SMAI Assignment 3 Report
201401074

Problem 1

Non-Linear (Kernelised) version of Fisher's Linear Discriminant Analysis

LDA can be extended to non linear mappings. The data given as the ℓ points x_i can be mapped to a new feature space F, via some function ϕ. In the new feature space, the function that needs to be maximized is :

$$J(w) = w^T S_B^\phi w \,/\, w^T S_W w$$

where

$$S_B^\phi = (m_2^\phi - m_1^\phi)(m_2^\phi - m_1^\phi)^T$$

$$S_W^\phi = \sum_{i=1,2} \sum_{n=1}^{l_i} (\phi(x_n^i) - m_i^\phi)(\phi(x_n^i) - m_i^\phi)^T$$

where $m_i^\phi = \frac{1}{l_i} \sum_{j=1}^{l_i} \phi(x_j^i)$.

$w \in F$

$$w = \sum_{i=1}^{l} \alpha_i \phi(x_i)$$

Note :

$$w^T m_i = \frac{1}{l_i} \sum_{j=1}^{l} \sum_{k=1}^{l_i} \alpha_j k(x_j, x_k^i) = \alpha^T M_i$$

$$(M_i)_j = \frac{1}{l_i} \sum_{k=1}^{l_i} k(x_j, x_k^i)$$

Numerator of $J(w)$:

$$w^T S_B^\phi w = w^T (m_2^\phi - m_1^\phi)(m_2^\phi - m_1^\phi)^T w$$

$$= \alpha^T M \alpha$$

$$M = (M_2 - M_1)(M_2 - M_1)^T$$

Similarly the denominator is :-

$$w^T S_W^\phi w = \alpha^T N \alpha$$

• $N = \sum_{j=1,2} K_j (I - 1_{l_j}) K_j^T$

n^{th} & m^{th} component of K_j defined as $K(x_n, x_m^j)$

I = identity matrix

1_{l_j} = Matrix with entries equal to $1/l_j$.

$$w^T S_W^{\phi} w = \left(\sum_{i=1}^{l} \alpha_i \phi^T(x_i)\right)\left(\sum_{j=1,2} \sum_{n=1}^{l_j} (\phi(x_n^j) - m_j^{\phi})(\phi(x_n^j) - m_j^{\phi})^T\right) \left(\sum_{k=1}^{l} \alpha_k \phi(x_k)\right)$$

$$= \sum_{j=1,2} \sum_{i=1}^{l} \sum_{n=1}^{l_j} \sum_{k=1}^{l} \alpha_i \phi^T(x_i) (\phi x_n^j - m_j^{\phi}) \left(\alpha_k K(x_k, x_n^j) - \frac{1}{l_j} \sum_{q=1}^{l_j} \alpha_k K(x_k, x_q^j)\right)$$

$$= \sum_{j=1,2} \left(\sum_{i=1}^{l} \sum_{n=1}^{l_j} \sum_{k=1}^{l} \left(\alpha_i \alpha_k K(x_i, x_n^j)\, K(x_k, x_n^j)\right) - 2\frac{\alpha_i \alpha_k}{l_j} \sum_{p=1}^{l_j} K(x_i, x_n^j)\, K(x_k, x_p^j) - \frac{\alpha_i \alpha_k}{l_j^2} \sum_{p=1}^{l_j} \sum_{q=1}^{l_j} K(x_i, x_p^j)\, K(x_k, x_q^j)\right)$$

$= \alpha^T N \alpha$

$\Rightarrow J(\alpha) = \dfrac{\alpha^T M \alpha}{\alpha^T N \alpha}$. Differentiating & equating to zero:

$(\alpha^T M \alpha) N \alpha = (\alpha^T N \alpha) M \alpha$

Since only the direction of w matters, & hence direction of α matters, the above can be solved for α as

$\alpha = N^{-1}(M_2 - M_1)$

Note: In practice N is singular & hence a multiple of identity is added to it ; $N_\epsilon = N + \epsilon I$

Given the solution for α, the projection of the new data point is given by $y(x) = (w \cdot \phi(x)) = \sum_{i=1}^{l} \alpha_i K(x_i, x)$.

