Lab 3

Reza Shisheie
2708062

October 29, 2018

Part 1: Data processing methods

Based on lab 1 I pick the following attributes for test:

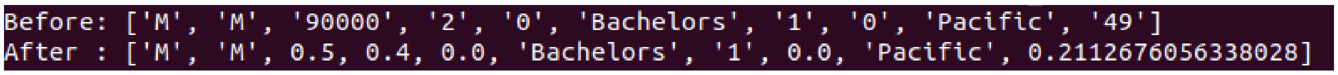
1. Marital status:
2. Gender:
3. Yearly income:
4. Total Children:
5. Number of children at home:
6. English education:
7. House owner flag:
8. number of cars owned:
9. Region:
10. Age:

For the rest of the program I am using the same code used for Lab1. Here are the preprocessing techniques:

Normalization :
In this section the non-nominal target data is normalized using min max method. Here is a list of attributes which were normalized:

1. YearlyIncome
2. TotalChildren
3. NumberChildrenAtHome
4. NumberCarsOwned
5. Age

a function in python named “normalize” is defined for this purpose. Here is a sample data before and after normalization:



```
Before: ['M', 'M', '90000', '2', '0', 'Bachelors', '1', '0', 'Pacific', '49']
After : ['M', 'M', 0.5, 0.4, 0.0, 'Bachelors', '1', 0.0, 'Pacific', 0.2112676056338028]
```

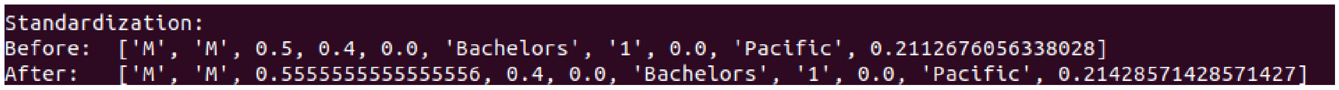
As you see only numeric items like salary and number of children are normalized. For instance the value 9000 in the second column which is salary is normalized to 0.5.

Standardization:
The normalized data is standardized using z-score method. A function called “z-score” is defined for this purpose. The following set is selected for standardization followed by the number of bins assigned to each one:

1. YearlyIncome	10
2. TotalChildren	6
3. NumberChildrenAtHome	6
4. NumberCarsOwned	5
5. Age	15

Number of bins for each attribute is selected experimentally and were increased based on the type of data and consistency of result after changing bins. For instance the maximum value for the number of children was 5 and min 0 so I found 6 bins as the best option.

Here is data before and after standardization:

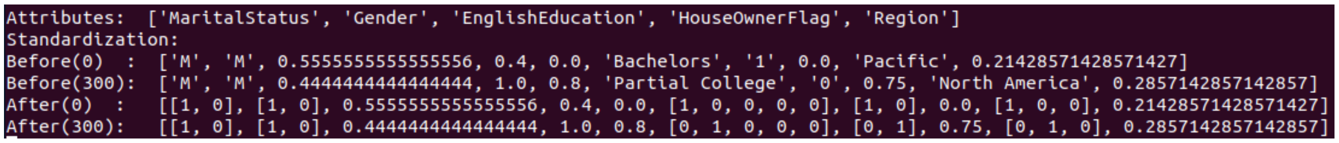


```
Standardization:
Before:  ['M', 'M', 0.5, 0.4, 0.0, 'Bachelors', '1', 0.0, 'Pacific', 0.2112676056338028]
After:   ['M', 'M', 0.5555555555555556, 0.4, 0.0, 'Bachelors', '1', 0.0, 'Pacific', 0.21428571428571427]
```

Binarization
in this section all the nominal values were binerized. Here is the list of nominal values which were binerized:

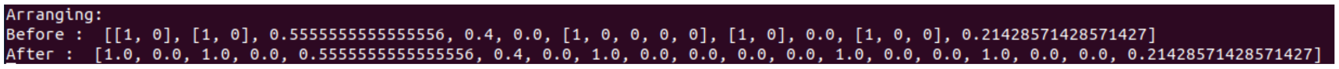
1. Marital status
2. Gender
3. English education
4. House owner flag
5. Region:

Here s data at 0 and 300 before and after binerization:



```
Attributes:  ['MaritalStatus', 'Gender', 'EnglishEducation', 'HouseOwnerFlag', 'Region']
Standardization:
Before(0)  :  ['M', 'M', 0.5555555555555556, 0.4, 0.0, 'Bachelors', '1', 0.0, 'Pacific', 0.21428571428571427]
Before(300):  ['M', 'M', 0.4444444444444444, 1.0, 0.8, 'Partial College', '0', 0.75, 'North America', 0.2857142857142857]
After(0)  :   [[1, 0], [1, 0], 0.5555555555555556, 0.4, 0.0, [1, 0, 0, 0, 0], [1, 0], 0.0, [1, 0, 0], 0.21428571428571427]
After(300):   [[1, 0], [1, 0], 0.4444444444444444, 1.0, 0.8, [0, 1, 0, 0, 0], [0, 1], 0.75, [0, 1, 0], 0.2857142857142857]
```

Arranging in an array
once all data is gathered, it is arranged in an array. All 0s and 1s and all other data is put into one single array. Here is one data sample before and after arranging:



```
Arranging:
Before :  [[1, 0], [1, 0], 0.5555555555555556, 0.4, 0.0, [1, 0, 0, 0, 0], [1, 0], 0.0, [1, 0, 0], 0.21428571428571427]
After :  [1.0, 0.0, 1.0, 0.0, 0.5555555555555556, 0.4, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.21428571428571427]
```

Test sets:
I make two set of tests:
1. The first set has all the methods mentioned above
2. The second one only has binerization and does not have standardization and normalization

Part 2: Classifier selection:

Two different classifiers are selected:

- *k* nearest neighbor (*k*NN)
- Support Vector Machine (SVM)

Test data is divided into two groups. The first 2/3 is for training and the 1/3 rest is for testing.

1. *k*NN Method

The range of neighbors for *k*NN ranges from 1 to 20 and based on the best accuracy, the optimum number of neighbors is selected.The NearestNeighbors library of scikit is used.

For each test set neighbors (2~20) are found by the algorithm and the weight is the inverse of distance from neighbors to the test set. Then the goal is calculated by averaging the class f neighbors with weights:

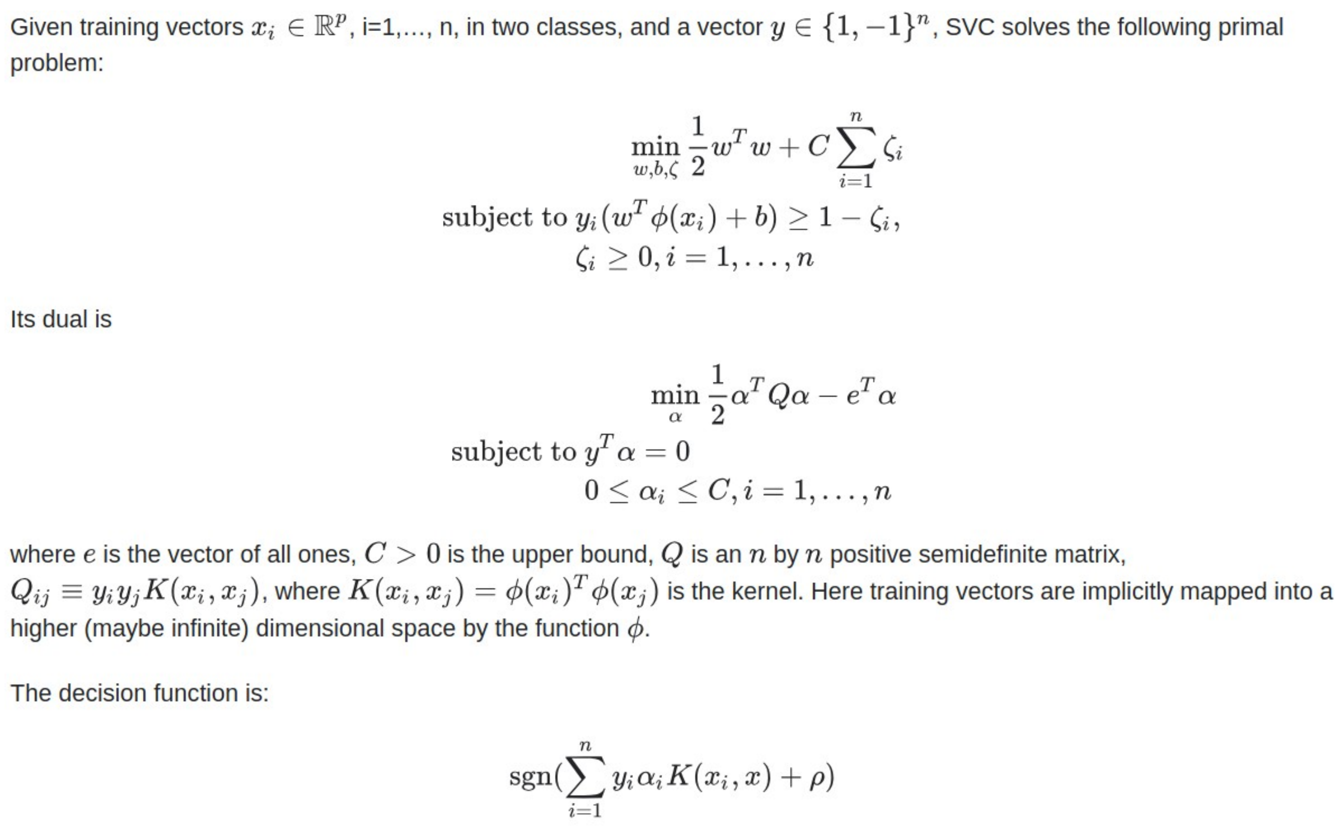
distance = sum(weight*testClass)/sum(weights)
weight = 1/testDistance

