This course has been developed in the framework of the ITEMS master program (Techniques for Analysis, Modeling and Simulation for Imaging, Bioinformatics and Complex Systems) financed through project POSDRU/86/1.2/S/61756.

Workbook Pattern Recognition I

An Introduction for Engineers and Scientists

C. Rasche

Abstract

The purpose of this workbook is to provide a practical access to the topic of pattern recognition. The emphasis lies on applying and exploring the statistical classification methods in Matlab directly. Plenty of code examples are given to play with these methods immediately. We start with the very simple and easily implementable k-Nearest-Neighbor classifier, followed by the popular and robust linear classifiers. We learn how to apply the principal component analysis (PCA) and how to properly 'fold' the data. We further introduce clustering methods, decision trees, ensemble classifiers and string matching methods. Eventually, we also mention Support Vector machines and the Naive Bayes classifier. The latter helps us to understand some of the theoretical aspects, e.g. the Bayesian formulation for classification.

Prerequisites: basic programming skills
Recommended: basic linear algebra, basic signal processing

Contents

1 Introduction

There are many wonderful textbooks on the subject of pattern recognition, but they often lack the exemplary approach, meaning the learning-by-doing approach (though I have not read all textbooks on this topic). Textbooks often provide the theoretical background first, followed by giving examples; and the practical tips are often lost in the text somewhere. But the theoretical background is easier to understand if one has worked through some specific examples. We therefore provide here an example-first approach - with its practical tips made explicit, thereby encountering the classifiers' advantages and disadvantages in practice. After having worked through these examples, any textbook should read fairly easy.

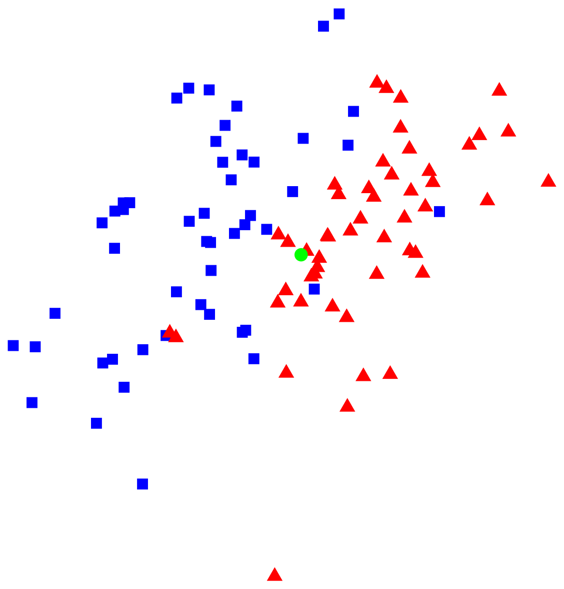


Figure 1: Illustrating the classification problem in 2D. We are given two sets of points representing two classes (squares and triangles, respectively) - they are our training samples (example data). To which group would we assign a new sample (testing point) such as the one marked as a circle?
The two classes may overlap due to measurement noise or because some samples are indeed a mixture of both classes - nevertheless, we would like to predict a new sample as well as possible.
In the simplest case we compare all training samples with the testing sample (section 2). Or we may attempt to model the point clouds with functions (like a Gaussian function; section 12). We could also find a straight line equation which separates best the two point clouds (section 3). Of course, each method has its advantages and disadvantages - there is no 'best classifier'.

1.1 The Two Principal Recognition Challenges **wiki: Machine_learning**

The two most common recognition challenges are *classification* and *clustering*. In classification, we try to get novel data discriminated by using example data that we have collected and analyzed before. For instance, we collect handwritten digits of several tens of persons, label the digits, then build a model to discriminate the digits, and then apply the model on a large scale (it is impossible to obtain the handwriting of each person on earth). In clustering we try to find trends in data for which we do not know their classes: if you were to learn the Chinese symbols (without translation) it could be beneficial to cluster the symbols in order to find those that are similar and possibly share meaning (this hypothetical example is given to contrast the example for classification). More elaborately:

Classification: For this task we collect some data and label them. For example in figure 1 we have points visually labeled as squares and triangles, that is we have two classes (categories, groups). We then create a model that separates the two classes. This will allow us to test new data, such as the round point: does it belong to the class marked with squares or to the one marked with triangles?

To create the model we use so-called *supervised* learning algorithms that particularly exploit the label information. For instance we can take the mean for each class and calculate the difference of a new data point to the corresponding two means: the distance would be a measure for decision making. One could say, training takes place with help of a 'teacher'. We will first treat this supervised learning (sections 2 and 3).

Clustering: Here we are given data and we are trying to make sense of them. For instance in figure 8 (section 6.1) there are points but we lack class information: it appears there exist two 'clusters'; maybe there are three? For example, a company/institution/organization tries to recognize trends in order to adjust their strategies or to react better. Because there is no 'teacher', it is said that the clustering algorithms perform *unsupervised* learning. We introduce clustering a bit more in subsection 6.1.

There are also other challenges, such as regression and reinforcement learning, but they are more rare, so we concentrate on the two principal challenges now.

1.2 Data Format - Formalism

Data often consist of samples or also called observations, which were collected with the same number of measurements. For instance in computer vision, images are samples, the number of pixels are the measurements. And because the number of measurements is often the same, one can conveniently represent the data in a 2D array or a $n \times d$ matrix D, which is organized as follows:

n samples	$\times$	d dimensions
observations	$\times$	features (or components or variables)
rows	$\times$	columns
left; vertical	$\times$	horizontal; top

whereby we listed different terms as well as some memorizing hints.

Each row describes a data sample (or observation), that is a vector $\mathbf{d}$, of which each dimension $\mathbf{d}(j)$ (or component) - sometimes also denoted as d_j - represents the measurement of a different *feature* (or variable), $j = 1..d$. Thus, each sample is a point in d-dimensional space; in figure 1, it is a two-dimensional space only. In praxis, the dimensionality can range from two to several thousand dimensions. In computer vision for example, often the image's individual pixels are taken as dimensions, that is for a 200x300 pixel image we have 60'000 dimensions (or features, variables,...). In Bioinformatics the dimensionality can also easily grow to several thousands, in particular in DNA microarray analysis. In Webmining as well, one easily reaches several thousands of dimensions.

In books and in software programs often the mathematical notation $[m \times n]$ matrix X is used, with m and n denoting the dimensionality of the matrix (n and d above); i and j are the preferred letters to denote indexing, thus $x_{i,j}$ is the matrix entry (value) at the ith row and the jth column. Y is often used to denote the label vector - whereas here we prefer G. However the notation is not consistent across books and we have to be careful when interpreting the equations.

Note: Sometimes the term 'feature vector' is used in the literature, which stands for 'a sample' (and not for a column, even though the name suggests it).

Should the data have been measured with different features for different samples, then the array contains misses essentially and it is most convenient to fill in NaN entries (not-a-number), in order to obtain a uniform dimensionality. Software programs can deal with NaN entries.

1.3 The Essence of Evaluating a Classifier Model

When we train a classifier, we would like to know how well our classifier will perform on new data, in other words, we would like to know its *generalization* performance when it is given untested (unknown) data. Or expressed in the terminology of statistics: we would like to know how accurately it predicts. In case of the (handwritten) digit classification task, this would give us an estimation of how many times the system errs when tested on a large crowd of people.

To evaluate properly, we partition our data set into a training set, that is used exclusively for training, and a testing set that is used for estimating the generalization performance. For example we divide our data set (the matrix D) in two equal halves. We train the classifier on one half and then test it on the other half. Then we swap the two halves and perform training and testing again to obtain a second performance estimate. We take the mean of the two estimates and have so obtained a rough but robust estimation. This is also called *hold-out estimation* or *two-fold cross-validation*. There are of course more refined methods than just using two halves (section 5).

Throughout the book we use $\mathcal{D}^L$ to denote the training set and $\mathcal{D}^T$ our testing set (note the font style of $\mathcal{D}$). In our code we use the variable names `TREN` and `TEST`, respectively (or `TRN` and `TST` only). We use G to denote the vector with group labels, in Matlab we prefer `Grp`.

1.4 Varia

Source for this Workbook A few text passages are copied/pasted/modified from various textbooks, as well as some of the figures - I have tried to compile the best pieces from each book and provide exact citations including page number. See appendix B.4 for titles. Our workbook distinguishes itself from the textbooks by specifying the algorithmic formulation more explicitly and by providing code, which is optimized for the use in a high-level language.

Code The code fragments I provide are written in Matlab; in other languages most of the commands are called the same or at least similar. The computations are written in *vectorized* form, which is equivalent to matrix notation: this type of vector/matrix thinking is unusual at the beginning, but highly recommended for 3 reasons: 1) computation time is shorter than using for-loops; 2) code is more compact; 3) code is less error-prone. However, the code fragments may contain unintended mistakes, as I copied/pasted them from my own Matlab scripts and made occasionally some unverified modifications for instruction purposes.
It can also be useful to check Matlab's file exchange website for demos of various kinds:
`http://www.mathworks.com/matlabcentral/fileexchange`

Advice We recommend implementing the simple classifier types by oneself, for instance in a high-level programming language such as Matlab. This can be done with a few lines. For more complex classifiers it is more convenient to employ existing routines (such as the Linear Discriminant Analysis and the Support Vector Machines). Why then would one want to implement the simple classifiers at all? There are several reasons. One is, that the existing routines sometimes do not account for special data entries, e.g. NaN (not a number) or are not optimized for large datasets. Another reason is that one may intend to build individual classifiers, e.g. ensemble classifiers for which it may be more convenient to write one's own code. Furthermore, by writing our own code we know exactly what parameters/conditions etc we have used. Finally, it is part of the learning process and one gains confidence if we use classification packages.

Testing Data Sets It is instructive to start with a toy data set with two dimensions only (see later), and then to approach higher-dimensional sets. A convenient way to practice is to use the data set of handwritten digits, `http://yann.lecun.com/exdb/mnist/`. Bishop provides also other collections, Bis p677:
`http://research.microsoft.com/en-us/um/people/cmbishop/PRML/webdatasets/datasets.htm`

1.5 Exercise

To get acquainted with Matlab we study an example of how to create a synthetic data set, see appendix C.1. In short, the example generates a synthetic data set of two classes and its corresponding group labels in the first section, then plots the data into figure no. 1. Then the data are split into a training and a testing set, matrices `TRN` and `TST` respectively, and are then plotted into figure no. 2.

1. Study exactly each command: one can look up information about a function by either using the command `help` or `doc` in the command window. For example looking up the plot command is done as follows: `help plot` [return] or `doc plot`.
2. What is the output range of `rand`? Try Gaussian (normal) noise using `randn` (n for normal). What is the output range for `randn`?
3. Add a third class of points, meaning add a vector `D3` and its corresponding `G3`. Plot the third class into figure 1 as well.
4. Every time you re-run the script, a new set of random coordinates is generated. This can be annoying if you are trying to verify code improvements. You can fix the random state to the same seed: `rng('default');`.
5. Type `who` to see your variables in workspace and study the different data types, for instance, 'double', 'single', etc. This is important to know because your data may come for instance in 'uint8' - look it up. Use `doc datatype` to obtain an overview of datatypes.
6. Familiarize with the way vectors and matrices are initialized and multiplied, see appendix B. If you had already linear algebra, it is nevertheless useful to be aware of the pitfalls of 'unconventional' initialization of vectors.

1.6 Load Your Real Data

Before you attempt to classify your data, it can be useful to prepare your data in a separate script and save the processed data in Matlab format. It is all the more recommended if you have multiple files. In this preparation step you convert your data to your preferred formant and you can also determine the range of variables in order to perform proper normalization. There's no general scheme of course, because each data set is individual. In the following are some tips how to organize your work and how to load the data:

Organize It is useful to create the following folders to organize your data and scripts:

- *DatRaw*	place your downloaded data into that folder.
- *DatPrep*	where the processed data will be saved.
- *Classif*	matlab scripts for classification and data manipulation.

Open a script called `PrepRawData` to prepare your raw data. You will load the data, convert them and then save them:

Loading Data Most files can be loaded with the command `importdata`. Should the function be insufficient, due to lack of specificity for example, then one has to start looking at commands such as `textscan`, `textread` etc., see also the section entitled 'See Also' at the end of each help document. For images there exists the special command `imread`. If all fails, it is not a shame to ask a system administrator to explain to you how to read your data. Some formats can be indeed tricky.

Data Preparation Assign your data to a matrix called `DAT` of format [Samples x Dimensions] for rows and columns (see again subsection 1.2). If you have many files, you may want to initialize the matrix beforehand in order to speed up the preparation step, e.g. `DAT = zeros(nSmp, nDim, 'single');`.

Initialize a global structure variable, which contains the path names to those folders, e.g.: `FOLD.DatDigits = 'C:/Classification/Data/Digits';`. With the command `dir` you can obtain a list of all images, e.g.

`ImgNames = dir(FOLD.ImgSatImg)` and then access the filenames `FileNames(1).name`. The first two entries will contain '.' and '..', thus start with `FileNames(3)`. Use `fileparts` to separate the path into its components.

If you need high computing precision, use *double* type, instead of *single* as just exemplified. Matlab does everything in double by default, but double requires also twice as much memory. Hardware memory may not be the issue, but RAM is often the limiting memory. In most classification tasks, the single datatype is sufficient.

Grouping Variable Prepare also your group variable `Grp`, the class/category labels for each sample. If your labels are not numerical, for instance consisting only of values between 1 and the number of classes, it is recommended that you use the command `grp2idx` to convert your labels into that format - it will facilitate later the classification procedure. Although Matlab is relatively flexible with labels, for instance allowing string labels, it becomes obscure to deal with this unless one enjoys the combination of flexibility and elegance.

Appendix C.3 gives an example of how to load and convert. It is a function, which returns the data already partitioned as training and testing set with the corresponding grouping variables called `Lbl`.

Saving Data - And Reload Later Saving the data is simply done by the Matlab command `save`. This will save the data in Matlab's own format, which is a type of compression format. When you reload the data in your classification script you use simply `load`.

Appendix C.2 shows codes fragments to understand how to program the individual steps.

2 k-Nearest Neighbor (kNN)

The Idea: The k-nearest neighbor algorithm is amongst the simplest of all machine learning algorithms. Given a testing set, we simply store all its samples as an exhaustive reference, quasi as a library. To classify a testing sample, we compare it to all the training samples and observe which class is the most frequent in a certain neighborhood. Hence, no real abstraction of the training samples is sought; no actual learning takes place. We only measure distances and look at the nearest neighbors, the k nearest neighbors.

> **Figurative Example.** Determining the country by looking at license plates: You drive across Europe and determine which country you currently drive through by looking at the cars' license plates. If there is a majority of license plates for one type of country, then it is likely you are currently in that country. In regions near country borders and near tourist resorts, this probability decreases.

The Procedure: Given is a training set, a matrix `TRN` with corresponding group (class) labels in vector `GrpTrn`, and a testing set, a matrix `TST` with corresponding `GrpTst`. To classify a sample from the testing set (one row vector of `TST`), we measure the distance to all samples in `TRN`, resulting in a vector `Dist` of length `GrpTrn`. We order the distances in `Dist` and choose the closest training sample and take its category label as the label of the testing sample - that would be the nearest neighbor, meaning $k = 1$. We can also look at more (nearest) neighbors, e.g. 3, 5, ...and determine which category label occurs the most amongst those k neighbors (for even k we may face parity).

In other words, a testing sample is classified by assigning it to the most frequent class label amongst its neighborhood of size k in `TRN` (figure 2). One can try different distance metrics, e.g. Euclidean, Manhattan,...(see also Appendix A.1). There is essentially no initialization required with exception of the possible need to normalize the data.

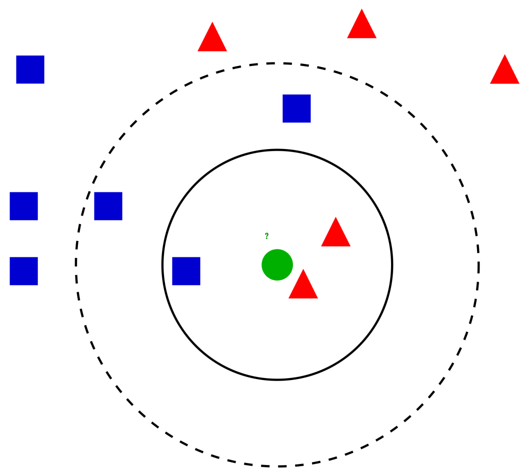


Figure 2: k-Nearest-Neighbor (kNN). Given are 11 training samples from 2 classes (marked as squares and triangles); 1 instance (testing sample marked as filled circle) is to be classified. Solid (thin) circle: 3NN; stippled circle: 5NN.

Algorithm 1 kNN classification. $\mathcal{D}^L$=`TRN` (training samples), $\mathcal{D}^T$=`TST` (testing samples). G vector with group labels (length = $\mathrm{n}_{\mathrm{TrainingSamples}}$).

```
Initialization  normalize data
Training        training samples $\mathcal{D}^L$ with class (group) labels $G$.
                    (In fact, no actual training takes place here)
Testing         for a testing sample ($\in \mathcal{D}^T$): compute distances to all training samples $\rightarrow D$,
                    rank (order) $D \rightarrow D^r$
Decision        observe the 1st $k$ (ranked) distances in $D^r$ (the $k$ nearest neighbors):
                    e.g. majority vote of the most frequent class label of the kNN determines category label
```

2.1 Implementation

Matlab offers the `knnclassify` command (as part of the bioinformatics toolbox), but coding a kNN classifier is fairly easy. Here are some fragments to understand how little it actually requires (see also ThKo p82):

```
%% --- Knn classification
nCls    = 2;                             % # of classes
nTrn    = size(TRN,1);                 % # of training samples
nTst    = size(TST,1);                   % # of testing samples
[GNN] = deal(zeros(nTst,11));            % we will check out 11 nearest neighbors
for i = 1:nTst
    iTst = repmat(TST(i,:), nTrn, 1);    % replicate to same size [nTrn nDim]
    Diff = TRN-iTst;                     % difference             [nTrn nDim]
    Dist = sum(abs(Diff),2);             % Manhattan distance     [nTrn 1]
    [dst ix]  = min(Dist);               % min distance    for 1-NN
    [~, O] = sort(Dist,'ascend');       % increasing dist for k-NN
    GNN(i,:)  = Grp.Trn(O(1:11));       % closest 11 samples
end

%% --- Knn analysis quick (for 5 NN)
HNN         = histc(GNN(:,1:5), 1:nCls, 2); % histogram for 5 NN
[Fq LbTst]  = max(HNN, [], 2);              % LbTst contains class assignment
Hit         = LbTst==Grp.Tst;
fprintf('Perc correct for 5NN %1.2f\n', nnz(Hit)/nTst*100);
```

See also the progamming hints in subsection A.3 for why we chose a for-loop in this case.

2.2 Evaluation

Optimal k We would like to know for which k nearest neighbors, we achieve the best classification results. Only systematic testing allows us to find the optimal number of k. In praxis, often $k = 1$ or $k = 3$ is sufficient, but one may also want to check larger neighborhoods. Appendix C.5 contains an example of how to analyze a range of different ks.

2.3 Normalization ThKo p263, s5.2.2, pdf 276

The range of values for different features may vary significantly. It could therefore be beneficial to normalize your data. There are different possibilities to perform the normalization, for instance:

1. by dividing the feature values by the mean and standard deviation (for that feature). The resulting normalized features will now have zero mean and unit variance. Matlab: `zscore`.
2. by limiting the feature values in the range of [0, 1] or [-1, 1] by proper scaling.
3. by scaling the feature values by an exponential or tangent function (e.g. `tanh`).
4. by performing a whitening transformation (DHS pp 34, pdf 54). This is a decorrelation method in which we multiply each sample by the covariance matrix of the dataset. The method is called ”whitening” because it transforms the input matrix to the form of white noise, which by definition is uncorrelated and has uniform variance (see subsection A.2 for details).

Note 1: To estimate generalization performance accurately, one should determine the scaling parameters for the training set only and then normalize training and testing set separately, e.g.

```
    [TRN Mu Sig] = zscore(TRN); % normalization and obtaining mean and standard deviation
    DF           = bsxfun(@minus, TST, Mu);  % normalization testing set
    TST          = bsxfun(@rdivide, DF, Sig);
```

Note 2: Normalization may distort the relations between dimensions and hence the distances between samples. Therefore, normalization does not necessarily improve classification (or clustering). It may be useful to look at the distribution of individual features (e.g. using a plotting command such as `hist`) too see what type of normalization may be appropriate.

2.4 Division by Zero, Infinity (Inf), Not a Number (NaN)

Often, some of the data contain useless or missing values. For instance, some dimensions may contain only zero values; or the feature extraction program may have returned a 'NaN' entry (not a number) or an 'Inf' entry (infinity). Here is how Matlab deals with that:

- Division by zero:returns a division-by-0 warning and creates an

  Inf entry, if the divisor (denominator) is 0;
  NaN entry, if both divisior and dividend (numerator) are 0.
- Any operation with a NaN or Inf entry remains or produces a NaN or Inf entry.

Because most classifiers will use multiplication operations, entries with NaN or Inf values can render results useless. Matlab classification functions typically take care of this. As a programmer you may want to eliminate dimensions with zero entries immediately and/or use the nan-commands, `nanmean, nanstd, nancov,...` to deal with NaN entries. To avoid the creation of Inf entries, one can add the smallest value possible (`eps` in Matlab) to a divisor, e.g. try `1/(0+eps)`, which so will use the largest value possible, thus permitting to further operate with the variable (as opposed to an Inf entry).

2.5 Recapitulation

Advantages
- With the nearest-neighbor classifier we obtain robust results with an easily implementable model.
- The kNN classifier even works when only few training samples are available, for instance $n < 5$ per class, for which other classifiers do not work well.

Disadvantages Classification duration can be slow if dimensionality d and/or training set n is large. The classifier has therefore $O(dn)$ complexity. We explain the meaning of this so-called 'Big-O' notation in section 5. To alleviate that problem a number of improvements have been suggested, such as partitioning the space into grids (see course II).

2.6 Notes

- Even though the kNN may not provide the best performance, it can serve as a comparison for other classifier performances. If we do not obtain a better performance with more complex classifiers, we should consider the possibility that we may not have applied the complex classifiers properly. Thus, in any case, the kNN performance can serve as a check.
- The kNN classifier does not have an actual learning process, that is, no effort was made in abstracting or manipulating the data to derive a simple decision model. In fact, we have implemented a decision rule only and nothing more.

2.7 Exercise

Synthetic Data To warm up, we get the classifier first running on the synthetic data set (appendix C.1) in a 'testing' script called `TestKnn` for instance.

1. Append the commands in subsection 2.1 to the synthetic data (of appendix C.1).
2. Try a different metric, e.g. the common Euclidean distance, in our example:
   `Dist = sqrt( sum(Diff.^2,2) );`. Do the results change a lot?
3. Take 3 classes again - if you haven't already.
4. Try different types of normalization. Do the results change a lot?

5. Swap your partitions, meaning obtain another classification estimate by exchanging training and data set. It is only now that you have performed the 2-fold cross-validation.

6. Insert some NaN values (not-a-number) somewhere to observe the resulting effects, e.g. `D1(3,1) = nan;` in the synthetic data set. Deal with this using the commands `nanmean, nanstd, nancov,...` .

7. Increase your sample sizes to hundreds of thousands. Watch out: plotting several thousands of samples may take some while, thus plotting is better turned off, e.g. use an if-then statement around the plotting section to prohibit plotting. Observe your RAM: in windows open the task manage with Ctrl-Alt-Delete and click on the tab labeled 'Performance'. It is always good to keep an eye on it.

Real Data

1. Open a new script called `ClassifyKnn` (leave your test script as is, so you have a functioning 'reference'. Load your real data, for instance `load('DatPrep')` (see subsection 1.6).

2. Inform yourself about your data by displaying simple statistics, see appendix C.4 for code.

3. Inform yourself about your groups (categories, classes): make a histogram, that counts the number of samples for each group (command `histc`). If one category had only 5 samples, what would be the maximal, feasible number of k one could test?

