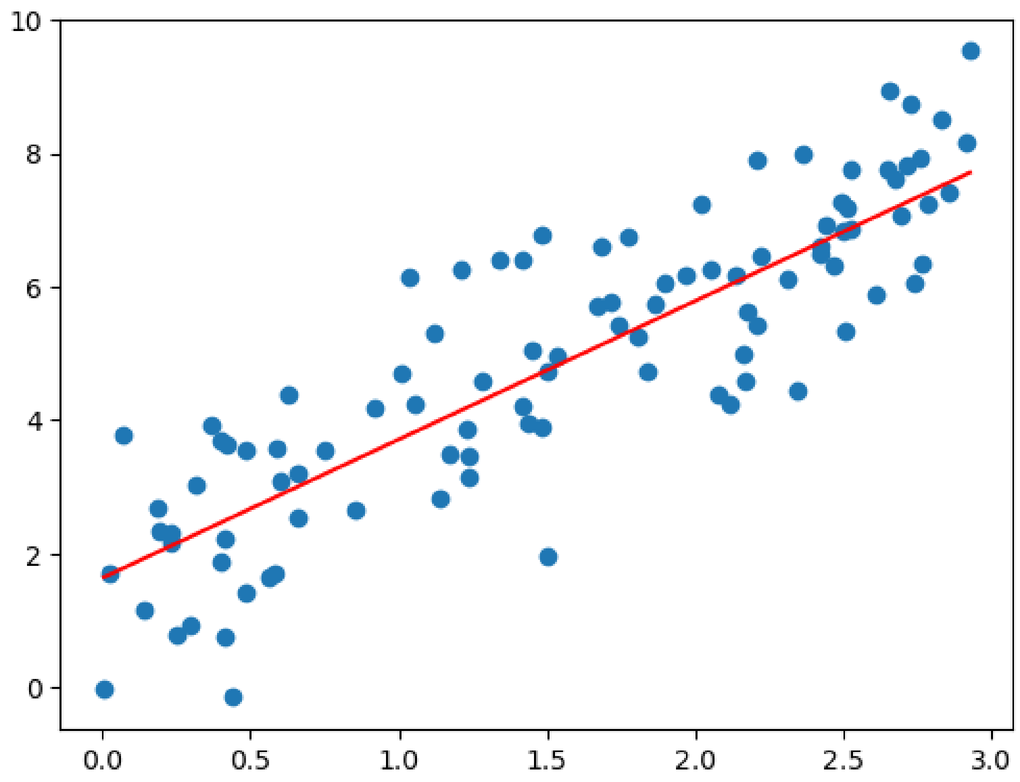
   2. What is the regression prediction equation (using coefficients) using LinearRegression()? Using the gradient descent algorithm (assuming you've found the right learning rate and epochs?) Are they the same?

   3. Three main Gradient Descent methods are Batch, Stochastic, and Mini Batch. In a few sentences, explain the differences between them. (these become important concepts with say building a logistic regression neural network that is very large, perhaps a classifier for images of animals)

[12]:
```
# Find a better learning rate
m = 0
b = 0
L = 0.01  # The learning Rate
epochs = 1000  # The number of iterations to perform gradient descent
n = float(len(X)) # Number of elements in X

# Gradient Descent
for i in range(epochs):
    Y_pred = m * X_train + b  # The current predicted value of Y
    D_m = (-2/len(X_train)) * sum(X_train * (y_train - Y_pred))  # Derivative␣
 ↪wrt m
    D_b = (-2/len(X_train)) * sum(y_train - Y_pred)  # Derivative wrt b
    m = m - L * D_m  # Update m
    b = b - L * D_b  # Update b

# Print out the final values of m and b
print(f"Final slope (m): {m}")
print(f"Final intercept (b): {b}")
```

```
Final slope (m): [2.08270173]
Final intercept (b): [1.64178786]
```

[13]:
```
plt.scatter(X, y)
plt.plot([min(X), max(X)], [min(Y_pred), max(Y_pred)], color='red')  #␣
 ↪regression line
plt.show()
```



2. What are the predictions? Do they agree? The prediction coefficients are m and b or θ_1 and θ_2 respectively.

The values we found for each are:

[38]:
```
print("Normal Equation: Intercept, Slope:", theta_best[0], theta_best[1])
print("Sklearn LinearRegression: Intercept, Slope:", y_intercept_sklearn,␣
  ↪y_slope_sklearn)
print("Gradient Descent:", (m,b))
```

```
Normal Equation: Intercept, Slope: [1.61520198] [2.13615551]
Sklearn LinearRegression: Intercept, Slope: [1.61520198] [[2.13615551]]
Gradient Descent: (array([2.08270173]), array([1.64178786]))
```

The values roughly agree, all approximating the expected (2,2) The intial two methods found fit with a slightly lower slope and slightly higher intercept. Gradient descent finds roughly the correct slope but is a bit off on the intercept.

4 Polynomial Regression

(Can be found in Chapter 4 of HOML) We are now going to use time series data from last year about Covid cases across the world. For this final section, we will use our 8 steps in ML

4.1 Step 1. Big Picture

For this task, we will be looking at positive Covid test rates all over the world, from Jan 2020- Aug 2020. Incidentally this is a very popular set used on Kaggle in inumerable ways. Our goals are as follows:
1. Try to model a slice of our dataset using Polynomial Regression. Our metrics will include: Mean squared error (MSE) to assess training vs test sets, and finding the optimal order (degree) of the polynomial, and then trying to assess if we've overfit, or underfit, and possible remedies. (If, during the lab, you are wondering 'why are we doing it this way?' Know that, this is an actual job interview assessment in "the real world". And, of course if you would like to propose a 'better way' to attack the problem (what order, lowest MSE, or, another type of fit, that would be a great discussion topic as well).

2. Read about the LinearRegression() and PolynomialRegression() classes in scikit learn, get a feel for how to use them. HOML is useful, as well as scikit-learn documentation.

Another important note: We will be using the PolynomialRegression() class to fit our data to a LinearRegression() model (Read HOML for a brief explanation). Because we are using LinearRegression() in scikit-learn, we will **not** be using Gradient Descent here! We rely on scikit-learn's LinearRegression() class which uses singular value decomposition in its calculation of ordinary least squares. (Extra credit: You can normalize as one of your parameters you are passing to the class, as always understanding the scikit-learn classes, what parameters you can set and pass, and how things work under the hood are always helpful to read up on)

So, keep it simple, there's not too much coding or work that goes into this task. Good luck! Ask for hints in slack or to check if you are on the right track.

4.2 Step 2. Get Data

[15]:
```python
import numpy.matlib as matlib
import pandas as pd
# load the time_series_covid_19_confirmed.csv using pandas in your lab3 folder.
data = pd.read_csv('time_series_covid_19_confirmed.csv')
```

4.3 3. Explore Data

4.4 Question 6

(5 pts)

Check the shape, and head, plot, visualize, and have a look at the data with whatever ways you think will help your task. Comment throughout on observations. Note: There are NaNs but you do NOT have to worry about them for this task.