Problem 2

```
import numpy as np
from numpy import linalg as LA
from scipy.spatial.distance import pdist, squareform
from sklearn import svm, preprocessing
from sklearn.model_selection import cross_val_score

def linearKernel(data1, data2):

      kernel = np.dot(np.transpose(data1), data2)
      return kernel

def polynomialKernel(data1, data2, p):

      mat = np.dot(np.transpose(data1), data2)
      mat = np.add(1, mat)
      kernel = np.power(mat, p)
      return kernel

def gaussianKernel(data1, data2, sigma):

      pairwise_dists = np.zeros((data1.shape[1], data2.shape[1]))
      for i in range(data1.shape[1]):
      for j in range(data2.shape[1]):
            pairwise_dists[i,j] = (LA.norm(data1[:,i]-data2[:,j]))**2
      # pairwise_dists = squareform(pdist(np.transpose(data),
'euclidean'))
      mat = np.divide(pairwise_dists, -2*sigma*sigma)
      kernel = np.exp(mat)
      return kernel

def kernelPCA(data, kerneltype = 'linear', p = 2, sigma = 0.5, numeig
= 1):

      ''' compute n x n Gram Matrix K using a kernel function '''
      if kerneltype == 'linear':
      kernel = linearKernel(data, data)
      elif kerneltype == 'polynomial':
      kernel = polynomialKernel(data, data, p)
```

```
    elif kerneltype == 'gaussian':
    kernel = gaussianKernel(data, data, sigma)
    else:
    print "Wrong kernel type"
    raise

    ''' normalise kernel matrix '''
    N = data.shape[1]
    oneN = np.divide(np.ones((N, N)), N)
    kernel = kernel - np.dot(oneN, kernel) - np.dot(kernel, oneN) +
np.dot(np.dot(oneN, kernel), oneN)

    ''' compute eigen-(values/vectors) of K '''
    eigenValues,eigenVectors = LA.eig(kernel)

    ''' sort eigen vectors according to corresponding eigen values
'''
    idx = eigenValues.argsort()[::-1]
    idx = idx[:numeig]
    eigenValues = eigenValues[idx]
    eigenVectors = eigenVectors[:, idx]

    ''' normalise the eigen vectors '''
    eigenVectors = np.divide(eigenVectors, eigenValues[None, :])

    ''' project data points into lower dimensional space '''
    projectedData = np.dot(np.transpose(eigenVectors), kernel)

    return [projectedData, eigenVectors]

def kernelLDA(data, labels, kerneltype = 'linear', p = 2, sigma =
0.5):

    ''' compute n x n Gram Matrix K using a kernel function '''
    if kerneltype == 'linear':
    kernel = linearKernel(data, data)
    elif kerneltype == 'polynomial':
    kernel = polynomialKernel(data, data, p)
    elif kerneltype == 'gaussian':
```

```
    kernel = gaussianKernel(data, data, sigma)
    else:
    print "Wrong kernel type"
    raise

    ''' compute number of elements in each class '''
    idx1 = np.argwhere(labels == 1)
    idx2 = np.argwhere(labels == -1)
    l1 = np.prod(idx1.shape)
    l2 = np.prod(idx2.shape)

    ''' seperate kernels of 2 classes '''
    K1 = kernel[:, idx1]
    K2 = kernel[:, idx2]
    K1 = K1[:,:,0]
    K2 = K2[:,:,0]

    ''' compute Mi's '''
    M1 = np.divide(K1.sum(axis=1), l1)
    M2 = np.divide(K2.sum(axis=1), l2)

    ''' compute N matrix '''
    I1 = np.subtract(np.identity(l1), np.divide(np.ones((l1, l1)),
l1))
    I2 = np.subtract(np.identity(l2), np.divide(np.ones((l2, l2)),
l2))
    N = np.add(np.dot(np.dot(K1, I1),
np.transpose(K1)),np.dot(np.dot(K2, I2), np.transpose(K2)))

    ''' check if N is invertible '''
    if N.shape[0] != LA.matrix_rank(N):
    #print LA.matrix_rank(N)
    eps = 0.000000001 * np.amin(N)
