# INF 4300 - Exercise for Friday 29.10.2010
# *Statistics and classification using Matlab*

**Goal**

This exercise aims to learn you the basics of the PRTools toolbox and how it can be used to do simple classification and clustering tasks. This exercise will also introduce you to the LIBSVM toolbox. Please read the **Stuff to learn** and try out the commands there, before you start doing the exercises.

**Stuff to learn**

*Installing PRTools, LIBSVM and exercise datasets*

PRTools can be downloaded from http://prtools.org/download.html, or by adding prtools to the Matlab path; `addpath('/local/opt/matlab/ifi/toolbox/prtools')`. Add `~inf4300/prdatasets` as well. Useful for this exercise is also `~inf4300/prdatasets/Hu` and `~inf4300/prdatasets/ddata`

Download LIBSVM including a simple Matlab interface from `http://www.csie.ntu.edu.tw/~cjlin/libsvm/#matlab`. Follow the installation instructions included. (I suggest you place all the files in a sub-directory, in your current Matlab working directory, called `libsvm/`. Note that the binaries available for this package will only run on linux and windows operating systems.)

*Data structures*

The MATLAB language usually works with the MATLAB array object type. The 2 - dimensional array – a matrix – is the most common type (although N-dimensional arrays are allowed).
The standard operations are defined as the matrix operations (a scalar is regarded as a 1x1 matrix). For example:

```
C=A^-1 * B
```

This equation first calculates the inverse of matrix A and then multiplies the result with matrix B. Matrix C is the result. The object-oriented structure in MATLAB allows us to define specific functions, operators, and object classes of our own. The PRTools 4.0 toolbox does this, and defines two classes (*dataset* and *mapping*) with the corresponding operators and functions.

In PRTools the easiest way to apply a *mapping* W to a *dataset* A is by `A*W`. The matrix multiplication symbol * is overloaded here. The operation may also be written as `map(A,W)`. Like anywhere else in MATLAB, concatenations of operations are possible, e.g. `A*W1*W2*W3`, and are executed from left to right.

*Datasets*

PRTools deals with sets of objects represented by vectors in a feature space. The central data structure is a so-called *dataset*. It consists of a matrix of size `m` by `k`, where `m` row

vectors representing the objects given by k features each. Attached to this matrix is a set of m labels (strings or numbers), one for each object and a set of k feature names (also strings or numbers), one for each feature. Moreover, a set of prior probabilities, one for each class, is stored. Objects with the same label belong to the same class. In most help files in PRTools, a dataset is denoted by A. Almost all routines can handle multi-class objects. Some useful routines to handle datasets are:

| | |
|---|---|
| `dataset` | Define dataset from data matrix and labels |
| `gendat` | Generate a random subset of a dataset |
| `genlab` | Generate dataset labels |
| `seldat` | Select a specified subset of a dataset |
| `setdat` | Define a new dataset from an old one by replacing its data |
| `getdata` | Retrieve data from dataset |
| `getlab` | Retrieve object labels |
| `getfeat` | Retrieve feature labels |
| `renumlab` | Convert labels to numbers |

Sets of objects may be given externally or may be generated by one of the data generation routines in PRTools. Their labels may be given externally or may be the results of classification. A dataset containing 10 objects with 5 random measurements can be generated by:

```
>> data = rand(10, 5);
>> a = dataset(data)
10 by 5 dataset with 0 classes: []
```

In this example no labels are supplied, therefore no classes are detected. Labels can be added to the dataset by:

```
>> labs = [1 1 1 1 1 2 2 2 2 2]'; % labs should be a column
>> a = dataset(a, labs)
10 by 5 dataset with 2 classes: [5 5]
```

Note that the labels have to be supplied as a column vector. A simple way to assign labels to a dataset is offered by the routine `genlab` in combination with the MATLAB char command:

```
>> labs = genlab([4 2 4], char('apple', 'pear', 'banana'))
>> a = dataset(a, labs)
10 by 5 dataset with 3 classes: [4 4 2]
```

Note that the order of the classes has changed. Use the routines `getlab` and `getfeat` to retrieve the object labels and the feature labels of a. The fields of a dataset can be made visible by converting it to a structure, e.g.:

```
>> struct(a)
data: [10x5 double]
lablist: [3x6 char]
nlab: [10x1 double]
labtype: 'crisp'
```

```
targets: []
featlab: [5x1 double]
featdom: {[] [] [] [] []}
prior: []
cost: []
objsize: 10
featsize: 5
ident: [10x1 double]
version: {[1x1 struct] '15-Nov-2005 10:06:44'}
name: []
user: []
```

