In [161]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

Assignment 2

Name: sh2432

Due: 26th Sept, 11:59pm

Problem 1: Spam, wonderful spam!

The dataset consists of a collection of 57 features relating to about 4600 emails and a label of whether or not the email is considered spam. You have a training set containing about 70% of the data and a test set containing about 30% of the data. Your job is to build effective spam classification rules using the predictors.

A Note about Features

The column names (in the first row of each .csv file) are fairly self-explanatory.

- Some variables are named `word_freq_(word)` , which suggests a calculation of the frequency of how many times a specific word appears in the email, expressed as a percentage of total words in the email multiplied by 100.
- Some variables are named `char_freq_(number)` , which suggests a count of the frequency of the specific ensuing character, expressed as a percentage of total characters in the email multiplied by 100. Note, these characters are not valid column names in R, but you can view them in the raw .csv file.
- Some variables are named `capital_run_length_(number)` which suggests some information about the average (or maximum length of, or total) consecutive capital letters in the email.
- `spam` : This is the response variable, 0 = not spam, 1 = spam.

Missing Values

Unfortunately, the `capital_run_length_average` variable is corrupted and as a result, contains a fair number of missing values. These show up as `NaN` (the default way of representing missing values in Python.)

Part a

Use k-nearest neighbors regression with $k = 15$ to **impute** the missing values in the `capital_run_length_average` column using the other predictors after standardizing (i.e. rescaling) them. You may use a function such as *KNeighborsRegressor* from the package *sklearn.neighbors* that performs k-nearest neighbors regression. There is no penalty for using a built-in function.

When you are done with this part, you should have no more NaN's in the `capital_run_length_average` column in either the training or the test set. To keep the training and test sets separate, you will need to build two models for imputing: one that is trained on, and imputes for, the training set, and another that is trained on, and imputes for, the test set. Make sure you show all of your work. (You may find the function *np.isnan()* useful for this problem.)

In [333]:
```
train = pd.read_csv("spam_train.csv")
test = pd.read_csv("spam_test.csv")
```

In [334]:
```
from sklearn.neighbors import KNeighborsRegressor as knr
```

In [335]:
```
#separate NAN
train1 = train[np.isnan(train["capital_run_length_average"])]
train2 = train[~np.isnan(train["capital_run_length_average"])]
test1 = test[np.isnan(test["capital_run_length_average"])]
test2 = test[~np.isnan(test["capital_run_length_average"])]
```

In [336]:
```
#create predictors for KNN
train1_y=train1["capital_run_length_average"]
train2_y=train2["capital_run_length_average"]
train1_x=train1.drop(columns=['capital_run_length_average', 'spam'])
train2_x=train2.drop(columns=['capital_run_length_average','spam'])

test1_y=test1["capital_run_length_average"]
test2_y=test2["capital_run_length_average"]
test1_x=test1.drop(columns=['capital_run_length_average'])
test2_x=test2.drop(columns=['capital_run_length_average'])
```

In [337]:
```
#standardize all predictors
from sklearn import preprocessing as pr
scaler = pr.StandardScaler()
train1_xs=scaler.fit_transform(train1_x)
train2_xs=scaler.fit_transform(train2_x)

test1_xs=scaler.fit_transform(test1_x)
test2_xs=scaler.fit_transform(test2_x)
```

In [500]:
```
#check the distribution (can skip this step)
#plt.hist(test2["capital_run_length_longest"])
#bin_values = np.arange(start=0, stop=20, step=1)
#plt.hist(test2_xs[-2])
```

In [339]:
```
#use KNN to predict nan
knn = knr(n_neighbors=15)
knn.fit(train2_xs, train2_y)
train1_pre = knn.predict(train1_xs)
len(train1_pre)

knn.fit(test2_xs, test2_y)
test1_pre = knn.predict(test1_xs)
len(test1_pre)
```

Out[339]:
```
259
```

In [341]:
```
#get the new train and test datasets with no NAN
train1.loc[: ,['capital_run_length_average']]= train1_pre
train_new=pd.concat([train1, train2], axis=0)

test1.loc[: ,['capital_run_length_average']]= test1_pre
test_new=pd.concat([test1, test2], axis=0)

print("Number of rows in train data: ", len(train_new))
print("Number of rows in test data: ",len(test_new))
```

```
Number of rows in train data:  3220
Number of rows in test data:  1381
```

In [342]:
```
#order by index
test_new=test_new.sort_index()
train_new=train_new.sort_index()
```

Part b

Write a function named `knnclass()` that performs k-nearest neighbors classification, without resorting to a package. Essentially, we are asking you to recreate the *sklearn.neighbors.KNeighborsClassifier* function; though, we do not expect you to implement a fancy nearest neighbor search algorithm like what *KNeighborsClassifier* uses, just the naive search will suffice. Additionally, this function will be more sophisticated in the following way:

- The function should automatically do a split of the training data into a sub-training set (80%) and a validation set (20%) for selecting the optimal k.(More sophisticated cross-validation is not necessary.)
- The function should standardize each column: for a particular variable, say x_1, compute the mean and standard deviation of x_1 **using the training set only**, say $\bar{x}_1$ and s_1; then for each observed x_1 in the training set and test set, subtract $\bar{x}_1$, then divide by s_1.

Note: You can assume that all columns will be numeric and that Euclidean distance is the distance measure.

The function skeleton is provided below.