4. Fold your data: Create a simple 2-fold partition for each class. First you need to obtain the indices for each class. If your group variable contains positive integer values with the maximal value equal the number of classes you can use a single line,

```
IxClass = accumarray(Grp,1:nSmp,[nCat 1], @(x) {x})
```

   where `Grp` is a column vector, `nSmp` the number of samples (length of `Grp`) and `nCat` is the number of classes. If that does not work you need to write a loop. Then, for the indices of each class, `IxClass1`, `IxClass2`, etc., you apply `crossvalind` as introduced previously.

5. Out of Memory: If the dataset creates out-of-memory error notifications, try using datatype 'single', which uses half as much storage space (see subsection 1.6). Calculate the required datasize for single using the utility function `f_GbSingle`, see appendix C.10; for double you multiply by 2.

6. Then normalize your data and display the same data statistics again to verify.

7. It is recommended to create frequent checks with the command `assert`.

3 Linear Classifier (I)

A linear classifier tries to separate the classes by use of a suitable line (or boundary) between the classes. A sample point would then be classified by determining on which side of the boundary it lies. Taking the data set in figure 1, a linear classifier essentially tries to place a straight line through the two points clouds such that it separates the two classes optimally in a statistical sense.

The line equation $y = mx + c$ is sufficient to understand the principle: we attempt to find a suitable m (slope) and c (offset) such that the x for one class are well separated from the x of another class. For 3 dimensions, we attempt to find a plane; for 4 or more dimensions we talk of hyperplanes. The line/planes represent the so-called *decision boundary*. To decide the category type of a sample point, we determine on which side it lies of the decision boundary.

Figurative Example. In our country-guessing example, a linear classifier would attempt to estimate the country borders and take those as a decision boundaries for making our best country guess.

In the terminology of pattern recognition terminology we speak of *weights* instead of slope and offset, and both are lumped into a weight vector $\mathbf{w}$. To explain now the classification procedure in more detail, we start with a binary classification task (2 classes only), and then elaborate on the multi-class procedure:

Binary classification (2 classes): In this case, the model can be depicted as in figure 3. Given an input vector $\mathbf{x}$, each component $\mathbf{x}(i)$ (sometimes also denoted as $\mathbf{x}_i$) is multiplied by a corresponding weight value $\mathbf{w}(i)$ (or $\mathbf{w}_i$), which represents a weight vector $\mathbf{w}$ (whose components represent the hyperplane parameters):

$$g(\mathbf{x}) = \sum_i x(i)w(i) \equiv \mathbf{x} \cdot \mathbf{w} \equiv \mathbf{x}^t\mathbf{w}. \tag{1}$$

g is also called the *discrimination function* and in this case g is simply the dot product (or scalar or inner product). If this notation is unfamiliar to you, then study appendix B - it was part of the exercise in the introductory section.

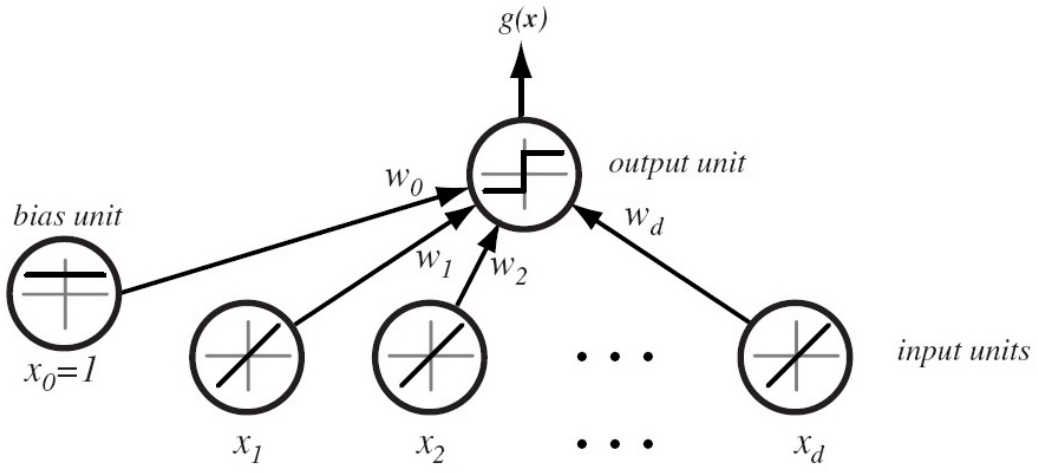


Figure 3: A simple linear, **binary** classifier having d input units, each corresponding to the values of the components of an input vector. Each input feature value x_i is multiplied by its corresponding weight w_i; the effective input at the output unit is the sum all these products, $\sum w_i x_i$. We show in each unit its effective input-output function. Thus each of the d input units is linear, emitting exactly the value of its corresponding feature value. The single bias unit unit always emits the constant value 1.0. The single output unit emits $a + 1$ if $\mathbf{w}^t\mathbf{x} + w_0 > 0$ or $a - 1$ otherwise. [Source: Duda,Hart,Storck 2001, Fig 5.1]

The dot product represents already the classification principal of the model - it is not really different from the line equation. To decide the class label now, we observe the sign of the scalar value $g(\mathbf{x})$: a positive value means the sample belongs to one class, a negative value means it belongs to the other class.

Multiple-Class Classification: For classification task with multiple classes, there is a weight vector $\mathbf{w}_k$ (of length d) for each individual class k. Those vectors are concatenated and expressed as a $k \times d$ weight matrix $\mathbf{W}$, whose size explicitly expressed is [number of classes x number of dimensions]. The classification procedure then consists of two steps: one step is the computation of 'posterior' (confidence) values for each class,

$$g_k(\mathbf{x}) = \mathbf{x}^t \mathbf{W} \tag{2}$$

The notation has seemingly not changed much in comparison to the dot product (eq. 1), but is here a matrix product, which results in an *array* g_k of length k, that is number of classes. In a second step, we chose the most likely category by finding the class, whose output is largest:

$$\text{argmax}_k g_k. \tag{3}$$

'arg' stands for argument, meaning the index of where in the array g_k the maximum occurs. In Matlab this is included in the command `max` by specifying a second output argument: `[vl ix] = max(Post)`.

Variants: The above equations represent only a principle and there exist many variants but all linear classifier models contain at their heart the matrix product between a testing sample and a weight matrix of corresponding dimensionality. Most modern linear classifier models also analyze the dependence between the feature variables using either the covariance matrix directly, or a similar analysis. The covariance matrix will be introduced in the following subsection 3.1. Famous - and by now venerable - linear classifiers are the Perceptron (workbook II) and the Naive Bayes classifier (section 12).

Learning: The challenge is of course to find the appropriate weight values which would best separate the classes. Mathematically speaking - and formulated for a two-class (binary) problem, we deal with a *linear programming problem* because trying to find the discrimination functions $g_i(\mathbf{x})$ is dealing with a set of linear inequalities: $\mathbf{w}^t \mathbf{x}_i > 0$. There exists a large number of methods to solve such inequalities of which many belong to two important categories: *gradient descent* procedures and *matrix decompositions* methods. We do not elaborate on these methods at this point, but merely explain how one matrix decomposition is implemented to find the appropriate $\mathbf{w}$ in the following subsection 3.2. In workbook II, we will also look at gradient descent procedures in more detail, namely in the Perceptron section. In section 12, we explain how $\mathbf{w}$ is estimated in a crude but straightforward manner.

Building and applying a linear classifier can be summarized as follows:

Algorithm 2 Linear classifier principle. $k = 1, .., n_{\text{classes}}$. $\mathbf{W}_{k \times d} = \{\mathbf{w}_k\}$, G vector with class labels.

```
Training    find optimal weight matrix $\mathbf{W}$ for $g_k(\mathbf{x}) = \mathbf{x}^t \mathbf{W}$ exploiting $G$    $(\mathbf{x} \in \mathcal{D}^L)$
              using a learning algorithm, covariance analysis or a combination thereof.
Testing     for a testing sample $\mathbf{x}$ determine $g_k(\mathbf{x}) = \mathbf{x}^t \mathbf{W}$                 $(\mathbf{x} \in \mathcal{D}^T)$
Decision    chose maximum of $g_k$: $\text{argmax}_k \; g_k$
```

3.1 Covariance Matrix $\boldsymbol{\Sigma}$ wiki: Covariance, Covariance_matrix

The covariance matrix expresses to what degree the individual variables (or features or dimensions) depend on each other. For a single variable, the variance measures how much its values are spread around their mean. Analogously, when we have two variables A and B, then we observe how the corresponding elements 'co-vary' with respect to the two corresponding means, $\overline{A}$ and $\overline{B}$ respectively:

$$q_{A,B} = \frac{1}{N-1}\sum_{i=1}^{N}(A_i - \overline{A})(B_i - \overline{B}), \tag{4}$$

where n is the number of observations. If the individual differences co-vary, then the co-variance is positive, otherwise it is negative. The divisor $N-1$ is typical for an *estimate* of the covariance - the unbiased value we typically do not know because we always deal with a subset of samples in praxis.

For reasons of practicality, one generates a full $d \times d$ matrix $\boldsymbol{\Sigma}$ for all *pair-wise* covariances, where the diagonal entries correspond to the variance of the variables; the entries below and above the diagonal hold the corresponding covariances, which are of the same value for pairs whose indices are swapped, because the covariance measure is symmetric. The covariance matrix is thus a square, symmetric matrix; its generation is sometimes denoted as cov(A, B) and its calculation can be expressed also in matrix notation - the exercise will reveal how.

Observe that $\boldsymbol{\Sigma}$ is denoted as boldface as opposed to the summation sign Σ. $\boldsymbol{\Sigma}$ is rather the symbol for the unbiased (theoretic) covariance matrix. The estimated matrix is denoted sometimes as $\hat{\boldsymbol{\Sigma}}$ in books. In Matlab, the matrix $\hat{\boldsymbol{\Sigma}}$ can be generated with the command `cov`, or `nancov` if the data contain not-a-number entries.

Small Sample Size Problem In those classifiers that make use of the covariance matrix, the matrix is often computed for each class separately, which will be made explicit in section12. Estimating a covariance matrix is easy, yet for many linear classifiers this matrix needs also to show certain properties, e.g. it needs to be *positive semi-finite*, meaning that *after* some transformation the covariance matrix has only positive values in a limited range - we omit the details. This constraint is often not fulfilled if the data set has only few samples. And if the covariance matrix is generated for each individual class, then problem of obtaining an 'adequate' covariance matrix is aggravated. This is known as the *small sample size problem*. If the appropriate covariance matrix can not be generated, then Matlab may complain with the following error:
`The pooled covariance matrix of TRAINING must be positive definite.`
To work around this barrier, it is easiest to apply a dimensionality reduction using the PCA and then retry with lower dimensionality, which will be the topic of the upcoming section 4.

3.2 Implementation (Matrix Decomposition)

In the following we point out how a linear classifier is implemented in Matlab in the command `classify`. The code was slightly modified to make it compatible with out variable names (`TREN, TEST,...`). There are two essential steps for training and one for testing (applying), enumerated as steps 1 to 3 now:

1. Learning: Group means: for each class the mean of its training samples is calculated:

```
gmeans = NaN(ngroups, nDim);
for k = 1:ngroups
    gmeans(k,:) = mean(TREN(Grp==k,:),1);
end
```

2. Learning: Now we estimate the covariance matrix using the orthogonal-triangular decomposition (`qr` in first line) for which we subtract the group means from the data, followed by a normalizing division (second line):

```
[~,R]    = qr(TREN - gmeans(Grp,:), 0);
R        = R / sqrt(nObs - ngroups); % SigmaHat = R'*R
```

```
s         = svd(R);
if any(s <= max(nObs,nDim) * eps(max(s)))
   error(message('stats:classify:BadLinearVar'));
end
logDetSigma = 2*sum(log(s)); % avoid over/underflow
```

Then, another matrix decomposition follows, namely the singular value decomposition (`svd` in third line), after which it is verified that the covariance matrix is adequate. We do not further explain these decompositions for reason of brevity. The result is a matrix `R` and a scalar `logDetSigma` which are the equivalent of the weight vector $\mathbf{w}$.

3. Applying: When we classify testing samples, generate a 'confidence' value for each class, which is also called the *posterior*: it is placed into the matrix `D` here (number of testing samples $\times$ number of dimensions). The operation is quasi the dot product (eq. 1):

```
for k = 1:ngroups
   A       = bsxfun(@minus,TEST, gmeans(k,:)) / R;
     D(:,k)  = log(prior(k)) - .5*(sum(A .* A, 2) + logDetSigma);
end
```

   Some of this will be clarified, when we look at the Naive Bayes classifier in section 12, but for the moment we simply apply this procedure without understanding every detail.

With the posteriors in D we determine the class label by the argmax operation (eq. 3), see the example in appendix C.7.

3.3 Applying `classify`

Matlab offers to learn linear classifiers with the command `classify`. The input arguments are as follows:

`sample`	(1st arg)	$n_{\text{test}} \times d$ matrix containing the testing samples
`training`	(2nd arg)	$n_{\text{train}} \times d$ matrix containing the learning samples.
`group`	(3rd arg)	$n_{\text{train}} \times 1$ vector with class/group labels, where each element corresponds to the class in the training matrix (both have the same number of rows of course).
`type`	(4th arg - optional)	allows to chose different types of classification. groups (classes) are fitted with a multivariate Gaussian (eq. 12).
`prior`	(5th arg - optional)	if not specified, it is assumed that classes occur with equal probability. In case of doubt simply use 'empirical': Matlab will calculate the probabilities from `group`.

The function allows also to test non-linear classifiers, but they all share the dot product (eq. 1) as the principle way of evaluating a sample. The output arguments are:

`outclass`	(1st arg)	a $n_{\text{test}} \times 1$ vector, which contains the class assignments for the samples: it is of same length as the grouping variable `group` (3rd input argument).
`err`	(2nd arg)	classification error for training data (`training`)
`Post`	(3rd arg)	$n_{\text{test}} \times c$ matrix containing the posterior values $\in [0, 1]$.

3.4 Recapitulation

Advantages A linear classifier is simple to apply, returns reasonable results and is fast in decision making. The computation of the weights is often done with a covariance matrix (or some variant), whose space complexity is therefore $O(d^2)$; for classification one needs to perform a matrix product only, whose space and time complexity is only $O(d)$, which makes it therefore a very popular classifier. Its efficiency is unparalleled. To obtain a better performance with a different classifier, the learning duration will increase substantially and it will require the adjustment of some parameters.

Disadvantages It is difficult to obtain reliable results for a small training set, for instance $n_{\text{Samples}} < 5$ for a class. For excessively high dimensionality, the generation of the covariance matrix can become unpractical due to its square complexity $O(d^2)$: a regular computer may not have sufficient RAM to calculate the matrix decompositions.

3.5 Exercise

Understanding the Covariance Measure

1. Study the script in appendix C.6: in how many lines can the covariance matrix be generated? To verify the implementation two verifications are carried out: one verifies the variances in the diagonal; another one verifies the entire matrix using the Matlab command `cov`. The differences should be zero: use the commands `assert` and `all` to verify automatically.
2. Change the mean level for one dimension in `D`, e.g. add a value of 1. Do you expect the covariance to change? Explain.
3. Now change the variance for one dimension, e.g. multiply one dimension by a factor. Any changes? If so, why?

Classify Synthetic Data

1. Study the beginners example in appendix C.7. It generates synthetic data and classifies them twice: once with the Matlab command, and once explicitly as introduced in subsection 3.2. At the end we verify that the output is the same for both. And we compare the covariance matrix with the decomposition output R.
2. As with the kNN classifier, proper evaluation is best done with repeated estimates, for instance the 2-fold cross-validation as introduced above.
3. Provoke an error by choosing only small sample sizes.
4. Specify a different type of classifier (for the Matlab command `classify`), e.g. 'diaglinear'. Do the results change significantly?

Classify Real Data

1. Apply the linear classifier to your real data.
2. If there is an error, verify the following: are there dimensions (variables, features) with only 0 values?; are there enough samples in each class? If still not working, then we should try the principal component analysis (PCA), treated in next section.
3. Observe the distribution of values for individual dimensions by histogramming.
4. Look at the covariance matrix. First create it with `Cov = cov(TRN);`. Then plot it with `imagesc`. Add a scale with the command `colorbar`. Plotting makes only sense if the dimensionality is smaller than 2000 maybe, because your screen resolution may not be sufficiently high. But you can plot part of the covariance matrix to obtain an idea of its pattern. What is the range of covariance values? What does a negative value mean?

4 Dimensionality Reduction

Sometimes it is useful, if not even necessary, to reduce the number of dimensions (variables) of the data. There can be different reasons why one seeks such a dimensionality reduction. For instance, the inverse of the covariance matrix can not be computed, which is needed for a linear classifier (previous section) or the Naive Bayes classifier (section 12); or we have very large dimensionality and hence slow classification, in which case one may seek to eliminate the least significant dimensions in order to increase classification speed; or our data may contain irrelevant dimensions which are better eliminated to achieve a higher performance; or we need to find patterns and tendencies in a high-dimensional space, which one tries to uncover by projecting the data onto a 2D or 3D space.

Dimensionality reduction can occur in two principally different ways, whereby here the term *feature* stands for dimension (not for a feature vector representing a sample):

Feature Selection is the selection of the best subset of the (original) input feature set. The most popular procedure is *sequential feature selection*.

Feature Extraction/Transformation is the transformation or combination of the original set of features to create a new (reduced) set of feature. Its most famous example is the principal component analysis (PCA).

4.1 Feature Extraction - PCA DHS p115, 568 Alp p113 ThKo p326 wiki: Principal_component_analysis

The most popular method for feature extraction is the principal component analysis (PCA), also called the Karhunen-Loeve transform. Here, the term *component* stands for feature (variable). The PCA works by realigning the coordinate system to the distribution of the data.

Example: Assume we have a 2D data set, whose overall distribution appears like the shape of an ellipse: the ellipse's larger diameter is rotated by 45 degrees, see figure 4 left side (point cloud is outlined already by the ellipse shape). It is clear that this elliptical point cloud has two major 'directions', which are denoted as z_1 and z_2. The first one is the dominant one, the second is aligned orthogonally to the first one. Then, the axes of the original coordinate system - x_1 and x_2 - are placed onto the new directions, illustrated on the right side in figure 4.

In other words, the PCA determines the 'directions' of greatest variance in the data, and then rotates the coordinate axes to those directions and it moves the origin of the coordinate axes onto the data's center.

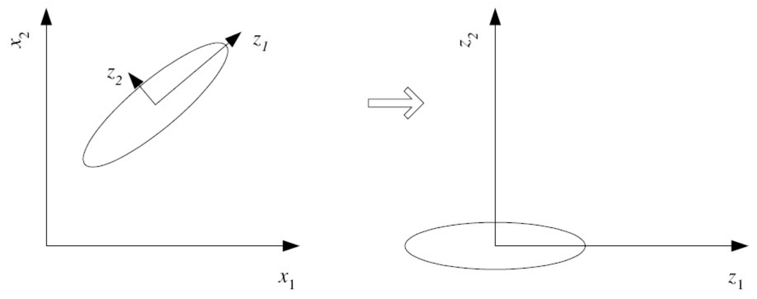


Figure 4: Principal components analysis. **Left**: the ellipse represents the outline of an elliptical point cloud with 'axes' z_1 and z_2. **Right**: the PCA procedure centers the samples and then rotates the coordinate axes to line up with the directions of highest variance. If the variance on z_2 is too small, it can be ignored and we have dimensionality reduction from two to one. [Source: Alpaydin 2010, Fig 6.1].

There are different procedures to perform the PCA. Here we sketch the one using the covariance matrix. It consists of five basic steps:

1. The mean and covariance for the data are determined. The mean results in a single vector $\boldsymbol{\mu}$ (dimensionality d); the $d \times d$ covariance matrix $\boldsymbol{\Sigma}$ was introduced before (subsection 3.1).

2. The eigenvectors $\mathbf{e}_i$ and eigenvalues λ_i are computed from the covariance matrix. For each dimension i there exists a eigenvector $\mathbf{e}$ and its corresponding eigenvalue λ. They represent the directions in the point distribution. The eigenvalues represent the 'significance' of the direction. We omit the details of how they are generated.
3. We now chose the k largest eigenvalues and their corresponding eigenvalues. There are different ways to choose k.
4. We build a $d \times k$ matrix $\mathbf{A}$ consisting of the k eigenvectors.
5. The original data $\mathbf{x}$ are multiplied with matrix $\mathbf{A}$ in order to arrive at the reduced data $\mathbf{x}^r$, which then is of dimensionality [number of samples $\times k$]. This multiplication occurs as explained in appendix B.

Algorithm 3 summarizes the individual steps. To make accurate estimates with our classifiers later, the optimal k is determined only for the training set and we thus apply the last step twice: once to the training set and once to the testing set.

Algorithm 3 The steps of the PCA (for the method using the covariance matrix): performed on $\mathcal{D}^L$.

```
  Parameters      $k$: number of principal components
  Initialization  none particular
  Input           $\mathbf{x}_j(i)$: list of observations ($\mathcal{D}^L$), $j = 1,..,$ n_Observations, $i = 1,..,d$ (n_Dimensions)
  1) Compute:     $\boldsymbol{\mu}$: $d$-dim mean vector
                  $\boldsymbol{\Sigma}$: $d \times d$ covariance matrix
  2) Compute eigenvectors $\mathbf{e}_i$ and eigenvalues $\lambda_i$   ($i$ is dimension index)
  3) Selection of $k$ largest eigenvalues and corresponding eigenvectors
  4) Build $d \times k$ matrix $\mathbf{A}$ with columns consisting of the $k$ eigenvectors
  5) Projection of data $\mathbf{x}_j$ onto $k$-dim subspace $\mathbf{x}'$:   $\mathbf{x}' = \boldsymbol{F}_1(\mathbf{x}) = \mathbf{A}^t(\mathbf{x} - \boldsymbol{\mu})$
  Output          $\mathbf{x}'_j$: list of transformed observations
```

4.1.1 Applying `pca`

Matlab provides the command `pca` (older Matlab versions use `princomp`), which carries out steps 1 and 2 of algorithm 3. How we apply the command and how we use its output is explained next (algorithm 4).

- The command `pca` returns a $d \times d$ matrix called `coeff` as well as a vector of latencies `lat`. Those two variables correspond to the eigenvectors and eigenvalues respectively - if the eigendecomposition were used (see step 2 in algorithm 3). The default for the command is actually a singular value decomposition (encountered already for the linear classifier).
- We then choose the number k (=`nPco` in the code), which can be done based on the values in `lat`. For simplicity, we choose here k based on dimensionality, where the proportion value of 0.7 is a suggestion and should return reasonable results. Should there be fewer samples than dimensions - hopefully not -, then k needs to be smaller then the number of observations minus one - that is why we use the minimum function on the size output. More on the choice of k will follow later.
- Then we create the submatrix `PCO` of dimensionality $d \times k$, corresponding to matrix $\mathbf{A}$ in algorithm 3.
- Finally, we multiply each sample ($\mathbf{x}$ = `DAT(i,:)`) by this submatrix and obtain the data `DATRed` with lower dimensionality (size $n \times k$). That is the data you would then use for classification.

Choice of number of principal components (k) There does not exist a receipt for choosing k. Here are some suggestions:

- One data set: if one classifies only a single data set, one could manually observe the 'variances' in variable `lat`.
- Several data sets: observe the `lat` values for some sets and try to derive a reasonable rule, e.g. the first k components until 99 percent of the total variance (`sum(lat)`) is used up.

Algorithm 4 Applying the PCA. `DAT` is of size $n \times d$. nObs=n (number of samples/observations).

```
[coeff,~,lat] = pca(DAT);
nPco        = round(min(size(DAT))*0.7);  % reduced dimensionality
PCO         = coeff(:,1:nPco);              % select the 1st nPco eigenvectors
% --- Reduce Data:
DATRed      = zeros(nObs,nPco);             % init reduced data matrix
for i = 1:nObs,
    DATRed(i,:) = DAT(i,:) * PCO;           % transform each sample
end
```

- Automatic search: we can automatically search for the maximal "useful" number of components that allows the singular value decomposition to be carried out for the linear classifier (step 2 in subsection 3.2). We would place step 2 into a loop and gradually increase k until s of step is below the threshold. We then take k-1 to make the data reduction.
- **Restriction:** a meaningful number of principal components has to be less than the number of samples minus one, hence the operation `min(size(DAT))` in algorithm 4. This is also mentioned in the Matlab documentation as the 'degrees of freedom'.