In the online information on datasets (`help datasets`) the meaning of these fields is explained. Each field may be changed by a set-command, e.g.

```
>> b = setdata(a, rand(10, 5));
```

Field values can be retrieved by a similar get-command, e.g.

```
>> classnames = getlablist(a)
```

In `nlab` an index is stored for each object to the list of class names `lablist`. Note that this list is alphabetically ordered. The size of the dataset can be found by both size and getsize:

```
>> [m, k] = size(a);
>> [m, k, c] = getsize(a);
```

The number of objects is returned in `m`, the number of features in `k` and the number of classes in `c`. The class prior probabilities are stored in `prior`. It is by default set to the class frequencies if the field is empty. Data in a dataset can also be retrieved by `+a`.

- Execute the commands given above to create the dataset `a`.
- Have a look at the help information of `seldat`. Use the routine to extract the banana class from a and check this by inspecting the result of `+a`.

Datasets can be manipulated in many ways as with MATLAB matrices. So `[a1;a2]` combines two datasets, provided that they have the same number of features. The feature set may be extended by `[a1 a2]` if `a1` and `a2` have the same number of objects.

- Generate 3 new objects of the classes 'apple' and 'pear' respectively, and add them to the dataset a. Check if the class size changes accordingly.
- Add a new, 6th feature to the whole dataset `a`.

Another way to inspect a dataset is to make a scatter plot of the objects in the dataset. For this the function `scatterd` is supplied. This plots each object in a dataset in a 2D graph using a colored marker when class labels are supplied. For example: `scatterd(a)`. When more than two features are present in the dataset, only the first two are used. To obtain a scatter plot of two other features they have to be explicitly extracted first, e.g. `a1 = a(:,[2 5]);` It is also possible to create 3D scatter plots.

- Use `scatterd` to make a scatter plot of the features 2 and 5 of dataset a.
- Make a 3-dimensional scatter plot by `scatterd(a, 3)` and try to rotate it by the mouse after pressing the right toolbar button.

*Classifiers*

In PRTools datasets are transformed by *mapping*s. These are procedures that map a set of objects from one space into another. Often a mapping has to be trained, i.e. it has to be adapted to a training set by some estimation or training procedures to minimize some error for the training set.

An example is the principal component analysis that performs an orthogonal rotation according to the directions with main variances in a given dataset (Principal component analysis is mentioned here only as an example of mapping. The concept, though known to some of you, will not be discussed in great detail in this course):

```
>> w = pca(a, 2)
Principal Component Analysis, 2 to 2 trained mapping --> affine
```

This just defines the mapping ('trains' it by a) for finding the first 2 principal components.

The fields of a mapping can be shown by `struct(w)`. By typing `help mappings` more information on mappings can be found. The mapping `w` may be applied to a or to another 10-dimensional dataset by:

```
>> b = map(a, w)
10 by 2 dataset with 3 classes: [4 4 2]
```

Instead of the routine map also the '`*`' operator may be used for applying mappings to datasets:

```
>> b = a*w
10 by 2 dataset with 3 classes: [4 4 2]
```

The '`*`' operator may also be used for training. `a*pca` is equivalent with `pca(a)` and `a*pca([], 2)` is equivalent to `pca(a, 2)`.

A special case of mapping is a *classifier*. It maps a dataset to distances of a discriminant function or to class posterior probability estimates. They can be used in an untrained as well as in a trained mode. When applied to a dataset, in the first mode the dataset is used for training and a classifier is generated, while in the second mode the dataset is classified.

