Homework 2 **Anirudh Ganesh**
CSE 5523
Prof. Mikhail Belkin
TA: Siyuan Ma Due Date: 03/27/18

Introduction

Imports

```
import scipy.io
import numpy as np
from sklearn.svm import LinearSVC

from numpy.linalg import inv, solve, matrix_rank
from random import shuffle, randint
from sklearn.preprocessing import scale

import numpy.linalg as la
import numpy.random as rnd

import scipy.spatial as spt
import scipy.sparse.linalg as sla
```

Load Files

```

 train = scipy.io.loadmat('79.mat')
 test = scipy.io.loadmat('test79.mat')

X = np.array(train['d79'])
 test_X = np.array(test['d79'])

a = np.ones(1000)*7
b = np.ones(1000)*9

y = np.append(a, b)
```

Problem 1

Question : Run a linear SVM on the two class dataset given online (you can use a standard toolbox). Compare its performance to that of the least squares linear classifier.

Linear SVM

Code :

```
clf = LinearSVC(random_state=0)
clf.fit(X, y)

print(''Accuracy = {}%''.format(clf.score(test_X, y)*100.0))
```

Output :

```
Accuracy = 92.6 %
```

Least Squares Linear Classifier

Code :

```
def train_lss(X, y):
    D = X.shape[1] + 1
    K = y.shape[1]

    sum1 = np.zeros((D, D))
    sum2 = np.zeros((D, K))
    i = 0

    for x_i in X:
        x_i = np.append(1, x_i)
        y_i = y[i]
        sum1 += np.outer(x_i, x_i)
        sum2 += np.outer(x_i, y_i)
        i += 1


    while matrix_rank(sum1) != D:

        sigma = 0.001

        sum1 = sum1 + sigma * np.eye(D)

    return np.dot(inv(sum1), sum2)

def predict(W, x):
    x = np.append(1, x)
    values = list(np.dot(W.T, x))
    winners =[i for i, x in enumerate(values) if x == max(values)]
    index = randint(0, len(winners) - 1)
    winner = winners[index]
    y = [0 for x in values]
    y[winner] = 1
    return y

def makeOneHot(y):
    oneHotY = []
    temp = [1, -1]
    for i in range(1000):
        oneHotY.append(temp)
    temp = [-1, 1]
    for i in range(1000):
        oneHotY.append(temp)
    return np.matrix(oneHotY)

def getPred(t):
    if t[0] == 1:
        return 7
    else:
        return 9

def test(X, y, X_test, y_test):
    ohy = makeOneHot(y)
    W = train_lss(X, ohy)
    total = y_test.shape[0]
    i = 0
    correct = 0
    for i in range(total):
        prediction = predict(W, X_test[i])
        actual = y_test[i]
        if getPred(prediction) == actual:
            correct += 1
    accuracy = correct/float(total) * 100
    print(``Accuracy is {}% ({} correct, out of {})''.format(accuracy,
        correct, total))

test(X, y, test_X, y)
```

Output :

```
Accuracy is 93.85% (1877 correct, out of 2000)
```

Comparision : We can see that the Least Squares Classifier performs slightly better than the linear SVM.

Problem 2

Question : Implement (do not use standard toolboxes) the Least Squares Kernel classifier with Gaussian and Laplace kernels. Learn a kernel classifier (choosing a proper value for the kernel width and λ using cross-validation). After it is done (no cheating!), download test79.mat and test your code on the test set. Compare your testing and training results. Compare results to linear SVM. Try $\lambda = 10^{-8}$. How do these results compare? What happens, when the kernel bandwidth σ is small? Large?

Gaussian Kernel

The kernel is given by,

$$k(x, y) = exp(-\frac{||x-y||^2}{2\sigma^2})$$

Code :

```
def gaussian_kernel(xi, xj, sigma = 2.5):
    N = np.linalg.norm(xi-xj)**2
    G = np.exp(-N * 0.5 / sigma)
    return G

def gauss_lss(X, y, penalty=1e-4, sigma=1e+4):
    X = scale(X)
    N = X.shape[0]
    K = np.zeros((N, N))
    for i in range(0, N):
        for j in range(0, N):
            K[i, j] = gaussian_kernel(X[i,:], X[j,:], sigma)

    KI = np.linalg.inv(np.add(K, penalty*np.identity(N)))
    KIy = np.matmul(KI, y)

    return KIy

def gauss_test(X, y, X_test, y_test, penalty=1e-4, sigma=1e+4):
    count = 0
    KIy = gauss_lss(X, y, penalty, sigma)
    N = X_test.shape[0]
    X_test = scale(X_test)
    test_score = 0
    for i in range(0, N):
        sum1 = 0
        for j in range(0, N):
            sum1 = sum1 + KIy[j]*gaussian_kernel(X_test[j,:], X_test[i, :],
                sigma)
        if sum1 >= 0:
            predict = 1
        else:
            predict = -1
        if predict == y[i]:
            count += 1

    accuracy = count/N
    print("Accuracy is {}% ({} correct, out of {})".format(accuracy*100.0,
        count, N))
```