4.1.2 Recapitulation

Advantages The PCA works quasi without parameters. We need to merely choose the desired number of components (k).
Disadvantages For very large dimensionality, the PCA may not be computable because it relies on a pair-wise analysis of features, that is its complexity is $O(d^2)$.

Advisory: Reducing dimensionality with the PCA does not necessarily mean that we have 'eliminated' useless dimensions. For the task of discrimination, we may in fact have eliminated useful dimensions: they may have shown low variance in the PCA analysis, but could still have been useful for discrimination!

4.2 Feature Selection Alp p110 ThKo p261, ch5, pdf 274

Should the combination of PCA and linear classifier have failed, we can try to select 'manually' in one of the following methods, which knock out features by evaluating a classifier.

The simplest way to select the best performing set of features would be to test all (binomial) combinations individually, which is also known as *exhaustive search*. This method becomes however unfeasible for high dimensionality. Instead, suboptimal but more time efficient methods are employed. The two most popular ones work by gradually increasing or decreasing the number of dimensions. In either case, checking the error should be done on a validation set, which is distinct from the training set because we want to test the generalization accuracy. With more features, generally we have lower training error, but not necessarily lower validation error.

Let us denote by F, a feature set of input dimensions, x_i, $i = 1, ..., d$. $E(F)$ denotes the error incurred on the validation sample when only the inputs in F are used. Depending on the application, the error is either the mean square error or misclassification error.

Sequential Forward Selection Here we start with features: $F = \emptyset$. At each step, for all possible x_i, we train our model on the training set and calculate $E(F \cup x_i)$ on the validation set. Then, we choose that input x_j that causes the least error,

$$j = \text{argmin}_i E(F \cup x_i)$$

and we

$$\text{add } x_j \text{ to } F \text{ if } E(F \cup x_j) < E(F)$$

We stop if adding any feature does not decrease E.

This process may be costly because to decrease the dimensions from d to k, we need to train and test the system $d + (d-1) + (d-2) + ... + (d-k)$ times, which is $O(d^2)$. This is a local search procedure and does not guarantee finding the optimal subset, namely, the minimal subset causing the smallest error. For example, x_i and x_j by themselves may not be good but together may decrease the error a lot, but because this algorithm is greedy and adds attributes one by one, it may not be able to detect this.

Sequential Backward Selection As the name reveals already, this is the backward procedure of what we just discussed. We start with F containing all features and do a similar process except that we remove one attribute from F as opposed to adding to it, and we remove the one that causes the least error

$$j = \mathrm{argmin}_i E(F - x_i)$$

and we

$$\text{remove } x_j \text{ from } F \text{ if } E(F - x_j) < E(F).$$

4.3 Exercise

Synthetic Data

1. Understand the principles of the PCA with the examples given in C.8. What is plotted in each of the 4 subplots?
2. What does the first component represent? What does the second component represent?
3. Change the amplitude of either synthetic data. Then the 'noise' level. What do the changes alter?
4. What strategy for selecting the number k of principal components can you think of?

Real Data

1. You apply the PCA to the training set `TRN` only. Make first a 'safe' selection of k, then you multiply the matrix A once with the training set and once with the testing set.
2. Then observe `lat`. Take different values for k. Do the results change a lot?
3. Try to determine automatically the maximal number of 'allowable' k for the command `classify`.

5 Evaluating and Improving Classifiers

We now elaborate on how to characterize the performance of a classifier in more detail. One important issue was already introduced, namely the proper estimation of the generalization (prediction) performance of our classifier model using cross-validation; we elaborate on this in the following subsection 5.1. Then we introduce performance measures that were developed for binary classifiers and that stem mostly from the domain of signal detection methods (subsection 5.2).

5.1 Types of Error Estimation DHS p465

When we estimate the classification error - or the percentage correct classification - there are two types of measures to characterize the estimate: bias and variance. They are roughly equal to the terms 'accuracy' and 'precision'. More specifically, they are defined as (see also figure 5):

Bias measures the *accuracy* or quality of the match, that is the difference between the estimated and the actual accuracy (the latter we typically do not know). High bias implies poor match.

Variance measures the *precision* or specificity of the match. High variance implies weak match.

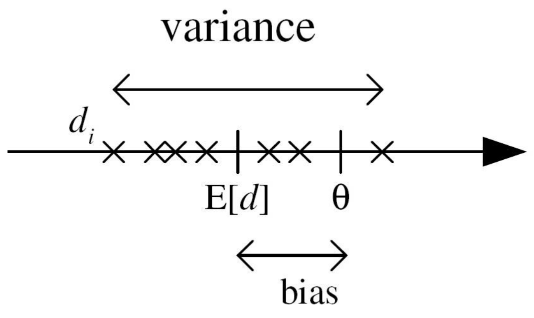


Figure 5: θ is the parameter to be estimated. d_i are several estimates (denoted by 'x') over different samples. Bias is the difference between the expected value of d (denoted as E[d]) and θ. Variance is how much d_i are scattered around the expected value. We would like both to be small. [Source: Alpaydin 2010, Fig 4.1].

Bias and variance are affected by the type of resampling, see table 1 for a summary of methods. So far we had used the holdout method, but a better method is a 5-fold cross-validation, in which the total data set is partitioned into 5 equally sized sets of which 4 partitions serve as training, whereas the remaining partition is used for testing. The partitions are then rotated (shifted) to obtain 5 different performance estimates, which are then averaged to obtain the overall estimate. In Matlab: `crossvalind` (bioinfo toolbox)

Table 1: Error Estimation Methods (from Jain et al. 2000). n: sample size, d: dimensionality. (See also 'Resampling' on wiki)

Method	Property	Comments
Resubstitution	All the available data is used for training as well as testing; training set = test set	Optimistically biased estimate, especially when n/d is small
Holdout	Half the data is used for training and the remaining data is used for testing; training and test sets are independent	Pessimistically biased estimate; different partitionings will give different estimates
Leave-one-out, Jackknife	A classifier is designed using (n-1) samples and evaluated on the one remaining sample; this is repeated n times with different trainings sets of size (n-1).	Estimate is unbiased but it has a large variance; large computational requirement because n different classifiers have to be designed.
Rotation, n-fold cross validation	A compromise between holdout and leave-one-out methods; divide the available samples into P disjoint subsets, 1≤P≤ n. Use (P-1) subsets for training and the remaining subset for test.	Estimate has lower bias than the holdout method and is cheaper to implement than the leave-one-out method.
Bootstrap	Generate many bootstrap sample sets of size n by sampling with replacement; several estimators of the error rate can be defined.	Bootstrap estimates can have lower variance than the leave-one-out method; computationally more demanding; useful for small n.

5.1.1 Validation Set

There are situations, when also a validation set is required: it is quasi a testing set inside the training set. For instance, for feature selection as introduced in subsection 4.2 we require a validation set in order to determine when to stop the process of selecting features otherwise the classification performance will rather deteriorate. Similarly, when training a neural networks (course II), it is recommended to employ a validation set. The validation set is typically taken to be 'one fold' of the training folds.

5.2 Binary Classifiers Alp p489 wiki: Binary_classification, Evaluation_of_binary_classifiers

If a classifier distinguishes between two classes only, then there exists an elaborate set of measures to characterize its performance. To understand the logic, imagine you are a doctor looking at some X-ray image and you need to decide whether the patient is sick or not by observing a specific pattern (signal) in the X-ray image: sometimes you feel sure about the presence of the pattern, sometimes not: in this scenario there exist four possible responses. This can be best understood by looking at the graph in figure 6, which depicts two overlapping density distributions: the one on the left represents the signal (the pattern); the one on the right represents the noise (or background or distracter). A decision threshold θ is set, which so generates four types of responses, see also table below the figure.

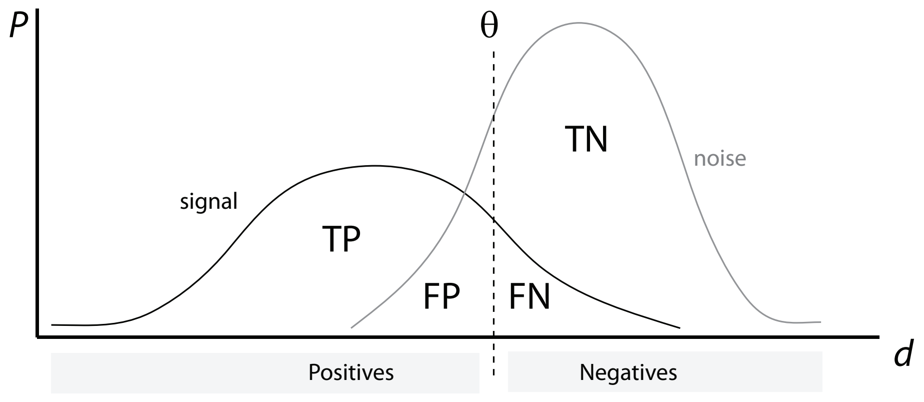


Figure 6: Discrimination of two 'overlapping' classes: probability/frequency versus variable (or feature, dimension). The left distribution represents the signal; the right distribution represents the noise. A decision threshold θ is set and we obtain four response types: true positives (TP; hit), false positives (FP), false negatives (FN; miss), and true negatives (TN).

TP	true positive	$\equiv$ hit	left of θ, under signal	Sick people correctly diagnosed as sick
TN	true negative	$\equiv$ correct rejection	right of θ, under noise	Healthy people correctly identified as healthy
FP	false positive	$\equiv$ false alarm	left of θ, under noise	Healthy people incorrectly identified as sick
FN	false negative	$\equiv$ miss	right of θ, under signal	Sick people incorrectly identified as healthy

Clearly, any decision results in a trade-off. If the doctor wants to avoid unnecessary treatment - of persons incorrectly identified as sick -, then he chooses a threshold more to the left, but thereby also missing actual false negatives. And vice versa, if the doctor attempts to treat all sick people, then he chooses a threshold more to the right, but thereby treating also some healthy persons. To quantify this trade-off there exists different measures. In a first step, the responses are arranged in a so-called *confusion matrix*, aka *contingency table* or *cross tabulation* (wiki: Confusion_matrix):

	Actual Matches	Actual Non-matches	
Predicted Matches	TP	FP	P'
Predicted Non-matches	FN	TN	N'
	P	N	

The columns sum up to the actual number of positives (P) and negatives (N), while the rows sum up to the predicted number of positives (P') and negatives (N'):

P	# positive actual instances	= TP + FN
N	# negative actual instances	= FP + TN
P'	# positive classified instances	= TP + FP
N'	# negative classified instances	= FN + TN

Example: Assume you have made 100 observation decisions in your task of which 22 times you predicted the presence of the signal and 78 times you predicted its absence (P' and N', respectively). Later you are informed that 18 of your 'presence' predictions were correct, and 76 of your absence predictions were false. Then, you can calculate the frequency for the four response types:

	Actual Matches	Actual Non-matches	
Predicted Matches	TP = 18	FP = 4	P' = 22
Predicted Non-matches	FN = 2	TN = 76	N' = 78
	P = 20	N = 80	100 total

Given the quantities above, we then can calculate the following measures, which often come in pairs and are typical for certain fields (wiki: Precision_and_recall):

Name	Formula	Other Names	Preferred Use
Error Accuracy	(FP + FN)/(P + N) (TP + TN)/(P + N) = 1 - Error		
True Pos Rate False Pos Rate	TP / P FP / N	Hit-Rate, Recall, Sensitivity Fall-Out Rate	ROC curve
Precision Recall	TP / P' TP / P	 True Pos Rate, Sensitivity	Information Retrieval
Sensitivity Specificity	TP / P TN / N = 1 - FP-rate	True Pos Rate, Recall	Medicine
F1 score	$2 \cdot \frac{(Precision \cdot Recall)}{(Precision + Recall)}$		

In our example, the true-positive rate (TP-rate or TPR) is 0.90; the false positive rate (FP-rate or FPR) is 0.05; the accuracy 0.94.

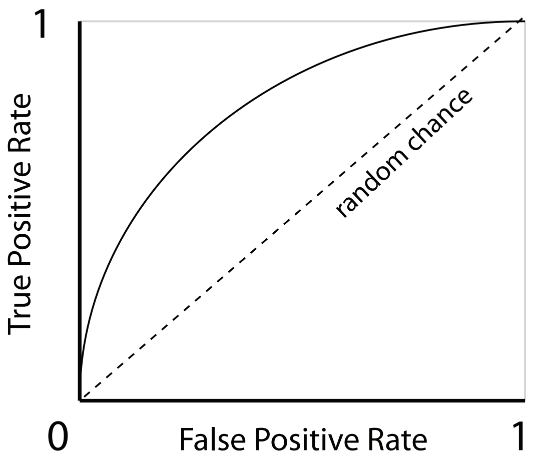
Threshold Selection: If our system permits to change the decision threshold, by means of manipulating some parameters, then we can set the threshold value such that it suits our needs. This is valuable because the cost of false alarms or misses is different for each task and one often tries to balance the costs.
Example: Google uses an algorithm to blur faces and car license plates in their street view, in order to avoid lawsuits by private persons who were accidentally photographed during the recording. How would you adjust the algorithm? Would you permit unblurred faces? Are false alarms costly?

> **Answer:** You probably want to detect all faces to ensure that no law suit is filed; false alarms means that objects that appear like faces and car license plates are blurred, something which does not really impair the advantages of street view, hence false alarms do have a low cost here.

On the other hand, if false positives are associated with a large cost, then a 'conservative' threshold may be the preferred choice. For instance, if the verification of a detected object required much labor, then one may choose a threshold in which some objects may go unnoticed, but which would save 'energy'.

In the search of an optimal threshold a so-called ROC curve can help, coming up next.

Figure 7: The ROC curve is generated by systematically manipulating the decision threshold and plotting the true positive rate against the false positive rate for each performance measurement. The curve lies typically above the diagonal. The diagonal represents random chance. Ideally the curve would rise steep. Sometimes the area under the curve (AUC) is given as a measure - the higher the value, the better the system.



ROC curve (receiver operating characteristic) **wiki: Receiver_operating_characteristic:** If there is an easy way to manipulate the decision threshold of the binary classifier - by a single parameter for instance - then can generate a so-called ROC curve, see figure 7b. It plots the true positive rate (on the y-axis) against the false positive rate (on the x-axis).

Assume that the green curve reflects the performance of a loosely 'tuned' model. If one hopes to find a better model, then the curve should bend more toward the upper left, that is it should rise faster toward 1. If the curve becomes flatter and closed to the stippled diagonal, then the model is worse. If the curve runs along the diagonal, then the decision making is pretty much random. If the curve runs below the diagonal, something is completely wrong - we may have swapped the classes by mistake.

Using the ROC curve, the classification performance is sometimes specified as the area underneath it (AUC), thus we report a scalar value between 0.5 (chance) and 1.0 (perfect).

Precision-Recall Curve This curve is popular for search and rankings in information retrieval, where the ordering of data is important.

5.3 Three or More Classes

For classifiers discriminating three or more classes, the above 'binary' characterization - the response table and its measures - is not directly applicable. Often, one only reports the percentage of correct classification or the error, that is the percentage of misclassification as we did so far in our examples. Nevertheless, we can gain more insight about the classifier model by applying the binary measure in a one-versus-remaining classification for each class.

The confusion matrix is of size $c \times c$, where c is the number of classes. In that table, actual and predicted classes are often swapped - as opposed to the response table introduced above: the given (actual) classes are listed rowwise, the predicted classes are given columnwise. For good classification, mostly the diagonal entries would show high values, namely where actual and predicted class agree.

Example: Assume you have trained a classifier to distinguish between cats, dogs and rabbits. You test the classifier with 27 samples, 8 cats, 6 dogs, and 13 rabbits. You observe that your model makes the following confusions (left column: actual; upper row: predicted)

Act / Pred	Cat	Dog	Rabbit
Cat	5	3	0
Dog	2	3	1
Rabbit	0	2	11

In Matlab we can use the Matlab command `confusionmat` (stats toolbox), or more directly

```
CM = accumarray([Grp LbTst],1,[nCat nCat]);
```

We then generate the binary measures for each class c in an evaluation, where one class is distinguished from all other categories. For a category under investigation, its diagonal entry represents the number of hits, the remaining sum of values along the row are the false positives, the remaining values along the column are the false negatives.

5.4 More Tricks & Hints

5.4.1 Class Imbalance Problem ThKo p237

In practice there are cases in which one class is represented by many more samples than another class, or some classes just have very few samples. This is usually referred to as the class imbalance problem. Such situations occur in a number of applications such as text classification, diagnosis of rare medical conditions, and detection of oil spills in satellite imaging. Class imbalance may not be a problem if the task is easy to learn, that is if classes are well separable; or if a large training data set is available. If not, one may consider trying to avoid possible harmful effects by 'rebalancing' the classes by either oversampling the small class and/or by undersampling the large class.

5.4.2 Estimating Classifier Complexity - Big O

We already discussed some of the advantages and disadvantages of the different classifier types in terms of their complexity. This is typically expressed with the so-called Big O notation wiki: Big_O_notation. In short, the notation classifies the algorithms by how they respond to changes in input size, e.g. how a change affects the processing time or working space requirements. In our case, we investigate changes in n or d (of our $n \times d$ data matrix). The issue is too complex to elaborate here and we merely summarize here, what we mentioned so far and what will be mentioned in later sections. For classifiers, we also make the distinction between the complexity during learning and the one of classifying a testing sample:

Classifier // Cluster alg.	Learning	Classification
kNN	-	$O(dn)$
Linear	$O(d^2)$	$O(d)$
Tree	$O(d)$	$O(d)$
K-Means	$O(ndkT)$	
Hierarchical	$O(d^2)$	

Table 2: Complexities of classifiers and clustering algorithms. d=number of dimensions; n=number of samples; k=number of clusters; T=number of repetitions

5.4.3 Learning Curve

It is common to test the classifier for different amounts of learning samples (e.g. 5, 10, 15, 20 training samples), and to plot classification accuracy (and/or error) as a function of the number of training samples, a graph called learning curve. An increase in sample size should typically lead to an increase in performance - at least initially; if performance only decreases then something is wrong. For some classifiers - neural networks typically -, the classification accuracy may start to decrease for very large amounts of training due to a phenomenon called *overtraining* (overfitting). In that case one would employ a validation set (see above subsection 5.1.1) to observe when to stop training: the performance on the validation set would increase initially and then saturate with learning duration; it would start to decrease when overtraining starts to occur, and that is the point when training should be stopped.

5.4.4 Improvement with Hard Negative Mining and Artificial Samples

Now we mention two tricks that may help to improve classifier performance. These are tricks that are typically used with other classifiers, but can very well be tested with a linear classifier at little cost. Trying to tune a more complex classifier can be more difficult than trying one of the following to tricks:

Hard Negative Mining: Here we focus on samples which trigger false alarms in our classifiers. More explicitly, if we train a classifier to categorize digits, than we observe and collect (mine) those digits that confuse one category, that is those samples that trigger false alarms. For instance, we collect those digit samples that confuse class '1', which could be '7's or '4's; or for category '3' it could be '2' or '8'. Then we train the classifier again with those 'hard negatives' in particular.

Creating 'Artificial' Samples: This is a trick popular in computer vision. Because the collection and labeling of image classes is tedious work, one sometimes expands the training set artificially by generating (automatically) more training images, which are slightly scaled and distorted variants of the original samples. This can be tried in combination with adding noise to the samples. Of course, artificial samples are created from the training set only `TRN` - not from the testing set.

5.5 Excercise

Synthetic Data

1. Study the example in appendix C.9. The example generates three ROC curves (outer loop). Two random distributions are generated, one is called the 'signal', the other 'background'. Which one lies where?
2. The code generates the confusion matrix twice and asserts that the two versions are the same. Where on the confusion matrix are aligned the actual and predicted classes? Observe what values are taken from the confusion matrix.
3. What happens if you change the relational operator of the decision step from 'greater than' to 'less than'?
4. Write a function `f_PerfBinClassif` that calculates the measures from the confusion matrix. As input the function takes two lists of labels.

Real Data

1. Generate the confusion matrix for your results.
2. Analyze the pairwise precision/recall curve for your system.
3. Create a case of class imbalance, for instance reduce the number of samples for one class to the several samples. Does it change the results significantly? If not, why not?

6 Clustering - K-Means

6.1 More on Clustering in General   wiki: Cluster_analysis

Sometimes we are given data, which we need to organize into meaningful groups (or partitions), which now are called clusters. We attempt to find clusters in order to uncover 'trends' in data. This is useful in many fields such as economy, bioinformatics, artifical intelligence, etc. In comparison to the previous classification algorithms (which required labeled data), we now deal with unlabeled data, that is we do not have knowledge of any class labels. Because we do not have any 'guidance' by labeled data, clustering is sometimes called unsupervised learning. Given a set of data points (see figure 8), how do we find its dense regions (clusters), which likely correspond to classes or trends?

Clustering is used for data reduction, hypothesis generations, hypothesis testing, prediction based on groups; in image analysis it is often used for scene segmentation. The two following examples are from ThKo p598:

Business example for hypothesis testing: cluster analysis is used for the verification of the validity of a specific hypothesis. Consider, for example, the following hypothesis: 'Big companies invest abroad.' One way to verify whether this is true is to apply cluster analysis to a large and representative set of companies. Suppose that each company is represented by its size, its activities abroad, and its ability to complete successfully projects on applied research. If, after applying cluster analysis, a cluster is formed that corresponds to companies that are large and have investments abroad (regardless of their ability to complete successfully projects on applied research), then the hypothesis is supported by the cluster analysis.

Medical example for prediction based on groups: cluster analysis is applied to a data set concerning patients infected by the same disease. This results in a number of clusters of patients, according to their reaction to specific drugs. Then for a new patient, we identify the most appropriate cluster for the patient and, based on it, we decide on his or her medication.

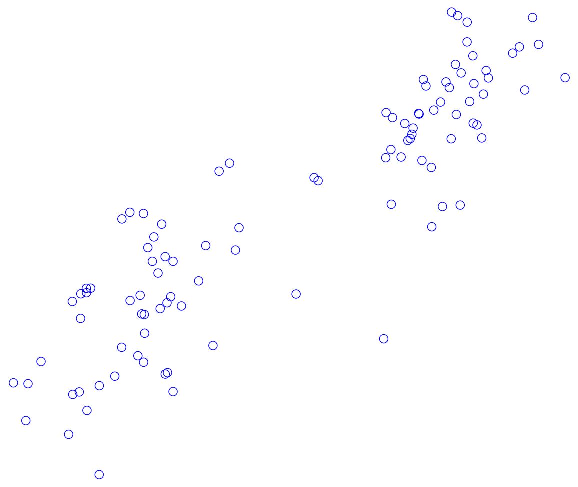


Figure 8: Illustrating the clustering problem in 2D. We are given a set of points and we are attempting to find dense regions (classes), which likely correspond to potential classes. Are there two, three or more classes?
Intuitively one would like to 'smoothen' the distribution (section 11) or to measure all point-to-point distances to obtain a detailed description of the point distribution (subsection 7), which however is computationally very intensive for large dimensionality; for large dimensionality or datasets we therefore use 'simpler' procedures such as the k-Means algorithm (subsection 6.2).

One challenging issue with all clustering algorithms is that one needs to specify roughly the degree of clustering we expect. The least information that we have to provide to a cluster analysis algorithm is either the assumed number of clusters k (that we suspect to reside in the data), or some threshold that separates data points when they are too distant from each other. Efficient clustering without any guidance at all appears impossible.

The first clustering algorithm we introduce now, is one where one specifies the number of expected clusters k. Its procedure is somewhat hastily but relatively quick. It is explained in the remainder of this section. The second clustering algorithm we introduce is one where one needs to specify a relative distance to arrive at a partitioning. That procedure is explained in the following section 7.