Unlike mappings, fixed classifiers do not exist. Some important classifiers are:

| | |
|---|---|
| `qdc` | Quadratic classifier assuming normal densities (sigma unconstrained) |
| `udc` | Quadratic classifier assuming normal densities with uncorrelated features |
| `ldc` | Linear classifier assuming normal densities with equal covariance matrices |
| `nmc` | Nearest mean classifier (sigma diagonal and equal) |

• Generate a dataset a by `gendath` and compute the Linear classifier by `w =`

`ldc(a)`. Make a scatter plot of a and plot the classifier by `plotc(w)`, and the underlying density estimate by `plotm(w)`. Different colors for classifiers can be realized by adding a color label, e.g. `plotc(w, 'r')`. Type help `plotc` for more information about the use of color. Classify the training set by `d=map(a, w)` or `d = a*w`. Show the result on the screen by `+d`.
This is a density estimate based classifier, and will after training give density estimators for all classes in the training set. Estimates for objects in some dataset b can be found by `d = b*w`. A posteriori probability estimates are found after normalisation to ensure that the posterior probabilities sum to one. This is enabled by `classc`. So `classc(map(a,w))`, or `a*w*classc` maps the dataset a on the trained classifier `w` and normalises the resulting posterior probabilities. If we include training as well then this can be written in a one-liner as `p = a*(a*qdc)*classc`. (Try to understand this expression: between the brackets the classifier is trained. The result is applied on the same dataset). The label assigned by the classifier can be found by `labeld`.

*Classifier evaluation and error estimation*

The following routines are useful to know for evaluation of classifiers:

`testc` test a dataset on a trained classifier
`crossval` train and test classifiers by cross validation
`cleval` classifier evaluation by computing a learning curve
`gendat` split a given dataset at random into a training set and a test set.
`confmat` calculate a confusion matrix for your classifier

A simple example of the generation and use of a test set is the following:
Load the `mfeat_kar` dataset, consisting of 64 Karhunen-Loeve coefficients measured for 10*200 written digits (’0’ to ’9’). A training set of 50 objects per class (i.e. a fraction of 0.25 of 200) can be generated by:

```
>> a = mfeat_kar
MFEAT KL Features, 2000 by 64 dataset with 10 classes: [200 ... 200]
>> [trainset,testset] = gendat(a,0.25)
MFEAT KL Features, 500 by 64 dataset with 10 classes: [50 ... 50]
MFEAT KL Features, 1500 by 64 dataset with 10 classes: [150 ... 150]
```

50 × 10 objects are stored in trainset, the remaining 1500 objects are stored in testset. Train the linear normal densities based classifier and test it:

```
>> w = ldc(trainset);
>> testset*w*testc
```

Compare the result with training and testing by all data:

```
>> a*ldc(a)*testc
```

which is probably better for two reasons. Firstly, it uses more objects for training, so a better classifier is obtained. Secondly, it uses the same objects for testing as well a for training, by which the test result is definitely positively biased. Because of that, the use of separate sets for training and testing has to be preferred.

*kNN and Parzen Density classifier (optional, Parzen Density very optional)*

Generate a dataset as follows:

```
a = gendatb([50, 50])
```

Make a scatter plot of dataset a. Compute the mappings of a quadratic classifier based on normal densities (`qdc`) and the Parzen classifier `w = parzenc(a)`
Plot the two classifiers on the same figure as that of the dataset.
Which of the two classifiers separates the classes better seeing from your plot? Explain that based on the data distribution and the characteristics of the classifiers.

Start a second figure and repeat the same with the k-nearest neighbor classifier with 50- (`w = knnc(a, 50)`) and optimal- (`[w, k_opt] = knnc(a)`) nearest neighbor(s), respectively. `knnc` calculates an optimal k for you. How do you think it does it? What is the value of `k_opt`? Which of the two classifiers separates the classes better seeing from your plot? Why?

*The k-means algorithm*

We will show the principle of the k-means algorithm graphically on a 2-dimensional dataset. This is done in several steps.

1. Take a 2-dimensional dataset, e.q. `a = gendatb;`. Set k=4.
2. Initialise the procedure by randomly taking k objects from the dataset:

```
>> L=randperm(size(a,1)); L=L(1:k);
```

3. Now, use these objects as the prototypes (or centres) of k centres. Defining labels 1 to k, the nearest mean classifier considers each object as a single cluster:

```
>> w=nmc(dataset(a(L,:),[1:k]'));
```

4. Repeat the following line until the plot does not change. Try to understand what happens:

```
>> lab=a*w*labeld; a=dataset(a,lab); w=nmc(a); scatterd(a);plotc(w)
```

Repeat the algorithm with another initialisation, on another dataset and some values for k. What happens when the `nmc` classifier in step 3 is replaced by `ldc` or `qdc`?
A direct way to perform the above clustering is facilitated by `kmeans`. Run `kmeans` on one of the digit databases (for instance `mfeat_kar`) with k>=10 and compare the resulting labels with the original ones (`getlab(a)`) using `confmat`.
Try to understand what a confusion matrix should show when the k-means clustering had resulted into a random labeling. What does this confusion matrix show about the data distribution?

