EN 685.621 PA1 - Noboru Hayashi

Problem 1:

1:

```
# Read_csv function:
# with the use of csv package, open & read the csv file
# then store sepal_length, sepal_width, petal_length, petal_width, species info into a
2D list.
def read_csv(file_name):
   with open(file_name) as f:
       reader = csv.reader(f)
       l = [row for row in reader]

   return l
```

2:

```
# Visualize_features function:
# Take sets of 2 features by their indices and the class of each observation,
# visually show the disribution by plotting.
def visualize_features(l, f1=0, f2=1):
   fig, ax = plt.subplots()
   setosa = np.array(l[1:51])
   versicolor = np.array(l[51:101])
   virginica = np.array(l[101:151])

   feat1 = setosa[:, f1]
   feat2 = setosa[:, f2]
   ax.scatter(feat1, feat2, color="b", label='Setosa')

   feat1 = versicolor[:, f1]
   feat2 = versicolor[:, f2]
   ax.scatter(feat1, feat2, color="g", label='Versicolor')

   feat1 = virginica[:, f1]
   feat2 = virginica[:, f2]
   ax.scatter(feat1, feat2, color="y", label='Virginica')

   ax.legend(scatterpoints=1)
   plt.xlabel(l[0][f1])
   plt.ylabel(l[0][f2])
   plt.show()
```

3:

a), c)

```python
def partition(arr_2d, sort_key, low, high):
   i = low - 1
   pivot = arr_2d[high]
   for j in range(low, high):
       if arr_2d[j][sort_key] <= pivot[sort_key]:
           i += 1
           arr_2d[i], arr_2d[j] = arr_2d[j], arr_2d[i]

   arr_2d[i+1], arr_2d[high] = arr_2d[high], arr_2d[i+1]

   return i+1

def quickSort(arr_2d, sort_key, low, high):
   if len(arr_2d) == 1:
       return arr_2d
   if low < high:

       pi = partition(arr_2d, sort_key, low, high)

       quickSort(arr_2d, sort_key, low, pi-1)
       quickSort(arr_2d, sort_key, pi+1, high)
```

b)

With the use of quicksort, sorting iris dataset with a specific sort key requires O(nlogn) time complexity on average and best case scenario. For the worst case, O(n^2)

d)

```python
if __name__ == '__main__':
   l = read_csv('./iris.csv')
   # visualize_features(l, 2, 0)

   data = l[1:]
   n = len(data)
   print("Quick Sort by index 0: sepal length")
   quickSort(data, 0, 0, n-1)
   for i in range (n):
       print(data[i])
   print("Quick Sort by index 1: sepal width")
   quickSort(data, 1, 0, n-1)
   for i in range (n):
       print(data[i])
```

```
print("Quick Sort by index 2: petal length")
quickSort(data, 2, 0, n-1)
for i in range (n):
    print(data[i])
print("Quick Sort by index 3: petal width")
quickSort(data, 3, 0, n-1)
for i in range (n):
    print(data[i])
```

With the output below, the 3rd and 4th features seem to be able to separate iris classes:

Quick Sort by index 0: sepal length
...
['5.6', '2.5', '3.9', '1.1', 'versicolor']
['5.6', '2.9', '3.6', '1.3', 'versicolor']
['5.6', '2.8', '4.9', '2.0', 'virginica']
['5.7', '2.6', '3.5', '1.0', 'versicolor']
['5.7', '4.4', '1.5', '0.4', 'setosa']
['5.7', '2.8', '4.5', '1.3', 'versicolor']
['5.7', '3.0', '4.2', '1.2', 'versicolor']
['5.7', '2.9', '4.2', '1.3', 'versicolor']
['5.7', '2.8', '4.1', '1.3', 'versicolor']
...
Quick Sort by index 1: sepal width
...
['5.0', '2.3', '3.3', '1.0', 'versicolor']
['5.5', '2.3', '4.0', '1.3', 'versicolor']
['5.9', '3.0', '5.1', '1.8', 'virginica']
['5.0', '3.0', '1.6', '0.2', 'setosa']
['5.4', '3.0', '4.5', '1.5', 'versicolor']
…
Quick Sort by index 2: petal length
['4.6', '3.6', '1.0', '0.2', 'setosa']
['4.3', '3.0', '1.1', '0.1', 'setosa']
['5.0', '3.2', '1.2', '0.2', 'setosa']
['5.8', '4.0', '1.2', '0.2', 'setosa']
['4.5', '2.3', '1.3', '0.3', 'setosa']
['4.7', '3.2', '1.3', '0.2', 'setosa']
['4.4', '3.2', '1.3', '0.2', 'setosa']
['5.0', '3.5', '1.3', '0.3', 'setosa']
['5.5', '3.5', '1.3', '0.2', 'setosa']
...
Quick Sort by index 3: petal width
['4.3', '3.0', '1.1', '0.1', 'setosa']
['4.8', '3.0', '1.4', '0.1', 'setosa']
['4.9', '3.1', '1.5', '0.1', 'setosa']
['4.9', '3.1', '1.5', '0.1', 'setosa']
['4.9', '3.1', '1.5', '0.1', 'setosa']
['5.2', '4.1', '1.5', '0.1', 'setosa']
['5.5', '3.5', '1.3', '0.2', 'setosa']
…

4:a) & c)

```
def mahalanobis_method(class_data):
    x_minus_mu = class_data - np.mean(class_data, axis=0)
    cov = np.cov(x_minus_mu.T)
    inv_covmat = np.linalg.inv(cov)
    left_term = np.dot(x_minus_mu, inv_covmat)
    mahal = np.dot(left_term, x_minus_mu.T)
    md = np.sqrt(mahal.diagonal())

    outlier = []
    C = np.sqrt(chi2.ppf((1-0.05), df=class_data.shape[1]))     #degrees of freedom = 
number of variables
    for index, value in enumerate(md):
        if value > C:
            outlier.append(index)
        else:
            continue
    return outlier, md
```

b)

The time complexity of calculating mahalanobis distance: assume N records with 4 features

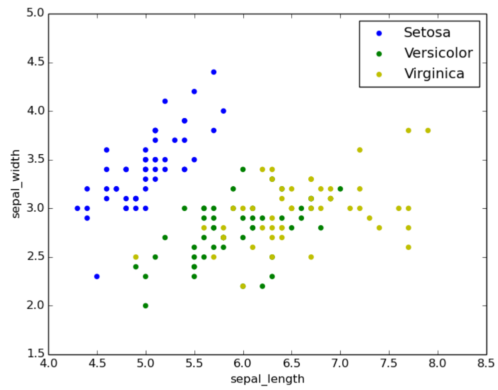
1. Calculate means of each class: O(N*n) = O(N)
2. Get the difference between class data and mean for each class: O(N*n) = O(N)
3. Cov: since cov matrix is calculated by XT*X, with X ~ Nxn matrix, the algorithm operates nxN matrix * Nxn matrix, the running time is O(n*N*n) = O(Nn^2) = O(N)
4. Inverse of cov: remap each element from cov matrix (nxn), so O(n^2) = O(1)
5. Dot products: (Nxn * nxn) * n*N, so O(Nn^2) + O(nN^2) = O(N) + O(N^2)
6. ⇒ Total running time is O(N^2)

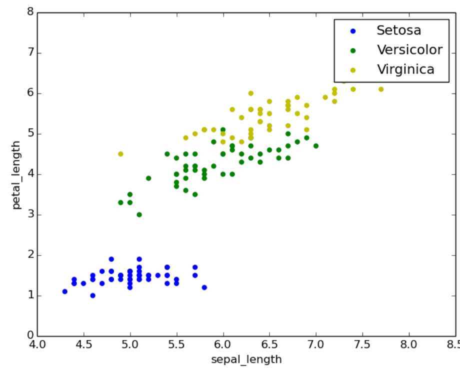
d) e)

