Machine Learning: Assignment

IMPORTANT: I am using IPython 3.4

At first we imported some libraries that we need for next functionalities:

In [39]:

```
%matplotlib inline
import sys
import os
from pandas import *
import pandas as pd
from matplotlib import *
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt
from collections import Counter
```

We will continue with the tasks:

Part 1: import the .csv files

Since the csv files might not be in the same location as the one I put them in, I am taking the absolute path of the specific files. However, IPython will take the path of the .ipynb code and that's why it's good to save copies of the .csv files in the same folder as the IPython script.

In [40]:

```
white_path = os.path.abspath('winequality-white.csv')
red_path = os.path.abspath("winequality-red.csv")

white_data = read_csv(white_path, sep=";")
# we use sep = ";", because the sefault is something else and will not read the files correctly
red_data = read_csv(red_path, sep = ";")
```

Then we take lists of values of both .csv files that are only for the target column - "quality".

Part 2: plot bar-plots of the number of examples with each target value for the two datasets

In [41]:

```
target_list_w = white_data['quality'] # we can take that, also, with the function white_data.drop('qual
ity', axis = 1)
target_list_r = red_data['quality']
```

Afterwards, we create a function for creating a bar chart from a specific list (that will be the lists from the values of target columns):

In [42]:

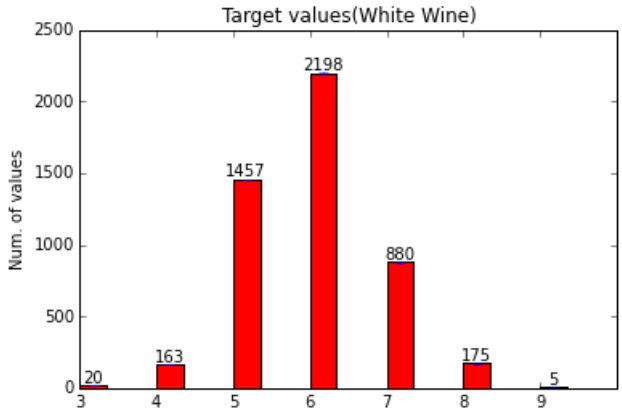
```
# bar charts for the number of values in the target list in both csv files
def bar_chart(list, str):
    target_w_dict = Counter(list)
    diff_values = len(target_w_dict.keys()) # it can be only the length of the dictionary
    width = 0.35
    num_values = []
    N = diff_values
    ind = np.arange(N)

    for i in range(0, diff_values):
        num_values.append(i)
    keys = []
    values = []
```

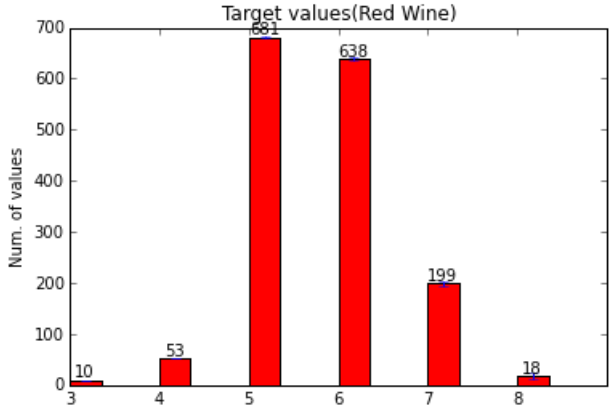
```
    for key in target_w_dict.keys():
        keys.append(key)
    for v in target_w_dict.values():
        values.append(v)


    fig, ax = plt.subplots()
    rects1 = ax.bar(ind, values, width, color='r', yerr=num_values)

    ax.set_ylabel('Num. of values')
    ax.set_title('Target values' + str)
    ax.set_ylim(bottom=0.)
    ax.set_xticks(ind)
    ax.set_xticklabels(keys)

    for rect in rects1:
        height = rect.get_height()
        ax.text(rect.get_x()+rect.get_width()/2., height, '%d'%int(height),
                ha='center', va='bottom')
```

Then we plot the bars:

In [43]:

```
str_red = "(Red Wine)"
str_white = "(White Wine)"
bar_chart(target_list_w, str_white)
bar_chart(target_list_r, str_red)
# figure(figsize(18,10)) #not working here, but it's good to make the bar bigger
plt.show()
```





Part 3: How might these distributions effect the analysis?

