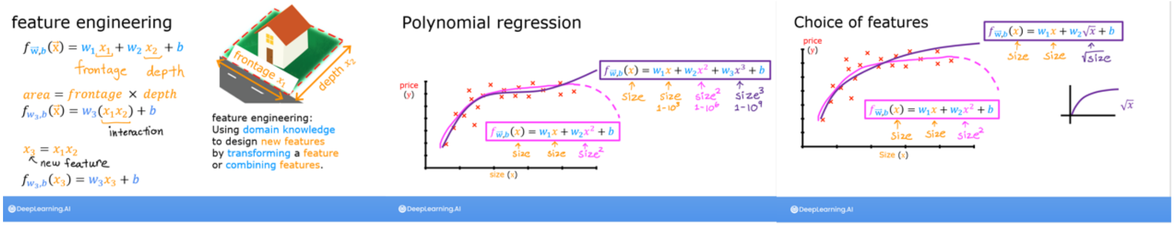
Optional Lab: Feature Engineering and Polynomial Regression



Goals

In this lab you will:

- explore feature engineering and polynomial regression which allows you to use the machinery of linear regression to fit very complicated, even very non-linear functions.

Tools

You will utilize the function developed in previous labs as well as matplotlib and NumPy.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from lab_utils_multi import zscore_normalize_features, run_gradient_descent_feng
np.set_printoptions(precision=2)  # reduced display precision on numpy arrays
```

Feature Engineering and Polynomial Regression Overview

Out of the box, linear regression provides a means of building models of the form:

$$f_{\mathbf{w},b} = w_0x_0 + w_1x_1 + ... + w_{n-1}x_{n-1} + b \tag{1}$$

What if your features/data are non-linear or are combinations of features? For example, Housing prices do not tend to be linear with living area but penalize very small or very large houses resulting in the curves shown in the graphic above. How can we use the machinery of linear regression to fit this curve? Recall, the 'machinery' we have is the ability to modify the parameters $\mathbf{w}$, $\mathbf{b}$ in (1) to 'fit' the equation to the training data. However, no amount of adjusting of $\mathbf{w}$,$\mathbf{b}$ in (1) will achieve a fit to a non-linear curve.

Polynomial Features

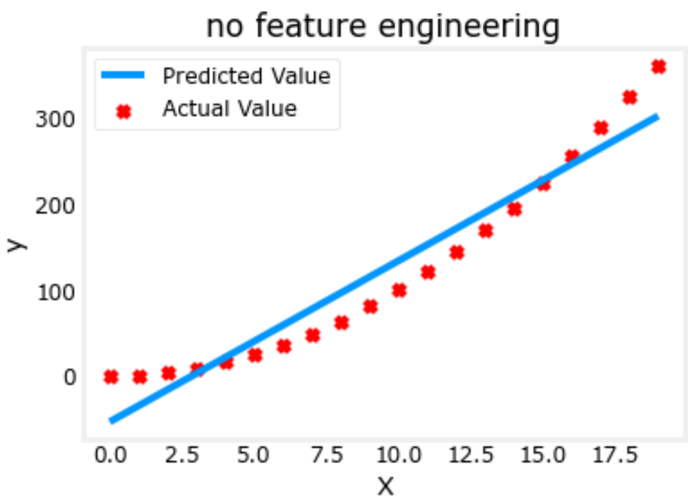
Above we were considering a scenario where the data was non-linear. Let's try using what we know so far to fit a non-linear curve. We'll start with a simple quadratic: $y = 1 + x^2$

You're familiar with all the routines we're using. They are available in the lab_utils.py file for review. We'll use `np.c_[..]` (https://numpy.org/doc/stable/reference/generated/numpy.c_.html) which is a NumPy routine to concatenate along the column boundary.

In [2]:

```
# create target data
x = np.arange(0, 20, 1)
y = 1 + x**2
X = x.reshape(-1, 1)

model_w,model_b = run_gradient_descent_feng(X,y,iterations=1000, alpha = 1e-2)

plt.scatter(x, y, marker='x', c='r', label="Actual Value"); plt.title("no feature
plt.plot(x,X@model_w + model_b, label="Predicted Value");  plt.xlabel("X"); plt.y
```