In [343]:
```
def knnclass(xtrain, xtest, ytrain):
    #randomly split to 80:20
    #use package to split the train dataset
    ##from sklearn.model_selection import train_test_split
    ##subxtrain, subxtest, subytrain, subytest = train_test_split(xtrain, ytra
in, test_size=0.2)
    from random import sample
    a=xtrain.shape[0]
    select=[i for i in range(1, a)]
    l=sample(select, int(a*0.8))
    subxtrain=xtrain.iloc[l,:]
    subxtest=xtrain[~xtrain.index.isin(l)]
    subytrain=ytrain.iloc[l]
    subytest=ytrain[~ytrain.index.isin(l)]
    #standardization
    ##use subtrain data mean and sd to standardize subtest data
    m1=subxtrain.mean()
    sd1=subxtrain.std()
    s_subxtrain=(subxtrain-m1)/sd1
    s_subxtest=(subxtest-m1)/sd1
    #calculate the euclidean distance
    d = np.empty((len(s_subxtest), len(s_subxtrain)))
    for i in range(0,len(s_subxtest)):
        for j in range(0,len(s_subxtrain)):
            d[i,j]=np.sqrt(np.sum((s_subxtest.iloc[i,:] - s_subxtrain.iloc[j,
:]) ** 2))
    error_list = []
    for k in range(1,101):
        list_k = []
        for ii in range(0,len(d)):
        #predict y row by row
            ed = pd.DataFrame(d)
            subytrain=pd.DataFrame(subytrain)
            subytest=pd.DataFrame(subytest)
            d1 = pd.DataFrame(ed.iloc[ii,:]) # select each test point
            d1.columns = ["distances"]
            dy = pd.DataFrame(subytrain["spam"]) # find spam values
            d1.index = dy.index   ######
            #sort by distance
            d1["spam"] = dy["spam"]
            d1 = d1.sort_values(by=['distances'])
            # classification
            d1_k = (sum(d1["spam"][0:k]) >= 0.5*k) * 1
            list_k.append(d1_k)
        list_k = pd.DataFrame(list_k)
        list_k.columns = ["predict"]
        # get spam values
        list_test = pd.DataFrame(subytest["spam"])
        list_test.columns = ["spam"]
        list_k.index = list_test.index ####
        # calculate error rate
        list_k["spam"] = list_test["spam"]
        list_k["compare"] = (list_k["spam"]== list_k["predict"]) *1
        error = 1 - sum(list_k["compare"])/len(list_k)
        error_list.append(error)
        #optimal k: k with min error rate
```

```
        ok=error_list.index(min(error_list)) +1
    #standardization
    ##use train data mean and sd to standardize test data
    m2=xtrain.mean()
    sd2=xtrain.std()
    s_xtrain=(xtrain-m2)/sd2
    s_xtest=(xtest-m2)/sd2
    #knn
    dist = np.empty((len(s_xtest), len(s_xtrain))) #calculate the euclidean di
stance
    for m in range(0,len(s_xtest)):
        for n in range(0,len(s_xtrain)):
            dist[m,n]=np.sqrt(np.sum((s_xtest.iloc[m,:] - s_xtrain.iloc[n, :])
** 2))
        ytest=[]
        for p in range(0,len(dist)):
            #predict y row by row
            edist = pd.DataFrame(dist)
            ytrain=pd.DataFrame(ytrain)
            dist1 = pd.DataFrame(edist.iloc[p,:]) # select each test point
            dist1.columns = ["distances"]
            disty = pd.DataFrame(ytrain["spam"]) # find spam values
            d1.index = dy.index   ######
            #sort by distance
            dist1["spam"] = disty["spam"]
            dist1 = dist1.sort_values(by=['distances'])
            #classification
            dist1_k = (sum(dist1["spam"][0:ok]) >= 0.5*ok) * 1
            ytest.append(dist1_k)
        ytest = pd.DataFrame(ytest)
        ytest.columns = ["predict"]
    return(ytest)
```

Part c

In this part, you will need to use a k-NN classifier to fit models on the actual dataset. If you weren't able to successfully write a k-NN classifier in Part b, you're permitted to use a built-in package for it. If you take this route, you may need to write some code to standardize the variables and select k, which `knnclass()` from part b already does.

Now fit 4 models and produce 4 sets of predictions of `spam` on the test set:

1. `knnclass()` using all predictors except for `capital_run_length_average` (say, if we were distrustful of our imputation approach). Call these predictions `knn_pred1` .
2. `knnclass()` using all predictors including `capital_run_length_average` with the imputed values. Call these predictions `knn_pred2` .
3. logistic regression using all predictors except for `capital_run_length_average` . Call these predictions `logm_pred1` .
4. logistic regression using all predictors including `capital_run_length_average` with the imputed values. Call these predictions `logm_pred2` .

In 3-4 sentences, provide a quick summary of your second logistic regression model (model 4). Which predictors appeared to be most significant? Are there any surprises in the predictors that ended up being significant or not significant?

Submit a .csv file called `assn2_NETID_results.csv` that contains 5 columns:

- `capital_run_length_average` : the predictor in your test set that now contains the imputed values (so that we can check your work on imputation).
- `knn_pred1`
- `knn_pred2`
- `logm_pred1`
- `logm_pred2`

Make sure that row 1 here corresponds to row 1 of the test set, row 2 corresponds to row 2 of the test set, and so on.

In [9]:
```
#knn_pred1:
train1 = pd.read_csv("spam_train.csv")
xtrain1=train1.drop(columns=['capital_run_length_average','spam'])
ytrain1=train1['spam']
test1=pd.read_csv("spam_test.csv")
xtest1=test1.drop(columns=['capital_run_length_average'])
```

In [10]:
```
knn_pred1=knnclass(xtrain1, xtest1, ytrain1)
```

In [86]:
```
knn_pred1.rename(columns={'predict':'knn_pred1'}, inplace=True)
knn_pred1.shape
```

Out[86]: (1381, 1)

In [346]:
```
#knn_pred2: use the train and test datasets gotten from 1a (nan replaced by pr
edicted values)
xtrain2=train_new.drop(columns=['spam'])
ytrain2=train_new['spam']
xtest2=test_new
```

In [347]:
```
knn_pred2=knnclass(xtrain2, xtest2, ytrain2)
```

In [348]:
```
knn_pred2.rename(columns={'predict':'knn_pred2'}, inplace=True)
knn_pred2.shape
```

Out[348]: (1381, 1)

In [350]:
```
#supress warnings from code
import warnings
warnings.filterwarnings("ignore")
```

In [351]:
```
#logistic regression
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
#create an instance and fit the model
#logm_pred1
logm1 = LogisticRegression()
logm1.fit(xtrain1, ytrain1)
logm_pred1= logm1.predict(xtest1)
logm_pred1=pd.DataFrame(logm_pred1)
logm_pred1.columns=['logm_pred1']
logm_pred1.shape
```

Out[351]: (1381, 1)

In [352]:
```
#logm_pred2
logm2 = LogisticRegression()
logm2.fit(xtrain2, ytrain2)
logm_pred2= logm2.predict(xtest2)
logm_pred2=pd.DataFrame(logm_pred2)
logm_pred2.columns=['logm_pred2']
logm_pred2.shape
```

Out[352]: (1381, 1)

In [353]:
```
#look at the logm_pred2 outputs
import statsmodels.api as sm
logit_model=sm.Logit(ytrain2,xtrain2)
result=logit_model.fit()
print(result.summary2())
```

```
Optimization terminated successfully.
         Current function value: 0.219940
         Iterations 15
                             Results: Logit
============================================================================
Model:                 Logit               Pseudo R-squared:     0.672
Dependent Variable:    spam                AIC:                  1530.4139
Date:                  2019-09-26 07:49    BIC:                  1876.8107
No. Observations:      3220                Log-Likelihood:       -708.21
Df Model:              56                  LL-Null:              -2158.3
Df Residuals:          3163                LLR p-value:          0.0000
Converged:             1.0000              Scale:                1.0000
No. Iterations:        15.0000
----------------------------------------------------------------------------
                            Coef.  Std.Err.    z    P>|z|   [0.025   0.975]
----------------------------------------------------------------------------
word_freq_make             -0.4221   0.2231 -1.8922 0.0585  -0.8592  0.0151
word_freq_address          -0.2456   0.0750 -3.2730 0.0011  -0.3927 -0.0985
word_freq_all              -0.0027   0.1311 -0.0209 0.9833  -0.2596  0.2542
word_freq_3d                2.0633   1.9025  1.0845 0.2781  -1.6655  5.7921
word_freq_our               0.3401   0.1187  2.8648 0.0042   0.1074  0.5728
word_freq_over              0.5013   0.2960  1.6935 0.0904  -0.0789  1.0814
word_freq_remove            1.6509   0.3293  5.0134 0.0000   1.0055  2.2963
word_freq_internet          0.4905   0.1969  2.4916 0.0127   0.1046  0.8763
word_freq_order             0.2567   0.2995  0.8570 0.3915  -0.3303  0.8437
word_freq_mail              0.1575   0.1073  1.4680 0.1421  -0.0528  0.3679
word_freq_receive          -0.0610   0.3317 -0.1840 0.8540  -0.7111  0.5890
word_freq_will             -0.2948   0.0811 -3.6372 0.0003  -0.4537 -0.1360
word_freq_people           -0.4371   0.2542 -1.7193 0.0856  -0.9354  0.0612
word_freq_report           -0.0034   0.1500 -0.0226 0.9820  -0.2973  0.2905
word_freq_addresses         0.9028   0.7268  1.2423 0.2141  -0.5216  2.3273
word_freq_free              0.6786   0.1586  4.2787 0.0000   0.3678  0.9895
word_freq_business          0.8309   0.2643  3.1433 0.0017   0.3128  1.3490
word_freq_email             0.0671   0.1408  0.4762 0.6339  -0.2089  0.3430
word_freq_you              -0.0925   0.0386 -2.3951 0.0166  -0.1683 -0.0168
word_freq_credit            1.0721   0.6487  1.6527 0.0984  -0.1993  2.3435
word_freq_your              0.1958   0.0633  3.0948 0.0020   0.0718  0.3199
word_freq_font              0.2227   0.2104  1.0581 0.2900  -0.1898  0.6351
word_freq_000               2.2071   0.5522  3.9971 0.0001   1.1249  3.2894
word_freq_money             0.8909   0.3683  2.4188 0.0156   0.1690  1.6127
word_freq_hp               -1.9385   0.3219 -6.0217 0.0000  -2.5694 -1.3075
word_freq_hpl              -1.0879   0.4604 -2.3627 0.0181  -1.9904 -0.1854
word_freq_george           -7.7056   1.8608 -4.1411 0.0000 -11.3526 -4.0585
word_freq_650               0.3288   0.2328  1.4123 0.1579  -0.1275  0.7851
word_freq_lab              -2.8708   1.5910 -1.8044 0.0712  -5.9891  0.2474
word_freq_labs             -0.4741   0.3284 -1.4435 0.1489  -1.1178  0.1696
word_freq_telnet           -0.4474   1.5516 -0.2883 0.7731  -3.4884  2.5936
word_freq_857               2.0650   3.3584  0.6149 0.5386  -4.5174  8.6474
word_freq_data             -0.8917   0.3417 -2.6098 0.0091  -1.5613 -0.2220
word_freq_415              -0.0146   1.4926 -0.0098 0.9922  -2.9400  2.9108
word_freq_85               -1.9402   0.8385 -2.3139 0.0207  -3.5835 -0.2968
word_freq_technology        0.4321   0.3796  1.1382 0.2550  -0.3119  1.1761
word_freq_1999             -0.1758   0.2057 -0.8548 0.3927  -0.5790  0.2274
word_freq_parts            -0.6318   0.4330 -1.4593 0.1445  -1.4804  0.2168
word_freq_pm               -1.1830   0.4767 -2.4817 0.0131  -2.1173 -0.2487
word_freq_direct            0.5327   0.8143  0.6542 0.5130  -1.0633  2.1288
word_freq_cs              -28.9950  23.3190 -1.2434 0.2137 -74.6995 16.7094
```

```
word_freq_meeting           -2.9225   0.9534 -3.0652 0.0022  -4.7911 -1.0538
word_freq_original          -1.2195   0.8523 -1.4308 0.1525  -2.8899  0.4510
word_freq_project           -1.9341   0.6802 -2.8434 0.0045  -3.2672 -0.6009
word_freq_re                -1.0412   0.1761 -5.9119 0.0000  -1.3863 -0.6960
word_freq_edu               -2.0313   0.3642 -5.5773 0.0000  -2.7451 -1.3174
word_freq_table             -1.6019   2.2375 -0.7159 0.4740  -5.9872  2.7835
word_freq_conference        -4.2555   1.6791 -2.5344 0.0113  -7.5466 -0.9645
char_freq_;                 -1.6625   0.6586 -2.5245 0.0116  -2.9533 -0.3718
char_freq_(                 -0.8993   0.3646 -2.4666 0.0136  -1.6139 -0.1847
char_freq_[                 -2.1438   1.5067 -1.4229 0.1548  -5.0968  0.8092
char_freq_!                  0.2096   0.0613  3.4223 0.0006   0.0896  0.3297
char_freq_$                  4.1633   0.7593  5.4834 0.0000   2.6752  5.6514
char_freq_#                  2.1391   1.3616  1.5709 0.1162  -0.5297  4.8078
capital_run_length_average  -0.0165   0.0064 -2.5883 0.0096  -0.0289 -0.0040
capital_run_length_longest   0.0093   0.0023  3.9979 0.0001   0.0047  0.0138
capital_run_length_total     0.0002   0.0002  1.2034 0.2288  -0.0001  0.0006
============================================================================
```

word_freq_remove, word_freq_free, word_freq_hp, word_freq_george, word_freq_re, word_freq_edu, char*freq*$, capital_run_length_longest, char*freq*!, word_freq_project, word_freq_meeting, word_freq_data, word_freq_000, word_freq_will, word_freq_our, word_freq_address are the most significant predictors. It's surprising to see that the frequency of hp and george being significant predictors. I don’t see why they’re predictive in the detection of spams.

In [499]:
```
pred_ave=test_new['capital_run_length_average']
pred_ave=pd.DataFrame(pred_ave)
pred_ave.columns=['capital_run_length_average']
```

In [501]:
```
assn2_sh2432_results = pd.concat([pred_ave, knn_pred1, knn_pred2, logm_pred1,
logm_pred2], axis=1)
assn2_sh2432_results.to_csv(r'C:/Users/kavan/Documents/My Study/SDS 555 Intro
 to Machine Learning/HW/HW 2/assn2_sh2432_results.csv')
assn2_sh2432_results.head(n=20)
```

Out[501]:

	capital_run_length_average	knn_pred1	knn_pred2	logm_pred1	logm_pred2
0	1.729000	0	1	1	1
1	1.312000	1	0	1	1
2	5.659000	0	0	1	1
3	1.320000	0	1	1	1
4	4.857000	1	1	1	1
5	4.000000	1	1	1	1
6	3.100000	1	1	1	1
7	4.000000	1	1	1	1
8	4.264733	1	1	1	1
9	2.145000	1	1	1	1
10	1.468000	1	1	1	1
11	5.891000	1	1	1	1
12	1.915200	0	1	1	1
13	2.368067	1	1	1	1
14	2.777000	1	1	1	1
15	1.411000	1	0	1	1
16	4.850000	1	0	1	1
17	72.500000	1	1	1	1
18	3.400000	1	1	1	1
19	2.829933	1	1	1	1

Problem 2: Gradient Descent

Consider the scenario of univariate logistic regression where we are trying to predict Y, which can take the value 0 or 1, from the variable X, which can take the value of any real number. Recall from lecture that we need to predict parameters β_0 and β_1 by minimizing the penalized loss function:

$$L(\beta_0, \beta_1) = \sum_{i=1}^{n} \left[log\left(1 + e^{\beta_0 + X_i\beta_1}\right) - Y_i\left(\beta_0 + X_i\beta_1\right)\right] + \lambda\left(\beta_0^2 + \beta_1^2\right) .$$

Run the next cell to simulate data from the true values of $\beta_0 = 2.5$ and $\beta_1 = 3.0$.

In [507]:
```
n = 10000
x1 = np.random.uniform(-5, 5, size=n)
beta0 = 2.5
beta1 = -3.0
p = np.exp(beta0 + x1*beta1)/(1 + np.exp(beta0 + beta1*x1))
y = np.random.binomial(1, p, size=n)
```

Part a

For given values of β_0 and β_1 the vector $\left(\frac{\partial}{\partial \beta_0} L(\beta_0, \beta_1), \frac{\partial}{\partial \beta_1} L(\beta_0, \beta_1)\right)^T$ is called the gradient of $L(\beta_0, \beta_1)$ and is denoted $\nabla L(\beta_0, \beta_1)$.

Calculate the derivative of $L(\beta_0, \beta_1)$ with respect to β_0, treating β_1 as a constant. (i.e. calculate $\frac{\partial}{\partial \beta_0} L(\beta_0, \beta_1)$).

Now calculate the derivative of $L(\beta_0, \beta_1)$ with respect to β_1, treating β_0 as a constant. (i.e. calculate $\frac{\partial}{\partial \beta_1} L(\beta_0, \beta_1)$).

Be sure to show your work by either typing it in here using LaTeX, or by taking a picture of your handwritten solutions and displaying them here in the notebook. (If you choose the latter of these two options, be sure that the display is large enough and legible. You may find the example shown in the *Introduction to Python.ipynb* notebook for the Yale image useful.)

In [182]:
```
import PIL
from PIL import Image
baseheight = 560
img = Image.open("HW2_2a.jpg")
hpercent = (baseheight / float(img.size[1]))
wsize = int((float(img.size[0]) * float(hpercent)))
img = img.resize((wsize, baseheight), PIL.Image.ANTIALIAS)
img
```

Out[182]:

$$L(\beta_0,\beta_1) = \sum_{i=1}^{n}\left[\log(1+e^{\beta_0+x_i\beta_1}) - y_i(\beta_0 + x_i\beta_1)\right] + \lambda(\beta_0^2+\beta_1^2)$$

① $$\frac{\partial}{\partial \beta_0} L(\beta_0,\beta_1) = \sum_{i=1}^{n}\left[\frac{e^{\beta_0+\beta_1 x_i}}{1+e^{\beta_0+\beta_1 x_i}} - y_i\right] + 2\lambda\beta_0$$

② $$\frac{\partial}{\partial \beta_1} L(\beta_0,\beta_1) = \sum_{i=1}^{n}\left[\frac{e^{\beta_0+\beta_1 x_i}}{1+e^{\beta_0+\beta_1 x_i}} - x_i y_i\right] + 2\beta_1\lambda$$

Part b

Complete the function in the following cell called *update()* which takes values for β_0 and β_1 as well as a step-size η and should return updated values for β_0 and β_1 from one step of gradient descent (using all the data and your answer to Part a). You may use the value 0.01 for λ.

In [510]:
```
def update(b0, b1, eta):
    a = np.exp(b0 + x1*b1)/(1 + np.exp(b0 + b1*x1))
    b0_new = b0 - (np.sum(a)-np.sum(y)+0.02*b0)*eta
    b1_new = b1 - (np.sum((a-y)*x1)+0.02*b1)*eta
    return(b0_new, b1_new)
```

In [511]:
```
update(0, 0, 0.01)
```

Out[511]:
```
(8.6, -119.00733387731293)
```

Now complete the function in the next cell called *loss()* which takes values for β_0 and β_1 and should return the value of the loss function evaluated at those two parameter values.

In [512]:
```
def loss(b0, b1):
    lo=  np.sum( np.log(1+np.exp(b0+b1*x1))-y*(b0+b1*x1) ) +0.01*(b0**2+b1**2)
    return(lo)
```

In [513]:
```
loss(0, 0)
```

Out[513]:
```
6931.471805599453
```

Part c

The following cell uses the two functions from Part b to implement gradient descent for this problem, keeping track of the values for β_0, β_1, and $L(\beta_0, \beta_1)$ at each iteration. In the cell below the code, answer each of the questions included as comments next to the code. Also, create individual plots of β_0, β_1, and $L(\beta_0, \beta_1)$ vs. iteration number. Do these three quantities behave as expected for gradient descent?

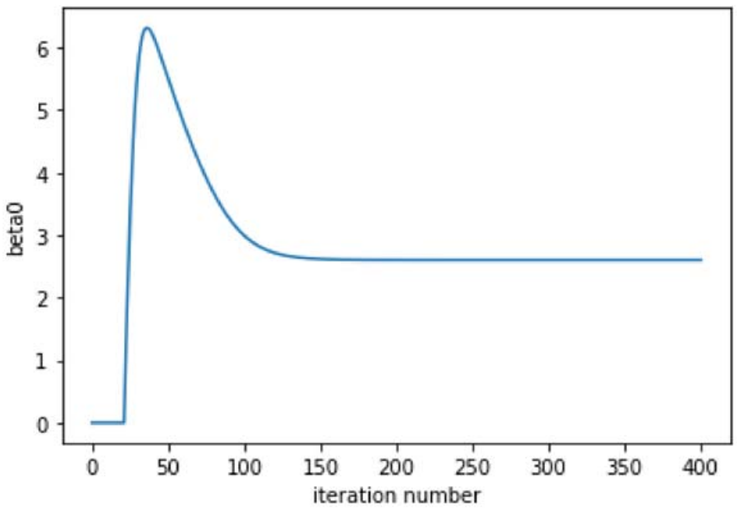
In [473]:
```
step = 0.01
beta0_hat = 0
beta1_hat = 0
l = loss(beta0_hat, beta1_hat)
beta0_all = [beta0_hat]
beta1_all = [beta1_hat]
loss_all = [l]
i=0
while i < 400 and step > 3e-8:                #1. What is the reasoning behind t
hese two stopping criteria?
    b = update(beta0_hat, beta1_hat, step)  #2. What is being calculated here?
    l_new = loss(b[0], b[1])                #3. What is being calculated here?
    if l_new < l:                           #4. What happens if the statement
 being tested here is True?
        beta0_hat = b[0]
        beta1_hat = b[1]
        l = l_new
    else:
        step = step*0.9                     #5. What happens if the statement
 tested above is False? What is the reasoning
                                            #   behind this?
    i = i+1
    beta0_all.append(beta0_hat)
    beta1_all.append(beta1_hat)
    loss_all.append(l)
```

In [517]:
```
# find the position of min loss value
loss_all.index(min(loss_all))
```

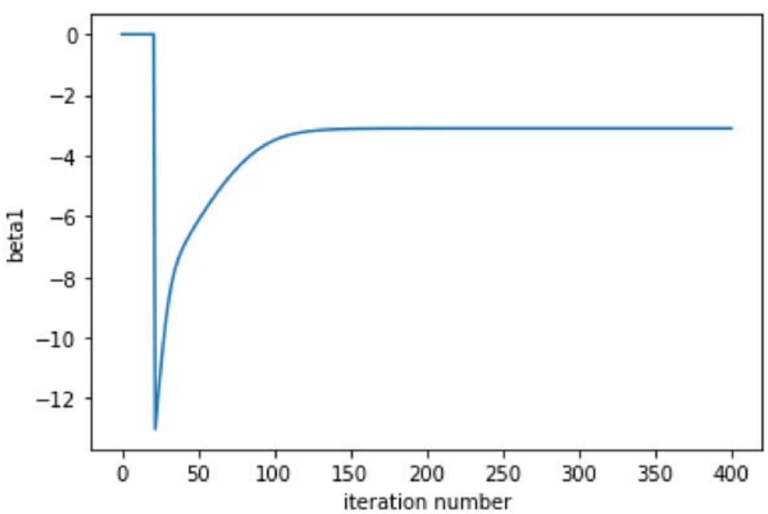
Out[517]: 322

1. Use i<400 and step > 3e-8 to limit the maximum time of iteration. After each iteration, step will decrease. The boundary indicated here gives the required precision of the iteration.
2. It calculates the updated values of β_0 and β_1 from each step of gradient descent
3. It takes values of β_0 and β_1 from each step of gradient descent and return the value of the loss function evaluated at the two parameter values.
4. If the statement here is True, which means we find a smaller value of the loss funcation, we want to keep the values of β_0 and β_1 which gives the smaller loss. Eventually, we will find the predict parameters that minimize the penalized loss function (converges to the desired local minimum).
5. If the statement here is False, we reduce the step by 0.1 and re-calculate the predict parameters and loss function. All local minima are also global minima, so in this case gradient descent can converge to the global solution.

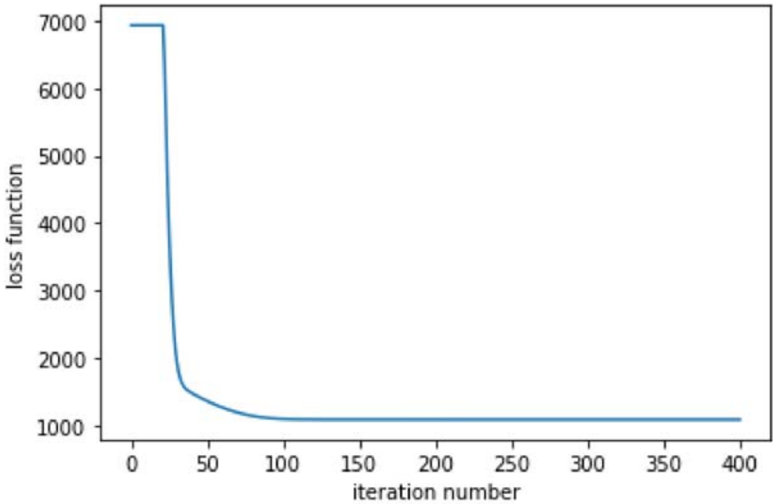
In [518]:
```
ii=range(0,i+1)
plt.plot(ii, beta0_all)
plt.xlabel('iteration number')
plt.ylabel('beta0');
```



In [519]:
```
plt.plot(ii, beta1_all);
plt.xlabel('iteration number')
plt.ylabel('beta1');
```



In [520]:
```
plt.plot(ii, loss_all)
plt.xlabel('iteration number')
plt.ylabel('loss function');
```



Yes the three quantities behave as expected for gradient descent. The minimized loss value was found at the 322rd time of iteration.

Part d

Is your gradient descent algorithm from Part b robust against initial estimates of β_0 and β_1? To help answer this question, take the code above that implements gradient descent and put it in a function that takes initial estimates of β_0 and β_1 as arguments, and returns the optimized values from gradient descent. Run this function using each of the following pairs of (β_0, β_1) as initial estimates: $(15, 3)$, $(-30, 5)$, and $(-8, -8)$. Are your final estimates approximately the same each time?

In [540]:
```
def gradient(b0, b1):
    step = 0.01
    beta0_hat=b0
    beta1_hat=b1
    l = loss(beta0_hat, beta1_hat)
    beta0_all = [beta0_hat]
    beta1_all = [beta1_hat]
    loss_all = [l]
    i=0
    while i < 400 and step > 3e-8:
        b = update(beta0_hat, beta1_hat, step)
        l_new = loss(b[0], b[1])
        if l_new < l:
            beta0_hat = b[0]
            beta1_hat = b[1]
            l = l_new
        else:
            step = step*0.9
        i = i+1
        beta0_all.append(beta0_hat)
        beta1_all.append(beta1_hat)
        loss_all.append(l)
    ol=loss_all.index(min(loss_all))
    bb0=beta0_all[ol]
    bb1=beta1_all[ol]
    return(bb0, bb1)
```

In [541]:
```
gradient(15,3)
```

Out[541]: (2.5998024017984296, -3.1018293098156335)

In [542]:
```
gradient(-30,5)
```

Out[542]: (2.5998023992263497, -3.101829312415696)

In [543]:
```
gradient(-8,-8)
```

Out[543]: (2.5998023973771938, -3.1018293142843456)

Yes the final estimates of the parameters $(\hat{\beta}_0, \hat{\beta}_1)$ are approximately the same each time.

Problem 3: Cross-Validation

Part a

Generate a simulated data set with the following cell:

In [498]:
```
np.random.seed(1)
x = np.random.normal(size=100)
y = x - 2*x**2 + np.random.normal(size=100)
```

In this data set, what is the value of n (the number of data points) and what is the value of p (the true number of model parameters)? Write out the model used to generate the data in equation form.

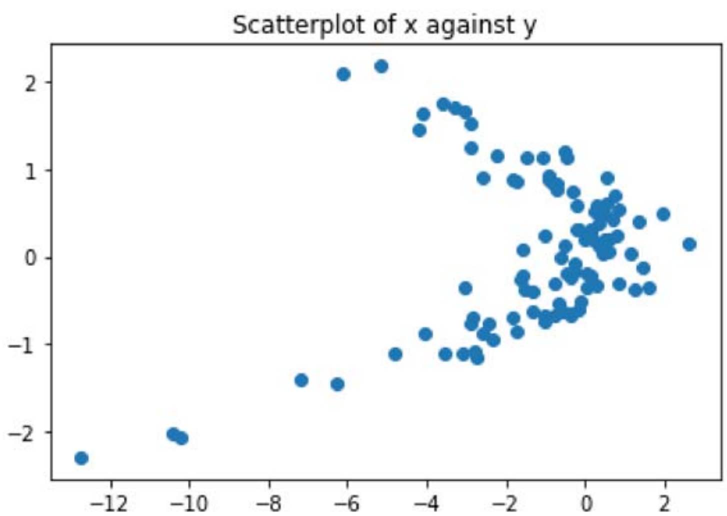
The number of data points is 100 (n=100). The number of model parameters is 2 (p=2).

$$Y = X - 2X^2 + \epsilon$$

Part b

Create a scatterplot of X against Y. Comment on what you find.

In [131]:
```
plt.scatter(y,x);
plt.title("Scatterplot of x against y");
```



The scatterplot is nearly symmetric to x=0. It has a V shape with the peak around x=0 and tails around x=2 & x=-2. There are more points around the peak than in other areas.

Part c

Set a random seed, and then compute the Leave-One-Out Cross-Validation (LOOCV) errors that result from fitting the following four models using least squares:

i. $Y = \beta_0 + \beta_1 X + \epsilon$

ii. $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \epsilon$

iii. $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 + \epsilon$

iv. $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 + \beta_4 X^4 + \epsilon$

Note: for linear regression, the LOOCV error can be computed via the following short-cut formula:

$$\text{LOOCV Error} = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{Y_i - \widehat{Y_i}}{1 - H_{ii}} \right)^2$$

where H_{ii} is the i^{th} diagonal entry of the projection matrix $H = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$, and $\mathbf{X}$ is a matrix of predictors (the design matrix). This formula is an alternative to actually carrying out the $n = 100$ regressions you would otherwise need for LOOCV. An example of how to calculate the projection matrix H is provided below for the case of $n = 5$ and the model $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \epsilon$. To get the diagonal elements of H you may find the function `np.diag()` useful.

In [248]:
```
example_x = np.array([-3, -4, -5, -6, -7])        #generate the x variable
design_x = np.vander(example_x, 3)          #calculate the design matrix for t
he polynomial model with 3 fit parameters
H = np.dot(design_x, np.dot(np.linalg.inv(np.dot(design_x.T, design_x)), desig
n_x.T))  #calculate H
```

In [325]:
```
def loocve(seed):
    np.random.seed(seed)
    x = np.random.normal(size=100)
    y = x - 2*x**2 + np.random.normal(size=100)
    #calculate the design matrix for the polynomial model with n fit parameter
s
    x1 = pd.DataFrame(np.vander(x, 2))
    x2 = pd.DataFrame(np.vander(x, 3))
    x3 = pd.DataFrame(np.vander(x, 4))
    x4 = pd.DataFrame(np.vander(x, 5))
    #calculate each H
    h1 = np.dot(x1, np.dot(np.linalg.inv(np.dot(x1.T, x1)), x1.T))
    h1=pd.DataFrame(h1)
    h2 = np.dot(x2, np.dot(np.linalg.inv(np.dot(x2.T, x2)), x2.T))
    h2=pd.DataFrame(h2)
    h3 = np.dot(x3, np.dot(np.linalg.inv(np.dot(x3.T, x3)), x3.T))
    h3=pd.DataFrame(h3)
    h4 = np.dot(x4, np.dot(np.linalg.inv(np.dot(x4.T, x4)), x4.T))
    h4=pd.DataFrame(h4)
    simu = pd.DataFrame({'x':x, 'y':y})
    simu['x2']=x**2
    simu['x3']=x**3
    simu['x4']=x**4
    #linear regression
    import statsmodels.api as sm
    import statsmodels.formula.api as smf
    r1 = smf.ols('y ~ x', data=simu).fit()
    p1=r1.predict()
    r2 = smf.ols('y ~ x+x2', data=simu).fit()
    p2=r2.predict()
    r3 = smf.ols('y ~ x+x2+x3', data=simu).fit()
    p3=r3.predict()
    r4 = smf.ols('y ~ x+x2+x3+x4', data=simu).fit()
    p4=r4.predict()
    #calculate LOOCV Error
    loocve1 = np.sum(((simu['y']-p1)/(1-np.diag(h1)))**2)/100
    loocve2 = np.sum(((simu['y']-p2)/(1-np.diag(h2)))**2)/100
    loocve3 = np.sum(((simu['y']-p3)/(1-np.diag(h3)))**2)/100
    loocve4 = np.sum(((simu['y']-p4)/(1-np.diag(h4)))**2)/100
    loocve=pd.DataFrame({'LOOCV_Err1':[loocve1], 'LOOCV_Err2':[loocve2], 'LOOC
V_Err3':[loocve3], 'LOOCV_Err4':[loocve4]})
    return(loocve)
```

In [326]:
```
loocve(199)
```

Out[326]:

	LOOCV_Err1	LOOCV_Err2	LOOCV_Err3	LOOCV_Err4
0	9.416064	1.272031	1.286743	1.309934

Part d

Repeat Part c using another random seed to generate data, and report your results. Are your results the same as what you got in Part c? Why?

In [327]:
```
#use the function I wrote in part c with a changed seed number
loocve(10000)
```

Out[327]:

	LOOCV_Err1	LOOCV_Err2	LOOCV_Err3	LOOCV_Err4
0	9.651378	1.13644	1.129351	1.159466

No, they are not the same. Values of x and y are changed because of the change of the seed. The models fitted also changed accordingly, therefore the LOOCV Errors are not the same. After the change of the seed, the best fit model also changed, previously it was Model 2 but now it's Model 3. I think it is understandable because the error term's distribution in the original model is the same as X's. They are at the same scale so that the influence of the error term to Y in the latter test is more like another parameter rather than error.

Part e

Which of the models in Part c had the smallest LOOCV error? Is this what you expected? Explain your answer.

In Part c, my Model 2 has the smallest LOOCV Error (1.272031). It is the same as I expected because the original model used to generate X and Y is exactly like model 2 with respect to parameters. The influence of the error term to Y is not very large in this situation.

Part f

Comment on the statistical significance of the coefficient estimates that results from fitting each of the models in Part c using least squares. Do these results agree with the conclusions drawn based on the cross-validation results?

In [328]:
```
#recall the results from part c
loocve(199)
```

Out[328]:

	LOOCV_Err1	LOOCV_Err2	LOOCV_Err3	LOOCV_Err4
0	9.416064	1.272031	1.286743	1.309934

In [329]:
```
#model 1 output
r1.summary()
```

Out[329]:

OLS Regression Results

Dep. Variable:	y	**R-squared:**	0.135
Model:	OLS	**Adj. R-squared:**	0.126
Method:	Least Squares	**F-statistic:**	15.30
Date:	Thu, 26 Sep 2019	**Prob (F-statistic):**	0.000170
Time:	00:33:48	**Log-Likelihood:**	-249.53
No. Observations:	100	**AIC:**	503.1
Df Residuals:	98	**BIC:**	508.3
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
Intercept	-1.9833	0.297	-6.684	0.000	-2.572	-1.395
x	1.1624	0.297	3.911	0.000	0.573	1.752

Omnibus:	36.829	**Durbin-Watson:**	1.941
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	66.612
Skew:	-1.559	**Prob(JB):**	3.43e-15
Kurtosis:	5.503	**Cond. No.**	1.05

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

In [330]:
```
#model 2 output
r2.summary()
```

Out[330]:

OLS Regression Results

Dep. Variable:	y	**R-squared:**	0.879
Model:	OLS	**Adj. R-squared:**	0.877
Method:	Least Squares	**F-statistic:**	352.7
Date:	Thu, 26 Sep 2019	**Prob (F-statistic):**	3.14e-45
Time:	00:34:59	**Log-Likelihood:**	-151.14
No. Observations:	100	**AIC:**	308.3
Df Residuals:	97	**BIC:**	316.1
Df Model:	2		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
Intercept	0.0278	0.139	0.201	0.841	-0.247	0.303
x	0.9692	0.112	8.656	0.000	0.747	1.191
x2	-2.0088	0.082	-24.434	0.000	-2.172	-1.846

Omnibus:	3.641	**Durbin-Watson:**	1.874
Prob(Omnibus):	0.162	**Jarque-Bera (JB):**	3.527
Skew:	-0.407	**Prob(JB):**	0.171
Kurtosis:	2.570	**Cond. No.**	2.43

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

In [331]:
```
#model 3 output
r3.summary()
```

Out[331]:

OLS Regression Results

Dep. Variable:	y	**R-squared:**	0.879
Model:	OLS	**Adj. R-squared:**	0.875
Method:	Least Squares	**F-statistic:**	232.9
Date:	Thu, 26 Sep 2019	**Prob (F-statistic):**	6.43e-44
Time:	00:35:11	**Log-Likelihood:**	-151.10
No. Observations:	100	**AIC:**	310.2
Df Residuals:	96	**BIC:**	320.6
Df Model:	3		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
Intercept	0.0298	0.139	0.214	0.831	-0.247	0.307
x	1.0187	0.211	4.818	0.000	0.599	1.438
x2	-2.0140	0.085	-23.759	0.000	-2.182	-1.846
x3	-0.0175	0.063	-0.277	0.783	-0.143	0.108

Omnibus:	3.633	**Durbin-Watson:**	1.862
Prob(Omnibus):	0.163	**Jarque-Bera (JB):**	3.525
Skew:	-0.407	**Prob(JB):**	0.172
Kurtosis:	2.573	**Cond. No.**	6.83

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

In [332]:
```
#model 4 output
r4.summary()
```

Out[332]:

OLS Regression Results

Dep. Variable:	y	**R-squared:**	0.879
Model:	OLS	**Adj. R-squared:**	0.874
Method:	Least Squares	**F-statistic:**	173.1
Date:	Thu, 26 Sep 2019	**Prob (F-statistic):**	9.97e-43
Time:	00:35:25	**Log-Likelihood:**	-151.03
No. Observations:	100	**AIC:**	312.1
Df Residuals:	95	**BIC:**	325.1
Df Model:	4		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	0.0009	0.160	0.006	0.996	-0.317	0.319
x	1.0148	0.213	4.772	0.000	0.593	1.437
x2	-1.9371	0.223	-8.672	0.000	-2.381	-1.494
x3	-0.0195	0.064	-0.306	0.760	-0.146	0.107
x4	-0.0168	0.045	-0.373	0.710	-0.106	0.073

Omnibus:	3.654	**Durbin-Watson:**	1.868
Prob(Omnibus):	0.161	**Jarque-Bera (JB):**	3.595
Skew:	-0.420	**Prob(JB):**	0.166
Kurtosis:	2.602	**Cond. No.**	17.5

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

In model 1, both the intercept and the coefficient of X are significantly different from zero (p<0.001). In model 2, all coefficients except the intercept are significantly different from zero (p<0.001). In modele 3 & 4, coefficients of X and X^2 are significantly different from zero (p<0.001), while not for the intercept and X^3 or X^4. These results agree with the conclusions from the cross-validation process.