2. SVM Method

For SVM has several different kernels and methods were tested which is listed below:

- 4 kernel types including: linear, polynomial, sigmod, and rbf
- 3 implementation: SVC, NuSVC, SVR

<u>**SVC**</u>



Given training vectors $x_i \in \mathbb{R}^p$, i=1,…, n, in two classes, and a vector $y \in \{1, -1\}^n$, SVC solves the following primal problem:

$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i$$
$$\text{subject to } y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i,$$
$$\zeta_i \geq 0, i = 1, \ldots, n$$

Its dual is

$$\min_{\alpha} \frac{1}{2} \alpha^T Q \alpha - e^T \alpha$$
$$\text{subject to } y^T \alpha = 0$$
$$0 \leq \alpha_i \leq C, i = 1, \ldots, n$$

where e is the vector of all ones, $C > 0$ is the upper bound, Q is an n by n positive semidefinite matrix, $Q_{ij} \equiv y_i y_j K(x_i, x_j)$, where $K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ is the kernel. Here training vectors are implicitly mapped into a higher (maybe infinite) dimensional space by the function ϕ.

The decision function is:

$$\operatorname{sgn}(\sum_{i=1}^{n} y_i \alpha_i K(x_i, x) + \rho)$$

NuSVC:

We introduce a new parameter ν which controls the number of support vectors and training errors. The parameter $\nu \in (0, 1]$ is an upper bound on the fraction of training errors and a lower bound of the fraction of support vectors.

It can be shown that the ν-SVC formulation is a reparameterization of the C-SVC and therefore mathematically equivalent.

SVR:

Given training vectors $x_i \in \mathbb{R}^p$, i=1,…, n, and a vector $y \in \mathbb{R}^n$ ε-SVR solves the following primal problem:

$$\begin{aligned} \min_{w,b,\zeta,\zeta^*} \frac{1}{2} w^T w + C \sum_{i=1}^{n} (\zeta_i + \zeta_i^*) \\ \text{subject to } y_i - w^T \phi(x_i) - b \leq \varepsilon + \zeta_i, \\ w^T \phi(x_i) + b - y_i \leq \varepsilon + \zeta_i^*, \\ \zeta_i, \zeta_i^* \geq 0, i = 1, \ldots, n \end{aligned}$$

Its dual is

$$\begin{aligned} \min_{\alpha,\alpha^*} \frac{1}{2} (\alpha - \alpha^*)^T Q (\alpha - \alpha^*) + \varepsilon e^T (\alpha + \alpha^*) - y^T (\alpha - \alpha^*) \\ \text{subject to } e^T (\alpha - \alpha^*) = 0 \\ 0 \leq \alpha_i, \alpha_i^* \leq C, i = 1, \ldots, n \end{aligned}$$

where e is the vector of all ones, $C > 0$ is the upper bound, Q is an n by n positive semidefinite matrix, $Q_{ij} \equiv K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ is the kernel. Here training vectors are implicitly mapped into a higher (maybe infinite) dimensional space by the function ϕ.

