Lab 02: Nonlinear Regression and Overfitting

In Lab 01, we explored the construction of linear regression models. Recall the assumptions we make in linear regression:

- $\mathbf{x} \in \mathcal{X} = \mathbb{R}^n$
- $y \in \mathcal{Y} = \mathbb{R}$
- The $\mathbf{x}$ data are drawn i.i.d. from some (unknown) distribution over $\mathcal{X}$
- There is a linear relationship between $\mathbf{x}$ and y with additive constant-variance Gaussian noise, i.e., $y \sim \mathcal{N}(\theta^\top \mathbf{x}, \sigma^2)$, where $\theta \in \mathbb{R}^{n+1}$ is unknown and $\mathbf{x}$ is an $n+1$-dimensional vector augemented with a constant value of 1 as its first element.

Today, we consider what we might do when the fourth assumption, linearity, does not hold. We introduce a particular form of nonlinear regression, *polynomial regression*, in which we account for nonlinear relationships between $\mathbf{x}$ and y by performing nonlinear transformations of the input variables in $\mathbf{x}$.

As an example, if we had a single input variable x, linear regression gives us the hypothesis

$$h_\theta(x) = \theta_0 + \theta_1 x.$$

We can add a new "variable" x^2, which is a nonlinear transformation of the input x:

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2.$$

The important thing to notice here is that although the hypothesis is *nonlinear* in x, allowing us to model a more complex function than ordinary linear regression, the hypothesis is *linear* in θ, allowing us to use the normal equations to find the optimal θ as before.

Polynomial Regression

More generally, polynomial regession is a form of linear regression in which the relationship between the independent variables $\mathbf{x}$ and the dependent variable y is modelled as a polynomial.

For a single input x, the hypothesis in a polynomial regression of degree d is

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_d x^d$$

$$h_\theta(x) = \sum_{i=0}^{d} \theta_i x^i$$

For a multivariate input $\mathbf{x}$, we introduce terms corresponding to every degree-d

combination of factors. For example, if $n = 3$ and $d = 2$, we have $$h_\theta(\mathbf{x}) = \theta_0

```
                   + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3
                   + \theta_4 x_1^2 + \theta_5 x_1 x_2 + \theta_6 x_1
x_3
                   + \theta_7 x_2^2 + \theta_8 x_2 x_3 + \theta_9 x_3^2
.$$
```

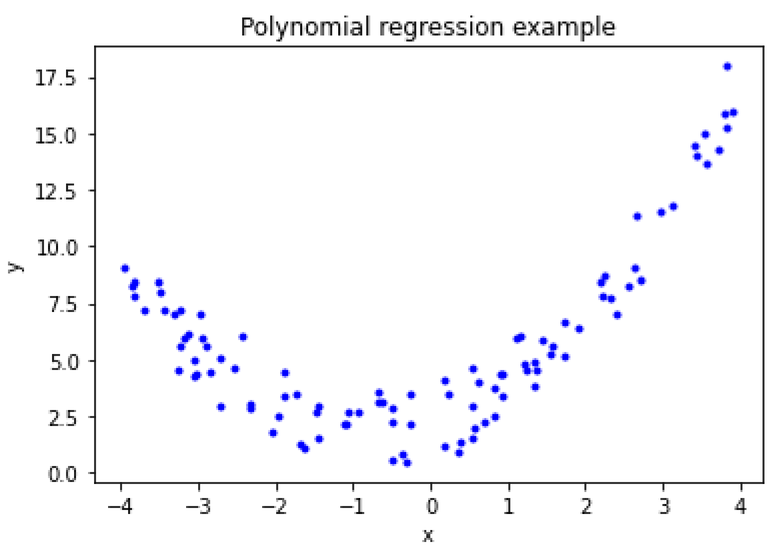
Example 1

Let's take a look at how polynomial regression as compared to simple linear regression model works for data with a simple quadratic nonlinearity. First, we generate 100 observations from a ground truth quadratic function with Gaussian noise:

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
import random

# please do not change the check result will be wrong
np.random.seed(0)
random.seed(0)
```

In [2]:
```
# Generate X
m = 100
X = np.random.uniform(-4, 4, (m,1))

# Generate y
a = 0.7
b = 1
c = 2
y = a * X**2 + b * X + c + np.random.randn(m, 1)
```

In [3]:
```
# Plot
plt.plot(X, y, 'b.')
plt.title('Polynomial regression example')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



Let's use the normal equations to find the θ minimizing $J(\theta)$:

$$\theta = (X^\top X)^{-1} X^\top \mathbf{y}$$

First, we use ordinary linear regression:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

Then, we use polynomial regression with $d = 2$:

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$$

Hypothesis Function

$$h_\theta(\mathbf{x}) = \theta^\top \mathbf{x}.$$

In [4]:
```
def h(X, theta):
    return X.dot(theta)
```

Regression Function

The Regression function can be created from normal equation.

$$\theta = (X^\top X)^{-1} X^\top \mathbf{y}$$

In [5]:
```
def regression(X, y):
    cov = np.dot(X.T, X)
    cov_inv = np.linalg.inv(cov)
    theta = np.dot(cov_inv, np.dot(X.T, y))
    return theta
```

Exercise 1.1 (2 points)

Create function RMSE (root mean squared error)

$$rms_{error} = \sqrt{\frac{\sum_{i=1}^{m}\left(y^{(i)} - \hat{y}^{(i)}\right)^2}{m}}$$

In [6]:
```
def rmse(y, y_pred):
    error = None
    ### BEGIN SOLUTION
    error = np.sqrt(np.dot((y - y_pred).T, y - y_pred) / y.shape[0])
    ### END SOLUTION
    return error
```

In [7]:
```
print(rmse(np.array([1,1.1,2,-1]), np.array([1.1,1.3,1.5,0.1])))

# Test function: Do not remove
assert np.round(rmse(np.array([1,1.1,2,-0.1]), np.array([1.1,1.3,1.5,0.1])), 5) == np.ro
print("success!")
# End Test function
```

```
0.6144102863722254
success!
```

Expect output: 0.6144102863722254

Simple Linear Model

In [8]:
```
# Add intercept column of all 1's
X_aug = np.insert(X, 0, 1, axis=1)

# Print first 5 rows of X
print(X_aug[0:5,:])

# Find optimal parameters
```

```
theta_slr = regression(X_aug, y)

# Predict y
y_pred_slr = h(X_aug, theta_slr)

print('Linear regression RMSE: %f' % rmse(y, y_pred_slr))
```

```
[[ 1.          0.39050803]
 [ 1.          1.72151493]
 [ 1.          0.82210701]
 [ 1.          0.35906546]
 [ 1.         -0.61076161]]
Linear regression RMSE: 3.413803
```

Exercise 1.2 (2 points)

From the simple linear model at above, create another Linear model by using **polynomial model with d=2**.