*Clustering by the EM-Algorithm "soft k-means" (very optional)*

A more general version of k-means clustering is supplied by `emclust` which can be used for several classification algorithms instead of `nmc` and which returns a classifier that may be used to label future datasets in the same way as the obtained clustering.
The following experiment investigates the clustering stability as a function of the sample

size. Take a dataset a and compute for a given choice of the number of clusters k the clustering of the entire dataset (e.g. using `ldc` as a classifier) by:

```
>> [lab,v] = emclust(a,ldc([],1e-6,1e-6),k);
```

Here v is a mapping that by `d = a*v` 'classifies' the dataset according to the final clustering (`lab = d*labeld`). Note that for small datasets or large values of k some clusters might become too small (`classsizes(d)`) for the use of `ldc`. Instead `nmc` may be used. The dataset a can now be given the cluster labels `lab` by:

```
>> a = dataset(a,lab)
```

This dataset will be used for studying the clustering stability in the following experiments. The clustering of a subset `a1` of n samples per cluster of a:

```
>> a1 = gendat(a,repmat(n,1,k))
```

can now be found from

```
>> [lab1,v1] = emclust(a1,ldc([],1e-6,1e-6));
```

As the clustering is initialized by the labels of `a1`, the difference e in labeling between a and the one defined by `v1` can be measured by `a*v1*testc`, or in a single line:

```
>> [lab1,v1]=emclust(gendat(a,n),ldc([],1e-6,1e-6)); e=a*v1*testc
```

Average this over 10 experiments and repeat for various values of n. Plot e as a function of n.

*Support vector machines*

A simple guide to SVM classification can be found at http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf, read this cursory to learn the simplest parameters. Included in the LIBSVM package you installed is a simple example. More details about this model can be found in LIBSVM FAQ (http://www.csie.ntu.edu.tw/~cjlin/libsvm/faq.html) and LIBSVM implementation document (http://www.csie.ntu.edu.tw/~cjlin/papers/libsvm.pdf).

Train and test on the provided data `heart_scale`, and inspect the output.

The `'svmtrain'` function is called as

```
>> model = svmtrain(training_label_vector, ...
training_instance_matrix [, 'libsvm_options']);
```

with parameters

- `training_label_vector`
  - An m by 1 vector of training labels (type must be double).
- `training_instance_matrix`
  - An m by n matrix of m training instances with n features.
  - It can be dense or sparse (type must be double).

- `libsvm_options`
  - A string of training options in the same format as that of LIBSVM

and returns a `model` which can be used for future prediction. It is a structure and is organized as `[Parameters, nr_class, totalSV, rho, Label, ProbA, ProbB, nSV, sv_coef, SVs]`:

- `Parameters`: parameters
- `nr_class`: number of classes; = 2 for regression/one-class svm
- `totalSV`: total #SV
- `rho`: -b of the decision function(s) wx+b
- `Label`: label of each class; empty for regression/one-class SVM
- `ProbA`: pairwise probability information; empty if -b 0 or in one-class SVM
- `ProbB`: pairwise probability information; empty if -b 0 or in one-class SVM
- nSV: number of SVs for each class; empty for regression/one-class SVM
- `sv_coef`: coefficients for SVs in decision functions
- SVs: support vectors

If you do not use the option `'-b 1'`, ProbA and ProbB are empty matrices. If the `'-v'` option is specified, cross validation is conducted and the returned model is just a scalar: cross-validation accuracy for classification and mean-squared error for regression.