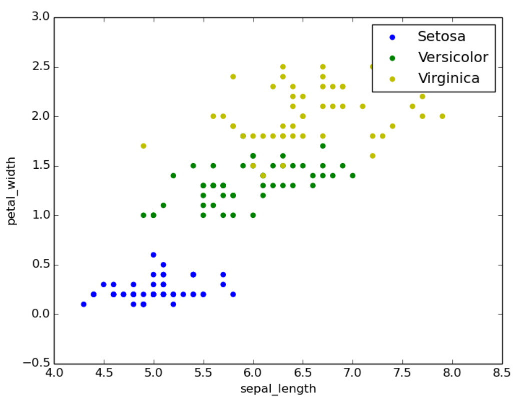
```
    setosa = np.array(l[1:51])[:, :4].astype(np.float)
    versicolor = np.array(l[51:101])[:, :4].astype(np.float)
    virginica = np.array(l[101:151])[:, :4].astype(np.float)

    outliers_mahal, md = mahalanobis_method(setosa)
    print(outliers_mahal)

    outliers_mahal, md = mahalanobis_method(versicolor)
    print(outliers_mahal)

    outliers_mahal, md = mahalanobis_method(virginica)
    print(outliers_mahal)
```

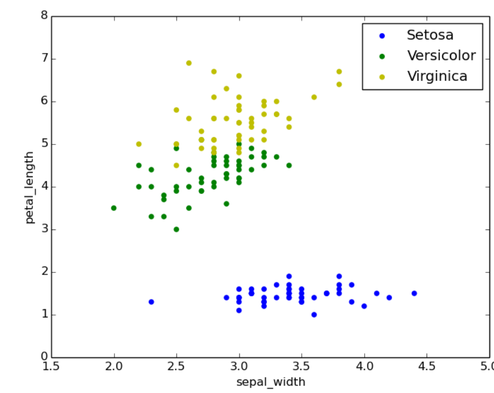
As the outputs below, there are 5, 2 and 2 outliers for each class with the 95% CI
$python p1.py
[14, 22, 24, 41, 43]
[18, 48]
[18, 31]

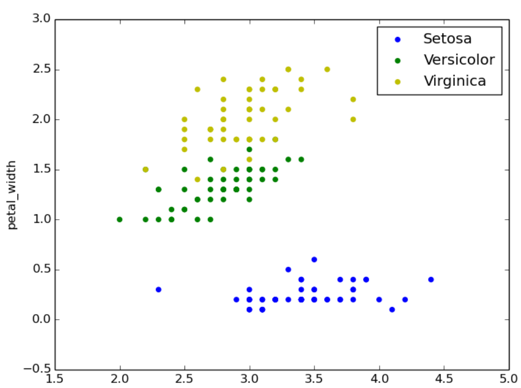
Plots:

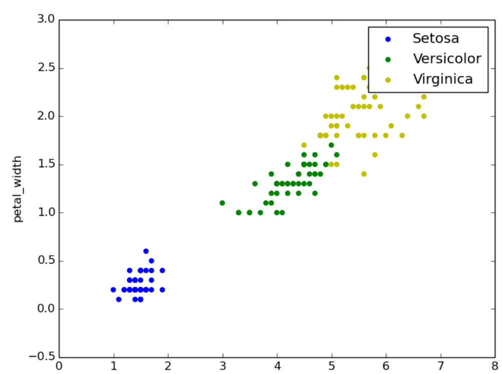












5

a) c)

```
def feature_ranking(l):
    setosa = np.array(l[1:51])[:, :4].astype(np.float)
    versicolor = np.array(l[51:101])[:, :4].astype(np.float)
    virginica = np.array(l[101:151])[:, :4].astype(np.float)
    success_rate = [0,0,0,0]
    S1 = np.std(setosa,axis=0)
    M1 = np.mean(setosa,axis=0)
    S2 = np.std(versicolor,axis=0)
    M2 = np.mean(versicolor,axis=0)
    S3 = np.std(virginica,axis=0)
    M3 = np.mean(virginica,axis=0)

    for i in range(4):
        success = 0
        for j in range(50):
            d1 = (setosa[j][i] - M1[i]) **2 * S1[i]
            d2 = (setosa[j][i] - M2[i]) **2 * S2[i]
            d3 = (setosa[j][i] - M3[i]) **2 * S3[i]

            if d1 < d2 and d1 < d3:
                success += 1

            d1 = (versicolor[j][i] - M1[i]) **2 * S1[i]
            d2 = (versicolor[j][i] - M2[i]) **2 * S2[i]
            d3 = (versicolor[j][i] - M3[i]) **2 * S3[i]

            if d2 < d1 and d2 < d3:
                success += 1

            d1 = (virginica[j][i] - M1[i]) **2 * S1[i]
            d2 = (virginica[j][i] - M2[i]) **2 * S2[i]
            d3 = (virginica[j][i] - M3[i]) **2 * S3[i]

            if d3 < d1 and d3 < d2:
                success += 1

        success_rate[i] = success / 150.0

    return success_rate
```

b)

Total running time of feature ranking, assume N records with n=4 features

1. Calculate mean and std for each class: O(N*n) = O(N)
2. Calculate the distance of specific features and the means: O(1) for each record & feature pair, so O(N*n) = O(N)
3. Compare the distances d1, d2, d3 for each observation: O(1)

⇒ total running time: O(N)

d) e)

```
if __name__ == '__main__':
    l = read_csv('./iris.csv')

    print(feature_ranking(l))
```

Output:

[0.7266666666666667, 0.5466666666666666, 0.94, 0.96]

From the output, the third and fourth features are quite high in the feature ranking, so with the use of these features, it seems to be possible to separate 3 classes of iris data.

Problem 2:
b) :
```python
def checkWinPos(place):
    # 0-1-2
    if place[0] == place[1] and place[0] != 0 and place[2] == 0: return 2
    if place[0] == place[2] and place[0] != 0 and place[1] == 0: return 1
    if place[1] == place[2] and place[1] != 0 and place[0] == 0: return 0
    # 0-3-6
    if place[0] == place[3] and place[0] != 0 and place[6] == 0: return 6
    if place[0] == place[6] and place[0] != 0 and place[3] == 0: return 3
    if place[3] == place[6] and place[3] != 0 and place[0] == 0: return 0
    # 0-4-9
    if place[0] == place[4] and place[0] != 0 and place[8] == 0: return 8
    if place[0] == place[8] and place[0] != 0 and place[4] == 0: return 4
    if place[4] == place[8] and place[4] != 0 and place[0] == 0: return 0
    # 3-4-5
    if place[3] == place[4] and place[3] != 0 and place[5] == 0: return 5
    if place[3] == place[5] and place[3] != 0 and place[4] == 0: return 4
    if place[4] == place[5] and place[4] != 0 and place[3] == 0: return 3
    # 6-7-8
    if place[6] == place[7] and place[6] != 0 and place[8] == 0: return 8
    if place[6] == place[8] and place[6] != 0 and place[7] == 0: return 7
    if place[7] == place[8] and place[7] != 0 and place[6] == 0: return 6
    # 1-4-7
    if place[1] == place[4] and place[1] != 0 and place[7] == 0: return 7
    if place[1] == place[7] and place[1] != 0 and place[4] == 0: return 4
    if place[4] == place[7] and place[4] != 0 and place[1] == 0: return 1
    # 2-5-8
    if place[2] == place[5] and place[2] != 0 and place[8] == 0: return 8
    if place[2] == place[8] and place[2] != 0 and place[5] == 0: return 5
    if place[5] == place[8] and place[5] != 0 and place[2] == 0: return 2
    return None
```

c) Running time of checkWinPos is O(1) since it scan through the 3x3 fixed TicTacToe gamestate