UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
Department of Electrical and Computer Engineering

ECE 544NA Pattern Recognition

Solutions 7

Fall 2013

Assigned: Thursday, October 31, 2013 Due: Thursday, November 14, 2013

Reading: NNPR Chapter 8, links posted on the class website

Problem 7.1

A kernel $k(x, y)$ is called a Mercer kernel if it satisfies the Mercer conditions: 1) $k(x, y)$ is symmetric, 2) continuous, and 3) positive semi-definite. For each of the following, we show 3) since 1) and 2) are obvious.

(a) If $k_1(x, y)$ is a Mercer kernel, by Mercer's theorem $k_1(x, y) = \sum_{i=1}^{\infty} \lambda_i^{(1)} \phi_i^{(1)}(x)\phi_i^{(1)}(y)$ and likewise $k_2(x, y) = \sum_{i=1}^{\infty} \lambda_i^{(2)} \phi_i^{(2)}(x)\phi_i^{(2)}(y)$. To show that $k(x, y) = k_1(x, y)k_2(x, y)$ is positive definite, we need to show

$$\int \int f(x)f(y)k_1(x, y)k_2(x, y)dxdy \geq 0$$

$\forall f$. By Mercer's theorem, the LHS is equivalent to

$$\int \int f(x)f(y) \left(\sum_{i=1}^{\infty} \lambda_i^{(1)} \phi_i^{(1)}(x)\phi_i^{(1)}(y) \right) \left(\sum_{j=1}^{\infty} \lambda_j^{(2)} \phi_j^{(2)}(x)\phi_j^{(2)}(y) \right) dxdy$$

Assuming continuity of $k_1(x, y)$, $k_2(x, y)$ and uniform convergence of the sums (Mercer's theorem), we can swap the order of the sums and integrals:

$$\sum_{i=1}^{\infty} \sum_{j=1}^{\infty} \lambda_i^{(1)} \lambda_j^{(2)} \int \int [\phi_i^{(1)}(x)\phi_j^{(2)}(x)f(x)][\phi_i^{(1)}(y)\phi_j^{(2)}(y)f(y)]dxdy$$

$$= \sum_{i=1}^{\infty} \sum_{j=1}^{\infty} \lambda_i^{(1)} \lambda_j^{(2)} \left[\int \phi_i^{(1)}(x)\phi_j^{(2)}(x)f(x)dx \right]^2 \geq 0$$

Hence, if $k_1(x, y)$ and $k_2(x, y)$ are Mercer kernels, then $k(x, y) = k_1(x, y)k_2(x, y)$ is also a Mercer kernel. The expression above also provides insight into the underlying feature map for $k(x, y)$: $\vec{\phi}(x) = \{\phi_i^{(1)}(x)\phi_j^{(2)}(x)\}_{\forall i \forall j}$ – it is the componentwise product of the eigenfunctions of k_1 and k_2.

(b) Another approach to proving that a given function is a Mercer kernel is to simply show that there exists some space in which the kernel is an inner product. We therefore prove this result by constructing such a kernel. Let $k_1(x, y) = \langle \phi^{(1)}(x), \phi^{(1)}(y) \rangle$ and $k_2(x, y) = \langle \phi^{(2)}(x), \phi^{(2)}(y) \rangle$. We want to show that $k(x, y) = ak_1(x, y) + bk_2(x, y)$ for $a, b \geq 0$ is a Mercer kernel if $k_1(x, y)$ and $k_2(x, y)$ are Mercer kernels. Let us construct $\phi(x) = [\sqrt{a}\phi^{(1)} \ \sqrt{b}\phi^{(2)}(x)]$. Clearly then $k(x, y) = \langle \phi(x), \phi(y) \rangle = a\langle \phi^{(1)}(x), \phi^{(1)}(y) \rangle + b\langle \phi^{(2)}(x), \phi^{(2)}(y) \rangle = ak_1(x, y) + bk_2(x, y)$ is also a Mercer kernel.

(c) $k(x, y) = (x^T y + c)^d$. A constant $c \geq 0$ is trivially positive semi-definite; $x^T y$ is the inner product in some n-dimensional euclidean space $\mathbb{R}^n$ and hence $x^T y$ is a Mercer kernel. From (b), we know that the sum of two kernels is a (Mercer) kernel, and therefore $(x^T y + c)$ is also a Mercer kernel. Since $(x^T y + c)^d$ is $(x^T y + c)$ multiplied d times, it follows from (a) that $(x^T y + c)^d$ is a Mercer kernel.