- Create x data in `X_aug`
- Find θ and input to `theta_pr`

▶ **Hint:**

In [9]:
```
# 1. Add constant column and x^2 column
X_aug = None
# 2. Find optimal parameters
theta_pr = None
### BEGIN SOLUTION
X_aug = np.insert(X, 0, 1, axis=1)
X_aug = np.insert(X_aug, 2, X[:,0]**2, axis=1)

theta_pr = regression(X_aug, y)
### END SOLUTION
```

In [10]:
```
# Predict y
y_pred_pr = h(X_aug, theta_pr)
print(X_aug[0:5,:])
print('Polynomial regression RMSE: %f' % rmse(y, y_pred_pr))

# Test function: Do not remove
assert np.array_equal(np.round(theta_pr.T), np.round([[1.90932595, 1.02311816, 0.7174783
assert np.round(X_aug[10,1] ** 2, 5) == np.round(X_aug[10,2], 5), "X_aug are incorrect"
assert np.round(rmse(y, y_pred_pr) ** 2 * y.shape[0], 5) == np.round(np.dot((y - y_pred_
print("success!")
# End Test function
```

```
[[ 1.          0.39050803  0.15249652]
 [ 1.          1.72151493  2.96361366]
 [ 1.          0.82210701  0.67585993]
 [ 1.          0.35906546  0.12892801]
 [ 1.         -0.61076161  0.37302974]]
Polynomial regression RMSE: 0.986690
success!
```

Expect output \ [[1. 0.39050803 0.15249652]\ [1. 1.72151493 2.96361366]\ [1. 0.82210701 0.67585993]\ [1. 0.35906546 0.12892801]\ [1. -0.61076161 0.37302974]]\ Polynomial regression RMSE: 0.986690

We see that the degree 2 polynomial fit is much better, reducing average error from 3.22 to 0.96.

Here's a plot of the predictions vs. observed data:

Exercise 1.3 (2 points)

Do the **get_prediction function** to predict $\hat{y}$

▶ **Hint:**

In [11]:
```
def get_predictions(x, theta):
    # Change the shape of x to support the function
    x = np.array([x]).T

    y_hat = None
    ### BEGIN SOLUTION
    x = np.insert(x, 0, 1, axis=1)
    while(x.shape[1] < theta.shape[0]):
        x = np.insert(x, x.shape[1], x[:,1] * x[:,-1], axis=1)
    y_hat = h(x, theta)
    ### END SOLUTION

    return y_hat
```

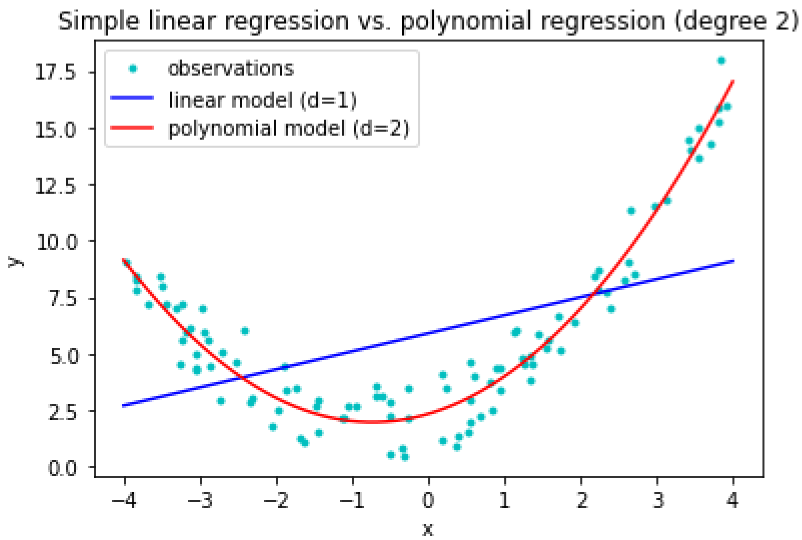
In [12]:
```
x_series = np.linspace(-4, 4, 1000)
y_series_slr = get_predictions(x_series, theta_slr)
y_series_pr = get_predictions(x_series, theta_pr)

print("y_series_slr:", y_series_slr[2:5].T)
print("y_series_pr:", y_series_pr[2:5].T)

# Test function: Do not remove
assert np.round(get_predictions(np.array([1, 9, 2, -9]), theta_slr).T, 5) is not None, "
assert np.round(get_predictions(np.array([1, 1, 0.1, 2]), theta_pr).T, 5) is not None, "
print("success!")
# End Test function
```

```
y_series_slr: [[2.72462183 2.73101513 2.73740842]]
y_series_pr: [[9.0812643  9.04632656 9.01147497]]
success!
```

Expect output:\ y_series_slr: [[2.72462183 2.73101513 2.73740842]]\ y_series_pr: [[9.0812643 9.04632656 9.01147497]]

Plot X, y, and the two regression models

In [13]:
```
plt.plot(X[:,0], y, 'c.', label='observations')
plt.plot(x_series, y_series_slr, 'b-', label='linear model (d=1)')
plt.plot(x_series, y_series_pr, 'r-', label='polynomial model (d=2)')
plt.title('Simple linear regression vs. polynomial regression (degree 2)')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```



Besides RMSE, let's also get the R^2 for our two models. Recall

$$R^2 = 1 - \frac{\sum_{i=1}^{m} \left(y^{(i)} - \hat{y}^{(i)}\right)^2}{\sum_{i=1}^{m} \left(y^{(i)} - \bar{y}^{(i)}\right)^2} \tag{1}$$

Exercise 1.4 (2 points)

Create R^2 from equation above

▶ **Hint:**

In [14]:
```
def r_squared(y, y_pred):
    r_sqr = None
    ### BEGIN SOLUTION
    r_sqr = 1 - np.square(y - y_pred).sum() / np.square(y - y.mean()).sum()
    if r_sqr < 0:
        r_sqr = 0
    ### END SOLUTION
    return r_sqr
```

In [15]:
```
print('Fit of simple linear regression model: %.4f' % r_squared(y, y_pred_slr))
print('Fit of polynomial regression model: %.4f' % r_squared(y, y_pred_pr))

# Test function: Do not remove
assert np.round(r_squared(np.array([1, 2, 3]), np.array([1, 2, 3]))) == np.round(1.0), "
assert np.round(r_squared(y, y_pred_pr), 4) == np.round(0.9353, 4), "r_squared is incorr
print("success!")
# End Test function
```

