Exercise 1: Linear Regression

This notebook is executed automatically. Failing to meet any of the submission requirements will results in a 25 point fine or your submission not being graded at all. Kindly reminder: the homework assignments grade is 50% of the final grade.

Do not start the exercise until you fully understand the submission guidelines.

Read the following instructions carefully:

1. This jupyter notebook contains all the step by step instructions needed for this exercise.
2. Submission includes this notebook only with the exercise number and your ID as the filename. For example: `hw1_123456789_987654321.ipynb` if you submitted in pairs and `hw1_123456789.ipynb` if you submitted the exercise alone.
3. Write **efficient vectorized** code whenever possible. Some calculations in this exercise take several minutes when implemented efficiently, and might take much longer otherwise. Unnecessary loops will result in point deduction.
4. You are responsible for the correctness of your code and should add as many tests as you see fit. Tests will not be graded nor checked.
5. Write your functions in this notebook only. **Do not create Python modules and import them**.
6. You are allowed to use functions and methods from the Python Standard Library (https://docs.python.org/3/library/) and numpy (https://www.numpy.org/devdocs/reference/) only. **Do not import anything else.**
7. Your code must run without errors. Make sure your `numpy` version is at least 1.15.4 and that you are using at least python 3.6. Changes of the configuration we provided are at your own risk. Any code that cannot run will not be graded.
8. Write your own code. Cheating will not be tolerated.

9. Answers to qualitative questions should be written in **markdown** cells (with $\LaTeX$ support). Answers that will be written in commented code blocks will not be checked.

In this exercise you will perform the following:

1. Load a dataset and perform basic data exploration using a powerful data science library called pandas (https://pandas.pydata.org/pandas-docs/stable/).
2. Preprocess the data for linear regression.
3. Compute the cost and perform gradient descent in pure numpy in vectorized form.
4. Fit a linear regression model using a single feature.
5. Visualize your results using matplotlib.
6. Perform multivariate linear regression.
7. Pick the best features in the dataset.
8. Experiment with adaptive learning rates.

I have read and understood the instructions: 206035313

In [2]:

```
import numpy as np # used for scientific computing
import pandas as pd # used for data analysis and manipulation
import matplotlib.pyplot as plt # used for visualization and plotting

np.random.seed(42)

# make matplotlib figures appear inline in the notebook
%matplotlib inline
plt.rcParams['figure.figsize'] = (14.0, 8.0) # set default size of plot
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'
```

Part 1: Data Preprocessing (10 Points)

For the following exercise, we will use a dataset containing housing prices in King County, USA. The dataset contains 5,000 observations with 18 features and a single target value - the house price.

First, we will read and explore the data using pandas and the `.read_csv` method. Pandas is an open source library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

In [8]:

```
# Read comma separated data
df = pd.read_csv('data.csv') # Make sure this cell runs regardless of y
# df stands for dataframe, which is the default format for datasets in
```

Data Exploration

A good practice in any data-oriented project is to first try and understand the data. Fortunately, pandas is built for that purpose. Start by looking at the top of the dataset using the `df.head()` command. This will be the first indication that you read your data properly, and that the headers are correct. Next, you can use `df.describe()` to show statistics on the data and check for trends and irregularities.

In [54]:

```
df.head(5)
```

Out[54]:

ıthrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grade
1.00	1180	5650	1.0	0	0	3	7
2.25	2570	7242	2.0	0	0	3	7
1.00	770	10000	1.0	0	0	3	6
3.00	1960	5000	1.0	0	0	5	7
2.00	1680	8080	1.0	0	0	3	8

In [4]:

```
df.describe()
```

Out[4]:

	id	price	bedrooms	bathrooms	sqft_liv
count	5.000000e+03	5.000000e+03	5000.0000	5000.000000	5000.000
mean	4.630823e+09	5.394699e+05	3.3714	2.062150	2061.036
std	2.870890e+09	3.873115e+05	0.9104	0.773592	923.727
min	1.000102e+06	7.500000e+04	0.0000	0.000000	380.000
25%	2.154075e+09	3.179062e+05	3.0000	1.500000	1410.000
50%	4.022900e+09	4.490000e+05	3.0000	2.000000	1890.000
75%	7.345078e+09	6.500000e+05	4.0000	2.500000	2500.000
max	9.842300e+09	7.060000e+06	9.0000	6.750000	10040.000

We will start with one variable linear regression by extracting the target column and

the `sqft_living` variable from the dataset. We use pandas and select both columns as separate variables and transform them into a numpy array.

In [56]:

```
# my test
print(type(df.columns))
```

```
<class 'pandas.core.indexes.base.Index'>
```

In [9]:

```
X = df['sqft_living'].values
y = df['price'].values
```

Preprocessing

As the number of features grows, calculating gradients gets computationally expensive. We can speed this up by normalizing the input data to ensure all values are within the same range. This is especially important for datasets with high standard deviations or differences in the ranges of the attributes. Use mean normalization (https://en.wikipedia.org/wiki/Feature_scaling) for the fearures (`X`) and the true labels (`y`).

Implement the cost function `preprocess` .

In [10]:

```
def preprocess(X, y):
    """
    Perform mean normalization on the features and true labels.

    Input:
    - X: Inputs (n features over m instances).
    - y: True labels.

    Returns a two vales:
    - X: The mean normalized inputs.
    - y: The mean normalized labels.
    """
    ###########################################################################
    # TODO: Implement the normalization function.
    ###########################################################################
    X = (X - X.mean(axis=0)) / X.std(axis=0)
    y = (y - y.mean(axis=0)) / y.std(axis=0)
    ###########################################################################
    #                                END OF YOUR CODE
    ###########################################################################
    return X, y
```

In [11]:

```
X, y = preprocess(X, y)
```

In [12]:

```
# my test
print(X.mean())
print(X.std())
```

```
1.5489831639570184e-16
1.0
```

We will split the data into two datasets:

1. The training dataset will contain 80% of the data and will always be used for model training.
2. The validation dataset will contain the remaining 20% of the data and will be used for model evaluation. For example, we will pick the best alpha and the best features using the validation dataset, while still training the model using the training dataset.

