```
In [ ]: %matplotlib inline
```

SYDE 522 Assignment 1 - Calvin Tran - 20826392

Perceptrons and Regression

Due: Monday Sept 25 at 11:59pm

As with all the assignments in this course, this assignment is structured as a Jupyter Notebook and uses Python. If you do not have Python and Jupyter Notebook installed, the easiest method is to download and install Anaconda https://www.anaconda.com/download. There is a quick tutorial for running Jupyter Notebook from within Anacoda at https://docs.anaconda.com/free/anaconda/getting-started/hello-world/#python-exercise-jupyter under "Run Python in a Jupyter Notebook"

Implement your assignment directly in the Jupyter notebook and submit your resulting Jupyter Notebook file using Learn.

While you are encouraged to talk about the assignment with your classmates, you must write and submit your own assignment. Directly copying someone else's assignment and changing a few small things here and there does not count as writing your own assignment.

Make sure to label the axes on all of your graphs.

Question 1: Implementing a Perceptron

The following code generates the same data that was used to demonstrate the Perceptron in class:

```
In [ ]: import sklearn.datasets
        data_x, data_y = sklearn.datasets.make_blobs(centers=[[-2, -2], [2, 2]],
                                                     cluster_std=[0.3, 1.5],
                                                     random_state=0,
                                                     n_samples=200,
                                                     n_features=2)
```

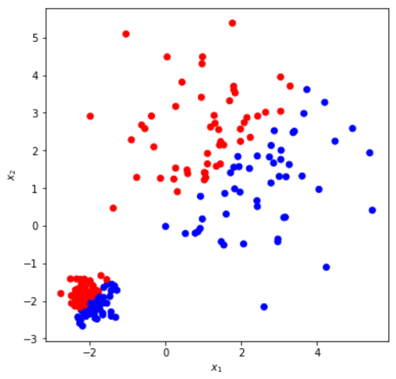
This produces two arrays, `data_x` which contains the input data (200 rows, each of which has 2 values $x_1$ and $x_2$), and `data_y` which contains the desired output data (either a 1 or a 0).

Implement a Perceptron to learn a classifier on this data. It should learn three values: ω_1, ω_2, and θ (of course you can use whatever variable names you like to encode them). You can treat θ separately, or you can consider it an extra weight variable ω_0 and have an extra input that is always 1. Implement this Perceptron yourself, rather than using the `sklearn.linear_model.Perceptron` implementation that we will use in Question 2.

Initialize the weights to $\omega_1 = 1; \omega_2 = -1; \theta = 0$.

a) [1 mark] Before doing any training, plot the data as a scatterplot and colour the dots such that the data points for which the model outputs a 1 are blue and the ones for which the model outputs a 0 are red. This can be done with the following code, if `y` is the list of outputs from your model. Compute how accurate the model is (i.e. what percentage of the time the model outputs the correct value) and report that number.

```
In [ ]: import matplotlib.pyplot as plt
        import numpy as np

        w = [1, -1, 0]
        y = []

        # Create a new array with '1' appended at the end to account for bias
        data_x_ones = np.c_[data_x, np.ones((data_x.shape[0]))]

        correct = 0

        for i in range(0,len(data_x_ones)):
            dp = np.dot(data_x_ones[i],w)
            y.append(1 if dp > 0 else 0)
            if dp > 0 and data_y[i] or dp < 0 and not data_y[i]:
                correct += 1

        plt.figure(figsize=(6,6))
        plt.scatter(data_x[:,0], data_x[:,1], c=np.where(y, 'blue', 'red'))
        plt.xlabel('$x_1$')
        plt.ylabel('$x_2$')
        plt.show()

        percentage = correct / len(data_x_ones) * 100
        print(f"The accuracy of the current model is {percentage}%")
```



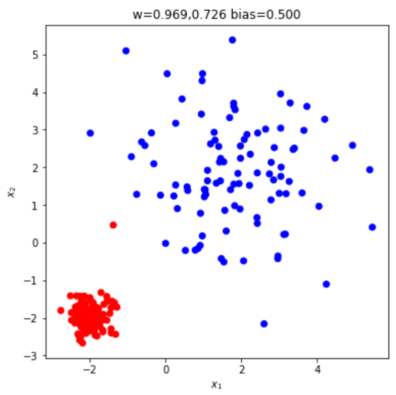
```
The accuracy of the current model is 51.5%
```

b) [1 mark] Train the model by going through each of the 200 elements in the data set in order once. For each input, check if the output is correct. If it is not correct, apply the Perceptron Learning Rule. Use a learning rate of 0.1.

Now produce the same plot as in part a), but with your trained weights. How accurate is the model now? Report the ω and θ values.

```
In [ ]: learning_rate = 0.1
        w_b = w
        # Training
        for i in range(0,len(data_x_ones)):
            res = 1 if np.dot(data_x_ones[i],w_b) > 0 else 0
            w_b += learning_rate*data_x_ones[i]*(data_y[i] - res)
```

```
In [ ]: # Checking
        correct = 0
        y_one_iter = []
        for i in range(0,len(data_x_ones)):
            dp = np.dot(data_x_ones[i],w_b)
            y_one_iter.append(1 if dp > 0 else 0)
            if dp > 0 and data_y[i] or dp < 0 and not data_y[i]:
                correct += 1

        plt.figure(figsize=(6,6))
        plt.scatter(data_x_ones[:,0], data_x_ones[:,1], c=np.where(y_one_iter, 'blue', 'red'))

        plt.xlabel('$x_1$')
        plt.ylabel('$x_2$')
        plt.title(f'w={w_b[0]:.3f},{w_b[1]:.3f} bias={w_b[2]:.3f}')
        plt.plot()
        plt.show()

        percentage = correct / len(data_x_ones) * 100
        print(f"The accuracy after one pass of the current model is {percentage}%, w_1 is {w_b[0]}, w_2 is {w_b[1]} and the bias term is {w_b[2]}")
```

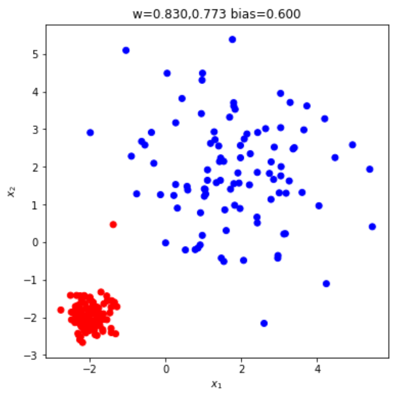


```
The accuracy after one pass of the current model is 99.5%, w_1 is 0.9685061036503375, w_2 is 0.7262395916938614 and the bias term is 0.5
```