The decision function is:

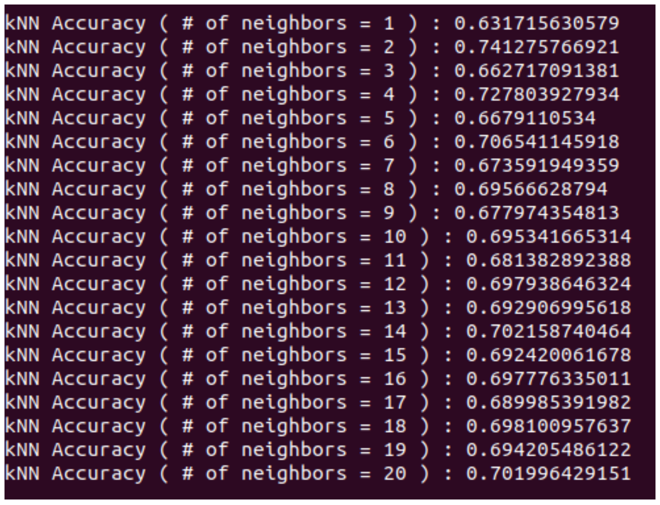
$$\sum_{i=1}^{n} (\alpha_i - \alpha_i^*) K(x_i, x) + \rho$$

Here is a list of all kernel functions:

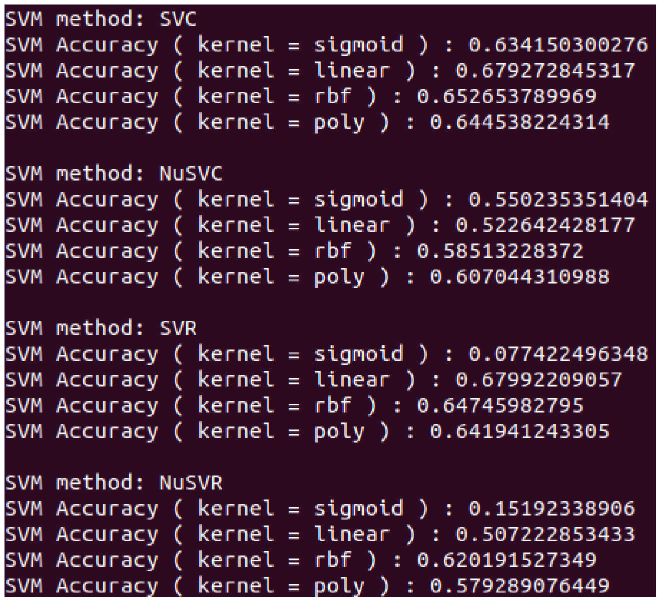
- linear: $\langle x, x' \rangle$.
- polynomial: $(\gamma \langle x, x' \rangle + r)^d$. d is specified by keyword `degree`, r by `coef0`.
- rbf: $\exp(-\gamma \|x - x'\|^2)$. γ is specified by keyword `gamma`, must be greater than 0.
- sigmoid ($\tanh(\gamma \langle x, x' \rangle + r)$), where r is specified by `coef0`.