As we can see from the bar chart of the white wine analysis the most common value of the quality of the white wine is 6. For the red wine is 5. Since the "quality" attribute is the target value for the analysis, this means that it's calculated and it depends from the other values(the features). Also, when we compute Linear Regression, Regularised linear regression or we classify the data, we might see that some points(on the charts that we will compute) are really close to each other, since we don't have a lot of variation of values in the quality. For the most common values in the "quality" attribute, the values for the features will be really close to each other, because they compute the same value(e.g. 6 for white wine or 5 for red wine).

Part 4: Linear regression:

The objective is to find the best-fitting straight line through a set of points that minimizes the sum of the squared offsets from the line.

4.(a)

For this part we will split the data (randomly) into training set and test set. 70% from the data will be for the training set and the rest for the test set.
The data that we will be using is: red_data.
The target set of our red wine data is: target_list_r Also, we will put the data from the red wine data into a list of the rows in the table(it will be easier to iterate through the data and after that manipulate it).

In [44]:

```
red_raw_data = []
for row in range(0, len(red_data)):
    red_raw_data.append(red_data.irow(row))
print(len(red_raw_data))
```

```
1599
```

In order to split the red wine data into training set and test set we need to know how much is 70% and 30% (in length) in the data:

In [45]:

```
train_len = int((7*len(target_list_r))/10) # The value is 1119.3. but we will try to be as close as pos
sible to this value
test_len = int(len(target_list_r) - train_len)
print(train_len)
print(test_len)
```

```
1119
480
```

Then we will take random rows from the data.
(The method below runs a little bit slower(you have to wait several seconds))

In [46]:

```
import random

training_set = []
test_set = []


for tr in range(0, len(red_raw_data)):
    rand_num = random.choice(red_raw_data)
    if(len(training_set)>=train_len):
        for t in test_set:
            if(t.equals(rand_num)):
                rand_num = random.choice(red_raw_data)
        test_set.append(rand_num)
    else:
        for t in training_set:
            if(t.equals(rand_num)):
                rand_num = random.choice(red_raw_data)
        training_set.append(rand_num)
    tr +=1
print(len(test_set))
```

```
480
```

4.(b)

In order to fit a linear regression we need first to convert the data into matrix form(or vector) and then calculate the optimal value:

In [47]:

```
train_x = []
column_list = red_data.columns.values
for x in training_set:
    train_x.append(x.drop('quality'))

    train_target = [] #dividing the target data from the other data
for x in training_set:
    train_target.append(x.drop(column_list[:11]))

test_x = []
for x in test_set:
    test_x.append(x.drop('quality'))

    test_target = [] #dividing the target data from the other data
for x in test_set:
    test_target.append(x.drop(column_list[:11]))
```

Matrix and vector form
We first defined $\mathbf{w},\mathbf{x}_n$ as: $ \mathbf{w} = \left[\begin{array}{c} w_0\\w_1\end{array}\right],~~\mathbf{x}_n = \left[\begin{array}{c} 1 \\ x_n \end{array}\right] $ and: $ \mathbf{t} = \left[\begin{array}{c} t_1\\ t_2\\ \vdots\\ t_N \end{array}\right]$ and: $ \mathbf{X} = \left[\begin{array}{c} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_N^T \end{array}\right] = \left[\begin{array}{cc} 1 & x_1 \\ 1 & x_2 \\ \vdots \\ 1 & x_N \end{array}\right] $

The optimal value of $\mathbf{w}$ is then given by: $ \mathbf{w} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{t} $

In [48]:

```
t = np.array(train_target)
train_x_copy = train_x
x = np.array(train_x_copy)
x = np.c_[np.ones(x.shape[0]), train_x_copy]
w = np.dot(np.linalg.inv(np.dot(x.T,x)),np.dot(x.T,t))
print(w)
```

```
[[  8.93559511e+00]
 [  1.64038890e-02]
 [ -1.15274969e+00]
 [ -4.45043275e-01]
 [  2.01739173e-02]
 [ -1.75634785e+00]
 [  1.06693372e-03]
 [ -2.38298770e-03]
 [ -4.58445471e+00]
 [ -5.21267451e-01]
 [  8.00971306e-01]
 [  3.12780589e-01]]
```