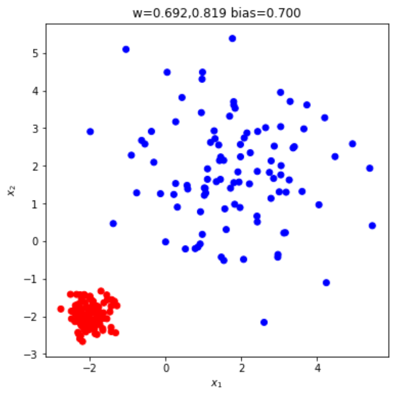
c) [1 mark] Repeat the training in part b) enough times that the model is perfect (in that it correctly classifies all the inputs). How many repetitions does this take? Produce the same plots as in part a) and b), but with your new weights. Report the ω and θ values.

```
In [ ]: # Training until perfection
        iteration_num = 1
        w_c = w_b
        while correct < 200:
            y_new = []
            for i in range(0,len(data_x_ones)):
                res = 1 if np.dot(data_x_ones[i],w_c) > 0 else 0
                w_c += learning_rate*data_x_ones[i]*(data_y[i] - res)

            correct = 0
            for i in range(0,len(data_x_ones)):
                dp = np.dot(data_x_ones[i],w_c)
                y_new.append(1 if dp > 0 else 0)
                if dp > 0 and data_y[i] or dp < 0 and not data_y[i]:
                    correct += 1

            iteration_num += 1

            percentage = correct / len(data_x_ones) * 100
            print(f"The accuracy of the current model is {percentage}%, w_1 is {w_c[0]}, w_2 is {w_c[1]} and the bias term is {w_c[2]}")
            print(f"The number of iterations is took to reach this accuracy is {iteration_num}")
            plt.figure(figsize=(6,6))
            plt.scatter(data_x_ones[:,0], data_x_ones[:,1], c=np.where(y_new, 'blue', 'red'))

            plt.xlabel('$x_1$')
            plt.ylabel('$x_2$')
            plt.title(f'w={w_c[0]:.3f},{w_c[1]:.3f} bias={w_c[2]:.3f}')
            plt.plot()
            plt.show()
```

```
The accuracy of the current model is 99.5%, w_1 is 0.8301714692400091, w_2 is 0.772863565148521 and the bias term is 0.6
The number of iterations is took to reach this accuracy is 2
```



```
The accuracy of the current model is 100.0%, w_1 is 0.6918368348296806, w_2 is 0.8194875386031805 and the bias term is 0.7
The number of iterations is took to reach this accuracy is 3
```

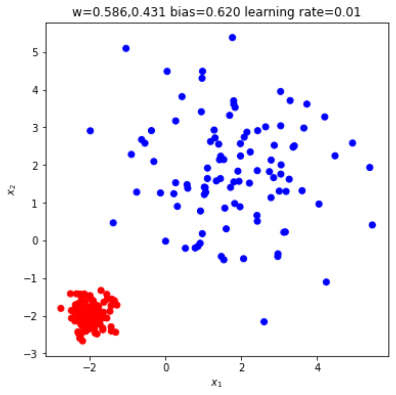


d) [1 mark] Create a new Perceptron identical to the above one, but with a learning rate of 1.0. Train this model until it reaches 100% accuracy. How many repetitions does this take? Produce the same plot again, but with your new weights. Report the ω and θ values.

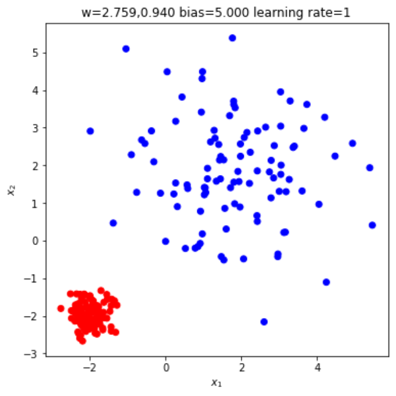
Now do the same thing with a learning rate of 0.01, and then again with a learning rate of 100.

In [ ]:
```
learning_rates = [0.01, 1, 100]
for rate in learning_rates:
    w_d = [1, -1, 0]
    y_new = []
    correct = 0
    iteration_num = 0
    while correct < 200:
        y_new = []
        for i in range(0,len(data_x_ones)):
            res = 1 if np.dot(data_x_ones[i],w_d) > 0 else 0
            w_d += rate*data_x_ones[i]*(data_y[i] - res)

        correct = 0
        for i in range(0,len(data_x_ones)):
            dp = np.dot(data_x_ones[i],w_d)
            y_new.append(1 if dp > 0 else 0)
            if dp > 0 and data_y[i] or dp < 0 and not data_y[i]:
                correct += 1

        iteration_num += 1

    percentage = correct / len(data_x_ones) * 100
    print(f"The accuracy of the current model with a learning rate of {rate} is {percentage}%, w_1 is {w_d[0]}, w_2 is {w_d[1]} and the bias term is {w_d[2]}")
    print(f"The number of iterations is took to reach this accuracy is {iteration_num}")
    plt.figure(figsize=(6,6))
    plt.scatter(data_x_ones[:,0], data_x_ones[:,1], c=np.where(y_new, 'blue', 'red'))

    plt.xlabel('$x_1$')
    plt.ylabel('$x_2$')
    plt.title(f'w={w_d[0]:.3f},{w_d[1]:.3f} bias={w_d[2]:.3f} learning rate={rate}')
    plt.plot()
    plt.show()
```

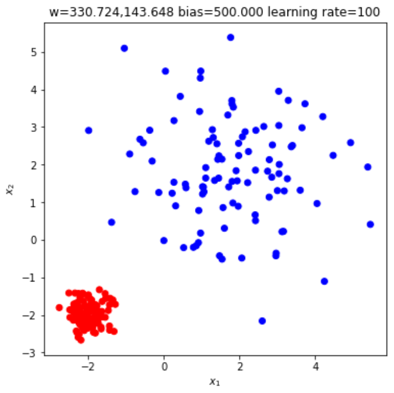
```
The accuracy of the current model with a learning rate of 0.01 is 100.0%, w_1 is 0.5857079089907371, w_2 is 0.4314016565704606 and the bias term is 0.6200000000000003
The number of iterations is took to reach this accuracy is 22
```



```
The accuracy of the current model with a learning rate of 1 is 100.0%, w_1 is 2.759373897418298, w_2 is 0.9404139300356067 and the bias term is 5.0
The number of iterations is took to reach this accuracy is 3
```



```
The accuracy of the current model with a learning rate of 100 is 100.0%, w_1 is 330.7242150951744, w_2 is 143.6484479236975 and the bias term is 500.0
The number of iterations is took to reach this accuracy is 3
```



Question 2:

We will now try a more complex dataset, and use a pre-written implementation of the Perceptron. The `sklearn` Python library https://scikit-learn.org/ has a large collection of machine learning algorithms, and comes with a variety of datasets. It comes pre-installed with Anaconda or can be installed with `pip install scikit-learn` .