Prediction from this model can be done with

```
>> [predicted_label, accuracy, decision_values/prob_estimates] =...
svmpredict(testing_label_vector, testing_instance_matrix, model [,
'libsvm_options']);
```

- `testing_label_vector`
  - An m by 1 vector of prediction labels. If labels of test data are unknown, simply use any random values. (type must be double)
- `testing_instance_matrix`
  - An m by n matrix of m testing instances with n features. It can be dense or sparse. (type must be double)
- `model`
  - The output of svmtrain.
- `libsvm_options`
  - A string of testing options in the same format as that of LIBSVM.

```
>> load heart_scale.mat
>> model = svmtrain(heart_scale_label, heart_scale_inst, '-c 1 -g
0.07');
>> [predict_label, accuracy, dec_values] = ...
svmpredict(heart_scale_label, heart_scale_inst, model);
```

For probability estimates, you need '`-b 1`' for training and testing:

```
>> load heart_scale.mat
```

```
>> model = svmtrain(heart_scale_label, heart_scale_inst, '-c 1 -g
0.07 -b 1');
>> [predict_label, accuracy, prob_estimates] =
svmpredict(heart_scale_label, heart_scale_inst, model, '-b 1');
```

**Exercises**

*Scatterplotting*

Load the 4-dimensional Iris dataset by `a = iris` and make scatterplots of all feature combinations using the `gridded` option of `scatterd`. Try also all feature combination using `scatterdui`. Plot in a separate figure the one-dimensional feature densities by `plotf`. Identify visually the best combination of two features. Create a new dataset `b` that contains just these two features. Create a new figure by the `figure` command and plot a scatterplot of `b`.

*Simple classification*

From the dataset `b`, make a scatter plot. Compute the quadratic classifier based on normal densities (`qdc`) and plot it on top of the scatter plot of the dataset. Repeat this for the classifier assuming uncorrelated features (`udc`) and the linear classifier (`ldc`) based on equal normal distributions. These classifiers should be plotted in different colors.
Plot the underlying density estimates. Do you understand why the different classifiers are placed as they are?

*Performance evaluation*

In PRTools a dataset `a` can be split into a training set `b` and a test set `c` by the `gendat` command, e.g. `[b,c] = gendat(a,0.5)`. In this case, for each class 50% of the objects are randomly chosen for dataset `b` and the remaining objects are stored in dataset `c`. After computing a classifier by the training set, e.g. `w = b*ldc`, the test set `c` can be classified by `d = c*w`. For each object, the label of the class with the highest confidence, or posterior probability, can be found by `d*labeld`. E.g.:

```
>> a = gendath;
>> [b,c] = gendat(a,0.9)
Higleyman Dataset, 90 by 2 dataset with 2 classes: [45 45]
Higleyman Dataset, 10 by 2 dataset with 2 classes: [5 5]
>> w = ldc(b); % the class names (labels) of b are stored in w
>> getlabels(w) % this rutine shows labels (classes labels are 1
% and 2)
>> d = c*w; % classify test set
>> lab = d*labeld; % get the labels of the test objects
>> disp([+d lab]) % show the posterior probabilities and labels
```

Note that in the last displayed column (lab) the labels of the classes with the highest classifier outputs are stored. The average error in a test set can be directly computed by `testc`:

```
>> d*testc
```

which may also be written as `testc(d)` or `testc(c,w)` (or `c*w*testc`).

Generate a training set a of 20 objects per class by `gendath` and a test set b of 1000 objects per class. Compute the performance of the Linear classifier by `b*(a*ldc)*testc`. Repeat this for some other classifiers. For which classifiers do the errors on the training set and test set differ most? Which classifier performs best? Do you have any good explanation?

*Another classification exercise*

Load the IMOX data by `a = imox`. This is a feature based character recognition dataset. What are the class labels? Split the dataset in two parts, 80% for training and 20% for testing. Store the true labels of the test set using `getlabels` into `lab_true`
Compute three different classifiers on this set. Classify the test set using these classifiers. Store the labels found by the classifiers for the test set into `lab_test1` etc
Display the true and estimated labels by `disp([lab_true lab_test])`
Predict the classification error of the test set by observing the output. Verify this number using `testc`.

*NIST Digit clustering*

Load a dataset A of 25 NIST digits for all classes 0-9. (Subsample the `nist16` set)
Compute the 7 Hu moments. Perform a cluster analysis by `kmeans` with `k = 10` neglecting the original labels. Compare the cluster labels with the original labels using `confmat`.

