Assignment 1

In this assignment, we will be predicting the prices of houses using features like the house's age, distance to public transportation, and latitude/longitude coordinates. The data that we are using is the Real Estate Valuation Data Set from https://archive.ics.uci.edu/ml/datasets/Real+estate+valuation+data+set. Download the file `data.csv` from the course website (*not* from the URL, since we made minor changes to the data format).

We will be exploring both k Nearest Neighbour models and Linear Regression models.

For this entire assignment, you may not add loops that are not provided in the starter code.

Question 1: Data, Indexing, and Vectorized Code

Before beginning a machine learning task, one of the first thing that you should do is to understand the data that you are working with. That is where we should start: with the data. Along the way, we will illustrate how to use Python's `numpy` package to vectorize computation.

```
import numpy as np
import numpy.random as rnd

# Read the data
# Download "data.csv" from the course website
data = np.genfromtxt('data.csv', delimiter=',', skip_header=1)

# Display the *shape* of the data matrix
print(data.shape)

# Please leave these print statements, to help your TAs grade quickly.
print('\n\nQuestion 1')
print('----------')
```

Part (a)

Print the first column and the first 10 rows of the data. **Recall that you should not add loops that are not already in the starter code.**

Note that `data` is a 2D numpy array (i.e. a matrix), and its elements can be *indexed*. For examples `data[0, 0]` indexes the first row and column. Additionally, similar to python lists, numpy arrays support *slicing*: e.g. `data[1:3, 0]` and `data[200:, :]`.

```
print('\nQuestion 1(a):') # Please leave print statements like these

# print(...) # TODO: Only print your answer.
```

Part (b)

Print the second column and the first 10 rows of the data.

```
print('\nQuestion 1(b):')
```

Part (c)

What do you think the columns in parts (a) and (b) represent? Find the answer by reading the data set information and "Attribute Information" in https://archive.ics.uci.edu/ml/datasets/Real+estate+valuation+data+set. You should understand the meaning of the remaining fields as well.

We will be predicting the housing price (last column) using the some of the remaining features.

```
# Include your response in your writeup
```

Part (d)

Remove the first column from the data, and overwrite the variable `data` with the result. Print the *shape* of the resulting matrix `data`.

```
print('\nQuestion 1(d):')
# data = ... # TODO
print(data.shape)
```

Part (e)

We will first separate the data into training, validation, and test sets. Rather than choosing a random percentage of data points to leave out in our test set, we will instead place the *most recent* data points in our test set. In particular, any data point with date *larger* than `2013.417` will be placed in our test set. The code to select the test set element is written for you below. Pay attention to the way a boolean numpy array like `data[:, 0] > 2013.417` can be used to index elements of another numpy array.

Explain why this is a better strategy than randomly selecting data points in our test set.

```
test = data[data[:, 0] > 2013.417]

# Include your response in your writeup
```

Part (f)

Create a matrix `train_valid` that contains the data points that will be in the training or validation set. Then, print the shape of both new matrices from parts (e) and (f).

```
print('\nQuestion 1(f):')

#train_valid = ... # TODO

# print(test.shape)
# print(train_valid.shape)
```

Part (g)

We will use the variable `randarray`, given below, to separate our training and validation sets. This array assigns a random integer (0, 1, 2, 3, 4) to every element of the `train_valid` dictionary.

For each data point in `train_valid`, if its corresponding value in `randarray` is `0`, place that data point in the validation set. Otherwise, place that data point in the training set. To earn credit, you should do this *without* using any loops. (Hint: consider the way we indexed numpy arrays in parts e and f)