[16]:
```python
data.shape
```

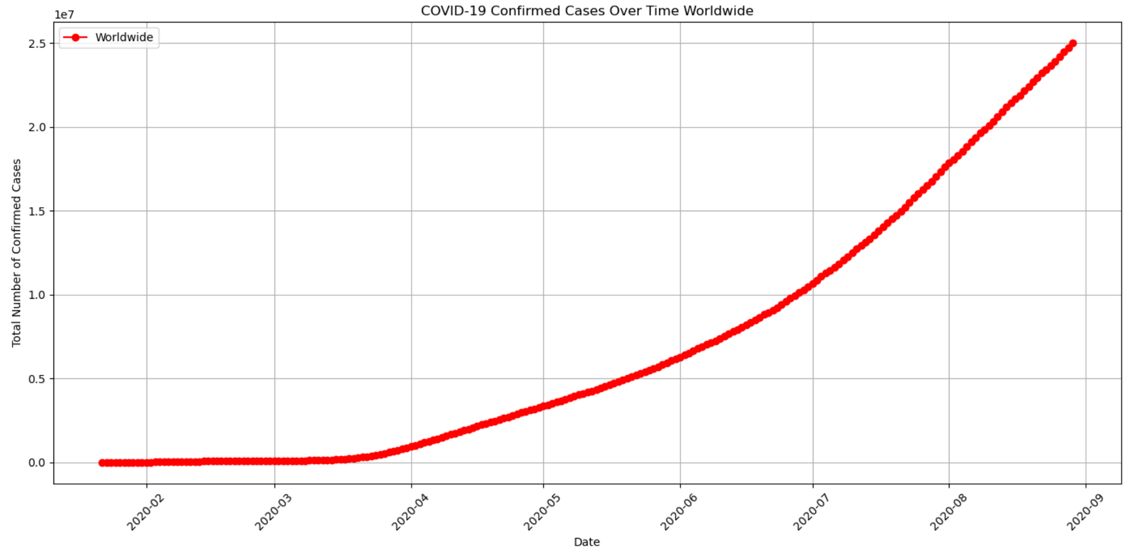
[16]:
```
(266, 225)
```

[17]:
```python
data.head()
```

```
[17]:   Province/State Country/Region       Lat       Long  1/22/20  1/23/20  \
      0            NaN    Afghanistan  33.93911  67.709953        0        0
      1            NaN        Albania  41.15330  20.168300        0        0
      2            NaN        Algeria  28.03390   1.659600        0        0
      3            NaN        Andorra  42.50630   1.521800        0        0
      4            NaN         Angola -11.20270  17.873900        0        0

         1/24/20  1/25/20  1/26/20  1/27/20  …  8/20/20  8/21/20  8/22/20  \
      0        0        0        0        0  …    37856    37894    37953
      1        0        0        0        0  …     7967     8119     8275
      2        0        0        0        0  …    40258    40667    41068
      3        0        0        0        0  …     1024     1045     1045
      4        0        0        0        0  …     2044     2068     2134

         8/23/20  8/24/20  8/25/20  8/26/20  8/27/20  8/28/20  8/29/20
      0    37999    38054    38070    38113    38129    38140    38143
      1     8427     8605     8759     8927     9083     9195     9279
      2    41460    41858    42228    42619    43016    43403    43781
      3     1045     1060     1060     1098     1098     1124     1124
      4     2171     2222     2283     2332     2415     2471     2551

      [5 rows x 225 columns]
```

Looks like this dataset lists 266 locations divided by country or by province/region. They are sorted in alphabetical order by country name and the latitude and longitude is given for each row entry. The remaining collumns are dates beginning with Jan 1st 2020 and counting day by day for just over 8 months. Each entry shows the a number, presumably the number of reported COVID-19 cases in the given region as of that date.

```
[18]: worldwide_data = data.iloc[:, 4:].sum(axis=0)

      # Converting the index to datetime format with a specified format
      # Assuming the date format is month/day/2-digit-year
      worldwide_data.index = pd.to_datetime(worldwide_data.index, format='%m/%d/%y')

      # Plotting the worldwide aggregated time series
      plt.figure(figsize=(14, 7))
      plt.plot(worldwide_data.index, worldwide_data.values, marker='o',␣
       ↪linestyle='-', color='red', label='Worldwide')
      plt.title('COVID-19 Confirmed Cases Over Time Worldwide')
      plt.xlabel('Date')
      plt.ylabel('Total Number of Confirmed Cases')
      plt.xticks(rotation=45)
      plt.legend()
      plt.grid(True)
      plt.tight_layout()
      plt.show()
```



4.5 4. Prepare Data

This step is mostly done for you. But- I highly recommend tracing through this so you understand this common data manipulation. Notice Pandas is being used to examine the data, but we will transform data into different arrays and don't need to think about Pandas when we are analyzing.

[19]:
```
# let's only use 50 rows and 100 columns, just to make this easier (and faster)
rows=50
cols=100
data_new=data.iloc[0:rows,4:cols+4]
print(data_new.shape)
```

```
(50, 100)
```

[20]:
```
# Now we are going to convert the Tablular data to format {X,Y}, where␣
 ↪X={Longitude, Latitude, Date}, Y={#infected}
#Finding the date indices
import matplotlib.pyplot as plt
data_row=data_new.sum(axis=0) # bb per day summing up positive cases into cols
days=range(0,data_row.shape[0])
days_mat=matlib.repmat(np.array(days),data_new.shape[0],1)
print(days_mat.shape)
```

```
(50, 100)
```

[21]:
```
#Lets create data X-{X1,X2,X3}, where X1=lat, X2=long, X3=date, Y=#affected
X=np.zeros((days_mat.shape[0]*days_mat.shape[1],3))
Y=np.zeros((days_mat.shape[0]*days_mat.shape[1],1))
lat_long=np.array(data.iloc[:,2:4])
```

