Classification & Clustering

Rein van den Boomgaard

02 May 2011

1 Pattern Recognition

A pattern recognition problem in practice is often of the form that we have a lot of *feature vectors* $\mathbf{x}_i$ for $i = 1, \ldots, N$ each vector representing an object under study.

- Consider the situation that each vector belongs to one of K classes (e.g.\ apples, pears, oranges, etc where the feature vector describes one piece of fruit with its color, weight, sizes, etc). An important task in pattern recognition is to **classify** a feature vector ($\mathbf{x}$) as belonging to class C_k (in our example that would be to decide on the sort of fruit given the color, weight and size).

  Such a classifier in general is hard to specify with a small set of rules based on generic knowledge. Instead we collect a set of *training samples* of feature vector $\mathbf{x}_i$ with known class c_i and we use statistical procedures to come up with a rule to classify a feature vector whose class is unknown.

- Another situation is where we are looking at the functional relation between a set of input features (collected in the feature vector $\mathbf{x}$) and an output value y (could be vectorial as well). Assuming there is a functional relation we could look for the function f such that $y = f(\mathbf{x})$.

  Again in practice the function is often too complex to specify in simple mathematical rules and we have to learn the relation from a lot of examples pairs: $\mathbf{x}_i$ and y_i. Learning such a functional relation is called **regression**. Regression and interpolation are related but not equivalent problems. In interpolation we assume that the examples are noise free, in regression problems that is most often not the case, so $y_i \approx f(\mathbf{x}_i)$.

- A third important pattern recognition problem is that we have a lot of feature vectors $\mathbf{x}_i$ but this time we are looking for relations and dependencies is our dataset. E.g. the big grocery stores keep track of a lot of data of their customers, what they buy, in what quantities, at which time of the year and day, etc etc. For every customer a feature vector full of data.

  Clustering tries to find the feature vectores (customers in our example) that are comparable. For our customer database that would lead to the discovery of the class of single men (beer, sigarettes, no diapers, frozen food), the class of elderly peoply (shopping several times per week, small quantities, cat food, paying with cash money), students (shopping late at night, discounted products, wine, cola, pasta), etc

  Whereas with classification we know the classes a priori, in the case of clustering we try to find out which classes (clusters) are present in our data.

2 Classification

Classification is an important subject in pattern recognition. Given a feature vector $\mathbf{x}$ a classifier assigns a class to it. E.g. given the weight and size of a piece of fruit a classifier assigns a class like 'apple' or 'pear'.

From a mathematical point of view a classifier makes a partitioning of the space $\mathbb{R}^n$ where n is the number of elements in the feature vector. A partitioned space can be characterized by its partitions or by the borders between the different classes. In n dimensional space the borders are $n-1$ dimensional hyper surfaces separating the different classes.

2.1 Types of Classifiers

- First estimate the joint probability density function $p_{\mathbf{X},\mathbf{C}}(\mathbf{x}, \mathbf{c})$. Then, given the pdf, formulate a function that assigns a class to a feature vector by optimizing some criterium.
  - Here we look at a classifier minimizing the probability of misclassification. This leads to the MAP classifier (where MAP stands for Maximum A Posterio Probability). The boundaries separating the classes are often implicitely defined in this classifier.
  - Other classifiers include some subjective loss function and finds a classification function that minimizes the loss (an obvious example for a loss function is to make a distinction between false positives (classifying a person to be ill when in fact he isn't) and false negatives (classifying a person as healthy when in fact he isn't). You can imagine that a false negative can have more serious consequences then a false positive).
  - Instead of letting the pdf functions implicitely define the boundaries between the classes we can also state that we are looking for separating planes and then define a statistical criterium to find an optimal plane (we will look at the Fischer Linear Discriminant).
- Directly train a function to map the input (feature vector) to a class. These are examples of classifiers that do not explicitly consider the statistical nature of the problem. Instead the samples in the training set in a sense are taken to be the distribution.
  - The nearest neighbor classifier is an important example of such a classifier. The resulting boundaries separating the classes in the n dimensional feature space are quite complex. We say the classifier is *overfitting*.
  - A well known class of classifiers are the *neural networks*. A neural network is conceptually very simple in that it builds a network from very simple *logistic regression functions*. These functions function like linear classifiers. The resulting class boundaries can become quite complex. Neural nets tend to overfit the data while learning is a slow process.
  - Another important class of classifiers starts with the shape of the boundaries separating the classes. E.g. assuming in a two class problem we may look for the hyper plane that optimally separates the two classes.
    - We will look at two examples of such *linear classifiers*: the *perceptron* (the basic ingredient for a neural network) and the (linear) *support vector machine* (SVM). SVM's (especially their generalization based on kernels) are nowadays by many considered to be the best 'off-the-shelves' classifier.
    - Not all classification problems can be solved with linear classifiers. In fact in most cases just one linear classifier is incapable of solving the problem. To deal with this we can either
      - explicitly introduce non linear separation boundaries, or
      - build a network or cascade of linear classifiers (e.g. neural networks and decision trees), or
      - transform the data to a (much) higher dimensional representation where we can find linear boundaries solving our classification problems (the 'kernel trick' in SVM's).