Print the shape of the training and validation matrices.

```
print('\nQuestion 1(g):')

# Below array was generated by calling
# randarray = rnd.randint(0, 5, train_valid.shape[0])
# Do NOT uncomment the above line of code. Instead, we are including
# the values you should use below.
randarray = np.array([2, 0, 1, 3, 0, 0, 0, 3, 2, 3, 1, 1, 2, 0, 4, 4, 0, 2, 1, 2, 2, 2,
                      4, 1, 3, 2, 0, 1, 2, 0, 3, 0, 3, 1, 3, 0, 4, 1, 4, 4, 0, 0, 1, 2,
                      4, 0, 0, 1, 1, 1, 2, 3, 4, 4, 3, 3, 0, 0, 0, 0, 2, 2, 3, 0, 0, 1,
                      4, 1, 4, 2, 2, 4, 4, 2, 0, 4, 0, 3, 2, 0, 4, 3, 1, 1, 0, 0, 0, 0,
                      1, 1, 4, 0, 3, 4, 2, 0, 0, 4, 4, 4, 4, 3, 3, 0, 0, 2, 2, 1, 3, 2,
                      4, 1, 2, 2, 3, 4, 1, 4, 3, 1, 1, 3, 0, 4, 4, 4, 0, 3, 3, 0, 4, 0,
                      0, 4, 3, 4, 1, 2, 2, 4, 4, 1, 2, 1, 1, 0, 4, 4, 4, 2, 0, 4, 2, 0,
                      4, 4, 1, 4, 0, 4, 0, 0, 1, 4, 3, 2, 4, 3, 1, 4, 1, 3, 4, 1, 0, 0,
                      4, 4, 2, 0, 4, 4, 4, 2, 3, 3, 4, 1, 0, 1, 2, 3, 1, 0, 1, 3, 4, 0,
                      0, 1, 2, 2, 2, 2, 4, 3, 1, 1, 4, 4, 1, 4, 2, 4, 0, 2, 4, 1, 3, 0,
```

```
                    4, 2, 3, 0, 4, 2, 3, 2, 2, 0, 2, 0, 2, 3, 2, 3, 4, 2, 4, 2, 2, 4,
                    3, 4, 0, 4, 4, 0, 1, 4, 2, 4, 2, 4, 0, 3, 4, 2, 1, 1, 3, 0, 1, 0,
                    3, 1, 3, 2, 4, 3, 1, 3, 0, 3, 0, 4, 2, 1, 2, 3, 2, 2, 4, 1, 4, 2,
                    1, 3, 1, 2, 2, 3, 1, 4, 2, 2, 4, 4, 1, 3, 2, 4, 1, 2, 4, 4, 0, 3,
                    1, 2, 1, 3, 3, 3, 2, 1, 3, 0, 2, 4, 2, 0, 3, 1, 0, 4, 2, 4, 1, 0,
                    4, 2, 2, 0, 1, 4, 3, 4, 0, 3, 2, 0, 2, 0])

# train = ... # TODO
# valid = ... # TODO

# print(train.shape)
# print(valid.shape)
```

Part (h)

Separate the input features and target values for each of the `train`, `valid`, and `test` sets. In particular, we will use the following columns as features: 1, 2, 3, 4, 5 (but not the date column 0). We will predict the housing price, which is in column 6.

We will refer to the five feature columns as x, and the housing price as t. We will need training, validation and testing versions of both x and t, for a total of 6 arrays. You should build these 6 arrays using the starter code below.

Print the first 2 rows of each of these six new numpy arrays.

```
print('\nQuestion 1(h):')

# TODO
train_x = None # TODO
train_t = None # TODO
valid_x = None # TODO
valid_t = None # TODO
test_x = None # TODO
test_t = None # TODO

# print(train_x[:2]) # TODO
# print(train_t[:2]) # TODO
# print(valid_x[:2]) # TODO
# print(valid_t[:2]) # TODO
# print(test_x[:2]) # TODO
# print(test_t[:2]) # TODO
```

Part (i)

Compute the mean and standard deviation of each column in `data`. Then, compute the mean and standard deviation of each column in `train_x`, saving the results in `x_mean` and `x_std`. Print both sets of means and standard deviations.