Output :

```
Accuracy is 98.1% (1962 correct, out of 2000)
```

Laplacian Kernel

The kernel is given by,

$$k(x, y) = exp(-\frac{||x - y||}{\sigma})$$

Code :

```
def laplace_kernel(xi, xj, sigma = 2.5):
    N = np.linalg.norm(xi-xj)
    L = np.exp(-N / sigma)
    return L

def laplace_lss(X, y, penalty=1e-4, sigma=1e+4):
    X = scale(X)
    N = X.shape[0]

    s = np.arange(X.shape[0])
    np.random.shuffle(s)

    X = X[s]
    y = y[s]

    K = np.zeros((N, N))
    for i in range(0, N):
        for j in range(0, N):
            K[i, j] = laplace_kernel(X[i,:], X[j,:], sigma)

    KI = np.linalg.inv(np.add(K, penalty*np.identity(N)))
    KIy = np.matmul(KI, y)

    return KIy

def laplace_test(X, y, X_test, y_test, penalty=1e-4, sigma=1e+4):
    count = 0
    KIy = laplace_lss(X, y, penalty, sigma)
    N = X_test.shape[0]
    X_test = scale(X_test)

    s = np.arange(X.shape[0])
    np.random.shuffle(s)

    X_test = X_test[s]
    y_test = y_test[s]

    test_score = 0
    for i in range(0, N):
        sum1 = 0
        for j in range(0, N):
                sum1 = sum1 + KIy[j]*gaussian_kernel(X_test[j,:], X_test[i, :],
                    sigma)
            if sum1 >= 0:
                predict = 1
            else:
                predict = -1
            if predict == y_test[i]:
                count += 1

    accuracy = count/N
    print("Accuracy is {}% ({} correct, out of {})".format(accuracy*100.0,
       count, N))

y_enc = np.ones(X.shape[0])
y_enc[0:1000] = -1
y_enc[1000:2000] = 1
laplace_test(X, y_enc, test_X, y_enc, penalty=1e-4, sigma=1e+4)
```

Output :

```
Accuracy is 97.95% (1962 correct, out of 2000)
```

Impact of parameters pn Gaussian Kernel: I noticed that upon using $\lambda = 10^{-8}$, I get a result of 97.75% which is a considerable drop from our previous performance of 98.1% with $\lambda = 10^{-4}$ which I found to be the optimal using cross-validation.

For $\sigma = 100, 1000, 10000, 100000$, we notice that the kernel gives an output of 97.85%, 98.25%, 98.1%, 95.8% respectively. If the kernel bandwidth σ is too small, we see that there is a drop in performance which is alleviated as we increase the bandwidth. But after a certain threshold, the increased bandwidth actually hurts our learning. This maybe due to it "devaluing" each pairwise input so much that contributions get undervalued to the point of irrelevancy.

Problem 3

Question : Use Fourier features to approximate the kernel. Try different numbers of features. Compare your results to Problem 2.

Code :

```
penalties=np.array([0.00001,0.0001,0.001,0.01,0.1,1])
data = {}

for D in range(100,700,100):
    data[D] = {}
    maxV = 0
    maxG = 0
    maxP = 0
    w=np.random.randn(D,d)
    b=2*math.pi*np.random.rand(D,1)
    for penalty in penalties:
      gamma=0.000001
      while(gamma<=0.01):

          X_train = X[:,0:1800]
          Y_train = y[0:1800]
          X_test = X[:,1800:2000]
          y_test = y[1800:2000]
          gwX=gamma*np.matmul(w, X_train)
          bI=b*np.ones((1,n-200))
          gwXbI=gwX+bI
          z=np.cos(gwXbI)
          newmat=(np.add(np.matmul(z,np.matrix.transpose(z)),penalty*np.
              identity(D)))
          KIy=np.linalg.lstsq(newmat,np.matmul(z,training_y))[0].ravel()
          KIy=KIy.reshape([D,1])
          cnt1=0
          for i in range(0,200):
              x_test = X_test[:,i].reshape([784,1])
              h=gamma*np.matmul(w, x_test)
              h=h.reshape([D,1])
              hb=np.cos(np.add(h,b))
              ztest=np.matmul(np.transpose(KIy),hb)
              if(ztest[0,0]>=0):
                predict=1
              else:
                predict=-1
              if(predict==test_y[i]):
                  cnt1=cnt1+1
          if (maxV <= cnt1/200):
              maxV = cnt1/200
              maxG=gamma
              maxP=penalty
          print('D = {} Penalty= {} Gamma= {} ValidationScore(1-fold)= {}
              '.format(D,penalty,gamma,cnt1/200))
          gamma=gamma*10
      data[D]['Penalty'] = maxP
      data[D]['Gamma'] = maxG
      data[D]['Validation'] = maxV
```