Part 3: Result and Analysis

Here are the result for kNN and SVM

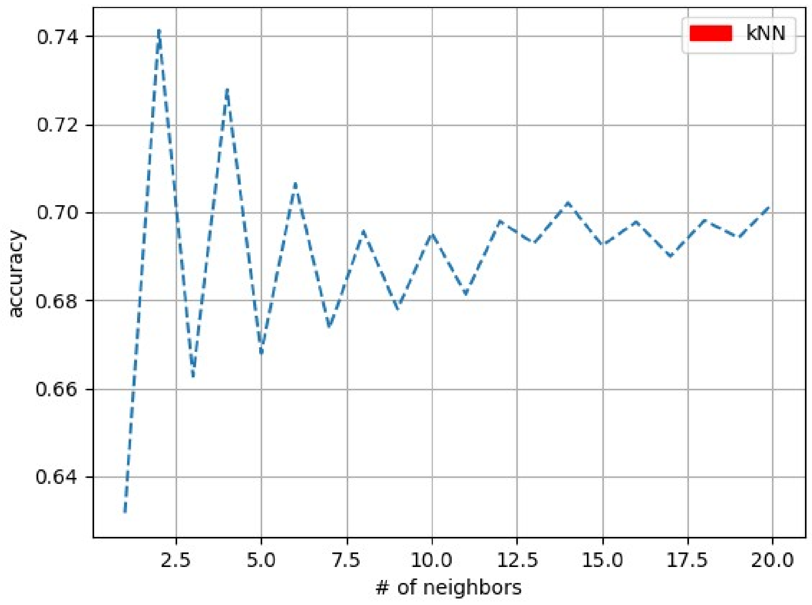


```
kNN Accuracy ( # of neighbors = 1 ) : 0.631715630579
kNN Accuracy ( # of neighbors = 2 ) : 0.741275766921
kNN Accuracy ( # of neighbors = 3 ) : 0.662717091381
kNN Accuracy ( # of neighbors = 4 ) : 0.727803927934
kNN Accuracy ( # of neighbors = 5 ) : 0.6679110534
kNN Accuracy ( # of neighbors = 6 ) : 0.706541145918
kNN Accuracy ( # of neighbors = 7 ) : 0.673591949359
kNN Accuracy ( # of neighbors = 8 ) : 0.69566628794
kNN Accuracy ( # of neighbors = 9 ) : 0.677974354813
kNN Accuracy ( # of neighbors = 10 ) : 0.695341665314
kNN Accuracy ( # of neighbors = 11 ) : 0.681382892388
kNN Accuracy ( # of neighbors = 12 ) : 0.697938646324
kNN Accuracy ( # of neighbors = 13 ) : 0.692906995618
kNN Accuracy ( # of neighbors = 14 ) : 0.702158740464
kNN Accuracy ( # of neighbors = 15 ) : 0.692420061678
kNN Accuracy ( # of neighbors = 16 ) : 0.697776335011
kNN Accuracy ( # of neighbors = 17 ) : 0.689985391982
kNN Accuracy ( # of neighbors = 18 ) : 0.698100957637
kNN Accuracy ( # of neighbors = 19 ) : 0.694205486122
kNN Accuracy ( # of neighbors = 20 ) : 0.701996429151
```



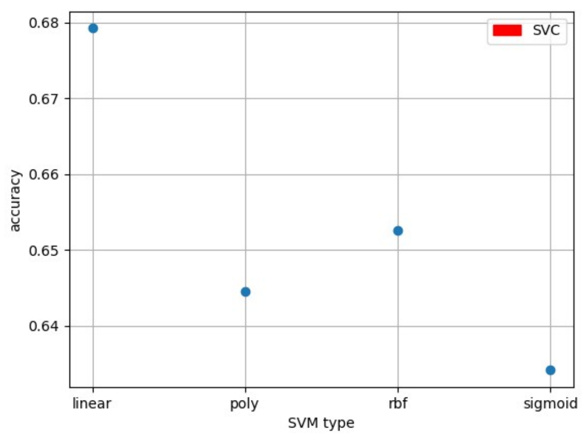
```
SVM method: SVC
SVM Accuracy ( kernel = sigmoid ) : 0.634150300276
SVM Accuracy ( kernel = linear ) : 0.679272845317
SVM Accuracy ( kernel = rbf ) : 0.652653789969
SVM Accuracy ( kernel = poly ) : 0.644538224314

SVM method: NuSVC
SVM Accuracy ( kernel = sigmoid ) : 0.550235351404
SVM Accuracy ( kernel = linear ) : 0.522642428177
SVM Accuracy ( kernel = rbf ) : 0.58513228372
SVM Accuracy ( kernel = poly ) : 0.607044310988

SVM method: SVR
SVM Accuracy ( kernel = sigmoid ) : 0.077422496348
SVM Accuracy ( kernel = linear ) : 0.67992209057
SVM Accuracy ( kernel = rbf ) : 0.64745982795
SVM Accuracy ( kernel = poly ) : 0.641941243305

SVM method: NuSVR
SVM Accuracy ( kernel = sigmoid ) : 0.15192338906
SVM Accuracy ( kernel = linear ) : 0.507222853433
SVM Accuracy ( kernel = rbf ) : 0.620191527349
SVM Accuracy ( kernel = poly ) : 0.579289076449
```