You may find the functions `np.mean` and `np.std` helpful. Find and read their documentations, and pay particular attention to the parameter `axis`.

```
print('\nQuestion 1(i):')
x_mean = None # TODO
x_std = None #TODO
```

Part (j)

For some of the models that we work with, we will be working with a *normalized* version of the features. In other words, we subtract the mean, and divide by the std, so that the features have zero mean and unit variance. Explain why using normalized data may be useful for some models, like the k-nearest neighbor model.

```
# Include your response in your writeup
```

Part (k)

Explain why we should compute the mean and standard deviation using the training data, rather than across the entire labeled data (including the validation/test sets).

```
# Include your response in your writeup
```

Part (l)

This part is meant to help you understand *broadcasting*, a method that numpy uses to perform vectorized computation. You will need to use broadcasting to complete part (m). Consider the following computation:

```
print('\nQuestion 1(l):')

tmp_a = np.array([[1.4, 2.5, 3.0], [9.1, 3.4, 2.3]])
tmp_b = np.sum(tmp_a, axis=0)
print(tmp_a)
print(tmp_b)
print(tmp_a - tmp_b)
```

Explain what computation was done to obtain the result.

```
# Include your response in your writeup
```

Part (m)

Using broadcasting (which you learned in the previous part), create the numpy array `norm_train_x`, which is the normalized version of the training data. Each column of this new matrix should have zero mean and unit variance.

Print the mean of the columns of the new matrix.

Print the first 2 rows of `norm_train_x`.

```
print('\nQuestion 1(m):')
norm_train_x = None # TODO
```

Part (n)

Consider the computation below, which is an alternative way of computing `norm_train_x` that uses loops rather than vectorized code. How much slower is this code compared to your code in the previous part? Include your response in your writeup.

```
print('\nQuestion 1(n):')
import time
nonvec_before = time.time()

norm_train_x_loop = np.zeros_like(train_x)
for i in range(train_x.shape[0]):
    for j in range(train_x.shape[1]):
        norm_train_x_loop = (train_x[i, j] - x_mean[j]) / x_std[j]

nonvec_after = time.time()
print("Non-vectorized time: ", nonvec_after - nonvec_before)


vec_before = time.time()
# TODO: Add your code here
vec_after = time.time()
print("Vectorized time: ", vec_after - vec_before)

# Include your response in your writeup
```

Question 2: Nearest Neighbour for Regression

In class we discussed nearest neighbours for classification, here we will use nearest neighbours for regression. In particular, we will use the nearest neighbor method to predict the housing prices, given the other features. Instead of taking a majority vote of the discrete target in the nearest neighbours, we will take the *average* of the continuous target (the housing prices). We will explore using both the *normalized* and *unnormalized* features.

For this question, you may not add loops that are not already in the starter code.

Part (a)

First, let's consider using a 1-nearest neighbour approach to predict the house price of the first data point `v` in the validation set. We will use the unnormalized version of the dataset.

Without using loops, compute the Euclidean distance between `v` and every data point in the training set `train_x`. Save the result in the numpy array `distances`. Print the first 10 rows of `distances`.

Then, find the index `n` with the minimum value of `distances`. Print the row `train_x[n]`, which is the closest data point to `v`, and the prediction `train_t[n]`. (There are several ways to do this!)

```
# Please leave these print statements, to help your TAs grade quickly.
print('\n\nQuestion 2')
print('----------')
print('\nQuestion 2(a):')

v = valid_x[0] # should be np.array([ 19.5    , 306.5947 ,   9.     ,  24.98034, 121.53951])

distances = None # TODO
n = None # TODO
```

Part (b)

Now, let's consider using a 3-nearest neighbour model to make a prediction for the same `v` from part (a), again using unnormalized data. In other words, we find the 3 smallest elements of `distances`, and average their corresponding values in `train_t` to obtain a prediction. Print this prediction.

You may want to consider sorting the list of distances, if you didn't do this in part (a). Again, there are several ways to do this sorting. You will also want to keep track of the indices (or the corresponding `t` values) as you sort the distances.

(Since this portion is unrelated to machine learning per se, how to do this is up to you to figure out. If this part is challenging, you may benefit from further computer science preparation before taking this course—i.e. any course that requires you to practise writing code to solve problems.)

```
print('\nQuestion 2(b):')
```

Part (c)

Complete the function `unnorm_knn(v, k)` that takes a feature vector `v`, and uses the k-nearest neighbour algorithm to make a prediction. Your code should be nearly identical to those from part (b), except `k` is now a parameter.