```
Fit of simple linear regression model: 0.2254
Fit of polynomial regression model: 0.9353
success!
```

Expect output:\ Fit of simple linear regression model: 0.2254\ Fit of polynomial regression model: 0.9353

Another useful analysis is to plot histograms of each model's residuals:

Exercise 1.5 (2 points)

Find error of

- `error_slr` is error from simple linear regression

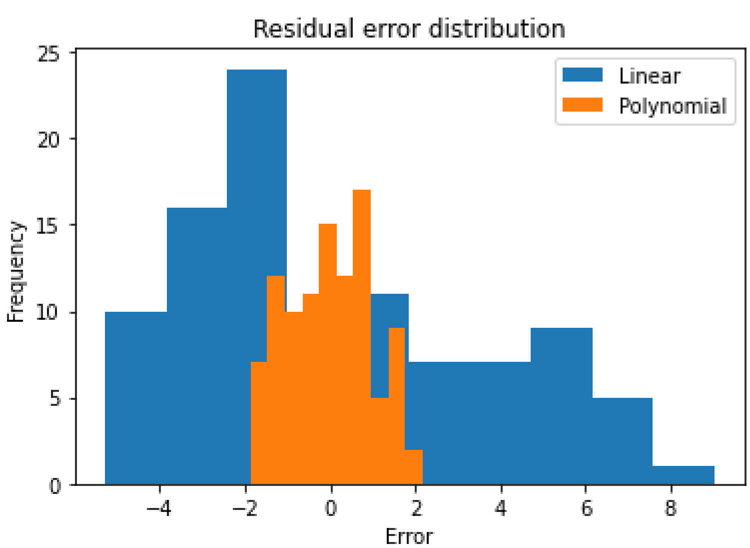
$$error = y - \hat{y}$$

- `error_pr` is error from polynomial linear regression

In [16]:
```
def residual_error(y, y_pred):
    error = None
    ### BEGIN SOLUTION
    error = y - y_pred
    ### END SOLUTION
    return error

error_slr = residual_error(y, y_pred_slr)
error_pr = residual_error(y, y_pred_pr)
```

In [17]:
```
# Plot distribution of residual error for each model
print("error_slr sample:", error_slr[0:5, 0].T)
print("error_pr sample:", error_pr[0:5, 0].T)

plt.hist(error_slr, bins=10, label = 'Linear')
plt.hist(error_pr, bins=10, label = 'Polynomial')
plt.xlabel('Error')
plt.ylabel('Frequency')
plt.title('Residual error distribution')
plt.legend()
plt.show()

# Test function: Do not remove
assert np.array_equal(np.round(get_predictions(np.array([1, 9, 2, -9]), theta_slr).T),
                      np.round([[6.70364883, 13.09055058, 7.50201155, -1.27997835]])), "
assert np.array_equal(np.round(get_predictions(np.array([0, 7, 1.5, -0.3]), theta_pr).T)
                      np.round([[2.34050076, 42.14663283, 5.3284002, 2.10566904]])), "pr
print("success!")
# End Test function
```

```
error_slr sample: [-4.88494741 -0.58280848 -2.8007543  -5.27887921 -2.27906541]
error_pr sample: [-1.49521216  0.67105966  0.15715854 -1.86746535  1.14869785]
```




```
success!
```

Expect output:\ error_slr sample: [-4.88494741 -0.58280848 -2.8007543 -5.27887921 -2.27906541]\ error_pr sample: [-1.49521216 0.67105966 0.15715854 -1.86746535 1.14869785]

The residual plot shows clearly how much better the polynomial model is than the linear model.

Example 2

Next, let's model some monthly sales data from Kaggle using polynomial regression with varying degree.

We will observe the effects of varying the degree of the polynomial regression fit on the prediction accuracy.

However, as models become more complex, we will encounter the issue of *overfitting*, in which a too-powerful model starts to model the noise in the specific training set rather than the overall trend.

To ensure that we're not fitting the noise in the training set, we will split the data into seaparte train and test/validation datasets. The training dataset will consist of 60% of the original observations, and the test dataset will consist of the remaining 40% of the observations.

For various polynomial degrees, we'll estimate optimal parameters θ, then we'll use the test dataset to measure accuracy of the optimized model.

In [18]:

```
# Import CSV
data = np.genfromtxt('MonthlySales_data.csv',delimiter = ',', dtype=str)

# Extract headers
headers = data[0,:]
print("Headers:", headers)

# Extract raw data
data = np.array(data[1:,:], dtype=float);
mean = np.mean(data,axis=0)
std = np.std(data,axis=0)
data_norm = (data-mean)/std

# Extract y column from raw data
y_index = np.where(headers == 'sale amount')[0][0];
y_data = data[:,y_index];

# Extract x column (just the month) from raw data
month_index = np.where(headers == 'month')[0][0]
# print(year_index, month_index)
X_data = data[:,[month_index]];
m = X_data.shape[0]
n = X_data.shape[1]
X_data = X_data.reshape(m, n)

print('Extracted %d monthly sales records' % m)
print(X_data.shape)
print(y_data.shape)
```

```
Headers: ['year' 'month' 'sale amount']
Extracted 240 monthly sales records
(240, 1)
(240,)
```

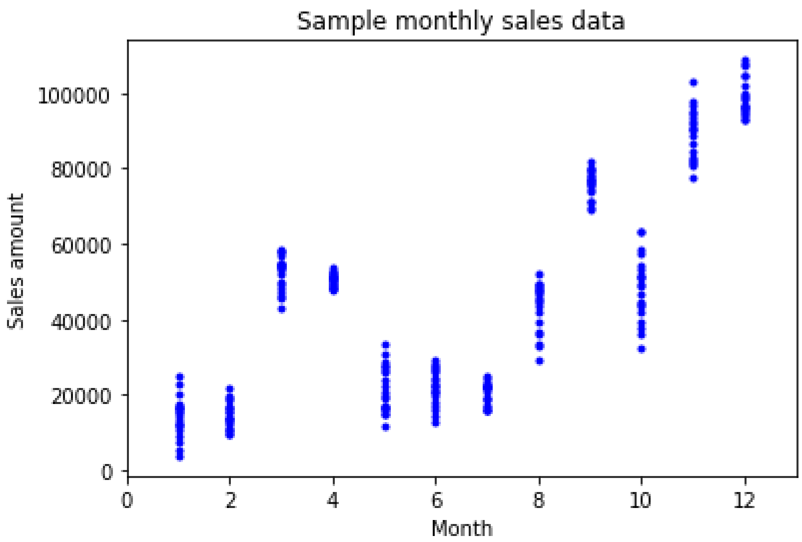
Plot the data

Plot 3D by using Axes3D

In [19]:

```
# Plot the data
fig = plt.figure()
xx1 =X_data[:,0]
zz1 =y_data

plt.plot(xx1, zz1, 'b.')
```

```
plt.xlim(0, 13)
plt.xlabel('Month')
plt.ylabel('Sales amount')
plt.title('Sample monthly sales data')
plt.show()
```



Exercise 1.6 (2 points)

Partion `X_data` and `y_data` into training and test datasets

- Do train set as 60% of all data
- Other are test set
- dataset must be shuffle

You can use [random.shuffle](https://www.w3schools.com/python/ref_random_shuffle.asp) to shuffle index of dataset

In [20]:

```
percent_train = .6

def partition(X, y, percent_train):
    # 1. create index list
    idx = np.arange(0,y.shape[0])
    random.seed(1412)   # just make sure the shuffle always the same please do not remov
    # do yourself follow the instruction
    # 2. shuffle index
    # 3. Create train/test index
    # 4. Separate X_Train, y_train, X_test, y_test
    X_train = None
    y_train = None
    X_test = None
    y_test = None
    ### BEGIN SOLUTION
    random.shuffle(idx)

    m_train = int(y.shape[0] * percent_train)
    train_idx = idx[0:m_train]
    test_idx = idx[m_train:y.shape[0]+1]

    X_train = X[train_idx]
    X_test = X[test_idx]

    y_train = y[train_idx]
    y_test = y[test_idx]
```

```
    ### END SOLUTION

    return idx, X_train, y_train, X_test, y_test
```

In [21]:
```
idx, X_train, y_train, X_test, y_test = partition(X_data, y_data, percent_train)
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
print(idx[5:9])

# Test function: Do not remove
assert not np.array_equal(np.round(X_data[0:144, :], 3), np.round(X_train,3)), "X_train
assert not np.array_equal(np.round(X_data[144:, :], 3), np.round(X_test,3)), "X_test mus
assert not np.array_equal(np.round(y_data[0:144], 3), np.round(y_train,3)), "y_train mus
assert not np.array_equal(np.round(y_data[144:], 3), np.round(y_test,3)), "y_test must b
assert np.array_equal(idx[5:9], [26, 75, 51, 162])
print("success!")
# End Test function
```

```
(144, 1)
(144,)
(96, 1)
(96,)
[ 26  75  51 162]
success!
```

Expect output:\ (144, 1)\ (144,)\ (96, 1)\ (96,)\ [26 75 51 162]

Exercise 1.7 (2 points)

Create `x_polynomial` function

$$X = [1, x, x^2, \ldots, x^n]$$

when n is number of polynomial set

In [22]:
```
def x_polynomial(x, n):
    X = None
    ### BEGIN SOLUTION
    X = np.ones((x.shape[0], 1))
    for i in range(n):
        X = np.concatenate((X,x**(i+1)), axis = 1)
    ### END SOLUTION
    return X
```

In [23]:
```
print(x_polynomial(np.array([[3],[2]]), 5))
print(x_polynomial(np.array([[3],[2]]), 5).shape)

Xi_train = x_polynomial(X_train, 1)
Xi_test = x_polynomial(X_test, 1)

# Test function: Do not remove
assert x_polynomial(np.array([[2],[3]]), 5).shape[1] == 5 + 1, "Size of polynomial incor
assert np.array_equal(np.round(x_polynomial(np.array([[2],[3]]), 5), 3),
                      np.round([[1, 2, 4, 8, 16, 32], [1, 3, 9, 27, 81, 243]],3)), "Poly
print("success!")
# End Test function
```

```
[[  1.   3.   9.  27.  81. 243.]
 [  1.   2.   4.   8.  16.  32.]]
```

```
(2, 6)
success!
```

Expect output:\ [[1. 3. 9. 27. 81. 243.]\ [1. 2. 4. 8. 16. 32.]]\ (2, 6)

Exercise 1.8 (2 points)

Create `cost` function (J)

In [24]:
```
def cost(theta,X,y):
    J = None
    ### BEGIN SOLUTION
    J = 1 / 2 / X.shape[0] * (h(X,theta)-y).T.dot(h(X,theta)-y)
    ### END SOLUTION
    return J
```

In [25]:
```
# calculate theta
theta = regression(Xi_train, y_train)

# calculate cost in train
J_train = cost(theta, Xi_train, y_train)

y_pred_test = h(Xi_test, theta)
J_test = cost(theta, Xi_test, y_test)

print("J_train:", J_train)
print("J_test:", J_test)

# Test function: Do not remove
assert type(J_train) == np.float64, "Cost function size must be 1"
assert np.round(J_train, 3) == np.round(174395635.44334993, 3), "Cost function in train
assert np.round(J_test, 3) == np.round(196382485.91395777, 3), "Cost function in test se
print("success!")
# End Test function
```

```
J_train: 174395635.44334993
J_test: 196382485.91395798
success!
```

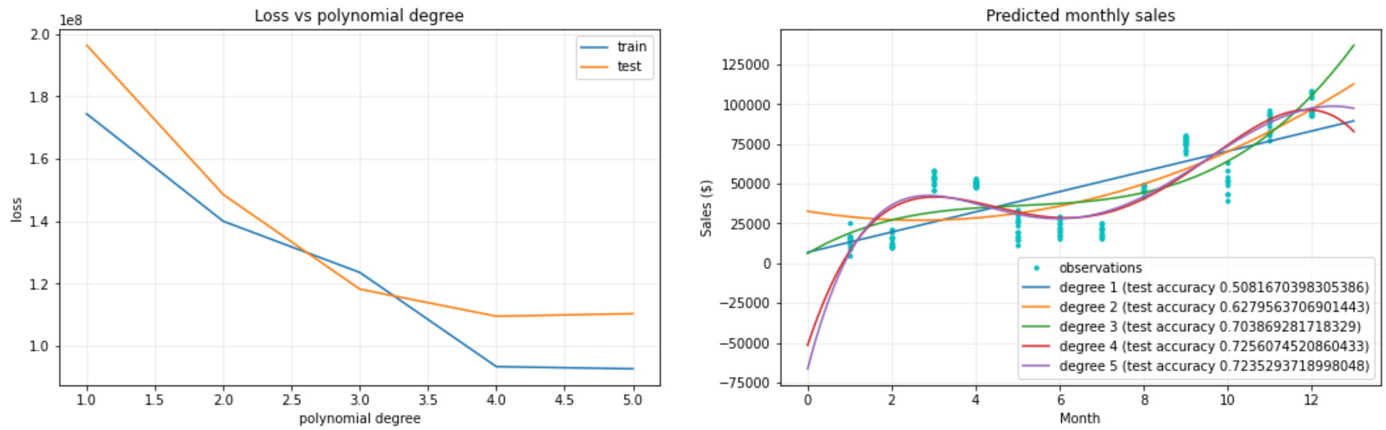
Expect output:\ J_train: 174395635.44334993\ J_test: 196382485.91395777

Mixed together

Build models of degree 1 to max_degree

In [26]:
```
max_degree = 5