```
Iteration         0, Cost: 1.65756e+03
Iteration       100, Cost: 6.94549e+02
Iteration       200, Cost: 5.88475e+02
Iteration       300, Cost: 5.26414e+02
Iteration       400, Cost: 4.90103e+02
Iteration       500, Cost: 4.68858e+02
Iteration       600, Cost: 4.56428e+02
Iteration       700, Cost: 4.49155e+02
Iteration       800, Cost: 4.44900e+02
Iteration       900, Cost: 4.42411e+02
w,b found by gradient descent: w: [18.7], b: -52.0834
```



Well, as expected, not a great fit. What is needed is something like $y = w_0x_0^2 + b$, or a **polynomial feature**. To accomplish this, you can modify the *input data* to *engineer* the needed features. If you swap the original data with a version that squares the x value, then you can achieve $y = w_0x_0^2 + b$. Let's try it. Swap `X` for `X**2` below:

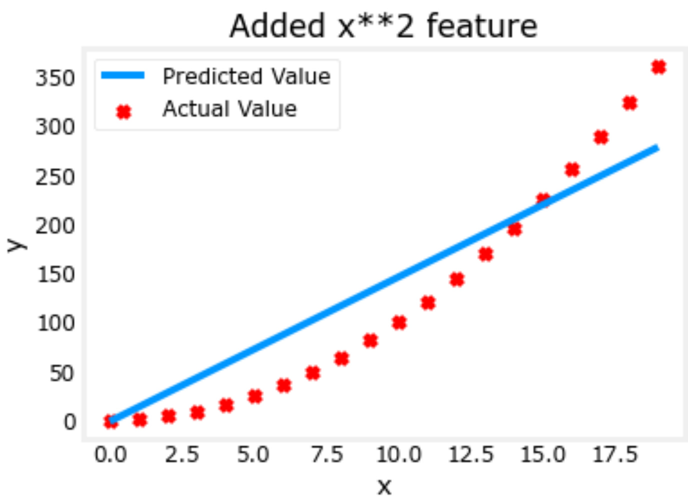
In [3]:

```
# create target data
x = np.arange(0, 20, 1)
y = 1 + x**2

# Engineer features
X = x**2       #<-- added engineered feature
```

In [3]:

```
X = X.reshape(-1, 1)  #X should be a 2-D Matrix
model_w,model_b = run_gradient_descent_feng(X, y, iterations=10000, alpha = 1e-5)

plt.scatter(x, y, marker='x', c='r', label="Actual Value"); plt.title("Added x**2
plt.plot(x, np.dot(X,model_w) + model_b, label="Predicted Value"); plt.xlabel("x"

```

```
Iteration         0, Cost: 1.41576e+04
Iteration      1000, Cost: 1.98104e+03
Iteration      2000, Cost: 9.65496e+02
Iteration      3000, Cost: 8.78850e+02
Iteration      4000, Cost: 8.69524e+02
Iteration      5000, Cost: 8.66644e+02
Iteration      6000, Cost: 8.64312e+02
Iteration      7000, Cost: 8.62037e+02
Iteration      8000, Cost: 8.59778e+02
Iteration      9000, Cost: 8.57531e+02
w,b found by gradient descent: w: [14.71], b: -0.3834
```



Great! near perfect fit. Notice the values of $\mathbf{w}$ and b printed right above the graph: `w,b found by gradient descent: w: [1.], b: 0.0490` . Gradient descent modified our initial values of $\mathbf{w}, b$ to be (1.0,0.049) or a model of $y = 1 * x_0^2 + 0.049$, very close to our target of $y = 1 * x_0^2 + 1$. If you ran it longer, it could be a better match.

Selecting Features

Above, we knew that an x^2 term was required. It may not always be obvious which features are required. One could add a variety of potential features to try and find the most useful. For example, what if we had instead tried : $y = w_0 x_0 + w_1 x_1^2 + w_2 x_2^3 + b$?

Run the next cells.

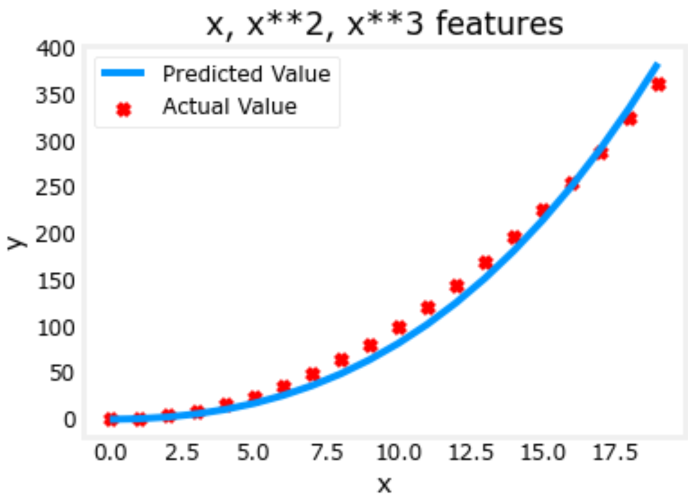
In [4]:

```
# create target data
x = np.arange(0, 20, 1)
y = x**2

# engineer features .
X = np.c_[x, x**2, x**3]   #<-- added engineered feature
```

In [5]:

```
model_w,model_b = run_gradient_descent_feng(X, y, iterations=10000, alpha=1e-7)

plt.scatter(x, y, marker='x', c='r', label="Actual Value"); plt.title("x, x**2, x
plt.plot(x, X@model_w + model_b, label="Predicted Value"); plt.xlabel("x"); plt.y
```

```
Iteration         0, Cost: 1.14029e+03
Iteration      1000, Cost: 3.28539e+02
Iteration      2000, Cost: 2.80443e+02
Iteration      3000, Cost: 2.39389e+02
Iteration      4000, Cost: 2.04344e+02
Iteration      5000, Cost: 1.74430e+02
Iteration      6000, Cost: 1.48896e+02
Iteration      7000, Cost: 1.27100e+02
Iteration      8000, Cost: 1.08495e+02
Iteration      9000, Cost: 9.26132e+01
w,b found by gradient descent: w: [0.08 0.54 0.03], b: 0.0106
```



Note the value of $\mathbf{w}$, `[0.08 0.54 0.03]` and b is `0.0106` .This implies the model after fitting/training is:

$$0.08x + 0.54x^2 + 0.03x^3 + 0.0106$$

Gradient descent has emphasized the data that is the best fit to the x^2 data by increasing the w_1 term relative to the others. If you were to run for a very long time, it would continue to reduce the impact of the other terms.

> Gradient descent is picking the 'correct' features for us by emphasizing its associated parameter

Let's review this idea:

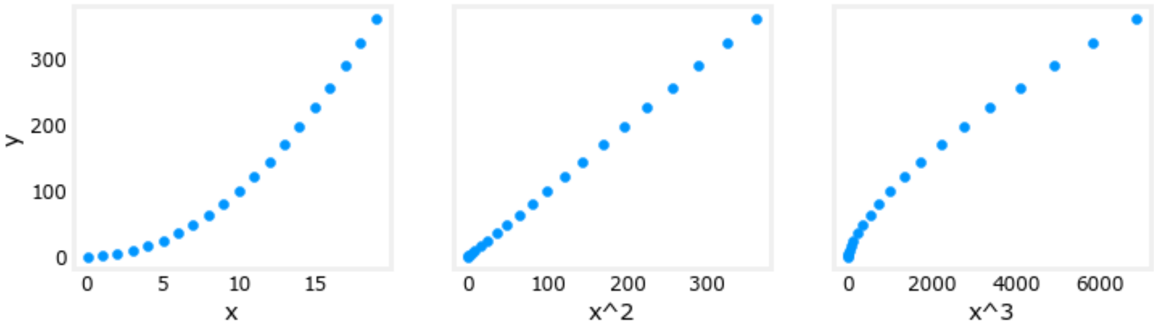
- Intially, the features were re-scaled so they are comparable to each other
- less weight value implies less important/correct feature, and in extreme, when the weight becomes zero or very close to zero, the associated feature is not useful in fitting the model to the data.
- above, after fitting, the weight associated with the x^2 feature is much larger than the weights for x or x^3 as it is the most useful in fitting the data.

An Alternate View

Above, polynomial features were chosen based on how well they matched the target data. Another way to think about this is to note that we are still using linear regression once we have created new features. Given that, the best features will be linear relative to the target. This is best understood with an example.

In [6]:
```python
# create target data
x = np.arange(0, 20, 1)
y = x**2

