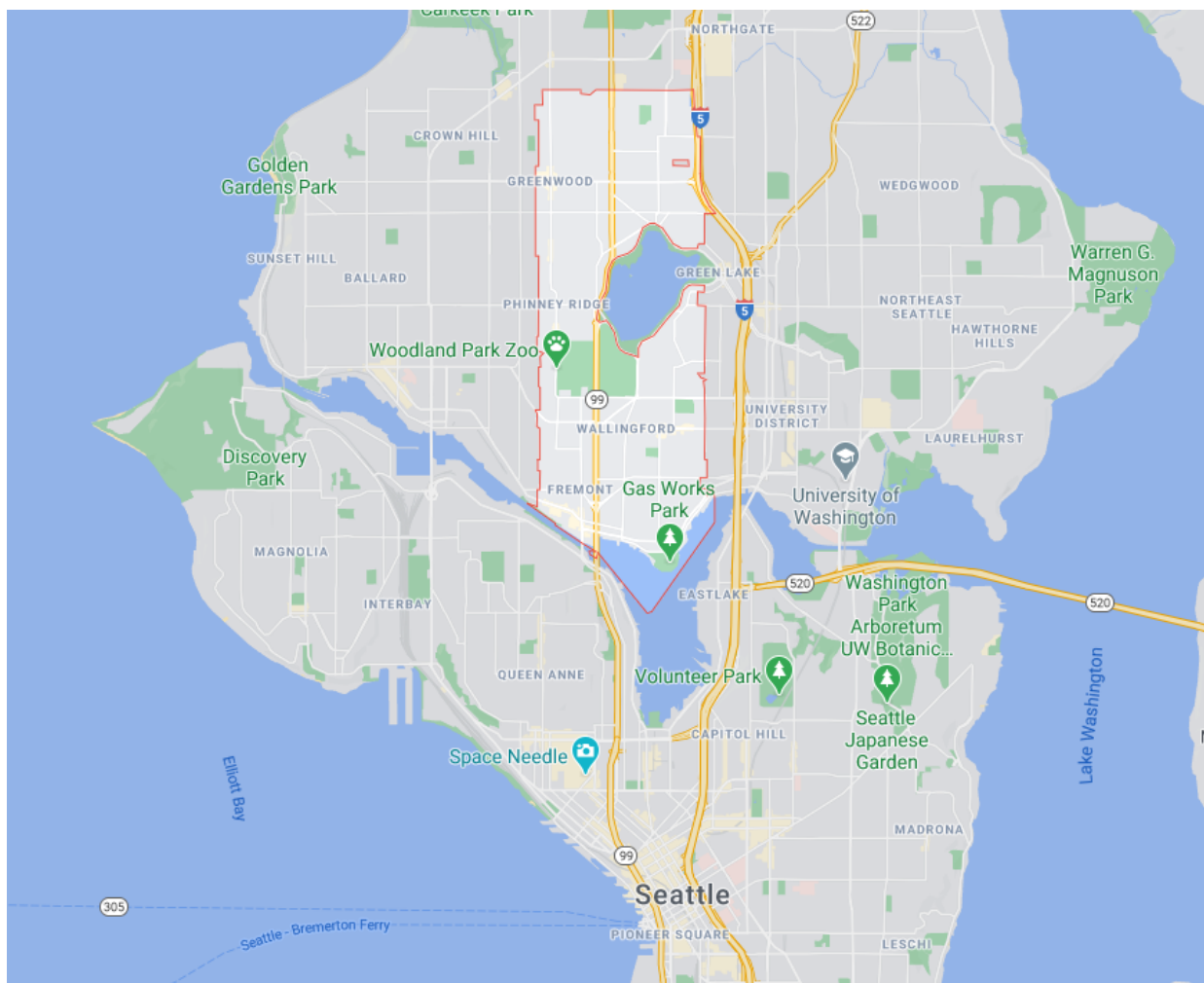


```
In [1]: import copy
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.style as style
import matplotlib.gridspec as gridspec
import seaborn as sns
from mpl_toolkits import mplot3d
from scipy import stats
%matplotlib inline
df = pd.read_csv('./data_mod_1.csv')
```