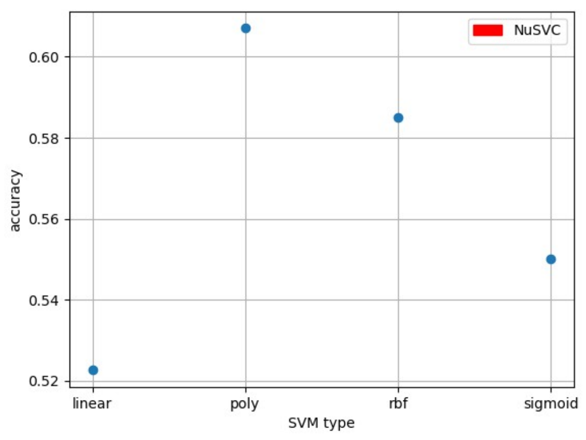
The best result for *k*NN comes from 2 neighbors and as more and more neighbors are added it converges to %70 accuracy. The best accuracy is %74. The less number of neighbors you use, the more susceptible it is to noisy data or misreading one. The more neighbors you choose the more robust it is.

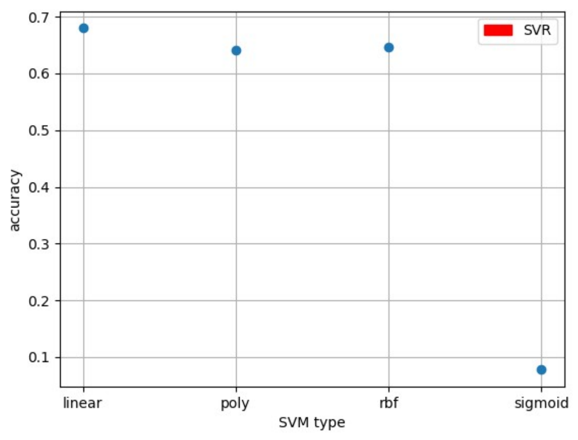


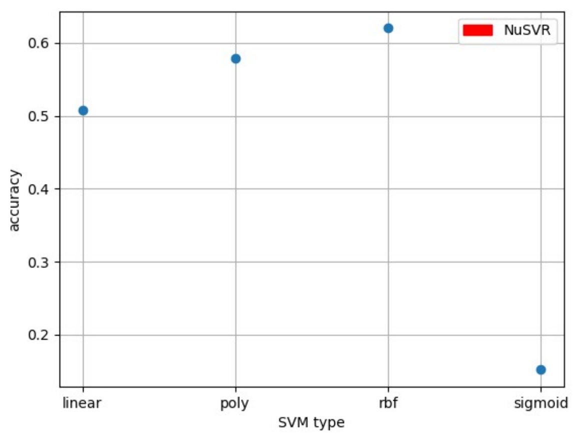
The best result for SVM comes from linear kernel of SVC and SVR. All nonlinear kernel such as rbf, polynomial, and sigmoid yield worse results. The justification for it is that the data is probably linear that nonlinear so a nonlinear kernel does not yield good.

The best result comes from linear SVC (%68) and linear SVR (%68).