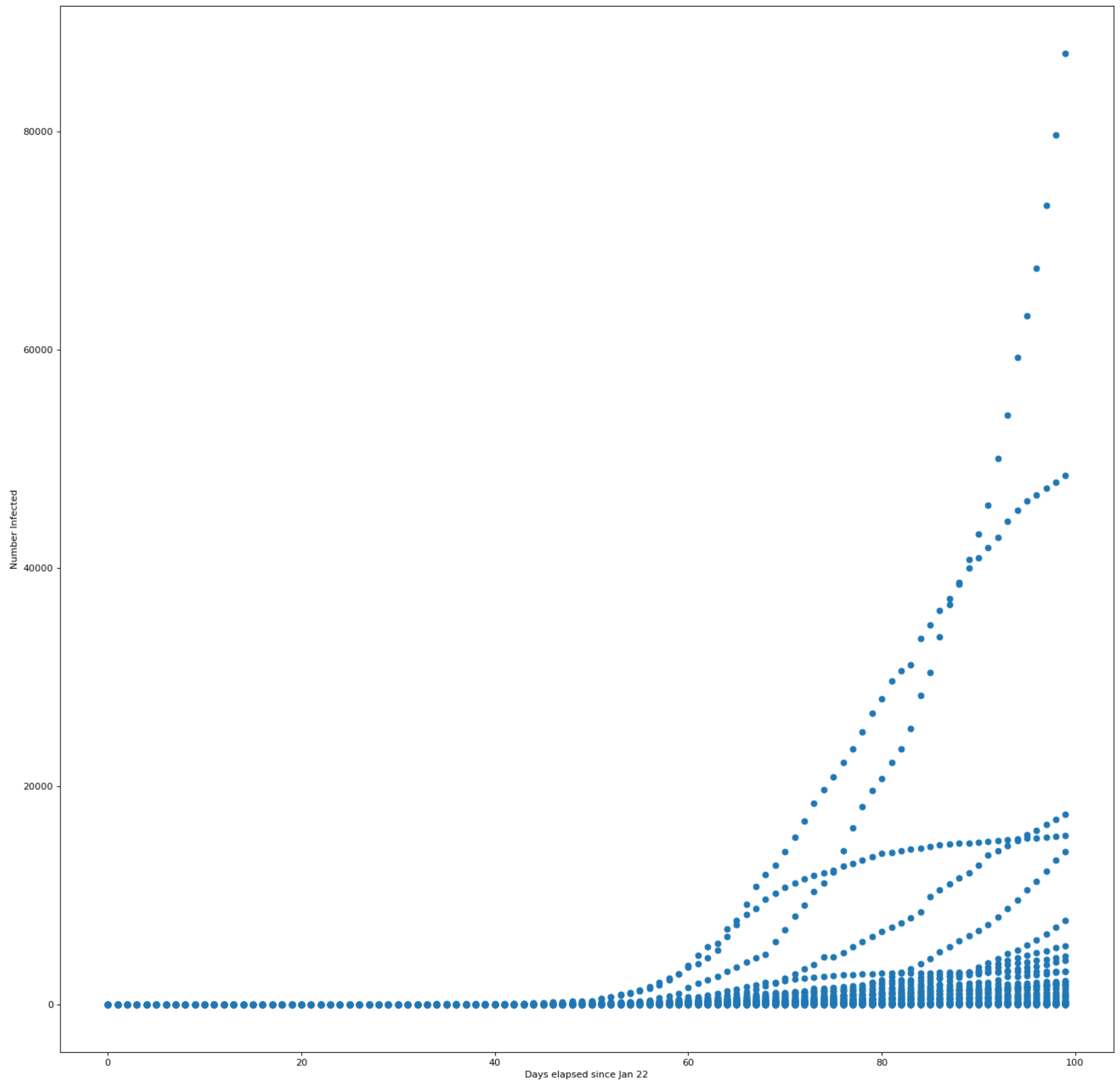
```
data_new=np.array(data_new)
for r in range(days_mat.shape[0]): #all locations
  X[r*days_mat.shape[1]:r*days_mat.shape[1]+days_mat.
↪shape[1],0]=lat_long[r,0]*np.ones((days_mat.shape[1],)) #setting Latitude
  X[r*days_mat.shape[1]:r*days_mat.shape[1]+days_mat.
↪shape[1],1]=lat_long[r,1]*np.ones((days_mat.shape[1],)) #setting Longitude
  X[r*days_mat.shape[1]:r*days_mat.shape[1]+days_mat.shape[1],2]=np.
↪reshape(days,(days_mat.shape[1],)) #setting the date
  Y[r*days_mat.shape[1]:r*days_mat.shape[1]+days_mat.shape[1]]=np.
↪reshape(data_new[r,:],((days_mat.shape[1],1)))
```

[22]:
```
print(np.shape(X))
#Data Preparation is Done!
```

```
(5000, 3)
```

[23]:
```
# Let's plot the data in this new format
plt.figure(figsize=(20, 20), dpi=80)
plt.scatter(X[:,2],Y[:])
plt.xlabel('Days elapsed since Jan 22')
plt.ylabel('Number Infected')
print(np.shape(X))
```

```
(5000, 3)
```



4.6 5. Select Model and Train-

If you are wondering why we've gone back to just test and train, it is merely to mimic the actual job interview assignment. (In general, train, test, val is better, this is faster)

4.7 Question 7

(5 pts)

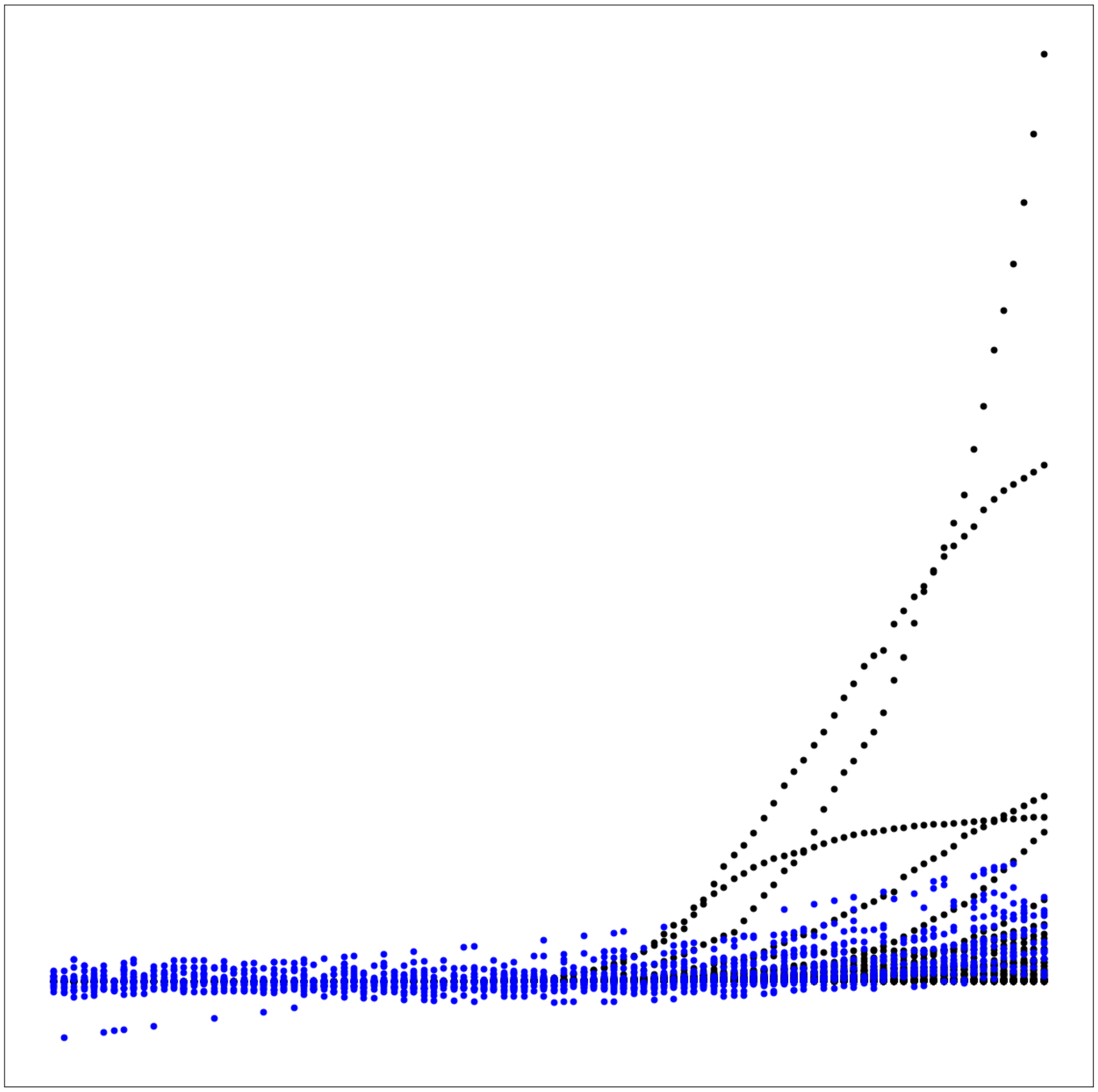
```
[24]: #Goal 1: Fit regression model for (days vs infections)
      from sklearn.model_selection import train_test_split
      #Split data to train and test
      X_train_1, X_test_1, y_train, y_test = train_test_split(X, Y, test_size=0.3,␣
       ↪random_state=10)
```

```
# If you are tinkering, you'd use X_train, etc to pass as parameters and rename␣
↪new sets to keep track.
```

