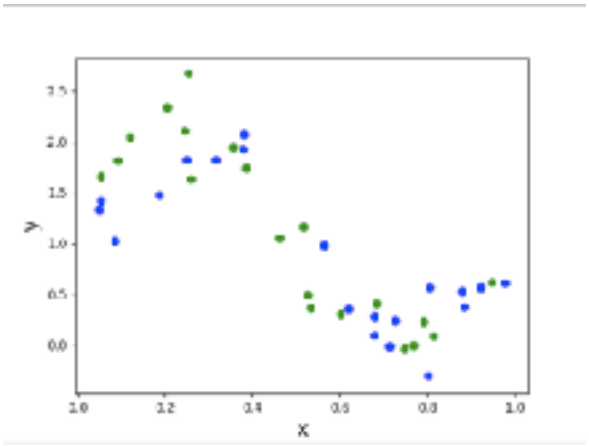
4.a) Visualizing Training & Test Data

In the figure, the green data represents training data, and the blue represents test data. Overall, it likes like our training data is a pretty good representation of our test data, suggesting we should be able to use a somewhat complex model and get good results without too much overfitting. Both the data sets exhibit a similar linear trend with a negative slope, so linear regression should do pretty well. For both the training and test data though there will be some error near the low and high x ends of a linear model…these could likely see better performance with a third degree regression model which would have a positive slope for low x values, negative slope in the middle, and positive slope again towards the higher x values.



4.b) Visualizing Training & Test Data

The matrices to the right represent the X-values from the training data set, and the feature matrix we will use for linear regression with an extended unity feature to simplify adding a bias to our linear model. The second matrix is a result of inputting the first into generate_polynomial_features().

```
[[2.515773]
 [2.790545]
 [2.685289]
 [2.946889]
 [2.53169 ]
 [2.118853]
 [2.600345]
 [2.899562]
 [2.355233]
 [2.204493]
 [2.812833]
 [2.952854]
 [2.527825]
 [2.260226]
 [2.744896]
 [2.383914]
 [2.767244]
 [2.461121]
 [2.746685]
 [2.052486]]

[[1.        2.515773]
 [1.        2.790545]
 [1.        2.685289]
 [1.        2.946889]
 [1.        2.53169 ]
 [1.        2.118853]
 [1.        2.600345]
 [1.        2.899562]
 [1.        2.355233]
 [1.        2.204493]
 [1.        2.812833]
 [1.        2.952854]
 [1.        2.527825]
 [1.        2.260226]
 [1.        2.744896]
 [1.        2.383914]
 [1.        2.767244]
 [1.        2.461121]
 [1.        2.746685]
 [1.        2.052486]]
```

4.c) Predict

The predict function can be implemented with a single vectorized line, simply the inner product of the extended feature vector and our model's current coefficient vector.

```
y = np.dot(X,self.coef_)
```

‘

4.d.i) Cost

The cost function can be implemented with a single line:

```
cost = np.sum((y-self.predict(X))**2)
```

Which implements the objective function:

$$J(w) = \sum_{n=1}^{N} (h_w(x_n) - y_n)^2.$$

With an all zero-coefficient linear model, the training cost get's printed as: `40.233647409671`

4.d.ii) Gradient Descent (Fixed learning rate)

The gradient descent algorithm is implemented through the following update step:

```
self.coef_ = self.coef_ - (2*eta*np.dot(np.transpose(X),(np.dot(X,self.coef_)-y)))
```

For this part we use a default learning rate of .01. For our training data this results in a number of iterations before convergence, a final objective function value, and a final coefficient vector shown in the following printout:

```
764, 3.912576405791487, array([ 2.44640703, -2.81635346]))
```

4.d.iil) Varying the learning rate

Learning Rate (eta)	Number of Iterations Before Convergence	Final Value of Objective Function
0.0001	10000	4.0863970367957645
0.001	7020	3.9125764057919463
0.01	764	3.912576405791487
0.0407	10000	2.710916520014198e+39

From the above table we see that 0.0001 was too small so the gradient descent algorithm wasn't able to converge after 10,000 iterations (our predetermined max number of iterations). 0.0407 was so large that the problem became unstable and the cost appears to have grown without bound…the predicted objective function would have oscillated on either side of the tire minimum indefinitely. Both 0.001 and 0.01 were valid in that they converged to about the same minimum after less than the max number of iterations, but 0.01 was clearly more efficient—converging in about 10 times fewer iterations.