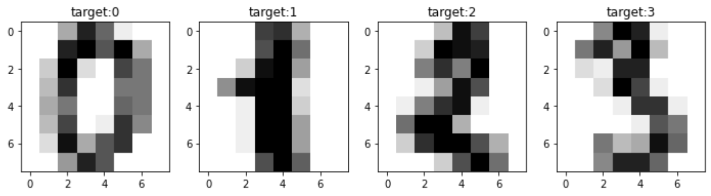
The dataset we will use is the UCI ML hand-written digits dataset https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

It is available with the following command:

```
In [ ]: import sklearn.datasets
        digits = sklearn.datasets.load_digits()
```

The inputs are 64 values, representing an 8x8 input image that is a low-resolution handwritten digit. You can access this data as `digits.data` . The correct label (i.e. the desired output) for each digit is accessed with `digits.target` . Here are the first four input-output pairs:

```
In [ ]: plt.figure(figsize=(12,3))
        for i in range(4):
            plt.subplot(1, 4, i+1)
            plt.imshow(digits.data[i].reshape(8,8), cmap='gray_r')
            plt.title(f'target:{digits.target[i]}')
        plt.show()
```



You can create a Perceptron using the following command, where `eta0` is the learning rate:

```
In [ ]: import sklearn.linear_model
        perceptron = sklearn.linear_model.Perceptron(eta0=1.0)
```

If you have an input data `X` and target output data `Y` , you can train the perceptron as follows:

```
In [ ]: perceptron.fit(X, Y)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [41], in <module>
----> 1 perceptron.fit(X, Y)

NameError: name 'X' is not defined
```

Once the Perceptron has been trained, you can see what outputs it actually generates given input `X` as follows:

```
In [ ]: perceptron.predict(X)
```

In addition, `sklearn` provides a useful tool for separating your data into training and test data.

```
In [ ]: import sklearn.model_selection
        X_train, X_test, Y_train, Y_test = sklearn.model_selection.train_test_split(
            digits.data, digits.target, test_size=0.2, shuffle=True,
        )
```

This splits all your data in `digits.data` into two parts, `X_train` and `X_test` (with the corresponding outputs in `Y_train` and `Y_test`). The setting `test_size=0.2` means that the test set will be 20% of the data, and `shuffle=True` means it will randomly choose that 20%.

Note that you can also use the same function to split your training data into training data and validation data.

a) [1 mark] Let's start with only considering the digit data for 0's and 1's. We can extract just that data with `X = digits.data[(digits.target == 0) | (digits.target == 1)]` and `Y = digits.target[(digits.target == 0) | (digits.target == 1)]` . Split the data into 80% training and 20% testing. Create a Perceptron with a learning rate of 1.0 and train it on your training data. Report the accuracy (i.e. how often the model gives the correct output) on your testing data.

```
In [ ]: import sklearn.model_selection
        import sklearn.linear_model
        import sklearn.datasets
        import sklearn.metrics

        digits = sklearn.datasets.load_digits()

        X = digits.data[(digits.target == 0) | (digits.target == 1)]
        Y = digits.target[(digits.target == 0) | (digits.target == 1)]

        X_train, X_test, Y_train, Y_test = sklearn.model_selection.train_test_split(
    X, Y, test_size=0.2, shuffle=True,
)

perceptron = sklearn.linear_model.Perceptron(eta0=1.0)
perceptron.fit(X_train, Y_train)
results = perceptron.predict(X_test)

correct = 0

accuracy = sklearn.metrics.accuracy_score(Y_test, results) * 100

print(f"The accuracy of the Perceptron with a learning rate of 1.0 is {accuracy}%.")
```

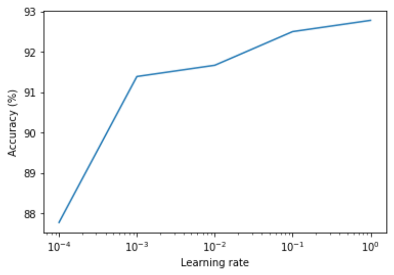
```
The accuracy of the Perceptron with a learning rate of 1.0 is 100.0%.
```

b) [1 mark] Repeat the above, but with the entire data set (i.e. all 10 digits). Report the accuracy. How does the accuracy change as you adjust the learning rate? Make a plot that shows this.

In [ ]:

```
digits = sklearn.datasets.load_digits()

X_train, X_test, Y_train, Y_test = sklearn.model_selection.train_test_split(
    digits.data, digits.target, test_size=0.2, shuffle=True,
)

rates = [0.0001, 0.001, 0.01, 0.1, 1]
accuracies = []
for i in range(len(rates)):
    sum_acc = 0
    # Accuracy is averaged over 100 runs, as accuracy varies greatly between each training session
    for j in range(100):
        perceptron = sklearn.linear_model.Perceptron(eta0=rates[i])
        perceptron.fit(X_train, Y_train)
        results = perceptron.predict(X_test)

        correct = 0

        sum_acc += sklearn.metrics.accuracy_score(Y_test, results) * 100
    # Get average accuracy
    accuracy = sum_acc / 100
    accuracies.append(accuracy)
    print(f"The accuracy of the Perceptron with a learning rate of {rates[i]} is {accuracy}%.")

plt.xscale("log")
plt.xlabel('Learning rate')
plt.ylabel('Accuracy (%)')
plt.plot(rates, accuracies)
plt.show()
```

```
The accuracy of the Perceptron with a learning rate of 0.0001 is 87.77777777777759%.
The accuracy of the Perceptron with a learning rate of 0.001 is 91.38888888888874%.
The accuracy of the Perceptron with a learning rate of 0.01 is 91.66666666666671%.
The accuracy of the Perceptron with a learning rate of 0.1 is 92.5%.
The accuracy of the Perceptron with a learning rate of 1 is 92.77777777777757%.
```



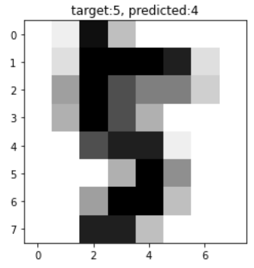
According to this graph, it seems like increasing the learning rate leads to an increase in accuracy. However, it should be noted that the graph changes drastically between runs, so this is not a conclusive statement.

c) [1 mark] What mistakes does the model make? What digits does it tend to mistake for other digits? Use the `plt.imshow(digits.data[i].reshape(8,8), cmap='gray_r')` command given above to plot some of the digits that it gets wrong. Why do you think it has problems with these digits?