4.(c)

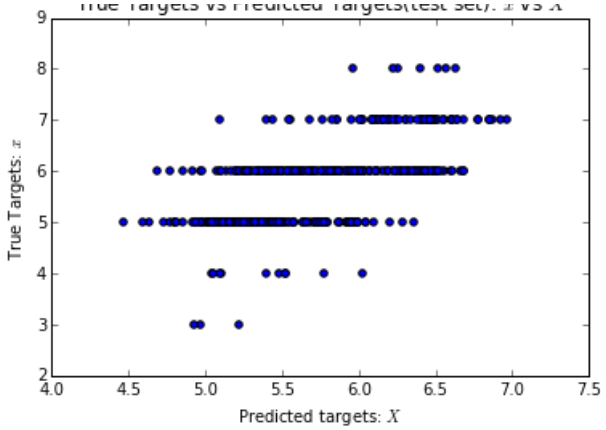
Now I am plotting(creating scatter plot) the predicted values against the True values.

In [49]:

```
test_x_copy = test_x
t_x = np.array(test_x_copy)
t_x_c = np.c_[np.ones(t_x.shape[0]), test_x_copy] # the matrix with additional column 1

f_X = np.dot(t_x_c,w) # obtain predictions
tr = np.array(test_target)
plt.scatter(f_X, tr)
#plt.plot(test_x,t,'ro')
plt.xlabel("Predicted targets: $X$")
plt.ylabel("True Targets: $x$") # x is the test_target -- > quality values in the test_set
plt.title("True Targets vs Predicted Targets(test set): $x$ vs $X$")
```

Out[49]:

```
<matplotlib.text.Text at 0x90eaeb8>
```



Now I will calculate the mean squared error(for the test set):

In [50]:

```
mean_se = np.mean((tr - f_X) ** 2)
print(mean_se)
```

```
0.395610697109
```

4.(d)

A mathematical procedure for finding the best-fitting curve to a given set of points by minimizing the sum of the squares of the offsets ("the residuals") of the points from the curve. The sum of the squares of the offsets is used instead of the offset absolute values because this allows the residuals to be treated as a continuous differentiable quantity. However, because squares of the offsets are used, outlying points can have a disproportionate effect on the fit, a property which may or may not be desirable depending on the problem at hand.
I implement a naive least squares method for linear regression, which is the simplest approach to performing a regression analysis of a dependent and a explanatory variable. This is a good approach for estimating if our mean square value is good. The linear least squares fitting technique is the simplest and most commonly applied form of linear regression and provides a solution to the problem of finding the best fitting straight line through a set of points.

4.(e)

Variance score: 1 is perfect prediction and 0 means that there is no linear relationship between X and Y.

In [51]:

```
from sklearn.linear_model import LinearRegression
regr = LinearRegression()
regr.fit(train_x, train_target)
regr.score(test_x, test_target)
```

Out[51]:

```
0.39595016022502472
```

In [52]:

```
np.linalg.lstsq(x, t)[0] # z = a.x+ b.y+ c
```

Out[52]:

```
array([[  8.93559511e+00],
       [  1.64038890e-02],
       [ -1.15274969e+00],
       [ -4.45043275e-01],
       [  2.01739173e-02],
       [ -1.75634785e+00],
       [  1.06693372e-03],
       [ -2.38298770e-03],
       [ -4.58445470e+00],
       [ -5.21267451e-01],
       [  8.00971306e-01],
       [  3.12780589e-01]])
```

4.(f)

From 4.(d) we can see reasons why the least square method might be better for estimating the bestt-fitting straight line through a set of points. From what we have compute in 4.(e), we see the least-squares solution to our linear matrix equation.