Print the prediction for `v=valid_x[1]`, with `k=5`.

```
print('\nQuestion 2(c):')

def unnorm_knn(v, k, features=train_x, labels=train_t):
    """
    Returns the k Nearest Neighbour prediction of housing prices for an input
    vector v.

    Parameters:
        v - The input vector to make predictions for
        k - The hyperparameter "k" in kNN
```

```
        features - The input features of the training data; a numpy array of shape [N, D]
                   (By default, `train_x` is used)
        labels - The target labels of the training data; a numpy array of shape [N]
                 (By default, `train_t` is used)
    """

print(unnorm_knn(v=valid_x[1], k=5))
```

Part (d)

We wrote most of the function `compute_mse` for you below. This function takes a parameter `predict`, which is itself a function that makes a prediction given a feature vector. This function is intended to compute the mean squared error of the predictions made using the `predict` method, across the provided dataset (by default, the entire unnormalized validation set is supplied).

Complete the Mean Square Error (MSE) computation. The Mean Square Error is another term for the average square loss across a data set. When you have done so, the code below will print the training and validation MSE for a (very simple) model that always predicts the *average* house price across the entire training set. We will call this a **baseline** model. Such a model is often used for sanity checking, and as a point of comparison. Your kNN model should be better than this baseline model.

```
print('\nQuestion 2(d):')
def compute_mse(predict, data_x=valid_x, data_t=valid_t):
    """
    Returns the Mean Squared Error of a model across a dataset

    Parameters:
        predict - A Python *function* that takes an input vector and produces a
                  prediction for that vector.
        data_x - The input features of the data set to make predictions for
                 (By default, `valid_x` is used)
        data_t - The target labels of the dataset to make predictions for
                 (By default, `train_t` is used)
    """

    errors = []
    for i in range(data_t.shape[0]):
        error = 0.0 # TODO: replace "0.0" with the actual computed error
        errors.append(error)
    return np.mean(errors)

def baseline(v):
    """
    Returns the average housing price given an input vector v.
    """
    return np.mean(train_t)

# compute and print the training and validation MSE
print(compute_mse(baseline, data_x=train_x, data_t=train_t))
print(compute_mse(baseline, data_x=valid_x, data_t=valid_t))
```

Part (e)

For each choice of `k` (1, 2, up to 30), compute the MSE on the training and validation sets for the corresponding kNN model. Store these values in the two lists `train_mse` and `valid_mse`. Print these two lists.

We include code below that plots the values in these two lists. Include the plot in your writeup. From the plot, what is the optimal value of `k`?

```
print('\nQuestion 2(e):')

train_mse = []
valid_mse = []
for k in range(1, 31):
    # create a temporary function `predict_fn` that computes the knn
    # prediction for the current value of the loop variable `k`
    def predict_fn(new_v):
        return unnorm_knn(new_v, k)

    # compute the training and validation MSE for this kNN model
    mse = compute_mse(predict_fn, data_x=train_x, data_t=train_t)
    train_mse.append(mse)
    mse = compute_mse(predict_fn, data_x=valid_x, data_t=valid_t)
    valid_mse.append(mse)


from matplotlib import pyplot as plt
plt.plot(range(1, 31), train_mse)
plt.plot(range(1, 31), valid_mse)
plt.xlabel("k")
plt.ylabel("MSE")
plt.title("Unnormalized kNN")
plt.legend(["Training", "Validation"])
plt.show()

# Include your response in your writeup
```

Part (f)

Let's consider the effect of normalization on a kNN model. Complete the function `norm_knn` below, that is identical to `unnorm_knn`, but uses normalized distances instead.

In other words, you should use the matrix `norm_train_x` form Question 1 to compute distances. Be careful about what validation data you use. In particular, should you normalized the validation data? If so, how? (Hint: How would you normalize the test data? What if you only have a single test point?)