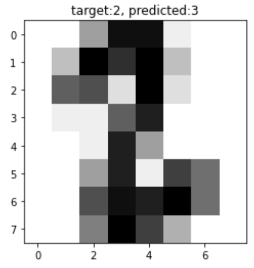
In [ ]:

```
digits = sklearn.datasets.load_digits()

X_train, X_test, Y_train, Y_test = sklearn.model_selection.train_test_split(
    digits.data, digits.target, test_size=0.2, shuffle=True,
)

# Create a perceptron using a learning rate of 0.1
perceptron_c = sklearn.linear_model.Perceptron(eta0=0.1)
perceptron_c.fit(X_train, Y_train)
results = perceptron_c.predict(X_test)

wrong = np.zeros(10)

for i in range(len(results)):
    if results[i] != Y_test[i]:
        # Store a count of all the target numbers that are incorrectly predicted, using the target number as the index of the count of incorrect predictions for that number
        wrong[Y_test[i]] += 1

        # Graph incorrect numbers
        plt.imshow(X_test[i].reshape(8,8), cmap='gray_r')
        plt.title(f'target:{Y_test[i]}, predicted:{results[i]}')
        plt.show()

plt.bar(np.arange(10), wrong)
plt.title(f'Number of incorrect predictions per target number')
plt.xlabel('Target number')
plt.ylabel('Number of incorrect predictions')
plt.show()
```