Part 4: Code

```
# This section of code is just dedicated to kNN and SVM. The previous pre-processing was already
explained in lab 1

####################
'''
PART 4.0: arranging in an array and spliting test and train datasets

'''

print 'Arranging: '
print 'Before : ', data[0]

divide = float(2/3)
trainingMin = 0
trainingMax = int(int(len(data))*2/3)
testMin = trainingMax + 1
testMax = int(len(data))

#print trainingMin, trainingMax, testMin ,testMax

trainSet = []
trainClass = []
for i in range (trainingMin,trainingMax):
    dataTemp = []
    for j in range(0,len(data[0])):
        if isinstance(data[i][j], list):
            for w in range(0,len(data[i][j])):
                #dataTemp.append(float(data[i][j][w])/2)
                dataTemp.append(float(data[i][j][w]))
        else:
            dataTemp.append(float(data[i][j]))
    trainSet.append(dataTemp)
    trainClass.append(classList[i])


testSet = []
testClass = []
for i in range (testMin,testMax):
    dataTemp = []
    for j in range(0,len(data[0])):
        if isinstance(data[i][j], list):
            for w in range(0,len(data[i][j])):
                #dataTemp.append(float(data[i][j][w])/2)
                dataTemp.append(float(data[i][j][w]))
        else:
            dataTemp.append(data[i][j])
    testSet.append(dataTemp)
    testClass.append(classList[i])


print 'After : ', trainSet[0]
#raw_input()


####################
'''
PART 4.1: kNN

'''

print ''
#print 'kNN'

from sklearn.neighbors import NearestNeighbors

kNNx = []
kNNy = []
kNNMin = 1
kNNMax = 21
#kNNMax = 2
for NN in range(kNNMin,kNNMax):

    neigh = NearestNeighbors(n_neighbors=NN)
    neigh.fit(trainSet)

    countTrue = 0
    CountFalse = 0
    for xx in range (testMin,testMax):

        targetCell = xx - testMin
        result = neigh.kneighbors([testSet[int(targetCell)]])

        #print result
        #raw_input()

        sumTemp = 0.0
        den = 0.0
        for i in range(0,len(result[0][0])):

            if float(result[0][0][i]) == 0.0:
                w = 1000.0
            else:
                w = 1.0/float(result[0][0][i])

            d = float(trainClass [result[1][0][i]])
            sumTemp = w*d + sumTemp
            den = w + den

        #decision = float(sumTemp)/float(len(result[0][0]))
        decision = float(sumTemp)/float(den)


        if abs(int(testClass[targetCell])-decision)>0.5:
            CountFalse = CountFalse + 1

        else:
            countTrue = countTrue + 1


    accuracy = (float(countTrue)/float(countTrue+CountFalse))
    print 'kNN Accuracy', '( # of neighbors =', NN, ') :' , accuracy

    kNNx.append(NN)
    kNNy.append(accuracy)


import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

plt.figure(1)
#plt.subplot(211)
#grid(color='r', linestyle='-', linewidth=2)
plt.plot((kNNx), kNNy, '--')
plt.xlabel('# of neighbors')
plt.ylabel('accuracy')
red_patch = mpatches.Patch(color='red', label='kNN')
plt.legend(handles=[red_patch])
plt.grid(True)

####################
'''
PART 4.2: SVM

'''

#print 'SVM'
from sklearn import svm
```

```
SVMType = ['sigmoid','linear','rbf', 'poly']


SVMx = []
SVMy = []
print ''
print 'SVM method: SVC'
for SVMLoop in range (0,len(SVMType)):

    #clf = svm.SVR(kernel=SVMType[SVMLoop])
    clf = svm.SVC(kernel=SVMType[SVMLoop])
    #clf = svm.NuSVC(kernel=SVMType[SVMLoop])
    #clf = svm.NuSVR(kernel=SVMType[SVMLoop])
    clf.fit(trainSet, trainClass)

    countTrue = 0
    CountFalse = 0
    for xx in range (testMin,testMax):

        targetCell = xx - testMin
        result = clf.predict([testSet[targetCell]])