In [15]:

```
# training and validation split
np.random.seed(42)
indices = np.random.permutation(X.shape[0])
idx_train, idx_val = indices[:int(0.8*X.shape[0])], indices[int(0.8*X.s
X_train, X_val = X[idx_train], X[idx_val]
y_train, y_val = y[idx_train], y[idx_val]
```

In [16]:

```
# my test
print(len(y_train))
```

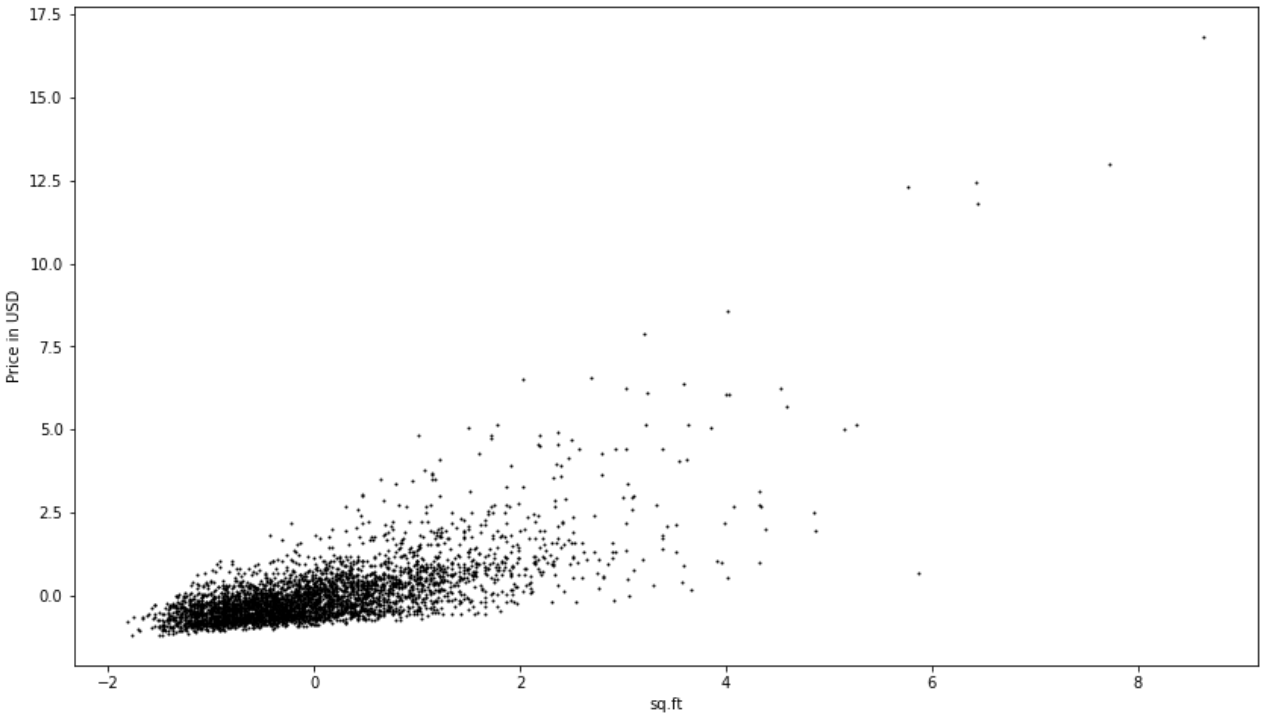
```
4000
```

Data Visualization

Another useful tool is data visualization. Since this problem has only two parameters, it is possible to create a two-dimensional scatter plot to visualize the data. Note that many real-world datasets are highly dimensional and cannot be visualized naively. We will be using `matplotlib` for all data visualization purposes since it offers a wide range of visualization tools and is easy to use.

In [17]:

```
plt.plot(X_train, y_train, 'ro', ms=1, mec='k') # the parameters contro
plt.ylabel('Price in USD')
plt.xlabel('sq.ft')
plt.show()
```



Bias Trick

Make sure that `X` takes into consideration the bias θ_0 in the linear model. Hint, recall that the predications of our linear model are of the form:

$$\hat{y} = h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1$$

Add columns of ones as the zeroth column of the features (do this for both the training and validation sets).

In [18]:

```
################################################################################
#                                START OF YOUR CODE
################################################################################
X_train = np.stack((np.ones_like(X_train), X_train), axis = 1)
X_val = np.stack((np.ones_like(X_val), X_val), axis = 1)
################################################################################
#                                END OF YOUR CODE
################################################################################
```

In [19]:

```
# my test
print(X_train.shape[1])
```

```
2
```

Part 2: Single Variable Linear Regression (40 Points)

Simple linear regression is a linear regression model with a single explanatory varaible and a single target value.

$$\hat{y} = h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1$$

Gradient Descent

Our task is to find the best possible linear line that explains all the points in our dataset. We start by guessing initial values for the linear regression parameters θ and updating the values using gradient descent.

The objective of linear regression is to minimize the cost function J:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{n} (h_\theta(x^{(i)}) - y^{(i)})^2$$

where the hypothesis (model) $h_\theta(x)$ is given by a **linear** model:

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1$$

θ_j are parameters of your model. and by changing those values accordingly you will be able to lower the cost function $J(\theta)$. One way to accopmlish this is to use gradient descent:

$$\theta_j = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

In linear regresion, we know that with each step of gradient descent, the parameters θ_j get closer to the optimal values that will achieve the lowest cost $J(\theta)$.

Implement the cost function `compute_cost` . (10 points)

In [20]:

```
def compute_cost(X, y, theta):
    """
    Computes the average squared difference between an obserbation's ac
    predicted values for linear regression.

    Input:
    - X: inputs  (n features over m instances).
    - y: true labels (1 value over m instances).
    - theta: the parameters (weights) of the model being learned.

    Returns a single value:
    - J: the cost associated with the current set of parameters (single
    """

    J = 0  # Use J for the cost.
    ###########################################################################
    # TODO: Implement the MSE cost function.
    ###########################################################################
    h_theta_sum = 0
    h_theta = 0
    h_theta = X.dot(theta)
    h_theta -= y
    h_theta *= h_theta
    h_theta_sum = np.sum(h_theta)
    J = h_theta_sum / (2.0 * len(X))
    ###########################################################################
    #                             END OF YOUR CODE
    ###########################################################################
    return J
```

In [21]:

```
theta = np.array([-1, 2])
J = compute_cost(X_train, y_train, theta)
print(J)
```

```
1.6003214548835494
```

Implement the gradient descent function `gradient_descent` . (10 points)

In [22]:

```
def gradient_descent(X, y, theta, alpha, num_iters):
    """
    Learn the parameters of the model using gradient descent using
    the *training set*. Gradient descent is an optimization algorithm
    used to minimize some (loss) function by iteratively moving in
    the direction of steepest descent as defined by the negative of
    the gradient. We use gradient descent to update the parameters
    (weights) of our model.

    Input:
    - X: Inputs  (n features over m instances).
    - y: True labels (1 value over m instances).
    - theta: The parameters (weights) of the model being learned.
    - alpha: The learning rate of your model.
    - num_iters: The number of updates performed.

    Returns two values:
    - theta: The learned parameters of your model.
    - J_history: the loss value for every iteration.
    """

    J_history = [] # Use a python list to save cost in every iteration
    theta = theta.copy() # avoid changing the original thetas
    ##########################################################
    # TODO: Implement the gradient descent optimization algorithm.
    ##########################################################
    for k in range(num_iters):
        J_history.append(compute_cost(X, y, theta))
        vec = X.dot(theta)
        vec -= y
        Xt_mult_vec = (np.transpose(X)).dot(vec)
        Xt_mult_vec *= (alpha / len(X))
        theta -= Xt_mult_vec
    ##########################################################
    #                             END OF YOUR CODE
    ##########################################################
    return theta, J_history
```