(d) The expression $\sum_{x,y} f(x)f(y)k(x,y) \geq 0$ for any nonzero sequence $f(x)$, $x = 1, 2, ..., 100$ is equivalent to showing that the matrix K constructed by $\{K_{ij} = min(i,j)\}_{i,j=1,...,100}$ is positive semi-definite.

$$K = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 2 & \cdots & 2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 2 & 3 & \cdots & 100 \end{pmatrix}$$

Let us define $O^{(0)}$ to be 100 by 100 matrix of all ones, which is trivially rank one and positive semi-definite; let us further define $O^{(i)}$ to be $O^{(0)}$, but with every row and column vectors, $1 \leq j \leq i$, set to 0. Since $O^{(0)}$ is rank one and positive semi-definite, $O^{(i)}$ is also rank one and positive semi-definite for $1 \leq i \leq 99$. We display $O^{(1)}$, and $O^{(2)}$ below:

$$O^{(1)} = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 1 & \cdots & 1 \\ 0 & 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & 1 & \cdots & 1 \end{pmatrix} \qquad O^{(2)} = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 1 & \cdots & 1 \end{pmatrix}$$

It can be observed that $K = \sum_{i=1}^{100} O^{(i-1)}$ and since $O^{(i)}$ is positive semi-definite for each i, K is positive semi-definite. Hence $k = min(x,y)$ is positive semi-definite.

This is called the histogram intersection kernel and a much simpler proof exists (Barla, Odoni, Verri, 2003) where we can define a mapping $\phi(x)\ \epsilon\ \mathbb{R}^N$ such that $\phi_i(x) = 1$ for $1 \leq i \leq x$ and 0 otherwise. Then, $\phi(x)^T\phi(y) = min(x,y) = k(x,y)$.

Matlab Exercises

Problem 7.2

Table 1 presents the average classification error of a linear least squares classifier on the full dataset, PCA, kernel PCA with a Gaussian kernel, and a restricted Boltzmann machine (RBM). It is expected that with only two dimensions, the three dimensionality reduction techniques are much worse than when the full dataset is used. Of these, kernel PCA has a slight advantage (but not much) and note that RBM and PCA return exactly the same results. Also note that these results are quite competitive – with only two dimensions, they outperform k-nearest neighbors and the perceptron algorithm trained on the original feature space. Table 2 summarizes the advantages and disadvantages of the three techniques.

Table 1: Average classification error across 100 random trials

classifier	average error	standard deviation
full data	0.23	0.03
PCA	0.29	0.03
kernel PCA	0.28	0.03
RBM	0.29	0.03

Table 2: Advantages and disadvantages of the three dimensionality reduction techniques