6.2 K-Means DHS p526 ThKo p741 Alp p145

K-means clustering is an iterative procedure where the cluster center (centroids) are chosen randomly first and then wander towards the actual cluster centers by repeated distance measurements. One specifies how many cluster k one expects in the data (from n total data points). Then k points are selected randomly (from the n samples), that are taken as initial centroids. Then, the remaining data points are assigned to the nearest centroids, meaning the 'membership' (or label) is determined for each datapoint based on distance - loosely speaking the inverse of the kNN assignment. The resulting partitions (clusters) are used to compute new centroids by simply taking the mean - hence the name kmeans -, which will have slightly moved from their previous location. With the new centroids a new nearest-centroid clustering is carried out, which will result in a new partitions closer to the actual clusters. By repeating these two steps, centroid computation and nearest-centroid clustering, the algorithm gradually moves towards its final clusters. To terminate this cycle, it requires the definition of a stopping criterion, e.g. we quit after the new centroids hardly move anymore, which means that the cluster development has 'settled'. Algorithm 5 summarizes the procedure:

Algorithm 5 k-Means clustering algorithm. Centroid = cluster center.

```
  Parameters      $k$: number of clusters.
  Initialization  randomly select $k$ samples as initial centroids.
  Input           $\mathbf{x}$ list of vectors
  Repeat
    1. Generate a new partition by assigning each pattern to its closest centroid
    2. Compute new centroids from the labels obtained in the previous step
    (3. Optional): adjust the number of clusters by merging and splitting existing clusters
                   or by removing small, or outlier clusters.
  Until stopping criterion fulfilled (e.g. new centroids hardly move anymore)
  Output   $L$ list of labels (a cluster label for each $\mathbf{x}_i$)
```

The kmeans procedure works relatively fast because most samples are 'looked' at only occasionally, namely for the number of iterations until the actual centroid is found. In contrast, the hierarchical clustering procedure (section 7) is much more exhaustive by observing each sample $n-1$ times. The shortcoming of the kmeans procedure is that is does not always find its actual centroids accurately. A simple solution to that shortcoming is to run the procedure repeatedly (always different starting centroids) and then to select the partitioning for which the total sum of final distance values is smallest. This type of repetition is not explicitly stated in algorithm 5, that is we repeat algorithm 5 T times and chose that L (out of T L's) whose total distance is smallest. This repeated application is no guarantee that the actual centroids are found, but it has been shown to be fairly reliable in practice.

6.3 Implementation, Applying `kmeans`

Implementation Implementing a primitive version of the k-Means algorithm is not so difficult - because it is a fairly straightforward algorithm. An example is given in appendix C.11.

Applying `kmeans` Matlab provides the function `kmeans` for which one has to specify only the number of clusters k as minimal parameter input. The number of repetitions T is given as parameter 'replicates'. The default value is 1.

The function firstly performs a coarse clustering using what is called *batch update*. Here, the assignment of each point to its closest centroid (step 1) is done at once for all points simultaneously - our example code in appendix C.11 does that. This is somewhat coarse, because we could individually investigate how the change in class label (membership) would affect the distances and labeling of all others; that would be more time consuming but more accurate. The individual analysis is called *online update*. The Matlab function carries out this online update, after the coarse batch update, see nested functions called `batchUpdate` and `onlineUpdate`. If you have very large datasets and you prefer faster computation, you may turn off the second step (the online update).

However, the Matlab implementation of this algorithm does not take care of NaN entries, that is, it throws out any rows (observations) where NaN occur. To deal with NaN entries, one has to modify the script as for instance: `http://alpha.imag.pub.ro/~rasche/course/patrec/xxxx`.

6.4 Application - More Examples

k-Means is very popular in data compression, it is specifically used as vector quantization in image compression (Alp p145). Let us say we have an image that is stored with 24 bits/pixel and can have up to 16 million colors. Assume we have a color screen with 8 bits/pixel that can display only 256 colors. We want to find the best 256 colors among all 16 million colors such that the image using only the 256 colors in the palette looks as close as possible to the original image. This is color quantization where we map from high to lower resolution. In the general case, the aim is to map from a continuous space to a discrete space; this process is called vector quantization. Of course we can always quantize uniformly, but this wastes the colormap by assigning entries to colors not existing in the image, or would not assign extra entries to colors frequently used in the image. For example, if the image is a seascape, we expect to see many shades of blue and maybe no red. So the distribution of the colormap entries should reflect the original density as close as possible placing many entries in high-density regions, discarding regions where there is no data.

6.5 Recapitulation

Complexity The overall complexity is fairly low, namely $O(ndkT)$, where n is the sample size, d the dimensionality, k the number of clusters and T the number of replicates (DHS p527). Note that if you perform an excessive number of replicates on a small dataset, you may exceed the complexity of the hierarchical clustering algorithm (section 7), in which case it may be more feasible to use that one.

Advantages Works relatively fast due to its relatively low complexity; suitable for very large datasets with tens or hundreds of thousands of samples - for which hierarchical clustering becomes unfeasible (see also Alp p158).

Disadvantages

1) Specification of the number of clusters k. There exist of course many attempts to find procedures, that determine k automatically, yet none has proven be effective for all patterns.

2) Does not guarantee optimal results due to random initial selection and non-exhaustive comparison.

6.6 Exercise

Synthetic Data

1. Study the example given in appendix C.11. There are two versions of k-means clustering: one using the Matlab command `kmeans`, and one being a primitive self-written version. The figure plots the labeling for each version in a separate subplot, but the centroids for both versions are plotted in both subplots for facilitated comparison.
2. The two versions do not produce exactly the same output? Why?
3. Write a function `f_ClustInfo`, which takes the labels as argument (once it was `Lb` and once `IxMin`) and produce a structure (`struct`) that contains information about the clusters, such as its member indices, cluster sizes, centroids, etc. The purpose of that would be that you later can access cluster information by calling for instance, `Cinf(1).IxMem` to access the cluster members.
4. Take unequal cluster sizes to generate your synthetic data. The example used 20 for each so far, try 20 and 80 for instance.
5. Record the trajectory of the centroids in a cell variable `aCtr = cell(nCls,1)`, starting with the initial random assignment, for which you also had. Then, at the end, plot the trajectory.

Real Data

1. Apply the K-Means to your real data, that you used for classification (you do not employ the known class labels for this). Then compare the clustering output with the known class labels. Between what classes do you find good agreement?
2. Turn off online update. Do the results change a lot?
3. Try different distance metrics. Which one is faster, which one is slower? Hint: think in the number of operations that need to be performed. **[Answer:** The hamming distance is faster because it sums only differences, whereas the Euclidean is slower because it squares and takes square root in addition.**]**

7 Clustering - Hierarchical DHS p550 ThKo p653

Hierarchical clustering is a more thorough clustering procedure than the k-Means method, but it is computationally more intensive because we measure the pairwise distances between all points. Hierarchical clustering consists of three principal computational steps:

1. The pairwise distances between all points are determined resulting in a $n \times n$ distance matrix. It is a similarity matrix if a similarity metric is used. In comparison, the k-Means algorithm does compute only a proportion of these pairwise distances.
2. Using the distance matrix, a nested hierarchy of all n data points is generated, by gradually linking the pairs starting with the closest pair. A hierarchy can be represented by a tree, in which case it is called *dendrogram*. This step is far less costly, as we only compare a list of distances.
3. We cut the tree horizontally at some distance and the resulting 'branches' form the clusters.

The linking procedure is the important step. There exist two principal types of linking, agglomerative and divisive. Agglomerative linking starts with the smallest distance pairs gradually links to more distal pairs; divisive works the opposite way, by considering all pairs and trying to gradually break down the links. Agglomerative linking is the more popular one and we therefore consider only that one. Divisive linking is computationally so demanding that it is rare in practice.

7.1 Agglomerative Linking

This method is also known as Bottom-Up clustering or Clumping. It starts with n (singleton) clusters and forms the hierarchy by successively merging clusters. The general algorithm is:

Algorithm 6 Generalized Agglomerative Scheme (GAS). From ThKo p654.

```
Parameters      cut threshold $\theta$
Initialization  $t = 0$
Initialization  choose $\Re_t = \{C_i = \{\mathbf{x}_i\}, i = 1,..,N\}$ as the initial clustering.
Repeat:
    $t = t + 1$
    Among all possible pairs of clusters $(C_r, C_s)$ in $\Re_{t-1}$ find the one, say $(C_i, C_j)$, such that
        $g(C_i, C_j) = \min_{r,s} g(C_r, C_s)$,    $g$ is a distance (dissimilarity) function    (5)
    Define $C_q = C_i \cup C_j$ and produce the new clustering $\Re_t = (\Re_{t-1} - \{C_i, C_j\}) \cup \{C_q\}$
Until all vectors lie in a single cluster.
Cut hierarchy at level $\theta$
```

The distance function in equation 5, can be implemented in different ways and its choice influences the clustering outcome:

Single-Linkage (aka Nearest Neighbor): uses the minimum distance to compute the distances between samples and clusters. The method tends to generate chain-like clusters.

Complete-Linkage (aka Furthest Neighbor): uses the maximum distance to compute the distances between samples and clusters. The method tends to generate compact clusters.

The difference between the two is explained in the following figure. The example data set is shown in figure 9a and contains 11 points $(x_1, .., x_{11})$, whereby the points are already connected by straight lines segments such that 2 clusters were formed - representing the outcome of the clustering procedure.

For the (near complete) trees in 9b and c imagine a y-axis that represents the distance between clusters. In both graphs, the trees lack the top level of the hierarchy, that is the trees were 'cut' already at a level, where it generates the two partitions. In this example, the two linkage methods result in the same partitioning, but for more complex data sets the outcome likely differs.

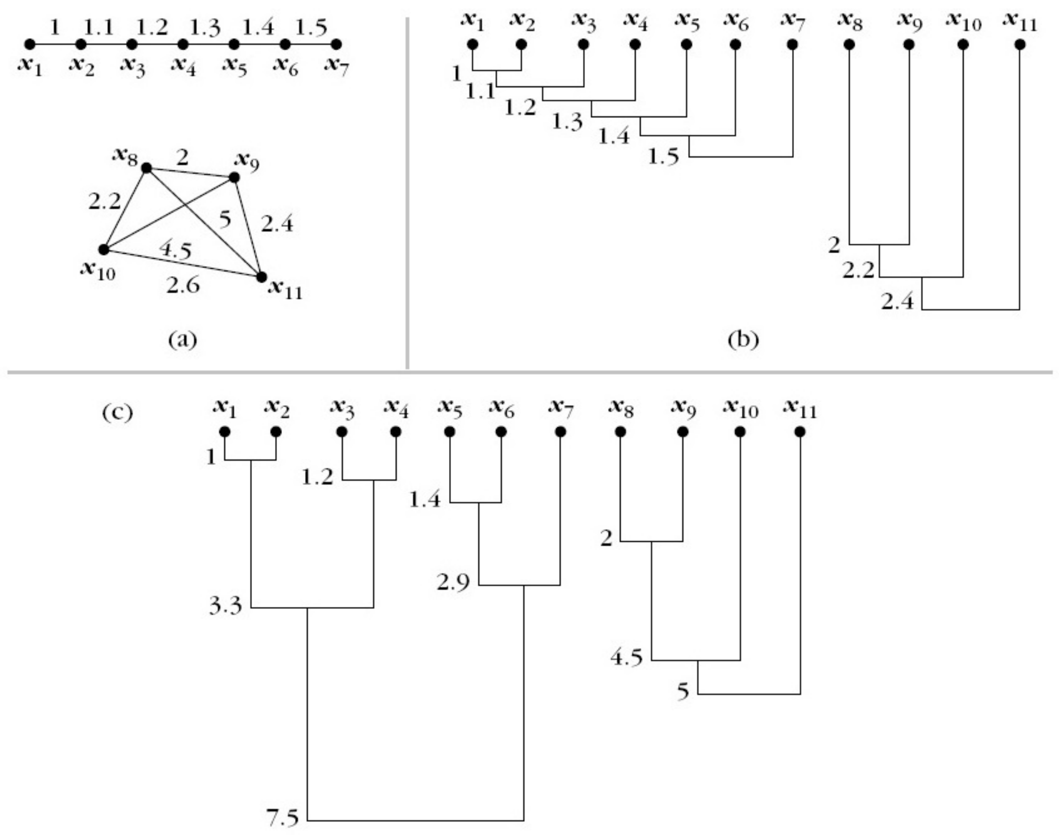


Figure 9: **a**. The data set: 11 points $(x_1, .., x_{11})$. **b**. the dissimiliarity dendrogram as generated by the single-link condition. **c**. the dendrogram as generated by the complete-link condition (the top level of the hierarchy is not shown). [Source: Theodoridis, Koutroumbas, 2008, Fig 13.3]

7.2 Applying `clusterdata`, Implementation

- Single Command: in Matlab the command `clusterdata` performs this type of clustering.The least parameter that needs to be provided is a cutoff frequency, which is given either as:

  - a real value between 0 and 2, in which case it represents an 'inconsistent' value
  - or as an integer value (≥ 2) specifying the desired number of clusters (like k as in kmeans):

  `Lbl = clusterdata(DAT, 1.25)`. Lbl contains the assigned cluster labels.

- Individual Steps: we program the steps individually as follows:

```
Dis = (DAT);                           % pairwise distances
Lnk = linkage(Dis, 'single');                 % single linkage method
Lbl = cluster(Lnk,'cutoff', 1.25);
```

  `Dis` contains the $N * (N - 1)/2$ pairwise distances between all observations as a row vector (N is the number of datapoints/samples/observations).

  `Lnk` is a $N \times 3$ array containing the connections of the tree, which can be displayed using the command `dendrogram`. More specifically, the first two columns contain the tree connections, the third row are the distances between clusters.

  To visualize the distance matrix, use `squareform` to rearrange the vector obtained from `pdist`.

7.3 Recapitulation

Application: Hierarchical clustering is applied in rather specific domains, e.g. biological taxonomy.

Advantages Exhaustive analysis which can roughly express the 'structure' of the data distribution, not necessarily by one linking method only, but either one may capture the rough structure.

Disadvantages

1. Optimal cutoff frequency can not be determined automatically - as can not k in the k-Means procedure; as mentioned before, this is the challenge of any clustering algorithm, namely the specification of some expectancy.
2. Its detailed analysis is its downside: the exhaustive (pair-wise) sample comparison implies square complexity: $O(N^2)$. Hierarchical cluster is thus suitable for databases of limited size only.

7.4 Exercise

1. Get the example in appendix C.12 running. It consists of a script and 3 function scripts. One function performs a transformation of the linkage output; the other two plot the MST in two different versions.
2. The (main) script clusters and plots two kind of patterns, a random point pattern, whose clusters appears in the left column, and a systematic pattern, whose clusters appear in the right column. The systematic pattern in turn consist of an arc placed above a grid of points.
3. Play with the thresholds and observe the output carefully.

8 Decision Tree ThKo p215, s 4.20, pdf 228 Alp p185, ch 9 DHS p395

A decision tree is a multistage decision system, in which classes are sequentially rejected until we reach a finally accepted class. It corresponds to the flow diagram we learned in school.

Figure 8 left shows an example for a 2D data set. On the left is shown an (artificial) data set, which consists of 6 regions belonging to 4 different classes (classes 1 and 3 have two instances each). On the right is shown the corresponding tree, which consists of 5 decision nodes (circles) and 6 leave nodes (squares; aka terminal nodes), which are connected by links or branches. Given a (testing) data point, e.g. $x_1 = 0.15, x_2 = 0.5$, the decision node t_0 (aka root node), tests the first component, x_1, by applying a threshold value of $1/4$: if the value is below, the data point is assigned to class ω_1; if not, the process continues with binary decisions of the general form of $x_i > \alpha$ (α = threshold value) until we found a likely class label. The example in figure 8 right is a binary decision tree and splits the space into rectangles with sides parallel to the axes (hyperrectangles for dimensionality > 2). Other types of trees are also possible, that split the space into convex polyhedral cells or into pieces of spheres. Note that it is possible to reach a decision without having tested all available feature components.

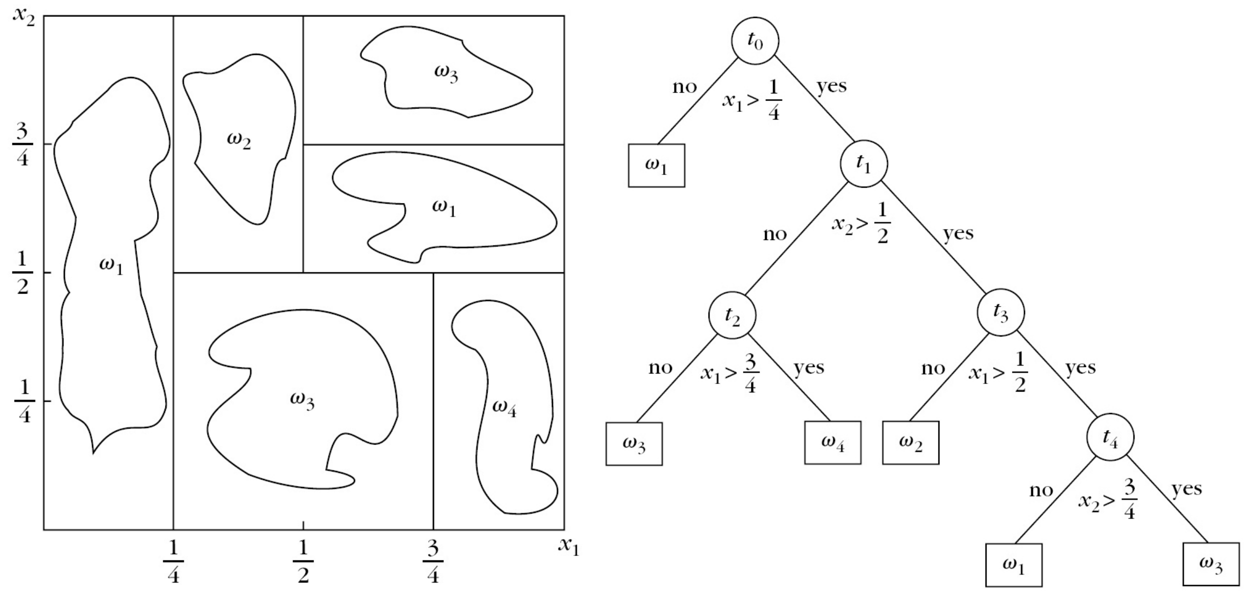


Figure 10: **Left:** a pattern divided into rectangular subspaces by a decision tree. **Right:** corresponding tree. Circles: decision nodes. Squares: leaf/terminal nodes. [Source: Theodoridis, Koutroumbas, 2008, Fig 4.27,4.28]

In praxis, we often have data of higher dimensionality and we therefore need to develop binary decision trees automatically, that is, we need to find out when which component x_i is tested with what threshold value α_i. We elaborate on this now, discussing 3 issues: impurity, stop splitting and class assignment rule.

Impurity Every binary split of a node, t, generates two descendant nodes, denoted as t_Y and t_N according to the 'Yes' or 'No' decision; node t is also referred to as the *ancestor* node (when viewing such a split). The descendant nodes are associated with two new subsets, that is, X_{tY}, X_{tN}, respectively (the root node is associated with the training set X).

Now the crucial point: every split must generate subsets that are more 'class homogeneous' compared to the ancestor's subset X_t. This means that the training feature vectors in each one of the new subsets show a higher preference for specific class(es), whereas data in X_t are more equally distributed among the classes.

Example: for a 4-class task: assume that the vectors in subset X_t are distributed among the classes with equal probability (percentage). If one splits the node so that the points that belong to classes ω_1 and ω_2 form subset X_{tY}, and the points from ω_3 and ω_4 form X_{tN} subset, then the new subsets are more homogeneous compared to X_t or 'purer' in the decision tree terminology.

The goal, therefore, is to define a measure that quantifies node impurity and split the node so that the overall impurity of the descendant nodes is optimally decreased with respect to the ancestor node's impurity. Let $P(\omega_i|t)$ denote the probability that a vector in the subset X_t, associated with a node t, belongs to class ω_i, $i = 1, 2, ..., M$. A commonly used definition of *node impurity*, denoted as $I(t)$, is the entropy for subset X_t:

$$I(t) = -\sum_{i=1}^{M} P(\omega_i|t) \log_2 P(\omega_i|t)$$

where $\log_2$ is the logarithm with base 2 (see Shannon's Information Theory for more details). We have:
- Maximum impurity $I(t)$ if all probabilities are equal to $1/M$ (highest impurity)
- Least impurity $I(t)$ = 0 if all data belong to a single class, that is, if only one of the $P(\omega_i|t) = 1$ and all the others are zero (recall that $0 \log 0 = 0$).

When determining the threshold α at node t, we attempt to chose a value such that $\Delta I(t)$ is large.

Example: given is a 3-class discrimination task and a set X_t associated with node t containing $N_t = 10$ vectors: 4 of these belong to class ω_1, 4 to class ω_2, and 2 to class ω_3. Node splitting results into: subset X_{tY}, with 3 vectors from ω_1, and 1 from ω_2; and subset X_{tN} with 1 vector from ω_1, 3 from ω_2, and 2 from ω_3. The goal is to compute the decrease in node impurity after splitting. We have that:

$$\begin{aligned}
I(t) &= -\frac{4}{10}\log_2\frac{4}{10} - \frac{4}{10}\log_2\frac{4}{10} - \frac{2}{10}\log_2\frac{2}{10} = 1.521 \\
I(t_Y) &= -\frac{3}{4}\log_2\frac{3}{4} - \frac{1}{4}\log_2\frac{1}{4} = 0.815 \\
I(t_N) &= -\frac{1}{6}\log_2\frac{1}{6} - \frac{3}{6}\log_2\frac{3}{6} - \frac{2}{6}\log_2\frac{2}{6} = 1.472
\end{aligned}$$

Hence, the impurity decrease after splitting is

$$\Delta I(t) = 1.521 - \frac{4}{10}(0.815) - \frac{6}{10}(1.472) = 0.315.$$

Stop Splitting The natural question that now arises is when one decides to stop splitting a node and declares it as a leaf of the tree. A possibility is to adopt a threshold T and stop splitting if the maximum value of $\Delta I(t)$, over all possible splits, is less than T. Other alternatives are to stop splitting either if the cardinality of the subset X_t is small enough or if X_t is pure, in the sense that all points in it belong to a single class.

Class Assignment Rule Once a node is declared to be a leaf, then it has to be given a class label. A commonly used rule is the majority rule, that is, the leaf is labeled as ω_j where

$$j = \text{argmax}_i P(\omega_i|t)$$

In words, we assign a leaf, t, to that class to which the majority of the vectors in X_t belong.

A critical factor in designing a decision tree is its size. The size of a tree must be large enough but not too large; otherwise it tends to learn the particular details of the training set and exhibits poor generalization performance. Experience has shown that use of a threshold value for the impurity decreases as the stop-splitting rule does not lead to trees of the right size. Many times it stops tree growing either too early or too late. The most commonly used approach is to grow a tree up to a large size first and then prune nodes according to a pruning criterion. A number of pruning criteria have been suggested in the literature. A commonly used criterion is to combine an estimate of the error probability with a complexity measuring term (e.g., number of terminal nodes) .

Algorithm 7 Growing a binary decision tree. From ThKo p219.

```
Parameters  Stop-splitting threshold $T$
Initialization  Begin with the root node $X_t = X$.
For each new node $t$
    For every feature $x_k$ $(k = 1, ..., l)$
        For every value $\alpha_{kn}$ $(n = 1, ..., N_{tk})$
            - Generate $X_{tY}$ and $X_{tN}$ for: $x_k(i) \leq \alpha_{kn}, i = 1, ..., N_t$
            - Compute $\Delta I(t|\alpha_{kn})$
        End
        $\alpha_{kn_0} = \mathrm{argmax}_\alpha \Delta I(t|\alpha_{kn})$
    End
    $[\alpha_{k_0 n_0}, x_{k_0}] = \mathrm{argmax}_\alpha \Delta I(t|\alpha_{kn_o})$
    If the stop-splitting rule is met
        declare node $t$ as a leaf and designate it with a class label
    Else
        Generate nodes $t_Y$, $t_N$ with corresponding $X_{tY}$, $X_{tN}$ for: $x_{k_0} \leq \alpha_{k_0 n_0}$
    End
End
```

Disadvantages It is not uncommon for a small change in the training data set to result in a very different tree, meaning there is a high variance associated with tree induction. The reason for this lies in the hierarchical nature of the tree classifiers. An error that occurs in a higher node propagates through the entire subtree, that is all the way down to the leaves below it. The variance can be improved by using random forests (see course II).

Advantages

- DT classifiers are particularly useful when the input is non-metric, that is when we have categorical variables. They also treat mixtures of numeric and categorical variables well.
- Due to their structural simplicity, DTs are easily interpretable.

