Homework 2 (Due: 10/15/2018)

COEN 281, Fall 2018
Professor Marwah

The objective of this HW is to implement k-NN and cross-validation to find the best value of k for a binary classification task. The task to diagnose breast cancer based on 30 numeric features. However, to keep things simple, we will only use two of those features. The output is binary: 0 benign, 1 malignant. In all there are 569 examples, which we will split into training and test sets. There are no missing values.

In [56]:
```
# load the data set
import numpy as np
import sklearn.datasets
dat = sklearn.datasets.load_breast_cancer()
```

In [57]:
```
# uncomment the following and run it to get a description of the data set
print(dat.DESCR)
```

```
Breast Cancer Wisconsin (Diagnostic) Database
=============================================

Notes
-----
Data Set Characteristics:
    :Number of Instances: 569

    :Number of Attributes: 30 numeric, predictive attributes and the class

    :Attribute Information:
        - radius (mean of distances from center to points on the perimeter)
        - texture (standard deviation of gray-scale values)
        - perimeter
        - area
        - smoothness (local variation in radius lengths)
        - compactness (perimeter^2 / area - 1.0)
        - concavity (severity of concave portions of the contour)
        - concave points (number of concave portions of the contour)
        - symmetry
        - fractal dimension ("coastline approximation" - 1)

        The mean, standard error, and "worst" or largest (mean of the three
        largest values) of these features were computed for each image,
        resulting in 30 features.  For instance, field 3 is Mean Radius, fiel
d
        13 is Radius SE, field 23 is Worst Radius.

        - class:
                - WDBC-Malignant
                - WDBC-Benign

    :Summary Statistics:

    ===================================== ====== ======
                                           Min    Max
    ===================================== ====== ======
    radius (mean):                        6.981  28.11
    texture (mean):                       9.71   39.28
    perimeter (mean):                     43.79  188.5
    area (mean):                          143.5  2501.0
    smoothness (mean):                    0.053  0.163
    compactness (mean):                   0.019  0.345
    concavity (mean):                     0.0    0.427
    concave points (mean):                0.0    0.201
    symmetry (mean):                      0.106  0.304
    fractal dimension (mean):             0.05   0.097
    radius (standard error):              0.112  2.873
    texture (standard error):             0.36   4.885
    perimeter (standard error):           0.757  21.98
    area (standard error):                6.802  542.2
    smoothness (standard error):          0.002  0.031
    compactness (standard error):         0.002  0.135
    concavity (standard error):           0.0    0.396
    concave points (standard error):      0.0    0.053
    symmetry (standard error):            0.008  0.079
    fractal dimension (standard error):   0.001  0.03
```

```
    radius (worst):                       7.93   36.04
    texture (worst):                      12.02  49.54
    perimeter (worst):                    50.41  251.2
    area (worst):                         185.2  4254.0
    smoothness (worst):                   0.071  0.223
    compactness (worst):                  0.027  1.058
    concavity (worst):                    0.0    1.252
    concave points (worst):               0.0    0.291
    symmetry (worst):                     0.156  0.664
    fractal dimension (worst):            0.055  0.208
    ==================================== ====== ======

    :Missing Attribute Values: None

    :Class Distribution: 212 - Malignant, 357 - Benign

    :Creator:  Dr. William H. Wolberg, W. Nick Street, Olvi L. Mangasarian

    :Donor: Nick Street

    :Date: November, 1995

This is a copy of UCI ML Breast Cancer Wisconsin (Diagnostic) datasets.
https://goo.gl/U2Uwz2

Features are computed from a digitized image of a fine needle
aspirate (FNA) of a breast mass.  They describe
characteristics of the cell nuclei present in the image.

Separating plane described above was obtained using
Multisurface Method-Tree (MSM-T) [K. P. Bennett, "Decision Tree
Construction Via Linear Programming." Proceedings of the 4th
Midwest Artificial Intelligence and Cognitive Science Society,
pp. 97-101, 1992], a classification method which uses linear
programming to construct a decision tree.  Relevant features
were selected using an exhaustive search in the space of 1-4
features and 1-3 separating planes.

The actual linear program used to obtain the separating plane
in the 3-dimensional space is that described in:
[K. P. Bennett and O. L. Mangasarian: "Robust Linear
Programming Discrimination of Two Linearly Inseparable Sets",
Optimization Methods and Software 1, 1992, 23-34].

This database is also available through the UW CS ftp server:

ftp ftp.cs.wisc.edu
cd math-prog/cpo-dataset/machine-learn/WDBC/

References
----------
   - W.N. Street, W.H. Wolberg and O.L. Mangasarian. Nuclear feature extracti
on
     for breast tumor diagnosis. IS&T/SPIE 1993 International Symposium on
     Electronic Imaging: Science and Technology, volume 1905, pages 861-870,
     San Jose, CA, 1993.
   - O.L. Mangasarian, W.N. Street and W.H. Wolberg. Breast cancer diagnosis
```

```
and
     prognosis via linear programming. Operations Research, 43(4), pages 570-
577,
     July-August 1995.
   - W.H. Wolberg, W.N. Street, and O.L. Mangasarian. Machine learning techni
ques
     to diagnose breast cancer from fine-needle aspirates. Cancer Letters 77
(1994)
     163-171.
```