Part 5: Regulized Linear Regression

5.(a)

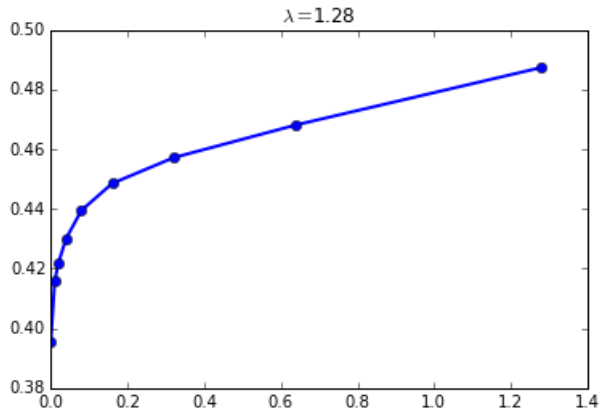
In [53]:

```
lambs = [0,0.01,0.02,0.04,0.08,0.16,0.32,0.64,1.28]
sq_err = []
for lamb in lambs:
    #print(lamb*np.identity(x.shape[1]))
    np_lamb = np.dot(x.T,x) + x.shape[0]*lamb*np.identity(x.shape[1])
    wL = np.dot(np.linalg.inv(np_lamb),np.dot(x.T,t))
    f_test = np.dot(t_x_c,wL) #predicted
    mse = np.mean((tr - f_test)**2)
    sq_err.append(mse)

print(sq_err)
print(len(lambs))
plt.figure()
plt.plot(lambs,sq_err,'-o',linewidth=2)
#plt.plot(x,y,'ro')
title = '$\lambda=$%g'%lamb
plt.title(title)
```

```
[0.39561069710939856, 0.41589348583500862, 0.4219510672289567, 0.43012232477622292, 0.43948585487949832
, 0.44850267513856351, 0.45712459011298123, 0.46811103872120119, 0.48734776848205225]
9
```

Out[53]:

```
<matplotlib.text.Text at 0x9029c18>
```



5.(b)

It's not a good way of determining the value of the regularisation parameter, because we are taking the test and training sets by randomisation - it's better to choose them by cross-validation (more precise).

5.(c)

Implement a 10-fold CV on the training data and use this to determine the value of the regularisation parameter.

In [16]:

```
K = 10
sizes = np.tile(np.floor(len(x)/10), (1,K))
```

```
sizes = np.tile(np.floor(len(x)/10),(1,K))
sizes[-1] = sizes[-1] + len(x) - sizes.sum()
c_sizes = np.hstack((0,np.cumsum(sizes)))

cv_loss = np.zeros((K,len(lambs)))
ind_loss = np.zeros((K,len(lambs)))
train_loss = np.zeros((K,len(lambs)))
perf_lamb = 0
opt_vals = []

# we will go through each of the K folds; then we iterate through the lambs as well;
for fold in range(K):
        X_fold = x[c_sizes[fold]:c_sizes[fold+1],:]
        X_train = np.delete(x,np.arange(c_sizes[fold],c_sizes[fold+1],1),0)
        t_fold = t[c_sizes[fold]:c_sizes[fold+1]]
        t_train = np.delete(t,np.arange(c_sizes[fold],c_sizes[fold+1],1),0)
        for lamb in lambs:
            np_lamb2 = np.dot(x.T,x) + x.shape[0]*lamb*np.identity(x.shape[1])
            w2 = np.dot(np.linalg.inv(np_lamb2),np.dot(x.T,t))
            fold_pred = np.dot(X_fold,w2)
            index_l = lambs.index(lamb)
            cv_loss[fold][index_l]=((fold_pred[:,0] - t_fold[:,0])**2).mean()
            ind_pred = np.dot(t_x_c,w2)
            ind_loss[fold][index_l]=((ind_pred[:,0] - tr[:,0])**2).mean()
            train_pred = np.dot(X_train,w2)
            if(lamb==0):
                perf_lamb = ((ind_pred[:,0] - tr[:,0])**2).mean()
            train_loss[fold][index_l]=((train_pred[:,0] - t_train[:,0])**2).mean()
            opt_vals.append(w2)
print("The optimal value here is 0 and its preformance in the test set is: " + str(perf_lamb))
```

```
The optimal value here is 0 and its preformance in the test set is: 0.421364298855
```

Below I am just plotting the training loss, CV loss and the test set loss in order to see how it looks on the plot.