[25]:
```
# What do you think about this data? How would you fit a regression model?
# We are going to try Polynomial Regression (in Chap. 4 of HOML)
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

poly = PolynomialFeatures(degree=3) # try out degree 3
#Transform both the train and test data
X_train = poly.fit_transform(X_train_1)
X_test = poly.fit_transform(X_test_1)
print("New shape of test data=",np.shape(X_test))

clf = LinearRegression()
#Fit the model on train data only
clf.fit(X_train, y_train)
print("Linear Regression Coefficients are=",clf.coef_)
print("constant coefficient=",clf.intercept_)
```

```
New shape of test data= (1500, 20)
Linear Regression Coefficients are= [[ 0.00000000e+00  1.49472285e+00
-1.77011169e+01 -7.44377620e+00
  -2.62371164e+00 -1.66783679e-01 -1.32523607e+00  1.32613637e-01
   1.60019698e-01 -5.80740444e-01  4.74622499e-02  8.72080430e-03
   4.50831003e-02 -2.04127549e-03  1.26028661e-02  2.75122310e-03
   2.97102909e-04 -9.33173379e-04 -3.45730697e-03  9.73845530e-03]]
constant coefficient= [515.00766133]
```

[26]:
```
# Predict regression outcome on test data
y_pred = clf.predict(X_test)
#next plot predictions
plt.figure(figsize=(20, 20), dpi=80)
plt.scatter(X[:,2], Y[:,0],  color='black')
plt.scatter(X_test_1[:,2], y_pred, color='blue', linewidth=1)
plt.xticks(())
plt.yticks(())

plt.show()
```

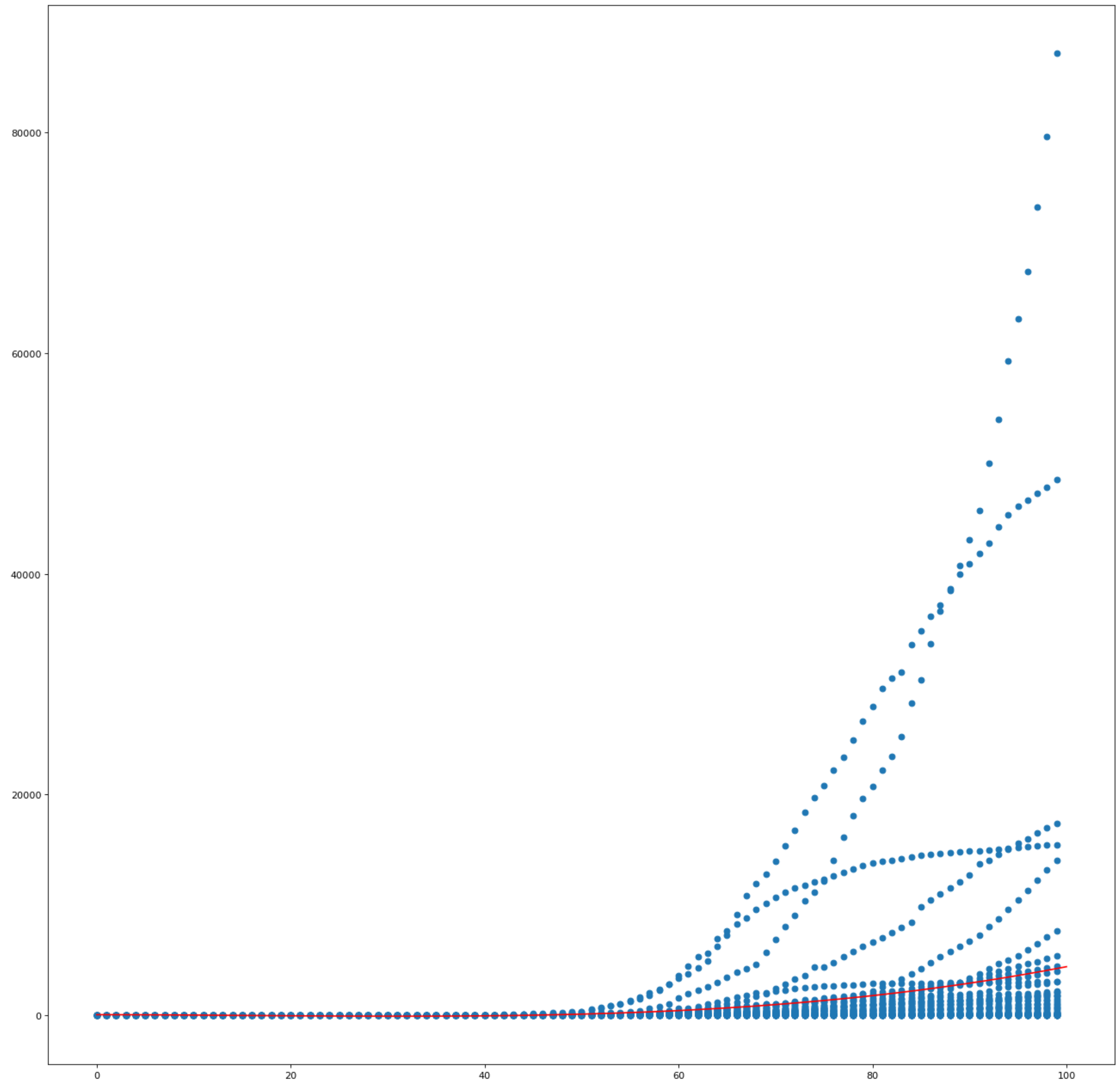


```
[27]: from sklearn.metrics import mean_squared_error as mse
      print(mse(y_test, y_pred))
```

```
20560548.002745155
```

```
[28]: # you can also use numpy
      #1D polynomial regression
      model = np.poly1d(np.polyfit(np.transpose(X_train_1[:,2]), np.
       ↪transpose(y_train[:,0]), 3))
      print(model)
      model_x = np.linspace(0, 100, 100)
      plt.figure(figsize=(20, 20), dpi=80)
      plt.scatter(X[:,2],Y[:])
      plt.plot(model_x, model(model_x),'r')
      plt.show()
```

```
          3           2
0.008161 x - 0.3748 x - 0.6255 x + 53.08
```