In [3]:
```
# dat is a dictionary with the data, let's see what keys it has
dat.keys()
```

Out[3]:
```
dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names'])
```

In [58]:
```
# we will use two features: 'mean area' and 'mean concave points'
ix1 = np.where(dat["feature_names"] == "mean area")[0][0]
ix2 = np.where(dat["feature_names"] == "mean concave points")[0][0]
```

In [59]:
```
X = dat["data"][:,(ix1,ix2)]
Y = dat["target"]

# verify shape of X and Y
print(X.shape, Y.shape)

# stats of the two features
from scipy import stats
st = stats.describe(X)
print("Number of points: %i" % st.nobs)
print("range (min, max), X1: (%.2f, %.2f), X2: (%.2f, %.2f)" % (st.minmax[0][0
], st.minmax[1][0], st.minmax[0][1], st.minmax[1][1]))
print("mean: %.2f, %.2f" % (st.mean[0], st.mean[1]))
print("variance: %.2f, %f" % (st.variance[0], st.variance[1]))

# Given the stats, is it a good idea to normalize the features?
#yes it is as both the features are in differnt scales and as they differ by a
 lot, the feature with the higher magnitude can dominate while calculating the
 euclidian distances.

#
# add code to normalize features
#
print()
print("yes it is as both the features are in differnt scales and as they diffe
r by a lot, the feature with the higher magnitude can dominate while calculati
ng the euclidian distances.")
for i in range(X.shape[1]):
    X[:,i]=(X[:,i]-X[:,i].min())/(X[:,i].max()-X[:,i].min())

#print((np.around(normalized_df,3)))
```

```
(569, 2) (569,)
Number of points: 569
range (min, max), X1: (143.50, 2501.00), X2: (0.00, 0.20)
mean: 654.89, 0.05
variance: 123843.55, 0.001506

yes it is as both the features are in differnt scales and as they differ by a
lot, the feature with the higher magnitude can dominate while calculating the
euclidian distances.
```

In [60]:
```
# split into training set / test set
#
# usually you would do the split randomly; here for deterministic results, we
 assume the data
# points are already shuffled and take the first 70% as training and the rest
 as test
#
nTot = X.shape[0]
nTr = int(nTot*0.7)
nTs = nTot - nTr

Xtr = X[0:nTr,]
Ytr = Y[0:nTr]

Xts = X[nTr:nTot,]
Yts = Y[nTr:nTot,]

# verify shapes
print(Xtr.shape, Ytr.shape, Xts.shape, Yts.shape)
```

```
(398, 2) (398,) (171, 2) (171,)
```