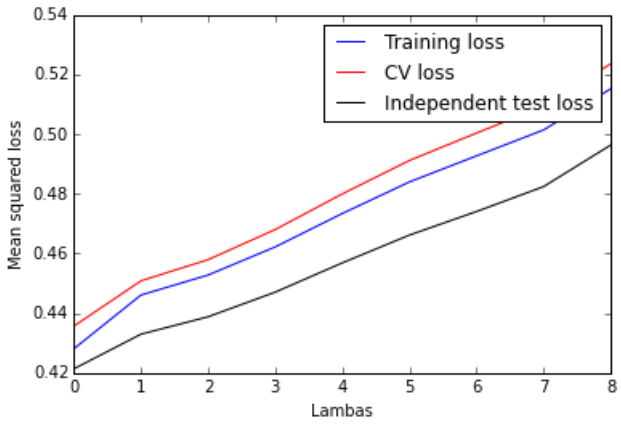
In [36]:

```
import pylab as plt
%matplotlib inline
order = np.arange(9)
print(train_loss.mean(axis=0))
plt.plot(order,train_loss.mean(axis=0),'b-',label="Training loss")
plt.plot(order,cv_loss.mean(axis=0),'r-',label="CV loss")
plt.plot(order,ind_loss.mean(axis=0),'k',label="Independent test loss")
plt.legend()
plt.xlabel('Lambas')
plt.ylabel('Mean squared loss')
```

```
[ 0.4280711   0.44607938  0.45277246  0.46218878  0.47338901  0.48396073
  0.49269804  0.50137802  0.51515431]
```

Out[36]:

```
<matplotlib.text.Text at 0xb6e6940>
```



5.(d)

The training loss, the cv loss and the test set loss really depends from the randomisation(of how we have distributed) of the data into the training set and the test set. Because of that we might have differneces(changes) in the performances of both of the

models. The minimum mean square loss in the CV goes(around) to 0.401 value, and for the linear regression - to 0.039 (below 0.401). The change in the performance might ocure, also, because of CV is distributing the data into folds - which is giving more precise results. The validation process(in the CV) can involve analyzing the goodness of fit of the regression, analyzing whether the regression residuals are random, and checking whether the model's predictive performance deteriorates substantially when applied to data that were not used in model estimation.

Part 6: Classification

6.(a)

The regression that we have leads to dense solutions, in which most coefficients are non-zero. The regression considers only residuals in the dependent variable. The dependent variables would be a subject to the same types of observation error as those in the data used for fitting. Regression means to predict the output value using training data. Classification means to group the output into a class.

6.(b)

I have choosen to implement the KNN(k-Nearest Neighbors) classifier.
Advantage: If our training set was small, high bias/low variance classifiers (Naive Bayes) have an advantage over low bias/high variance classifiers (KNN), since the latter will overfit. But low bias/high variance classifiers sare better, because our training set is big (they have lower asymptotic error), since high bias classifiers aren't powerful enough to provide accurate models.
Disadvantage: Naive Bayes has simpler implementation than the KNN. If the NB conditional independence assumption actually holds, a Naive Bayes classifier will converge quicker. And even if the NB assumption doesn't hold, a NB classifier still often performs surprisingly well in practice.

6.(c)

The data that I pre-process is the test and train sets of the red wine data CSV. All of the pre-processing method for this algorithm are explained below. The list of the methods is:
euclideanDistance: Calculate the distance between two data instances. **getNeighbors**: Locate k most similar data instances. **getResponse**: Generate a response from a set of data instances. **getAccuracy**: Summarize the accuracy of predictions. **The last calculations**: Tie it all together.

6.(d)

To make predictions I am calculating the similarity between any two given data instances. I am locating the k most similar data instances in the training set for a given member of the test set and in turn make a prediction. I am using the Euclidean distance algorithm. It's calculating the square root of the sum of the squared differences between the two arrays of numbers. I am including the first 4 attributes. I am putting the euclidean distance into a fixed length, ignoring the final dimension.