J_train = np.zeros(max_degree)
J_test = np.zeros(max_degree)

# Initalize plots for predictions and loss
fig, ax = plt.subplots(1,2)
fig.set_figheight(5)
fig.set_figwidth(20)
fig.subplots_adjust(left=.2, bottom=None, right=None, top=None, wspace=.2, hspace=.2)
plt1 = plt.subplot(1,2,1)
plt2 = plt.subplot(1,2,2)
plt2.plot(X_train, y_train, 'c.', label='observations')

for i in range(1, max_degree+1):
```

```
    # Fit model on training data and get cost for training and test data
    Xi_train = x_polynomial(X_train, i)
    Xi_test = x_polynomial(X_test, i);
    theta = regression(Xi_train, y_train)
    J_train[i-1] = cost(theta, Xi_train, y_train)
    y_pred_test = h(Xi_test, theta)
    J_test[i-1] = cost(theta, Xi_test, y_test)

    # Plot
    x_series = np.linspace(0, 13, 1000)
    y_series = get_predictions(x_series, theta)
    plt2.plot(x_series, y_series, '-', label='degree ' + str(i) + ' (test accuracy ' + s

plt1.plot(np.arange(1, max_degree + 1, 1), J_train, '-', label='train')
plt1.plot(np.arange(1, max_degree + 1, 1), J_test, '-', label='test')
plt1.set_title('Loss vs polynomial degree')
plt1.set_xlabel('polynomial degree')
plt1.set_ylabel('loss')
plt1.grid(axis='both', alpha=.25)
plt1.legend()

plt2.set_title('Predicted monthly sales')
plt2.set_xlabel('Month')
plt2.set_ylabel('Sales ($)')
plt2.grid(axis='both', alpha=.25)
plt2.legend()
plt.show()
```



Take some time to undserstand the code. You should see that training loss falls as the degree of the polynomial increases. However, depending on your particular train/test split of the data, you may observe at $d = 4$ or $d = 5$ that test loss starts to increase. This is the phenomenon of overfitting!

If you don't see any evidence of overfitting, you might regenerate the test/train splits (rerun the previous cell as well as the training cell).

You may also increase max_degree to a point. However, without normalization of the data, the matrix $\mathbf{X}^\top \mathbf{X}$ we invert in the solution to the normal equations will become numerically close to singularity, and you will observe unstable solutions. The result is usually a parameter vector θ that is suboptimal that gives poor results on both the training set and test set.

If you want to evaluate the numerial stability of the correlation matrix $\mathbf{X}^\top \mathbf{X}$, try this code:

In [27]:
```
corr = Xi_train.T.dot(Xi_train)
print('Correlation matrix:', corr)
cond = np.linalg.cond(corr)
print('Condition number: %0.5g' % cond)
```

```
Correlation matrix: [[1.44000000e+02 9.34000000e+02 7.73800000e+03 7.24420000e+04
```

```
  7.25962000e+05 7.58679400e+06]
 [9.34000000e+02 7.73800000e+03 7.24420000e+04 7.25962000e+05
  7.58679400e+06 8.15402980e+07]
 [7.73800000e+03 7.24420000e+04 7.25962000e+05 7.58679400e+06
  8.15402980e+07 8.94004282e+08]
 [7.24420000e+04 7.25962000e+05 7.58679400e+06 8.15402980e+07
  8.94004282e+08 9.94854740e+09]
 [7.25962000e+05 7.58679400e+06 8.15402980e+07 8.94004282e+08
  9.94854740e+09 1.11986452e+11]
 [7.58679400e+06 8.15402980e+07 8.94004282e+08 9.94854740e+09
  1.11986452e+11 1.27211760e+12]]
Condition number: 6.5793e+12
```

Read more about the condition number on [Wikipedia](https://en.wikipedia.org/wiki/Condition_number). Roughly speaking, if our condition number is 10^k, we may lose up to k digits of accuracy in the inverse of the matrix. If $k = 12$ as above, then we have an extremely poorly conditioned problem, because the IEEE 64 bit floating point representation of reals we're using in Python only has around 16 digits of accuracy (see [Wikipedia's page on IEEE floating point numbers](https://en.wikipedia.org/wiki/IEEE_754)).

One way to improve the numerical conditioning of the problem is normalization. If the values of the variable's we're correlating in this matrix have relatively small positive and negative values, the condition number of the correlation matrix will be much smaller and you'll get better results.

Take some time to undserstand the code. Depending on your random test/train split, you should see that training loss falls as the degree of the polynomial increases. However, you may observe at some point that test loss starts to increase, and you may see some very strange behavior of the model function beyond the range 1-12. If not, go ahead and increase the variable `max_degree` until you see an increase in test loss. This is the phenomenon of overfitting!

In-lab exercise

During the lab session, you should perform the following exercises:

1. Add the `year` variable from the monthly sales dataset to your simple linear regression model and quantify whether including it improves test set performance. Show the observations and predictions in a 3D surface plot.
2. Develop polynomial regression models of degree 2 and 3 based on the two input variables. Show results as 3D surface plots and discuss whether you observe overfitting or not.

Exercise 2.1 (2 points)

Import **MonthlySales_data.csv** file into `data_csv` and extract **headers** at the top of `data_csv` into `headers_csv`

In [28]:
```
headers_csv = None
data_csv = None
### BEGIN SOLUTION
data_csv = np.genfromtxt('MonthlySales_data.csv',delimiter = ',', dtype=str)
headers_csv = data_csv[0,:]
data_csv = np.array(data_csv[1:,:], dtype=float)
### END SOLUTION
```

In [29]:
```
print(headers_csv)
print(data_csv[:5])
```

```
# Test function: Do not remove
assert type(data_csv[0,0]) == np.float64, "You must remove the header"
assert headers_csv.shape[0] == 3, "Headers must have 3 values"
assert type(headers_csv[0]) == np.str_, "Headers must be string"
assert np.round(data_csv[30, 2], 3) == np.round(2.222027e+04, 3), "Data is incorrect"
print("success!")
# End Test function
```