Construct the same plot as in Part(e), but for `norm_knn`. Construct and print the lists `train_mse` and `valid_mse`.

```
print('\nQuestion 2(f):')

def norm_knn(v, k, features=norm_train_x, means=x_mean, stds=x_std, labels=train_t):
    """
    Returns the k Nearest Neighbour prediction of housing prices for an input
    vector v.

    Parameters:
        v - The input vector to make predictions for
        k - The hyperparameter "k" in kNN
        features - The normalized input features of the training data
                   (By default, `norm_train_x` is used)
        means - The means over the training data (By default, `x_mean` is used)
        stds - The standard deviations of the training data (By default, `x_std` is used)
        labels - The target labels of the training data; a numpy array of shape [N]
                 (By default, `train_t` is used)
    """

train_mse = []
valid_mse = []
```

```
for k in range(1, 31):
    # create a temporary function `predict_fn` that computes the knn
    # prediction for the current value of the loop variable `k`
    def predict_fn(new_v):
        return norm_knn(new_v, k)

    # compute the training and validation MSE for this kNN model
    mse = compute_mse(predict_fn, data_x=train_x, data_t=train_t)
    train_mse.append(mse)
    mse = compute_mse(predict_fn, data_x=valid_x, data_t=valid_t)
    valid_mse.append(mse)


from matplotlib import pyplot as plt
plt.plot(range(1, 31), train_mse)
plt.plot(range(1, 31), valid_mse)
plt.xlabel("k")
plt.ylabel("MSE")
plt.title("Normalized kNN")
plt.legend(["Training", "Validation"])
plt.show()

# Include your response in your writeup
```

Part (g)

Is it true that the training MSE of a kNN model is always 0, for any k? If so, prove it. If not, construct a small counter example.

```
# Include your response in your writeup
```

Part (h)

For this data set, does normalization improve the performance of kNN models? Justify your answer using the two plots from Part (f).

```
# Include your response in your writeup
```

Part (i)

So far, we have discussed two different ways of computing distances for the kNN model: Euclidean distance over the normalized and unnormalized features. What other ways of computing distances do you expect would *improve* performance? In a few sentences, describe your proposal, and explain why you would expect performance to improve. You do not need to implement your idea.

```
# Include your response in your writeup
```

Question 3: Polynomial Regression

For this question, you may not add loops that are not already in the starter code.

In this part of the assignment, we will use polynomial regression to predict the housing prices. We will use only one of the features, namely the house's age.

```
# Please leave these print statements, to help your TAs grade quickly.
print('\n\nQuestion 3')
print('----------')

train_d = train_x[:, 0] # house age
```

```
# Plot this feature against the target (house price)
plt.scatter(train_d, train_t)
plt.xlabel("House Age")
plt.ylabel("House Price")
plt.title("House Age vs. Price")
plt.show()
```

Part (a)

As a warm-up, we will fit a straight line to the data (a polynomial of degree 1).

Your job is to estimate the coefficients $w = [a, b]$ of the model

$$y(x) = a + bx$$

where x is the house's age and y is the predicted house price. To do this, find the values of a and b that minimize the cost function

$$\mathcal{J}(a, b) = \frac{1}{2N} \sum_{n=1}^{N} (t^{(n)} - y(x^{(n)}))^2$$

where the sum is over all the training points $(x^{(n)}, t^{(n)})$. Recall that the values of a and b that minimize this loss are given by the following equation:

$$w = (X^T X)^{-1} X^T t$$

where $w = [a, b]$ is the weight vector, t is a column vector of target values, and X is the data matrix (design matrix). Here, X should have two columns: the first column is all 1's, and the second column consists of the values in `train_d` (created for you).

You can construct the design matrix `X` by using functions like `np.ones`, `np.ones_like`, `np.stack` and `np.concatenate`, which you can look up in the numpy user guide. Do not use any loops. To avoid loops, you will also need to use matrix multiplication `@`, `np.linalg.inv`, and the transpose operation `X.T`, which you can also look up.

Store your weights (coefficients) $[a, b]$ in the variable `linear_coef`.

Print `linear_coef`.

```
print('\nQuestion 3(a):')

linear_coef = None # TODO


print(linear_coef)
```

Part (b)

Write a function `pred_linear` that uses your weights from Part (c) and makes predictions for a *numpy vector* `v` consisting of the ages of several houses that we would like to make predictions for. Note that unlike the prediction functions for KNN that you wrote in Question 2, which returned a single prediction for a single input point, the function `predict_linear` here returns a vector of many predictions given a vector of many input points. Your code for `predict_linear` should be completely vectorized and should not use any loops.

