Predicting the progression of diabetes using least-squares regression

The **diabetes** data set is provided as a single file, `diabetes-data.csv` . We obtained it at https://web.stanford.edu/~hastie/Papers/LARS/diabetes.data (https://web.stanford.edu/~hastie/Papers/LARS/diabetes.data). For some background information on the data, see this seminal paper:

Bradley Efron, Trevor Hastie, Iain Johnstone and Robert Tibshirani (2004) "Least Angle Regression," Annals of Statistics (with discussion), 407-499.

Set up notebook and load data set

In [2]:

```
# Standard includes
%matplotlib inline
import numpy as np
import sys
import matplotlib
import matplotlib.pyplot as plt
# Routines for linear regression
from sklearn import linear_model
from sklearn.metrics import mean_squared_error
# Set label size for plots
matplotlib.rc('xtick', labelsize=14)
matplotlib.rc('ytick', labelsize=14)
```

This next snippet of code loads in the diabetes data. There are 442 data points, each with 10 predictor variables (which we'll denote `x`) and one response variable (which we'll denote `y`).

Make sure the file `'diabetes-data.csv'` is in the same directory as this notebook.

In [3]:

```
data = np.genfromtxt('diabetes-data.csv', delimiter=',')
features = ['age', 'sex', 'body mass index', 'blood pressure',
            'serum1', 'serum2', 'serum3', 'serum4', 'serum5', 'serum6']
x = data[:,0:10] # predictors
y = data[:,10] # response variable

print("shape of data", np.shape(x))
print("shape of x", np.shape(x))
print("shape of y", np.shape(y))
```

```
shape of data (442, 10)
shape of x (442, 10)
shape of y (442,)
```

Predict `y` without using `x`

If we want to predict `y` without knowledge of `x`, what value would be predict? The mean value of `y`.

In this case, the mean squared error (MSE) associated with the prediction is simply the variance of `y`.

In [4]:

```
print ("Prediction: ", np.mean(y))
print ("Mean squared error: ", np.var(y))
```

```
Prediction:  152.13348416289594
Mean squared error:  5929.884896910383
```

Predict `y` using a single feature of `x`

To fit a linear regression model, we could directly use the formula we saw in lecture. To make things even easier, this is already implemented in `sklearn.linear_model.LinearRegression()`.

Here we define a function, **one_feature_regression**, that takes `x` and `y`, along with the index `f` of a single feature and fits a linear regressor to `(x[f],y)`. It then plots the data along with the resulting line.

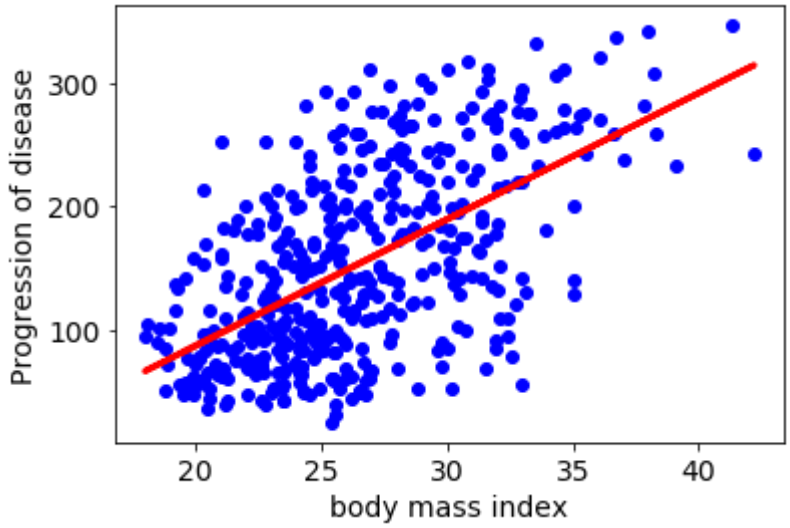
In [5]:

```
def one_feature_regression(x,y,f):
    if (f < 0) or (f > 9):
        print ("Feature index is out of bounds")
        return
    regr = linear_model.LinearRegression()
    x1 = x[:,[f]]
    regr.fit(x1, y)
    # Make predictions using the model
    y_pred = regr.predict(x1)
    # Plot data points as well as predictions
    plt.plot(x1, y, 'bo')
    plt.plot(x1, y_pred, 'r-', linewidth=3)
    plt.xlabel(features[f], fontsize=14)
    plt.ylabel('Progression of disease', fontsize=14)
    plt.show()
    print ("Mean squared error: ", mean_squared_error(y, y_pred))
    return regr
```

Let's try this with feature #2 (body mass index).