## \$0 Exploring State Zip WA 98103

The 148 samples in WA 98103 is a comfortable amount of work with, to verify our machine learning algorithm results with hand calculations. The algorithm can be extended to the rest of the 4550 samples in the original dataframe, that comprises the entire Seattle region.



```
In [118... df_4 = df.loc[ (df['statezip'] == 'WA 98103'), ['bedrooms', 'bathrooms', 'sqft_living',

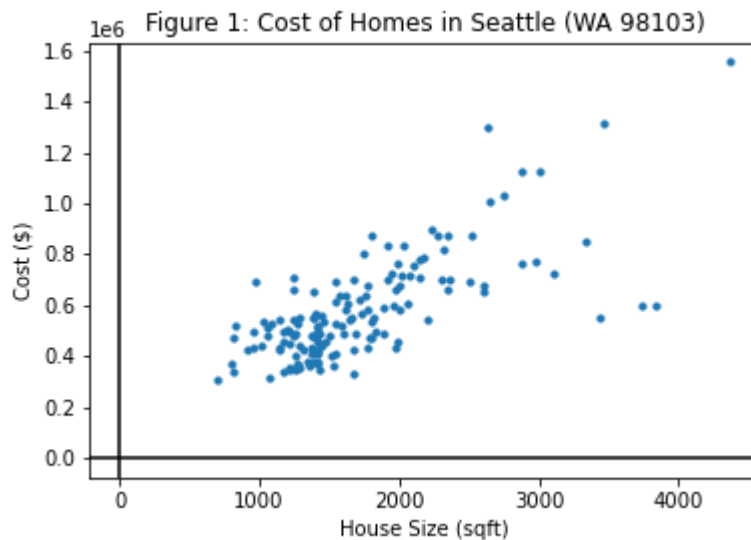
X_ = df_4.sqft_living # sqft is the most telling of price
X = X_.to_numpy() # we want to convert from series to a numpy array
ones = np.ones(len(X)) # adding a ones column in front
X_with_ones_col = np.column_stack((ones, X)) # Data Preprocessing
```

```
y_ = df_4.price # target
y = y_.to_numpy()
```

## §1.0 Applying Machine Learning Models

```
In [805... fig4, ax4 = plt.subplots()

ax4.scatter(X, y, s=10) # plot looks fine
ax4.axhline(0, color='black')
ax4.axvline(0, color='black')
ax4.set_title("Figure 1: Cost of Homes in Seattle (WA 98103)")
ax4.set_xlabel("House Size (sqft)")
ax4.set_ylabel("Cost ($)")
```



```
Out[805... Text(0, 0.5, 'Cost ($)')
```

## §1.1 Linear Regression with the sklearn Library

```
In [21]: from sklearn import datasets
from sklearn import linear_model
from sklearn.model_selection import train_test_split
```

```
In [108... model = l_reg.fit(X_with_ones_col, y)
```

```
In [109... l_reg.coef_
```

```
Out[109... array([ 0.          , 229.49521863])
```

```
In [80]: l_reg.intercept_
```

```
Out[80]: 184435.24482113082
```

From the sklearn, the weights for  $w_0$  and  $w_1$  are 184435.2 and 229.5 respectively.

## §1.2 An Analytic Approach

The advantages of an analytic approach:

- no need for gradient descent

- no need to perform thousands of iterations
- no need for learning rate guesswork
- no need for feature scaling
- have to ensure vectorization

```
In [86]: def normal_equation(X, y):
         return np.linalg.inv(X.T @ X) @ X.T @ y
```

```
In [556... normal_equation(X_with_ones_col, y)
```

```
Out[556... array([184435.24482113,    229.49521863])
```

From our analytic approach, the weights for  $w_0$  and  $w_1$  are 184435.2 and 229.5 respectively. These values are exactly the same as the values from sklearn.

### §1.3 Developing a Cost Function

Approach 2 is more involved, for example, one has to adjust the number of iterations, guess the learning rate and more (more discussed below). Since more things can go wrong with gradient descent than the analytic solution, the results from gradient descent+cost function is compared to the analytic solution.

We explore how to solve for the weights with gradient descent on a cost function because in some situation, gradient descent works where the normal equation cannot be applied, like for example in logistic regression, knn.

#### §1.3.1 Scaling Down

For plotting purposes, it would be ideal to have weights close in value so that the cost function looks decent. I'm going to scale the X by 1000 and the scale the y by 100,000.

The cost function can be contrived with the original values, and the weights correctly come out as  $w_0=184435.2$  and  $w_1= 229.5$ , but the cost function does not look good.

Note that, I could scale down with standard feature scaling techniques as well ( subtracting the mean and divide by the standard deviation from each value) but I've opted not to because I want to get a scaled down value by tens (ie 184435 -> 1.84) of what we got from sklearn and from the analytic solution, so that all three approaches are comparable.

```
In [112... X_scaled = X/1000
X_scaled_with_ones_col = np.column_stack((np.ones(len(X_scaled)), X_scaled))
y_scaled = y/100000
```

If we use the normal equation, we get the following weights

```
In [116... normal_equation(X_scaled_with_ones_col, y_scaled)
```

```
Out[116... array([1.84435245,  2.29495219])
```

Note that, the weight values changed from

$w_0=184435.2$  and  $w_1= 229.5$  to  **$w_0=1.84$  and  $w_1= 2.29$**

are much closer in value now, it will be much easier to visualize in a plot.

If the cost function and gradient descent are done correctly below, it will yield  $w_0=1.84$  and  $w_1=2.29$ .  $w_1=2.29$ .