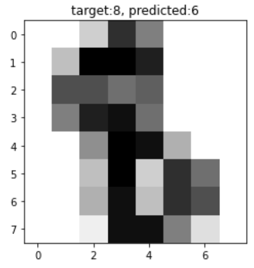
target:5, predicted:4



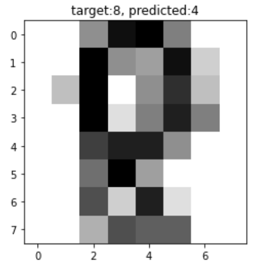
target:2, predicted:3



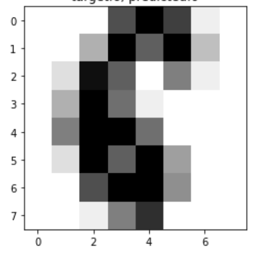
target:8, predicted:6



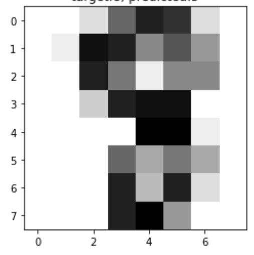
target:8, predicted:4



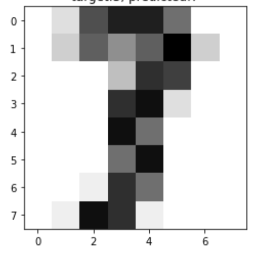
target:6, predicted:0



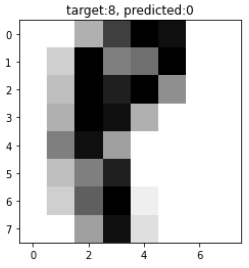
target:8, predicted:5



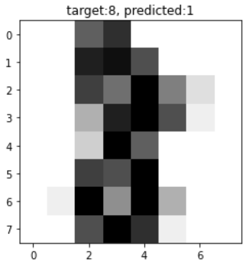
target:3, predicted:7



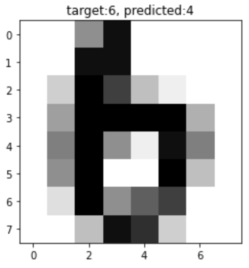
target:8, predicted:0



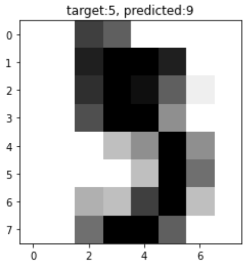
target:8, predicted:1



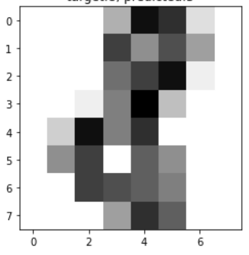
target:6, predicted:4



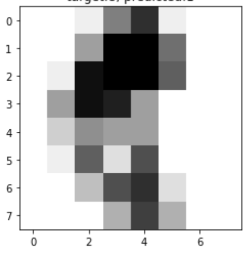
target:5, predicted:9



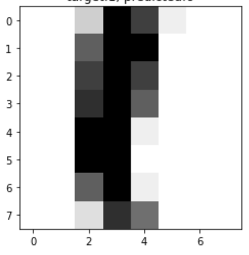
target:8, predicted:3

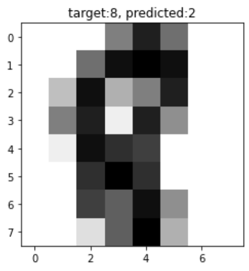


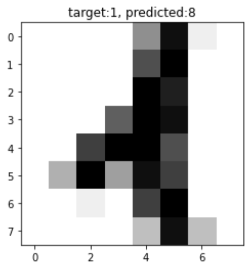
target:8, predicted:1

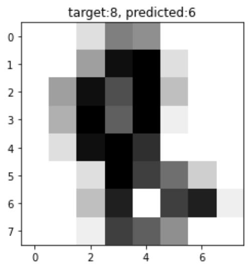


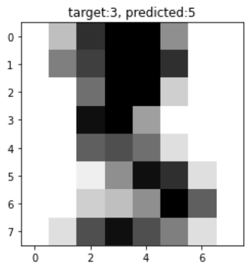
target:1, predicted:6

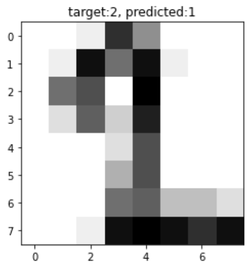


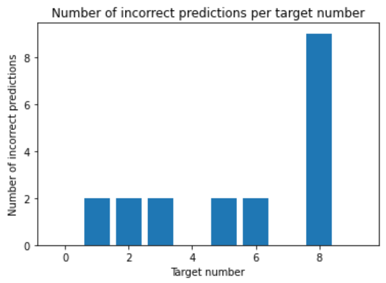












Taking a look at the bar graph above, we notice how the number that gets the most incorrect predictions is 8. Taking a look at some of the plots of the digits, we can notice how the number 8 often gets mistaken for numbers like 0, 2, 3, 5 and 6, all of which look very similar to 8. Since the Perceptron is trained to predict results based on similarity to previous training data, if it sees an image of an 8 that looks very similar to training images of other numbers, it will be more likely to mistake it for those numbers. Interestingly enough, some of the other digits that get incorrectly predicted are 1, 2, 3, 5 and 6; four of these digits are quite similar to the number 8.

Question 3:

The following code generates the same data that was used to demonstrate curve fitting in class. `train_x` and `train_y` are the 10 data points we use for doing the curve fitting, and `test_x` and `test_y` are the data we used to test how well the fit generalizes.

In [ ]:
```
import numpy as np
rng = np.random.RandomState(seed=0)
train_x = np.linspace(0, 1, 10)
train_y = np.sin(train_x*2*np.pi) + rng.normal(0,0.1,size=10)
```

```
test_x = np.linspace(0, 1, 500)
test_y = np.sin(test_x*2*np.pi)
```

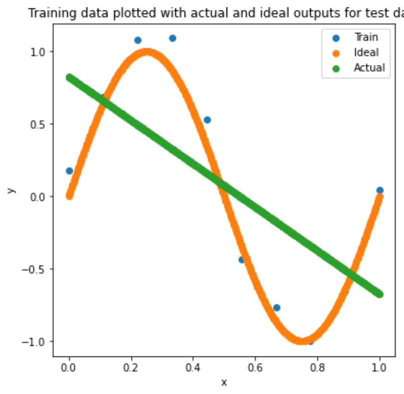
a) [1 mark] Find the weights that best fit this data using linear regression. This should generate two weights: one that is multiplied by the input value and one that is mulitplied by the feature that is constantly a 1. Implement this yourself, rather than using the `sklearn.linear_model.LinearRegression` implementation that we will use in Question 4. To invert the matrix, use `np.linalg.pinv` .

Plot the training data, the ideal testing output, and the actual testing output. Report the weights found by regression. Compute and report the Root Mean Squared Error (`np.sqrt(np.mean((Y-output)**2))` where `Y` is the vector of desired outputs and `output` is the vector of the actual outputs from the model) for both the training data and the testing data.

In [ ]:
```
import numpy as np
rng = np.random.RandomState(seed=0)
train_x = np.linspace(0, 1, 10)
train_y = np.sin(train_x*2*np.pi) + rng.normal(0,0.1,size=10)
test_x = np.linspace(0, 1, 500)
test_y = np.sin(test_x*2*np.pi)

# Augment training data to have a column of ones to account for bias
train_x_ones = np.column_stack((np.power(train_x,0),np.power(train_x,1)))

# Calculate weights
w = np.matmul(np.matmul(np.linalg.pinv(np.matmul(train_x_ones.transpose(),train_x_ones)),train_x_ones.transpose()),train_y)

print(f"The weights calculated are {w[0]} for the x^0 term and {w[1]} for the x^1 term.")

actual_y = w[1]*test_x+w[0]

# Calculate root mean squared error
rmse = np.sqrt(np.mean((test_y-actual_y)**2))
print(f"The root mean squared error is {rmse}.")

plt.figure(figsize=(6,6))
train = plt.scatter(train_x, train_y)
ideal = plt.scatter(test_x, test_y)
actual = plt.scatter(test_x, actual_y)
plt.legend([train, ideal, actual], ["Train", "Ideal", "Actual"])
plt.xlabel("x")
plt.ylabel("y")
plt.title("Training data plotted with actual and ideal outputs for test data")
```

```
The weights calculated are 0.8247443469239097 for the x^0 term and -1.501884059702053 for the x^1 term.
The root mean squared error is 0.4648326295012366.
```

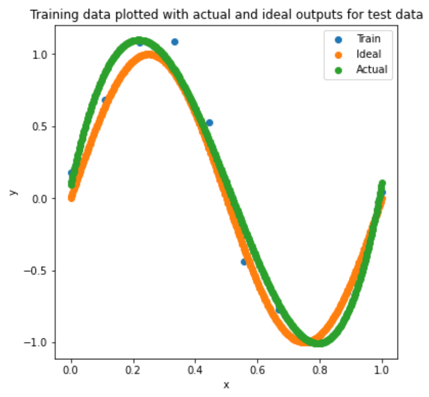
Out[ ]:
```
Text(0.5, 1.0, 'Training data plotted with actual and ideal outputs for test data')
```



b) [1 mark] Repeat part a), but use the first 5 polynomials as features $(x^0, x^1, x^2, x^3, x^4)$. Plot the training data, the ideal testing output, and the actual testing output. Report the weights found by regression. Compute and report the Root Mean Squared Error for both the training data and the testing data. Do not use regularization.

In [ ]:
```
# Augment training data to have first 5 polynomials as features
train_x_poly = np.column_stack((np.power(train_x,0),np.power(train_x,1),np.power(train_x,2),np.power(train_x,3),np.power(train_x,4)))

# Calculate weights
w = np.matmul(np.matmul(np.linalg.pinv(np.matmul(train_x_poly.transpose(),train_x_poly)),train_x_poly.transpose()),train_y)
print(f"The weights for the first 5 polynomials x^0, x^1, x^2, x^3 and x^4 are {w[0]}, {w[1]}, {w[2]}, {w[3]} and {w[4]}, respectively.")

actual_y = w[4]*test_x**4 + w[3]*test_x**3 + w[2]*test_x**2 + w[1]*test_x + w[0]

# Calculate root mean squared error
rmse = np.sqrt(np.mean((test_y-actual_y)**2))
print(f"The root mean squared error is {rmse}.")

plt.figure(figsize=(6,6))
train = plt.scatter(train_x, train_y)
ideal = plt.scatter(test_x, test_y)
actual = plt.scatter(test_x, actual_y)
plt.legend([train, ideal, actual], ["Train", "Ideal", "Actual"])
plt.xlabel("x")
plt.ylabel("y")
plt.title("Training data plotted with actual and ideal outputs for test data")
plt.show()
```

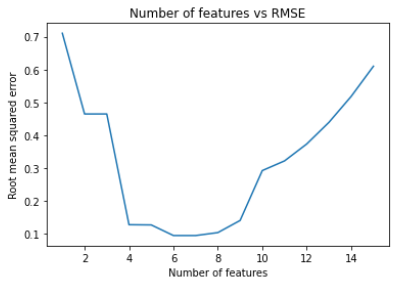
```
The weights for the first 5 polynomials x^0, x^1, x^2, x^3 and x^4 are 0.09121908074989839, 9.976542023061818, -27.720739739349455, 13.542529597397994 and 4.219493904579686, respectivel
y.
The root mean squared error is 0.12757359049537179.
```



c) [1 mark] Vary the number of polynomials you use from 1 up to 15. Compute the Root Mean Squared Error for the training and testing data and plot the results.