# engineer features .
X = np.c_[x, x**2, x**3]   #<-- added engineered feature
X_features = ['x','x^2','x^3']
```

In [7]:
```python
fig,ax=plt.subplots(1, 3, figsize=(12, 3), sharey=True)
for i in range(len(ax)):
    ax[i].scatter(X[:,i],y)
    ax[i].set_xlabel(X_features[i])
ax[0].set_ylabel("y")
plt.show()
```



Above, it is clear that the x^2 feature mapped against the target value y is linear. Linear regression can then easily generate a model using that feature.

Scaling features

As described in the last lab, if the data set has features with significantly different scales, one should apply feature scaling to speed gradient descent. In the example above, there is x, x^2 and x^3 which will naturally have very different scales. Let's apply Z-score normalization to our example.

In [8]:
```python
# create target data
x = np.arange(0,20,1)
X = np.c_[x, x**2, x**3]
print(f"Peak to Peak range by column in Raw        X:{np.ptp(X,axis=0)}")

# add mean_normalization
X = zscore_normalize_features(X)
print(f"Peak to Peak range by column in Normalized X:{np.ptp(X,axis=0)}")
```

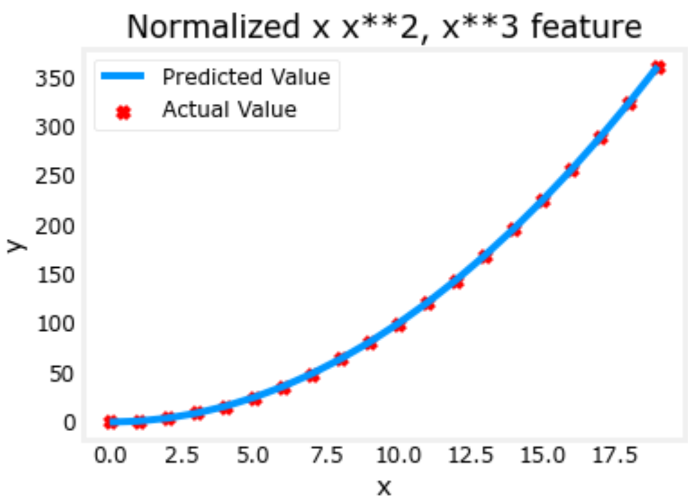
```
Peak to Peak range by column in Raw        X:[  19  361 6859]
Peak to Peak range by column in Normalized X:[3.3   3.18 3.28]
```

Now we can try again with a more aggressive value of alpha:

In [9]:

```
x = np.arange(0,20,1)
y = x**2

X = np.c_[x, x**2, x**3]
X = zscore_normalize_features(X)

model_w, model_b = run_gradient_descent_feng(X, y, iterations=100000, alpha=1e-1)

plt.scatter(x, y, marker='x', c='r', label="Actual Value"); plt.title("Normalized
plt.plot(x,X@model_w + model_b, label="Predicted Value"); plt.xlabel("x"); plt.yl
```

```
Iteration         0, Cost: 9.42147e+03
Iteration     10000, Cost: 3.90938e-01
Iteration     20000, Cost: 2.78389e-02
Iteration     30000, Cost: 1.98242e-03
Iteration     40000, Cost: 1.41169e-04
Iteration     50000, Cost: 1.00527e-05
Iteration     60000, Cost: 7.15855e-07
Iteration     70000, Cost: 5.09763e-08
Iteration     80000, Cost: 3.63004e-09
Iteration     90000, Cost: 2.58497e-10
w,b found by gradient descent: w: [5.27e-05 1.13e+02 8.43e-05], b: 123.5000
```



Feature scaling allows this to converge much faster.
Note again the values of $\mathbf{w}$. The w_1 term, which is the x^2 term is the most emphasized. Gradient descent has all but eliminated the x^3 term.

Complex Functions

With feature engineering, even quite complex functions can be modeled:

In [*]:

```
x = np.arange(0,20,1)
y = np.cos(x/2)

X = np.c_[x, x**2, x**3,x**4, x**5, x**6, x**7, x**8, x**9, x**10, x**11, x**12,
X = zscore_normalize_features(X)

model_w,model_b = run_gradient_descent_feng(X, y, iterations=1000000, alpha = 1e-

plt.scatter(x, y, marker='x', c='r', label="Actual Value"); plt.title("Normalized
plt.plot(x,X@model_w + model_b, label="Predicted Value"); plt.xlabel("x"); plt.yl
```

```
Iteration         0, Cost: 2.24887e-01
Iteration    100000, Cost: 2.31061e-02
Iteration    200000, Cost: 1.83619e-02
Iteration    300000, Cost: 1.47950e-02
```

Congratulations!

In this lab you:

- learned how linear regression can model complex, even highly non-linear functions using feature engineering
- recognized that it is important to apply feature scaling when doing feature engineering

In [ ]:

```

```