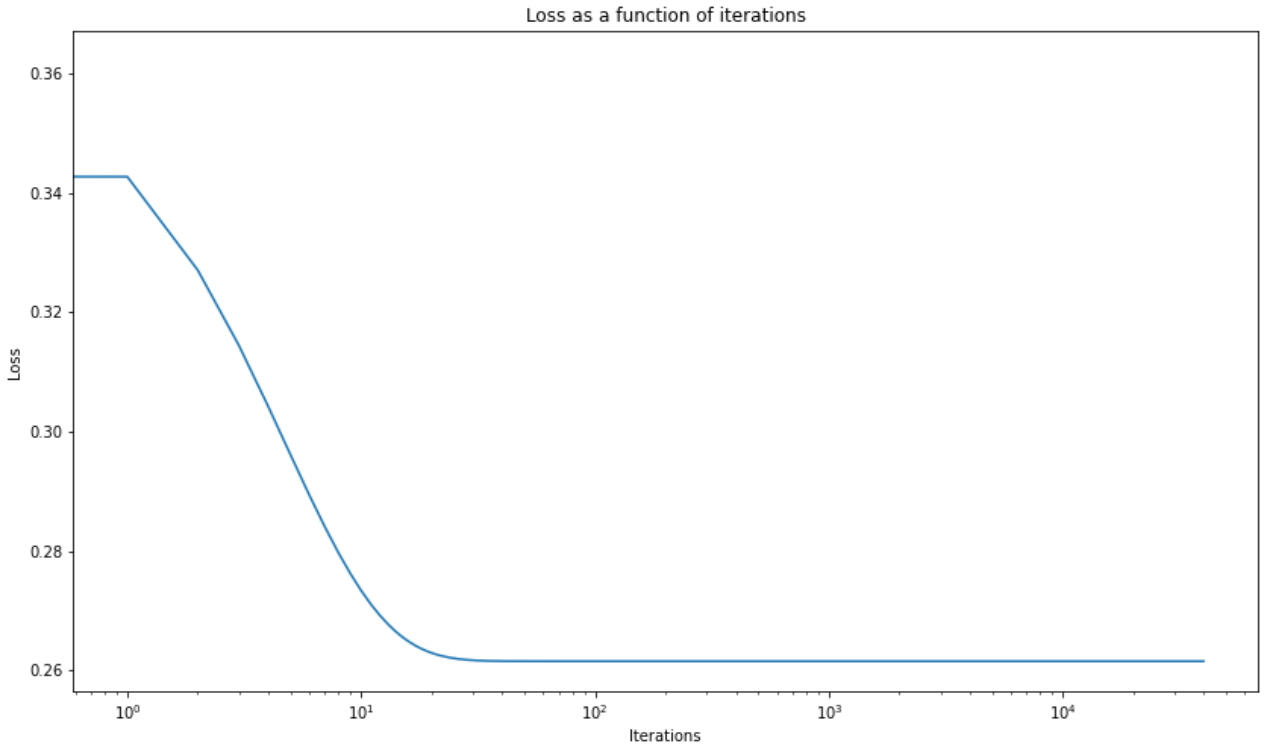
In [23]:

```
np.random.seed(42)
theta = np.random.random(size=2)
iterations = 40000
alpha = 0.1
theta, J_history = gradient_descent(X_train ,y_train, theta, alpha, ite
```

You can evaluate the learning process by monitoring the loss as training progress. In the following graph, we visualize the loss as a function of the iterations. This is possible since we are saving the loss value at every iteration in the `J_history` array. This visualization might help you find problems with your code. Notice that since the network converges quickly, we are using logarithmic scale for the number of iterations.

In [24]:

```
plt.plot(np.arange(iterations), J_history)
plt.xscale('log')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss as a function of iterations')
plt.show()
```



Implement the pseudo-inverse function `pinv`. **Do not use `np.linalg.pinv`**, instead use only direct matrix multiplication as you saw in class (you can calculate the inverse of a matrix using `np.linalg.inv`). (10 points)

In [25]:

```
def pinv(X, y):
    """
    Calculate the optimal values of the parameters using the pseudoinve
    approach as you saw in class using the *training set*.

    Input:
    - X: Inputs  (n features over m instances).
    - y: True labels (1 value over m instances).

    Returns two values:
    - theta: The optimal parameters of your model.

    ########## DO NOT USE np.linalg.pinv ##############
    """

    pinv_theta = []
    ###########################################################################
    # TODO: Implement the pseudoinverse algorithm.
    ###########################################################################
    X_T = np.transpose(X)
    invOf_X_T_mult_X = np.linalg.inv(X_T.dot(X))
    X_T_mult_y = X_T.dot(y)
    pinv_theta = invOf_X_T_mult_X.dot(X_T_mult_y)
    ###########################################################################
    #                                END OF YOUR CODE
    ###########################################################################
    return pinv_theta
```

In [26]:

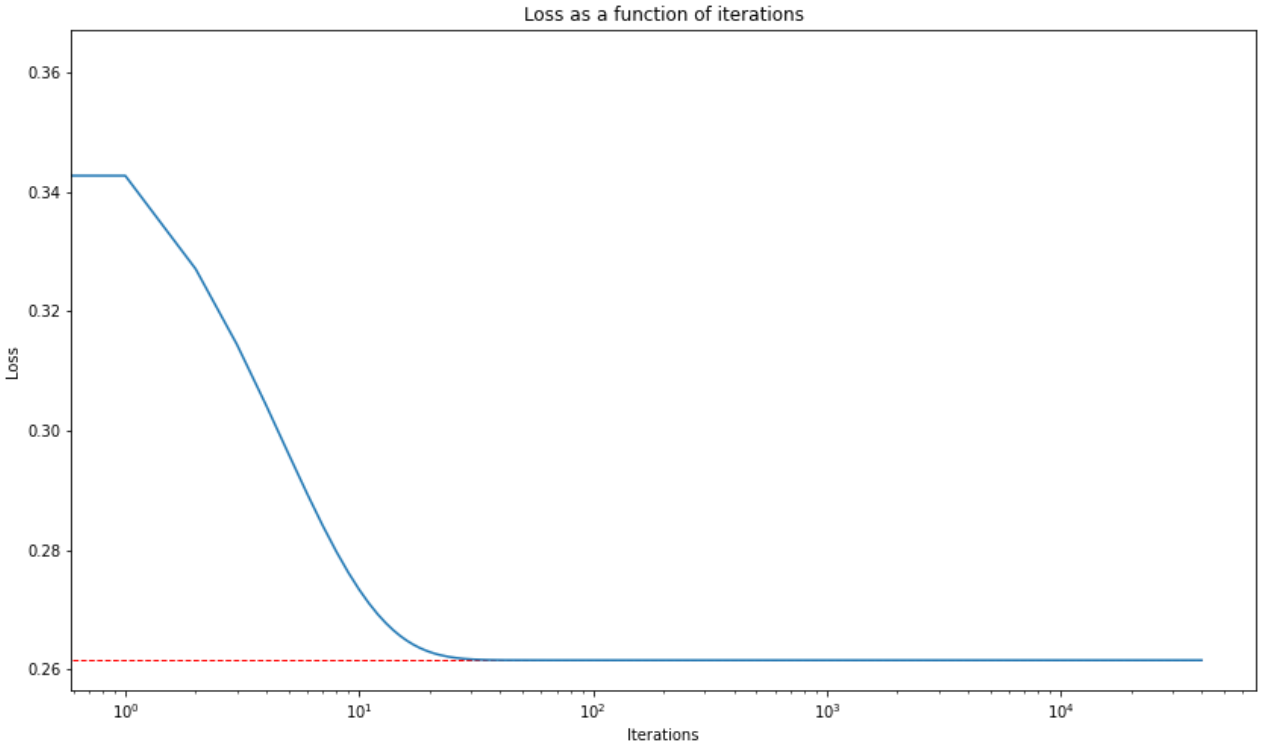
```
theta_pinv = pinv(X_train ,y_train)
J_pinv = compute_cost(X_train, y_train, theta_pinv)
```

