Homework 1: This HW is based on the code for Single Variable Linear Regression

Instructions:

Place the answer to your code only in the area specified. Also, make sure to run all your code, meaning, press >> to "Restart Kernel and Run All Cells". This should plot all outputs including your answers to homework questions. After this, go to file (top left) and select "Print". Save your file as a PDF and upload the PDF to Canvas.

Question 1:

Why do we divide the gathered data into test and training subsets?

Answer to Question 1:

When training a model we reserve some of the labelled data in order to evaluate the performance of the model during the test phase. This should be data that the model was not trained on as we want to test the ability to process new data.

Loading the Data

The python `pandas` library is a powerful package for data analysis. In this course, we will use a small portion of its features -- just reading and writing data from files. After reading the data, we will convert it to `numpy` for all numerical processing including running machine learning algorithms.

We begin by loading the packages.

In [ ]:
```
import pandas as pd
import numpy as np
```

The data for this demo comes from a survey of cars to determine the relation of mpg to engine characteristics. The data can be found in the UCI library: https://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg

Try 2: Fixing the Errors in the loading

The problems above are common. Often it takes a few times to load the data correctly. That is why it is good to look at the first few elements of the dataframe before proceeding. After some googling you can find out that you need to specify some other options to the `read_csv` command. First, you need to supply the names of the columns. In this case, I have supplied them manually based on the description in the UCI website. I store them in a list object, you can check its type with a 'type' command.

In [ ]:
```
names = ['mpg', 'cylinders','displacement', 'horsepower',
         'weight', 'acceleration', 'model year', 'origin', 'car name']
type(names)
```

Out[ ]:
```
list
```

In [ ]:
```
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/'+
                 'auto-mpg/auto-mpg.data',
                 header=None,delim_whitespace=True,names=names,na_values='?')
```

If you re-run `head` command now, you can see the loading was correct. You can see the column names, index, and values:

Manipulating the Data

We can get the `shape` of the data, which indicates the number of samples and number of attributes

In [ ]:
```
df.shape
```

Out[ ]:
```
(398, 9)
```

You can also see the three components of the `dataframe` object. The dataframe is stored in a table (similar to a SQL table if you know databases). In this case, there is one row for each car and the attributes of the car are stored in the columns. The command `df.columns` returns the names of the columns.

In [ ]:
```
df.head(6)
```

Out[ ]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino
5	15.0	8	429.0	198.0	4341.0	10.0	70	1	ford galaxie 500

In [ ]:
```
df.columns
```

Out[ ]:
```
Index(['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'model year', 'origin', 'car name'],
      dtype='object')
```

The field `df.index` returns the indices of the rows. In this case, they are just enumerated 0,1,...

In [ ]:
```
df.index
```

Out[ ]:
```
RangeIndex(start=0, stop=398, step=1)
```

Finally, `df.values` is a 2D array with values of the attributes for each car. Note that the data can be *heterogeneous*: Some entries are integers, some are floating point values and some are strings.

In [ ]:
```
df.values
```

Out[ ]:
```
array([[18.0, 8, 307.0, ..., 70, 1, 'chevrolet chevelle malibu'],
       [15.0, 8, 350.0, ..., 70, 1, 'buick skylark 320'],
       [18.0, 8, 318.0, ..., 70, 1, 'plymouth satellite'],
       ...,
       [32.0, 4, 135.0, ..., 82, 1, 'dodge rampage'],
       [28.0, 4, 120.0, ..., 82, 1, 'ford ranger'],
       [31.0, 4, 119.0, ..., 82, 1, 'chevy s-10']], dtype=object)
```

The `df.columns` attribute is not a python list, but a `pandas` -specific data structure called an `Index` . To convert to a list, use the `tolist()` method:

In [ ]:
```
df.columns.tolist()
```

Out[ ]:
```
['mpg',
 'cylinders',
 'displacement',
 'horsepower',
 'weight',
 'acceleration',
 'model year',
 'origin',
 'car name']
```

You can select subsets of the attributes with indexing. For example, this selects one attribute, which returns what is called a pandas `Series`

In [ ]:
```
df2 = df['cylinders']
df2.head(6)
```

Out[ ]:
```
0    8
1    8
2    8
3    8
4    8
5    8
Name: cylinders, dtype: int64
```

You can also select a list of column names which returns another dataframe. Note the use of the double brackets `[[ ... ]]` .

In [ ]:
```
df2 = df[['cylinders','horsepower']]
df2.head(6)
```

Out[ ]:

	cylinders	horsepower
0	8	130.0
1	8	165.0
2	8	150.0
3	8	150.0
4	8	140.0
5	8	198.0

Plotting the Data

We load the `matplotlib` module to plot the data. This module has excellent plotting routines that are very similar to those in MATLAB

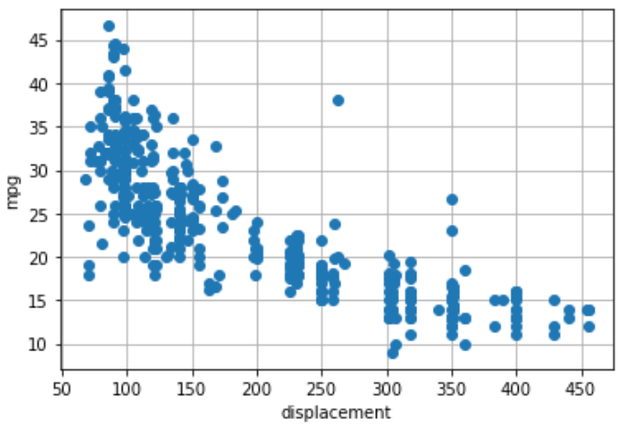
In [ ]:
```
import matplotlib
import matplotlib.pyplot as plt
```

First, we need to convert the dataframes to numpy arrays:

In [ ]:
```
xstr = 'displacement'
x = np.array(df[xstr])
y = np.array(df['mpg'])
```

Then, we can create a scatter plot

In [ ]:
```
plt.plot(x,y,'o')
plt.xlabel(xstr)
plt.ylabel('mpg')
plt.grid(True)
```



Manipulating Numpy arrays

Once the data is converted to a numpy array, we can perform many useful simple calculations. For example, we can compute the sample mean:

In [ ]:
```
mx = np.mean(x)
my = np.mean(y)
print('Mean {0:s} = {1:5.1f}, mean mpg= {2:5.1f}'.format(xstr, mx, my))
```

```
Mean displacement = 193.4, mean mpg=  23.5
```

We can also find fraction of cars with > 25 mpg:

In [ ]:
```
np.mean(y > 25)
```

Out[ ]:
```
0.3969849246231156
```

Sample mean displacement for the cars that have mpg > 25

In [ ]:
```
I = (y>25)
print(np.mean(x*I)/np.mean(I))
```

```
110.08227848101266
```

You can also do the previous command with boolean indexing.

In [ ]:
```
np.mean(x[I])
```

Out[ ]:
```
110.08227848101266
```

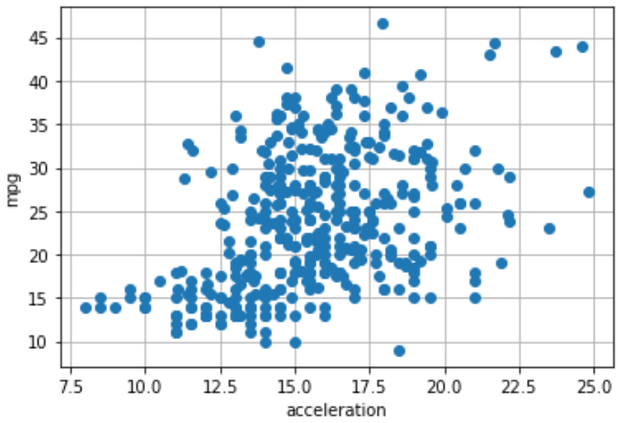
Question 2:

Use the techniques above to:

- Load the acceleration variables in `df['acceleration']` to a `np.array` called `acc` .
- Create a scatter plot of `mpg` vs. `acc` .
- Add grid lines to your plot and label the axes with the `plt.xlabel` and `plt.ylabel` functions.

Note that the acceleration here is the time from going to 0 to 60 mph, so a higher number is a *lower* acceleration. That is, `acc` is really inverse acceleration.

In [ ]:
```
# Answer to Question 2:
xstr = 'acceleration'
ystr = 'mpg'
acc = np.array(df[xstr])
mpg = np.array(df[ystr])
plt.plot(acc,mpg,'o')
plt.xlabel(xstr)
plt.ylabel(ystr)
plt.grid(True)
```



Question 3:

Find the average mpg of cars that have `acc > 15` . Print your result.

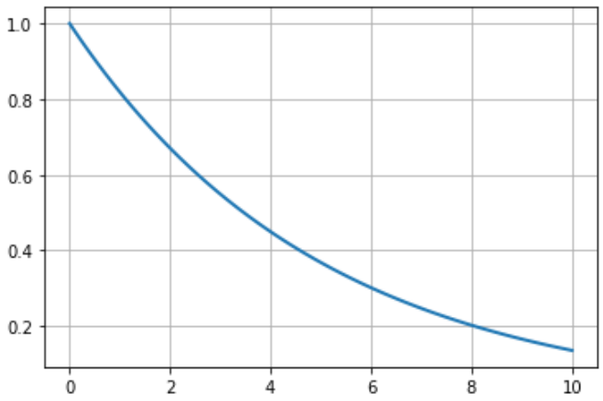
In [ ]:
```
# Answer to Question 3:
np.mean(mpg[acc > 15])
```

Out[ ]:
```
25.850230414746544
```

Plotting Functions

The `matplotlib.pyplot` package combined with `numpy` has very powerful tools for creating plots. For example, suppose we wish to plot `f1(t) = exp(-0.2*t)` vs. `t` . For those familiar with MATLAB, the syntax is very similar.

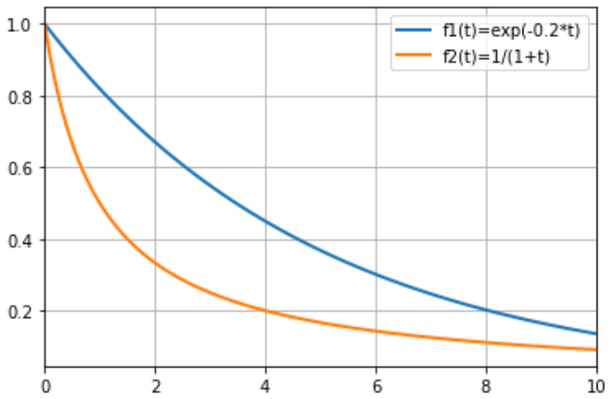
In [ ]:
```
t = np.linspace(0,10,100)  # 100 points linearly spaced from 0 to 5
f1 = np.exp(-0.2*t)
plt.plot(t,f1,lw=2)  # lw=2 makes a little thicker line.  easier to read
plt.grid()
```



We can also super-impose plots:

In [ ]:
```
f2 = 1/(1+t)
plt.plot(t,f1,lw=2)
plt.plot(t,f2,lw=2)
plt.grid()
plt.xlim([0,10])
plt.legend(['f1(t)=exp(-0.2*t)', 'f2(t)=1/(1+t)'])
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7fea540f6d30>
```



Missing Data and NaN Values

Now, try a different field, horsepower

In [ ]:
```
xstr = 'horsepower'
x = np.array(df[xstr])
y = np.array(df['mpg'])
np.mean(x)
```

Out[ ]:
```
nan
```

When you get the mean, it gives `nan` which means not a number. The reason is that there was missing data in the orginal file and the `load_csv` function put `nan` values in the places where the data was missing. This is very common. To remove the rows with the missing data, we can use the `dropna` method:

In [ ]:
```
df1 = df[['mpg','horsepower']]
df2 = df1.dropna()
df2.shape
```

Out[ ]:
```
(392, 2)
```

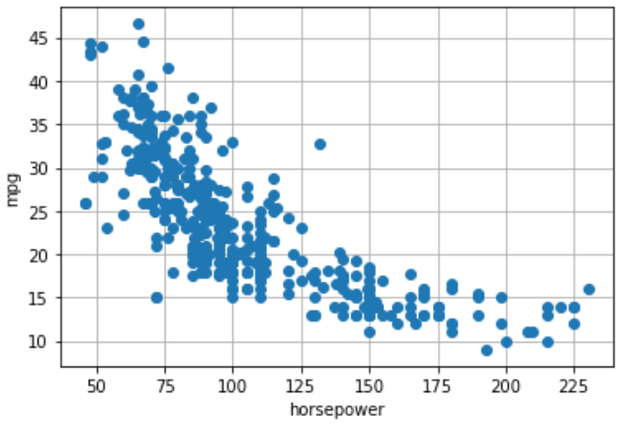
We can see that some of the rows have been dropped. Specifically, the number of samples went from 396 to 392. We can now compute the mean using the reduced dataframe.

In [ ]:
```
x = np.array(df2['horsepower'])
y = np.array(df2['mpg'])
np.mean(x)
```

Out[ ]:
```
104.46938775510205
```

And, we can plot the data.

In [ ]:
```
plt.plot(x,y,'o')
plt.xlabel(xstr)
plt.ylabel('mpg')
plt.grid(True)
```



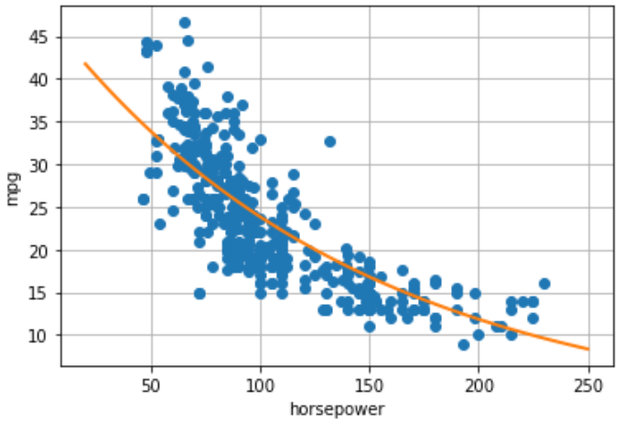
Question 4:

Guessing a fit for data. From the scatter plot above, you can see there is a relation between `y` vs. `x`. Machine learning is about learning these relations. We will discuss many ways to fit these relations automatically, but let us see first if you can guess a decent relation.

- Guess some relation `yhat = f(x)` where `yhat` is the predicted value of `y` given `x`. So, `f(x)` should be some function that matches the data you see well.
- To visualize the relation, create a vector `xp = np.linspace(20,250,100)` on which you will plot the values of your predicted function.
- Compute `yhatp` on the values of `xp`.
- On a single plot, plot the `yp` vs. `xp` as well as the scatter plot of `y` vs. `x`.

In [ ]:
```
# Answer to Question 4:

plt.plot(x,y,'o')
plt.xlabel(xstr)
plt.ylabel('mpg')
plt.grid(True)

xp = np.linspace(20,250,100)
yp = 48 * np.exp(-0.007*xp)
plt.plot(xp,yp,lw=2)
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7fea54303250>]
```



Splitting Data into Training and Test

Now we will try to optimally *fit* a model to the data. When doing this, we need to split the data into training and test, where we fit the model on the training data and evaluate it on the test data. We will discuss this in detail in subsequent lectures, but the reason we need to do this is that we want to evaluate the fit on *new* data points, not in the test data.

To split the data, we take some fraction, say 0.5, for training and the other fraction for test.

In [ ]:
```
n = len(x)      # Total number of samples
ntr = n // 2    # number of training samples
nts = n - ntr   # number of test samples

print('number of samples = %d' % n)
print('number of training = %d' % ntr)
print('number of test = %d' % nts)
```

```
number of samples = 392
number of training = 196
number of test = 196
```

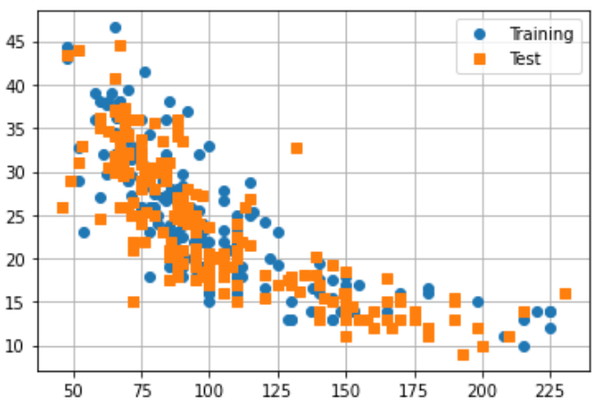
We then shuffle the samples and get the first `ntr` for training and the remaining for test. Note that square brackets are used for indexing and : is used for slicing: https://docs.scipy.org/doc/numpy-1.13.0/user/basics.indexing.html

In [ ]:
```
I = np.random.permutation(n)

# Training samples
xtr = x[I[:ntr]]
ytr = y[I[:ntr]]

# Test
xts = x[I[ntr:]]
yts = y[I[ntr:]]
```

We can plot the training and test samples.

In [ ]:
```
plt.plot(xtr,ytr,'o')
plt.plot(xts,yts,'s')
plt.grid()
plt.legend(['Training', 'Test'])
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7fea54325610>
```



Computing and Plotting a Linear Fit

One simple and widely-used model is the linear fit, `yhat = beta0 + beta1*x` where:

- `beta0` is called the *intercept* or *bias*
- `beta1` is called the *slope* or or *weight*

In class, we derive optimal formulae:

`beta1 = syx / sxx,  beta0 = ym - beta1*xm`

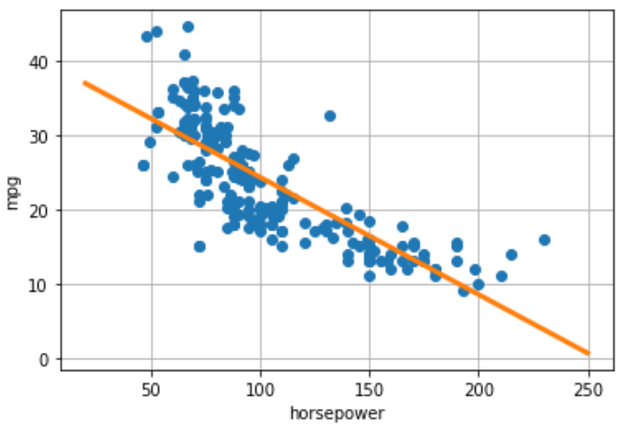
where `xm` and `ym` are the sample means and `syx` and `sxx` are the cross- and auto-covariances. We find the parameters on the training data.

In [ ]:
```
xm = np.mean(xtr)
ym = np.mean(ytr)
syy = np.mean((ytr-ym)**2)
syx = np.mean((ytr-ym)*(xtr-xm))
sxx = np.mean((xtr-xm)**2)
beta1 = syx/sxx
beta0 = ym - beta1*xm

print("xbar     ={0:7.2f},       ybar={1:7.2f}".format(xm,ym))
print("sqrt(sxx)={0:7.2f},  sqrt(syy)={1:7.2f}".format(np.sqrt(sxx),np.sqrt(syy)))
print("beta0={0:7.2f}, beta1={1:7.2f}".format(beta0,beta1))
```

```
xbar     = 104.13,       ybar=  23.67
sqrt(sxx)=  37.49,  sqrt(syy)=   7.71
beta0=  40.11, beta1=  -0.16
```

We can create a plot of the regression line on top of the scatter plot. The vector `xplt` are the x-coordinates of the two endpoints of the line. They are chosen so that the line fits nicely in the plot.

In [ ]:
```
# Points on the regression line
xplt = np.array([20,250])
yplt = beta1*xplt + beta0

plt.plot(xts,yts,'o')                    # Plot the data points
plt.plot(xplt,yplt,'-',linewidth=3)  # Plot the regression line
plt.xlabel(xstr)
plt.ylabel('mpg')
plt.grid(True)
```



We next compute the mean squared error (MSE) of the fit:

- `mse_tr` : MSE on the training data. This number is how well model fits the training data.
- `mse_ts` : MSE on the test data. This number is more useful since it represents how well the fit is on *new* points not in the training data set.

We will discuss the difference between training and test more later. In this case, you may see that the MSE on the test data is a little higher.

In [ ]:
```
# Training MSE
yhat_tr=beta0+beta1*xtr
mse_tr = np.mean((ytr-yhat_tr)**2)

# Test MSE
yhat_ts =beta0+beta1*xts
mse_ts = np.mean((yts-yhat_ts)**2)

print("MSE training = %7.4f" % mse_tr)
print("MSE test     = %7.4f" % mse_ts)
```

```
MSE training = 24.4272
MSE test     = 23.5152
```

Let us see whether this is the same as the analytically derived minimal RSS

In [ ]:
```
rxy=syx/np.sqrt(sxx)/np.sqrt(syy)
mse_min =(1-rxy*rxy)*syy
print('MSE theoretical = %7.4f' % mse_min)
```

```
MSE theoretical = 24.4272
```

Question 5:

Find the MSE on the test data points for the following three regions of `x` :

- `x <= 100`
- `x > 100 and x <= 150`
- `x > 150`

In [ ]:
```
# Answer to Question 5

# x <= 100
mask = (xts <= 100)
mse_ts = np.mean((yts-yhat_ts)[mask] ** 2)
print(mse_ts)

# x > 100 and x <= 150
mask = ((xts > 100) & (xts <= 150))
mse_ts = np.mean((yts-yhat_ts)[mask] ** 2)
print(mse_ts)

# x > 150
```

```
mask = (xts > 150)
mse_ts = np.mean((yts-yhat_ts)[mask] ** 2)
print(mse_ts)
```

```
28.979997438946935
15.872259970092555
13.097987341041911
```