In [1]:
```
%matplotlib inline
```

SYDE 522 Assignment 1

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 [2]:
```
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)
```

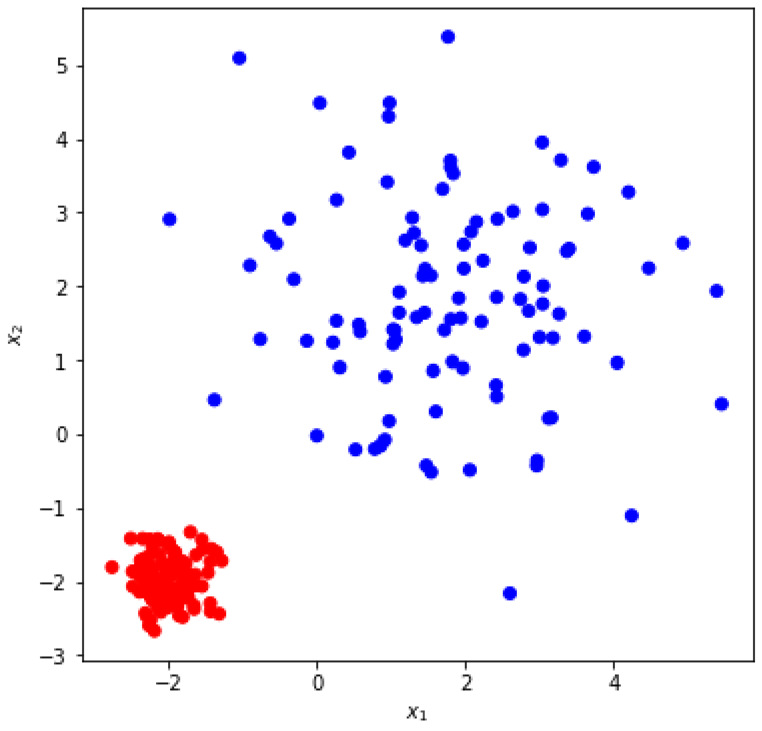
This produces two arrays, `data_x` which contains the input data (200 rows, each of which has 2 values x_1 and x_2), and `data_y` which contains the desired output data (either a 1 or a 0).

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 [9]:
```
import matplotlib.pyplot as plt
import numpy as np
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()
```



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.

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.

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.

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` .

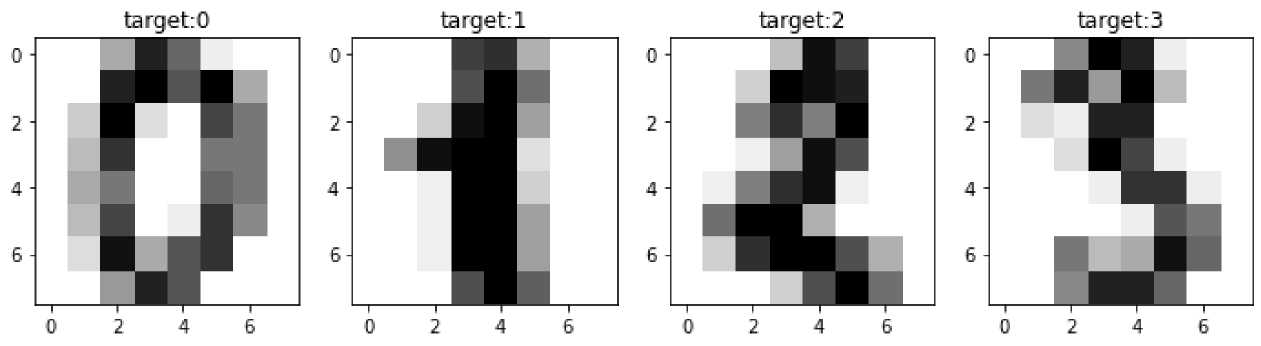
The dataset we will use is the UCI ML hand-written digits dataset https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

It is available with the following command:

In [15]:
```
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 [19]:
```
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)
```

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 [21]:
```
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.

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.

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?

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 [13]:
```
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.

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.

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.

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.

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 λ in class).

In [23]:
```python
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 [ ]:
```python
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 [24]:
```python
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 $e^{-20)$toe^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.

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?

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 ($x_1; x_2; x_3; \ldots x_{10}$) so when converted to polynomials up to degree 2 it will include $x_1^2, x_1x_2, x_1x_3, \ldots x_2^2, x_2x_3$, 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 [25]: F = sklearn.preprocessing.PolynomialFeatures(degree=2).fit_transform(diabetes.data)
```

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

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?