We can add the loss value for the theta calculated using the psuedo-inverse to our

graph. This is another sanity check as the loss of our model should converge to the psuedo-inverse loss.

In [27]:

```
plt.plot(np.arange(iterations), J_history)
plt.xscale('log')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss as a function of iterations')
plt.hlines(y = J_pinv, xmin = 0, xmax = len(J_history), color='r',
           linewidth = 1, linestyle = 'dashed')
plt.show()
```



We can use a better approach for the implementation of `gradient_descent`. Instead of performing 40,000 iterations, we wish to stop when the improvement of the loss value is smaller than `1e-8` from one iteration to the next. Implement the function `efficient_gradient_descent`. (5 points)

In [28]:

```
def efficient_gradient_descent(X, y, theta, alpha, num_iters):
    """
    Learn the parameters of your model using the *training set*, but st
    the learning process once the improvement of the loss value is smal
    than 1e-8. This function is very similar to the gradient descent
    function you already implemented.

    Input:
    - X: Inputs  (n features over m instances).
    - y: True labels (1 value over m instances).
    - theta: The parameters (weights) of the model being learned.
    - alpha: The learning rate of your model.
    - num_iters: The number of updates performed.

    Returns two values:
    - theta: The learned parameters of your model.
    - J_history: the loss value for every iteration.
    """

    J_history = [] # Use a python list to save cost in every iteration
    theta = theta.copy() # avoid changing the original thetas
    ###########################################################
    # TODO: Implement the gradient descent optimization algorithm.
    ###########################################################
    k = 0
    curr_cost = compute_cost(X, y, theta)
    J_history.append(0)
    while (abs(curr_cost - J_history[len(J_history) - 1]) > 1E-8) and (
        if k == 0:
            J_history.pop()
        k += 1
        J_history.append(curr_cost)
        vec = X.dot(theta)
        vec -= y
        Xt_mult_vec = (np.transpose(X)).dot(vec)
        Xt_mult_vec *= (alpha / len(X))
        theta -= Xt_mult_vec
        curr_cost = compute_cost(X, y, theta)
    ###########################################################
    #                                 END OF YOUR CODE
```

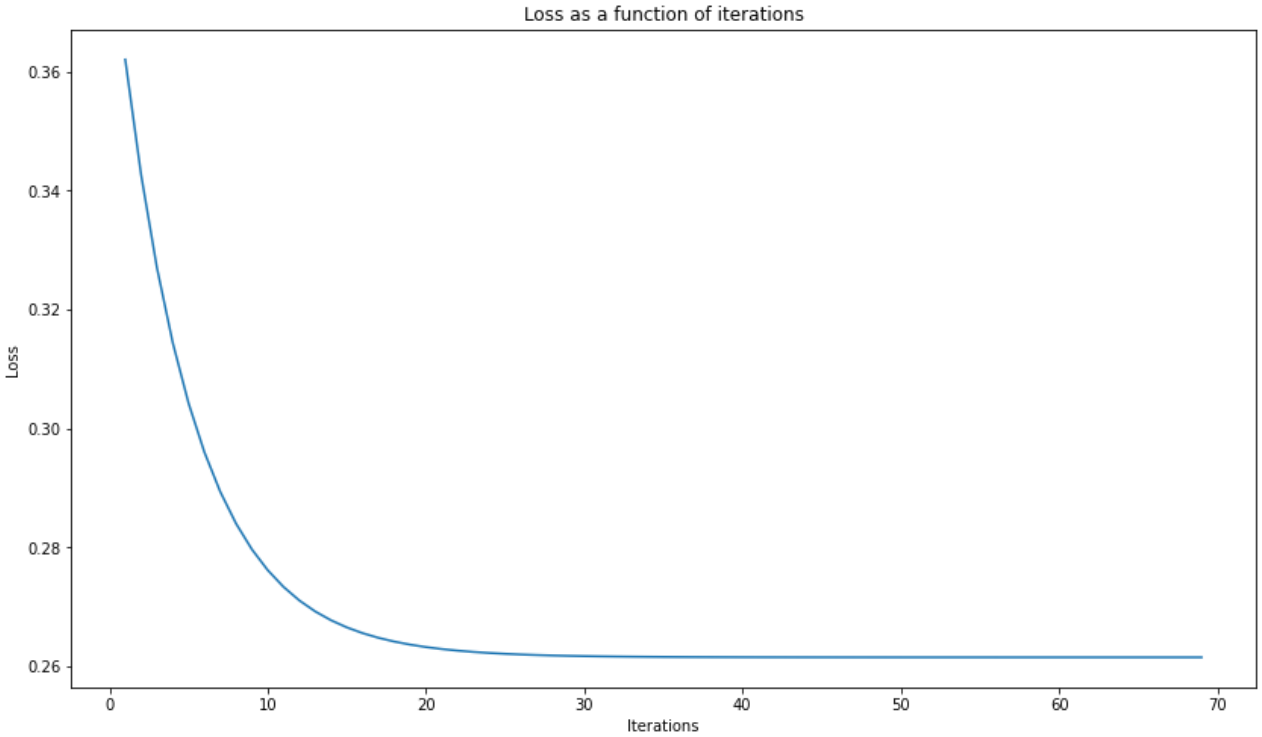
```
    ############################################################################
    return theta, J_history
```

In [29]:

```
# my test
np.random.seed(42)
theta = np.random.random(size=2)
iterations = 40000
alpha = 0.1
theta, J_history = efficient_gradient_descent(X_train ,y_train, theta,
```

In [30]:

```
# my test
plt.plot([x for x in range(1,70)], J_history)
#plt.xscale('log')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss as a function of iterations')
plt.show()
```



The learning rate is another factor that determines the performance of our model in terms of speed and accuracy. Complete the function `find_best_alpha` . Make sure you use the training dataset to learn the parameters (thetas) and use those

parameters with the validation dataset to compute the cost.