The seven Hu invariants (Hu moments) can be calculated by:

```
% Mass and center
[c, mass] = masscenter(F);
% Central moments
mu20 = mupq(F, 2, 0, c);
mu11 = mupq(F, 1, 1, c);
mu02 = mupq(F, 0, 2, c);
mu30 = mupq(F, 3, 0, c);
mu21 = mupq(F, 2, 1, c);
mu12 = mupq(F, 1, 2, c);
mu03 = mupq(F, 0, 3, c);
mus = [mu20 mu11 mu02 mu30 mu21 mu12 mu03];
% Invariants
[sims,orthos,simorthos]= HUinvariants(mass, mus);
```

*Color image segmentation by clustering*

A full-color image may be segmented by clustering the color feature space. For example, read the famous (in pattern recognition) Len(n)a image in a 256 × 256 version

```
>> a=lena;
>> show(a)
```

The image may be reconstructed as a full colour images by:

```
>> figure; imagesc(reshape(+I,256,256,3));
```

The 3 colours may be used to segment the images on its pixel values only. We use a small subset for finding 4 clusters in the 3d color space:

```
>> testset=gendat(a,500) % create small test set
>> [d,w]=emclust(testset,nmc([]),4) % cluster the data
```
The retrieved classifier w may be used to classify all image pixels in the colour space:

```
>> lab = classim(a,w);
>> figure
>> imagesc(lab) % view image labels
```

Finally we will replace each of the clusters by its colour mean:

```
>> aa=dataset(a,lab(:)) % create labeled dataset
>> map=+meancov(aa) % compute class means
>> colormap(map) % set colour map accordingly
```

Note that the mean colours are very equal. Try to improve the result by using more clusters.

Remember color dataset is three-dimensional. You can visualize your clustering, and the original data by using `scatterd`

*Texture segmentation (optional)*

A dataset a in the MAT file `texturet` contains a 256x256 image with 7 features (bands): 6 were computed by some texture detector; the last one represents the original gray-level values. The data can be visualised by `show(a,7)`. Segment the image by `[lab,w] = emclust(a,nmc,5)`. The resulting label vector `lab` may be reshaped into a label image and visualised by `imagesc(reshape(lab,a.objsize))`. Alternatively, we may use the trained mapping `w`, re-apply it to the original dataset a and obtain the labels by `classim`, and display as an image, `imagesc(classim(a*w))`.
Investigate the use of alternative models (classifiers) in `emclust` such as the mixture of Gaussians (using `qdc`) or non-parametric approach by the nearest neighbour rule `knnc([],1)`. How do the segmentation results differ and why? The segmentation speed may be significantly increased if the clustering is performed only on a small subset of pixels.

*SVM and Digit Recognition I*

To generate data for the exercise you will use the function `SURF_LightB.m` and `LoadImsComputeSURFB.m` from the folder prdatasets. The function `SURF_LightB.m` takes a parameter value `ng` that defines the size of the grid. Thus the image is split into `ng` × `ng` equally sized regions. Secondly, the image pixel data is scaled to be between `0` and `1`.
Upon starting Matlab add the directory containing the digit figures and the directory containing the SVM code to your path. Extract the feature vectors from the training and test data:

```
>> [X train, lab train, X test, lab test] = LoadImsComputeSURFB(7);
```

To begin with we will construct a classifier to separate the digit “0” from the other digits. Construct the indicator vector `tt`.

```
>> nt = size(X train, 1);
>> I = 1:nt;
>> dig = 0;
>> ind = I(lab train == dig);
>> tt = -1*ones(1, nt);
>> tt(ind) = 1;
```

The feature vectors should also be scaled so that each value is roughly between -1 and 1.

```
>> X_train = X_train / 400;
>> X_test = X_test / 400;
```

First we will train a linear SVM. Set the parameters for the linear SVM and use the data X train to train it. Note we have to transpose X train as the function SVMTrain expects each column of the data array to correspond to a feature vector.

Set parameters for a linear kernel, with cost 1 and shrinking heuristics.

```
>> params = '-s 0 –c 1 –t 0 –h 1 ';
>> model = svmtrain(X_train', tt, params);
```

Next use the function `svmpredict` to evaluate the SVM you've just trained on the test data `X_test`:

```
>> [TestLabels, accuracy, prob_estimates] = ...
svmpredict(X_test',[],model);
```

Note that if we had given something input as test label we would get accuracy output. Calculate the true positive and the false positive rate.

```
>> n test = length(lab train);
>> Ia = 1:n test;
>> pind = Ia(lab test == dig);
>> n tp = sum(TestLabels(pind) == 1);
>> tpr = n tp / length(pind);
>> nind = Ia(lab test _= dig);
>> n fp = sum(TestLabels(nind) == 1);
>> fpr = n fp / length(nind);
>> disp([tpr, fpr]);
```

Record these numbers and comment on the performance.

