In [226]:
```
from download_data import download_data
import numpy as np
import matplotlib.pyplot as plt
from GD import gradientDescent
from dataNormalization import rescaleMatrix
```

## Step 1: Loading the data and normalisation using Min-max

In [229]:
```
from sklearn.preprocessing import MinMaxScaler
```

For normalising the data, we are using MinMax scaler. It a pre-processing technique in Python's scikit-learn library that transforms the features of a dataset by scaling each feature to a given range, usually between 0 and 1. This method is used to normalize the features in a dataset so that they have a consistent scale and to reduce the effect of outliers.

In [230]:
```
min_max_scaler = MinMaxScaler(feature_range=(0,1))
df = pd.read_csv("/Users/snehilshandilya/Desktop/sat.csv", header=0)
df = df[["math_SAT","verb_SAT","univ_GPA"]]
df.columns = ["math_SAT","verb_SAT","univ_GPA"]
df["univ_GPA"] = df["univ_GPA"].astype(str)
x = df["univ_GPA"].map(lambda x: float(x.rstrip(';')))
X = df[["math_SAT","verb_SAT"]]
X = np.array(X)
X = min_max_scaler.fit_transform(X)
Y = df["univ_GPA"].map(lambda x: float(x.rstrip(';')))
Y = np.array(Y)
```

In [231]:
```
df
```

Out[231]:

| | math_SAT | verb_SAT | univ_GPA |
|---|---|---|---|
| **0** | 643 | 589 | 3.52 |
| **1** | 558 | 512 | 2.91 |
| **2** | 583 | 503 | 2.4 |
| **3** | 685 | 602 | 3.47 |
| **4** | 592 | 538 | 3.47 |
| **...** | ... | ... | ... |
| **100** | 605 | 590 | 3.64 |
| **101** | 692 | 683 | 3.42 |
| **102** | 680 | 692 | 3.25 |
| **103** | 617 | 503 | 2.76 |
| **104** | 516 | 528 | 3.41 |