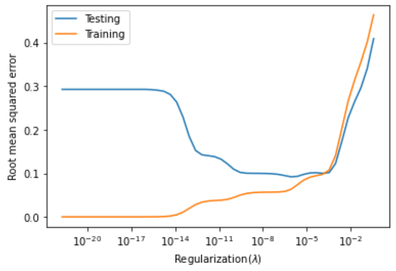
```
In [ ]: rmse_vals = []

for i in range(1,16):
    # Initialize matrix
    train_x_poly = np.zeros((len(train_x), i))
    for j in range(0,i):
        train_x_poly[:, j] = train_x**j

    # Calculate weights
    w = np.matmul(np.matmul(np.linalg.pinv(np.matmul(train_x_poly.transpose(),train_x_poly)),train_x_poly.transpose()),train_y)

    # Create polynomial using weights and determine actual y values using the polynomial
    poly = np.poly1d(np.flip(w))
    actual_y = poly(test_x)

    # Calculate root mean squared error
    rmse = np.sqrt(np.mean((test_y-actual_y)**2))
    rmse_vals.append(rmse)

plt.plot(list(range(1,16)), rmse_vals)
plt.title("Number of features vs RMSE")
plt.xlabel("Number of features")
plt.ylabel("Root mean squared error")
```

```
Out[ ]: Text(0, 0.5, 'Root mean squared error')
```



d) [1 mark] Now intruduce regularization to your model. Set the number of polynomials to 10 and vary the amount of regularization. Use `lambds = np.exp(np.linspace(-50,-1, 50))` to generate the list of 50 different regularization values to try (logarithmically spaces between e^{-50} and e^{-1}). Compute the Root Mean Squared Error for the training and testing data and plot the results. Note that `plt.semilogx` lets you create a plot where the x-axis is on a log scale, like the version of this plot we saw in class.

```
In [ ]: train_x_poly = np.zeros((len(train_x), 10))
for i in range(0,10):
    train_x_poly[:, i] = train_x**i

lambds = np.exp(np.linspace(-50, -1, 50))

rmse_test = []
rmse_train = []

# Calculate weights for each lambda value
for lambd in lambds:
    w = np.matmul(np.matmul(np.linalg.pinv(np.matmul(train_x_poly.transpose(),train_x_poly)+np.identity(10)*lambd),train_x_poly.transpose()),train_y)

    # Create polynomial using weights and determine actual y values using the polynomial
    poly = np.poly1d(np.flip(w))
    actual_y_test, actual_y_train = poly(test_x), poly(train_x)

    # Calculate root mean squared error
    rmse_test.append(np.sqrt(np.mean((test_y-actual_y_test)**2)))
    rmse_train.append(np.sqrt(np.mean((train_y-actual_y_train)**2)))

plt.semilogx(lambds, rmse_test)
plt.semilogx(lambds, rmse_train)
plt.xlabel("Regularization($\lambda$)")
plt.ylabel("Root mean squared error")
plt.legend(["Testing", "Training"])
plt.show()
```



Question 4:

We will now use the regression tool built in to `sklearn`. We create it as follows. Note that it is called `Ridge` due to how regularization is implemented: we add a value onto the diagonal of the matrix being inverted. You can think of this as adding a diagonal ridge to whatever data is in the matrix. For this reason, this is often called "ridge regression". The parameter `alpha` sets the amount of regression (it is the same as what we called $\lambda$ in class).

In [ ]:
```
import sklearn.linear_model
reg = sklearn.linear_model.Ridge(alpha=0.000001)
```

We use the regression system using exactly the same functions as the Perceptron. Here we train it using `X` and `Y`, and then determine what the outputs are given `X`.

In [ ]:
```
reg.fit(X, Y)
output = reg.predict(X)
```

For data, we are going to use the Diabetes dataset from https://www4.stat.ncsu.edu/~boos/var.select/diabetes.html which is also built in to `sklearn`. You can load this data set using

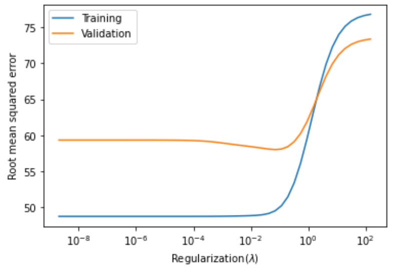
In [ ]:
```
diabetes = sklearn.datasets.load_diabetes()
```

As with the digits dataset, you can access the `X` values with `diabetes.data` and the `Y` values with `diabetes.target`. See https://scikit-learn.org/stable/datasets/toy_dataset.html#diabetes-dataset for an explanation of what the different data values mean.

a) [1 mark] Split the data evenly into three parts: 1/3rd training, 1/3rd validation, and 1/3rd testing. This will involve calling `sklearn.model_selection.train_test_split` twice. Train the model using various different amounts of regularization from e^{-20} to e^{5} (`lambds = np.exp(np.linspace(-20,5,50))`). Compute the Root Mean Squared Error on the training and validation datasets and plot how this error changes for different amounts of regularization. Using these results, pick a good value for regularization and then apply this to your testing data. Report the Root Mean Squared Error for the testing data.