In [6]:

```
regr = one_feature_regression(x,y,2)
print ("w = ", regr.coef_)
print ("b = ", regr.intercept_)
```




```
Mean squared error:  3890.456585461273
w =  [10.23312787]
b =  -117.77336656656527
```

For you to try: Feature #2 ('body mass index') is the single feature that yields the lowest mean squared error. Which feature is the second best?

In [7]:

```
### You can use this space to figure out the second-best feature
def feature_MSE(x,y,f):
    if (f < 0) or (f > 9):
        print ("Feature index is out of bounds")
        return
    regr = linear_model.LinearRegression()
    x1 = x[:,[f]]
    regr.fit(x1, y)
    # Make predictions using the model
    y_pred = regr.predict(x1)
    return  mean_squared_error(y, y_pred)

for i in [x for x in range(0,10) ]:
    print( "feature#" +str(i)+ ", MSE = " + str(feature_MSE(x,y,i)))

featuresM= list(features)
del featuresM[2]
MSEs = [feature_MSE(x,y,f) for f in [x for x in range(0,10) if x!=2]]
print("\nsecond best feature is " + featuresM[np.argmin(MSEs)])
```

```
feature#0, MSE = 5720.5470172056475
feature#1, MSE = 5918.888899586022
feature#2, MSE = 3890.456585461273
feature#3, MSE = 4774.113902368687
feature#4, MSE = 5663.315623739354
feature#5, MSE = 5750.241102677782
feature#6, MSE = 5005.661620710652
feature#7, MSE = 4831.13838643409
feature#8, MSE = 4030.9987225912855
feature#9, MSE = 5062.380594520542

second best feature is serum5
```

Predict `y` using a specified subset of features from `x`

The function **feature_subset_regression** is just like **one_feature_regression**, but this time uses a list of features `flist` .

In [8]:

```
def feature_subset_regression(x,y,flist):
    if len(flist) < 1:
        print ("Need at least one feature")
        return
    for f in flist:
        if (f < 0) or (f > 9):
            print ("Feature index is out of bounds")
            return
    regr = linear_model.LinearRegression()
    regr.fit(x[:,flist], y)
    return regr
```

Try using just features #2 (body mass index) and #8 (serum5).

In [9]:

```
flist = [2,8]
regr = feature_subset_regression(x,y,[2,8])
print ("w = ", regr.coef_)
print ("b = ", regr.intercept_)
print ("Mean squared error: ", mean_squared_error(y, regr.predict(x[:,flist])))
```

```
w =  [ 7.27600054 56.05638703]
b =  -299.95751508023613
Mean squared error:  3205.1900768248533
```

Finally, use all 10 features.

In [10]:

```
regr = feature_subset_regression(x,y,range(0,10))
print ("w = ", regr.coef_)
print ("b = ", regr.intercept_)
print ("Mean squared error: ", mean_squared_error(y, regr.predict(x)))
```

```
w =  [-3.63612242e-02 -2.28596481e+01  5.60296209e+00  1.11680799e+00
 -1.08999633e+00  7.46450456e-01  3.72004715e-01  6.53383194e+00
  6.84831250e+01  2.80116989e-01]
b =  -334.5671385187874
Mean squared error:  2859.6963475867506
```

Splitting the data into a training and test set

We define a procedure **split_data** that partitions the data set into separate training and test sets. It is invoked as follows:

- `trainx, trainy, testx, testy = split_data(n_train)`

Here:

- `n_train` is the desired number of training points
- `trainx` and `trainy` are the training points and response values
- `testx` and `testy` are the test points and response values

The split is done randomly, but the random seed is fixed, and thus the same split is produced if the procedure is called repeatedly with the same `n_train` parameter. **Note:** You can also use python built-in libraries for splitting data like: `from sklearn.model_selection import train_test_split`

In [11]:

```
def split_data(n_train):
    if (n_train < 0) or (n_train > 442):
        print ("Invalid number of training points")
        return
    np.random.seed(0)
    perm = np.random.permutation(442)
    training_indices = perm[range(0,n_train)]
    test_indices = perm[range(n_train,442)]
    trainx = x[training_indices,:]
    trainy = y[training_indices]
    testx = x[test_indices,:]
    testy = y[test_indices]
    return trainx, trainy, testx, testy
```

1. Implementing the closed-form solution

To fit a linear regression model, we can directly use the closed-form formula we saw in lecture. Implement a method to get the parameters of the linear regression using the closed-form solution. The method should take features `x` and predictions `y` of the training set and return back the parameter values including the bias term.