The function `plot_prediction` visualizes the house price predictions of a model. It is written for you. Uncomment the call to `plot_prediction(...)` to visualize the resulting linear model function. Include this plot in your write up.

```
def pred_linear(v, coef=linear_coef):
    """
    Returns the linear regression predictions of house prices, given
```

```
    a vector consisting of the ages of several houses that we would
    like to make predictions for.

    Parameters:
        v - A vector of house ages
        coef - The linear regression coefficient
    """
    return None # TODO Replace this with the actual solution

def plot_prediction(predict, title):
    """
    Display a plot that superimposes the model predictions on a scatter
    plot of the training data (train_d, train_t)

    Parameters:
        predict - A Python *function* that takes an input vector and produces a
                  prediction for that vector.
        title - A title to display on the figure.
    """
    # start with a scatter plot
    plt.scatter(train_d, train_t)

    # create several "house age" values to make predictions for
    min_age = np.min(train_d)
    max_age = np.max(train_d)
    v = np.arange(min_age, max_age, 0.1)

    # make predictions for those values
    y = predict(v)

    # plot the result
    plt.plot(v, y)
    plt.title(title)
    plt.xlabel("House Age")
    plt.ylabel("House Price")
    plt.show()

# plot_prediction(pred_linear, title="Linear Regression (no feature expansion)")
```

Part (c)

Complete the function `compute_mse_vectorized`. This function takes a parameter `predict`, which is itself a function that makes a prediction given a vector of house ages.

Unlike the function compute_mse in Question 2(d), `compute_mse_vectorized` should be completely vectorized and should not use any loops.

```
print('\nQuestion 3(c):')

def compute_mse_vectorized(predict, data_x=valid_x, data_t=valid_t):
    """
    Returns the Mean Squared Error of a model across a dataset

    Parameters:
        predict - A Python *function* that takes a vector of house ages,
                  and produces a vector of house price predictions
        data_x - The input features of the data set to make predictions for
                 (By default, `valid_x` is used)
```

```
        data_t - The target labels of the dataset to make predictions for
                 (By default, `train_t` is used)
    """
    v = data_x[:, 0] # house age
    return None # TODO
```

Part (d)

In the rest of this question, you will fit models of increasing complexity to the data. Here, for instance, you will fit a quadratic model (a polynomial of degree 2). That is, you should estimate the coefficients $[a, b, c]$ to the model

$$y(x) = a + bx + cx^2$$

where, like in part (a), x is the age of the house and y is its predicted price. Find the coefficients that minimize the same cost function as in Part (a). Once again, do not use any loops. Store your weights (coefficients) $[a, b, c]$ in the variable `quad_coef`.

Print `quad_coef`

```
print('\nQuestion 3(d):')

quad_coef = None # TODO

print(quad_coef)
```

Part (e)

Write a function `pred_quad` that uses your model from part (d) and makes predictions for a *numpy vector* `v` consisting of the ages of houses that we would like to make predictions for. As in part (b), do not use loops.

Uncomment the call to `plot_prediction(...)` to visualize the predictions of the model from part (d). Include this plot in your write up.

```
def pred_quad(v, coef=quad_coef):
    """
    Returns the degree 2 polynomial regression predictions of
    house prices, given a vector consisting of the ages of several houses
    that we would like to make predictions for.

    Parameters:
        v - A vector of house ages
        coef - The linear regression coefficient
    """
    return None # TODO replace this line

# plot_prediction(pred_quad, title="Polynomial Regression (M=2)")
```

Part (f)

Use the function `compute_mse_vectorized` and the `pred_quad` function to compute the training and validation MSE of your model in Part (d).

```
print('\nQuestion 3(f):')
# TODO
```

Part (g)

Compute the training and validation MSEs for polynomial regression, for polynomial degrees 0, 1, 2, ... up to M=10. In other words, for each of the polynomial degrees (0 to 10), compute the optimal, least-squares coefficients directly

like you did in parts (a) and (d). Then, compute the training and validation MSE using the `compute_mse_vectorized` like in parts (b-c) and (e-f).