k-NN Implementation

In [61]:
```
# Implement the following functions for k-NN
#

#
# knn_predict(Xtr, Ytr, Xts, k)
#
# input: Xtr - training examples input features, size nXd
#        Ytr - label (can assume to be binary 0/1), size nX1
#        Xts - test examples input features, for which labels (Yts)
#              need to be predicted, size mXd
#        k   - k for the k-NN algo
#
# output: Yts - 0/1 labes for Xts, size mX1
#
#  This function predicts the binary labels for Xts, given the training
#  data Xtr, Ytr and k, using the k-NN algorithm
#
import math
import random
def knn_predict(Xtr, Ytr, Xts, k):
    # compute in two steps

    # step 1: compute dist matrix between Xts and Xtr

    dist = compute_dist_mat(Xtr, Xts)
    dist = np.asarray(dist)
    index_sort = np.argsort(dist)

    # step 2: use the dist matrix and use k-nn to find labels for Xts
    # hint: function numpy.argsort may be useful
    # in case of a tie, pick a class randomly
    # YOUR CODE HERE

    neighbours = []
    for i in range(0, len(dist)):
        neighbours.append([])
        neighbours[i] = index_sort[i][0:k].tolist()
        for j in range(0, len(neighbours[i])):
            neighbours[i][j] = Ytr[neighbours[i][j]]
    prediction = []
    for i in range(0,len(neighbours)):
        prediction.append([])
        count_one = 0
        count_zero = 0
        for j in range(0, k):
            if neighbours[i][j] == 1:
                count_one = count_one + 1
            if neighbours[i][j] == 0:
                count_zero = count_zero + 1
        if count_one > count_zero:
            res = 1
        elif count_zero > count_one:
            res = 0
        else:
            res = random.randint(0,1)
        prediction[i].append(res)
```

```
    pred = np.asarray(prediction)
    return pred
    pass

#  compute_dist_mat(Xts, Xtr)
#
#  input: Xts - test examples, size mXd
#         Xtr - training examples, size nXd
#  output: L2 distance matrix mXn
#
#   if Xts is mXd, and Xtr is nXd, this function returns a matrix of size mXn
 with the L2 distances; the (i,j)
#     entry of the matrix is the L2 distance between ith test and jth training
 example

def compute_dist_mat(Xtr, Xts):
    # use two for loops to compute the matrix
    # YOUR CODE HERE
    dist_matrix = []
    for i in range(0, len(Xts)):
        dist_matrix.append([])
        for j in range(0, len(Xtr)):
            x = Xts[i][0] -  Xtr[j][0];
            y = Xts[i][1] -  Xtr[j][1];
            dist = math.sqrt(x**2 + y**2)
            dist_matrix[i].append(dist)
    return dist_matrix
    pass
```

Problem 1 (30 points): Fill-in code for normalizing features, and the above two functions to implement k-NN.

Problem 2 (20 points): Run your k-NN implementation on the test data set. Use k=5. Compute accuracy, recall and precision of the test data set (do not use python library functions to compute these).

In [62]:
```
# problem 2 solution
predected_label = knn_predict(Xtr, Ytr, Xts, 5)
predected_label = predected_label.flatten()
Yts = np.asarray(Yts)
diff_accuracy = np.subtract(Yts,predected_label)
sum_accuracy = np.add(Yts,predected_label)
unique_sum, counts_sum = np.unique(sum_accuracy, return_counts=True)
unique_diff, counts_diff = np.unique(diff_accuracy, return_counts=True)
true_values = dict(zip(unique_diff, counts_diff))[0]
total = true_values + dict(zip(unique_diff, counts_diff))[-1]+ dict(zip(unique
_diff, counts_diff))[1]
accuracy = true_values/total
precession = dict(zip(unique_sum, counts_sum))[2]/(dict(zip(unique_sum, counts
_sum))[2] + dict(zip(unique_diff, counts_diff))[-1])
recall = dict(zip(unique_sum, counts_sum))[2]/(dict(zip(unique_sum, counts_sum
))[2] + dict(zip(unique_diff, counts_diff))[1])
print("accuracy = {}".format(accuracy))
print("precession = {}".format(precession))
print("recall = {}".format(recall))
```

```
accuracy = 0.9122807017543859
precession = 0.9915966386554622
recall = 0.8939393939393939
```

Cross-Validation

Priblem 3 (30 points): Now we will implement 5-fold cross-validation to find the best value of k. And then using that value of k, re-run k-NN on the test data set. (This is adapted from a past Stanford cs231n assignment)