Output :

```
Maximum Values:

{100: {'Gamma': 0.01, 'Penalty': 1.0, 'Validation': 0.955},
 200: {'Gamma': 0.001, 'Penalty': 0.01, 'Validation': 0.94},
 300: {'Gamma': 0.01, 'Penalty': 0.01, 'Validation': 0.98},
 400: {'Gamma': 0.01, 'Penalty': 1.0, 'Validation': 0.98},
 500: {'Gamma': 0.01, 'Penalty': 1.0, 'Validation': 0.985},
 600: {'Gamma': 0.01, 'Penalty': 0.1, 'Validation': 0.99}}
```

Thus we can see that for most configurations, a gamma of 0.01 gives us the most optimal validation score.

Problem 4

Question : Why is the Hilbert space of square integrable functions on $[0, 1]$, i.e., functions f, such that $\int_0^1 |f(x)|^2 dx$, not an RKHS?

Solution :

The space of square integrable functions f, such that $\int_0^1 |f(x)|^2 dx$ forms a Hilbert space with the inner product $\langle f, g\rangle = \int_0^1 f(x)g(x)dx$. We know this is an inner product because, it satisfies :

1. Symmetry,

   $\langle f, g\rangle = \int_0^1 f(x)g(x)dx = \int_0^1 g(x)f(x)dx = \langle g, f\rangle \forall f, g$

2. Bilinearity,

   $\langle \alpha u+\beta v, w\rangle = \int_0^1 f(\alpha u)g(w)dx+\int_0^1 f(\beta v)g(w)dx = \alpha \int_0^1 f(u)g(w)dx+\beta \int_0^1 f(v)g(w)dx = \alpha\langle u, w\rangle + \beta\langle v, w\rangle$

3. Positive Definiteness,

   $\langle f, g\rangle \geq 0$ and $\langle f, f\rangle = 0 \iff f = 0$

Now that we know that the given set of functions is actually a hilbert space, we need to see if it is a reproducing kernel of hilbert space. A function $k(., .)$ is a reproducing kernel of hilbert space $\mathcal{H}$ if $\forall f \in \mathcal{H}, f(x) = \langle k(x, .), f(.)\rangle$.

Now we easily see from this definition that, our set of square integrable functions is not an RKHS because, the δ function which has reproducing property given by

$$f(x) = \int_0^1 \delta(x - u)f(u)du$$

does not satisfy the square integrable condition.

$$\int_0^1 \delta(u)^2 du \nless \infty$$

This means that there can be a δ such that for some point in the function, it may not be a bounded operator on $\mathcal{H}$, i.e., for some point, δ tends to ∞.

Thus the Hilbert space of square integrable functions is not an RKHS.

Problem 5

Question : Read about smoothing splines. What is the connections to the material discussed in class?

Solution : Smoothing splines are like kernel regression that we have covered extensively in this homework and also in the material presented in class. But before we can take a look into how smoothing splines work in regression, we need to know what splines are. Simply put, splines are a set of piecewise polynomial functions of an order. The degree of these polynomials is determined by the knot number, which is the degree of the spline.

This collection of knots property of splines is particularly useful when it comes to performing regression as we can fit data to splines by choosing good knot points. But this also means that the quality of our regression depends on the choice of our knot points. Apart from this, the splines also suffer from high variance as our degree gets higher. Fortunately, smoothing splines take care of both of these problems for us. They perform regularized regression over our natural spline basis, i.e. the data points. The smoothing directly helps circumvent the knot selection problem and the expansion of basis leads to a shrinking of coefficients, acting as a check on overfitting.

Another interesting observation regarding smoothing splines is that they can be alternatively motivated from a functional minimization perspective. If we want to minimize over all functions f we have,

$$\sum_{i=1}^{n}(y_i - f(x_i))^2 + \lambda \int (f''(x))^2 dx$$

This can be viewed as a least squares error of f over $(x_i, y_i), i = 1, ..n$

Practically, smoothing splines can be viewed as an simpler alternative to the kernel methods. The only thing we need to tune for is the smoothing parameter λ, which can be typically easily achived through cross-validation. Also due to the nature of splines, they are also computationally more efficient.