    N = np.add(N,np.dot(eps,np.identity(N.shape[0])))

    ''' compute alpha vector '''
    alpha = np.dot(LA.inv(N), np.subtract(M1, M2))

    ''' project data to one dimensional space '''
```

```
      #projectedData = np.transpose(np.dot(kernel, alpha))
      projectedData = np.dot(np.transpose(alpha), kernel)

      return [projectedData, alpha]

def train(data, labels, vdata, vlabels):

      #classifier = svm.SVC()
      classifier = svm.LinearSVC()

      #scores = cross_val_score(classifier, data, labels, cv=10)
      #print "Accuracy: %0.9f (+/- %0.9f)" % (scores.mean(),
scores.std() * 2)

      classifier.fit(preprocessing.scale(data), labels)
      pvlabels = classifier.predict(preprocessing.scale(vdata))
      error = np.mean( vlabels != pvlabels )
      print "Accuracy: %0.9f" % (1-error)

def main():
      # data = np.loadtxt('data/madelon_train.data')
      # labels = np.loadtxt('data/madelon_train.labels')
      # vdata = np.loadtxt('data/madelon_valid.data')
      # vlabels = np.loadtxt('data/madelon_valid.labels')
      data = np.loadtxt('data/arcene_train.data')
      labels = np.loadtxt('data/arcene_train.labels')
      vdata = np.loadtxt('data/arcene_valid.data')
      vlabels = np.loadtxt('data/arcene_valid.labels')
      data = np.transpose(data)
      vdata = np.transpose(vdata)
      #for kk in range(43, 44):
      #     print kk,

      [PCAdata, eigenVectors] = kernelPCA(data, 'gaussian',
sigma=10000, numeig = 43)
      [LDAdata, alpha] = kernelLDA(data, labels, 'gaussian',
sigma=10000)

      ''' create kernel for validation data and then normalise it '''
```

```
    N1 = data.shape[1]
    N2 = vdata.shape[1]
    kernel = gaussianKernel(vdata, data, 10000)
    kernelt = gaussianKernel(data, data, 10000)
    oneN2 = np.divide(np.ones((N1, N1)), N1)
    kernelt = kernelt - np.dot(oneN2, kernelt) - np.dot(kernelt,
oneN2) + np.dot(np.dot(oneN2, kernelt), oneN2)
    oneN = np.divide(np.ones((N2, N1)), N1)
    oneN1 = np.divide(np.ones((N1, N1)), N1)
    print kernel.shape, oneN.shape, oneN1.shape, kernelt.shape
    kernel1 = kernel - np.dot(oneN, kernelt) - np.dot(kernel, oneN1)
+ np.dot(np.dot(oneN, kernelt), oneN1)

    PCAvdata = np.dot(np.transpose(eigenVectors),
np.transpose(kernel))
    LDAvdata = np.dot(np.transpose(alpha), np.transpose(kernel))
    train(np.transpose(data), labels, np.transpose(vdata), vlabels)
    train(np.transpose(PCAdata), labels, np.transpose(PCAvdata),
vlabels)
    train(np.transpose(LDAdata).reshape(-1,1), labels,
np.transpose(LDAvdata).reshape(-1,1), vlabels)


if __name__ == '__main__':
    main()
```

UCI Arcene Dataset (p = 3, sigma = 10000)

Linear SVM PCA

K	Linear	Polynomial	Gaussian
10	79	78	73
20	82	80	83
30	83	81	80
40	84	82	81
50	86	83	74
60	87	86	78
70	82	86	77
80	81	86	86
90	82	85	83
100	79	81	85

RBF SVM LDA

K	Linear	Polynomial	Gaussian
10	74	74	77
20	73	76	82
30	83	79	79
40	79	77	84
50	78	76	85
60	80	74	85
70	82	82	83
80	80	80	81
90	85	81	81
100	86	82	83

Linear SVM LDA

Linear	Polynomial	Gaussian
84	86	86

RBF SVM LDA

Linear	Polynomial	Gaussian
84	86	86

UCI Madelon Dataset (p = 3, sigma = 1000000)

Linear SVM PCA

K	Linear	Polynomial	Gaussian
20	70.83	70.83	71
80	70.83	71.16	70.66

RBF SVM PCA

K	Linear	Polynomial	Gaussian
20	71.66	72.16	71.9
80	72.83	72.60	72

Linear SVM LDA

Linear	Polynomial	Gaussian
53	53	67

RBF SVM LDA

Linear	Polynomial	Gaussian
53	53	67