4.e.i) Closed Form Solution Implementation

The closed form solution can be found with the simple matrix equation: $w = (X^T X)^{-1} X^T y.$

We implement this with the following line:

```
self.coef_ = np.dot((np.dot(np.linalg.pinv(np.dot(np.transpose(X),X)),np.transpose(X))),y)
```

4.e.ii) Closed Form vs Gradient Descent

Using the learning rate of 0.01, we rerun the gradient descent algorithm on our training data and compare it to the results of the closed form solver. The gradient descent algorithm ran in **0.04192090034484863 seconds** with a final coefficient vector of [2.44640703 -2.81635346] and a final cost of 3.912576405791487. The closed form solver ran in **0.005042076110839844 seconds** with a final coefficient vector of [2.44640709 -2.81635359] and a final cost of 3.9125764057914645. Therefore while the closed form found the exact solution, the GD algorithm found something correct to many digits of precision in a shorter amount of time.

4.f) Using learning rate as function of iteration:

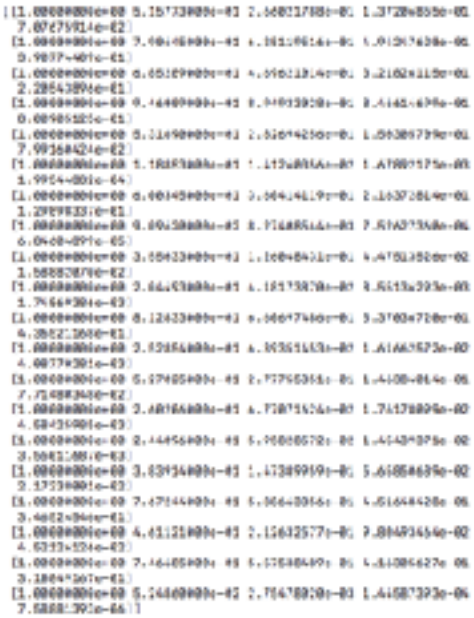
If we use a learning rate that depends on the iteration number by the following function:

$$\eta_k = \frac{1}{1+k}$$

We see the gradient descent takes a little bit longer— **0.04964303970336914 seconds**— and takes 1678 iterations (about twice as many as the fixed learning rate version). The final cost is about the same: 3.912576405792011. Note this lengthening occurs because at the beginning with the large learning rate, the gradient descent simply oscillates on either side of the minimum, before eventually strictly decreasing towards that minimum when the learning rate is small enough.

4.g) Generating a higher degree feature vector:

The image below shows a feature vector for m = 4:



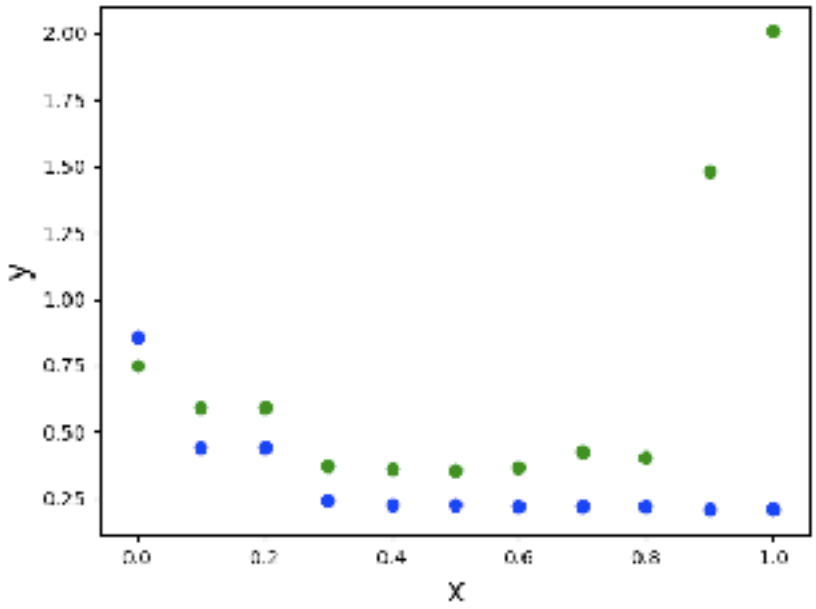
4.h) RMSE:

The RMSE is implemented with the following:

```
cost = self.cost(X,y)
error = np.sqrt(cost/X.shape[0])
```

We prefer RMSE to simply using the objective function because it normalizes the error to compare between models of different numbers of samples, else the error would simply depend on if there were a lot of samples or not.

4.i) Finding best degree:



The above figure shows the RMSE for training data (blue) and test data (green) with increasing model degree (from 0 to 10 in increments of 1). We see that the best fit model is for degree 4 which has the lowest test error. Note though that it doesn't necessarily have the lowest training error because past degree 4 there appears to be slight overfitting to the training data…most drastically seen for degree 9 and 10 where the training data is low and the test data is extremely high.