method	**advantages**	**disadvantages**
PCA	Simple, gauranteed to be optimal no additional parameters to tune	Eigenvalue decomposition (can can be cumbersome in high dimensions)
kernel PCA	Recovers underlying nonlinearities Guaranteed to be optimal	Computing kernels is costly needs entire training set at test time infinite choice of kernels and parameters
RBM	Fast approximation to PCA	gradient descent (sensitive to initialization, local optima)

```
function results =  runexps(numtrials)
%runexps(numtrials) runs numtrials independent experiments for this
%problem set
%results contains results for 1) full dataset, 2) PCA, 3) kernel PCA
%and 4) RBM
results = zeros(4, numtrials);
load diabetes_normalized.mat
diab_labels = diabetes_normalized(:,1);
diab_features = diabetes_normalized(:,2:9);
numsamples = length(diab_labels);
numtrain = ceil(numsamples*0.8);
numtest = numsamples - numtrain;

numeigs = 2;

for i = 1:numtrials
    randinds = randperm(numsamples); %random partitioning of the entire dataset
    traininds = randinds(1:numtrain); %first 80% is training data
    testinds = randinds(numtrain + 1: numsamples); %next 20% is test data

    mytraindata = diab_features(traininds,:);
    mytraindatabias = [mytraindata ones(numtrain, 1)]; %include the bias term

    mytestdata = diab_features(testinds, :);
    mytestdatabias = [mytestdata ones(numtest, 1)];

    y_train = diab_labels(traininds);
    y_test = diab_labels(testinds);

    %Standard linear classifier
    w_full = (mytraindatabias'*mytraindatabias) \ (mytraindatabias'*y_train);
    y_full = 2.*(mytestdatabias*w_full >= 0) - 1;
    results(1, i) = sum(y_full ~= y_test)/length(y_test);

    %% Principal components analysis
    estmean = mean(mytraindata, 1); %mean center
    meancentered = mytraindata - repmat(estmean, numtrain, 1);
    corrmat = meancentered'*meancentered; %8 x 8 correlation matrix
    [eigvecs, ~] = eigs(corrmat, numeigs); %picks the two largest eigenvals
```

```
pcafeats = meancentered*eigvecs;

%train classifier
pcabias = [pcafeats ones(numtrain, 1)]; %should not be necessary
w_pca = (pcabias'*pcabias) \ (pcabias'*y_train);

%apply the transformation to the test set
testmeancentered = mytestdata - repmat(estmean, numtest, 1);
pcafeats_test = testmeancentered*eigvecs;
testpca = [pcafeats_test ones(numtest, 1)];
y_pca = 2.*(testpca*w_pca >= 0) - 1;
results(2,i) = sum(y_pca ~= y_test)/length(y_test);

%% Kernel PCA
%Compute the kernel matrix (note that this can also be vectorized with
%some restrictions on ||x||, but our dataset is small enough

%learning gamma requires a grid search, it controls the "influence" a
%particular datapoint has.  Since the data is somewhat normalized, we
%arbitrarily select gamma = 1/2 [since 2 is the maximum distance
%between any two points]
gamma = 0.5;
K = zeros(numtrain, numtrain);
for j = 1:numtrain
    for k = 1:numtrain
        K(j,k) = exp(-norm(mytraindata(j,:) - mytraindata(k,:))*gamma);
    end
end

%Mean center K
kappa = sum(K, 2)./numtrain;
kconst = sum(sum(K))/numtrain^2;
trainones = ones(numtrain, 1);
K_centered = K - trainones*kappa' - kappa*trainones' + kconst.*trainones*trainones';

%Find the top 2 eigenvalues (eigs is better here)
[alphas, ~] = eigs(K_centered, numeigs);

%find lower dimensional representation
lowdimtrain = K_centered*alphas;
lowdimtrainbias = [lowdimtrain ones(numtrain, 1)];

%train a classifier
w_kpca = (lowdimtrainbias'*lowdimtrainbias) \ (lowdimtrainbias'*y_train);

%compute test kernels and mean center them
K_test = zeros(numtest, numtrain);
for n = 1:numtest
    for m = 1:numtrain
        K_test(n,m) = exp(-gamma*norm(mytraindata(m,:)-mytestdata(n,:)));
    end
```

```
    end
    kappatest = sum(K_test, 2);
    testones = ones(numtest, 1);
    Ktestcent = K_test - testones*kappa' - kappatest*trainones' + kconst.*testones*trainones';

    lowdimtest = Ktestcent*alphas; %2 -dimensional representation
    lowdimtestbias = [lowdimtest ones(numtest, 1)];

    %test the classifier
    y_kpca = 2.*(lowdimtestbias*w_kpca >= 0) - 1;

    results(3,i) = sum(y_kpca ~= y_test)/length(y_test);

    %% Restricted Boltzmann Machine
    rbmmap = myrbm(meancentered, numeigs, 10^(-7));
    rbmproj = meancentered*rbmmap;
    rbmbias = [rbmproj ones(numtrain, 1)];

    %learn a linear classifier
    w_rbm = (rbmbias'*rbmbias) \ (rbmbias'*y_train);

    %project test data
    rbmtest = testmeancentered*rbmmap;
    rbmtestbias = [rbmtest ones(numtest, 1)];
    %classify
    y_rbm = 2.*(rbmtestbias*w_rbm >= 0) - 1;

    results(4,i) = sum(y_rbm ~= y_test)/length(y_test);


end


function [W, c] = myrbm(traindata, dim, thresh)
%W = myrbm(traindata, dim, thresh) reduces the dimension of traindata to
%dim dimensions using a restricted boltzmann machine (RBM) with Gaussian
%nodes.  The stopping criterion is determined by thresh.

[numtrain, numfeats] = size(traindata); %get the size of the training data
W = zeros(numfeats, dim);
b = zeros(1, numfeats);
c = zeros(1, dim);
eta = 0.001;

%Set the initial variance
sigma = max(diag(traindata'*traindata)./numtrain);

trainones = zeros(numtrain, 1);

%Note that this is done in batch mode.  When numfeats is really large
%estimating h, v as well as the gradient update is more efficient in
```

```
%an online setting
%when the variance of the Gaussian is really small, then it's like an
%impulse
while sigma >= thresh
    %generate h
    h = normrnd(traindata*W + trainones*c, sigma);

    %approximation error
    h_err = traindata*W + trainones*c - h;
    v_err = h*W' + trainones*b - traindata;

    %Gradient updates
    W = W - eta*(traindata'*h_err + v_err'*h);
    b = b - eta*sum(v_err, 1);
    c = c - eta*sum(h_err, 1);

    %update sigma
    sigma = sigma*0.99;
    %For a Gaussian, we could alternatively adapt eta as a function of
    %iteration


end


end
```