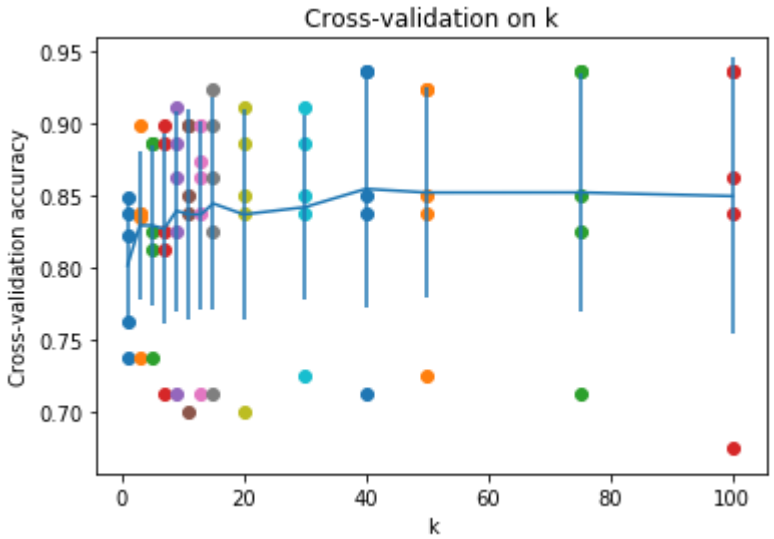
In [63]:
```
num_folds = 5
k_choices = [1, 3, 5, 7, 9, 11, 13, 15, 20, 30, 40, 50, 75, 100]

X_train_folds = []
y_train_folds = []
################################################################################
##
# TODO:
 #
# Split up the training data into folds. After splitting, X_train_folds and
 #
# y_train_folds should each be lists of length num_folds, where
 #
# y_train_folds[i] is the label vector for the points in X_train_folds[i].
 #
# Hint: Look up the numpy array_split function.
 #
################################################################################
##
X_train_folds = np.array(np.array_split(Xtr, num_folds))
y_train_folds = np.array(np.array_split(Ytr, num_folds))
pass
################################################################################
##
#                                 END OF YOUR CODE
 #
################################################################################
##

# A dictionary holding the accuracies for different values of k that we find
# when running cross-validation. After running cross-validation,
# k_to_accuracies[k] should be a list of length num_folds giving the different
# accuracy values that we found when using that value of k.
k_to_accuracies = {}


################################################################################
##
# TODO:
 #
# Perform k-fold cross validation to find the best value of k. For each
 #
# possible value of k, run the k-nearest-neighbor algorithm num_folds times,
 #
# where in each case you use all but one of the folds as training data and the
 #
# last fold as a validation set. Store the accuracies for all fold and all
 #
# values of k in the k_to_accuracies dictionary.
 #
################################################################################
##
for k in k_choices:
    for n in range(num_folds):
        test_num_fold_set = [x for x in range(num_folds) if x != n]
        x_training_dat = np.concatenate(X_train_folds[test_num_fold_set])
```

```
        y_training_dat = np.concatenate(y_train_folds[test_num_fold_set])
        predected_label = knn_predict(x_training_dat, y_training_dat, X_train_
folds[n], k)
        predected_label = predected_label.flatten()
        sum_accuracy = np.add(y_train_folds[n],predected_label)
        unique, counts = np.unique(sum_accuracy, return_counts=True)
        true_values = dict(zip(unique, counts))[0] + dict(zip(unique, counts))
[2]
        total = true_values + dict(zip(unique, counts))[1]
        accuracy = true_values/total
        k_to_accuracies.setdefault(k, []).append(accuracy)
pass
################################################################################
##
#                                 END OF YOUR CODE
 #
################################################################################
##

# Print out the computed accuracies
for k in sorted(k_to_accuracies):
    for accuracy in k_to_accuracies[k]:
        print('k = %d, accuracy = %f' % (k, accuracy))
```

```
k = 1, accuracy = 0.787500
k = 1, accuracy = 0.787500
k = 1, accuracy = 0.875000
k = 1, accuracy = 0.873418
k = 1, accuracy = 0.898734
k = 3, accuracy = 0.787500
k = 3, accuracy = 0.812500
k = 3, accuracy = 0.887500
k = 3, accuracy = 0.911392
k = 3, accuracy = 0.886076
k = 5, accuracy = 0.825000
k = 5, accuracy = 0.837500
k = 5, accuracy = 0.900000
k = 5, accuracy = 0.936709
k = 5, accuracy = 0.911392
k = 7, accuracy = 0.837500
k = 7, accuracy = 0.837500
k = 7, accuracy = 0.875000
k = 7, accuracy = 0.949367
k = 7, accuracy = 0.924051
k = 9, accuracy = 0.850000
k = 9, accuracy = 0.837500
k = 9, accuracy = 0.900000
k = 9, accuracy = 0.924051
k = 9, accuracy = 0.936709
k = 11, accuracy = 0.850000
k = 11, accuracy = 0.825000
k = 11, accuracy = 0.912500
k = 11, accuracy = 0.924051
k = 11, accuracy = 0.936709
k = 13, accuracy = 0.887500
k = 13, accuracy = 0.825000
k = 13, accuracy = 0.912500
k = 13, accuracy = 0.924051
k = 13, accuracy = 0.911392
k = 15, accuracy = 0.875000
k = 15, accuracy = 0.837500
k = 15, accuracy = 0.925000
k = 15, accuracy = 0.924051
k = 15, accuracy = 0.911392
k = 20, accuracy = 0.850000
k = 20, accuracy = 0.850000
k = 20, accuracy = 0.912500
k = 20, accuracy = 0.911392
k = 20, accuracy = 0.936709
k = 30, accuracy = 0.850000
k = 30, accuracy = 0.837500
k = 30, accuracy = 0.937500
k = 30, accuracy = 0.936709
k = 30, accuracy = 0.936709
k = 40, accuracy = 0.825000
k = 40, accuracy = 0.837500
k = 40, accuracy = 0.937500
k = 40, accuracy = 0.924051
k = 40, accuracy = 0.949367
k = 50, accuracy = 0.837500
k = 50, accuracy = 0.850000
```

```
k = 50, accuracy = 0.937500
k = 50, accuracy = 0.924051
k = 50, accuracy = 0.962025
k = 75, accuracy = 0.825000
k = 75, accuracy = 0.850000
k = 75, accuracy = 0.937500
k = 75, accuracy = 0.924051
k = 75, accuracy = 0.962025
k = 100, accuracy = 0.800000
k = 100, accuracy = 0.862500
k = 100, accuracy = 0.925000
k = 100, accuracy = 0.924051
k = 100, accuracy = 0.962025
```