105 rows × 3 columns

In [287]:
```
X
```

Out[287]:
```
array([[0.62871287, 0.43253968],
       [0.20792079, 0.12698413],
       [0.33168317, 0.09126984],
       [0.83663366, 0.48412698],
       [0.37623762, 0.23015873],
       [0.22772277, 0.02380952],
       [0.28217822, 0.26984127],
       [0.21287129, 0.22222222],
       [0.17821782, 0.40873016],
       [0.5       , 0.44047619],
       [0.83168317, 0.67063492],
       [0.25742574, 0.44444444],
       [0.43564356, 0.4047619 ],
       [0.50990099, 0.57142857],
       [0.62376238, 0.5515873 ],
       [0.82673267, 0.64285714],
       [0.92574257, 0.80952381],
       [0.97029703, 0.68253968],
       [0.23762376, 0.08333333],
       [0.2029703 , 0.27380952],
       [0.37128713, 0.41269841],
       [0.41089109, 0.32539683],
       [0.45049505, 0.5515873 ],
       [0.50990099, 0.30952381],
       [0.91089109, 0.95634921],
       [1.        , 1.        ],
       [0.31683168, 0.23015873],
       [0.22772277, 0.10714286],
       [0.82673267, 0.66666667],
       [0.9950495 , 0.96825397],
       [0.91584158, 0.92857143],
       [0.86633663, 0.80952381],
       [0.98019802, 0.8968254 ],
       [0.85643564, 0.76587302],
       [0.18811881, 0.10714286],
       [0.23762376, 0.25      ],
       [0.75247525, 0.49206349],
       [0.86633663, 0.72222222],
       [0.28217822, 0.44047619],
       [0.25742574, 0.1468254 ],
       [0.45049505, 0.57142857],
       [0.66831683, 0.80555556],
       [0.43564356, 0.40873016],
       [0.21782178, 0.24603175],
       [0.43564356, 0.54365079],
       [0.28712871, 0.26984127],
       [0.23762376, 0.07936508],
       [0.45049505, 0.19047619],
       [0.50990099, 0.36904762],
       [0.64851485, 0.50793651],
       [0.66831683, 0.80555556],
       [0.27227723, 0.25      ],
       [0.33168317, 0.11904762],
       [0.18811881, 0.23015873],
       [0.25742574, 0.1547619 ],
       [0.28712871, 0.48412698],
       [0.44059406, 0.5515873 ],
       [0.37128713, 0.41269841],
       [0.62376238, 0.50793651],
       [0.45544554, 0.36904762],
       [0.28712871, 0.23015873],
       [0.62871287, 0.50396825],
       [0.45544554, 0.67063492],
       [0.95544554, 0.82539683],
       [0.86633663, 0.6547619 ],
       [0.74752475, 0.40873016],
       [0.69306931, 0.51190476],
       [0.18811881, 0.24603175],
       [0.87128713, 0.32936508],
       [0.83168317, 0.76190476],
       [0.9950495 , 0.67063492],
       [0.97029703, 0.9047619 ],
       [0.61881188, 0.50793651],
       [0.78712871, 0.6031746 ],
       [0.87128713, 0.86507937],
       [0.83168317, 0.51190476],
       [0.23762376, 0.44047619],
       [0.18811881, 0.11507937],
       [0.83663366, 0.84920635],
       [0.76732673, 0.51190476],

       [0.27227723, 0.09126984],
       [0.32673267, 0.44047619],
       [0.51980198, 0.43253968],
       [0.66831683, 0.64285714],
       [0.77722772, 0.79761905],
       [0.66831683, 0.63492063],
       [0.77227723, 0.50396825],
       [0.37128713, 0.42460317],
       [0.32673267, 0.52380952],
       [0.46039604, 0.29761905],
       [0.18811881, 0.        ],
       [0.47524752, 0.43650794],
       [0.55445545, 0.3968254 ],
       [0.25247525, 0.48412698],
       [0.50990099, 0.56746032],
       [0.86633663, 0.80555556],
       [0.23762376, 0.27380952],
       [0.66336634, 0.80952381],
       [0.17326733, 0.29365079],
       [0.25742574, 0.24206349],
       [0.44059406, 0.43650794],
       [0.87128713, 0.80555556],
       [0.81188119, 0.84126984],
       [0.5       , 0.09126984],
       [0.        , 0.19047619]])
```

```
In [227]: sat = rescaleMatrix(sat)
```

```
In [228]: sat
```

```
Out[228]: array([[0.62871287, 0.43253968, 0.83236994],
       [0.20792079, 0.12698413, 0.47976879],
       [0.33168317, 0.09126984, 0.1849711 ],
       [0.83663366, 0.48412698, 0.80346821],
       [0.37623762, 0.23015873, 0.80346821],
       [0.22772277, 0.02380952, 0.16763006],
       [0.28217822, 0.26984127, 0.1849711 ],
       [0.21287129, 0.22222222, 0.09248555],
       [0.17821782, 0.40873016, 0.5433526 ],
       [0.5       , 0.44047619, 0.71676301],
       [0.83168317, 0.67063492, 0.87283237],
       [0.25742574, 0.44444444, 0.26589595],
       [0.43564356, 0.4047619 , 0.6416185 ],