*SVM and Digit Recognition II*

As you probably noted the linear SVM did not produce good results. Next we'll try applying a kernel-SVM classifier where you'll use a Gaussian/Radial basis function as a kernel. Train and apply it by running the following commands; RBF kernel, $\gamma$=1, cost 1 and shrinking heuristics

```
>> params = '-s 0 –c 1 –t 2 –g 1 –h 1 ';
>> model = svmtrain(X_train', tt, params);
```

```
>> [TestLabels, accuracy, prob_estimates] = ...
svmpredict(X_test',[],model);
```

Compute and record the true positive and false positive rate in this case.

At this stage you're probably wondering why the SVM is not producing better results. Don't worry you are just encountering one of the weaknesses of kernel-SVM. There are two parameters which have to be set in our current set-up and these are vital to the performance of the resulting classifier. First there is the width of the Gaussian kernel $\gamma$:

$$K(x_i, x) = e^{-\gamma||x_i, x||^2}$$

Secondly there is the value of the penalty term (C in the lecture) associated with the misclassification of a training example included in the cost function the SVM training process minimizes. A large penalty forces the SVM to misclassify as few of the training examples as possible and may result in choosing a separating hyperplane with small margin and these often do not generalize well.
While too small a penalty term results in a classifier with a large margin but that does not discriminate between the two classes. Unfortunately, there is no good rule of thumb for setting these two parameters and in general some form of exhaustive search is performed over the space of (C, $\gamma$) values. Luckily, someone have performed a coarse exhaustive search on your behalf. Thus run the training and testing process again. But this time set the parameters such that

```
>> params = '-s 0 –c 2 –t 2 –g 4 –h 1 ';
```

that is with $\gamma$ =4 and C=2. Record the true positive and false positive rate is this case.

Repeat the whole process for learning a SVM to discriminate between the digit "8" and the other digits. Note in this case set $\gamma$ =8 and C=2. Record the true positive and false positive rate.

*SVM and Digit Recognition III (very optional)*

Some of you may be interested in how I learned the values for $\gamma$ and C. This was done by *k*-fold cross validation and here is its explanation.

First split the set of positive training examples into k subsets of equal size such that each example appears in only one subset. Similarly split the set of negative training examples into k subsets of equal size. Fix the values of $\gamma$ and C. Use the first *k −1* subsets of the positive and negative examples to train an SVM. Apply this learned SVM to the kth subset of positive and negative examples that you omitted from the training process and record the number of true positives and false positives. Then retrain the SVM using all the training data minus the *(k − 1)th* subset. Use the retrained SVM on the *k − 1* subset and record the number of true positives and false positives and add these to the numbers calculated for the previously tested SVM. Repeat the process with the *(k − 2)th* subset omitted, then the (k−3)rd and so on until you've omitted the *1st* subset. Record the total number of true positives and false positives you have found and use some combination of these two numbers to obtain a score. (Here was used *(number of true positives)- (number of false positives)*). This score process represents how good the parameter settings of and C were. Repeat the process for different settings of and C. Then choose the setting which produced the best score. Typically a loose grid search is performed with C=$2^{-5}$, $2^{-3}$, ... , $2^{15}$ and $\gamma$=$2^{-15}$, $2^{-13}$, ... , $2^{3}$. That is you perform the process just described for

$(C, \gamma) = (2^{-5}, 2^{-15}), (2^{-5}, 2^{-13}), (2^{-5}, 2^{-11}), \ldots, (2^{-5}, 2^{3}), (2^{-3}, 2^{-15}), \ldots, (2^{15}, 2^{3})$

and then when this search indicates a good region in the (C, $\gamma$) space a finer grid search is conducted centered around this region. The values of C and quoted in the previous exercise were found using only a coarse search in the (C, $\gamma$) space. You can try and improve upon these values by performing a finer search around them.

(Don't let $\gamma$ be set too much higher than 8. High values of $\gamma$ may produce good results on the training data but the learned classifier doesn't generalize so well. Why?)