8.1 Applying

In Matlab there exists the command `classregtree` for training, which returns an object (as in object oriented programming). To apply testing data, one can simply pass them to the object as an argument and a vector of predicted class labels is returned:

```
T           = classregtree(TREN, Grp.Trn);
R           = T(TEST);
bHit        = round(R)==Grp.Tst;
```

The output R is not only an integer value, but a floating-point value, whose after comma value represents to which side it leans. We use the round function to enforce a decision.

8.2 Exercise

Appendix C.13 gives an example running on synthetic data. We have tried to find a synthetic data set, where the tree is in fact at advantage over a linear classifier. For that purpose we have generated binary sets using the rounding function and the uniform random generator `rand`. Changing the constants that we added to the random generator can drastically favor one classifier. Why?

9 Combining Classifiers [Ensemble Classifiers] Alp p419, ch 17

The previously introduced classifiers (kNN, Linear Classifier, Decision Tree) attempt to obtain an optimal performance with a single classifier, e.g. with a 'perfect' (single) discrimination function. In contrast, the principle of combining classifiers is to use multiple less-than-perfect classifiers, each one with a 'mediocre' discrimination function for instance; these *base* classifiers (or base learners) are then combined to form a single (total) decision. The classifier that combines the base learners is called ensemble classifier or simply combiner. There are two principal motivations for combining classifiers:

1. We have measurements from separate sources, e.g. a visual and an audio signal, each with its own set of dimensions. Then, it is obvious to test whether a combination of separate classifiers, with each one geared toward those sources, performs better than a single classifier (it is not as obvious for the following motivation) - in this case aka data fusion. Subsection 9.1 introduces the basics combining classifiers.
2. We may try to solve the classification problem with a set of classifiers, whereby an individual classifier performs merely above chance level. By the combination of these 'opinions' we may obtain an expert advice, which is hopefully better than the expert advice of a single classifier. An example is given in subsection 9.2.

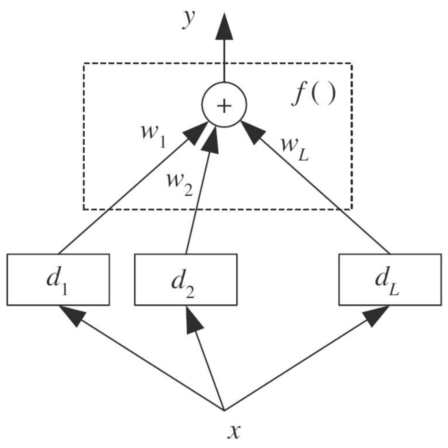


Figure 11: Simplest combination (ensemble) classifier. Input $\mathbf{x}$ feeds into L different base-learners, whose output d_j is combined using $f()$ to generate the final decision. In this example graph, all learners observe the same input; it may be the case that different learners observe different representations of the same input, as in bagging for instance. [Source: Alpaydin 2010, Fig 17.1]

General formulation: We have L base learners h_j $(j = 1, ..., L)$ and input vector $\mathbf{x}$. Each base learner makes a prediction $d_j(x)$, which in turn is combined with the other predictions to arrive at a final decision:

$$y = f(d_1, d_2, ..., d_L|\Phi), \tag{6}$$

where $f()$ is the combining function with Φ denoting its parameters. For a multi-class discrimination task each base learner generates K outputs and we then deal with a $K \times L$ matrix $d_{ji}(x)$ (number of classes $\times$ number of learners).

9.1 Voting

The simplest way to combine multiple classifiers is by voting, which corresponds to taking a linear combination of the learners

$$y_i = \sum_j w_j d_{ji} \quad \text{where} \quad w_j \geq 0, \ \sum_j w_j = 1. \tag{7}$$

This is also known as ensembles and linear opinion pools. In the simplest case, all learners are given equal weight ($w_j = 1/L$), which is also called "simple voting": it corresponds to taking an average. Other combination rules are

Median	$y_i = \mathrm{median_j\ d_{ji}}$	robust to outliers
Minimum	$y_i = \min_j d_{ji}$	pessimistic
Maximum	$y_i = \max_j d_{ji}$	optimistic
Product	$y_i = \prod_j d_{ji}$	veto power

If the outputs d_{ji} are not posterior probabilities, these rules require that outputs be normalized to the same scale. Note that after the combination rules, y_i do not necessarily sum up to 1.

If the data set consists of features obtained from different sources, then one should definitely try an ensemble classifier with a voting scheme as it does not involve any particular tuning, that is it comes at little effort to test this variant. For instance, we have data with audio and visual features: we train solely the visual features with one LDA and obtain the corresponding posterior values (3rd argument, see subsection 3.3), and we train solely the audio features with another LDA and obtain the corresponding posterior values. We then combine the two sets of posteriors with any rule that gives us the maximum performance.

9.2 Bagging

Bagging is a voting method whereby base learners h_j are made different by training them on different subsets of the training sets. Bagging can reduce variance and thus reduce the generalization error performance.

The subsets are generated by bootstrap, that is by drawing randomly a subset of samples from the training set with replacement (hence the name bagging = bootstrap aggregation). Given a training set X, we create B variants, $X_1, X_2, ..., X_B$, by uniformly sampling from X with replacement. (Because sampling is done with replacement, it is possible that some instances are drawn more than once and that certain instances are not drawn at all). One can use `randsample` to create different subsets of X, e.g.

```
for i = 1:nSub
    Ixr  = randsample(nTrnSamp, nSubSize);     % random sampling
    Xsub = X(Ixr,:);                            % select only first nSubSize of Ixr and thus X
    ...train a classifier on Xsub...
end
```

For each of the training set variants, X_i, a classifier h_i, is constructed. The final decision is in favor of the class predicted by the majority of the subclassifiers, $h_i, i = 1, 2, ..., B$.

By randomly selecting a subset, the individual base learners will be slightly different (remember motivation no. 2 above). To increase diversity, bagging works better, if the base learner is trained with an unstable algorithm, such as a decision tree, a single or multilayer perceptron, or a condensed NN. Unstable means that small changes in the training set cause a large difference in the generated learner, namely a high performance variance.

Bagging as such is a method worth trying as it also involves little complications. Bagging is successfully used in some applications (e.g. Kinect Microsoft motion recognition system), specifically together with decision trees, so-called 'random forests'.

9.3 Component Classifiers without Discriminant Functions DHS p498, s. 9.7.2, pdf 576

If we create an ensemble classifier, whose base learners consist of different classifier types, e.g. one is a LDA and the other is a kNN classifier, then we need adjust their outputs in particular if they do not compute discriminant functions. In order to integrate the information from the different (component) classifiers we must convert the their outputs into discriminant values. It is convenient to convert the classifier output $\tilde{g}_i$

to a range between 0 to 1, now g_i, in order to match them to posterior values of a (regular) discriminant classifiers. The simplest heuristics to this end are the following:

Analog (e.g. NN): *softmax* transformation:

$$g_i = \frac{e^{\tilde{g}_i}}{\sum_{j=1}^{c} e^{\tilde{g}_i}}. \tag{8}$$

Rank order (e.g. kNN): If the output is a rank order list, we assume the discriminant function is linearly proportional to the rank order of the item on the list. The values for g_i should thus sum to 1, that is normalization is required.

One-of-c (e.g. decision tree): If the output is a one-of-c representation, in which a single category is identified, we let $g_j = 1$ for the j corresponding to the chosen category, and 0 otherwise.

The table gives a simple illustration of these heuristics.

Analog value		Rank order		One-of-c	
$\tilde{g}_i$	g_i	$\tilde{g}_i$	g_i	$\tilde{g}_i$	g_i
0.4	0.158	3rd	$4/21 = 0.194$	0	0
0.6	0.193	6th	$1/21 = 0.048$	1	1.0
0.9	0.260	5th	$2/21 = 0.095$	0	0
0.3	0.143	1st	$6/21 = 0.286$	0	0
0.2	0.129	2nd	$5/21 = 0.238$	0	0
0.1	0.111	4th	$3/21 = 0.143$	0	0

Other normalization schemes are certainly possible too. The Matlab command `classify` returns the discriminant values as the 3rd argument, called 'posteriors', which are already normalized to a range between 0 and 1. Before combining those posteriors with the discriminant values from other component classifiers, it is useful to plot the posterior matrix to see what range of values we deal with.

9.4 Learning the Combination

Instead of choosing a combination rule (see table in subsection 9.1), we may try to optimize the combination stage by training a classifier on the discriminant values being combined. For instance, we train an 'optimization' classifier to combine the discriminant values for an LDA and a kNN classifier, for which the optimization classifier takes a $2 \times K$ matrix as input (2 because we have the LDA and the kNN classifier; K=number of classes) and returns a vector of length K as the final posterior. There are also other ways to combine component classifiers.

To provide a correct generalization performance, we need to train the base classifiers and the combination stage separately. That means we need to split the training set into a subset for training the base classifiers only, and a subset for the combination stage. Ultimately, it is more complex and requires more training data, but we may gain another few percent by cleverly combining the component classifiers and may thus beat any other classifier.

9.5 One-vs-All Classifier

One may also try to learn K classifiers, with each one discriminating one class versus all other classes (one-vs-all). When constructing such an ensemble classifier, one should pay attention to the class imbalance problem (subsection 5.4.1).

10 Non-Metric Classification DHS p394, ch 8, pdf 461

If data are nominal, meaning if they are discrete and without any natural notion of similarity or even ordering, then one uses *lists* of attributes.

A common approach is to specify the values of a fixed number of properties by a *property d-tuple*. For example, consider describing a piece of fruit by the four properties of color, texture, taste and smell. Then a particular piece of fruit might be described by the 4-tuple red, shiny, sweet, small, which is a shorthand for color = red, texture = shiny, taste = sweet and size = small. Such data can be classified with decision trees (section 8).

Another common approach is to describe the pattern by a variable length *string* of nominal attributes, such as a sequence of base pairs in a segment of DNA, e.g., 'AGCTTCAGATTCCA'; or the letters in word/text. In that case we use methods dealing with sequences, which we elaborate next.

10.1 Recognition with Strings DHS p413, s 8.5, pdf 481 ThKo p487, s 8.2.2

A particularly long string is denoted *text*. Any contiguous string text that is part of $\mathbf{x}$ is called a substring, segment, or more frequently a *factor* of $\mathbf{x}$. For example, 'GCT' is a factor of 'AGCTTC'. There is a large number of problems in computations on strings. The ones that are of greatest importance in pattern recognition are:

- String matching: Given $\mathbf{x}$ and $text$, test whether $\mathbf{x}$ is a *factor* of text, and if so, determine its position.
- Edit distance: Given two strings $\mathbf{x}$ and $\mathbf{y}$, compute the minimum number of basic operations - character insertions, deletions and exchanges - needed to transform $\mathbf{x}$ into $\mathbf{y}$.
- String matching with errors: Given $\mathbf{x}$ and $text$, find the locations in $text$ where the 'cost' or 'distance' of $\mathbf{x}$ to any factor of $text$ is minimal.
- String matching with the 'dont care' symbol: This is the same as basic string matching, but with a special symbol, $\varnothing$, the dont care symbol, which can match any other symbol.

We introduce only the first two.

10.1.1 String Matching Distance

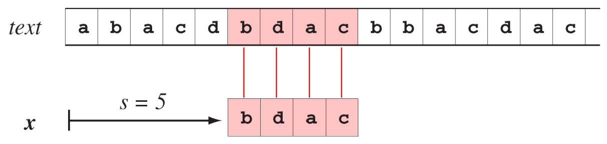


Figure 12: The general string-matching problem is to find all shifts s for which the pattern $\mathbf{x}$ appears in $text$. Any such shift is called valid. In this case $\mathbf{x} = "bdac"$ is indeed a factor of $text$, and $s = 5$ is the only valid shift. [Source: Duda,Hart,Storck 2001, Fig 8.7]

The simplest detector method is to test each possible shift, which is also called 'naive string matching'. A more sophisticated method, the Boyer-Moore algorithm, uses the matched result at one position to predict better possible matches, thus not testing every position and accelerating the search.

10.1.2 Edit Distance

The edit distance between $\mathbf{x}$ and $\mathbf{y}$ describes how many fundamental operations are required to transform $\mathbf{x}$ into $\mathbf{y}$. The fundamental operations are:

- substitutions: A character in $\mathbf{x}$ is replaced by the corresponding character in $\mathbf{y}$.
- insertions: A character in $\mathbf{y}$ is inserted into $\mathbf{x}$, thereby increasing the length of $\mathbf{x}$ by one character.
- deletions: A character in $\mathbf{x}$ is deleted, thereby decreasing the length of $\mathbf{x}$ by one character.

Let $\mathbf{C}$ be an $m \times n$ matrix of integers associated with a cost or distance and let $\delta(\cdot, \cdot)$ denote a generalization of the Kronecker delta function, having value 1 if the two arguments (characters) match and 0 otherwise. The basic edit-distance algorithm (algorithm 8) starts by setting $\mathbf{C}[0, 0] = 0$ and initializing the left column and top row of $\mathbf{C}$ with the integer number of steps away from $i = 0, j = 0$. The core of this algorithm finds

Algorithm 8 Edit distance. From DHS p486.

```
Initialization  $\mathbf{x}$, $\mathbf{y}$, $m \leftarrow length[\mathbf{x}], n \leftarrow length[\mathbf{y}]$
Initialization  $\mathbf{C}[0,0] = 0$
Initialization  For $i = 1..m, \mathbf{C}[i,0] = i$, End
Initialization  For $j = 1..n, \mathbf{C}[0,j] = j$, End
For $i = 1..m$
    For $j = 1..n$
        $Ins = \mathbf{C}[i-1,j] + 1;$                                      % insertion cost
        $Del = \mathbf{C}[i,j-1] + 1;$                                      % deletion cost
        $Exc = \mathbf{C}[i-1,j-1] + 1 - \delta(\mathbf{x}[i], \mathbf{y}[j])$    % no (ex)change cost
        $\mathbf{C}[i,j] = \min(Ins, Del, Exc)$                             % the minimum of the 3 costs
    End
End
Return $C[m,n]$
```

the minimum cost in each entry of $\mathbf{C}$, column by column (figure 13). Algorithm 8 is thus greedy in that each column of the distance or cost matrix is filled using merely the costs in the previous column.

As shown in figure 13, $\mathbf{x} = "excused"$ can be transformed to $\mathbf{y} = "exhausted"$ through one substitution and two insertions. The table shows the steps of this transformation, along with the computed entries of the cost matrix $\mathbf{C}$. For the case shown, where each fundamental operation has a cost of 1, the edit distance is given by the value of the cost matrix at the sink, i.e., $\mathbf{C}[7,9] = 3$.

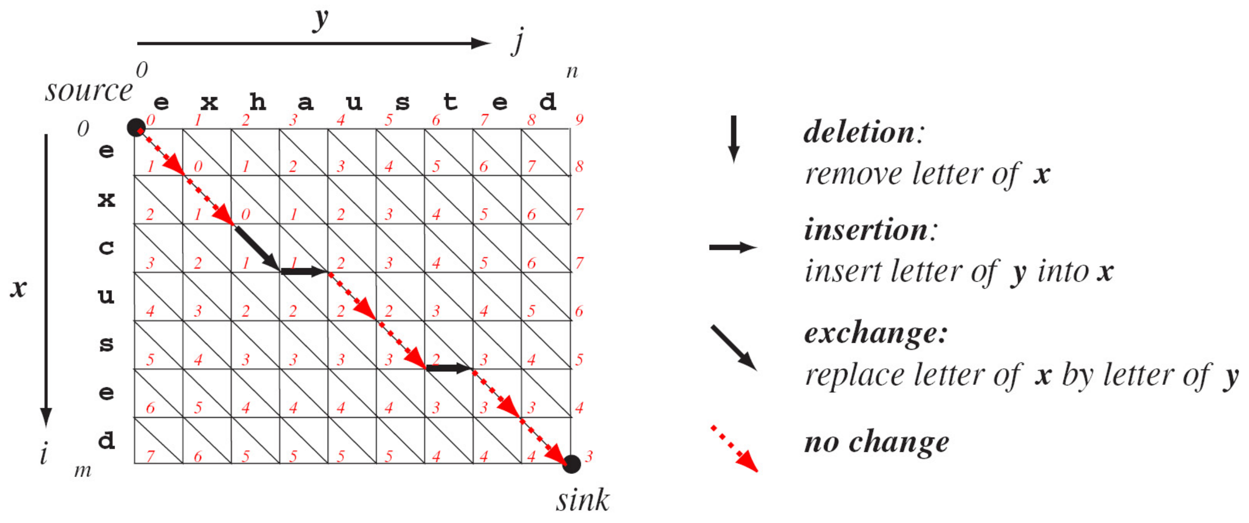

y
j
x
i
source
sink
0
n
m
e x h a u s t e d
e x c u s e d
deletion: remove letter of x
insertion: insert letter of y into x
exchange: replace letter of x by letter of y
no change

Figure 13: The edit distance calculation for strings $\mathbf{x}$ and $\mathbf{y}$ can be illustrated in a table. Algorithm 3 begins at source, $i = 0$, $j = 0$, and fills in the cost matrix $\mathbf{C}$, column by column (shown in red), until the full edit distance is placed at the sink, $\mathbf{C}[i = m, j = n]$. The edit distance between excused and exhausted is thus 3. [Source: Duda,Hart,Storck 2001, Fig 8.9]

The algorithm has complexity $O(mn)$ and is rather crude; optimized algorithms have $O(m+n)$ complexity only. Linear programming techniques can also be used to find a global minimum, though this nearly always requires greater computational effort.

Note: as mentioned in the introduction, the pattern can consist of any (limited) set of ordered elements, and not just letters. Example: The edit distance is sometimes applied in computer vision, specifically shape recognition, for which a shape is expressed as a sequence of classified segments.

11 Density Estimation

Density estimation is the characterization of a data distribution. Density estimation is in principal similar to clustering (section 6), where we had attempted to identify cluster centers in the entire dataset. In density estimation however, we rather focus on the distribution of individual variables (features) by trying to identify their modes (maxima).

One can distinguish between non-parametric and parametric methods, sections 11.1 and 11.2, respectively. In non-parametric methods, the distribution is merely transformed and we typically specify a single parameter for this transformation. In parametric methods we are more explicit: we specify the number of expected densities for instance.

11.1 Non-Parametric Methods Alp p165

In non-parametric methods, the distribution is observed through different 'windows' or 'local neighborhoods' placed across the range of the data. There are two methods of 'windowing'. In the histogramming method, the windows are called *bins* and all we do is to count the number of datapoints that lie within a bin, which results in a simple bar plot (subsection 11.1.1). In the kernel-estimation method, the window is called *kernel* or '*Parzen* window' and we take some weighted average for the points within, which results in a smooth distribution function - as opposed to the histogramming method (subsection 11.1.2).

11.1.1 Histogramming Alp p165 wiki: Histogram

Histogramming is the simplest kind of density estimation - and the fastest one. When we generate a histogram, we typically specify the range and bin width, which in Matlab can be conveniently specified with the function `linspace` - standing for linear spacing. The choice of range and bin boundaries affects of course the estimation outcome, whereby the bin width has a larger influence on the estimate. The estimate is zero if no instance falls in a bin; there occur discontinuities at bin boundaries.

Histogramming can be done in multiple dimensions too. For two dimensions, also known as a *bi-variate* histogram, we can employ the function `hist3`. In more dimensions, it becomes difficult to display the data.

Histogramming is so effective, one is tempted to generate this density estimate for the entire dataset. What is the limitation in analyzing high-dimensional data sets? The answer is a simple one and we will return to it during the exercise.

11.1.2 Kernel Estimator (Parzen Windows) Alp p167 ThKo p51 wiki: Kernel_density_estimation, Kernel_smoother

In kernel estimation, the window is typically placed equally spaced throughout the data range at specified points x. At each point x, the distribution x_t ($t = 1, .., N$, N=number of datapoints) is observed through a lens - metaphorically speaking: only points in the center are given 'attention' and points further in the periphery are given less 'attention'. In other words, at each specified point x the points of the distribution x_t are weighted by a function, the so-called *kernel* K.

When we plot the kernel, then it appears like a (symmetric) bump: the peak corresponds to the the center of the lens and the farther away we move, the lower becomes the function value; this is expressed as $K\big((x - x_t)/h\big)$, where the difference expresses the separation from the center x and the divisor h is the so called *bandwidth* and it controls the width of the lens. There are many different kernel functions, see for example wiki: Kernel_(statistics)#Kernel_functions_in_common_use. The most popular kernel function for density estimation is the Gaussian function:

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\Big(-\frac{1}{2}\Big[\frac{x-\mu}{\sigma}\Big]^2\Big) \tag{9}$$

where μ (greek mu) corresponds to the center (equal x) and σ (greek sigma) corresponds to the bandwidth h (wiki: Gaussian_function).

Returning to our formulation for the density estimate: it is expressed as the function $f(x)$, which consists of the sum of weighted values for each x_t obtained by the kernel function K with center x:

$$f(x) = \frac{1}{Nh} \sum_{t=1}^{N} K\Big(\frac{x - x_t}{h}\Big), \tag{10}$$

whereby the divisor Nh normalizes the function. In Matlab we can apply the function `ksdensity`. The function allows even to omit the specification of a bandwidth value, in which case it will be estimated by some simple rule.

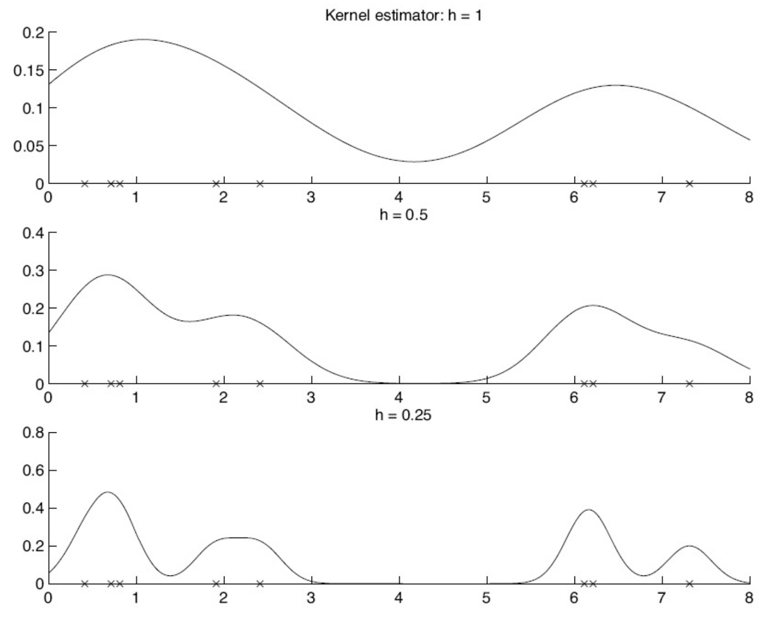


Figure 14: Kernel density estimation. Effect of different kernel widths h: from top to bottom: value of 1.0, 0.5 and 0.25, respectively. On the x-axis are shown the data points by x's. [Source: Alpaydin 2010, Fig 8.3].

11.2 Parametric Methods Alp p61

Parametric means we express the distribution by parameters, that is, by an equation, which is also called the *probability density function (PDF)* in the context of density estimation. The simplest parametric description is to take the mean μ and standard deviation σ of the data distribution, also know as the first-order statistics of the distribution. It is then obvious to take the Gaussian function again (equation 9) and one can employ this distribution estimate to determine the distance of new samples. This is exactly what is done for the Naive Bayes classifier (section 12).

Parameterizing a distribution with first-order statistics were ideal, if the distribution contained only a single mode (a *uni-modal* distribution). In practice, this is hardly true, as discovered above by histogramming the individual dimension (see previous subsection). But assuming a uni-modal distribution is simply done for computational convenience. There are situations however, where we wish to parameterize distributions with multiple modes, which is introduced next.

11.2.1 Gaussian Mixture Models (GMM)

Here we expect that the distribution consists of multiple modes, meaning we know that there are two or more sources giving rise to bi-modal or multi-modal distributions. The goal is then to locate the exact position of the source and to estimate the standard deviation it introduces into the data, again a case for the Gaussian function. We specify the number of Gaussians we expect and then try to find the corresponding centers and standard deviations, the μs and σs, respectively. One therefore calls this a Gaussian mixture model (GMM): the model simply adds the output of k Gaussian functions, whose means and standard deviations correspond to the location of the modes and to the width of the assumed underlying distributions.