```
['year' 'month' 'sale amount']
[[1.995000e+03 1.000000e+00 1.238611e+04]
 [1.995000e+03 2.000000e+00 1.532923e+04]
 [1.995000e+03 3.000000e+00 5.800217e+04]
 [1.995000e+03 4.000000e+00 5.130520e+04]
 [1.995000e+03 5.000000e+00 1.645247e+04]]
success!
```

Expect output:\ ['year' 'month' 'sale amount']\ [[1.995000e+03 1.000000e+00 1.238611e+04]\ [1.995000e+03 2.000000e+00 1.532923e+04]\ [1.995000e+03 3.000000e+00 5.800217e+04]\ [1.995000e+03 4.000000e+00 5.130520e+04]\ [1.995000e+03 5.000000e+00 1.645247e+04]]

Exercise 2.2 (2 points)

- Extract **sale amount** column into `y_csv`
- Extract **year** and **month** columns into `X_csv` by use **year** at column index 0 and **month** at column index 1

In [30]:

```
# Extract y column from raw data
# Extract x column (year and month) from raw data
y_csv = None
X_csv = None
### BEGIN SOLUTION
y_index = np.where(headers_csv == 'sale amount')[0][0];
y_csv = data[:,y_index]

x_year = np.where(headers_csv == 'year')[0][0];
x_month = np.where(headers_csv == 'month')[0][0];
X_csv = data[:,[x_year, x_month]]
### END SOLUTION
```

In [31]:

```
m = X_csv.shape[0]
n = X_csv.shape[1]
X_csv = X_csv.reshape(m, n)
print('Extracted %d sales records' % m)
print('number of x set:', n)

# Test function: Do not remove
assert m == 240, "Sales records incorrect"
assert n == 2, "Need to extract 2 columns of X set"
assert np.max(X_csv[:,0]) == 2014 and np.min(X_csv[:,0]) == 1995, "Year is filled wrong
assert np.max(X_csv[:,1]) == 12 and np.min(X_csv[:,1]) == 1, "Month is filled wrong colu
print("success")
# End Test function
```

```
Extracted 240 sales records
number of x set: 2
success
```

Expect output:\ Extracted 240 sales records\ number of x set: 2

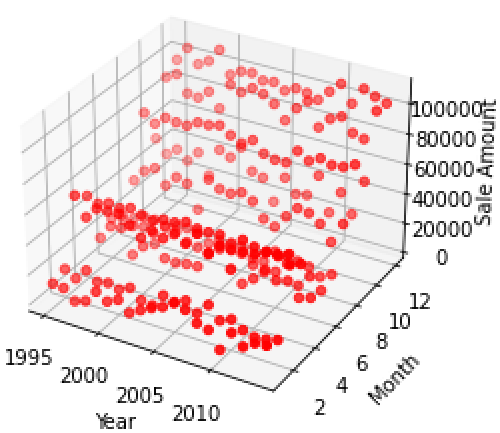
Exercise 2.3 (2 points)

- Plot 3D graph using `mpl_toolkits.mplot3d`

▶ **Hint:**

In [32]:

```
# Plot the data
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
# 1. Set plot graph as 3D
ax = fig.add_subplot(projection='3d')

# 2. Extract data
# extract year at x-axis
# extract month at y-axis
# extract sale amount at z-axis
x_year = None
y_month = None
z_sale = None

# 3. plot by using scatter

# 4. set x, y, z label
### BEGIN SOLUTION
x_year =X_csv[:,0]
y_month =X_csv[:,1]
z_sale =y_csv

ax.scatter(x_year, y_month, z_sale, c='r', marker='o')

ax.set_xlabel('Year')
ax.set_ylabel('Month')
ax.set_zlabel('Sale Amount')
### END SOLUTION

plt.show()
```



In [33]:

```
# Test function: Do not remove
assert ax.get_xbound()[1] >= 2014 and ax.get_xbound()[0] <= 1995, "Year is filled wrong
assert ax.get_ybound()[1] >= 12 and ax.get_ybound()[0] <= 1, "Month is filled wrong colu
assert ax.get_zbound()[1] >= 100000 and ax.get_zbound()[0] <= 0, "Year is filled wrong c
assert 'year' in ax.get_xlabel().lower(), "x-axis label is incorrect"
assert 'month' in ax.get_ylabel().lower(), "y-axis label is incorrect"
assert 'sale' in ax.get_zlabel().lower(), "y-axis label is incorrect"
print("success")
# End Test function
```

```
success
```

Expect output:\

Exercise 2.4 (2 points)

Extract data to 60% of training set and 40% of test set with shuffle

- You can use `partitions` function or create your new function and make sure that you must use `random.seed(1412)` in the code (to make sure that the result will be the same as the expect result)
- Please use `idx, X_train, y_train, X_test, y_test` for the answer result.

In [34]:

```
idx, X_train, y_train, X_test, y_test = None, None, None, None, None
### BEGIN SOLUTION
percent_train_csv = 0.6
idx, X_train, y_train, X_test, y_test = partition(X_csv, y_csv, percent_train_csv)
### END SOLUTION
```

In [35]:

```
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
print(idx[5:9])

# Test function: Do not remove
assert not np.array_equal(np.round(X_csv[0:144, :], 3), np.round(X_train,3)), "X_train m
assert not np.array_equal(np.round(X_csv[144:, :], 3), np.round(X_test,3)), "X_test must
assert not np.array_equal(np.round(y_csv[0:144], 3), np.round(y_train,3)), "y_train must
assert not np.array_equal(np.round(y_csv[144:], 3), np.round(y_test,3)), "y_test must be
assert np.array_equal(idx[5:9], [26, 75, 51, 162])
print("success!")
# End Test function
```

```
(144, 2)
(144,)
(96, 2)
(96,)
[ 26  75  51 162]
success!
```

Expect output:\ (144, 2)\ (144,)\ (96, 2)\ (96,)\ [26 75 51 162]

Exercise 2.5 (2 points)

- Create `Xi_train, Xi_Test`. X sets must be polynomial of $n = 1$.
- Calculate `theta`
- Calculate `y_pred_test`
- Calculate cost function J from train and test set

In [36]:

```
Xi_train, Xi_test = None, None
theta = None
y_pred_test = None
J_train, J_test = None, None

### BEGIN SOLUTION
Xi_train = x_polynomial(X_train, 1)
Xi_test = x_polynomial(X_test, 1)
theta = regression(Xi_train, y_train)
```