In [ ]:
```
import sklearn.linear_model

# Split dataset
diabetes = sklearn.datasets.load_diabetes()

X_train, X_remain, Y_train, Y_remain = sklearn.model_selection.train_test_split(
    diabetes.data, diabetes.target, test_size=0.667, shuffle=True,
)
X_validation, X_test, Y_validation, Y_test = sklearn.model_selection.train_test_split(
    X_remain, Y_remain, test_size=0.5, shuffle=True,
)

lambds = np.exp(np.linspace(-20,5,50))
rmse_train = []
rmse_validation = []

for lambd in lambds:
    reg = sklearn.linear_model.Ridge(alpha=lambd)
    # Train the data using training dataset
    reg.fit(X_train, Y_train)
    # Determine the outputs of the model for the training and validation datasets
    output_train = reg.predict(X_train)
    output_validation = reg.predict(X_validation)
    # Compute root mean squared error
    rmse_train.append(np.sqrt(np.mean((Y_train-output_train)**2)))
    rmse_validation.append(np.sqrt(np.mean((Y_validation-output_validation)**2)))

# Plot the RMSE for training and validation datasets
plt.semilogx(lambds, rmse_train)
plt.semilogx(lambds, rmse_validation)
plt.xlabel("Regularization($\lambda$)")
plt.ylabel("Root mean squared error")
plt.legend(["Training", "Validation"])
plt.plot()

ideal_lambda = lambds[rmse_validation.index(min(rmse_validation))]
min_rmse = min(rmse_validation)
print(f"The minimum root mean squared error is {min_rmse}, which occurs at a lambda of {ideal_lambda}")
```

```
The minimum root mean squared error is 58.03715866404364, which occurs at a lambda of 0.07043526453886602
```



In [ ]:
```
# Use ideal lambda and apply it to test dataset.
reg = sklearn.linear_model.Ridge(alpha=ideal_lambda)
# Train the data using training dataset
reg.fit(X_train, Y_train)
# Determine the outputs of the model for the training and validation datasets
output_test = reg.predict(X_test)
# Compute root mean squared error
rmse_test_val = np.sqrt(np.mean((Y_test-output_test)**2))
```

```
print(f"The root mean squared error using a lambda of {ideal_lambda} is {rmse_test_val}.")
```

```
The root mean squared error using a lambda of 0.07043526453886602 is 57.14720566848628.
```

b) [1 mark] How consistent is this result? That is, if you redo part a) but with a different randomly chosen split in the data, do you get the same results? What overall pattern do you see? Do the results show signs of overfitting? Would you expect overfitting here? Why or why not?

This result is not very consistent. Redoing part a) with a different split in the data results in relatively different results. One big difference is the root mean squared error of the validation data in smaller values of regularization. The root mean squared error of the validation dataset seems to vary significantly between runs. However, in larger values of regularization, the root mean squared error of the validation dataset remains consistent.

Another characteristic of the graph that seems to remain consistent between runs is the regularization at which the minimum root mean squared error occurs at, which seems to stay around 10^{-1}.

As regularization increases, we see that the root mean squared error of the training data increases as well, meaning the accuracy of the data is decreasing. Thus, increasing regularization does not result in overfitting. This makes sense because regularization is meant to add variance to the dataset to improve generalization at the cost of the model's accuracy to the original test data. Due to the added variance, overfitting does not occur.

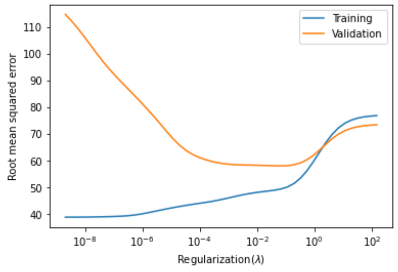
c) [1 mark] Now let's try regression using polynomials as our features. Again, `sklearn` has a tool to convert our `X` data into a version with all the polynomials calculated. Note that our `X` data has 10 inputs ($x_1; x_2; x_3; \dots x_{10}$) so when converted to polynomials up to degree 2 it will include $x_1^2, x_1x_2, x_1x_3, \dots x_2^2, x_2x_3$, and so on. Here is how you convert the raw input data into the features `F` that you can then use instead of `X` :

In [ ]:
```
F = sklearn.preprocessing.PolynomialFeatures(degree=2).fit_transform(diabetes.data)
```

Now repeat part a) using the new features. How does this change the result?

In [ ]:
```
lambds = np.exp(np.linspace(-20,5,50))

rmse_train_poly = []
rmse_validation_poly = []

F_train = sklearn.preprocessing.PolynomialFeatures(degree=2).fit_transform(X_train)
F_validation = sklearn.preprocessing.PolynomialFeatures(degree=2).fit_transform(X_validation)
F_test = sklearn.preprocessing.PolynomialFeatures(degree=2).fit_transform(X_test)

for lambd in lambds:
    reg = sklearn.linear_model.Ridge(alpha=lambd)
    # Train the data using training dataset
    reg.fit(F_train, Y_train)
    # Determine the outputs of the model for the training and validation datasets
    output_train = reg.predict(F_train)
    output_validation = reg.predict(F_validation)
    # Compute root mean squared error
    rmse_train_poly.append(np.sqrt(np.mean((Y_train-output_train)**2)))
    rmse_validation_poly.append(np.sqrt(np.mean((Y_validation-output_validation)**2)))

# Plot the RMSE for training and validation datasets
plt.semilogx(lambds, rmse_train_poly)
plt.semilogx(lambds, rmse_validation_poly)
plt.xlabel("Regularization($\lambda$)")
plt.ylabel("Root mean squared error")
plt.legend(["Training", "Validation"])
plt.plot()

ideal_lambda = lambds[rmse_validation_poly.index(min(rmse_validation_poly))]
min_rmse = min(rmse_validation_poly)
print(f"The minimum root mean squared error is {min_rmse}, which occurs at a lambda of {ideal_lambda}")
```

```
The minimum root mean squared error is 57.986588535353604, which occurs at a lambda of 0.07043526453886602
```

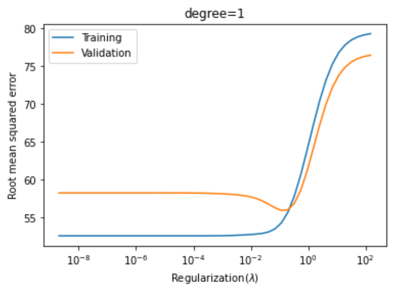


Comparing the graphs between parts a) and c), we notice that for the validation dataset, the root mean squared error starts at a much higher value with the polynomial regression. In addition to that, the root mean squared error begins decreasing almost immediately, whereas in part a), the root mean squared error of the validation data stays almost the same until a regularization of about 10^{-2}. Furthermore, the root mean squared error of the training data also increases almost immediately with the polynomial regression whereas in part a), it only starts increasing at a regularization of about 10^{-2}.