## \$1.4 Plotting the Cost Function

```
In [119... def plot(g): # The cost function, tests all pairs of weights in a specified range.
    g2 = g*g
    steps = np.linspace(-4, 6, g)
    w0, w1 = np.meshgrid(steps, steps)
    w0_ = w0.ravel()
    w1_ = w1.ravel()
    w = np.column_stack([w0_, w1_])

    def calculateCostArray(X, y, i):
        return computeCost(X, y, w[i])

    def computeCost(X, y, w):
        t = X @ w - y
        t_squared = t * t # vectorized. element wise squaring
        cost = 1/(2*len(X)) * sum(t_squared)
        return cost

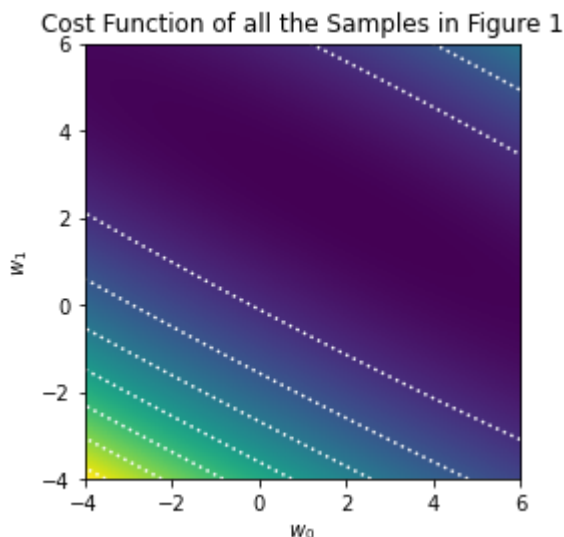
    costs = np.zeros(g2)
    for i in range(g2):
        costs[i] = calculateCostArray(X_scaled_with_ones_col, y_scaled, i)

    costs = costs.reshape(g, g, -1)
    costs = costs.squeeze()

    fig3, ax3 = plt.subplots()

    ax3.imshow(costs, origin='lower', extent=(w0.min(), w0.max(), w1.min(), w1.max()))
    ax3.set_title("Cost Function of all the Samples in Figure 1")
    ax3.set_xlabel('$w_0$')
    ax3.set_ylabel('$w_1$')
    ax3.contour(w0, w1, costs, colors='white', linestyle=':')

plot(100)
```



The darkest region within this cost function represents the lowest point of our cost function. We want to let the algorithm know, that has to pay a cost, for the the pair of weights ( $w_0$ ,  $w_1$ ) it picks. Every pair of weights has an associated cost. An algorithm will pick the lowest point on the cost function as this corresponds to the lowest cost.

We have a cost function representation as a contour and a surface plot. Now we need to apply gradient descent to our cost function and hopefully we get  $w_0=1.84$  and  $w_1= 2.29$

## §1.5 Gradient Descent

Our initial guess is  $w_0=0$  and  $w_1= 0$ . After gradient descent, we expect  $w_0=1.84$  and  $w_1=2.29$

```
In [89]: def gradient_descent(X, y, learn_rate=0.00004, num_steps= 10000):
          w = np.array([0, 0]) # so instead of guessing x, we're guessing w
          for i in range(num_steps):
              grad = (X.T @ X) @ w - X.T @ y # so.. this is a very useful equation
              w = w - learn_rate*grad
          return w
```

```
In [819... w_from_gradient_descent = gradient_descent(X_scaled_with_ones_col, y_scaled)
           w_from_gradient_descent
```

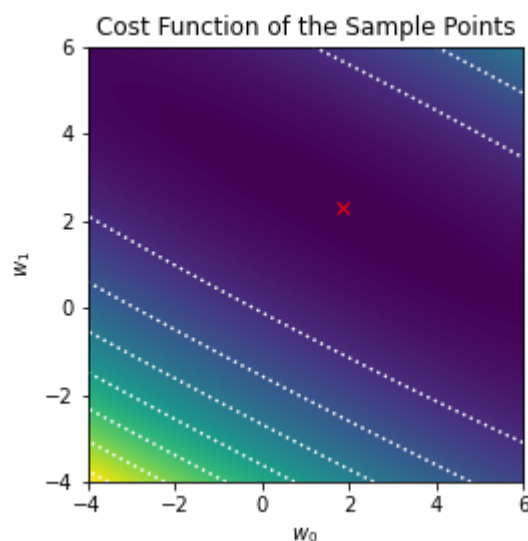
```
Out[819... array([1.84270283, 2.29581986])
```

With gradient descent, we come up with values  $w_0=1.84$  and  $w_1= 2.29$ , the same answer as our above approaches and the answer we were expecting.

## §1.6 Gradient Descent on our Cost Function

We can plot the weights on our contour plot we've created in §1.3.1

```
In [820... plot(100)
           plt.plot(*w_from_gradient_descent, 'xr')
```



```
Out[820... [<matplotlib.lines.Line2D at 0x1dfc9176cd0>]
```