In [35]:

```
import math
def euclideanDistance(ins1, ins2, leng):
    dist = 0
    for x in range(leng):
        dist += pow((ins1[x] - ins2[x]), 2)
    return math.sqrt(dist)
```

I am collecting the k most similar instances for a given unseen instance. I am calculating the distance for all instances and selecting a subset with the smallest distance values. The method below returns k most similar neighbors from the training set for a given test instance (using the already defined euclideanDistance function).

In [34]:

```
import operator
def getNeighbors(trainSet, testIns, k):
    distances = []
    length = len(testIns)-1
    for x in range(len(trainSet)):
        dist = euclideanDistance(testIns, trainSet[x], length)
```

```
            distances.append((trainSet[x], dist))
        distances.sort(key=operator.itemgetter(1))
        neighbors = []
        for x in range(k):
            neighbors.append(distances[x][0])
        return neighbors
```

The next method computes the predicted response based on those neighbors. I am doing this by allowing each neighbor to vote for their class attribute, and take the majority vote as the prediction. The method assumes that the class is the last attribute for each neighbor.

In [30]:

```
def getResponse(neighb):
    classVotes = {}
    for x in range(len(neighb)):
        resp = neighb[x][-1]
        if resp in classVotes:
            classVotes[resp] += 1
        else:
            classVotes[resp] = 1
    sortedVotes = sorted(classVotes.items(), key=operator.itemgetter(1), reverse=True) # using Python3.
4 --> items() is the same as iteritems() here
    return sortedVotes[0][0]
```

Now we will evaluate the accuracy of predictions. To do that is to calculate a ratio of the total correct predictions out of all predictions made, called the classification accuracy. The method below sums the total correct predictions and returns the accuracy as a percentage of correct classifications.

In [31]:

```
# compute the accuracy
def getAccuracy(testSet, predict):
    correct = 0
    for x in range(len(testSet)):
        a1 = predict[x]
        b1 = testSet[x][-1]
        if (b1==a1):
            correct += 1
    return (correct/float(len(testSet))) * 100.0
```

In [32]:

```
# genereta the main method for the KNN algoritm
import timeit
start = timeit.default_timer()
print ('Train set: ' + repr(len(training_set)))
print ('Test set: ' + repr(len(test_set)))

predict=[] # generate predictions
k = 5
for x in range(len(test_set)):
        neighb = getNeighbors(training_set, test_set[x], k)
        result = getResponse(neighb)
        predict.append(result)
        #print('> predicted=' + repr(result) + ', actual=' + repr(test_set[x][-1]))  ---> a good way to
track the process of the algorithm
accuracy = getAccuracy(test_set, predict)
print('Accuracy: ' + repr(accuracy) + '%')
stop = timeit.default_timer()
print (stop - start)
```

```
Train set: 1119
Test set: 480
Accuracy: 56.458333333333336%
85.6365663832321
```

6.(e)

I am using the sklearn confusion matrix here, but I have explained how it works clearly in the coments and in 6.(f).

In [99]:

```
from sklearn.metrics import confusion_matrix

confusion_matrix(test_target, predictions)
# builds a matrix; it's comparing the actual value of the target in the test
# set of the data agianst the predicted value and if they are equal it increments a counter (variable)
# that is then a value in the cells of the confusion matrix --> it's a lot like the accuracy method
# that I implemented for the KNN algorithm;
```

Out[99]:

```
array([[  1,   0,   1,   2,   1,   0],
       [  0,   5,   8,   4,   0,   0],
       [  0,   5, 144,  44,   3,   1],
       [  0,   3,  70, 114,   9,   0],
       [  1,   5,   8,  26,  21,   0],
       [  0,   0,   1,   2,   1,   0]])
```

6.(f)

The **confusion matrix**, also know as the error matrix, is a specific table layout that allows visualization of the performance of an algorithm. Each column of the matrix represents the instances in a predicted class, while each row represents the instances in an actual class. The biggest values in our confusion matrix is 144 and it appears 2 cells in the matrix ([3,3],[4,4]). This means that the prediction values are picking a lot of values as the actual values - all correct guesses are located in the diagonal of the table, so it's easy to visually inspect the table for errors, as they are represented by values outside the diagonal. We can see from the matrix that the system has trouble distinguishing between the fifth value of the actual target and the fourth value ([5, 4] cell in the confuison matrix).