```
J_train = cost(theta, Xi_train, y_train)
y_pred_test = h(Xi_test, theta)
J_test = cost(theta, Xi_test, y_test)
### END SOLUTION
```

In [37]:
```
print("Xi_train[:3]:", np.round(Xi_train[:3], 2))
print("Xi_test[:3]:", np.round(Xi_test[:3], 2))
print("theta:", theta)
print("y_pred_test[:5]:", np.round(y_pred_test[:5].T, 2))
print("J_train:", J_train)
print("J_test:", J_test)

# Test function: Do not remove
assert np.array_equal(np.round(theta, 3), np.round([5.74503812e+05, -2.83158807e+02, 6.3
assert np.round(J_train, 0) == np.round(172968387.44854635, 0), "Train cost is incorrect
assert np.round(J_test, 0) == np.round(204275431.7643744, 0), "Test cost is incorrect"
print("success")
# End Test function
```

```
Xi_train[:3]: [[1.000e+00 2.003e+03 1.100e+01]
 [1.000e+00 2.004e+03 3.000e+00]
 [1.000e+00 2.002e+03 6.000e+00]]
Xi_test[:3]: [[1.000e+00 2.008e+03 1.000e+01]
 [1.000e+00 1.997e+03 5.000e+00]
 [1.000e+00 2.006e+03 1.100e+01]]
theta: [ 5.74503812e+05 -2.83158807e+02  6.37579347e+03]
y_pred_test[:5]: [69678.86 40914.64 76620.97 79169.4  48852.53]
J_train: 172968387.44854638
J_test: 204275431.76525488
success
```

Expect output:\ Xi_train[:3]: [[1.000e+00 2.003e+03 1.100e+01]\ [1.000e+00 2.004e+03 3.000e+00]\ [1.000e+00 2.002e+03 6.000e+00]]\ Xi_test[:3]: [[1.000e+00 2.008e+03 1.000e+01]\ [1.000e+00 1.997e+03 5.000e+00]\ [1.000e+00 2.006e+03 1.100e+01]]\ theta: [5.74503812e+05 -2.83158807e+02 6.37579347e+03]\ y_pred_test[:5]: [69678.86 40914.64 76620.97 79169.4 48852.53]\ J_train: 172968387.44854635\ J_test: 204275431.7643744

Exercise 2.6 (2 points)

Create **mesh grid point** to plot **surface**

▶ **Hint:**

In [38]:
```
# 1. Create mesh grid x_mesh, y_mesh
#    Hint: this step do in input X dataset only (year, and month series)
# 1.1 use numpy.linspace() to generate x_series and y_series
#     - do x_series in between min(year) - 1 to max(year) + 1
#     - do y_series in between min(month) - 1 to max(month) + 1
#     - num_linspace = 100
# 1.2 use numpy.meshgrid() to generate x_mesh, and y_mesh
# 1.3 merge x_mesh and y_mesh to be xy_mesh
num_linspace = 100
x_series, y_series = None, None
x_mesh, y_mesh, xy_mesh = None, None, None

# 2. predict output from xy_mesh to be z_series
#    Hint: use mesh_predictions function instead of get_prediction
def mesh_predictions(x, theta):
    x = np.insert(x, 0, 1, axis=x.ndim-1)
    theta = theta.reshape(-1,1)
    y = x@theta
```

```
    return y
z_series = None

### BEGIN SOLUTION
y_series = np.linspace(0, 13, num_linspace)
x_series = np.linspace(1995,2013,num_linspace)
x_mesh, y_mesh = np.meshgrid(x_series, y_series)
xy_mesh = np.append(x_mesh[...,np.newaxis],y_mesh[...,np.newaxis],axis=2)

z_series = mesh_predictions(xy_mesh, theta).reshape(100,100)
### END SOLUTION
```

In [39]:
```
print("xy_mesh.shape", xy_mesh.shape)
print("z_series.shape", z_series.shape)
#print("xy_mesh", xy_mesh)
#print("z_series", z_series)

# Test function: Do not remove
assert xy_mesh.shape == (num_linspace, num_linspace, 2), "mesh shape is incorrect"
assert z_series.shape == (num_linspace, num_linspace), "z_series is incorrect"
print("success")
# End Test function
```

```
xy_mesh.shape (100, 100, 2)
z_series.shape (100, 100)
success
```

Expect output:\ xy_mesh.shape (100, 100, 2)\ z_series.shape (100, 100)

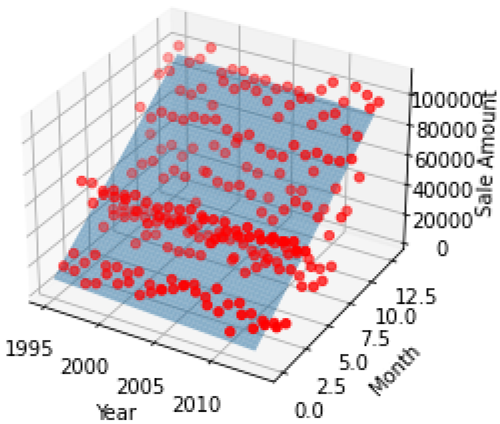
Exercise 2.6 (2 points)

Plot **surface** of theta with the dataset points from xy_mesh and z_series above.

▶ **Hint:**

In [40]:
```
fig = plt.figure()
# 1. Set plot graph as 3D
ax = fig.add_subplot(projection='3d')

# 2. Extract data
# extract year at x-axis
# extract month at y-axis
# extract sale amount at z-axis
x_year = None
y_month = None
z_sale = None

# 3. plot by using scatter
# 4. set x, y, z label
#    Hint: In these 3, 4 steps, you can copy Exercise 2.3
# 5. Plot surface from x_mesh, y_mesh, and z_series

### BEGIN SOLUTION
x_year =X_csv[:,0]
y_month =X_csv[:,1]
z_sale =y_csv

ax.scatter(x_year, y_month, z_sale, c='r', marker='o')

ax.set_xlabel('Year')
ax.set_ylabel('Month')
ax.set_zlabel('Sale Amount')
```

```
ax.plot_surface(x_mesh,y_mesh,z_series,alpha=0.5)
### END SOLUTION

plt.show()
```



In [41]:
```
# Test function: Do not remove
assert ax.get_xbound()[1] >= 2014 and ax.get_xbound()[0] <= 1995, "Year is filled wrong
assert ax.get_ybound()[1] >= 12 and ax.get_ybound()[0] <= 1, "Month is filled wrong colu
assert ax.get_zbound()[1] >= 100000 and ax.get_zbound()[0] <= 0, "Year is filled wrong c
assert 'year' in ax.get_xlabel().lower(), "x-axis label is incorrect"
assert 'month' in ax.get_ylabel().lower(), "y-axis label is incorrect"
assert 'sale' in ax.get_zlabel().lower(), "y-axis label is incorrect"
print("success")
# End Test function
```