In [31]:

```
def find_best_alpha(X_train, y_train, X_val, y_val, iterations):
    """
    Iterate over provided values of alpha and train a model using the
    *training* dataset. maintain a python dictionary with alpha as the
    key and the loss on the *validation* set as the value.

    Input:
    - X_train, y_train, X_val, y_val: the training and validation data
    - iterations: maximum number of iterations

    Returns:
    - alpha_dict: A python dictionary - {key (alpha) : value (validatio
    """

    alphas = [0.00001, 0.00003, 0.0001, 0.0003, 0.001, 0.003, 0.01, 0.0
    alpha_dict = {}
    ###########################################################################
    # TODO: Implement the function.
    ###########################################################################
    for alpha in alphas:
        np.random.seed(42)
        theta = np.random.random(size=X_train.shape[1])
        theta, _ = efficient_gradient_descent(X_train, y_train, theta,
        curr_cost = compute_cost(X_val, y_val, theta)
        alpha_dict[alpha] = curr_cost
    ###########################################################################
    #                                 END OF YOUR CODE
    ###########################################################################
    return alpha_dict
```

In [32]:

```
alpha_dict = find_best_alpha(X_train, y_train, X_val, y_val, 40000)
```

```
c:\users\feith\pycharmprojects\pythonproject\venv\lib\si
te-packages\numpy\core\fromnumeric.py:87: RuntimeWarnin
g: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwarg
s)
<ipython-input-28-6b7623ceb264>:28: RuntimeWarning: inva
lid value encountered in double_scalars
  while (abs(curr_cost - J_history[len(J_history) - 1])
> 1E-8) and (k < num_iters):
```

In [33]:

```
# my test
print(alpha_dict)
```

```
{1e-05: 0.280695641290968, 3e-05: 0.2425213024747859 3,
0.0001: 0.230790961796441 87, 0.0003: 0.230639151867692 2
4, 0.001: 0.230553786275113 22, 0.003: 0.230512613161441 6
2, 0.01: 0.23048792248885927, 0.03: 0.230475327160238 8,
0.1: 0.2304675485623277, 0.3: 0.2304625908996922, 1: 0.2
3045960301228233, 2: 2.3545693014908423e+304, 3: inf}
```

Obtain the best learning rate from the dictionary `alpha_dict` . This can be done in a single line using built-in functions.

In [34]:

```
best_alpha = None
################################################################
#                              START OF YOUR CODE
################################################################
best_alpha = list(alpha_dict.keys())[list(alpha_dict.values()).index(mi
################################################################
#                               END OF YOUR CODE
################################################################
print(best_alpha)
```

```
1
```

Pick the best three alpha values you just calculated and provide **one** graph with three lines indicating the training loss as a function of iterations (Use 10,000 iterations). Note you are required to provide general code for this purpose (no hard-coding). Make sure the visualization is clear and informative. (5 points)

In [78]:

```
# my test
sorted_alpha_dict_values = list(sorted(alpha_dict.values()))
print(list(alpha_dict.keys())[list(alpha_dict.values()).index(sorted_al
```

```
0.1
```

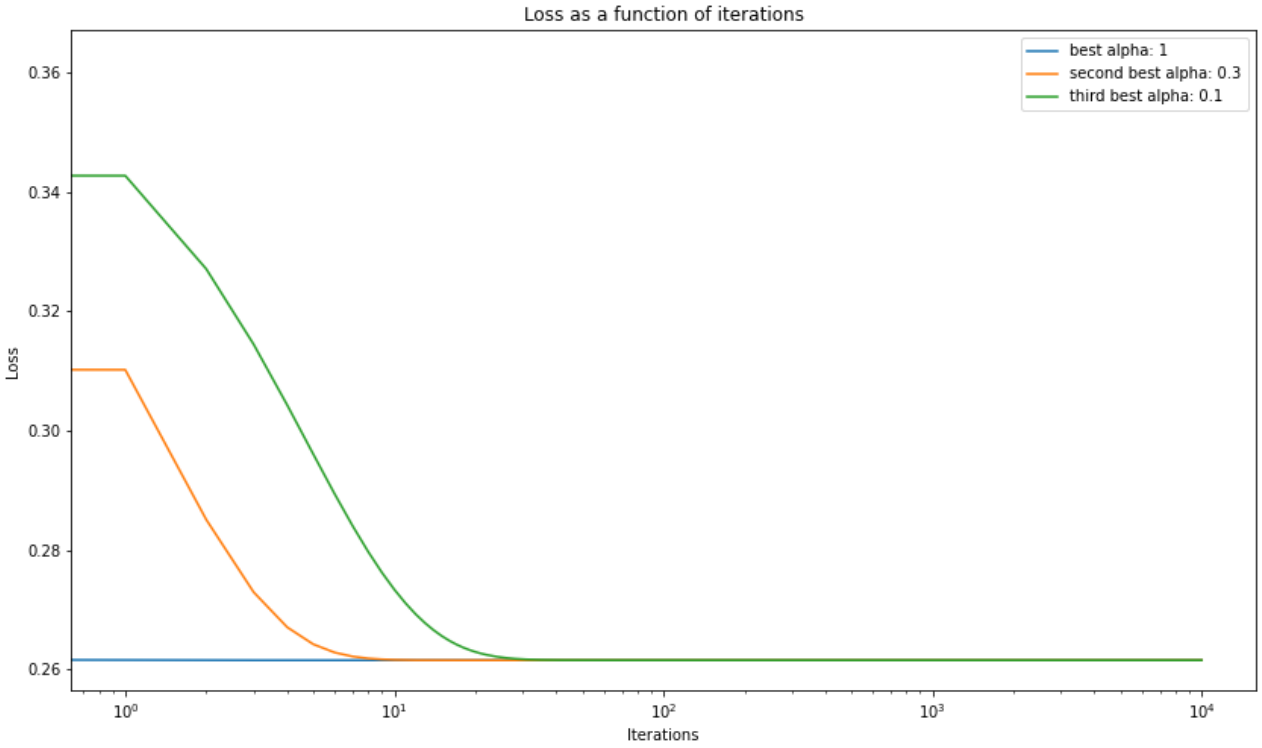
In [35]:

```
################################################################################
#                                START OF YOUR CODE
################################################################################
np.random.seed(42)
temp_theta = np.random.random(size=X_train.shape[1])

sorted_alpha_dict_values = list(sorted(alpha_dict.values()))
first_best_alpha = list(alpha_dict.keys())[list(alpha_dict.values()).in
second_best_alpha = list(alpha_dict.keys())[list(alpha_dict.values()).i
third_best_alpha = list(alpha_dict.keys())[list(alpha_dict.values()).in

_, second_J_history = gradient_descent(X_train ,y_train, temp_theta, se
_, third_J_history = gradient_descent(X_train ,y_train, temp_theta, thi
_, first_J_history = gradient_descent(X_train ,y_train, temp_theta, fir

plt.plot(np.arange(10000), first_J_history)
plt.plot(np.arange(10000), second_J_history)
plt.plot(np.arange(10000), third_J_history)
plt.xscale('log')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss as a function of iterations')
plt.legend(["best alpha: {}".format(first_best_alpha), "second best alp
plt.show()
################################################################################
#                                END OF YOUR CODE
################################################################################
```



This is yet another sanity check. This function plots the regression lines of your model and the model based on the pseudoinverse calculation. Both models should exhibit the same trend through the data.