In [18]:
```
# plot the raw observations
import matplotlib.pyplot as plt
%matplotlib inline
for k in k_choices:
    accuracies = k_to_accuracies[k]
    plt.scatter([k] * len(accuracies), accuracies)

# plot the trend line with error bars that correspond to standard deviation
accuracies_mean = np.array([np.mean(v) for k,v in sorted(k_to_accuracies.items
())])
accuracies_std = np.array([np.std(v) for k,v in sorted(k_to_accuracies.items
())])
plt.errorbar(k_choices, accuracies_mean, yerr=accuracies_std)
plt.title('Cross-validation on k')
plt.xlabel('k')
plt.ylabel('Cross-validation accuracy')
plt.show()
```



Problem 4 (20 points): Based on the cross-validation results above, choose the best value for k. Repeat problem 2 with this k (using the entire training data set).

In [64]:
```
mean = []
for k in sorted(k_to_accuracies):
    mean.append(np.mean(k_to_accuracies.get(k)))
mean = np.asarray(mean)
mean = np.argsort(mean)
best_k = k_choices[mean[len(mean) - 1]]
print("Best K value is {}".format(best_k))
predected_label = knn_predict(Xtr, Ytr, Xts, best_k)
predected_label = predected_label.flatten()
Yts = np.asarray(Yts)
diff_accuracy = np.subtract(Yts,predected_label)
sum_accuracy = np.add(Yts,predected_label)
unique_sum, counts_sum = np.unique(sum_accuracy, return_counts=True)
unique_diff, counts_diff = np.unique(diff_accuracy, return_counts=True)
true_values = dict(zip(unique_diff, counts_diff))[0]
total = true_values + dict(zip(unique_diff, counts_diff))[-1]+ dict(zip(unique
_diff, counts_diff))[1]
accuracy = true_values/total
precession = dict(zip(unique_sum, counts_sum))[2]/(dict(zip(unique_sum, counts
_sum))[2] + dict(zip(unique_diff, counts_diff))[-1])
recall = dict(zip(unique_sum, counts_sum))[2]/(dict(zip(unique_sum, counts_sum
))[2] + dict(zip(unique_diff, counts_diff))[1])
print("accuracy = {}".format(accuracy))
print("precession = {}".format(precession))
print("recall = {}".format(recall))
```

```
Best K value is 50
accuracy = 0.9239766081871345
precession = 0.976
recall = 0.9242424242424242
```

Note: these extra credit problems are optional and only increase your score marginally.

Extra Credit Problem 1 (5 points): Plot decision boundaries for the best k, similar to how it is done here: http://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.html (http://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.html)

Extra Credit Problem 2 (5 points): In problem 1 above, re-write the compute_dist_mat() function with no loops. That may seem non-intuitive, but it is possible to compute the L2 distances using matrix operations (matrix multiplication, addition, etc.) without explicitly doing the double for loop. The advantage of using matrix operations is that they are highly optimized and enable "vectorization", and for such computations can give 10-100x speed improvements.