In [12]:

```
trainx, trainy, testx, testy = split_data(100)

def linear_regression_CF(trainx, trainy):
    # inputs: trainx and trainy, the features and the target in the training set
    # output: a vector of weights including the bias term

     ### START CODE HERE ###

    # cearting the ones array to be appended to our feature array as the bias
    ones = np.ones(shape=trainy.shape) [..., None]

    # concatenating the ones to the feature array
    trainx = np.concatenate((ones, trainx), 1)

    # W = (X^T . X)^-1 (X^T Y)
    return np.linalg.inv(trainx.transpose().dot(trainx)).dot(trainx.transpose().dot(tra
iny))
    ### END CODE HERE ###

print(linear_regression_CF(trainx, trainy))
```

```
[-5.53865895e+02  3.90990178e-01 -2.86605041e+01  4.05085972e+00
  1.22344601e+00 -4.22191751e+00  4.11113931e+00  3.38820921e+00
  4.54286320e+00  1.23236278e+02  7.06978005e-01]
```

2. Implementing the iterative solution

In this section, you you are required to implement the iterative (gradient descent) solution. The method should take features `x` and predictions `y` of the training set and return back the parameter values including the bias term. You should also initialize the hyper-parameters in the beginning of the method. Also, plot the the cost function at different iterations. Here, the input consists of:

- training data `trainx, trainy`, where `trainx` and `trainy` are numpy arrays of dimension `m`-by-`n` and `m`, respectively (if there are `m` training points and `n` features)

The function should find the `n`-dimensional vector `w` and offset `b` that minimize the MSE loss function, and return:

- `w` and `b`
- `losses`, an array containing the MSE loss at each iteration

Advice: First figure out the derivative, which has a relatively simple form. Next, when implementing gradient descent, think carefully about two issues.

1. What is the step size (learning rate)?
2. When has the procedure converged?

Take the time to experiment with different ways of handling these.

Note: You can use additional methods as helpers if you feel the need.

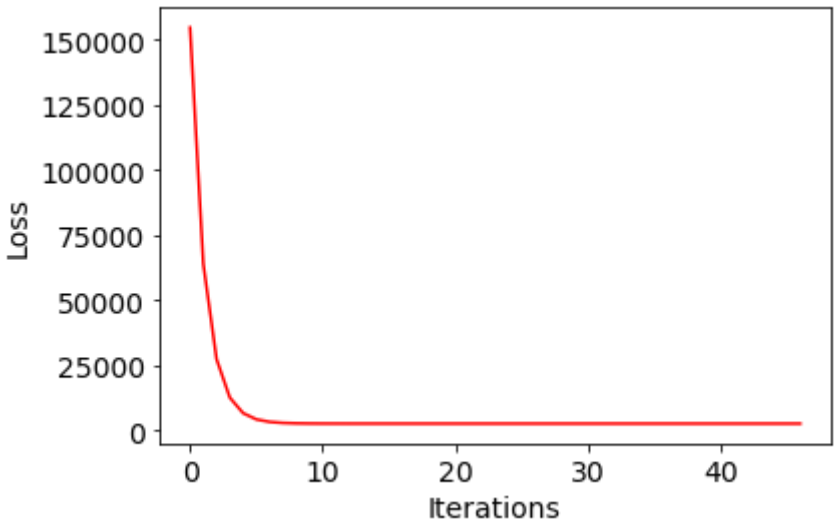
Note: MSE is the RSS value divided by the number of samples to get the mean.

In [13]:

```
# m: the number of samples

def concat_bias_column_as_feature(X):
    """@param X - the training featuers m by n, appends a bias column as 1's to the beg
inning"""
    return np.hstack((np.ones(X.shape[0])[np.newaxis].T, X)) # adding a column of 1's
 (this is for the bias)

def predict(X, W):
    return np.dot(X, W)

def loss(X, Y, W, C=0, ridge=False):
    """
    @param c - the regularization constant
    @param ridge - if True, will compute "Ridge" regularization
    """
    m = X.shape[1]
    Y_hat = predict(X, W)
    reg_term = np.linalg.norm(W) ** 2 if ridge else np.linalg.norm(W)

    return 1.0/2.0 * np.mean((Y - Y_hat) ** 2) + C * reg_term / m


def loss_derivative(X, Y, W, C=0, ridge=False):
    m = X.shape[1]
    reg_term = 2 * np.linalg.norm(W) if ridge else 1

    return -1.0 / m * (np.sum(np.dot(Y - predict(X, W), X)) + C * reg_term)


def regression_GD_general(x, y, C=0, ridge=False, n_epochs=1e4, lr=0.001, epsilon=1e-14
, random_state=None):
    """
    :param x: training data points
    :param y: training labels
    :param C: regularization constant
    :param ridge: boolean, True if ridge regularization, else Lasso
    :param n_epochs:
    :param lr: learning rate
    :param epsilon: stop when delta_loss < epsilon
    :param random_state: random seed for weight initialization
    :return: w, b, training_losses
    """
    # adding the bias in the beginning of the X (to make things simpler)
    X = concat_bias_column_as_feature(x)

    np.random.seed(random_state) # setting random seed for weight initialization
    # creating a weights vector that contains the bias as well
    W = np.random.randn(X.shape[1])
    Y = y
    m = X.shape[1]

#     print("X:", X.shape)
#     print("Y:", Y.shape)
#     print("W:", W.shape)

    last_loss = np.Inf
    training_losses = []
    for e in range(int(n_epochs)):
```

```
        delta_W = loss_derivative(X, y, W, C=C, ridge=ridge)
#         print('delta_W',delta_W.shape)
        err = loss(X, y, W, C=C, ridge=ridge)

        W = W - lr * ( delta_W)  # updating weights

        # print("{}: Loss={}".format(e, err))
        # the bellow is just logging/printing stuff
        training_losses.append(err)
        if np.abs(err - last_loss) < epsilon:
            print(f'STOPPING EARLY at epoch {e} (epsilon={epsilon})')
            break
        else:
            last_loss = err

    w, b = W[1:], W[0] # splitting the joint weights:bias, into 2 variables

    print('final loss={:.5f}'.format(training_losses[-1]))
    return w, b, training_losses


c=0
w, b, losses = regression_GD_general(trainx, trainy,c, lr=0.0000001)

print("Bias:")
print(b)
print("weights:")
print( w )
print("costs:")
print(losses)

plt.plot(losses,'r')
plt.xlabel('Iterations', fontsize=14)
plt.ylabel('Loss', fontsize=14)
plt.show()
```

```
STOPPING EARLY at epoch 46 (epsilon=1e-14)
final loss=2501.75425
Bias:
1.8426294354560415
weights:
[ 1.74663541  0.83572401  1.38651062  1.47557253 -0.4670259  -0.35331728
 -0.14666838  0.28933114 -0.22472769  0.29962886]
costs:
[154750.62740237557, 63914.1628258714, 27273.589003110155, 12493.933538530
599, 6532.285210960415, 4127.543715680141, 3157.546603911602, 2766.2802666
832667, 2608.4557280421564, 2544.7942713123825, 2519.1152412461333, 2508.7
571278912606, 2504.578990633132, 2502.8936614279305, 2502.213852614953, 25
01.9396391136524, 2501.8290300087683, 2501.7844137735174, 2501.76641698699
5, 2501.7591576491486, 2501.756229460502, 2501.755048321267, 2501.75457188
6836, 2501.7543797081653, 2501.7543021893325, 2501.754270920672, 2501.7542
58307878, 2501.7542532202738, 2501.7542511680936, 2501.7542503403088, 250
1.7542500064064, 2501.7542498717216, 2501.754249817393, 2501.754249795479,
2501.754249786639, 2501.7542497830736, 2501.7542497816353, 2501.7542497810
555, 2501.7542497808213, 2501.754249780727, 2501.7542497806885, 2501.75424
97806735, 2501.754249780667, 2501.7542497806644, 2501.754249780664, 2501.7
54249780663, 2501.754249780663]
```

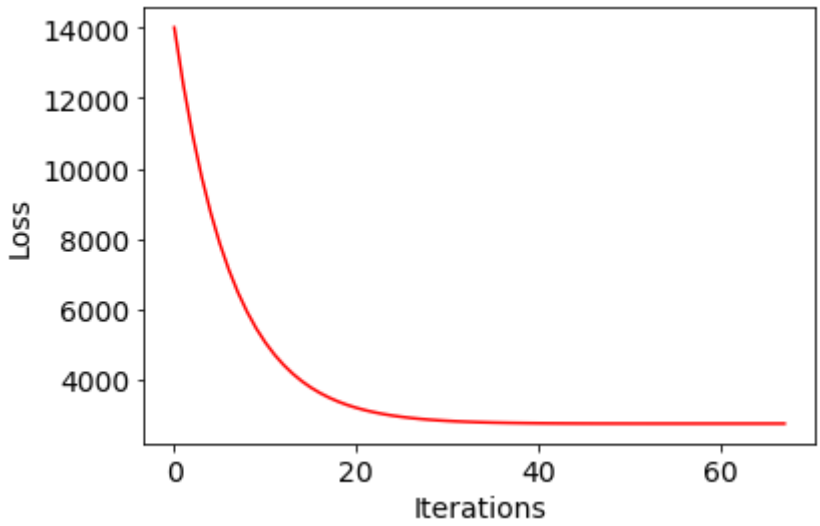


In [18]:

```
# def calc_cost(trainx, trainy, W):
#     squares = np.power((  trainy - (trainx @ W.T)  ), 2)
#     return np.sum(squares) / (len(trainy) * 2)

def linear_regression_GD_vectorized(trainx, trainy):
    # inputs: trainx and trainy, the features and the target in the training set
    # output: a vector of weights including the bias term

    ### START CODE HERE ###

    #initialize learning rate
    LR = 0.000001
    print( "trainy shape ")
    print(trainy.shape)


    # concatenate ones for the bias term
    ones = np.ones([trainy.shape[0], 1])
    trainx= np.concatenate((ones, trainx), axis=1)
    print( "trainx shape ")
    print(trainx.shape)

    # initializing weights + bias to zeros
    W = np.zeros([1, trainx.shape[1]])
    print( "W shape ")
    print(W.shape)

    costs = []
    # append initial cost
    costs.append(loss(trainx, trainy, W.reshape(11,1)))

    for i in range(sys.maxsize):

        # update the weights
        W = W - (LR/len(trainx)) * np.sum(trainx * ((trainx @ W.T) - trainy.reshape(len
(trainy),1)), axis=0)

        # calculate the new cost and append it to costs
        newCost = loss(trainx, trainy, W.reshape(11,1))
        costs.append(newCost)

#         print(newCost - costs[i])
#         print("old cost: "+ str(costs[i]))
#         print("new cost: "+str(newCost))

        # check the difference between the old and the new cost, if less than 0.01 brea
k out
        if abs(newCost - costs[i]) < 0.001:
            break


    # split into bias and weights
    w, b = W[0][1:], W[0][0]

    return b, w, np.array(costs)
    ### END CODE HERE ###


B, W, costs = linear_regression_GD_vectorized(trainx, trainy)
```

```
print("Bias:", B)
print("weights:", W )
print("costs:", costs)

plt.plot(costs,'r')
plt.xlabel('Iterations', fontsize=14)
plt.ylabel('Loss', fontsize=14)
plt.show()
```

```
trainy shape
(100,)
trainx shape
(100, 11)
W shape
(1, 11)
Bias: 0.002049291473899618
weights: [0.1062837  0.00292911 0.06198143 0.21847692 0.38292093 0.2376601
7
 0.08429978 0.00947249 0.01020354 0.19989775]
costs: [13997.27       12362.67980268 10963.75720593  9766.68690566
  8742.49352521  7866.34978014  7116.98347002  6476.16918491
  5928.29262914  5459.9771921   5059.7638764   4717.83696303
  4425.78888131  4176.41868394  3963.55932711  3781.92964109
  3627.00746431  3494.92091729  3382.35522501  3286.47286605
  3204.84514423  3135.39355057  3076.33951619  3026.16135707
  2983.55738245  2947.41428586  2916.78006343  2890.84081209
  2868.90085282  2850.36570331  2834.72749243  2821.55246709
  2810.47029205  2801.16488603  2793.36657419  2786.84536841
  2781.40521388  2776.8790635   2773.12466141  2770.02093407
  2767.46490162  2765.36903506  2763.65899497  2762.2716973
  2761.15365894  2760.25958312  2759.5511499   2758.99598242
  2758.56676347  2758.24048077  2757.99778244  2757.82242664
  2757.70081185  2757.62157614  2757.57525534  2757.55399178
  2757.55128592  2757.56178505  2757.58110327  2757.6056685
  2757.63259239  2757.65955982  2757.68473512  2757.70668266
  2757.72429942  2757.73675813  2757.74345908  2757.74398955]
```

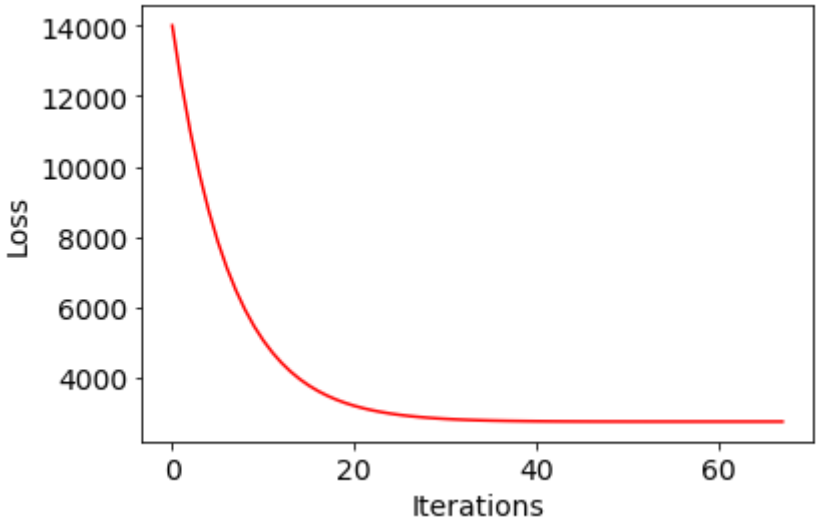


In [19]:

```
def linear_regression_GD_iterative(trainx, trainy):
    # inputs: trainx and trainy, the features and the target in the training set
    # output: a vector of weights including the bias term

    ### START CODE HERE ###

    #initialize learning rate
    LR = 0.000001
    print( "trainy shape ")
    print(trainy.shape)


    # concatenate ones for the bias term
    ones = np.ones([trainy.shape[0], 1])
    trainx= np.concatenate((ones, trainx), axis=1);
    print( "trainx shape ")
    print(trainx.shape)

    # initializing weights + bias to zeros
    W = np.zeros([1, trainx.shape[1]])
    print( "W shape ")
    print(W.shape)

    costs = []
    # append initial cost
    costs.append(loss(trainx, trainy, W.reshape(11,1)))

    for i in range(sys.maxsize):

        # features*weights = y_predictions =  XW (including bias)
        y_predicted = trainx.dot(W.transpose())
#         y_predicted = y_predicted[0]
#         print("y_predicted")
#         print(y_predicted.shape)


        # y_predicted - y_actual
        differences = []
        for x in range(len(trainy)):
            differences.append(y_predicted[x] - trainy[x])

        differences = np.array(differences)
#         print("differences")
#         print(differences.shape)
#         print(differences)

        # X ( y_predicted - y_actual)
        derivatives = []
        for j in range(len(trainy)):
            derivatives.append(trainx[j] * differences[j] )

        derivatives = np.array(derivatives)
#         print("derivatives")
#         print(derivatives.shape)


        derivative_sum = np.sum(derivatives, axis=0)
#         print("derivative_sum")
#         print(derivative_sum.shape)
```

```
#         # update weights:
        W = W - (LR/len(trainy)) * derivative_sum



        # calculate the new cost and append it to costs
        newCost = loss(trainx, trainy, W.reshape(11,1))
        costs.append(newCost)

#         print("old cost: "+ str(costs[i]))
#         print("new cost: "+str(newCost))

        # check the difference between the old and the new cost, if less than 0.01 brea
k out
        if abs(newCost - costs[i]) < 0.001:
            break

    # split into bias and weights
    w, b = W[0][1:], W[0][0]

    return b, w, np.array(costs)
    ### END CODE HERE ###

B_iterative, W_iterative, cost_iterative = linear_regression_GD_iterative(trainx, train
y)
print("Bias:")
print(B_iterative)
print("weights:")
print( W_iterative )
print("costs:")
print(cost_iterative)

plt.plot(cost_iterative,'r')
plt.xlabel('Iterations', fontsize=14)
plt.ylabel('Loss', fontsize=14)
plt.show()
```

```
trainy shape
(100,)
trainx shape
(100, 11)
W shape
(1, 11)
Bias:
0.002049291473899618
weights:
[0.1062837  0.00292911 0.06198143 0.21847692 0.38292093 0.23766017
 0.08429978 0.00947249 0.01020354 0.19989775]
costs:
[13997.27       12362.67980268 10963.75720593  9766.68690566
  8742.49352521  7866.34978014  7116.98347002  6476.16918491
  5928.29262914  5459.9771921   5059.7638764   4717.83696303
  4425.78888131  4176.41868394  3963.55932711  3781.92964109
  3627.00746431  3494.92091729  3382.35522501  3286.47286605
  3204.84514423  3135.39355057  3076.33951619  3026.16135707
  2983.55738245  2947.41428586  2916.78006343  2890.84081209
  2868.90085282  2850.36570331  2834.72749243  2821.55246709
  2810.47029205  2801.16488603  2793.36657419  2786.84536841
  2781.40521388  2776.8790635   2773.12466141  2770.02093407
  2767.46490162  2765.36903506  2763.65899497  2762.2716973
  2761.15365894  2760.25958312  2759.5511499   2758.99598242
  2758.56676347  2758.24048077  2757.99778244  2757.82242664
  2757.70081185  2757.62157614  2757.57525534  2757.55399178
  2757.55128592  2757.56178505  2757.58110327  2757.6056685
  2757.63259239  2757.65955982  2757.68473512  2757.70668266
  2757.72429942  2757.73675813  2757.74345908  2757.74398955]
```



3. Use diffrent amounts of training data to fit the model

Using the **split_data** procedure to partition the data set, compute the training MSE and test MSE when fitting a regressor to *all* features, for the following training set sizes:

- `n_train = 20`
- `n_train = 50`
- `n_train = 100`
- `n_train = 200`
- `n_train = 300`

1. Compare your results for the three approaches, i.e., using library, using closed-form solution, and using the iterative solution. Provide your comments on the results.
2. Compare the parameter values for the three solutions when using `n_train = 300` training samples.

Analytical Questions:

1. What changes you need to do if the unit of `y` is different?
2. What changes you need to do if the unit of one of the features was different? For example if **age** was in months and not in years.
3. What if both 3 and 4 apply?

In [20]:

```
#function that coputes mse from w,b,x,y
def compute_mse(w,b,x,y):
    ### START CODE HERE ###
    X = concat_bias_column_as_feature(x) # adding the bias in the beginning of the X (t
o make things simpler)
    W = np.hstack((b, w))
    m = X.shape[1]

    Y_hat = predict(X, W)
    return 1.0/(2.0 * m) * np.sum((y - Y_hat) ** 2)
```

In [21]:

```
C=0
#using training set of 20
print("computing the training MSE and test MSE when fitting a regressor to all features
with a data set of 20")
trainx, trainy, testx, testy = split_data(20)
regr = feature_subset_regression(trainx,trainy,range(0,10))
CF=linear_regression_CF(trainx, trainy)
CF_w, CF_b = CF[1:], CF[0] # splitting the joint weights:bias, into 2 variables
w, b, losses = regression_GD_general(trainx, trainy, C, n_epochs=100, lr=0.0000001)

#training
print("MSE of built-in linear regression(Training), data(20):", mean_squared_error(trai
ny, regr.predict(trainx)))
print("MSE of closed-form solution (Training), data(20)     :", compute_mse(CF_w,CF_b,t
rainx, trainy))
print("MSE of gradient descent solver (Training), data(20)  :", compute_mse(w,b,trainx,
trainy))

#testing
print("MSE of built-in linear regression(Testing ), data(20):", mean_squared_error(test
y, regr.predict(testx)))
print("MSE of closed-form solution (Training), data(20)     :", compute_mse(CF_w,CF_b,t
estx, testy))
print("MSE of gradient descent solver (Training), data(20)  :", compute_mse(w,b,testx,
testy))

#using training set of 50
print("\ncomputing the training MSE and test MSE when fitting a regressor to all featur
es with a data set of 50")
trainx, trainy, testx, testy = split_data(50)
regr = feature_subset_regression(trainx,trainy,range(0,10))
CF=linear_regression_CF(trainx, trainy)
CF_w, CF_b = CF[1:], CF[0] # splitting the joint weights:bias, into 2 variables
w, b, losses = regression_GD_general(trainx, trainy, C, n_epochs=100, lr=0.0000001)

#training
print("MSE of built-in linear regression(Training), data(50):", mean_squared_error(trai
ny, regr.predict(trainx)))
print("MSE of closed-form solution (Training), data(50)     :", compute_mse(CF_w,CF_b,t
rainx, trainy))
print("MSE of gradient descent solver (Training), data(50)  :", compute_mse(w,b,trainx,
trainy))

#testing
print("MSE of built-in linear regression(Testing ), data(50):", mean_squared_error(test
y, regr.predict(testx)))
print("MSE of closed-form solution (Training), data(50)     :", compute_mse(CF_w,CF_b,t
estx, testy))
print("MSE of gradient descent solver (Training), data(50)  :", compute_mse(w,b,testx,
testy))

#using training set of 100
print("\ncomputing the training MSE and test MSE when fitting a regressor to all featur
es with a data set of 100")
trainx, trainy, testx, testy = split_data(100)
regr = feature_subset_regression(trainx,trainy,range(0,10))
CF=linear_regression_CF(trainx, trainy)
CF_w, CF_b = CF[1:], CF[0] # splitting the joint weights:bias, into 2 variables
w, b, losses = regression_GD_general(trainx, trainy, C, n_epochs=100, lr=0.0000001)
```

```
#training
print("MSE of built-in linear regression(Training), data(100):", mean_squared_error(tra
iny, regr.predict(trainx)))
print("MSE of closed-form solution (Training), data(100)     :", compute_mse(CF_w,CF_b,
trainx, trainy))
print("MSE of gradient descent solver (Training), data(100)  :", compute_mse(w,b,trainx
, trainy))

#testing
print("MSE of built-in linear regression(Testing ), data(100):", mean_squared_error(tes
ty, regr.predict(testx)))
print("MSE of closed-form solution (Training), data(100)     :", compute_mse(CF_w,CF_b,
testx, testy))
print("MSE of gradient descent solver (Training), data(100)  :", compute_mse(w,b,testx,
testy))

#using training set of 200
print("\ncomputing the training MSE and test MSE when fitting a regressor to all featur
es with a data set of 200")
trainx, trainy, testx, testy = split_data(200)
regr = feature_subset_regression(trainx,trainy,range(0,10))
CF=linear_regression_CF(trainx, trainy)
CF_w, CF_b = CF[1:], CF[0] # splitting the joint weights:bias, into 2 variables
w, b, losses = regression_GD_general(trainx, trainy, C, n_epochs=100, lr=0.0000001)

#training
print("MSE of built-in linear regression(Training), data(200):", mean_squared_error(tra
iny, regr.predict(trainx)))
print("MSE of closed-form solution (Training), data(200)     :", compute_mse(CF_w,CF_b,
trainx, trainy))
print("MSE of gradient descent solver (Training), data(200)  :", compute_mse(w,b,trainx
, trainy))

#testing
print("MSE of built-in linear regression(Testing ), data(200):", mean_squared_error(tes
ty, regr.predict(testx)))
print("MSE of closed-form solution (Training), data(200)     :", compute_mse(CF_w,CF_b,
testx, testy))
print("MSE of gradient descent solver (Training), data(200)  :", compute_mse(w,b,testx,
testy))

#using training set of 300
print("\ncomputing the training MSE and test MSE when fitting a regressor to all featur
es with a data set of 300")
trainx, trainy, testx, testy = split_data(300)
regr = feature_subset_regression(trainx,trainy,range(0,10))
CF=linear_regression_CF(trainx, trainy)
CF_w, CF_b = CF[1:], CF[0] # splitting the joint weights:bias, into 2 variables
w, b, losses = regression_GD_general(trainx, trainy, C, n_epochs=100, lr=0.0000001)

#training
print("MSE of built-in linear regression(Training), data(300):", mean_squared_error(tra
iny, regr.predict(trainx)))
print("MSE of closed-form solution (Training), data(300)     :", compute_mse(CF_w,CF_b,
trainx, trainy))
print("MSE of gradient descent solver (Training), data(300)  :", compute_mse(w,b,trainx
, trainy))

#testing
print("MSE of built-in linear regression(Testing ), data(300):", mean_squared_error(tes
```

```
ty, regr.predict(testx)))
print("MSE of closed-form solution (Training), data(300)     :", compute_mse(CF_w,CF_b,
testx, testy))
print("MSE of gradient descent solver (Training), data(300)  :", compute_mse(w,b,testx,
testy))
```

```
computing the training MSE and test MSE when fitting a regressor to all fe
atures with a data set of 20
final loss=2310.40238
MSE of built-in linear regression(Training), data(20): 1636.1425922669437
MSE of closed-form solution (Training), data(20)     : 1487.402356606312
MSE of gradient descent solver (Training), data(20)  : 4200.731313536892
MSE of built-in linear regression(Testing ), data(20): 6764.187243690177
MSE of closed-form solution (Training), data(20)     : 129749.40985660281
MSE of gradient descent solver (Training), data(20)  : 90890.92885300855

computing the training MSE and test MSE when fitting a regressor to all fe
atures with a data set of 50
STOPPING EARLY at epoch 94 (epsilon=1e-14)
final loss=2424.82103
MSE of built-in linear regression(Training), data(50): 2414.3515888695906
MSE of closed-form solution (Training), data(50)     : 5487.1627019763255
MSE of gradient descent solver (Training), data(50)  : 11021.913756674812
MSE of built-in linear regression(Testing ), data(50): 7991.270972912112
MSE of closed-form solution (Training), data(50)     : 142389.9191520407
MSE of gradient descent solver (Training), data(50)  : 108482.38943939838

computing the training MSE and test MSE when fitting a regressor to all fe
atures with a data set of 100
STOPPING EARLY at epoch 42 (epsilon=1e-14)
final loss=3115.34301
MSE of built-in linear regression(Training), data(100): 2883.778520121509
MSE of closed-form solution (Training), data(100)     : 13108.084182370492
MSE of gradient descent solver (Training), data(100)  : 28321.300091205616
MSE of built-in linear regression(Testing ), data(100): 3583.0085115303177
MSE of closed-form solution (Training), data(100)     : 55699.495951881254
MSE of gradient descent solver (Training), data(100)  : 121654.4791441117

computing the training MSE and test MSE when fitting a regressor to all fe
atures with a data set of 200
STOPPING EARLY at epoch 17 (epsilon=1e-14)
final loss=2748.15618
MSE of built-in linear regression(Training), data(200): 2858.8241614696626
MSE of closed-form solution (Training), data(200)     : 25989.310558815116
MSE of gradient descent solver (Training), data(200)  : 49966.475937998475
MSE of built-in linear regression(Testing ), data(200): 3028.472046591937
MSE of closed-form solution (Training), data(200)     : 33313.19251250897
MSE of gradient descent solver (Training), data(200)  : 74773.95680065462

computing the training MSE and test MSE when fitting a regressor to all fe
atures with a data set of 300
STOPPING EARLY at epoch 10 (epsilon=1e-14)
final loss=2568.29320
MSE of built-in linear regression(Training), data(300): 2905.187852860912
MSE of closed-form solution (Training), data(300)     : 39616.19799355788
MSE of gradient descent solver (Training), data(300)  : 70044.35996982417
MSE of built-in linear regression(Testing ), data(300): 2877.9542154612873
MSE of closed-form solution (Training), data(300)     : 18575.886299792895
MSE of gradient descent solver (Training), data(300)  : 37077.80557957066
```

In [ ]:

```
```