In [119]:

```
# my test
theta
```

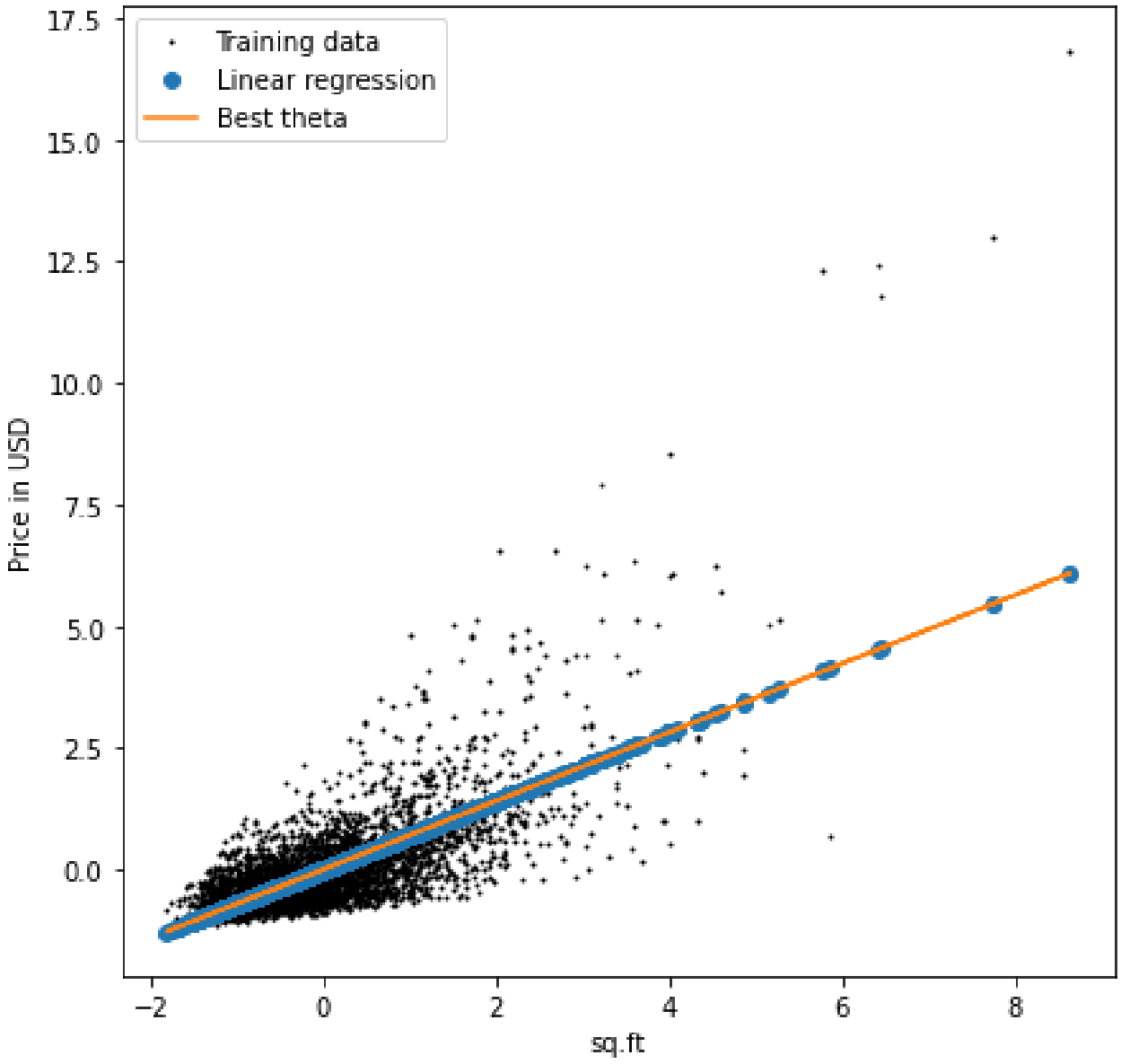
Out[119]:

```
array([0.0035045 , 0.70490832])
```

In [36]:

```
plt.figure(figsize=(7, 7))
plt.plot(X_train[:,1], y_train, 'ro', ms=1, mec='k')
plt.ylabel('Price in USD')
plt.xlabel('sq.ft')
plt.plot(X_train[:, 1], np.dot(X_train, theta), 'o')
plt.plot(X_train[:, 1], np.dot(X_train, theta_pinv), '-')

plt.legend(['Training data', 'Linear regression', 'Best theta']);
```



Part 2: Multivariate Linear Regression (30 points)

In most cases, you will deal with databases that have more than one feature. It can be as little as two features and up to thousands of features. In those cases, we use a multiple linear regression model. The regression equation is almost the same as the simple linear regression equation:

$$\hat{y} = h_\theta(\vec{x}) = \theta^T \vec{x} = \theta_0 + \theta_1 x_1 + \ldots + \theta_n x_n$$

If you wrote vectorized code, this part should be straightforward. If your code is not vectorized, you should go back and edit your functions such that they support both multivariate and single variable regression. **Your code should not check the dimensionality of the input before running**.

In [37]:

```
# Read comma separated data
df = pd.read_csv('data.csv')
df.head()
```

Out[37]:

	id	date	price	bedrooms	bathrooms	sq
0	7129300520	20141013T000000	221900.0	3	1.00	
1	6414100192	20141209T000000	538000.0	3	2.25	
2	5631500400	20150225T000000	180000.0	2	1.00	
3	2487200875	20141209T000000	604000.0	4	3.00	
4	1954400510	20150218T000000	510000.0	3	2.00	

Preprocessing

Like in the single variable case, we need to create a numpy array from the dataframe. Before doing so, we should notice that some of the features are clearly irrelevant.

In [38]:

```
X = df.drop(columns=['price', 'id', 'date']).values
y = df['price'].values
```

Use the **same** `preprocess` function you implemented previously. Notice that proper vectorized implementation should work regardless of the dimensionality of the input. You might want to check that your code in the previous parts still works.

In [39]:

```
# preprocessing
X, y = preprocess(X, y)
```

In [40]:

```
# my test
print(y.std())
print(y.mean())
```

```
1.0
9.805489753489382e-17
```

In [41]:

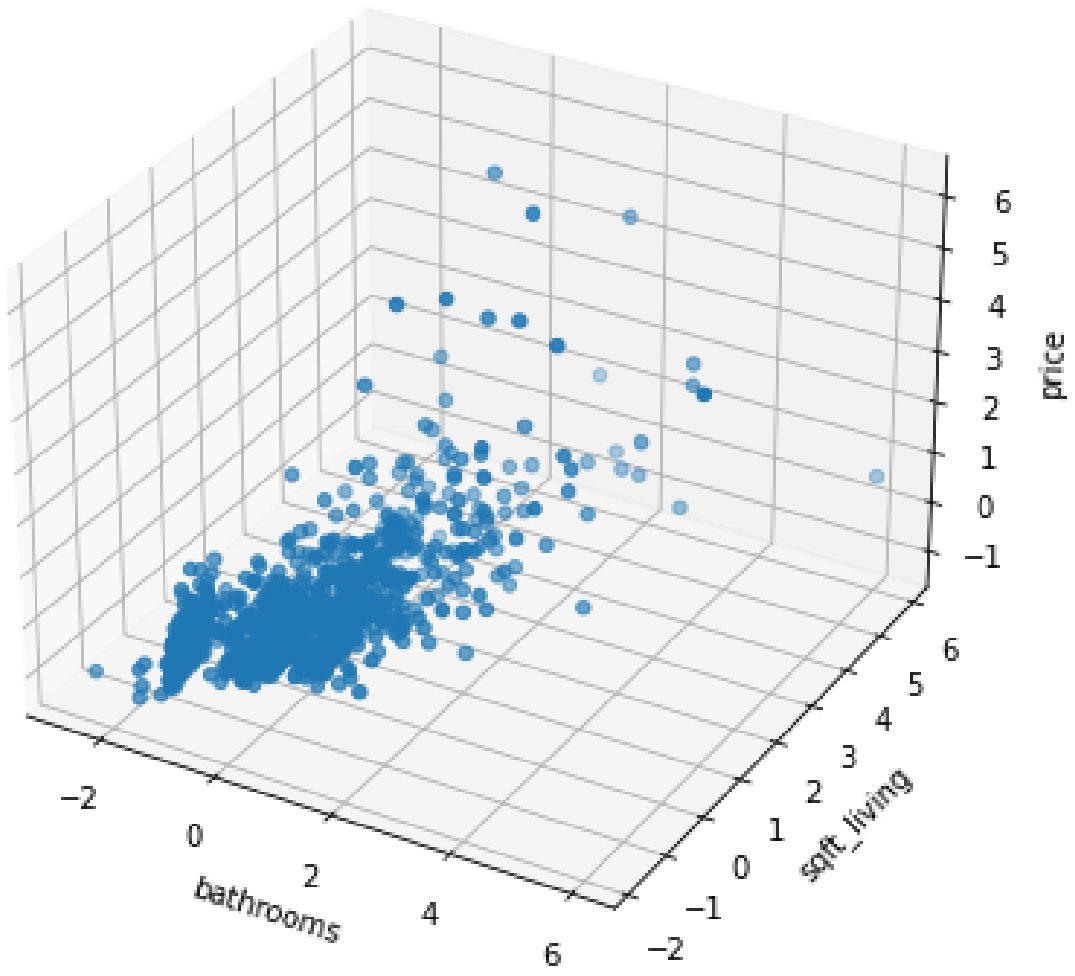
```
# training and validation split
np.random.seed(42)
indices = np.random.permutation(X.shape[0])
idx_train, idx_val = indices[:int(0.8*X.shape[0])], indices[int(0.8*X.s
X_train, X_val = X[idx_train,:], X[idx_val,:]
y_train, y_val = y[idx_train], y[idx_val]
```

Using 3D visualization, we can still observe trends in the data. Visualizing additional

dimensions requires advanced techniques we will learn later in the course.

In [42]:

```
%matplotlib inline
import mpl_toolkits.mplot3d.axes3d as p3
fig = plt.figure(figsize=(5,5))
ax = p3.Axes3D(fig)
xx = X_train[:, 1][:1000]
yy = X_train[:, 2][:1000]
zz = y_train[:1000]
ax.scatter(xx, yy, zz, marker='o')
ax.set_xlabel('bathrooms')
ax.set_ylabel('sqft_living')
ax.set_zlabel('price')
plt.show()
```



Use the bias trick again (add a column of ones as the zeroth column in the both the

training and validation datasets).

In [43]:

```
################################################################################
#                                START OF YOUR CODE
################################################################################
vec_train = np.ones(X_train.shape[0])
vec_train = vec_train[:, None]
X_train = (np.hstack((vec_train, X_train)))

vec_val = np.ones(X_val.shape[0])
vec_val = vec_val[:, None]
X_val = (np.hstack((vec_val, X_val)))
################################################################################
#                                END OF YOUR CODE
################################################################################
```

In [44]:

```
# my test
print(X_val)
```

```
[[ 1.         -1.50652181 -1.37314792 ... -0.6709526
0.4048726
  -0.32631008]
 [ 1.         -0.40799344 -0.08034754 ...  0.04840879 -
0.58820766
  -0.23755814]
 [ 1.         -0.40799344  0.24285255 ... -0.48758597
0.50862725
  -0.01324545]
 ...
 [ 1.         -0.40799344  0.56605264 ... -0.34653472
0.52344934
   0.09674208]
 [ 1.          1.7890633  -0.72674773 ... -0.01506427 -
0.79571697
   0.10990588]
 [ 1.         -2.60505018 -1.69634801 ... -0.75558336 -
1.19591349
   0.05841995]]
```

Make sure the functions `compute_cost` (10 points), `gradient_descent` (15 points), and `pinv` (5 points) work on the multi-dimensional dataset. If you make any changes, make sure your code still works on the single variable regression model.

In [45]:

```
# my test
print(X_train.shape[1])
```

```
18
```

In [46]:

```
alpha_dict = find_best_alpha(X_train, y_train, X_val, y_val, 40000)
```

```
c:\users\feith\pycharmprojects\pythonproject\venv\lib\si
te-packages\numpy\core\fromnumeric.py:87: RuntimeWarnin
g: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwarg
s)
<ipython-input-28-6b7623ceb264>:28: RuntimeWarning: inva
lid value encountered in double_scalars
  while (abs(curr_cost - J_history[len(J_history) - 1])
> 1E-8) and (k < num_iters):
<ipython-input-20-7ebfbf4dec3c>:23: RuntimeWarning: over
flow encountered in multiply
  h_theta *= h_theta
```

In [47]:

```
# my test
print(alpha_dict)
```

```
{1e-05: 0.7527448732759496, 3e-05: 0.3722922348167418,
0.0001: 0.15775506745570503, 0.0003: 0.1380597903033054
8, 0.001: 0.137838536308431, 0.003: 0.13780025037302404,
0.01: 0.1377863144060405, 0.03: 0.13778862577888917, 0.
1: 0.13780486381476698, 0.3: 0.1378277157148855, 1: inf,
2: inf, 3: inf}
```

In [48]:

```
shape = X_train.shape[1]
theta = np.ones(shape)
J = compute_cost(X_train, y_train, theta)
```

In [49]:

```
best_alpha = None
best_alpha = list(alpha_dict.keys())[list(alpha_dict.values()).index(mi
np.random.seed(42)
shape = X_train.shape[1]
theta = np.random.random(shape)
iterations = 40000
theta, J_history = gradient_descent(X_train ,y_train, theta, best_alpha
```

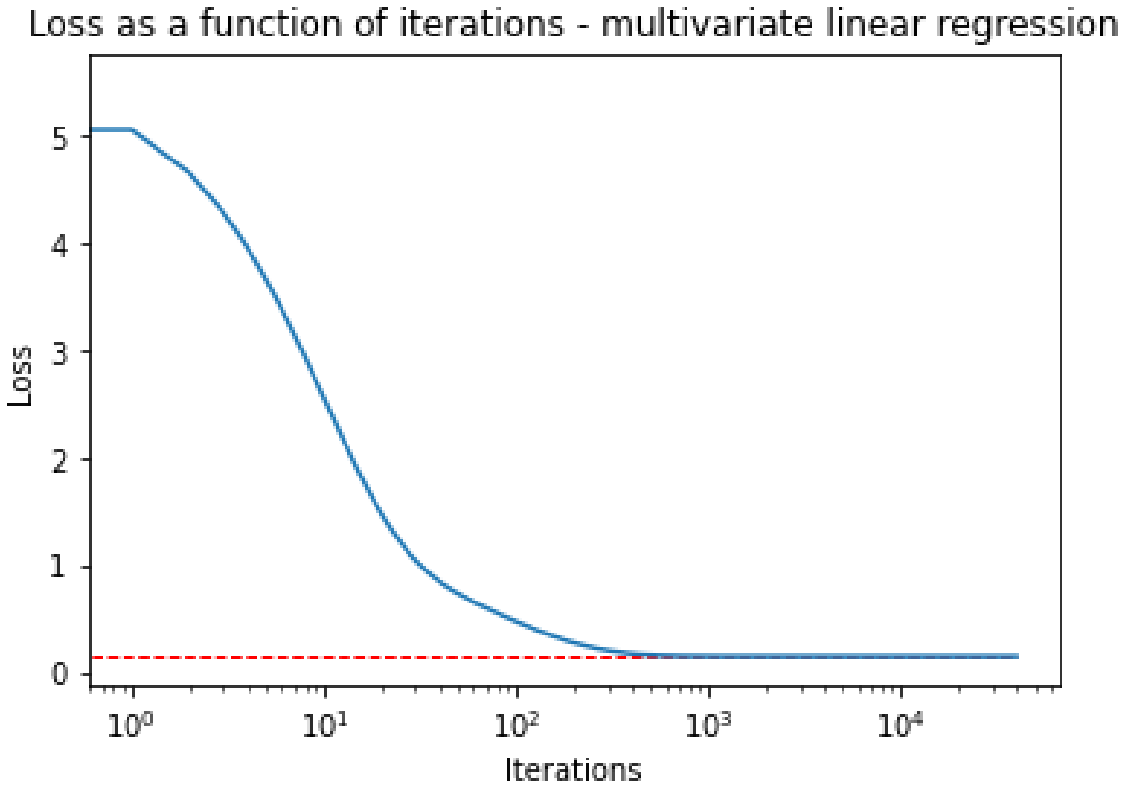
In [50]:

```
theta_pinv = pinv(X_train ,y_train)
J_pinv = compute_cost(X_train, y_train, theta_pinv)
```

We can use visualization to make sure the code works well. Notice we use logarithmic scale for the number of iterations, since gradient descent converges after ~500 iterations.

In [51]:

```
plt.plot(np.arange(iterations), J_history)
plt.xscale('log')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss as a function of iterations - multivariate linear regre
plt.hlines(y = J_pinv, xmin = 0, xmax = len(J_history), color='r',
           linewidth = 1, linestyle = 'dashed')
plt.show()
```



Part 3: Find best features for regression (10 points)

Adding additional features to our regression model makes it more complicated but does not necessarily improves performance. Use forward and backward selection and find 4 features that best minimizes the loss. First, we will reload the dataset as a dataframe in order to access the feature names.

In [52]:

```
columns_to_drop = ['price', 'id', 'date']
all_features = df.drop(columns=columns_to_drop)
all_features.head(5)
```

Out[52]:

	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view
0	3	1.00	1180	5650	1.0	0	0
1	3	2.25	2570	7242	2.0	0	0
2	2	1.00	770	10000	1.0	0	0
3	4	3.00	1960	5000	1.0	0	0
4	3	2.00	1680	8080	1.0	0	0

In [57]:

```
# my test
```

```
<class 'pandas.core.frame.DataFrame'>
```

Forward Feature Selection

Complete the function `forward_selection` . Train the model using a single feature at a time, and choose the best feature using the validation dataset. Next, check which feature performs best when added to the feature you previously chose. Repeat this process until you reach 4 features + bias. You are free to use any arguments you need.

In [ ]:

```
def forward_selection():
    """
    Train the model using the training set using a single feature.
    Choose the best feature according to the validation set. Next,
    check which feature performs best when added to the feature
    you previously chose. Repeat this process until you reach 4
    features and the bias. Don't forget the bias trick.

    Returns:
    - The names of the best features using forward selection.
    """
    np.random.seed(42)
    best_features = None
    ###########################################################################
    # TODO: Implement the function.
    ###########################################################################
    X = all_features.values
    ###########################################################################
    #                                 END OF YOUR CODE
    ###########################################################################
    return best_features
```

Backward Feature Selection

Complete the function `backward_selection` . Train the model with all but one of the features at a time and remove the worst feature (the feature that its absence yields the best loss value using the validation dataset). Next, remove an additional feature along with the feature you previously removed. Repeat this process until you reach 4 features + bias. You are free to use any arguments you need.

In [ ]:

```
def backward_selection():
    """
    Train the model using the training set using all but one of the
    features at a time. Remove the worst feature according to the
    validation set. Next, remove an additional feature along with the
    feature you previously removed. Repeat this process until you
    reach 4 features and the bias. Don't forget the bias trick.

    Returns:
    - The names of the best features using backward selection.
    """
    np.random.seed(42)
    best_features = None
    ###########################################################################
    # TODO: Implement the function.
    ###########################################################################
    pass
    ###########################################################################
    #                             END OF YOUR CODE
    ###########################################################################
    return best_features
```

In [ ]:

```
backward_selection()
```

Give an explanations to the results. Do they make sense?

Use this Markdown cell for your answer

Part 4: Adaptive Learning Rate (10 points)

So far, we kept the learning rate alpha constant during training. However, changing alpha during training might improve convergence in terms of the global minimum found and running time. Implement the adaptive learning rate method based on the gradient

descent algorithm above.

Your task is to find proper hyper-parameter values for the adaptive technique and compare this technique to the constant learning rate. Use clear visualizations of the validation loss and the learning rate as a function of the iteration.

Time based decay: this method reduces the learning rate every iteration according to the following formula:

$$\alpha = \frac{\alpha_0}{1 + D \cdot t}$$

Where α_0 is the original learning rate, D is a decay factor and t is the current iteration.

In [ ]:

```
### Your code here ###
```