       [0.50990099, 0.57142857, 0.94219653],
       [0.62376238, 0.5515873 , 0.86705202],
       [0.82673267, 0.64285714, 0.76300578],
       [0.92574257, 0.80952381, 0.95375723],
       [0.97029703, 0.68253968, 0.8150289 ],
       [0.23762376, 0.08333333, 0.0982659 ],
       [0.2029703 , 0.27380952, 0.16763006],
       [0.37128713, 0.41269841, 0.69942197],
       [0.41089109, 0.32539683, 0.6416185 ],
       [0.45049505, 0.5515873 , 0.69364162],
       [0.50990099, 0.30952381, 0.74566474],
       [0.91089109, 0.95634921, 0.88439306],
       [1.        , 1.        , 1.        ],
       [0.31683168, 0.23015873, 0.1849711 ],
       [0.22772277, 0.10714286, 0.07514451],
       [0.82673267, 0.66666667, 0.86705202],
       [0.9950495 , 0.96825397, 0.8265896 ],
       [0.91584158, 0.92857143, 0.89017341],
       [0.86633663, 0.80952381, 0.87861272],
       [0.98019802, 0.8968254 , 0.90751445],
       [0.85643564, 0.76587302, 0.97109827],
       [0.18811881, 0.10714286, 0.10982659],
       [0.23762376, 0.25      , 0.15606936],
       [0.75247525, 0.49206349, 0.6300578 ],
       [0.86633663, 0.72222222, 0.80346821],
       [0.28217822, 0.44047619, 0.53179191],
       [illegible],
       [0.25742574, 0.1468254 , 0.38150289],

       [0.45049505, 0.57142857, 0.74566474],
       [0.66831683, 0.80555556, 0.84393064],
       [0.43564356, 0.40873016, 0.69364162],
       [0.21782178, 0.24603175, 0.75722543],
       [0.43564356, 0.54365079, 0.69364162],
       [0.28712871, 0.26984127, 0.6416185 ],
       [0.23762376, 0.07936508, 0.25433526],
       [0.45049505, 0.19047619, 0.57803468],
       [0.50990099, 0.36904762, 0.53757225],
       [0.64851485, 0.50793651, 0.77456647],
       [0.66831683, 0.80555556, 0.87861272],
       [0.27227723, 0.25      , 0.1849711 ],
       [0.33168317, 0.11904762, 0.43352601],
       [0.18811881, 0.23015873, 0.1734104 ],
       [0.25742574, 0.1547619 , 0.65317919],
       [0.28712871, 0.48412698, 0.09248555],
       [0.44059406, 0.5515873 , 0.76300578],
       [0.37128713, 0.41269841, 0.57225434],
       [0.62376238, 0.50793651, 0.83236994],
       [0.45544554, 0.36904762, 0.80346821],
       [0.28712871, 0.23015873, 0.57803468],
       [0.62871287, 0.50396825, 0.75144509],
       [0.45544554, 0.67063492, 0.76878613],
       [0.95544554, 0.82539683, 0.9017341 ],
       [0.86633663, 0.6547619 , 0.94219653],
       [0.74752475, 0.40873016, 0.53757225],
       [0.69306931, 0.51190476, 0.74566474],
       [0.18811881, 0.24603175, 0.15028902],
       [0.87128713, 0.32936508, 0.69942197],
       [0.83168317, 0.76190476, 0.76300578],
       [0.9950495 , 0.67063492, 0.75144509],
       [0.97029703, 0.9047619 , 0.69364162],
       [0.61881188, 0.50793651, 0.71098266],
       [0.78712871, 0.6031746 , 0.77456647],
       [0.87128713, 0.86507937, 0.75722543],
       [0.83168317, 0.51190476, 0.8265896 ],
       [0.23762376, 0.44047619, 0.6300578 ],
       [0.18811881, 0.11507937, 0.64739884],
       [0.83663366, 0.84920635, 0.76878613],
       [0.76732673, 0.51190476, 0.69942197],
       [0.27227723, 0.09126984, 0.6300578 ],
       [0.32673267, 0.44047619, 0.60115607],
       [0.51980198, 0.43253968, 0.94219653],
       [0.66831683, 0.64285714, 0.82080925],
       [0.77722772, 0.79761905, 0.7283237 ],
       [0.66831683, 0.63492063, 0.80924855],
       [0.77227723, 0.50396825, 0.78612717],
       [0.37128713, 0.42460317, 0.87283237],
       [0.32673267, 0.52380952, 0.69364162],
       [0.46039604, 0.29761905, 0.53179191],
       [0.18811881, 0.        , 0.77456647],
       [0.47524752, 0.43650794, 0.76878613],
       [0.55445545, 0.3968254 , 0.8150289 ],
       [0.25247525, 0.48412698, 0.69364162],
       [0.50990099, 0.56746032, 0.6300578 ],
       [0.86633663, 0.80555556, 0.67052023],
       [0.23762376, 0.27380952, 0.15028902],
       [0.66336634, 0.80952381, 0.69364162],
       [0.17326733, 0.29365079, 0.12138728],
       [0.25742574, 0.24206349, 0.        ],
       [0.44059406, 0.43650794, 0.9017341 ],
       [0.87128713, 0.80555556, 0.77456647],
       [0.81188119, 0.84126984, 0.67630058],
       [0.5       , 0.09126984, 0.39306358],
       [0.        , 0.19047619, 0.76878613]])
```