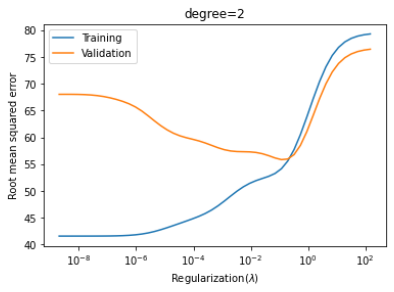
However, for the validation dataset, we can see that the minimum root squared error and the regularization at which it occurs remains the same between parts a) and c). Both have a minimum root mean squared error of about 55, occurring at a regularization of about 10^{-1}.

d) [1 mark] Increase the degree of polynomials used. Try values up to at least 5. Compute the same plots as in part a). How does this change the plots? Why does this happen? What happens if you increase the degree up to even larger values like 10 or 20? Why?

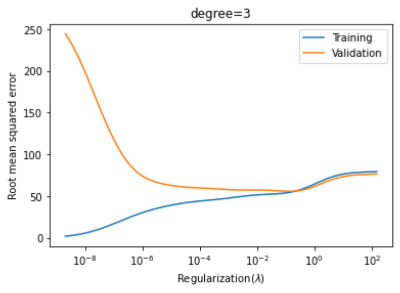
In [ ]:
```
X_train, X_remain, Y_train, Y_remain = sklearn.model_selection.train_test_split(
    diabetes.data, diabetes.target, test_size=0.667, shuffle=True,
)
X_validation, X_test, Y_validation, Y_test = sklearn.model_selection.train_test_split(
    X_remain, Y_remain, test_size=0.5, shuffle=True,
)
lambds = np.exp(np.linspace(-20,5,50))
degs = [1,2,3,4,5,6]

for deg in degs:
    rmse_train_poly = []
    rmse_validation_poly = []

    F_train = sklearn.preprocessing.PolynomialFeatures(degree=deg).fit_transform(X_train)
    F_validation = sklearn.preprocessing.PolynomialFeatures(degree=deg).fit_transform(X_validation)
    F_test = sklearn.preprocessing.PolynomialFeatures(degree=deg).fit_transform(X_test)

    for lambd in lambds:
        reg = sklearn.linear_model.Ridge(alpha=lambd)
        # Train the data using training dataset
        reg.fit(F_train, Y_train)
        # Determine the outputs of the model for the training and validation datasets
        output_train = reg.predict(F_train)
        output_validation = reg.predict(F_validation)
        # Compute root mean squared error
        rmse_train_poly.append(np.sqrt(np.mean((Y_train-output_train)**2)))
        rmse_validation_poly.append(np.sqrt(np.mean((Y_validation-output_validation)**2)))

    # Plot the RMSE for training and validation datasets
    plt.plot(lambds, rmse_train_poly)
    plt.plot(lambds, rmse_validation_poly)
    plt.xlabel("Regularization($\lambda$)")
    plt.xscale("log")
    plt.ylabel("Root mean squared error")
    plt.legend(["Training", "Validation"])
    plt.title(f"degree={deg}")
    plt.show()

    ideal_lambda = lambds[rmse_validation_poly.index(min(rmse_validation_poly))]
    min_rmse = min(rmse_validation_poly)
    print(f"The minimum root mean squared error is {min_rmse}, which occurs at a lambda of {ideal_lambda}")
```



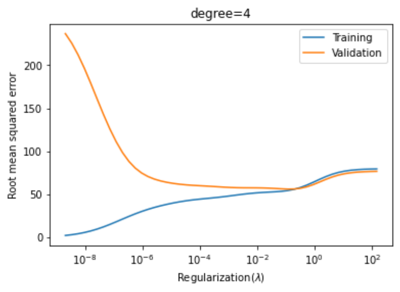
```
The minimum root mean squared error is 55.897005854008086, which occurs at a lambda of 0.11731916609425083
```



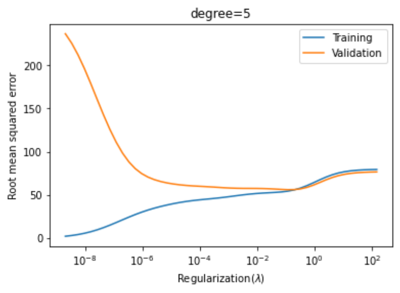
```
The minimum root mean squared error is 55.83075612708455, which occurs at a lambda of 0.11731916609425083
```



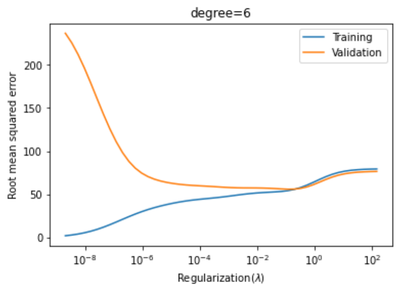
```
The minimum root mean squared error is 55.8305388625276, which occurs at a lambda of 0.11731916609425083
```



```
The minimum root mean squared error is 55.830528254800406, which occurs at a lambda of 0.11731916609425083
```



```
The minimum root mean squared error is 55.830528195438696, which occurs at a lambda of 0.11731916609425083
```



```
The minimum root mean squared error is 55.83052819303447, which occurs at a lambda of 0.11731916609425083
```

Increasing the degree seems to have the most effect when the degrees are between 1 and 3. Increasing the degree to values greater than 3 seem to have minimal effects on the graph. However, between degrees of 1 and 3, we note the following observations:

- Higher degrees lead to higher initial root mean squared error for validation data and lower initial root mean squared error for training data
- All graphs leads to almost the same ideal regularization value and minimum root mean squared error
- Since higher degrees have a higher initial root mean squared error but the same minimum root mean squared error at the same regularization, the graphs of higher degrees tend to have steeper slopes when approaching the ideal regularization
- After the ideal regularization point, all the graphs seem to have very similar root mean squared error values

The decrease in the initial root mean squared error for the training data likely occurs since increasing the number of polynomials creates more data points to train the data on. This causes the model to become much more accurate to the training data at the cost of generalization. Overfitting occurs to a much higher extent, leading to a decrease in accuracy in the validation dataset.

Increasing the degree to higher values like 10 or 20 leads to extremely long processing times. This is because the number of calculations that need to be made for each dataset increases exponentially. However, as seen in the first few graphs, increasing the degree does not have much effect on the final ideal regularization value. Because of this, using a degree of 1 or 2 is appropriate to maximize accuracy of the regularization value while minimizing processing time.