```
success
```

Expect result:

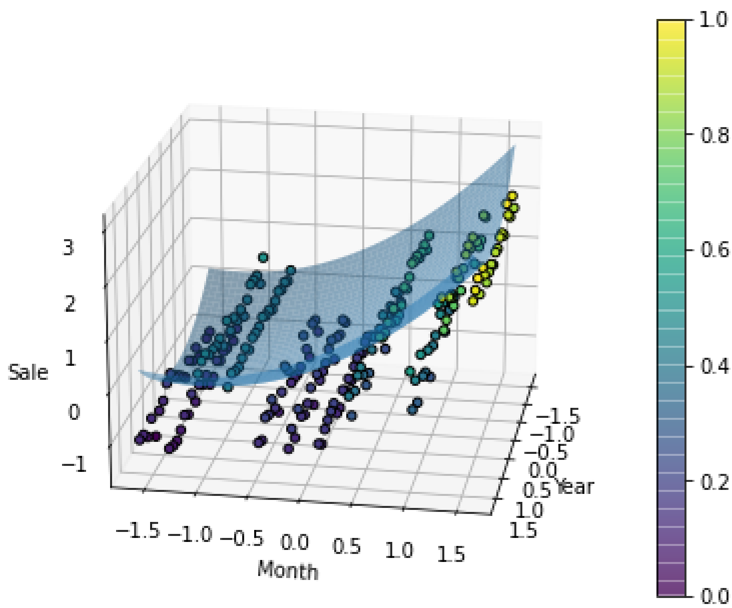
Exercise 2.7 (20 points)

Develop polynomial regression models of degree 2 and 3 based on the two input variables. Show results as 3D surface plots and discuss whether you observe overfitting or not.

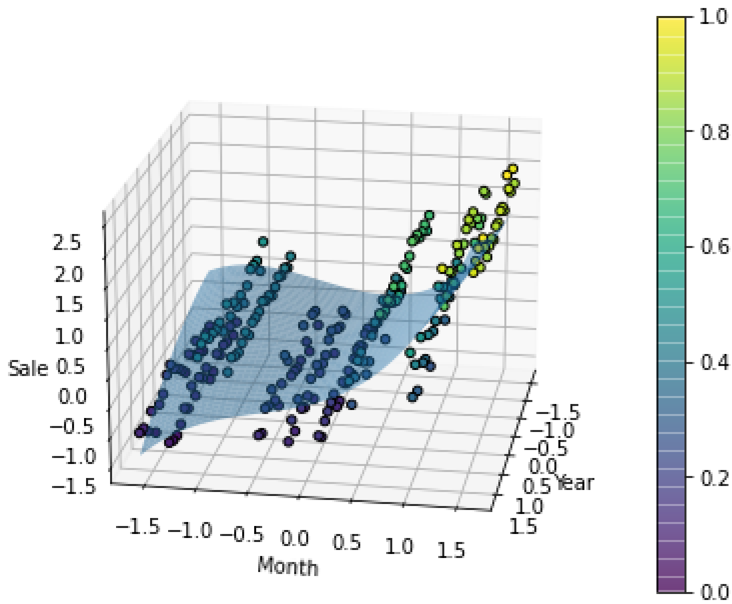
In [42]:
```
data_csv = (data-np.mean(data, axis = 0))/np.std(data, axis = 0)
y_label = 'sale amount';
y_index = np.where(headers == y_label)[0][0];
y = data_csv[:,y_index];
X = data_csv[:,0:y_index];
m = data_norm.shape[0]

percent_train = .6
random.shuffle(idx)

m_train = int(m * percent_train)
train_idx = idx[0:m_train]
test_idx = idx[m_train:m+1]

X_train = data_csv[train_idx, 0:y_index];
X_test = data_csv[test_idx, 0:y_index];
y_train = data_csv[train_idx, y_index];
y_test = data_csv[test_idx, y_index];

#==============================

# Polynomial regression model d=2, 3
```

```python
for i in range(2):
    Xi_train = x_polynomial(X_train, i + 2)
    Xi_test = x_polynomial(X_test, i + 2)

    theta = regression(Xi_train, y_train)
    J_train = cost(theta, Xi_train, y_train)
    y_pred_test = h(Xi_test, theta)
    J_test = cost(theta, Xi_test, y_test)

    # 3D plot
    print("3D plot: Polynomial degree 2","\n")
    from mpl_toolkits.mplot3d import Axes3D

    fig = plt.figure()
    ax = Axes3D(fig)
    x_year = data_csv[:, 0]
    y_month = data_csv[:, 1]
    z_sale = data_csv[:, 2]

    # 3. plot by using scatter
    p = ax.scatter(x_year,y_month, z_sale,edgecolors='black', c=data_norm[:,2],alpha=1)

    # 4. set x, y, z label
    ax.set_xlabel('Year')
    ax.set_ylabel('Month')
    ax.set_zlabel('Sale')

    # plot observation
    x_series = np.linspace(min(data_csv[:,0]), max(data_csv[:,0]),len(y_csv))
    y_series = np.linspace(min(data_csv[:,1]), max(data_csv[:,1]),len(y_csv))

    x_mesh, y_mesh = np.meshgrid(x_series, y_series)

    if i == 0: # degree 2
        yy =(theta[0] +theta[1]*x_mesh.T+theta[2]*y_mesh+theta[3]*(x_mesh*y_mesh)+theta[
    else: # degree 3
        yy=(theta[0]+theta[1]*(x_mesh+y_mesh).T+theta[2]*x_mesh*y_mesh +theta[3]*x_mesh*

    p = ax.plot_surface(x_mesh, y_mesh,yy,alpha=0.5)
    ax.view_init(elev=20, azim=10)
    plt.colorbar(p)
    plt.show()
```

```
3D plot: Polynomial degree 2
```




```
3D plot: Polynomial degree 2
```



In [ ]:

Exercise 3 Take-home exercise (50 points)

Using the dataset you played with for the take-home exercise in Lab 01, perform the same analysis. You won't be able to visualize the model well, as you will have more than two inputs, but try to give some idea of the performance of the model visually. Also, depending on the number of variables in your dataset, you may not be able to increase the polynomial degree beyond 2. Discuss whether the polynomial model is better than the linear model and whether you observe overfitting.

Write all code in youre new file

To turn in

Before the next lab, turn in a brief report in the form of a Jupyter notebook documenting your work in the lab and the take-home exercise, along with your observations and discussion.

In [ ]: