CS224 - Winter 2023 - PROGRAMMING ASSIGNMENT 2 - LINEAR AND LOGISTIC REGRESSION

Due: March 3, 2023 @ 11:59pm PDT

Maximum points: 20

Enter your information below:

(full) Name: Shahriar M Sakib
Student ID Number: 862393922

By submitting this notebook, I assert that the work below is my own work, completed for this course. Except where explicitly cited, none of the portions of this notebook are duplicated from anyone else's work or my own previous work.

Academic Integrity

Each assignment should be done individually. You may discuss general approaches with other students in the class, and ask questions to the TA, but you must only submit work that is yours . If you receive help by any external sources (other than the TA and the instructor), you must properly credit those sources. The UCR Academic Integrity policies are available at http://conduct.ucr.edu/policies/academicintegrity.html.

Overview

In this assignment you will implement and test two supervised learning algorithms: linear regression (Question 1) and logistic regression (Question 2).

For this assignment we will use the functionality of Pandas, Matplotlib, and Numpy.

If you are asked to **implement** a particular functionality, you should **not** use an existing implementation from the libraries above (or some other library that you may find). When in doubt, please ask.

Before you start, make sure you have installed all those packages in your local Jupyter instance.

Read **all** cells carefully and answer **all** parts (both text and missing code). You will complete all the code marked `TODO` and answer descriptive/derivation questions.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
# import random as rand
from sklearn.model_selection import train_test_split
```

Question 1: Linear Regression [12 points]

We will implement linear regression using direct solution and gradient descent.

We will first attempt to predict output using a single attribute/feature. Then we will perform linear regression using multiple attributes/features.

Getting data [1 point]

In this assignment we will use the Boston housing dataset.

The Boston housing data set was collected in the 1970s to study the relationship between house price and various factors such as the house size, crime rate, socio-economic status, etc. Since the variables are easy to understand, the data set is ideal for learning basic concepts in machine learning. The raw data and a complete description of the dataset can be found on the UCI website: https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names, https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data

or

http://www.ccs.neu.edu/home/vip/teach/MLcourse/data/housing_desc.txt

I have supplied a list `names` of the column headers. You will have to set the options in the `read_csv` command to correctly delimit the data in the file and name the columns correctly.

In [2]:
```
names =[
    'CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE',  'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'PRICE']

df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data',
                 header=None,delim_whitespace=True,names=names,na_values='?')
print(df.head())
```

```
      CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX  \
0  0.00632  18.0   2.31     0  0.538  6.575  65.2  4.0900    1  296.0
1  0.02731   0.0   7.07     0  0.469  6.421  78.9  4.9671    2  242.0
2  0.02729   0.0   7.07     0  0.469  7.185  61.1  4.9671    2  242.0
3  0.03237   0.0   2.18     0  0.458  6.998  45.8  6.0622    3  222.0
4  0.06905   0.0   2.18     0  0.458  7.147  54.2  6.0622    3  222.0

   PTRATIO       B  LSTAT  PRICE
0     15.3  396.90   4.98   24.0
1     17.8  396.90   9.14   21.6
2     17.8  392.83   4.03   34.7
3     18.7  394.63   2.94   33.4
4     18.7  396.90   5.33   36.2
```

Create a response vector `y` with the values in the column `PRICE`. The vector `y` should be a 1D `numpy.array` structure.

In [3]:
```
# TODO
y = np.array(df["PRICE"])
print(y[:10])
```

```
[24.  21.6 34.7 33.4 36.2 28.7 22.9 27.1 16.5 18.9]
```

Use the response vector `y` to find the mean house price in thousands and the fraction of homes that are above \$40k. (You may realize this is very cheap. Prices have gone up a lot since the 1970s!). Create print statements of the form (replace `a`'s and `b`'s with the number you get):

```
The mean house price is aa.bb thousands of dollars.
Only a.b percent are above $40k.
```

In [4]:
```
# TODO
#claculating the mean
mean_price = "{:.2f}".format(np.mean(y))
```

```
print("The mean house price is:", mean_price, "thousands of dollars.")
#claculating the  fraction of homes that are above $40k
above_40k = len(y[y > 40.00])
total_count = len(y)
percentage = (above_40k / total_count) * 100
Percantage_above_40k="{:.2f}%".format(percentage)
print("Number of houses with price above 40k:", above_40k)
print("Only",Percantage_above_40k,"percent are above $40k"  )
```

```
The mean house price is: 22.53 thousands of dollars.
Number of houses with price above 40k: 31
Only 6.13% percent are above $40k
```

Visualizing the Data [1 point]

Python's `matplotlib` has very good routines for plotting and visualizing data that closely follows the format of MATLAB programs. You can load the `matplotlib` package with the following commands.

In [5]:
```
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
```

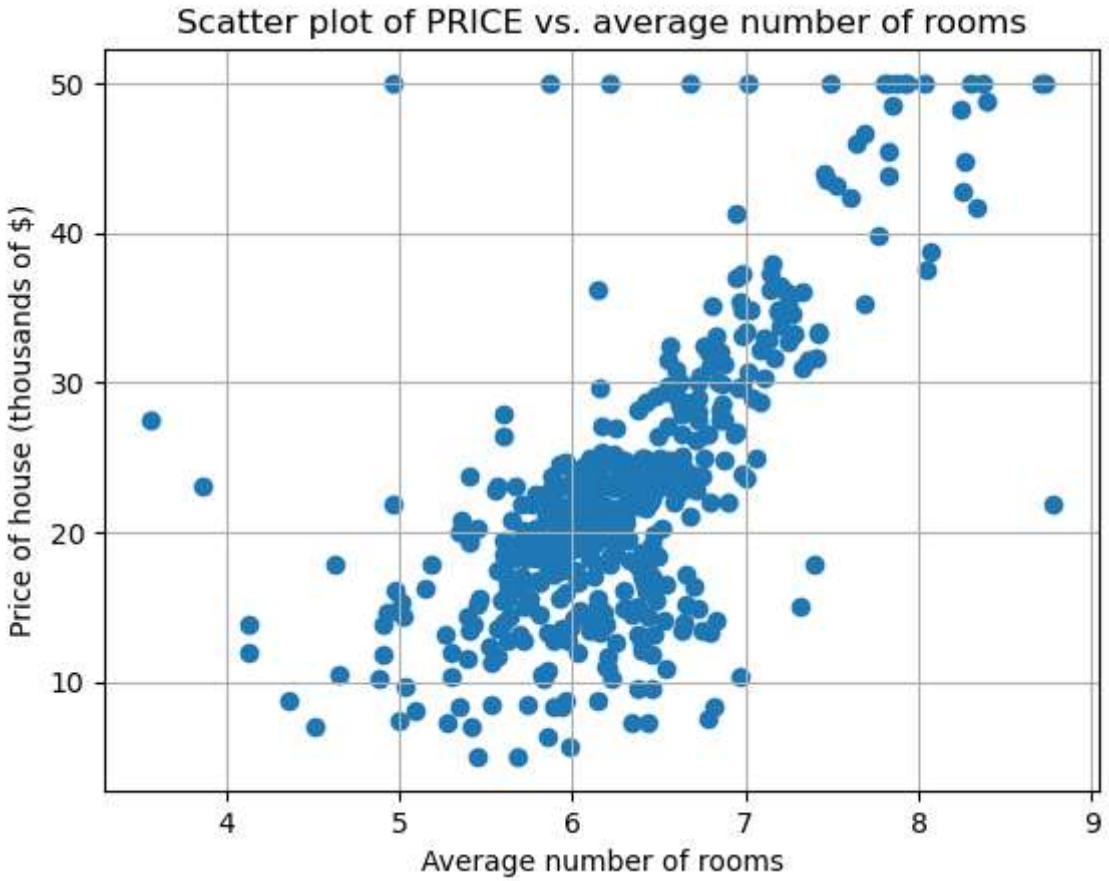
Similar to the `y` vector, create a predictor vector `x` containing the values in the `RM` column, which represents the average number of rooms in each region.

In [6]:
```
# TODO
x = np.array(df["RM"])
print(x[:10])
```

```
[6.575 6.421 7.185 6.998 7.147 6.43  6.012 6.172 5.631 6.004]
```

Create a scatter plot of the `PRICE` vs. the `RM` attribute. Make sure your plot has **grid lines** and label the axes with reasonable **labels** so that someone else can understand the plot.

In [7]:
```
# TODO
plt.scatter(x, y)
plt.grid(True)
plt.xlabel("Average number of rooms")
plt.ylabel("Price of house (thousands of $)")
plt.title("Scatter plot of PRICE vs. average number of rooms")
plt.show()
```



The number of rooms and price seem to have a linear trend, so let us try to predict price using number of rooms first.

Derivation of a simple linear model for a single feature

Suppose we have N pairs of training samples $(x_1, y_1), \ldots, (x_N, y_N)$, where $x_i \in \mathbb{R}$ and $y_i \in \mathbb{R}$.

We want to perform a linear fit for this 1D data as

$$y = wx + b,$$

where $w \in \mathbb{R}$ and $b \in \mathbb{R}$.

The optimal values of w^*, b^* that minimize the loss function

$$L(w,b) = \sum_{i=1}^{N} (wx_i + b - y_i)^2$$

can be written as

$$w^* = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2}$$

and

$$b^* = \bar{y} - w^* \bar{x},$$

where $\bar{x} = \frac{1}{N}\sum_i x_i, \bar{y} = \frac{1}{N}\sum_i y_i$ are mean values of x_i, y_i, respectively.

Fitting a linear model using a single feature [3 points]

Use the formulae above to compute the parameters w, b in the linear model $y = wx + b$.

In [8]:
```
def fit_linear(x, y):
    x_mean = np.mean(x)
    y_mean = np.mean(y)
    w = np.sum((x - x_mean) * (y - y_mean)) / np.sum((x - x_mean) ** 2)
    b = y_mean - w * x_mean
    return w, b
```

Using the function `fit_linear` above, print the values `w`, `b` for the linear model of price vs. number of rooms.

In [9]:
```
# TODO
w, b = fit_linear(x,y)
print('w = {0:5.1f}, b = {1:5.1f}'.format(w,b))
```

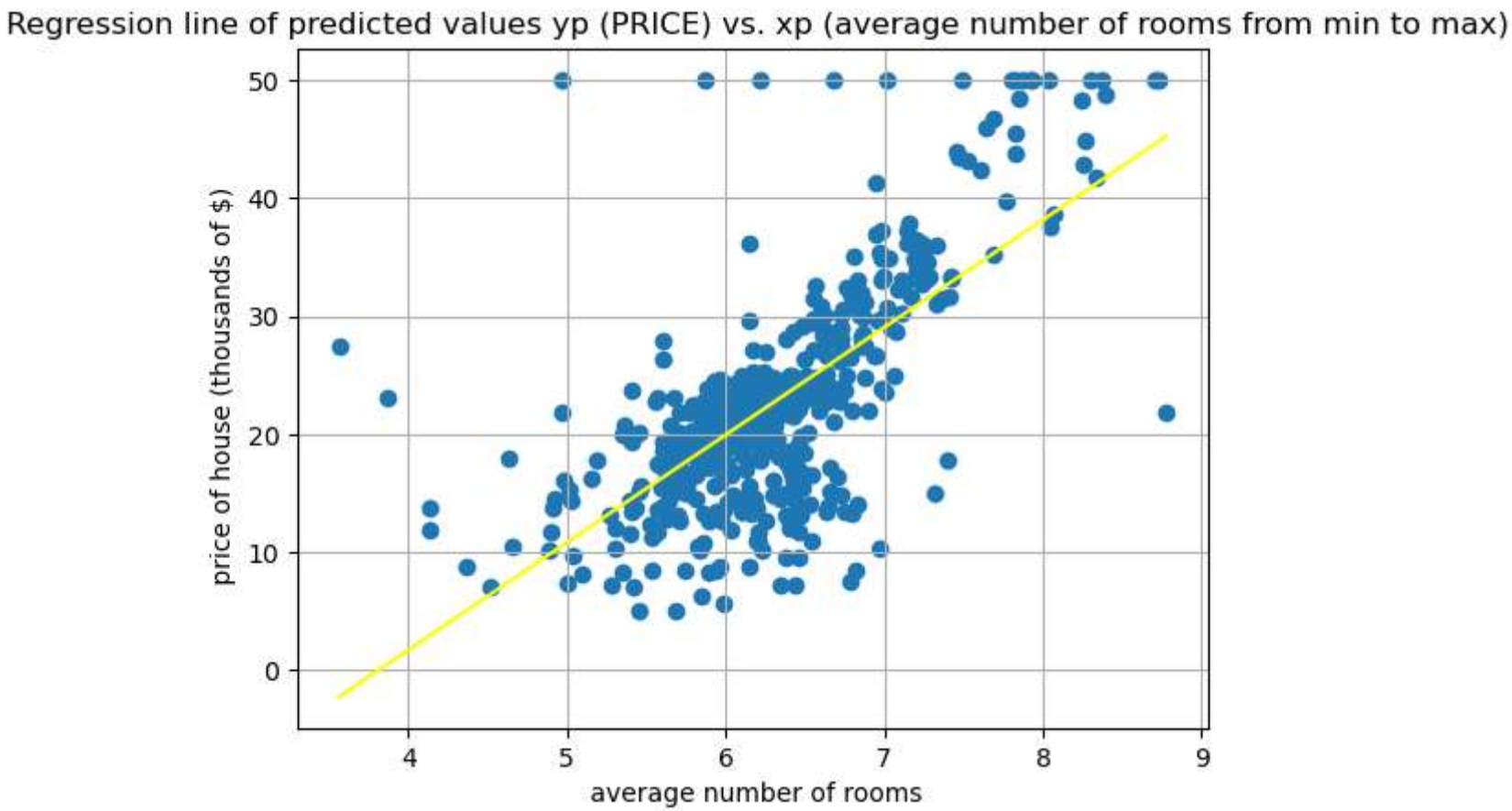
```
w =   9.1, b = -34.7
```

Does the price increase or decrease with the number of rooms?

- The equation of the line is $y=wx+b$, where w is the slope. we can see that w = 9.1 which is positive, so we can tell that on average the price increase with the number of rooms

Replot the scatter plot above, but now with the regression line. You can create the regression line by creating points `xp` from say min(x) to max(x), computing the linear predicted values `yp` on those points and plotting `yp` vs. `xp` on top of the above plot.

In [10]:
```
# TODO
# Points on the regression line
# Compute the regression line
w, b = fit_linear(x, y)

# Create a set of xp values from minimum to maximum
xp = np.linspace(np.min(x), np.max(x), 100)
# Compute the corresponding yp values using the linear model
yp = w * xp + b
# Create the scatter plot
plt.scatter(x, y)
plt.grid(True)
plt.xlabel(" average number of rooms")
plt.ylabel(" price of house (thousands of $)")
plt.title(" Regression line of predicted values yp (PRICE) vs. xp (average number of rooms from min to max)")

# Add the regression line
plt.plot(xp, yp, color="yellow")

plt.show()
```



Linear regression with multiple features/attributes [3 points]

One possible way to try to improve the fit is to use multiple variables at the same time.

In this problem, the target variable will still be the `PRICE`. We will use multiple attributes of the house to predict the price.

The names of all the data attributes are given in variable `names`.

- We can get the list of names of the columns from `df.columns.tolist()`.
- Remove the last items from the list using indexing.

In [11]:
```
xnames = names[:-1]
print(names[:-1])
```

```
['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT']
```

Let us use `CRIM`, `RM`, and `LSTAT` to predict `PRICE`.

Get the data matrix `X` with three features (`CRIM`, `RM`, `LSTAT`) and target vector `y` from the dataframe `df`.

Recall that to get the items from a dataframe, you can use syntax such as

```
s = np.array(df['RM'])
```

which gets the data in the column `RM` and puts it into an array `s`. You can also get multiple columns with syntax like

```
X12 = np.array(df[['CRIM', 'ZN']])
```

In [12]:
```
# TODO
# X = ...
X = np.array(df[['CRIM', 'ZN', 'LSTAT']])
```

Linear regression in scikit-learn

To fit the linear model, we could create a regression object and then fit the training data with regression object.

```
from sklearn import linear_model
regr = linear_model.LinearRegression()
regr.fit(X_train,y_train)
```

You can see the coefficients as

```
regr.intercept_
regr.coef_
```

We can predict output for any data as

```
y_pred = regr.predict(X)
```

Instead of taking this approach, we will implement the regression function directly.

Split the Data into Training and Test

Split the data into training and test. Use 30% for test and 70% for training. You can do the splitting manually or use the `sklearn` package `train_test_split`. Store the training data in `Xtr,ytr` and test data in `Xts,yts`.

In [33]:
```
# TODO

# Split the data into training and test sets
Xtr, Xts, ytr, yts = train_test_split(X, y, test_size=0.3, random_state=42)

# Print the shapes of the resulting arrays
print("Xtr shape:", Xtr.shape)
print("Xts shape:", Xts.shape)
print("ytr shape:", ytr.shape)
print("yts shape:", yts.shape)
```

```
Xtr shape: (478, 2)
Xts shape: (205, 2)
ytr shape: (478,)
yts shape: (205,)
```

Compute the predicted values `yhat_tr` on the training data and print the average square loss value on the **training** data.

In [34]:
```
def fit_linear_regression(X, y):
    X_T = np.transpose(X)
    X_T_X = np.dot(X_T, X)
    X_T_X_inv = np.linalg.inv(X_T_X)
    X_T_y = np.dot(X_T, y)
    w = np.dot(X_T_X_inv, X_T_y)
    b = w[0]
    return w, b
```

In [35]:
```
Xtr_new = np.hstack((np.ones((Xtr.shape[0], 1)), Xtr))
(w,b)= fit_linear_regression(Xtr_new, ytr)
# Predict values yhat_tr on the training data
yhat_tr = np.dot(Xtr_new, w)+b

# Compute the average square loss value on the training data
avg_sq_loss_tr = (np.sum((yhat_tr - ytr)**2))

# Print the average square loss value on the training data
print("Average square loss on training data = %f" % avg_sq_loss_tr)
```

```
Average square loss on training data = 35.278058
```

Create a scatter plot of the actual vs. predicted values of `y` on the **training** data.

In [16]:
```
plt.scatter(ytr, yhat_tr)
plt.grid(True)
plt.xlabel('Actual Prices')
plt.ylabel('Predicted Prices')
plt.title('Actual vs Predicted Prices on Training Data')
plt.show()
```

Compute the predicted values `yhat_ts` on the test data and print the average square loss value on the **test** data.

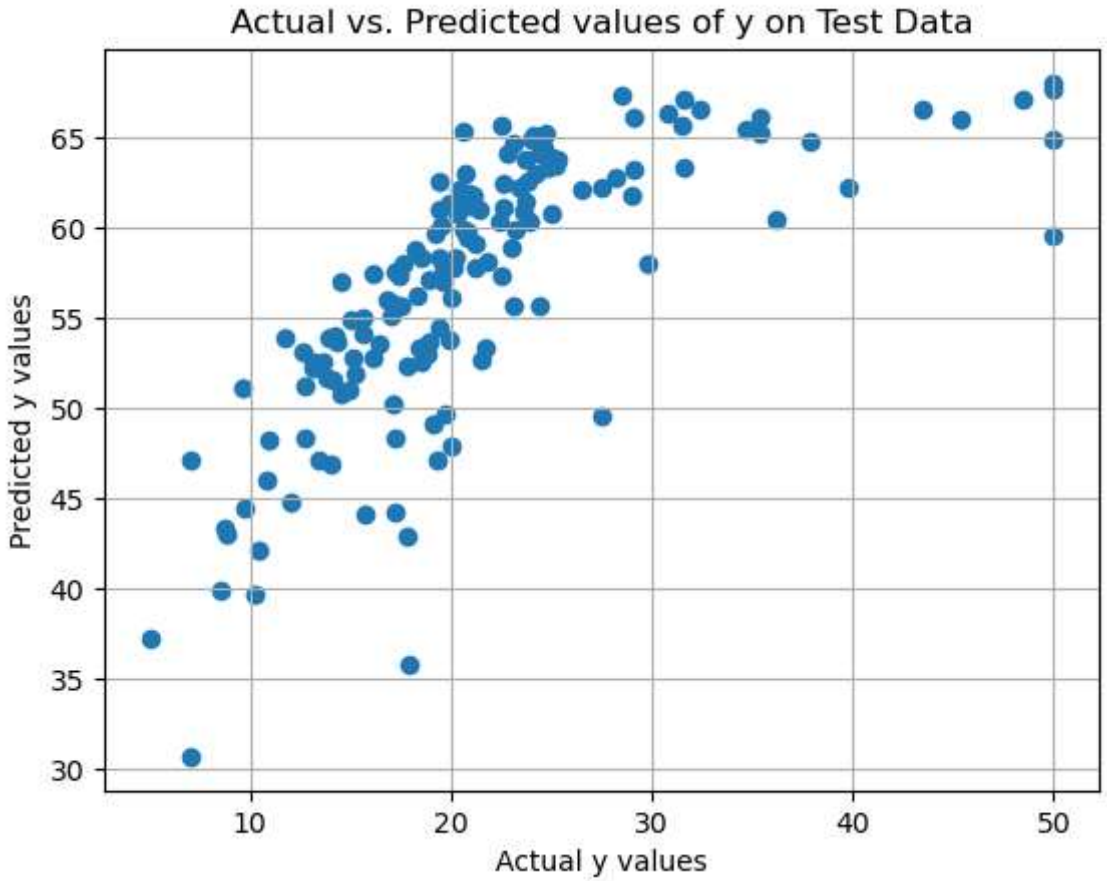
In [32]:
```
# TODO
# Add intercept column to Xts
Xts2 = np.hstack((np.ones((Xts.shape[0], 1)), Xts))

# Compute predicted values on test data
```

```
yhat_ts = np.dot(Xts2, w) + b

# Compute average square loss value on test data
avg_square_loss_ts = np.sum((yhat_ts - yts)**2)/len(yts)
print(f'Average Square Loss Value on test data is {avg_square_loss_ts:0.2f}')
```

```
Average Square Loss Value on test data is 18.02
```

Create a scatter plot of the actual vs. predicted values of `y` on the **test** data.

In [18]:
```
# TODO
# Compute predicted values on test data
yhat_ts = np.dot(Xts2, w) + b

# Create scatter plot of actual vs predicted values on test data
plt.scatter(yts, yhat_ts)
plt.grid(True)
plt.title("Actual vs. Predicted values of y on Test Data")
plt.xlabel("Actual y values")
plt.ylabel("Predicted y values")
plt.show()
```



Gradient descent for linear regression [2 points]

Finally, we will implement the gradient descent version of linear regression.

In particular, the function implemented should follow the following format:

```
def linear_regression_gd(X,y,learning_rate = 0.00001,max_iter=10000,tol=pow(10,-5)):
```

where `X` is the same data matrix used above (with ones column appended), `y` is the variable to be predicted, `learning_rate` is the learning rate used (α or ρ_t in the slides), `max_iter` defines the maximum number of iterations that gradient descent is allowed to run, and `tol` is defining the tolerance for convergence (which we'll discuss next).

The return values for the above function should be (at the least) 1) `w` which are the regression parameters, 2) `all_cost` which is an array where each position contains the value of the objective function $L(\mathbf{w})$ for a given iteration, 3) `iters` which counts how many iterations did the algorithm need in order to converge to a solution.

Gradient descent is an iterative algorithm; it keeps updating the variables until a convergence criterion is met. In our case, our convergence criterion is whichever of the following two criteria happens first:

- The maximum number of iterations is met
- The relative improvement in the cost is not greater than the tolerance we have specified. For this criterion, you may use the following snippet into your code:

  `np.absolute(all_cost[it] - all_cost[it-1])/all_cost[it-1] <= tol`

In [19]:
```
# TODO
# Implement gradient descent for linear regression

def compute_cost(X, w, y):
    # Compute the cost using mean squared error
    L = np.sum((np.dot(X, w) - y)**2)/(2*len(y))
    return L

def linear_regression_gd(X, y, learning_rate=0.00001, max_iter=10000, tol=pow(10, -5)):
    # Append a column of ones to X
    X = np.hstack((np.ones((X.shape[0], 1)), X))

    # Initialize w
    w = np.zeros(X.shape[1])

    # Initialize variables
    iters = 0
    all_cost = np.zeros(max_iter)
    converged = False

    # Gradient descent loop
    while not converged:
        iters += 1
        # Compute the gradient
        grad = np.dot(X.T, (np.dot(X, w) - y)) / len(y)
        # Update w
        w = w - learning_rate * grad
        # Compute the cost
        all_cost[iters-1] = compute_cost(X, w, y)
```

```
        # Check if converged
        if iters >= max_iter or (iters > 1 and np.absolute(all_cost[iters-1] - all_cost[iters-2])/all_cost[iters-2] <=
            converged = True

    return w, all_cost[:iters], iters
```

Convergence plots [2 points]

After implementing gradient descent for linear regression, we would like to test that indeed our algorithm converges to a solution. In order see this, we are going to look at the value of the objective/loss function $L(\mathbf{w})$ as a function of the number of iterations, and ideally, what we would like to see is $L(\mathbf{w})$ drops as we run more iterations, and eventually it stabilizes.

The learning rate plays a big role in how fast our algorithm converges: a larger learning rate means that the algorithm is making faster strides to the solution, whereas a smaller learning rate implies slower steps. In this question we are going to test two different values for the learning rate:

- 0.00001
- 0.000001

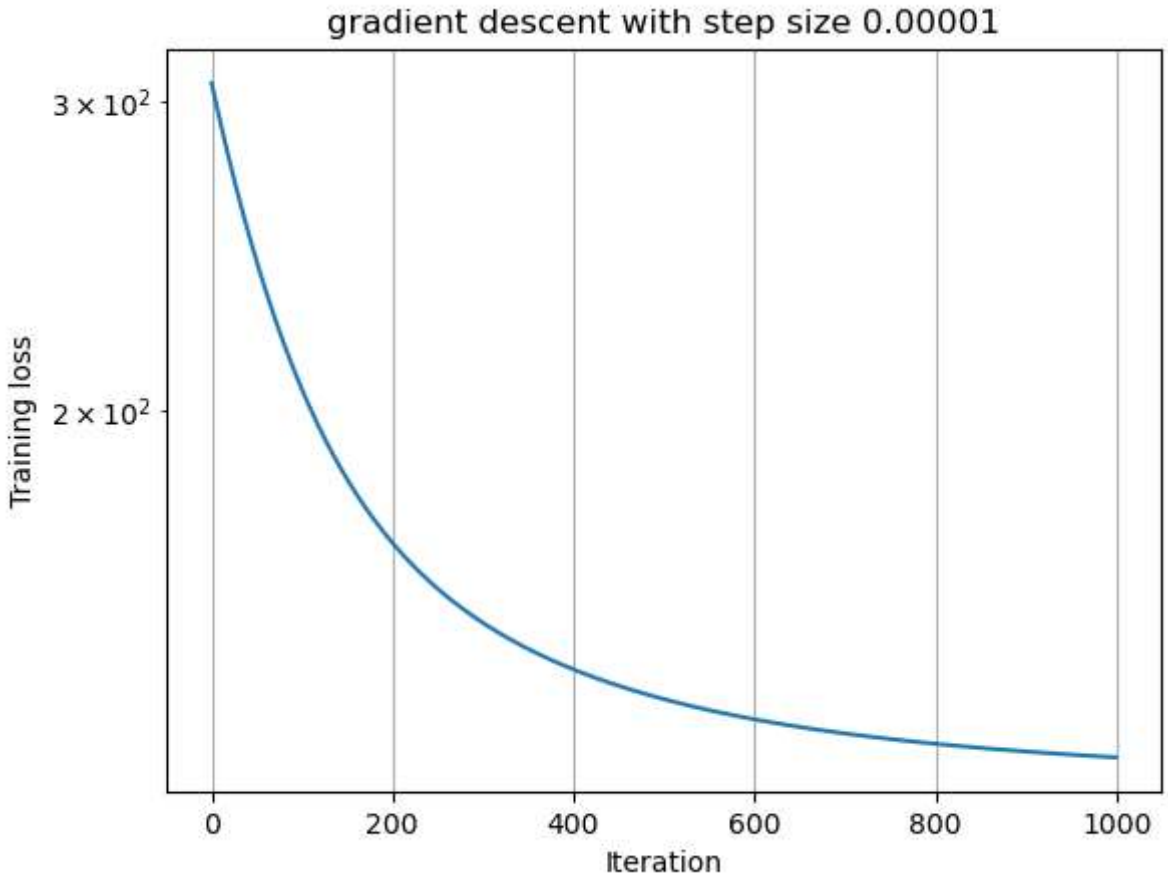
while keeping the default values for the max number of iterations and the tolerance.

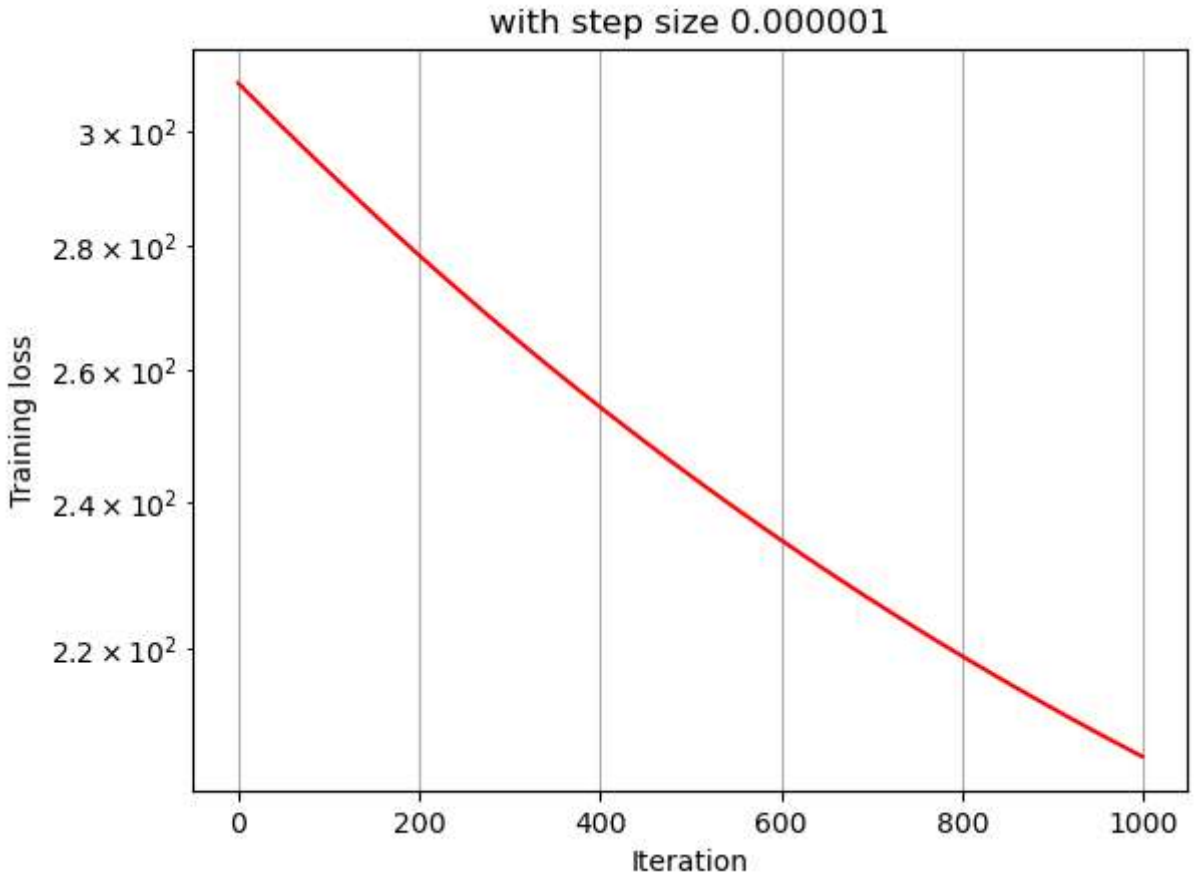
- Plot the two convergence plots (cost(loss) vs. iterations)
- What do you observe?

In [20]:
```
# TODO
# test gradient descent with step size 0.00001
(w, all_cost,iters) = linear_regression_gd(Xtr,ytr,learning_rate = 0.00001,max_iter = 1000, tol=pow(10,-6))
plt.figure(0)
plt.semilogy(all_cost[0:iters])
plt.grid(True)
plt.title("gradient descent with step size 0.00001")
plt.xlabel('Iteration')
plt.ylabel('Training loss')

# test gradient descent with step size 0.000001
(w, all_cost,iters) = linear_regression_gd(Xtr,ytr,learning_rate = 0.000001,max_iter = 1000, tol=pow(10,-6))
plt.figure(1)
plt.semilogy(all_cost[0:iters], color="red")
plt.grid()
```

```
plt.title("with step size 0.000001")
plt.xlabel('Iteration')
plt.ylabel('Training loss')
# complete the rest
```

Out[20]: Text(0, 0.5, 'Training loss')





Observations:

1. The learning rate affects the convergence rate of the algorithm. A higher learning rate (0.00001) leads to a slower convergence rate, as indicated by the curved line decreasing slowly after a certain point. On the other hand, a lower learning rate (0.000001) leads to a faster convergence rate, as indicated by the close-to-straight line decreasing rapidly.

2. A higher learning rate may be appropriate when the cost function is not very sensitive to changes in the parameters, while a lower learning rate may be more appropriate when the cost function is highly sensitive to changes in the parameters.

Question 2. Logistic Regression [8 points]

In this question, we will plot the logistic function and perform logistic regression. We will use the breast cancer data set. This data set is described here: https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin.

Each sample is a collection of features that were manually recorded by a physician upon inspecting a sample of cells from fine needle aspiration. The goal is to detect if the cells are benign or malignant.

We could use the `sklearn` built-in `LogisticRegression` class to find the weights for the logistic regression problem. The `fit` routine in that class has an *optimizer* to select the weights to best match the data. To understand how that optimizer works, in this problem, we will build a very simple gradient descent optimizer from scratch.

Loading and visualizing the Breast Cancer Data

We load the data from the UCI site and remove the missing values.

In [21]:
```
names = ['id','thick','size_unif','shape_unif','marg','cell_size','bare',
         'chrom','normal','mit','class']
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/' +
                 'breast-cancer-wisconsin/breast-cancer-wisconsin.data',
                 names=names,na_values='?',header=None)
df = df.dropna()
df.head(6)
```

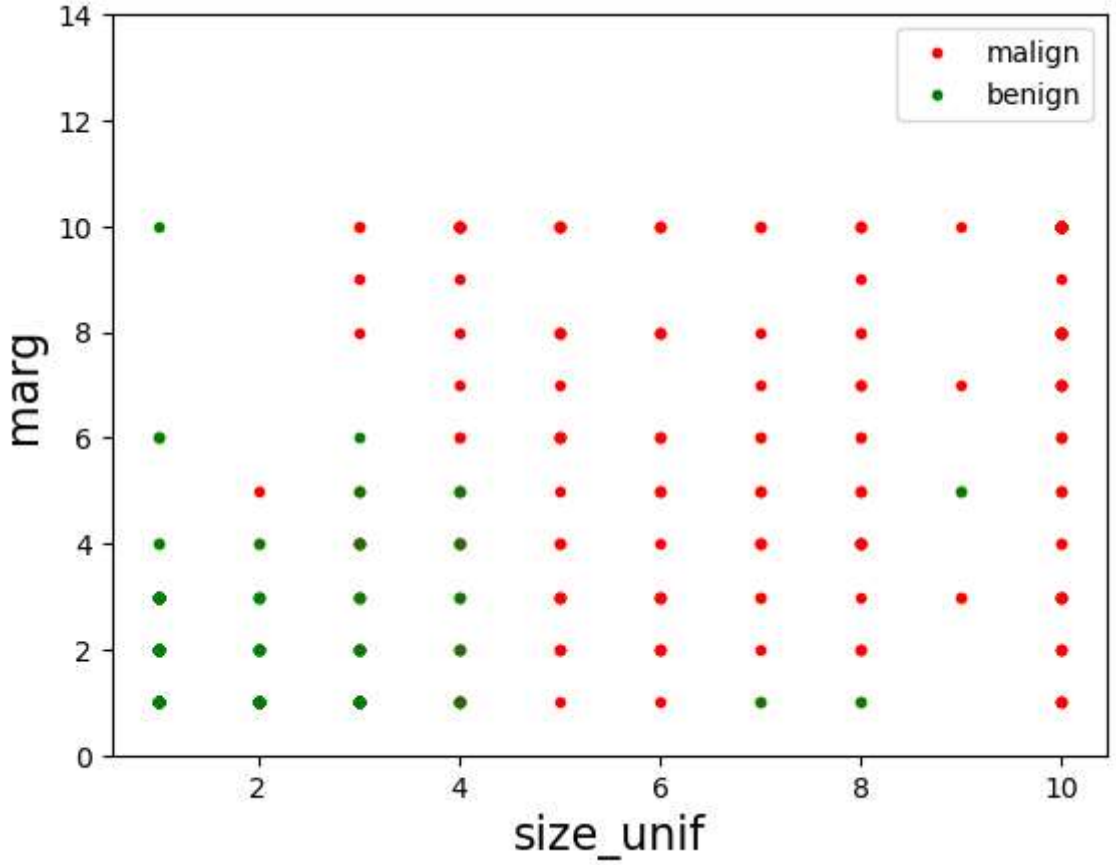
Out[21]:

	id	thick	size_unif	shape_unif	marg	cell_size	bare	chrom	normal	mit	class
0	1000025	5	1	1	1	2	1.0	3	1	1	2
1	1002945	5	4	4	5	7	10.0	3	2	1	2
2	1015425	3	1	1	1	2	2.0	3	1	1	2
3	1016277	6	8	8	1	3	4.0	3	7	1	2
4	1017023	4	1	1	3	2	1.0	3	1	1	2
5	1017122	8	10	10	8	7	10.0	9	7	1	4

After loading the data, we can create a scatter plot of the data labeling the class values with different colors. We will pick two of the features.

In [22]:
```
# Get the response.  Convert to a zero-one indicator
yraw = np.array(df['class'])
BEN_VAL = 2   # value in the 'class' label for benign samples
MAL_VAL = 4   # value in the 'class' label for malignant samples
y = (yraw == MAL_VAL).astype(int)
Iben = (y==0)
Imal = (y==1)

# Get two predictors
xnames =['size_unif','marg']
X = np.array(df[xnames])

# Create the scatter plot
plt.plot(X[Imal,0],X[Imal,1],'r.')
plt.plot(X[Iben,0],X[Iben,1],'g.')
plt.xlabel(xnames[0], fontsize=16)
plt.ylabel(xnames[1], fontsize=16)
plt.ylim(0,14)
plt.legend(['malign','benign'],loc='upper right')
```

Out[22]:
```
<matplotlib.legend.Legend at 0x2649e394a60>
```



The above plot is not informative, since many of the points are on top of one another. Thus, we cannot see the relative frequency of points.

We see that $\sigma(wx+b)$ represents the probability that $y=1$. The function $\sigma(wx)>0.5$ for $x>0$ meaning the samples are more likely to be $y=1$. Similarly, for $x<0$, the samples are more likely to be $y=0$. The scaling w determines how fast that transition is and b influences the transition point.

Fitting the Logistic Model on Two Variables

We will fit the logistic model on the two variables `size_unif` and `marg`.

In [23]:
```
# load data
xnames =['size_unif','marg']
```

```
X = np.array(df[xnames])
#Adding a column of ones to the feature matrix X allows the logistic regression model to have a non-zero intercept term
X_= np.hstack((np.ones((X.shape[0], 1)), X))
print(X.shape)
```

```
(683, 2)
```

Next we split the data into training and test. Use 30% for test and 70% for training. You can do the splitting manually or use the `sklearn` package `train_test_split`. Store the training data in `Xtr,ytr` and test data in `Xts,yts`.

In [24]:
```
# TODO
# Split the data into training and test sets
Xtr, Xts, ytr, yts = train_test_split(X_, y, test_size=0.3, random_state=123)

# Print the shapes of the resulting arrays
print("Xtr shape:", Xtr.shape)
print("Xts shape:", Xts.shape)
print("ytr shape:", ytr.shape)
print("yts shape:", yts.shape)
```

```
Xtr shape: (478, 3)
Xts shape: (205, 3)
ytr shape: (478,)
yts shape: (205,)
```

Logistic regression in scikit-learn

The actual fitting is easy with the `sklearn` package. The parameter `C` states the level of inverse regularization strength with higher values meaning less regularization. Right now, we will select a high value to minimally regularize the estimate.

We can also measure the accuracy on the test data. You should get an accuracy around 90%.

In [25]:
```
from sklearn import datasets, linear_model, preprocessing
reg = linear_model.LogisticRegression(C=1e5)
reg.fit(Xtr, ytr)

print(reg.coef_)
print(reg.intercept_)

yhat = reg.predict(Xts)
acc = np.mean(yhat == yts)
print("Accuracy on test data = %f" % acc)
```

```
[[-2.57550463  1.14260826  0.44225878]]
[-2.57550816]
Accuracy on test data = 0.946341
```

In [ ]:

Instead of taking this approach, we will implement the regression function using gradient descent.

Gradient descent for logistic regression [4 points]

The weight vector can be found by minimizing the negative log likelihood over N training samples. The negative log likelihood is called the *loss* function. For the logistic regression problem, the loss function simplifies to

$$L(\mathbf{w}) = -\sum_{i=1}^{N} y_i \log \sigma(\mathbf{w}^T\mathbf{x}_i + b) + (1 - y_i) \log[1 - \sigma(\mathbf{w}^T\mathbf{x}_i + b)].$$

Gradient can be computed as

$$\nabla_{\mathbf{w}} L = \sum_{i=1}^{N} (\sigma(\mathbf{w}^T\mathbf{x}_i) - y_i)\mathbf{x}_i, \quad \nabla_b L = \sum_{i=1}^{N} (\sigma(\mathbf{w}^T\mathbf{x}_i) - y_i).$$

We can update $\mathbf{w}, b$ at every iteration as

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} L,$$
$$b \leftarrow b - \alpha \nabla_b L.$$

Note that we could also append the constant term in $\mathbf{w}$ and append 1 to every $\mathbf{x}_i$ accordingly, but we kept them separate in the expressions above.

Gradient descent function implementation

We will use this loss function and gradient to implement a gradient descent-based method for logistic regression.

Recall that training a logistic function means finding a weight vector `w` for the classification rule:

$$P(y = 1|x, w) = \frac{1}{1 + e^{-z}}, z = w[0] + w[1] \cdot x[1] + \cdots + w[d] \cdot x[d]$$

The function implemented should follow the following format:

```
def logistic_regression_gd(X,y,learning_rate = 0.001,max_iter=1000,tol=pow(10,-5)):
```

Where `X` is the training data feature(s), `y` is the variable to be predicted, `learning_rate` is the learning rate used (α in the slides), `max_iter` defines the maximum number of iterations that gradient descent is allowed to run, and `tol` is defining the tolerance for convergence (which we'll discuss next).

The return values for the above function should be (at the least):

1. `w` which are the regression parameters,
2. `all_cost` which is an array where each position contains the value of the objective function $L(\mathbf{w})$ for a given iteration,
3. `iters` which counts how many iterations did the algorithm need in order to converge to a solution.

Gradient descent is an iterative algorithm; it keeps updating the variables until a convergence criterion is met. In our case, our convergence criterion is whichever of the following two criteria happens first:

- The maximum number of iterations is met
- The relative improvement in the cost is not greater than the tolerance we have specified. For this criterion, you may use the following snippet into your code:

  ```
  np.absolute(all_cost[it] - all_cost[it-1])/all_cost[it-1] <= tol
  ```

In [26]:

```
# TODO
# Your code for logistic regression via gradient descent goes here

def compute_cost(X, w, y):

    N = len(y)
    z = np.dot(X, w)
    y_hat = 1 / (1 + np.exp(-z))
    L = (-1/N) * np.sum(y*np.log(y_hat) + (1-y)*np.log(1-y_hat))
    return L

def logistic_regression_gd(X, y, learning_rate, max_iter=1000, tol=pow(10,-5)):
    N, D = X.shape
    w = np.zeros(D)
    all_cost = np.zeros(max_iter)
    converged = False
    iters = 0
```

```
    while iters < max_iter and not converged:
        z = np.dot(X, w)
        y_hat = 1 / (1 + np.exp(-z))
        gradient = np.dot(X.T, (y_hat - y)) / N
        w = w - learning_rate * gradient
        all_cost[iters] = compute_cost(X, w, y)
        if iters > 0 and np.absolute(all_cost[iters] - all_cost[iters-1])/all_cost[iters-1] <= tol:
            converged = True
        iters += 1

    return w, all_cost[0:iters], iters
```

Convergence plots and test accuracy [4 points]

After implementing gradient descent for logistic regression, we would like to test that indeed our algorithm converges to a solution. In order see this, we are going to look at the value of the objective/loss function $L(\mathbf{w})$ as a function of the number of iterations, and ideally, what we would like to see is $L(\mathbf{w})$ drops as we run more iterations, and eventually it stabilizes.

The learning rate plays a big role in how fast our algorithm converges: a larger learning rate means that the algorithm is making faster strides to the solution, whereas a smaller learning rate implies slower steps. In this question we are going to test two different values for the learning rate:

- 0.001
- 0.00001

while keeping the default values for the max number of iterations and the tolerance.

- Plot the two convergence plots (cost vs. iterations)
- Calculate the accuracy of classifier on the test data `Xts`
- What do you observe?

Calculate accuracy of your classifier on test data

To calculate the accuracy of our classifier on the test data, we can create a predict method.

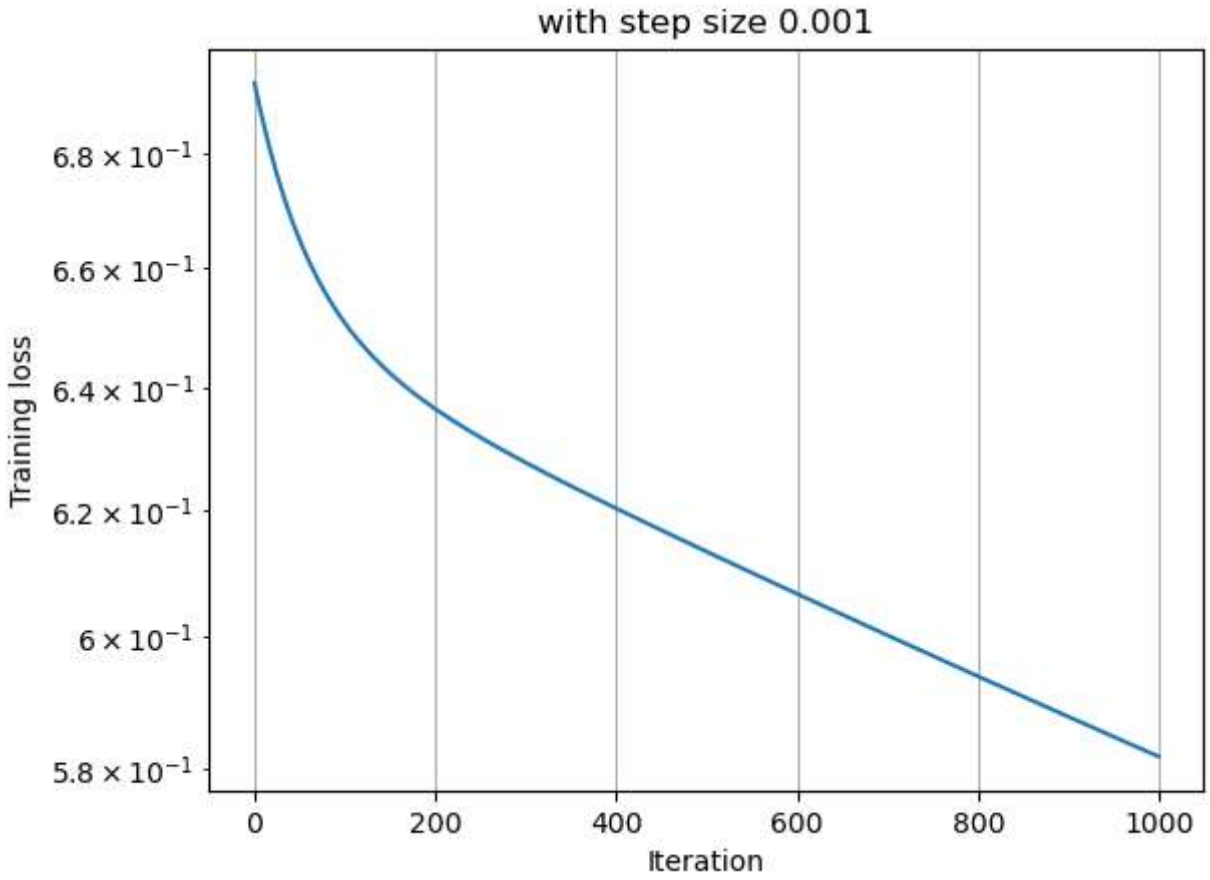
Implement a function `predict(X,w)` that provides you label 1 if $\mathbf{w}^T\mathbf{x} + b > 0$ and 0 otherwise.

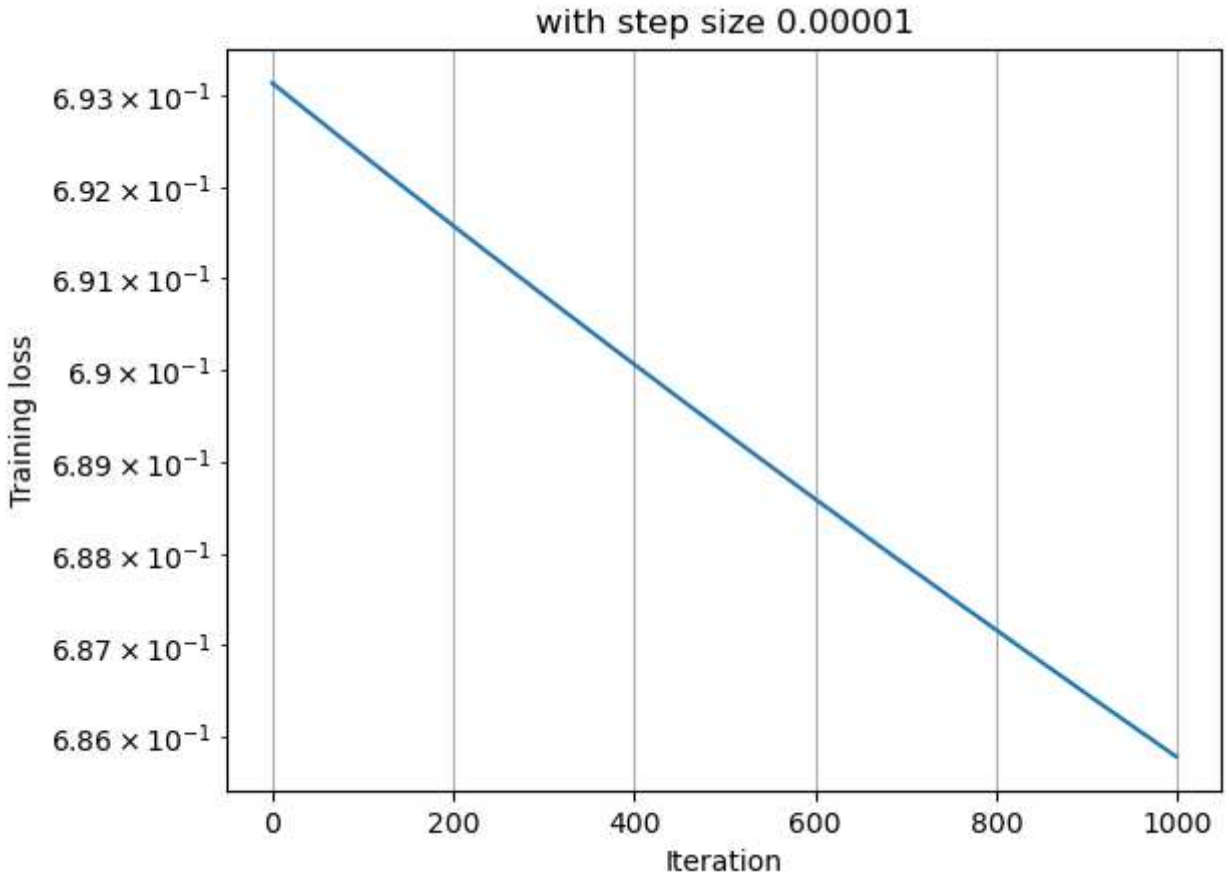
In [27]:
```
# TODO
# Predict on test samples and measure accuracy
```

```
def predict(X, w):
    yhat = np.zeros((X.shape[0],))
    z = np.dot(X, w)
    yhat[z > 0] = 1
    return yhat.astype(int)
```

In [28]:
```
# TODO
# test gradient descent with step size 0.001

(w, all_cost,iters) = logistic_regression_gd(Xtr,ytr,learning_rate = 0.001,max_iter = 1000, tol=pow(10,-6))
plt.figure(0)
plt.semilogy(all_cost[0:iters])
plt.grid()
plt.title("with step size 0.001")
plt.xlabel('Iteration')
plt.ylabel('Training loss')

yhat = predict(Xts,w)
acc = np.mean(yhat == yts)
print("Test accuracy = %f" % acc)

# complete the rest
# test gradient descent with step size 0.00001
(w, all_cost,iters) = logistic_regression_gd(Xtr,ytr,learning_rate = 0.00001,max_iter = 1000, tol=pow(10,-6))
plt.figure(1)
plt.semilogy(all_cost[0:iters])
plt.grid()
plt.title("with step size 0.00001")
plt.xlabel('Iteration')
plt.ylabel('Training loss')

yhat = predict(Xts,w)
acc = np.mean(yhat == yts)
print("Test accuracy = %f" % acc)
```

```
Test accuracy = 0.853659
Test accuracy = 0.356098
```


with step size 0.001
Training loss
6.8×10^{-1}
6.6×10^{-1}
6.4×10^{-1}
6.2×10^{-1}
6×10^{-1}
5.8×10^{-1}
0
200
400
600
800
1000
Iteration



Observations:

1. The test accuracy is significantly different for the two different learning rates. The higher learning rate (0.001) resulted in a much better accuracy (0.853659) compared to the lower learning rate (0.00001) which resulted in a much lower accuracy (0.356098). However, it is noted that if we increase the maximum number of iterations or decrease the learning rate, the accuracy tends to improve. This indicates that the model needs more time to converge to an optimal solution.

2. The training loss graph for the higher learning rate (0.001) shows a curved line decreasing slowly initially and then rapidly decreasing after a certain point, whereas the graph for the lower learning rate (0.00001) is close to a straight line decreasing rapidly. This implies that the higher learning rate requires more iterations to converge to an optimal solution, as the loss initially decreases slowly but then rapidly decreases, while the lower learning rate converges faster as the loss decreases rapidly from the

start. However, a very high learning rate may cause the algorithm to overshoot the minimum and diverge, while a very low learning rate may take too long to converge to an optimal solution.