5 6. Fine Tune Model

5.1 Question 8

(5 pts)

So, let's say we've decided to use polynomial regression for this task. What order of polynomial will give us the best model? How can we check and make sure we did not overfit or underfit? Here order(degree) is sort of a hyperparameter (but not exactly). 1. Use PolynomialFeatures from above and create an algorithm below that just checks one degree, for example degree = 8.

[29]:
```
# let's try an exponential fit of deg 8
poly = PolynomialFeatures(degree=8)
#Transform both the train and test data
X_train_8 = poly.fit_transform(X_train_1)
X_test_8 = poly.fit_transform(X_test_1)
# print the new shape of the test data
print("New shape of test data=",np.shape(X_test_8))

clf_8 = LinearRegression()
#Fit the model on train data only
clf_8.fit(X_train_8, y_train)
print("Linear Regression Coefficients are=",clf_8.coef_)
print("constant coefficient=",clf_8.intercept_)

# print out mse (mean squared error) How does it compare to the polynomial fit␣
↪of order 3?
y_pred_8 =clf_8.predict(X_test_8)
print('=====================================')

# Compare Error
mse_degree_3 = mean_squared_error(y_test, y_pred)
mse_degree_8 = mean_squared_error(y_test, y_pred_8)

# Printing the MSE for both models to compare
print(f"MSE for polynomial degree 3: {mse_degree_3}")
print(f"MSE for polynomial degree 8: {mse_degree_8}")

# Comparing the errors
if mse_degree_3 < mse_degree_8:
    print("Degree 3 polynomial has lower MSE and may be a better fit.")
else:
    print("Degree 8 polynomial has lower MSE and may be a better fit.")
```

```
New shape of test data= (1500, 165)
Linear Regression Coefficients are= [[-1.44443081e-06 -9.22478204e-03
-1.31418906e-02  8.58407627e-04
   2.68203591e-02 -1.76055460e-01 -5.82394908e-02  1.26530703e-01
   4.64571915e-02  1.13957757e-02 -5.32530809e-01 -7.83261213e-01
   1.88739308e+00 -2.25621526e-01  1.86954049e+00 -8.08479008e-01
  -2.91398210e-02  4.35057400e-01 -3.93215646e-01 -1.15607574e-01
   7.50275582e-03  5.27820330e-02 -8.89387302e-03  1.84305813e-02
   1.24454227e-02 -1.73089691e-02 -5.12212647e-03  1.14416084e-02
  -2.74415967e-02  2.55003333e-02 -1.83550190e-03  2.24331480e-03
  -7.36035287e-03  1.27423411e-02  7.24062080e-03 -7.98412739e-04
  -1.01735211e-03 -3.01007444e-03 -7.44850161e-05 -3.61254796e-03
```

```
   1.22582112e-03  3.20627526e-04 -1.71976204e-03  5.43663242e-04
  -1.73276984e-04  1.31651383e-04 -5.47763372e-04  4.71615459e-05
   1.02394005e-04 -4.56142627e-04  1.83851788e-05 -9.19093210e-05
   1.97701378e-05  5.47086047e-05 -2.49962594e-04 -1.75438430e-04
   7.51796179e-05  2.51368570e-05  5.45538017e-05 -2.46438136e-05
   3.07782882e-05  2.37519610e-06 -1.78771292e-05  2.56069688e-07
   2.28411993e-05 -1.50797966e-05 -2.45786751e-06 -2.95428354e-06
   2.12743165e-05 -9.96413550e-06  5.67852282e-06  1.11672187e-06
  -8.42866934e-07  7.75310975e-06 -3.18501143e-06  1.64891390e-06
   5.31708233e-06  2.67500410e-07 -9.30025890e-08  1.05273529e-06
  -9.11067039e-07 -2.06067659e-07  3.43099620e-06  2.07170719e-06
  -1.82968663e-06 -9.17776486e-07  3.15148498e-07  6.07505043e-07
   1.13193811e-06 -5.97334370e-07  6.65461533e-07  8.51299451e-07
  -1.93247874e-07  1.95634701e-07  1.50508108e-07  4.27446057e-07
   1.04928133e-08 -6.60773620e-09  5.19469436e-08 -4.93593751e-08
   1.82246925e-07 -2.52004642e-08 -1.04148865e-07  5.06133972e-09
  -4.48250592e-08 -2.80054230e-08  5.36208995e-08 -1.59765451e-08
  -4.32920698e-08  5.11773377e-09 -5.81365079e-09 -3.55937679e-08
  -3.67393891e-09  7.34591159e-09 -2.52791718e-09 -4.27104083e-09
   5.40408059e-09  3.98711042e-09 -2.70852376e-08 -1.19578261e-08
   1.33674927e-08  1.04663721e-08 -8.53625728e-09 -3.54411256e-09
  -1.49113067e-08  6.62729571e-09 -7.67187950e-09 -7.08914083e-09
  -2.76214057e-09 -6.49553744e-10 -3.42627013e-09 -1.59204822e-09
  -7.57771890e-09  1.31078945e-09 -2.19391949e-10 -1.73354692e-10
  -7.91459953e-10 -4.65766276e-09  2.50083076e-09 -1.57623914e-11
  -1.82483584e-10  3.61796620e-10 -6.12563278e-10 -1.48823964e-09
   1.68596398e-09 -1.18823340e-10 -6.05028701e-12  1.49887103e-10
   1.15998960e-10 -2.18030705e-10 -2.52696641e-10  4.78810117e-10
  -1.40643719e-10  2.52604604e-11  1.28469457e-11  9.65527658e-11
   1.13020704e-11 -2.86992929e-11 -2.06124007e-11  5.22777377e-11
  -3.76486776e-11 -3.31326633e-12 -1.46441748e-11  8.51869686e-11
   2.70291012e-11]]
constant coefficient= [-11.05636025]
=====================================
MSE for polynomial degree 3: 20560548.002745155
MSE for polynomial degree 8: 1647352.0698174376
Degree 8 polynomial has lower MSE and may be a better fit.
```

[30]:

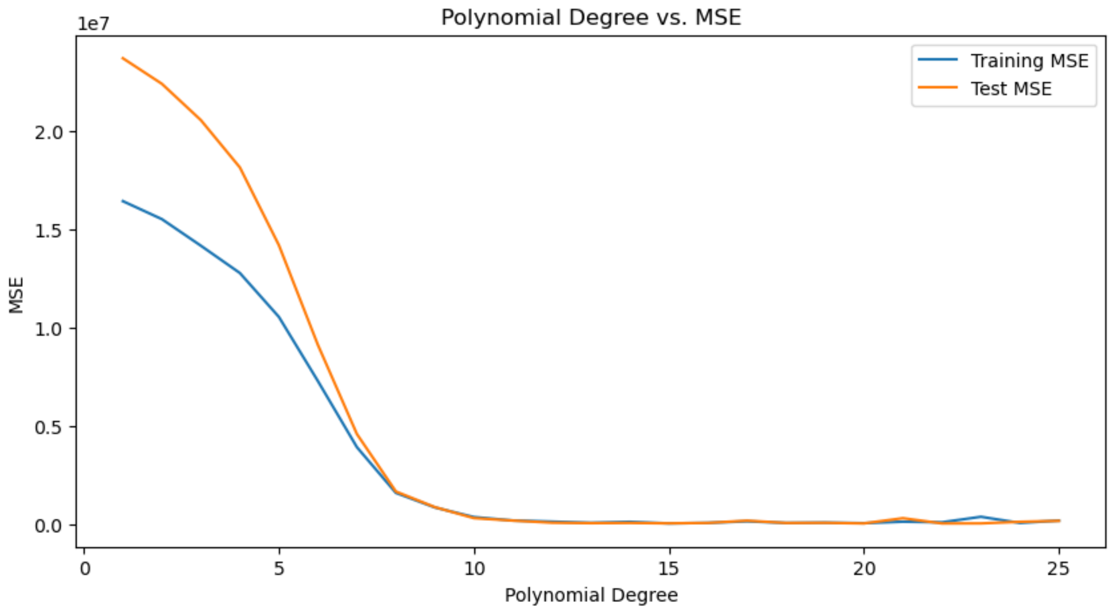
```
# Build an algorithm that finds us the best order for the training set (lowest␣
 ↪MSE), then the test set, using the lowest mse
#  Hint- use a for loop of degrees. (be aware as the orders get higher this␣
 ↪might really slow down, you don't have
# to let it run for hours though! Not for this assignment...) append MSE scores␣
 ↪for train and test. (you'll want these
# for plotting.) return the best order for both train and test.
# Hint 2: use from sklearn.model_selection import cross_val_score and
# have a line in your code like this: scores =␣
↪cross_val_score(polynomial_regressor, X_train_poly, y_train, cv=5)
# you could also use scoring =‘neg_mean_squared_error’ (from metrics.
↪mean_squared_error) but you don't have to,
# scoring will return the 'highest score'- which will pair with the lowest MSE␣
↪for a particular order.
# which is why MSE is often made negative.
```

[31]:
```
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.pipeline import make_pipeline

def find_best_polynomial_order(X_train, X_test, y_train, y_test, max_degree=10):
    best_degree_train = 0
    best_degree_test = 0
    lowest_mse_train = float('inf')
    lowest_mse_test = float('inf')
    mse_scores_train = []
    mse_scores_test = []

    for degree in range(1, max_degree + 1):
        # Creating a polynomial regression model for the current degree
        polynomial_regressor = make_pipeline(PolynomialFeatures(degree),␣
↪LinearRegression())

        # Evaluating the model using cross-validation on the training set
        scores = cross_val_score(polynomial_regressor, X_train, y_train, cv=5,␣
↪scoring='neg_mean_squared_error')
        mse_score_train = -scores.mean()  # Taking the negative to get positive␣
↪MSE values
        mse_scores_train.append(mse_score_train)

        if mse_score_train < lowest_mse_train:
            lowest_mse_train = mse_score_train
            best_degree_train = degree

        # Fitting the model to the training data and evaluating on the test set
        polynomial_regressor.fit(X_train, y_train)
        y_pred = polynomial_regressor.predict(X_test)
        mse_score_test = mean_squared_error(y_test, y_pred)
        mse_scores_test.append(mse_score_test)

        if mse_score_test < lowest_mse_test:
            lowest_mse_test = mse_score_test
            best_degree_test = degree
    return best_degree_train, best_degree_test, mse_scores_train,␣
↪mse_scores_test
```

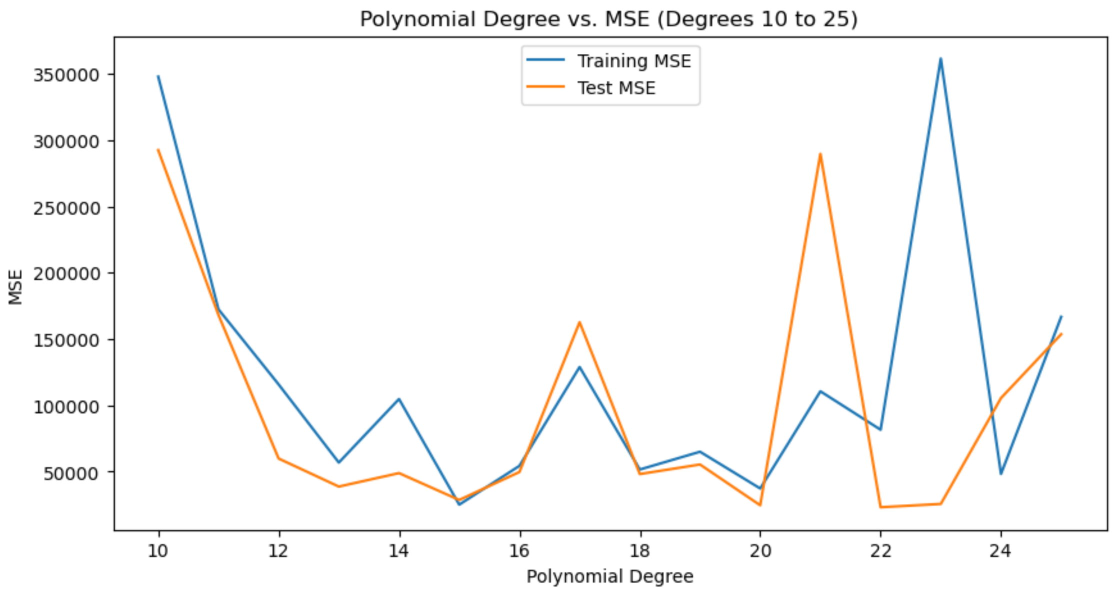
1. What was the best order (and lowest MSE) for the test set?
2. What was the best order (and lowest MSE) for the training set? 3. plot training and test errors (MSEs) as a function of order.

[32]:
```
# Test
max_degree = 25  # Beware computational cost of increasing this number
best_degree_train, best_degree_test, mse_scores_train, mse_scores_test =␣
 ↪find_best_polynomial_order(X_train_1, X_test_1, y_train, y_test, max_degree)

print(f"Best polynomial degree for the training set: {best_degree_train}")
print(f"Best polynomial degree for the test set: {best_degree_test}")

# Plotting the MSE scores for both train and test sets across different degrees

plt.figure(figsize=(10, 5))
plt.plot(range(1, max_degree + 1), mse_scores_train, label='Training MSE')
plt.plot(range(1, max_degree + 1), mse_scores_test, label='Test MSE')
plt.xlabel('Polynomial Degree')
plt.ylabel('MSE')
plt.legend()
plt.title('Polynomial Degree vs. MSE')
plt.show()
```

```
Best polynomial degree for the training set: 15
Best polynomial degree for the test set: 22
```

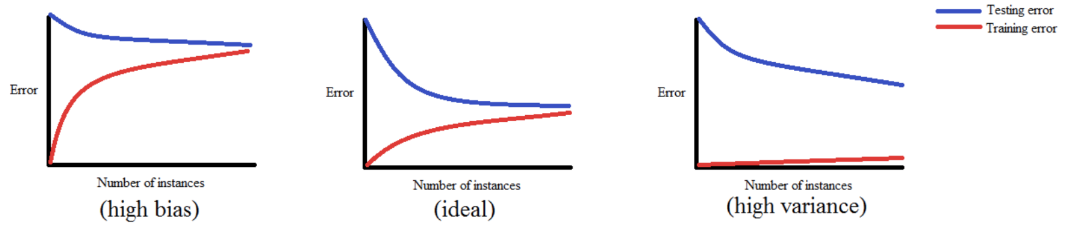


[33]:
```
# Adjusting the plot to focus on degrees 10 to 25
start_degree = 10  # Starting degree for the focused plot

plt.figure(figsize=(10, 5))
# Adjust the ranges for plotting by subtracting start_degree - 1 from the␣
↪indexes
# This is because Python uses zero-based indexing
plt.plot(range(start_degree, max_degree + 1), mse_scores_train[start_degree - 1:
↪], label='Training MSE')
plt.plot(range(start_degree, max_degree + 1), mse_scores_test[start_degree - 1:
↪], label='Test MSE')
plt.xlabel('Polynomial Degree')
plt.ylabel('MSE')
plt.legend()
plt.title('Polynomial Degree vs. MSE (Degrees 10 to 25)')
plt.show()
```



6 7. Present Solution

Hopefully you found in both train and test low MSEs that began to rise again, so you think these might be appropriate orders for your polynomial fit. Cross-validation helps us to find the best fit. Another way is to look at the learning curves of train vs. test (or validation set).



High bias is underfitting- notice the first image on the left the error goes up and stays up. For the image on the right with high variance, the model is overfitting. Notice the error is lower on the training data, but there is a gap between the curves, meaning it performs better on the training data than on the validation data.

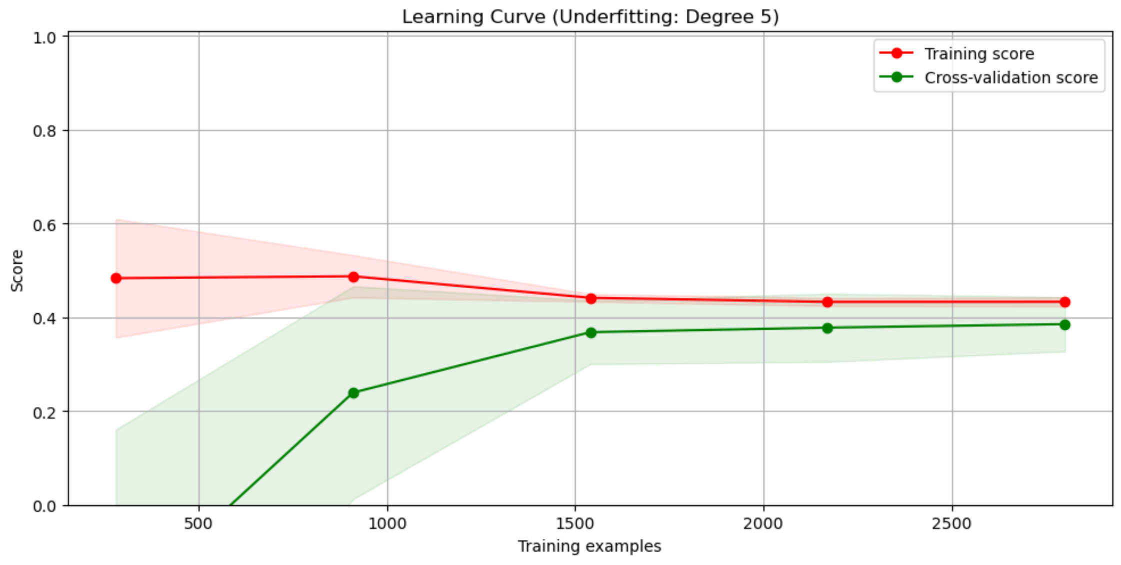
6.1 Question 9

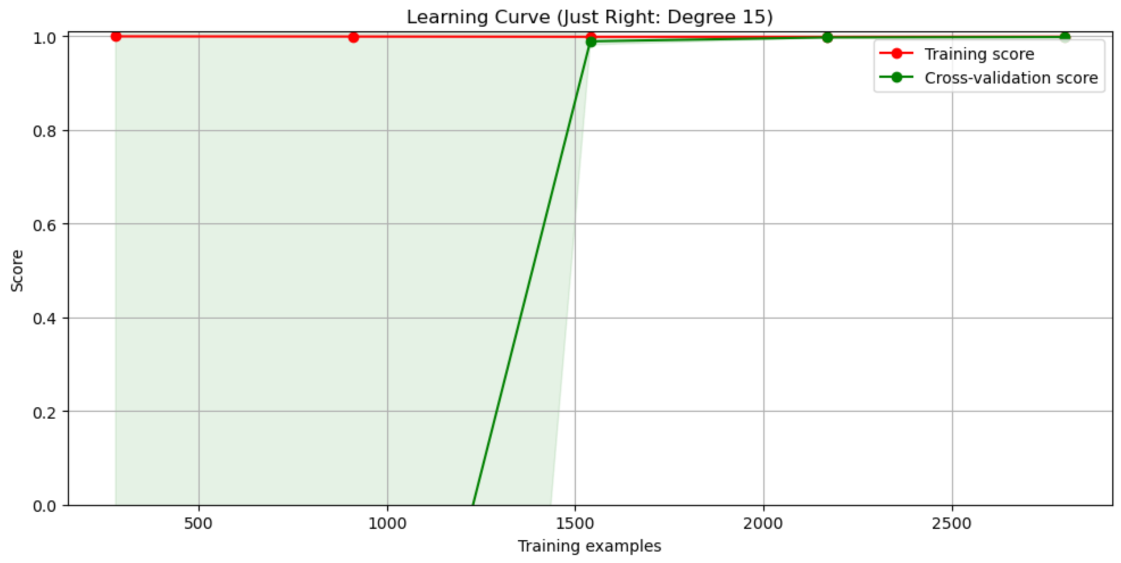
(3 pts)