To find the appropriate values for the individual means and standard deviations, one uses a so-called *Expectation-Maximization* (EM) algorithm. The algorithm gradually approaches the optimal values by a search procedure that is very akin to the k-Means algorithm (algorithm 5), hence the relation of density estimation to clustering.

We do not treat this topic in further detail and merely point out that GMMs can be modeled in Matlab with the command `gmdistribution` (available in statistics toolbox). We give an example in appendix C.14.2, without any further explanation.

11.3 Recapitulation

Density estimation is done for low dimensionality, typically 1 to 3 dimensions at most. It is used for analyzing variables or specific distributions of low dimensionality. For higher dimensionality the application of the methods becomes unfeasible. In case of histogramming we reach memory limitations and in case of the kernel-density estimation method (parametric or non-parametric) it is the complexity.

11.4 Exercise

Study the script in appendix C.14.1, which compares histogramming and Parzen window estimation, as well as an example of how to generate the latter method explicitly. We first focus on histogramming, then on the Parzen window method.

11.4.1 Histogramming

1. We have applied histogramming already before (command `histc`), but you should now understand exactly the discontinuities at boundaries. Specify a bin range that exceeds the range of data values, e.g. `Bin = linspace(min(X)-1,max(X)+1,40)`; then a bin range exactly on the extrema values `Bin = linspace(min(X),max(X),40)`; and then a smaller range, e.g. `Bin = linspace(min(X)+1,max(X)-1,40)`. Observe the count in the outside bins carefully each time.
2. Now look at the individual dimensions of your real data again. Have you applied `histc` properly the first time?
3. Look at the bivariate histogram for some of your dimensions using `hist3`.
4. Why is histogramming not practical for analysis of higher dimensional data? Hint: think of how would initialize the histogram? Assume you created the histogram with 10 bins per dimension and use the datatype single to save on memory - it requires 4 bytes only. Or asked directly how many dimensions could you analyze practically? **[Answer:** A 10-dimensional 10-bin histogram requires ca. 37 Gbs: $10^{10} * 4/(1024^3)$. Even with 9 dimensions this probably becomes impracticable already. In short: there are memory limitations.**].**

11.4.2 Parzen Window Estimation

1. How many points for estimation are generated in the line `[Pz Ve] = ksdensity(X);`? Change that number to some other number. First fewer, then more points.
2. What is the bandwidth that has been chosen for this estimation? Open the Matlab script for `ksdensity` and find the line that calculates the bandwidth. Hint: in the Matlab code the bandwidth is also called sigma (or sig).
3. Change the points at which the Parzen windows are applied.
4. Generate a plot with multiple estimations for varying bandwidths, e.g. loop the bandwidth 1 to 4. Plot the estimates into the same graph.
5. Can you find the actual density computation in the Matlab script `ksdensity`? Hint: it is in another script. **[Answer:** It is performed in script `statkscompute` in the function `dokernel`.**]**

6. In our own implementation we use the function `pdf` to generate the estimate. Write your own function in which the Gaussian is calculated.
7. Expand the implementation to some other kernel function. You can compare with `ksdensity` again.
8. Could one apply density estimation to data of higher dimensionality? **[Answer:** In principle yes, it becomes just unfeasible slow because the computational complexity depends on dimensionality.**]**

12 Naive Bayes Classifier (Linear Classifier II) wiki: Naive_Bayes_classifier

We now study a specific type of linear classifier in more detail, namely the Naive Bayes classifier. It is a fairly simple procedure, has theoretical elegance, but practically it is not quite as competitive as other classifiers. It maintains its place amongst the competition, when it comes to specific tasks or when sample size is small.

Naive Bayes assumes that the features (variables) are independent, meaning any information sharing between features is not considered, which earns the name 'naive'. Yet in praxis, many data sets contain features that are correlated to some extent, one can simply observe that with the covariance matrix.

One version of Naive Bayes classifier performs density estimation assuming uni-modal Gaussian distributions as introduced in subsection 11.2, namely by taking the mean and the standard deviations of the individual feature dimensions for each class (group). This is also a 'naive' assumption, because as we have seen previously, the distribution of feature values is often far from being a Gaussian function.

Figurative Example. In our country-guessing example, we would approximate the distribution of cars for each country by a separate density function and then determine our location by using the density functions only. For a given (spatial) location we compute the values for the different countries (from their individual functions), and the one that returns the highest value determines our choice of country.

Learning and Classifying: Taking figure 1 as an illustration, we would fit a 2D Gaussian to each data set - the two distributions were in fact generated with Gaussian functions. We then classify a new sample based on the parameters of these two Gaussian models: we compute the Gaussian function value for both classes and the larger function value then determines the preferred class label.

Gaussian Function in 2D (and nD): We have seen the 1D Gaussian function in equation 9 already, it has two parameters, a center μ and a variance σ. In 2D, we have those values for each axis: μ_x and μ_y as well as σ_x and σ_y:

$$g(x,y) = \frac{1}{2\pi\sigma_x\sigma_y\sqrt{1-\rho^2}} \exp\left(-\frac{1}{2(1-\rho^2)} \left(\left[\frac{x-\mu_x}{\sigma_x}\right]^2 + \left[\frac{y-\mu_y}{\sigma_y}\right]^2 - \frac{2\rho(x-\mu_x)(y-\mu_y)}{\sigma_x\sigma_y} \right) \right) \tag{11}$$

In addition there is the parameter ρ (rho), which is the correlation between X and Y and both variances are positive $\sigma_x > 0$ and $\sigma_y > 0$. This can also be expressed more compactly in matrix notation, whereby we form a vector for the mean parameters and a matrix for the variance parameters:

$$\boldsymbol{\mu} = \begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix} \qquad \text{and} \qquad \boldsymbol{\Sigma} = \begin{bmatrix} \sigma_x^2 & \rho\sigma_x\sigma_y \\ \rho\sigma_x\sigma_y & \sigma_y^2 \end{bmatrix}$$

Now $\boldsymbol{\Sigma}$ is the covariance matrix as encountered before. ρ can also be negative. Thus in matrix notation we write: DHS p33

$$g(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\Big[-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^t \boldsymbol{\Sigma}^{-1} (\mathbf{x}-\boldsymbol{\mu}) \Big] \quad \sim \quad N(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \tag{12}$$

which is also the formula for the *multivariate* Gaussian function (2 or more dimensions) wiki: Multivariate_normal_distribution. This formula again is short-noted as $N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$. We summarize some of the important expressions in that equation:

$\boldsymbol{\mu}$	mean vector, $E\big[[x_1, x_2, .., x_d]^t\big] = [\mu_1, \mu_2, .., \mu_t]^t$
$\boldsymbol{\Sigma}$	$d \times d$ covariance matrix, $\boldsymbol{\Sigma} = E[(\mathbf{x}-\boldsymbol{\mu})(\mathbf{x}-\boldsymbol{\mu})^t]$
$\lvert\boldsymbol{\Sigma}\rvert$	determinant of the covariance matrix
$\boldsymbol{\Sigma}^{-1}$	inverse of the covariance matrix
$(\mathbf{x}-\boldsymbol{\mu})^t\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})$	is also called Mahalanobis distance

$\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ are as introduce before; see also again Appendix A.1. The determinant and the inverse are algebraic operations which are beyond the scope of this course. The Mahalanobis distance is obtain by matrix multiplications.

Thus we train and apply our classifier as follows:

Algorithm 9 Naive Bayes Classifier. $k = 1, .., c$ (n$_{\text{classes}}$, K)

```
Training    ∀ $c$ classes ($\in \mathcal{D}^L$):
                mean $\boldsymbol{\mu}_k$, covariance $\boldsymbol{\Sigma}_k$, determinant $|\boldsymbol{\Sigma}_k|$, inverse $\boldsymbol{\Sigma}_k^{-1}$, prior $P(k)$
                $\rightarrow g_k$ as in equation 12
Testing     1) for a testing sample $\mathbf{x} \in \mathcal{D}^T$ determine $g(\mathbf{x})\ \forall\ c$ classes $\rightarrow g_k$.
            2) multiply each $g_k$ with the class prior $P(k)$: $f_k = g_k \cdot P(k)$
Decision    chose maximum of $f_k$: $\mathrm{argmax}_k\ f_k$
```

If the classes occur with uneven frequencies, we need to determine the frequency for each class, also called prior, and include this as pointed out in the training step and in step no. 2 in the testing phase.

The Naive Bayes classifier suffers from the same problems as mentioned before already (section 3). It can be difficult to compute the covariance matrix in particular for few training samples (small sample size problem).

12.1 Implementation

With the commands `cov`, `det` and `inv` (or `pinv`), one can conveniently build a Naive Bayes' classifier, see appendix C.15 for an example (see also ThKo p81). We did not include the prior in this code fragment, which one can generate with `Prior = Hgrp./sum(Hgrp(:))` for instance, where `Hgrp` is the sample count for each class (histogram of group variable, see appendix C.4).

The computation of the inverse is preferably done with the command `inv`, but if the inverse is difficult to compute, for instance due to small sample size, then one can estimate the inverse with `pinv`. If the inverse can still not be computed, then we need to we perform a dimensionality reduction (section 4).

As pointed out previously, Matlab offers to apply the Naive Bayes classifier with the command `classify` and the option 'diaglinear' (or 'diagquadratic'), see also section 3.

12.2 Recapitulation

The Naive Bayes classifier was introduced for instructional purposes only. The advantages and disadvantages are essentially the same as mentioned in section 3. The Naive Bayes has maintained its place in specific applications; and it is of theoretical value in pattern recognition (see also section 14).

12.3 Exercise

Synthetic Data

1. Study the example given in appendix C.15.
2. The example tests only a single example. Generate a loop to test all samples. Initialize the variable `Prob` to `zeros(nTst,nCat)`.
3. Place the learning part into a function called `f_NaiveLearn`, the testing part into a function `f_NaiveApply`.

Real Data

1. Try to apply the code to your real data. If it does not compute the inverse, i.e. an error is returned for the command `pinv`, then use the PCA first.

13 Support Vector Machines ThKo p119

Support Vector Machines (SVM) are sometimes assigned to the class of linear classifiers (e.g. with Linear Discriminant Analysis). They typically perform better than other linear classifiers but also require more tuning. They are designed as binary (two-category) classifiers; during the learning procedure the SVM focus on samples that are difficult to classify, somewhat akin to the hard negative mining technique mentioned before (subsection 5.4.4). The learning duration of SVMs is typically long and they may only work if the classes are reasonably well separable. The following characteristics make SVMs distinct from 'ordinary' linear classifiers:

1. Kernel function: The SVM uses such functions to project the data into a higher-dimensional space in which the data are hopefully better separable than in their original lower-dimensional space. Kernel functions can be Radial-Basis functions, quadratic,...
2. Support Vectors: The SVM uses only a few sample vectors for generating the decision boundaries and those are called support vectors. For a 'regular' linear classifier, there exist multiple reasonable decision boundaries, that separate the classes of the training set. For instance, the optimal hyperplane in figure 15 could actually show slightly different orientations. The SVM finds the hyperplane, that also gives a good generalization performance, whereby the support vectors are exploited to what is called 'maximizing the margin' - the two bidirectional arrows delineate the margin.

Although the SVM is a binary classifier, it can also be used for multiple categories using the technique mentioned in subsection 9.5.

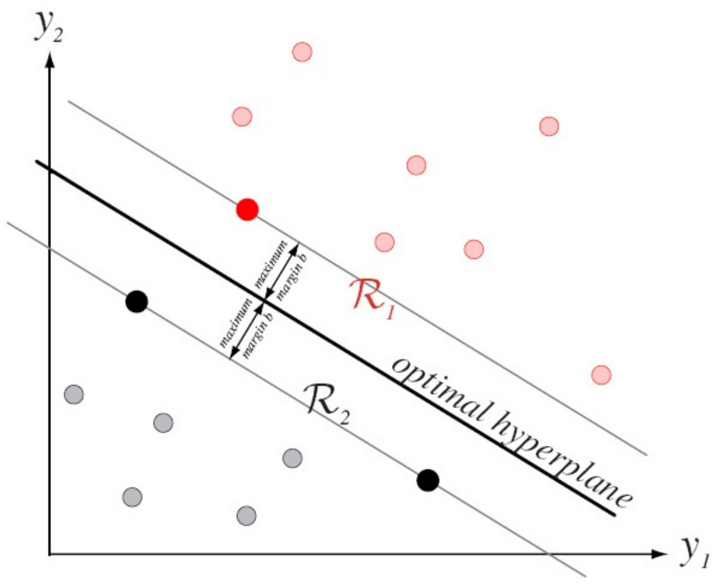


Figure 15: Training a support vector machine consists of finding the optimal hyperplane, that is, the one with the maximum distance from the nearest training patterns. The support vectors are those (nearest) patterns, a distance b from the hyperplane. The three support vectors are shown as solid dots. [Source: Duda,Hart,Storck 2001, Fig 5.19]

The SVM are too complex to code them quickly. The bioinformatic toolbox offers an implementation; one function trains the model, another one tests the data:

```
Svm    = svmtrain(TRN, Grp);      % returns a structure...
GrpTst = svmclassify(Svm, TST);   % ...which is fed together with the testing data
```

Appendix C.16 shows an example of how to apply these two commands. The command `classperf` can perform different actions: it can initialize the training procedure as well evaluate the classification output.

13.1 Recapitulation

Advantages SVMs are probably the best binary classifiers in average.
Disadvantages They require parameter tuning, as opposed to linear classifiers, but probably less so than neural networks. The learning duration is somewhat long. A SVM may not work well, if classes are not

reasonably separable. When SVMs are used in multi-class classification tasks, they loose somewhat their ’binary’ advantage.
Recommendation If a binary classification task needs to be optimized, it is worth trying a SVM.

14 Rounding the Picture DHS p84

14.1 Bayesian Formulation

A typical textbook on pattern classification (with mathematical ambition) starts by introducing the Bayesian formalism and its application to the decision and classification problem. We introduce this formalism late, because it can be better understood after one has employed the different classifiers. Bayes' formalism expresses a decision problem in a probabilistic framework:

$$\textbf{Bayes rule}: \quad P(\omega_j|\mathbf{x}) = \frac{p(\mathbf{x}|\omega_j)P(\omega_j)}{p(\mathbf{x})} \qquad posterior = \frac{likelihood \times prior}{evidence} \tag{13}$$

In natural language, see right side of equation: DHS p22,23
- Posterior: is the probability for the presence of a specific category ω_j in the sample $\mathbf{x}$. Alp p50
- Likelihood: is the computed value using the density function. In the example of the Naive Bayes classifier (section 12), it is the value of equation 12.
- Prior: is the probability for the category being present in general, that is, it is the frequency of its occurrence. We called this prior already (see algorithm 9 and subsection 12.1).
- Evidence: is the marginal probability that an observation $\mathbf{x}$ is seen (regardless of whether it is a positive or negative example) and ensures normalization. (This was not explicitly calculated.)

More formally: Given a sample, $\mathbf{x}$, the probability $P(\omega_j|\mathbf{x})$, that it belongs to class ω_j, is the fraction of the class-conditional probability density function, $p(\mathbf{x}|\omega_j)$, multiplied by the probability with which the class appears, $P(\omega_j)$, divided by the evidence $p(\mathbf{x})$. We can formalize evidence as follows:

$$p(\mathbf{x}) = \sum_{j=1}^{c} p(\mathbf{x}|\omega_j)P(\omega_j) = \sum(likelihood \times prior) = \quad \text{Normalizer to ensure} \sum_j P(\omega_j|\mathbf{x}) = 1 \tag{14}$$

14.1.1 Rephrasing Classifier Methods

Given the above Bayesian formulation, we can now rephrase the working principle of the three classifier types (sections 2, 3, 12) as follows:

k-Nearest-Neighbor (section 2): estimates the posterior values $P(\omega_j|\mathbf{x})$ directly, without attempting to compute any density functions (likelihoods); in short, it is a non-parametric method, because no effort is made to find functions, that approximate the density $p(\mathbf{x}|\omega_j)$.

kNN is a type of instance-based learning, or lazy learning where the function is only approximated locally and all computation is deferred until classification.

Naive Bayes Classifier (section 12): is essentially the simplest version of the Bayesian formulation and that classifier makes the following two assumptions in particular:

1. It assumes that the features are independent and identically drawn (i.i.e.), in short statistically independent. This is also called *Naive Bayes' Rule*. But often we do not know beforehand, whether the dimensions are uncorrelated.
2. It assumes that the features are Gaussian distributed ($\boldsymbol{\mu} \equiv \varepsilon[\mathbf{x}], \boldsymbol{\Sigma} \equiv \varepsilon[(\mathbf{x}-\boldsymbol{\mu})(\mathbf{x}-\boldsymbol{\mu})^t]$)

For most data, these are two strong assumptions because most data distributions are more complex. Despite those two strong assumptions, the Naive Bayes classifier often returns acceptable performance.

Discriminative Model (section 13; perceptron in course II): they are similar to the kNN approach in the sense that they do not require knowledge of the form of the underlying probability distributions. Some researchers argue, that attempting to find the density function is a more complex problem than trying to directly develop discriminants functions.

14.2 Parametric (Generative) vs. Non-Parametric (Discriminative)

Along with the Bayesian framework comes also the distinction between parametric and non-parametric methods (as already implied above and made in section 11). The parametric methods pursue the approximation of density distributions $p(\mathbf{x}|\omega_j)$ by functions with a few essential parameters. Non-parametric methods in contrast find approximations without any explicit models (and hence parameters), such as the kNN and the Parzen window. Chapters in textbooks are often organized according to this distinction. Here we summarize the typical characterization of methods:

Parametric	Multi-Variate Methods, (MLE, EM)
Semi-parametric	Clustering, k-means, (EM)
Non-parametric	Parzen, kNN, SVM, Decision Trees

Note 1: the semi-parametric classification I found in Alpaydin's textbook.
Note 2: EM: expectation-maximization algorithm (subsection 11.2.1); MLE: maximum-likelihood estimation algorithm. Both are density estimation methods. To be introduced in course II.
Note 3: the EM algorithm can obviously be classified differently, depending on the exact viewpoint.
Note 4: Bishop uses the terms Generative vs. Discriminative.

14.3 Other (Supervised) Statistical Classifiers

- Perceptron: is essentially a linear classifier with a different learning method (course II).
- Neural Networks (NN): are elaborations of the perceptron. The simplest versions are 3 layers networks, which can be regarded as consisting of 2 layers of perceptrons (course II).
- Hidden Markov Models (HMM): are especially suited for classifying dynamic patterns (course II).

14.4 Algorithm-Independent Issues

Curse of Dimensionality Intuitively, one would think that the more dimensions (attributes) we have at our disposal (through measurements), the easier it is to separate the classes (with any classifier). However, one often finds that with increasing number of dimensions, it is more challenging to find the appropriate separability, which is also refered to as the *curse of dimensionality*. On the one hand, if there are irrelevant and possibly obstructive dimensions, it may indeed be better to reduce the dimensionality (as introduced with the PCA for instance). On the other hand, the clever use of kernel functions, as in Support Vector Machines, shows that more parameters can also be useful.

No Free Lunch theorem **DHS p454** The theorem essentially states that no classifier technique is superior to any other one. Virtually any powerful algorithm, whether it be kNN, artificial NN, unpruned decision trees, etc. can solve a problem decently if sufficient parameters are created for the problem at hand.

The machine learning community tended to regard the most recently developed classifier methodology as a breakthrough in the quest of a (supposed) superior classification method. However, after decades of research, it has become clear (to most researchers) that no classifier model is absolutely better than any other one: each classifier has its advantages and disadvantages and their underlying, individual theoretical motivations are all justified in principle. In order to find the best performing classifier for a given problem, a practitioner simply has to test them all essentially.

A Varia

A.1 Distances (Metrics) wiki: Distance

There are different distance measures (metrics). The most common one is the Euclidean distance, calculated by the Pythagorean formula, as we know it from school. But there are also other distances and one formula expresses several of them, namely the *Minkowski* metric,

$$L_k(\boldsymbol{a}, \boldsymbol{b}) = \Big(\sum_{i=1}^{d} |a_i - b_i|^k\Big)^{1/k} \tag{15}$$

which is also referred to as the L_k norm; $\mathbf{a}$ and $\mathbf{b}$are two vectors. For the following values of k the distance is also known as:

- $k = 1$: L_1 norm, Manhattan distance, city-block distance, taxi-cab distance. In Matlab `mandist`.
- $k = 2$: L_2 norm, Euclidean distance. In Matlab `dist`.
- $k = \infty$: L_∞ norm, Chebyshev distance.

In most applications the Euclidean distance will do it: often, the use of other metrics does not change the classification or clustering results significantly. The Manhattan distance can also be tried, because it has the advantage that it calculates faster than the other metrics, because it measures only the sum of absolute distances. Because squaring and taking the square root in the Euclidean metric is costly, computations are sometimes done without taking the root, if the actual Euclidean distance value is not necessary.

Another popular distance metric is the *Mahalanobis* distance, which uses a covariance matrix S to arrive at a distance measure; see also section 3.1 for covariance matrix and appendix B for notation:

$$D_M(\mathbf{x}) = \sqrt{(\mathbf{x} - \boldsymbol{\mu})^T S^{-1} (\mathbf{x} - \boldsymbol{\mu})} \tag{16}$$

where $\boldsymbol{\mu}$ is a mean vector - for instance obtained by averaging over the samples for a class -; and $\mathbf{x}$ is a sample. The measure is used in the Naive Bayes classifier for instance (section 12). In the special case where the covariance matrix is the identity matrix (only 1s along the diagonal, 0 elsewhere), the Mahalanobis distance reduces to the Euclidean distance (L_2 norm above). In Matlab: `mahal`.

A.2 Whitening Transform

Input: `DAT`, a $n \times d$ matrix; output: `DWit`, the whitened data.

```
CovMx       = cov(DAT);        % covariance -> [nDim,nDim] matrix
[EPhi ELam] = eig(CovMx);      % eigenvectors & -values [nDim,nDim]
Ddco        = DAT * EPhi;      % DECORRELATION
LamS        = ELam.^(-0.5);
LamS        = diag(diag(LamS)); % ensure it's a diagonal matrix
DWit        = Ddco * LamS;     % EQUAL VARIANCE
% verify
COVwhi      = cov(Ddco);        % covariance of decorrelated data (should be a diagonal matrix)
Df          = diag(ELam)-diag(COVdco); % difference of diagonal elements
if sum(Df)>0.1,  error('odd: differences of diagonal elements very large!?');  end
```

See also `http://courses.media.mit.edu/2010fall/mas622j/whiten.pdf`

A.3 Programming Hints

Speed To write fast-running code in Matlab, one should exploit Matlab's matrix-manipulating commands in order to avoid the costly `for` loops (see for instance `repmat` or `accumarray`). Writing a kNN classifier can be conveniently done using the `repmat` command. However, when dealing with high dimensionality and large number of samples, exploiting this command can in fact slow down computation because the machine will spend a significant amount of time allocating the required memory for the large matrices. In that case, it may in fact be faster to maintain one for loop, and to use `repmat` only limitedly.

Vector Multiplication In mathematical notation a vector is assumed a column vector (see also appendix B). In Matlab however if you define a vector as `a=[1 2 3]`, it is a row vector - in fact as you write. To conform with mathematical notation, either transpose the vector immediately by using the transpose sign `'` (e.g., `a=[1 2 3]'`) or by using semi-colons (e.g., `a=[1; 2; 3];`); otherwise you are forced to change place of the transpose sign later when applying the dot product (`a*b'` instead of `a'*b`), in which case it appears reverse to the mathematical notation! Or simply use the command `dot`, for which the column/row orientation is irrelevant.

A.4 Mathematical Notation

The mathematical notation in this workbook is admittedly a bit messy, because I took equations from different textbooks. I did not make an effort to create a consistent notation, so that the reader can easily compare the equations to the original text. In the majority of textbooks a vector is denoted with a lower-case letter in bold face, e.g. $\mathbf{x}$; a matrix is denoted as an upper-case letter in bold face, e.g. $\mathbf{\Sigma}$. But there are deviations from this 'norm'.