Store these MSE values in the two lists `train_mse_r` and `valid_mse_r`. Print these two lists. Each list should have length 11.

```
print('\nQuestion 3(g):')

train_mse_r = []
valid_mse_r = []
for M in range(0, 11):
    X = None # TODO
    # train_mse_r.append(0) # TODO replace this
    # valid_mse_r.append(0) # TODO replace this


print(train_mse_r)
print(valid_mse_r)
```

Part (h)

Explain why `train_mse_r[0]` and `valid_mse_r[0]` should be identical to your result from Question 2 (d).

```
# Include your response in your writeup
```

Part (i)

Use the code below to plot the training and validation MSEs that you computed in part (h). Describe the shape of this graph, and briefly explain why we see this shape.

```
plt.plot(range(0, 11), train_mse_r)
plt.plot(range(0, 11), valid_mse_r)
plt.xlabel("Polynomial Degree")
plt.ylabel("MSE")
plt.title("Polynomial Regression")
plt.legend(["Training", "Validation"])
plt.show()

# Include your response in your writeup
```

Part (j)

Use the function `plot_prediction` to plot the model predictions for polynomial degrees 3-10. These plots are analogous to those in Part (b) and Part (e), and should have appropriate titles.

Which model shows an example of overfitting? Which model shows an example of underfitting? Justify your reasoning.

```
for M in range(3, 11):
    X = None # TODO
    coef = None # TODO
    # plot_prediction(pred, title="Polynomial Regression (M=%d)" % M)

# Include your response in your writeup
```

Part (k)

Will normalization affect the linear regression validation MSE when $M = 1$? If so, provide an example where the validation MSE differs. If not, provide a proof.

```
# Include your response in your writeup
```

Part (l)

Will normalization affect the linear regression validation MSE when $M > 1$? If so, provide an example where the validation MSE differs. If not, provide a proof.

```
# Include your response in your writeup
```

Part 4: Gradient Descent

In this part of the assignment, we will solve the polynomial regression problem by optimizing the square error cost function via gradient descent.

Consider this model $y(x) = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M$ and the cost function $\mathcal{J}(\mathbf{w}) = \frac{1}{2N} \sum_{i=1}^{N} (t^{(i)} - y(x^{(i)}))^2$

Part (a)

Derive, by hand, an equation for the derivative $\frac{\partial \mathcal{J}}{\partial w_0}$.

```
# Include your response in your writeup
```

Part (b)

Derive, by hand, an equation for the derivative $\frac{\partial \mathcal{J}}{\partial w_1}$.

```
# Include your response in your writeup
```

Part (c)

More generally, derive, by hand, the derivative $\frac{\partial \mathcal{J}}{\partial w_j}$.

```
# Include your response in your writeup
```

Part (d)

Using your result from Part (c), show that the gradient vector $\nabla \mathcal{J}(w) = \left[\frac{\partial \mathcal{J}}{\partial w_0} \quad \frac{\partial \mathcal{J}}{\partial w_1} \quad \cdots \quad \frac{\partial \mathcal{J}}{\partial w_M} \right]^T$ has $\nabla \mathcal{J}(w) = \frac{1}{N}(\mathbf{y} - \mathbf{t})^T \mathbf{X}$, where $\mathbf{X}$ is the design matrix.

```
# Include your response in your writeup
```

Part (e)

Complete the function `grad` that takes the a weight vector `weight`, the training data `X` and `t`, and computes the gradient $\nabla \mathcal{J}(w)$ at that `weight`.

```
# Please leave these print statements, to help your TAs grade quickly.
print('\n\nQuestion 4')
print('----------')
print('\nQuestion 4(e):')

def grad(weight, X, t):
    '''
    Return gradient of each weight evaluated at the current value

    Parameters:
        `weight` - a current "guess" of what our weights should be,
                   a numpy array of shape (D)
        `X` - matrix of shape (N,D) of input features
        `t` - target y values of shape (N)

    '''
    return None
```

```
# Please leave this print statement for grading:
print(grad(np.array([1]), np.array([[1], [1]]), np.array([2, 2])))
```

Part (f)

We can check that our `grad` function in part (e) is implemented correctly using the finite difference rule. In 1D, the finite difference rule tells us that for small h, we should have

$$\frac{f(x+h) - f(x)}{h} \approx f'(x)$$

Prove to yourself (and your TA) that `grad` is implement correctly by comparing the result from `grad` and a gradient estimation obtained from computing $\mathcal{J}(w)$ and using the finite difference rule. Briefly justify your choice(s) of `w`, `b`, and `X`.

```
# Include your response in your writeup
```

Part (g)

Complete the function `solve_via_gradient_descent` that takes the maximum degree k of a polynomial regression, the learning rate `alpha`, and the number of iterations `niter`. This function begins by initializing all of your polynomial regression weights to 0, and returns the value of the weights after `niter` updates of gradient descent.