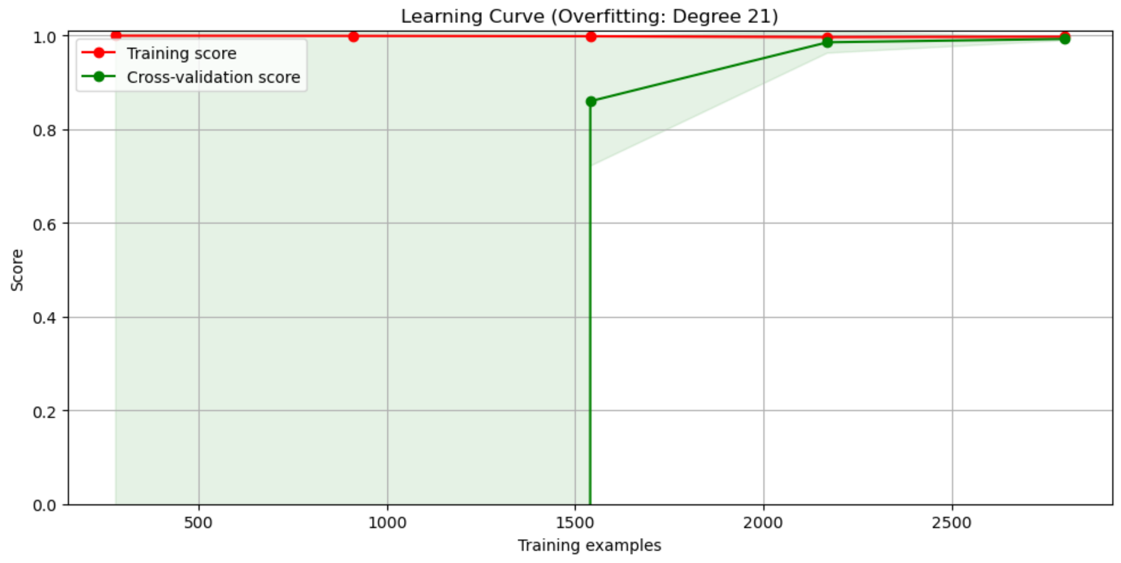
For your final task, using your training set only, and 'from sklearn.model_selection import learning_curve' you will be making 3 plots. First, choose orders where you think the model is underfitting, just right, and overfitting. Next, you'll be using learning_curves (from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.learning_curve.html) A learning curve shows how error changes as the training set size increases. It turns out learning_curves does the work for us of separating out a validation set using the cv parameter. So for ease, just use your training set.

1. Plot a learning curve for an underfitting polynomial fit of a particular order.
2. Plot a learning curve for the best order polynomial fit you found for your training set.
3. Plot a learning curve of an order higher than your best fit to show overfitting.

[35]:
```python
from sklearn.model_selection import learning_curve

def plot_learning_curve(estimator, title, X, y, axes=None, ylim=None, cv=None,
                        n_jobs=None, train_sizes=np.linspace(.1, 1.0, 5)):

    if axes is None:
        _, axes = plt.subplots(1, 1, figsize=(10, 6))

    axes.set_title(title)
    if ylim is not None:
        axes.set_ylim(*ylim)
    axes.set_xlabel("Training examples")
    axes.set_ylabel("Score")

    train_sizes, train_scores, test_scores, fit_times, _ = learning_curve(
        estimator, X, y, cv=cv, n_jobs=n_jobs, train_sizes=train_sizes,
        return_times=True)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)

    # Plot learning curve
    axes.grid()
    axes.fill_between(train_sizes, train_scores_mean - train_scores_std,
                      train_scores_mean + train_scores_std, alpha=0.1,
                      color="r")
    axes.fill_between(train_sizes, test_scores_mean - test_scores_std,
                      test_scores_mean + test_scores_std, alpha=0.1,
                      color="g")
    axes.plot(train_sizes, train_scores_mean, 'o-', color="r",
              label="Training score")
    axes.plot(train_sizes, test_scores_mean, 'o-', color="g",
              label="Cross-validation score")
    axes.legend(loc="best")

    return plt

fig, axes = plt.subplots(3, 1, figsize=(10, 15))

# Plot for underfitting model
plot_learning_curve(make_pipeline(PolynomialFeatures(5), LinearRegression()),␣
 ↪"Learning Curve (Underfitting: Degree 5)", X_train_1, y_train, axes=axes[0],␣
 ↪ylim=(0.0, 1.01), cv=5, n_jobs=4)

# Plot for optimal fitting model
plot_learning_curve(make_pipeline(PolynomialFeatures(15), LinearRegression()),␣
 ↪"Learning Curve (Just Right: Degree 15)", X_train_1, y_train, axes=axes[1],␣
 ↪ylim=(0.0, 1.01), cv=5, n_jobs=4)

# Plot for overfitting model
plot_learning_curve(make_pipeline(PolynomialFeatures(21), LinearRegression()),␣
 ↪"Learning Curve (Overfitting: Degree 21)", X_train_1, y_train, axes=axes[2],␣
 ↪ylim=(0.0, 1.01), cv=5, n_jobs=4)

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


Learning Curve (Underfitting: Degree 5)
Training score
Cross-validation score
Score
1.0
0.8
0.6
0.4
0.2
0.0
500
1000
1500
2000
2500
Training examples


Learning Curve (Just Right: Degree 15)
Training score
Cross-validation score
Score
1.0
0.8
0.6
0.4
0.2
0.0
500
1000
1500
2000
2500
Training examples


Learning Curve (Overfitting: Degree 21)
Training score
Cross-validation score
Score
1.0
0.8
0.6
0.4
0.2
0.0
500
1000
1500
2000
2500
Training examples

7 8. Launch, Monitor and Maintain

Nothing to do at phase 8– but think about what you would do if you were updating your Covid model with new data. How often would you retrain? What if different periods of time looked like different polynomials or looked linear?

7.1 Great Job!

In an ideal scenario to get the most accurate and up to date information we might retrain daily with each update of the data. The effect of a single day on the model should be small however, and increasingly of less impact as mor days accumulate since each day then represents an increasingly small fraction of the dataset. Likley constant updating is unreasonable due cost of compute time. If this were a real world scenario we would use the entire dataset, slowing the training somewhat and making retraining more costly. The total amount of data also increases over time as the dataset updates. The ammount of time that is worthwhile also depends on the importance of the project and who is relying on the model's predictions.

If I were working on this project long term, I might experiment with days, weeks, biweekly, and mothly updates etc. To see which time scales resulted in a significant update to the model. I'd then try to implement an automatic pipeline to aggregate the data, reformulate the training set, and try to run automatic optimization similar to what we did above.

```
[ ]:
```