A.5 Some Software Packages

MatLab	Unfortunately expensive and mostly available either in academia or industry.	
Weka	Free software package written in Java.	wiki: Weka_(machine_learning)
R	Free software package supposed to be a replacement for MatLab.	wiki: R_(programming_language)
Python	high-level language similar to Matlab.	wiki: Python_(programming_language)

A.6 Parallel Computing Toolbox in Matlab

Should you be lucky owner of the parallel computing toolbox in Matlab, then you can even use it on your home PC or laptop, as nowadays home PCs have multiple cores and that permits parallel computing in principle. It is relatively simple to exploit the parallel computing features in for-loops that are suitable for parallel processing: simply open a pool of cores, carry out the loop using the `parfor` command and then close the pool again.

```
matlabpool local 2;              % opening two cores (workers)
parfor i = 1:1000
    A(i) = SomeFunction(Dat, i);   % the data are manipulated in some function by counter i
end
matlabpool close;
```

The parfor loop can not be used if your computations in the loop depend on previous results, for example in an iterative process where `A(i)` depended on `A(i-1)`. It also only makes sense if the process that is supposed to be repeated in parallel is computationally intensive, otherwise the assignment of the individual steps to the corresponding cores (workers) may slow down the computation.

B Matrices (& Vectors): Multiplication and Special Matrices

There are several types of multiplications of vectors and matrices. We summarize only the ones that are used most frequently in 'basic' pattern recognition. First we need to distinguish between the 'orientation' of vectors, namely row and column vectors:

Row vector: 'horizontal' sequence of numbers, e.g. $A = \begin{bmatrix}1 & 5 & 3 & -2\end{bmatrix}$. In Matlab entered as follows: `A = [1 5 3 -2];`, that is as written in mathematical notation.

Column vector: 'vertical' sequence of numbers, e.g. $B = \begin{bmatrix}-1 \\ 4 \\ 2\end{bmatrix}$, which is often abbreviated using the 'transpose' B' or B^T, because it is more compact than the space consuming vertical notation. In Matlab entered with semi-colon, e.g. `B = [-1; 4; 2];` or as a row vector above, which is transposed using the $'$ sign, e.g. `B = [-1 4 2]';`.

If you have troubles remembering the two orientations, then think of 'row of seats' (horizontal) and 'columns of a temple' (vertical).

Note: In mathematical notation, a vector is assumed to be a column vector by default. It is thus recommended that vectors in Matlab are defined as column vectors immediately, such that multiplications in the code appear in accordance with the mathematical notation - otherwise it can become truly confusing.

B.1 Dot Product (Vector Multiplication) wiki: Dot_product

In this case, the orientation of vectors (row or column) does not matter. Given two vectors of equal length, $\mathbf{A} = [A_1, A_2, ..., A_n]$ and $\mathbf{B} = [B_1, B_2, ..., Bn]$, the dot product is defined as the summation of their element-wise products:

$$\mathbf{A} \cdot \mathbf{B} = \sum_{i=1}^{n} A_i B_i = A_1 B_1 + A_2 B_2 + ... + A_n B_n = \mathbf{A}'\mathbf{B} \tag{17}$$

where the left side ($\mathbf{A} \cdot \mathbf{B}$) is the matrix notation using the dot $\cdot$; the center uses the summation notation $\sum$; and the right side ($\mathbf{A}'\mathbf{B}$) is the matrix notation using the transpose. This is also known as the *scalar* product, because the result is a single number. In Matlab you can use the command `dot` to obtain the product, in which case the order and orientation of vectors does not matter. The dot product can also be regarded as a special case of the matrix multiplication (coming up next), in which case the orientation of the vectors does matter.

B.2 Matrix Multiplication wiki: Matrix_multiplication

A $n \times m$ matrix $\mathbf{A}$ consists of n rows and m columns. It is sort of intuitive that if you add or subtract a scalar value from a matrix, or multiply or divide a matrix by a scalar, that this is done for each element of the matrix. It is also intuitive that if the matrices are of the exact same size, then you can perform the operations with corresponding elements. In Matlab one uses `.*` and `./` to specify those element-wise operations - if not, it may generate completely wrong results.

It is less intuitive however, how the operations are carried out when we multiply two matrices of different sizes with each other. In order to perform such a matrix *product*, it requires that the number of columns of the first matrix is equal the number of rows of the second matrix: If $\mathbf{A}$ is an $n \times m$ matrix and $\mathbf{B}$ is an $m \times p$ matrix, then their matrix product $\mathbf{AB}$ is an $n \times p$ matrix, in which the m entries across the rows of $\mathbf{A}$ are multiplied with the m entries down the columns of $\mathbf{B}$. Remember the expression $nmmp \rightarrow np$ to memorize that requirement. Let us look at the special case when m equals 1, or n and p are equal 1:

Product of a Row and Column Vector:

- Row * Column (nmmp=1mm1→11): this corresponds to the dot product as introduced above. In Matlab: `A'*B`, but only if `A` and `B` were defined as row and vector respectively.
- Column * Row (nmmp=n11p→np): creates a $n \times p$ matrix, where n and p correspond to the vector lengths. Here, the elements are pairwise multiplied, no actual summation takes place.

The product of two matrices is then simply the application of the dot product in two loops, one iterating through the rows of the first matrix and the other iterating through the columns of the second matrix. Instead of formulating this more explicitly we give a code example, which includes several verification steps using the command `assert`:

```
clear;

a = [2 1 3 5]';      % column vector
b = [-1 2 0 3]';     % column vector

s   = a' * b         % dot/scalar product
M   = a  * b'        % matrix product

s1  = dot(a,b);      % dot product
s2  = dot(b,a);      % order does not matter
assert(all(s==s1), 'something is wrong');
assert(all(s1==s2), 'something is wrong');

A = [a'; 4 7 8 -3];
B = [b [2 6 -2 5]'];

A*B                  % the matrix product
B*A                  % works too - even though we reversed the order! Why?

%% --- Add another column to A. Calculate product explicitly.
A   = [A; [1 1 -1 7]];

[n m1]  = size(A);
[m2 p]  = size(B);
assert(m1==m2, 'Dimensionality not correct');
Mx      = nan(n,p);
for i = 1:n
    a   = A(i,:);
    for k = 1:p
        b = B(:,k);
        Mx(i,k) = dot(a,b);
    end
end
assert(all(all(Mx==(A*B))), 'not properly programmed');
```

- The loop is given merely for the purpose of illustrating the product of matrices. Of course, one would prefer to write merely `A*B` in a code.
- Why did then `B*A` work as well? **[Answer:** Because the size of A is equal the size of B' (transpose).**]**
- Observe what error you obtain when you insert the product `B*A` at the very end of the code again.

B.3 Appendix - Matrices **wiki: List_of_matrices**

One of the most important matrices is the identity matrix given by

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix},$$

meaning it is a matrix with values equal one along the diagonal entries and zero elsewhere.

Table 3: Matrices with explicitly constrained entries

Name	Explanation	Notes, References
Binary Matrix	see logical matrix.	
Boolean Matrix	see logical matrix.	
Diagonal Matrix	A square matrix with all entries outside the main diagonal equal to zero.	
Identity Matrix	as introduced above.	
Logical Matrix	A matrix with all entries either 0 or 1.	Synonym for binary or Boolean matrix
Sparse Matrix	A matrix with relatively few non-zero elements.	
Symmetric Mx	A square matrix which is equal to its transpose, $A = A^T(a_{i,j} = a_{j,i})$.	
Triangular Mx	A matrix with all entries above the main diagonal equal to zero (lower triangular) or with all entries below the main diagonal equal to zero (upper triangular).	

B.4 Reading

See references below for publication details.

(Alpaydin, 2010): An introductory book. Reviews some topics from a different perspective than Duda/Hart/Stork for example. It can be regarded as complementary to this workbook, but also complementary to other textbooks.

(Theodoridis and Koutroumbas, 2008): Contains the most practical tips of those books, that also intend to provide the theoretical background. Treats clustering very thoroughly - in more depth than any other textbook. Contains code examples.

(Witten et al., 2011): The most 'practical' machine learning book probably, but rather short on the motivation of the individual classifier types. It accompanies the 'Weka' machine learning suite (see link above).

(Duda et al., 2001): The professional book. A must have if one intends to further deepen one's knowledge about pattern classification. The book excels at relating the different classifier philosophies and emphasizes the similarities between classifiers and neural networks. Due to its 'age' (already 12 years for the 2nd version), it lacks in depth treatment for recent advances such as combining classifiers and graph methods for instance.

(Bishop, 2007): Another professional book. Contains beautiful illustrations and some historic comments, but aims rather at an advanced readership (upper-level undergraduate and graduate students).

(Jain et al., 2000): A review with some useful summaries. Should be available on the internet. Use scholar google.

Wikipedia: Always good for looking up definitions, formulations and different viewpoints. But wikipedia's 'variety' - originating from the contribution of different authors - is also its shortcoming: it is hard to comprehend the topic as a whole from the individual articles (websites). Hence, textbooks are still irreplaceable.

References

Alpaydin, E. (2010). *Introduction to Machine Learning*. MIT Press, Cambridge, MA, 2nd edition.

Bishop, C. (2007). *Pattern Recognition and Machine Learning*. Springer, New York.

Duda, R., Hart, P., and Stork, D. (2001). *Pattern Classification*. John Wiley and Sons Inc, 2nd edition.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York.

Jain, A., Duin, R., and Jianchang, M. (2000). Statistical pattern recognition: a review. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(1):4–37.

Theodoridis, S. and Koutroumbas, K. (2008). *Pattern Recognition*. Academic Press, 4th edition.

Witten, I., Frank, E., and Hall, M. (2011). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 3rd edition.

C Code Examples

C.1 Synthetic Data for Classification

```
clear; close all;

nC1     = 50;                  % number of samples for class 1
nC2     = 40;                  % number of samples for class 2
nDim    = 2;                   % dimensionality (number of dimensions)
D1      = rand(nC1,nDim);      % class 1 is uniformly dist. random noise
D2      = rand(nC2,nDim)+.5;   % class 2 is gaussian random noise
G1      = ones(nC1,1);         % group variable for class 1
G2      = ones(nC2,1)*2;       % group variable for class 2
ntSmp   = nC1 + nC2;           % number of total samples

%% --- Plotting the 2 Classes
figure(1); clf;
plot(D1(:,1), D1(:,2), 'r.'); hold on;
plot(D2(:,1), D2(:,2), 'g.'); title('All Data for Both Classes');

%% --- Generate 2 Folds (one half training, other testing)
FldC1   = crossvalind('kfold', nC1, 2);     % 1s and 2s randomly selected
FldC2   = crossvalind('kfold', nC2, 2);
bTrnC1  = FldC1==1;     % we take 1s as training, here for class 1...
bTrnC2  = FldC2==1;     % ...and here for class 2
bTstC1  = FldC1==2;     % we take 2s as testing, here for class 1...
bTstC2  = FldC2==2;     % ... and here for class 2
TRN     = [D1(bTrnC1,:); D2(bTrnC2,:)];     % entire training set
TST     = [D1(bTstC1,:); D2(bTstC2,:)];     % entire testing set
Grp.Trn = [G1(bTrnC1); G2(bTrnC2)];         % grouping variable for training
Grp.Tst = [G1(bTstC1); G2(bTstC2)];         % grouping variable for testing

%% --- Plotting Folds
figure(2); clf;
subplot(1,2,1), scatter(TRN(:,1), TRN(:,2), Grp.Trn); title('Training Samples');
subplot(1,2,2), scatter(TST(:,1), TST(:,2), Grp.Tst); title('Testing Samples');
```

C.2 Loading and Converting Data

```
clear; close all;
addpath('c:/Data/');        % adds a folder path to the variable path

%% ---- Import a Single File
DAT     = importdata('filename for data');
Grp     = importdata('filename for class/group label');
DAT     = single(DAT);         % if you do not need double precision
sfp     = 'c:/DataMat/DatPrep'; % where data will be save to
save(sfp,'DAT','Grp');          % will be save in compact matlab format

%% ---- Import Multiple Files
FOLD.DatRaw     = 'c:/DataRaw/';    % folder with different data files
FOLD.DatSave    = 'c:/DataMat/';    % folder where we save converted data

FilesAndDir = dir(FOLD.DatRaw);     % includes ('.' and '..')
FileNames   = FilesAndDir(3:end);   % omit first two dir ('.' and '..')
nFileNames  = length(FileNames);
DAT         = zeros(nFileNames,nDim);  % nDim: # of dimensions - if known already
Grp         = zeros(nFileNames,1);
for i = 1:nFileNames
    fp  = [FOLD.DatRaw FileNames(i).name]; % +2: jump '.' and '..'
    F   = load(fp);                 % a feature vector
    DAT(i,:)  = F;                 % assign to DAT matrix
    Grp(i)    = label;             % assign to group vector
end
%% --- Now Save
save(sfp, 'DAT', 'Grp');
```

C.3 Loading the MNIST dataset

Note that this function contains two subfunctions, `ff_LoadImg` and `ff_ReadLab`.

```
% Loads MNIST data and converts them from ubyte to single.
%
function [TREN LblTren TEST LblTest] = LoadMNIST()
filePath  = 'C:\DatOrig\MNST\';
Filenames = cell(4,1);
Filenames{1} = [filePath 'train-images.idx3-ubyte'];
Filenames{2} = [filePath 'train-labels.idx1-ubyte'];
Filenames{3} = [filePath 't10k-images.idx3-ubyte'];
Filenames{4} = [filePath 't10k-labels.idx1-ubyte'];

TREN    = ff_LoadImg(Filenames{1});
LblTren = ff_ReadLab(Filenames{2});
TEST    = ff_LoadImg(Filenames{3});
LblTest = ff_ReadLab(Filenames{4});

TREN    = single(TREN)/255.0;
TEST    = single(TEST)/255.0;

LblTren = single(LblTren);
LblTest = single(LblTest);
end % MAIN FUNCTION

%% ==========   Load Digits
function IMGS = ff_LoadImg(imgFile)
    fid     = fopen(imgFile, 'rb');
    idf  = fread(fid, 1, '*int32',0,'b');    % identifier
    nImg    = fread(fid, 1, '*int32',0,'b');
    nRow    = fread(fid, 1, '*int32',0,'b');
    nCol    = fread(fid, 1, '*int32',0,'b');
    IMGS    = fread(fid, inf, '*uint8',0,'b');
    fclose( fid );
    assert(idf==2051, '%s is not MNIST image file.', imgFile);
    IMGS = reshape(IMGS, [nRow*nCol, nImg])';
    for i=1:nImg
        Img         = reshape(IMGS(i,:), [nRow nCol])';
        IMGS(i,:) = reshape(Img, [1 nRow*nCol]);
    end
end % SUB FUNCTION

%% ==========   Load Labels
function Lab = ff_ReadLab(labFile)
    fid     = fopen(labFile, 'rb');
    idf     = fread(fid, 1, '*int32',0,'b');
    nLabs   = fread(fid, 1, '*int32',0,'b');
    ind     = fread(fid, inf, '*uint8',0,'b');
    fclose(fid);
    assert(idf==2049, '%s is not MNIST label file.', labFile);
    Lab = zeros(nLabs, 10);
    ind = ind + 1;
    for i=1:nLabs
        Lab(i,ind(i)) = 1;
    end
end % SUB FUNCTION
```

C.4 Analyse Your Data

The example generates fake data and grouping variables (`DAT` and `Grp`, respectively) - apart from those two lines, anything can be applied immediately.

```
%% ---- Display Simple Statistics
DAT             = rand(100, 5);         % fake data: 100 samples, 5 dimensions
[nSmp nDim]     = size(DAT);
fprintf('# Samples %d \t  # Dimensions %d\n', nSmp, nDim);
[dMin dMax] = deal(min(DAT(:)), max(DAT(:)));
[dMen dStd] = deal(mean(DAT(:)), std(DAT(:)));
fprintf('Raw Range %1.4f - %1.4f;  mean value %1.4f, std dev %1.4f', dMin, dMax, dMen, dStd);
if any(isnan(DAT(:))), warning('Data contain NaN'); end
if any(isinf(DAT(:))), warning('Data contain Inf entries'); end

%% ---- Analyse Grouping Variable
nCat        = 3;                                % # classes we expect
Grp         = round(rand(100,1)*(nCat-1));      % fake grouping variable
ClsLb       = unique(Grp);                      % unique group labels
nClsFound  = length(ClsLb);                    % # of groups we found
fprintf('Found %d groups\n', nClsFound);
Hgrp        = histc(Grp,ClsLb);
% --- Plotting
figure(1);clf;bar(Hgrp);
xlabel('Groups'); ylabel('Count');
title('Histogram of Group Count');
% --- Verification
assert(nClsFound==nCat, 'Class count not correct'); % check we have 10 classes
```

C.5 kNN Analysis Systematic

We assume here the distance measurements have been carried out already and saved in `GNN`, see code blocks in subsection 2.1.

```
%% --- Knn analysis systematic
kNN     = [3:2:11];
nNN     = length(kNN);      % number of NN we are testing
Pc      = zeros(nNN,1);     % init array
c       = 0;                % counter
for k = kNN
    HNN         = histc(GNN(:,1:k), 1:nCls, 2); % histogram for 5 NN
    [Fq LbTst]  = max(HNN, [], 2);              % LbTst contains class assignment
    Hit         = LbTst==Grp.Tst;
    c           = c + 1;
    Pc(c)       = nnz(Hit)/nTst*100;
end
figure(3);clf;
plot(kNN, Pc, '*-'); title('Perc correct for different NN');
xlabel('k (# of NN)');
ylabel('Perc Correct');
set(gca,'ylim',[0 100]);
```

C.6 Estimating the Covariance Matrix

```
clear;
D           = randn(10,3);
[nO nDim]   = size(D);              % # observations/dimensions

Mn   = mean(D,1);                % mean
```

```
Dc      = bsxfun(@minus, D, Mn);    % data - mean
Cv      = (Dc' * Dc) / (nO-1);      % covariance

%% ---- Verification
Vnc     = var(D,[],1);          % variance per dimension
Vnc-Cv(diag(true(nDim,1)))'

Cv2  = cov(D);
Cv-Cv2
```

C.7 Linear Classifier: Usage Example

This is a stand-alone example, which should work by copy/paste and should not require the PCA.

```
clear;
rng('default');
S1      = [2 1.5; 1.5 3];  % covariance for multi-variate normal distribution
MuCls1  = [0.3 0.5];       % two means (mus) for class 1
MuCls2  = [3.2 0.5];       %  "    "     "    "  class 2
PC1    = mvnrnd(MuCls1, S1, 50);   % training class 1
TEST    = mvnrnd(MuCls1, S1, 30);   % testing (class 1)
PC2    = mvnrnd(MuCls2, S1, 50);  % training class 2
TREN    = [PC1; PC2];       % training set
Grp     = [ones(size(PC1,1),1); ones(size(PC2,1),1)*2]; % group variable

Lb      = classify(TEST, TREN, Grp);
H       = histc(Lb,[1 2]);
pcCorrect   = H(1)/size(TEST,1);
fprintf('Perc correct %1.4f\n', pcCorrect*100);

%% -------- Plotting
figure(2); clf; hold on;
scatter(PC1(:,1), PC1(:,2), 'sb', 'markerfacecolor', 'b');
scatter(PC2(:,1), PC2(:,2), 'r^', 'markerfacecolor', 'r');
scatter(TEST(:,1), TEST(:,2), 'go', 'markerfacecolor', 'g');
title('blue and green are same class');

%% -------  Extracted from classify: 'linear'
[nObs nDim] = size(TREN);
ngroups   = 2;
prior       = [1 1]/ngroups;    % classes in equal proportion
gmeans = NaN(ngroups, nDim);
for k = 1:ngroups
    gmeans(k,:) = mean(TREN(Grp==k,:),1);
end

% Pooled estimate of covariance.  Do not do pivoting, so that A can be
% computed without unpermuting.  Instead use SVD to find rank of R.
[~,R]   = qr(TREN - gmeans(Grp,:), 0);
R       = R / sqrt(nObs - ngroups); % SigmaHat = R'*R
s       = svd(R);
if any(s <= max(nObs,nDim) * eps(max(s)))
    error(message('stats:classify:BadLinearVar'));
end
logDetSigma = 2*sum(log(s)); % avoid over/underflow

% MVN relative log posterior density, by group, for each sample
for k = 1:ngroups
    A       = bsxfun(@minus,TEST, gmeans(k,:)) / R;
    D(:,k)  = log(prior(k)) - .5*(sum(A .* A, 2) + logDetSigma);
end

% Decision
[maxD outclass] = max(D, [], 2);
if any(outclass-Lb), error('not same'); end
%% -----    Verify covariance (diff should be small)
cov(TREN-gmeans(Grp,:)) - R'*R
```

C.8 Study Cases for PCA

```
clear;

%% === Example 1: increasing values, noisy
Dat1                    = sort(rand(100,50)*50, 2); % increasing values
[coeff1 score1 lat1]    = pca(Dat1);
figure(2); clf;
subplot(2, 2, 1); plot(Dat1');
subplot(2, 2, 2); plot(coeff1(:, 1:3), 'linewidth', 2);  legend('1', '2', '3', 'location', 'best');
subplot(2, 2, 3); plot(lat1, '*-');
subplot(2, 2, 4); plot(coeff1(:, end-2:end));

%% === Example 2: straight lines with vertical offset
for i = 1:100,
    Dat2(i, :) = [1:50]*rand*5;
end
[coeff2 score2 lat2]    = pca(Dat2);
figure(3); clf;
subplot(2, 2, 1); plot(Dat2');
subplot(2, 2, 2); plot(coeff2(:, 1:3), 'linewidth', 2);  legend('1', '2', '3', 'location', 'best');
subplot(2, 2, 3); plot(lat2, '*-');
subplot(2, 2, 4); plot(coeff2(:, end-2:end));

%% === Example 3: Sigmoids
for i = 1:100
    Dat3(i, :) = normcdf([1:50], rand*2+22, 5)*(rand*0.1+1); % x, mu, sigma   + rand(1, 50)/10;
end
[coeff3 score3 lat3]    = pca(Dat3);
figure(4); clf;
subplot(2, 2, 1); plot(Dat3');
subplot(2, 2, 2); plot(coeff3(:, 1:3), 'linewidth', 2);  legend('1', '2', '3', 'location', 'best');
subplot(2, 2, 3); plot(lat3, '*-');
subplot(2, 2, 4); plot(coeff3(:, end-2:end));

%% === Example 4: Just noise
Dat4                    = randn(100, 50)*20;
[coeff4 score4 lat4]    = pca(Dat4);
figure(5); clf;
subplot(2, 2, 1); plot(Dat4');
subplot(2, 2, 2); plot(coeff4(:, 1:3), 'linewidth', 2);  legend('1', '2', '3', 'location', 'best');
subplot(2, 2, 3); plot(lat4, '*-');
subplot(2, 2, 4); plot(coeff4(:, end-2:end));
```

C.9 Example ROC

```
clear all; close all;
nSig    = 20;
nBkg    = 40;
Sig     = randn(nSig,1);      % signal
LbSig   = ones(nSig,1);
ntSmp   = nSig+nBkg;
%% -------- 3 Different Degrees of Separation
[ATPR AFPR] = deal([]);
for i = 1:3

    Bkg     = randn(nBkg,1)+i;          % background
    Dat     = [Sig; Bkg];               % data
    Lb      = [LbSig; ones(nBkg,1)*2];  % labels (1=signal,2=background)

    % ===== Moving threshold
    [aTPR aFPR c] = deal([],[],0);
    for t = -0.5:0.25:2,
        c       = c+1;                  % increase counter
        bLrg    = Dat > t;              % decision

        % ===== Evaluate
        bHit    = bLrg & Lb==1; % hits
        LbDec   = bLrg + 1;
        CM      = confusionmat(Lb, LbDec);
       CM2  = accumarray([Lb LbDec], 1, [2 2]); % confusion matrix
        assert(all(CM(:)==CM2(:)), 'CMs not the same');

        aTPR(c)     = CM(1,1)/nSig;
        aFPR(c)     = CM(2,1)/nBkg;     % coordinates FLIPPED!

        % --- Plotting
        figure(1); imagesc(CM,[0 ntSmp]);  title(sprintf('%d  %1.2f',i,t));
        colorbar;
        pause();

    end

    ATPR(i,:)   = aTPR;
    AFPR(i,:)   = aFPR;

    %%
    figure(2);clf;
    plot(aFPR, aTPR, 'g*');
    set(gca,'xlim', [0 1], 'ylim', [0 1]);
    pause();

end
%% ------   ALL ROCs
figure(3);clf; hold on
for i = 1:3
    plot(AFPR(i,:), ATPR(i,:), '*-');
end
set(gca,'xlim', [0 1], 'ylim', [0 1]);
```