You should use the `grad` function you wrote in the Part (e) as a helper function.

Print the value of `solve_via_gradient_descent(M, alpha=0.0025, niter=10000)`

```
print('\nQuestion 4(g):')

def solve_via_gradient_descent(M, alpha=0.0025, niter=10000):
    '''
    Given `M` - maximum degree of the polynomial regression model
          `alpha` - the learning rate
          `niter` - the number of iterations of gradient descent to run
    Solves for linear regression weights.
    Return weights after `niter` iterations.
    '''
    # initialize all the weights to zeros
    w = np.zeros([M + 1])

    # construct the design matrix
    for i in range(niter):
        w = w # TODO
    return w

print(solve_via_gradient_descent(M=1, alpha=0.0025, niter=10000))
```

Part (h)

How do your gradient descent weights from Part (g) compare to the weights that you got from Question 3(a)?

```
# Include your response in your writeup
```

Part (i)

Print the value of `solve_via_gradient_descent(M=1, alpha=0.01, niter=50)`.

In your writeup, explain why this result differs from the one in Part(g).

```
print('\nQuestion 4(i):')
print(solve_via_gradient_descent(M=1, alpha=0.01, niter=50))
```

```
# Include your response in your writeup
```

Question 5: Cross Validation and Model Selection

When we have a small amount of data, evaluating models is challenging. We may not have the luxury of leaving out data like we did in Questions 2-4 for the purposes of validation. In this part of the assignment, we will explore a more data efficient way of evaluating machine learning models called **cross validation**.

Recall that in Question 1, we first set aside a test set, then we split the remaining data into 80% training and 20% validation. The validation performance (e.g. MSE) helps us select hyper-parameters.

The idea behind cross validation is to repeat this process of train/validation split *multiple* times, so that we get several *different* estimates of the validation accuracy. In particular, in Question 1(g), we gave each element of the (non-test) data set a random label (0, 1, 2, 3, 4). This way, we effectively split the data set into 5 groups, and decided that group 0 was the validation set.

In this question, we will perform k-Fold Cross Validation. In other words, we will split our data set into k random groups, and repeat the training, allowing each of the k groups a chance to be the validation set. In our case, since we have 5 different groups, what we are proposing is called "5-Fold Cross Validation".

In each of the 5 training iterations, we will choose a different group (out of the 5) to be the validation set. The remaining 4 groups will be used for training.

Part (a)

By repeating training 5 different times for 5 different training/validation splits, we will get 5 different estimates for the validation MSE, which can be averaged together.

What is the advantage of averaging 5 MSE measures, rather than just using a single MSE measure, especially when the data set size is small?

```
# Include your response in your writeup
```

Part (b)

We will perform 5-fold cross validation on the **unnormalized** kNN model. Produce a plot of the average validation MSE across the 5 folds, similar to the plot from Quesiton 2(e). Your plot should have values of "k" on the x-axis (ranging from 1 to 30 like before), and "Average validation MSE" on the y-axis. Clearly label your plot.

Part (c)

Compare your plot from Question 5(b) with the plot from Question 2(e). Which plot appears more *smooth* (less noisy)? Explain why you think that is.

```
# Include your response in your writeup
```

Part (d)

What are some advantages and disadvantages of using k-fold cross validation?

```
# Include your response in your writeup
```

Part (e)

If you were to choose a model from questions 2-5 to deploy, which model will you deploy? Compute the test MSE for this optimal model.

```
# print(...) # TODO
```

Part (f)

Explain why you chose the model from the previous part.

```
# Include your response in your writeup
```