Comparing our results with that of rescaleMatrix(), we see that we get similar results.

## Training Data

In [233]:
```
from sklearn.model_selection import train_test_split
```

In [234]:
```
X_train,X_test,Y_train,Y_test = train_test_split(X,Y,test_size=0.43)
```

Since 60 out of the 105 observations are supposed to be in the training set, the remaining 40 (40/105 = 0.43) will be in the training set.

## Step 2: Calling the Gradient Descent function

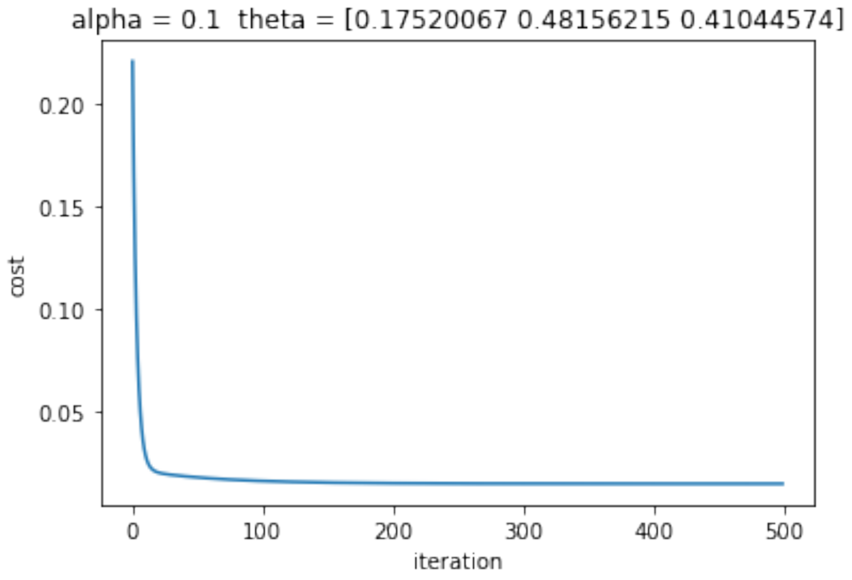
Gradient descent is a powerful optimization algorithm that helps to find the minimum of a cost function by iteratively adjusting the parameters of a model in the direction of steepest decrease in the cost.

In [257]:
```
ALPHA = 0.1
MAX_ITER = 500
theta = np.zeros(3)
```

In [258]:
```
xValues = np.ones((60, 3))
xValues[:, 1:3] = satTrain[:, 0:2]
yValues = satTrain[:, 2]
# call the GD algorithm, placeholders in the function gradientDescent()
[theta, arrCost] = gradientDescent(xValues, yValues, theta, ALPHA, MAX_ITER)
```

In [259]:
```
import numpy as np

def gradientDescent(X, y, theta, alpha, numIterations):
    '''
    # This function returns a tuple (theta, Cost array)
    '''
    m = len(y)
    arrCost =[]
    transposedX = np.transpose(X) # transpose X into a vector  -> XColCount X m matrix
    for iteration in range(0, numIterations):
        # calculate the hypothesis
        hypothesis = X.dot(theta)

        # calculate the error
        error = hypothesis - y

        # calculate the gradient
        gradient = (1/m) * transposedX.dot(error)

        # update theta
        change = alpha * gradient
        theta = theta - change

        # calculate the cost
        cost = (1/(2*m)) * np.sum(np.square(error))
        arrCost.append(cost)

    return theta, arrCost
```

In [260]:
```
plt.plot(range(0,len(arrCost)),arrCost);
plt.xlabel('iteration')
plt.ylabel('cost')
plt.title('alpha = {}  theta = {}'.format(ALPHA, theta))
plt.show()
```



Theta refers to the model parameters or coefficients of a linear regression model. These parameters represent the weights of the features in the model and determine the slope and intercept of the line that fits the data. Iterations refer to the number of times the gradient descent algorithm updates the values of theta.

As for Alpha, it refers to the learning rate, also known as the step size, in the gradient descent algorithm. It determines the size of the update step that is taken towards the minimum value of the cost function in each iteration. The learning rate alpha controls the pace at which the algorithm converges to the minimum value of the cost function.

Here, we see that alpha is 0.1, i.e. it takes a small towards converging its minimum value. However, since the number of iterations i.e. the number of times the gradient descent algorithm updates the value of theta is set high at about 500.

The purpose of the plot is to show the convergence of the gradient descent algorithm, where the cost should decrease as the number of iterations increases until it reaches a minimum value. From the above graph, we see that the cost decreases as the number of iterations decrease.

In [261]:
```
testXValues = np.ones((len(satTest), 3))
testXValues[:, 1:3] = satTest[:, 0:2]
tVal =  testXValues.dot(theta)
```

In [262]:
```
tError = np.sqrt([x**2 for x in np.subtract(tVal, satTest[:, 2])])
print('results: {} ({})'.format(np.mean(tError), np.std(tError)))
```

```
results: 0.1727749787556687 (0.12688442297897462)
```

This shows us the mean and standard deviation of the RMSE, which are summary statistics of the model's performance on the test data. A low mean and low standard deviation of the RMSE indicate that the model has a good fit to the data and makes accurate predictions.

## Changing the Alpha and Max Itereations

**a)**

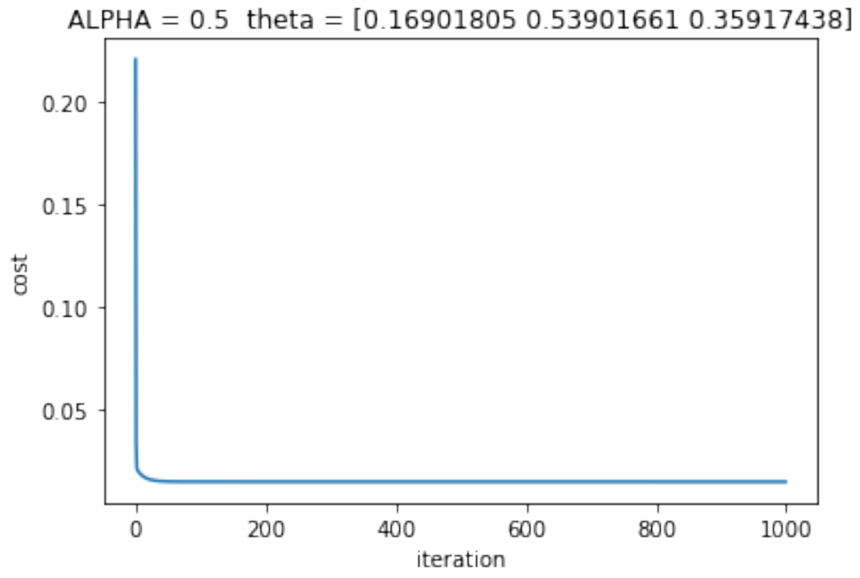
In [273]:
```
ALPHA = 0.5
MAX_ITER = 1000
theta = np.zeros(3)
```

We now change the alpha to 0.5, this means that the learning rate has increased and hence the gradient descent now converges to its minimum at a faster rate as compared to 0.1. Furthermore, now that the number of iterations is also higher, the theta is updated more times.

In [274]:
```
xValues = np.ones((60, 3))
xValues[:, 1:3] = satTrain[:, 0:2]
yValues = satTrain[:, 2]
# call the GD algorithm, placeholders in the function gradientDescent()
[theta, arrCost] = gradientDescent(xValues, yValues, theta, ALPHA, MAX_ITER)
```

In [275]:
```
import numpy as np

def gradientDescent(X, y, theta, alpha, numIterations):
    '''
    # This function returns a tuple (theta, Cost array)
    '''
    m = len(y)
    arrCost =[]
    transposedX = np.transpose(X) # transpose X into a vector  -> XColCount X m matrix
    for iteration in range(0, numIterations):
        # calculate the hypothesis
        hypothesis = X.dot(theta)

        # calculate the error
        error = hypothesis - y

        # calculate the gradient
        gradient = (1/m) * transposedX.dot(error)

        # update theta
        change = alpha * gradient
        theta = theta - change

        # calculate the cost
        cost = (1/(2*m)) * np.sum(np.square(error))
        arrCost.append(cost)

    return theta, arrCost
```

In [276]:
```
plt.plot(range(0,len(arrCost)),arrCost);
plt.xlabel('iteration')
plt.ylabel('cost')
plt.title('ALPHA = {}  theta = {}'.format(ALPHA, theta))
plt.show()
```



In [277]:
```
testXValues = np.ones((len(satTest), 3))
testXValues[:, 1:3] = satTest[:, 0:2]
tVal =  testXValues.dot(theta)
```

In [278]:
```
tError = np.sqrt([x**2 for x in np.subtract(tVal, satTest[:, 2])])
print('results: {} ({})'.format(np.mean(tError), np.std(tError)))
```

```
results: 0.17475247640012 (0.12723648781309851)
```

**b)**

In [279]:
```
ALPHA = 0.01
MAX_ITER = 100
theta = np.zeros(3)
```

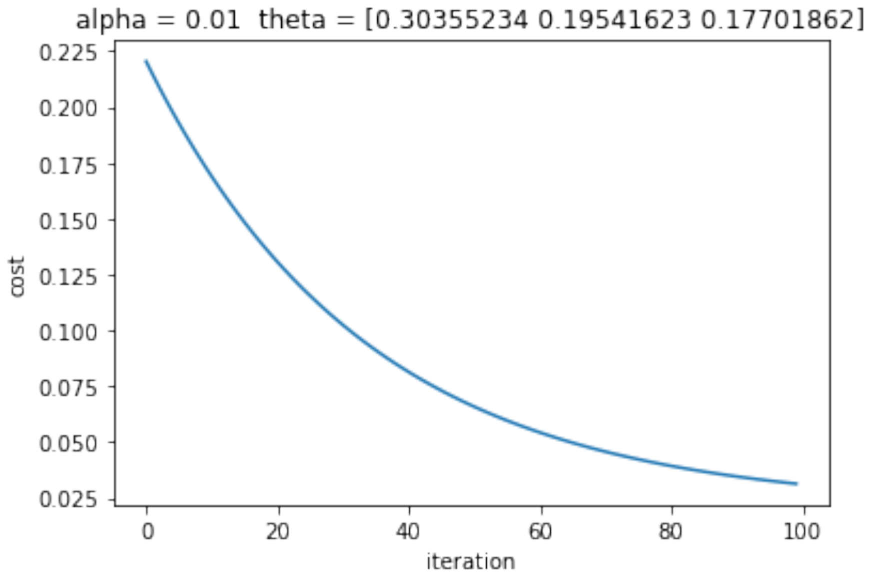
The learning rate is 0.01 and the theta is updated 100 times

In [280]:
```
xValues = np.ones((60, 3))
xValues[:, 1:3] = satTrain[:, 0:2]
yValues = satTrain[:, 2]
# call the GD algorithm, placeholders in the function gradientDescent()
[theta, arrCost] = gradientDescent(xValues, yValues, theta, ALPHA, MAX_ITER)
```

In [281]:

```python
import numpy as np

def gradientDescent(X, y, theta, alpha, numIterations):
    '''
    # This function returns a tuple (theta, Cost array)
    '''
    m = len(y)
    arrCost =[]
    transposedX = np.transpose(X) # transpose X into a vector  -> XColCount X m matrix
    for iteration in range(0, numIterations):
        # calculate the hypothesis
        hypothesis = X.dot(theta)

        # calculate the error
        error = hypothesis - y

        # calculate the gradient
        gradient = (1/m) * transposedX.dot(error)

        # update theta
        change = alpha * gradient
        theta = theta - change

        # calculate the cost
        cost = (1/(2*m)) * np.sum(np.square(error))
        arrCost.append(cost)

    return theta, arrCost
```

In [282]:
```
plt.plot(range(0,len(arrCost)),arrCost);
plt.xlabel('iteration')
plt.ylabel('cost')
plt.title('alpha = {}  theta = {}'.format(ALPHA, theta))
plt.show()
```



Here, we see that the alpha is pretty low at 0.01. That means that the learning late is pretty low, hence influencing the shape of the curve. Unlike the curves above, it is flatter. It is because of the low alpha that the gradient descent takes a longer time to converege to its lowest value.

In [283]:
```
testXValues = np.ones((len(satTest), 3))
testXValues[:, 1:3] = satTest[:, 0:2]
tVal =  testXValues.dot(theta)
```

In [284]:
```
tError = np.sqrt([x**2 for x in np.subtract(tVal, satTest[:, 2])])
print('results: {} ({})'.format(np.mean(tError), np.std(tError)))
```

```
results: 0.22286048011536305 (0.11293418605757809)
```

**c)**

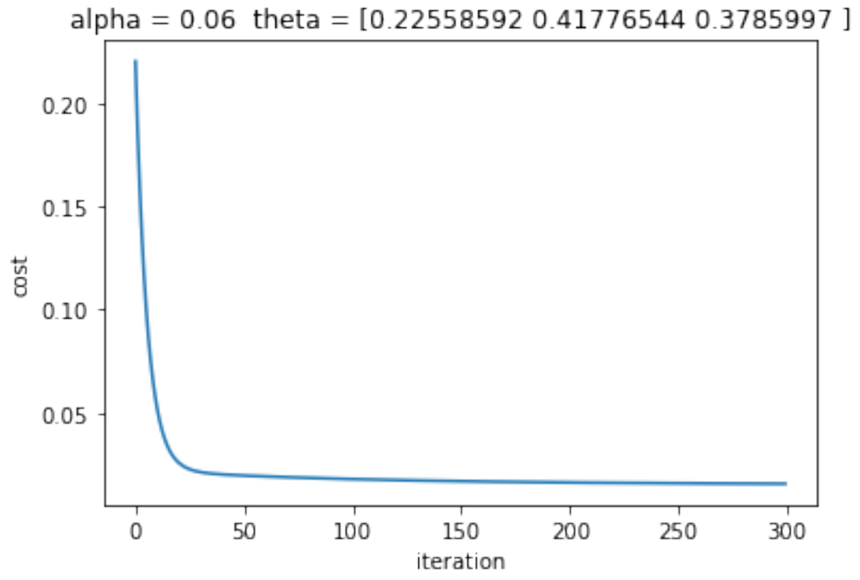
In [296]:
```
ALPHA = 0.06
MAX_ITER = 300
theta = np.zeros(3)
```

The learning rate is 0.06 and the theta is updated 300 times

In [297]:
```
xValues = np.ones((60, 3))
xValues[:, 1:3] = satTrain[:, 0:2]
yValues = satTrain[:, 2]
# call the GD algorithm, placeholders in the function gradientDescent()
[theta, arrCost] = gradientDescent(xValues, yValues, theta, ALPHA, MAX_ITER)
```

In [298]:
```python
import numpy as np

def gradientDescent(X, y, theta, alpha, numIterations):
    '''
    # This function returns a tuple (theta, Cost array)
    '''
    m = len(y)
    arrCost =[]
    transposedX = np.transpose(X) # transpose X into a vector  -> XColCount X m matrix
    for iteration in range(0, numIterations):
        # calculate the hypothesis
        hypothesis = X.dot(theta)

        # calculate the error
        error = hypothesis - y

        # calculate the gradient
        gradient = (1/m) * transposedX.dot(error)

        # update theta
        change = alpha * gradient
        theta = theta - change

        # calculate the cost
        cost = (1/(2*m)) * np.sum(np.square(error))
        arrCost.append(cost)

    return theta, arrCost
```

In [299]:
```
plt.plot(range(0,len(arrCost)),arrCost);
plt.xlabel('iteration')
plt.ylabel('cost')
plt.title('alpha = {}  theta = {}'.format(ALPHA, theta))
plt.show()
```



We again notice here that as we change the alpha from 0.01 to 0.06, the curve becomes more steeper. This signifies that the gradient descent converges to its minimum more quickly.

Moreover, the theta in updated more number of times (up from 100 to 300)

In [300]:
```
testXValues = np.ones((len(satTest), 3))
testXValues[:, 1:3] = satTest[:, 0:2]
tVal =  testXValues.dot(theta)
```

```
In [301]: tError = np.sqrt([x**2 for x in np.subtract(tVal, satTest[:, 2])])
          print('results: {} ({})'.format(np.mean(tError), np.std(tError)))
```

```
results: 0.1629023591919987 (0.12198645621211272)
```

From the above changes in Alpha and the number of iterations, we see that the mean of RMSE for alpha = 0.06 and number of iterations 500 is the lowest while the standard deviation of RMSE for alpha is 0.01 and number of iterations = 100 is the lowest.

Lower mean and standard deviation of the RMSE indicate that it is a good fit to the model.

We also see that both alpha and the number of iterations affect the ultimate outcome of the graph as well as the mean and standard deviation of the RMSE.

```
In [ ]:
```