        if abs(float(result[0])-float(testClass[targetCell]))>0.5:
            CountFalse = CountFalse + 1

        else:
            countTrue = countTrue + 1


    accuracy = (float(countTrue)/float(countTrue+CountFalse ))
    print 'SVM Accuracy', '( kernel =',SVMType[SVMLoop],') :', accuracy
    SVMx.append(SVMType[SVMLoop])
    SVMy.append(accuracy)

plt.figure(2)
#plt.subplot(212)
plt.plot(SVMx, SVMy,'o')
plt.xlabel('SVM type')
plt.ylabel('accuracy')
red_patch = mpatches.Patch(color='red', label='SVC')
plt.legend(handles=[red_patch])
plt.grid(True)




SVMx = []
```

```
SVMy = []
print ''
print 'SVM method: NuSVC'
for SVMLoop in range (0,len(SVMType)):

    #clf = svm.SVR(kernel=SVMType[SVMLoop])
    #clf = svm.SVC(kernel=SVMType[SVMLoop])
    clf = svm.NuSVC(kernel=SVMType[SVMLoop])
    #clf = svm.NuSVR(kernel=SVMType[SVMLoop])
    clf.fit(trainSet, trainClass)

    countTrue = 0
    CountFalse = 0
    for xx in range (testMin,testMax):

        targetCell = xx - testMin
        result = clf.predict([testSet[targetCell]])

        if abs(float(result[0])-float(testClass[targetCell]))>0.5:
            CountFalse = CountFalse + 1

        else:
            countTrue = countTrue + 1


    accuracy = (float(countTrue)/float(countTrue+CountFalse ))
    print 'SVM Accuracy', '( kernel =',SVMType[SVMLoop],') :', accuracy
    SVMx.append(SVMType[SVMLoop])
    SVMy.append(accuracy)


plt.figure(3)
plt.plot(SVMx, SVMy,'o')
plt.xlabel('SVM type')
plt.ylabel('accuracy')
red_patch = mpatches.Patch(color='red', label='NuSVC')
plt.legend(handles=[red_patch])
plt.grid(True)





SVMx = []
SVMy = []
print ''
print 'SVM method: SVR'
for SVMLoop in range (0,len(SVMType)):

    clf = svm.SVR(kernel=SVMType[SVMLoop])
    #clf = svm.SVC(kernel=SVMType[SVMLoop])
    #clf = svm.NuSVC(kernel=SVMType[SVMLoop])
    #clf = svm.NuSVR(kernel=SVMType[SVMLoop])
    clf.fit(trainSet, trainClass)

    countTrue = 0
    CountFalse = 0
    for xx in range (testMin,testMax):

        targetCell = xx - testMin
        result = clf.predict([testSet[targetCell]])

        if abs(float(result[0])-float(testClass[targetCell]))>0.5:
            CountFalse = CountFalse + 1

        else:
            countTrue = countTrue + 1


    accuracy = (float(countTrue)/float(countTrue+CountFalse ))
    print 'SVM Accuracy', '( kernel =',SVMType[SVMLoop],') :', accuracy
    SVMx.append(SVMType[SVMLoop])
    SVMy.append(accuracy)


plt.figure(4)
plt.plot(SVMx, SVMy,'o')
plt.xlabel('SVM type')
plt.ylabel('accuracy')
red_patch = mpatches.Patch(color='red', label='SVR')
plt.legend(handles=[red_patch])
plt.grid(True)



SVMx = []
SVMy = []
print ''
print 'SVM method: NuSVR'
for SVMLoop in range (0,len(SVMType)):

    #clf = svm.SVR(kernel=SVMType[SVMLoop])
    #clf = svm.SVC(kernel=SVMType[SVMLoop])
    #clf = svm.NuSVC(kernel=SVMType[SVMLoop])
    clf = svm.NuSVR(kernel=SVMType[SVMLoop])
    clf.fit(trainSet, trainClass)

    countTrue = 0
    CountFalse = 0
    for xx in range (testMin,testMax):

        targetCell = xx - testMin
        result = clf.predict([testSet[targetCell]])

        if abs(float(result[0])-float(testClass[targetCell]))>0.5:
            CountFalse = CountFalse + 1

        else:
            countTrue = countTrue + 1


    accuracy = (float(countTrue)/float(countTrue+CountFalse ))
    print 'SVM Accuracy', '( kernel =',SVMType[SVMLoop],') :', accuracy
    SVMx.append(SVMType[SVMLoop])
    SVMy.append(accuracy)


plt.figure(5)
plt.plot(SVMx, SVMy,'o')
plt.xlabel('SVM type')
plt.ylabel('accuracy')
red_patch = mpatches.Patch(color='red', label='NuSVR')
plt.legend(handles=[red_patch])
plt.grid(True)


plt.show()
```