C.10 Utility Functions

```
function Gb = f_GbSingle(nSingle)
Gb     = nSingle*4/(1024^3);
fprintf('%.2f Gb ', Gb);
end
```

C.11 K-Means: Applying and Example

```
clear;
nP      = 20;
X       = [randn(nP,2)+ones(nP,2); randn(nP,2)-ones(nP,2)];
nP      = size(X,1);
nCls    = 2;
%% ---- Kmeans
[Lb CtrMb] = kmeans(X, nCls, 'dist','city', 'rep',5, 'disp','final');
% ---- Cluster info
IXC   = cell(nCls,1);
for i = 1:nCls
    IXC{i}  = find(Lb==i);
end

%% ---- The Principle
IxCtr   = randsample(nP,2);
Ctr     = X(IxCtr,:);       % initial centroids
D       = zeros(size(X));
minErr  = 0.1;
mxIter  = 100;
for i = 1:mxIter
    % === Distances
    for c = 1:nCls
        ctr     = Ctr(c,:);  % one centroid [1 nDim]
        Df      = bsxfun(@minus, ctr, X);
        D(:,c)  = sum(Df.^2,2);
    end
    % === Find Nearest
    [v IxMin] = min(D,[],2);
    for c = 1:nCls
        Ctr(c,:)    = mean(X(IxMin==c,:));
    end
end
% ---- Cluster info
IXC2  = cell(nCls,1);
for i = 1:nCls
    IXC2{i}  = find(IxMin==i);
end

%% ---- Plotting
figure(1); clf;
M   = colormap;
Mr  = M(randsample(64,64),:);
subplot(1,2,1); hold on;
    for i = 1:nCls
        plot(X(IXC{i},1),X(IXC{i},2),'.', 'color', Mr(i,:));
    end
    plot(CtrMb(:,1),CtrMb(:,2),'kx');
    plot(Ctr(:,1),Ctr(:,2),'ro');

subplot(1,2,2); hold on;
    for i = 1:nCls
        plot(X(IXC2{i},1),X(IXC2{i},2),'.', 'color', Mr(i,:));
    end
    plot(CtrMb(:,1),CtrMb(:,2),'kx');
    plot(Ctr(:,1),Ctr(:,2),'ro');
```

C.12 Hierarchical Clustering

```
clear;
nP      = 20;
rng('default');
%%                      All Random
PtsRnd  = rand(nP,2);   % all random
%%                      Arc & Square Grid
degirad = pi/180;
wd      = 45*degirad;
nap     = 10;
yyarc   = cos(linspace(-wd,wd,nap))*(0.5)+0.4;
xxarc   = linspace(.15,.85,nap);
nsp     = 5;
yysqu   = repmat(linspace(0.1,0.3,nsp),nsp,1); yysqu = yysqu(:);
xxsqu   = repmat(linspace(0.3,0.7,nsp),1,nsp);
PtsPat  = [xxarc' yyarc'];
PtsPat  = [PtsPat; [xxsqu' yysqu]];     % append
%%                      Clustering Random
DisRnd  = pdist(PtsRnd);                    % pairwise distances
LnkRnd  = linkage(DisRnd, 'single');
[Ln2Rnd NConRnd] = f_LnkTrans(LnkRnd);
ClsRnd  = cluster(LnkRnd, 'cutoff', 0.29, 'criterion', 'distance'); % 1.14);
DisLnk  = sort(LnkRnd(:,3), 'descend');
DMrnd   = squareform(DisRnd);
DMrnd(diag(true(nP,1))) = inf;
[DMrndO ORnd]   = sort(DMrnd,2);
NNdi            = DMrndO(:,1);
[mxNN1 ixNN1mx] = max(NNdi);
%%                      Clustering Pattern
DisPat  = pdist(PtsPat);
LnkPat  = linkage(DisPat, 'single');
[Ln2Pat NConPat] = f_LnkTrans(LnkPat);
ClsPat  = cluster(LnkPat, 'cutoff', 1.15);
ClsPat  = cluster(LnkPat, 'cutoff', 0.11, 'criterion', 'distance');
%%                      General Stats
fprintf('#Cls Rnd %d\n', max(ClsRnd(:)));
fprintf('#Cls Pat %d\n', max(ClsPat(:)));
mxl = max([LnkRnd(:,3); LnkPat(:,3)])*1.05; % y-limit
%%                      Plotting
[rr cc] = deal(3,2);
figure(1); clf;
subplot(rr,cc,1);
    scatter(PtsRnd(:,1), PtsRnd(:,2), 100, ClsRnd, 'filled');
    set(gca, 'xlim', [0 1]);
    set(gca, 'ylim', [0 1]);
    p_MST(PtsRnd, LnkRnd, ClsRnd);
    title('Random', 'fontweight', 'bold', 'fontsize', 12);
    plot(PtsRnd(ixNN1mx,1), PtsRnd(ixNN1mx,2), 'k*');
subplot(rr,cc,2);
    scatter(PtsPat(:,1), PtsPat(:,2), 100, ClsPat, 'filled');
    set(gca, 'xlim', [0 1]);
    set(gca, 'ylim', [0 1]);
    p_MST(PtsPat, LnkPat, ClsPat);
    title('Pattern', 'fontweight', 'bold','fontsize', 12);
subplot(rr,cc,3);
    [HRnd TRng] = dendrogram(LnkRnd);
    set(gca,'ylim',[0 mxl], 'fontsize', 7);
subplot(rr,cc,4);
    [HPat TPat] = dendrogram(LnkPat, 40);
    set(gca,'ylim',[0 mxl], 'fontsize', 7);
subplot(rr,cc,5);
    p_MST2(PtsRnd, Ln2Rnd, ClsRnd, 'entirenum');
    %plot(DisLnk, '.-');
subplot(rr,cc,6);
    p_MST2(PtsPat, Ln2Pat, ClsPat, 'entire');
```

C.12.1 Three Functions

Now follow 3 functions for the above script. The first one rearranges the linkage output. The remaining two ones are plotting functions.

• Linkage transform:

```
%TRANSZ Translate output of LINKAGE into another format.
%   This is a helper function used by DENDROGRAM and COPHENET.

%   In LINKAGE, when a new cluster is formed from cluster i & j, it is
%   easier for the latter computation to name the newly formed cluster
%   min(i,j). However, this definition makes it hard to understand
%   the linkage information. We choose to give the newly formed
%   cluster a cluster index M+k, where M is the number of original
%   observation, and k means that this new cluster is the kth cluster
%   to be formed. This helper function converts the M+k indexing into
%   min(i,j) indexing.
function [Z Ncon] = f_LnkTrans(Z)

nL = size(Z,1)+1;       % # of leaves

for i = 1:(nL-1)
    if Z(i,1) > nL,     Z(i,1) = traceback(Z,Z(i,1));   end
    if Z(i,2) > nL,     Z(i,2) = traceback(Z,Z(i,2));   end
    if Z(i,1) > Z(i,2),Z(i,1:2) = Z(i,[2 1]);          end
end

Pairs   = Z(:,1:2);
Ncon    = histc(Pairs(:),1:nL);     % # of connections/links

%%
function a = traceback(Z,b)

nL = size(Z,1)+1;   % # of leaves

if Z(b-nL,1) > nL,  a = traceback(Z,Z(b-nL,1));
else                a = Z(b-nL,1);                  end
if Z(b-nL,2) > nL,  c = traceback(Z,Z(b-nL,2));
else                c = Z(b-nL,2);                  end

a = min(a,c);
```

• Plotting MST, version I:

```
% Plots minimum spanning tree (single-link clustering) for all points
% and for the individual clusters of Cls.
%
function [] = p_MST(Pts, Lnk, Cls, type)
if ~exist('type', 'var'),  type = '';  end
hold on;
nPtot   = size(Pts,1);
nL      = size(Lnk,1);
if nPtot~=(nL+1),  error('Lnk probably not correct: #Pts %d,  #Lnk %d');  end
if nPtot==1,  pp_Singleton(Pts);  return;  end
Dis     = Lnk(:,3);         % distances
Lnk     = Lnk(:,1:2);       % cluster indices (ix to points and intermed clusters)
maxDist = max(Dis);
Sim     = 1.1-Dis./maxDist; % similarity for linewidth
if any(Sim<eps),
    warning('linewidth < 0:  %1.5f', min(Sim));
end
%% ============ ENTIRE MST
Cen     = zeros(nL,2);
LnkVec  = zeros(nL,2,2,'single');
for i = 1:nL
    Ixp         = Lnk(i,:);             % pair indices
    bLef        = Ixp<=nPtot;           % leaves
    if all(bLef),                       % both are leaves (points)
```

```
        Xco   = Pts(Ixp,1);
        Yco   = Pts(Ixp,2);
    elseif sum(bLef)==1                 % one is a leaf (point), the other a cluster
        if bLef(1),     ixp = Ixp(1);   ixc = Ixp(2);
        else            ixp = Ixp(2);   ixc = Ixp(1);
        end
        Xco  = [Pts(ixp,1); Cen(ixc-nPtot,1)];
        Yco   = [Pts(ixp,2); Cen(ixc-nPtot,2)];
    else                                % both are clusters
        Xco   = Cen(Ixp-nPtot,1);
        Yco  = Cen(Ixp-nPtot,2);
    end
    Cen(i,:)        = mean([Xco Yco],1);
    LnkVec(i,:,:)   = [Xco Yco];
    % --- prints entire tree if desired
    if strcmp(type, 'entire')
        hp      = plot(Xco, Yco, 'color', ones(1,3)*0.5, 'linestyle', '-');
        set(hp, 'linewidth', Sim(i)*4);
    end
end

%% ============ CLUSTER MST
if iscell(Cls),nCls = length(Cls);
else          nCls = max(Cls);
end
for i = 1:nCls
    if iscell(Cls),     IxG = Cls{i};  % pt ixs of cluster (group)
    else                IxG = find(Cls==i);
    end
    szG     = length(IxG);              % group size (#Pts)
    if szG==1,  pp_Singleton(Pts(IxG,:));  continue;  end
    Brg     = [];
    for k = 1:szG
        bOcc    = Lnk==IxG(k);          % find leafs in tree
        IxOcc   = find(sum(bOcc,2));    % indices
        IxL     = Lnk(IxOcc,:);
        IxB     = sum(IxL,2)+nPtot;
        Brg     = [Brg; setdiff(IxL(:),IxG(k))];
        for l = IxOcc
            Xco = LnkVec(l,:,1);
            Yco = LnkVec(l,:,2);
            hp  = plot(Xco, Yco, 'color', 'k');
            set(hp, 'linewidth', Sim(l)*4);
        end
    end
    B   = false(nL,2);
    for k = 1:length(Brg)
        if Brg(k)<=nPtot, continue; end
        B(Lnk==Brg(k))  = true;
    end
    IxB     = []; % find(B(:,1)&B(:,2));
    for l = IxB'
        Xco = LnkVec(l,:,1);
        Yco = LnkVec(l,:,2);
        hp  = plot(Xco, Yco, 'color', 'g');
        set(hp, 'linewidth', Sim(l)*4);
    end
    % --- connect group's center point to remaining points
    PtsSel  = Pts(IxG,:);
    cen     = mean(PtsSel,1);
    for k = 1:szG
        plot([PtsSel(k,1) cen(1)], [PtsSel(k,2) cen(2)], 'color', ones(1,3)*0.7);
    end
end

%% ------------ Singleton Point
function [] = pp_Singleton(Pts)
```

```
plot(Pts(1), Pts(2), 'ko', 'markersize', 5);
plot(Pts(1), Pts(2), 'ko', 'markersize', 10);
```

• Plotting MST, version II:

```
% Plots minimum spanning tree (single-link clustering) for all points
% and for the individual clusters of Cls.
% sa p_MST
function [] = p_MST2(Pts, Lnk, Cls, type)
if ~exist('type', 'var'),  type = '';  end
hold on;
nPtot   = size(Pts,1);
nL      = size(Lnk,1);
if nPtot~=(nL+1),  error('Lnk probably not correct: #Pts %d,  #Lnk %d');  end
if nPtot==1,  pp_Singleton(Pts);  return;  end
Dis     = Lnk(:,3);         % distances
Lnk     = Lnk(:,1:2);       % cluster indices (ix to points and intermed clusters)
maxDist = max(Dis);
Sim     = 1.1-Dis./maxDist; % similarity for linewidth

%% ============ ENTIRE MST
Cen     = zeros(nL,2);
%LnkVec  = zeros(nL,2,2,'single');
for i = 1:nL
    Ixp   = Lnk(i,:);             % pair indices
    Xco    = Pts(Ixp,1);
    Yco     = Pts(Ixp,2);
    Cen(i,:)= mean([Xco Yco],1);
    %LnkVec(i,:,:)   = [Xco Yco];
    % --- prints entire tree if desired
    if strfind(type, 'entire')
        hp      = plot(Xco, Yco, 'color', ones(1,3)*0.5, 'linestyle', '-');
        set(hp, 'linewidth', Sim(i)*4);
    end
end
%% ============
if strfind(type, 'num')
    for i = 1:nPtot
        Pt  = double(Pts(i,:));
        text(Pt(1), Pt(2), num2str(i), 'fontsize', 8);
    end
end

return

%% ------------ Singleton Point
function [] = pp_Singleton(Pts)

plot(Pts(1), Pts(2), 'ko', 'markersize', 5);
plot(Pts(1), Pts(2), 'ko', 'markersize', 10);
```

C.13 Example Decision Tree

```
clear;
nSmp    = 2500;
nTrn    = 2000;
nTst    = nSmp-nTrn;
nDim    = 2;
nCls    = 3;
C1      = round(rand(nSmp,nDim)+.5);
C2      = round(rand(nSmp,nDim));
C3      = round(rand(nSmp,nDim)-.1);
IxTrn   = 1:nTrn;
IxTst   = setdiff(1:nSmp,IxTrn);
TREN    = [C1(IxTrn,:); C2(IxTrn,:); C3(IxTrn,:)];  % training set
TEST    = [C1(IxTst,:); C2(IxTst,:); C3(IxTst,:)];  % testing set
GrpMx   = repmat(1:nCls,nTrn,1);                    % groups as matrix
Grp.Trn = GrpMx(:);                                 % now as vector
GrpMx   = repmat(1:nCls,nTst,1);
Grp.Tst = GrpMx(:);

%% =====    Classify with Tree
T           = classregtree(TREN, Grp.Trn);
R           = T(TEST);
bHit        = round(R)==Grp.Tst;
pcTree      = nnz(bHit)/size(TEST,1);
fprintf('Perc correct for Tree %1.4f\n', pcTree*100);

%% =====    Classify Linearly
Lb          = classify(TEST, TREN, Grp.Trn);
bHit        = Lb==Grp.Tst;
pcLin       = nnz(bHit)/size(TEST,1);
fprintf('Perc correct for LinC %1.4f\n', pcLin*100);

if pcTree>pcLin,        fprintf('***** Tree wins! *****\n');
elseif pcTree<pcLin,    fprintf('***** Linc wins! *****\n'); end
```

C.14 Example Density Estimation

C.14.1 Histogramming and Parzen Window

```
clear
rng('default');
X       = [randn(30,1)*5; 10+rand(60,1)*8]; % synthetic data
nP      = length(X);                       % number of data points
Bin     = linspace(-20,20,40);             % bins for histogram

H       = histc(X, Bin);                  % histogramming
[Pz Ve] = ksdensity(X);                 % parzen window

%% ---- Plotting
figure(1); clf;
bar(Bin, H, 'histc'); hold on;
plot(Ve, Pz*nP, 'g');
plot(X, zeros(nP,1)-.5,'.');
legend('Histogram', 'Density Est', 'location', 'northwest');
set(gca,'ylim', [-.8 max(H(:))]);

%% ==== Own Implementation
bandWth = 1;
PtEv    = linspace(X(1),X(end),nP);        % locations of evaluation
PzOwn   = zeros(nP,1);
for i = 1:nP
    PzOwn(i)   = sum(pdf('Normal', X, PtEv(i), bandWth)) / (nP*bandWth);
end
PzMlb      = ksdensity(X,PtEv,'width',bandWth);% for comparison
% --- Plotting
figure(2);clf;
plot(PtEv, PzOwn, 'g.'); hold on;
plot(PtEv, PzMlb, 'b');
```

C.14.2 Gaussian Mixture Model

```
clear
rng('default');
X       = [randn(30,1)*5; 10+rand(60,1)*8];     % synthetic data
nP      = length(X);                            % number of data points
EvPt    = linspace(min(X),max(X),120);

Ogm     = gmdistribution.fit(X,2);              % we assume 2 peaks
Gm      = pdf(Ogm,EvPt');                       % create the estimate
[Pz Ve] = ksdensity(X, EvPt);                   % parzen window for comparison

%% ---- Plotting
figure(1); clf; hold on;
plot(Ve, Gm*nP,'.m');
plot(Ve, Pz*nP, 'g');
plot(X, zeros(nP,1)-.5,'.');
legend('GMM', 'Parzen Win', 'location', 'northwest');
set(gca,'ylim', [-.8 max([Ve(:); max(Gm(:))])]);
```

C.15 Example Naive Bayes

```
clear;
rng('default');
S1      = [2 1.5; 1.5 3];  % covariance for multi-variate normal distribution
MuCls1  = [0.3 0.5];       % two means (mus) for class 1
MuCls2  = [3.2 0.5];       %  "    "     "    "  class 2
PC1    = mvnrnd(MuCls1, S1, 50);   % training class 1
TEST    = mvnrnd(MuCls1, S1, 30);   % testing (class 1)
PC2    = mvnrnd(MuCls2, S1, 50);  % training class 2
TREN    = [PC1; PC2];       % training set
Grp     = [ones(size(PC1,1),1); ones(size(PC2,1),1)*2]; % group variable

%% ==========   NAIVE BAYES     =============
[nCat nDim]  = deal(2,2);
% ===== Build class information for TRAINING set:
AVG          = zeros(nCat,nDim);
[COV COVInv] = deal(zeros(nCat,nDim,nDim));
CovDet       = zeros(nCat,1);
for k = 1:nCat
    TrnCat         = TREN(Grp==k, :);    % [nCatSamp nDim]
    AVG(k,:)        = mean(TrnCat);         % [nCat nDim]
    CovCat          = cov(TrnCat);          % [nDim nDim]
    COV(k,:,:)      = CovCat;               % [nCat, nDim, nDim]
    CovDet(k)       = det(CovCat);          % determinant
    COVInv(k,:,:)   = pinv(CovCat);      % p inverse
end
% ===== Testing a (single) sample with index ix (from TESTING set):
Prob = zeros(nCat,1);                       % initialize probabilites
for k = 1:nCat
    detCat  = abs(CovDet(k));               % retrieve class determinant
    CovInv  = squeeze(COVInv(k,:,:));       % retrieve class inverse
    fct     = 1/( ( (2*pi)^(nDim/2) )*sqrt(detCat) +eps);
    Df      = AVG(k,:)-TEST(1,:);           % diff between avg and sample
    Mah     = (Df * CovInv * Df')/2;        % Mahalanobis distance
    Prob(k) = fct * exp(-Mah);            % probability for this class
end
[mxc ixc] = max(Prob);                      % final decision (class winner)
```

C.16 Example SVM

```
clear;
load fisheriris
DAT             = [meas(:,1), meas(:,2)];
Grp             = ismember(species,'setosa');
[bTrain bTest]  = crossvalind('holdOut',Grp);
Cp              = classperf(Grp);

%% ------   Classifying and Plotting
figure(1); clf;
Svm   = svmtrain(DAT(bTrain,:), Grp(bTrain), 'showplot', true);
title(sprintf('Kernel Function: %s',...
              func2str(Svm.KernelFunction)),...
              'interpreter','none');
pause(); % displays the data without classified points
Res     = svmclassify(Svm, DAT(bTest,:), 'showplot', true);

classperf(Cp, Res, bTest);
Cp.CorrectRate
```

D Example Questions

D.1 Questions

1. You are given a completely new data set of medium size (a few hundred samples in total, up to dimensionality 50; with class labels). What was suggested (in the course) on how you proceed with the analysis?
2. Advantages/disadvantages of kNN, Bayes, LDA, SVM,...(other methods)?
3. What can we learn from a 'learning curve'? Why would one bother to train with smaller amounts of data and not use the entire training set only?
4. What normalization schemes do you know?
5. You have only few data, but still want to model a classifier to obtain an idea about the classification performance. Let's say you have 3 classes with 3, 5 and 7 samples resp. Which classifier is preferred?
6. You trained c binary (one-versus-all) classifiers and observe that for increasing training data, the performance decreases?
7. How does the kNN, Bayes, LDA (or other) classifier work?
8. What is characteristic for the SVM?
9. How is the performance of a binary classifier analyzed?
10. You have data whose features (dimensions, variables) comes from different sources, e.g. audio and visual. Do you train a single classifier for all features?
11. You have satisfactory results, let's say with the LDA-PCA combination. But now you want to optimize and improve if necessary by another 1-2 percent. What could you try?
12. What does the PCA do? How do you apply it in Matlab?
13. You are given a set of patterns whose features are drawn from a (limited) set of elements. What classifier do you recommend? Some of the patterns have different (vector) length - which classifier could you try now?
14. Your data contain components, that have only zero values, or some values maybe missing and expressed with NaN. How do you proceed?
15. Does normalization improve performance?
16. You intend to datamine (explore) a huge set and are given no labels (class information). How do you begin?
17. Compare hierarchical clustering with k-means clustering.
18. What types of error estimation do you know?
19. You perform density estimation in Matlab with the command `ksdensity`. Does it compare to s-moothening the histogram, meaning applying `histc` followed by `convn` for instance?
20. What does the parameter 'bandwidth' mean in density estimation? How does the result of density estimation with a small bandwidth value compare to the result with a large bandwidth value?

D.2 Answers (as hints)

1. Start with kNN to obtain a ’reference’ performance, that represents a lower bound; then use a linear discriminant analysis; apply the principal component analysis if the data set is ’complicated’.
2. As in script.
3. a) Overfitting: there is the possibility that we obtain better performance for a smaller training set: the learning curve should be increasing, but may also decrease for excessive training data. b) Verification: we gain certainty that we’ve done everything correct.
4. As in script.
5. kNN is the first choice, at it can essentially work with single samples only. You may also try a Naive Bayes classifier. LDA is unlikely to return reliable results.
6. a) Overfitting (see learning curve). b) Class imbalance problem.
7. kNN: storage-based classifier. Each testing sample is compared to all other ...
   Bayes: we use Gaussians to approximate the distributions...
   LDA: a weight matrix $\mathbf{W}$ is generated that separates the classes...
8. a) Focuses on samples that were difficult to classify. b) Uses a kernel function to project data into a higher-dimensional space.
9. With a 4-response table (hit, miss, ...). Ideally the system has parameters with which we can influence the performance and so create an ROC curve (see script for details).
10. One can. But we can also try ensemble classifiers (such as bagging) - it sometimes gives better results.
11. a) SVM. b) Search for optimal number of princ. comp. combined with LDA. c) Feature selection. d) Ensemble classifier.
12. Finds axes of variation for each dimensions and rotates the data such that it is aligned with those axes. Applied in Matlab with the command `princomp` which returns a $d \times d$ matrix from which we select components and then transform the data.
13. Decision tree. If patterns of unequal length: string matching.
14. It can be ignored if we use for instance the PCA and the LDA of Matlab. However, we need to take care of it, when clustering for instance or when building our own classifier. See script for details.
15. Often, but not always, because normalization can also lead to a distortion of the samples’ relations (distances).
16. Clustering. k-means. see script for details.
17. See script for details.
18. Hold-out estimation, cross-fold validation, ...see script.
19. It compares approximately only. `ksdensity` estimation works directly with the data points and generates a smooth output taking individual data values in account. In contrast, if you use the histogram, then you obtain a simpler output, whereby the filtering may achieve a smoother ’look’, but it remains a simpler estimation than using `ksdensity` immediately.
20. Bandwidth signifies the width of the kernel function, e.g. the sigma in the Gaussian function. The result of density estimation with a small bandwidth value looks ’finer’ as compared with a large bandwidth value, for which the result looks ’coarser’ (smoother).