2.2 Experimental Setup

- **Learning Set, Test Set**

  Let $(\mathbf{x}_i, c_i)$ for $i = 1, \ldots, N$ be the total set of feature vectors and corresponding class labels.

  The classifier is learned from a subset of the total dataset. Let L be a subset of the integers $1, \ldots, N$ then $(\mathbf{x}_i, c_i)$ for $i \in L$ is the learning set.

  The test set is defined equivalently $(\mathbf{x}_i, c_i)$ for $i \in T$. The classifier learned from data in the learning set is then tested with data from the testset. It is very wrong to test a classifier on the

same data as the data used to train the classifier. E.g. a nearest neighbor classifier assigns to an unknown feature vector the class of the vector that is closest in the learning set. Such a classifier by definition scores a 100% on the learning set. It kind of memorizes all examples. Nevertheless a nearest neighbor classifier will turn out to be a classifier that is quite good in classification even of feature vectors not in the learning set.

- **Confusion Matrix**

  For a parameterless classifier we learn the classifier c from the samples in L and test the classifier with the samples in T.

  A convenient way to assess the performance of the classifier is to make a **confusion matrix**. We make a matrix of $K \times K$ entries where K is the number of possible classes $c = 1, \ldots, K$. Entry i, j in the tabel then equals the number of times that an object of type i is classified as being of type j. Along the diagonal then are the objects that are correctly classified and in the off diagonals the confused objects can be found.

  In a subsequent section we will see the confusion matrix for a nearest neighbor classifier.

- **Cross Validation**

  Sometimes the set of examples with 'ground truth' (i.e. example feature vectors for which the correct class are known) is too small to divide the set into a test and learning set of meaningfull size.

  Then we can use a technique known as *cross validation*. Let E be our set of examples $(\mathbf{x}_i, c_i)$. Let $E_1 \ldots E_k$ be a partition of E. k-fold cross validation then learns a classifier using $k-1$ of the subsets E_i and validates (tests) the classifier using the remaining subset as testset.

  We can repeat this k times. Every time we leave out one subset as testset, learn from the remaining subsets and test with our chosen subset. The total test for our classifier then is the mean result for all these k tests.

  A special case is the *leave one out cross validation* method. Then we select only one example from E, then learn from all other examples and test on just the one selected example. We can repeat this every time taking apart one example for the test set. This is the same as $k - fold$ cross validation where k equals the total number of examples in E. Leave one out cross validation is most often very expensive in terms of having to learn the classifier as many times as there are examples.

2.3 Nearest Neighbor Classification

Let S_L be the learning set with feature vectors $\mathbf{x}_i$ and class c_i for $i = 1, \ldots, N$. The NNb classifier is then defined as:

$$c(\mathbf{x}) = c_k, \qquad k = \arg \min_{i=1,\ldots,N} \|\mathbf{x} - \mathbf{x}_i\|$$

i.e. we decide on the class of the feature vector in the learning set that is closest to the vector $\mathbf{x}$.

2.3.1 The straightforward implementation

The straight forward implementation does an exhaustive search over the entire learning set each time we have to classify an unknown feature vector $\mathbf{x}$. Note that we may consider the square of the distance (so we don't have to take the square root).

Let X be the $d \times N$ matrix of all features vectors and let $\mathbf{c}$ be the vector of the corresponding classes. In Python we can use the following code for NNb classification:

```
from pylab import tile, sum, argmin
class NNb:
   def __init__(self, X, c):
       self.n, self.N = X.shape
```

```
        self.X = X
        self.c = c


    def classify(self, x):
        d = self.X - tile(x.reshape(self.n,1), self.N);
        dsq = sum(d*d,0)
        minindex = argmin(dsq)
        return self.c[minindex]
```

This code looks deceivingly simple as all loops are 'hidden' in the Python/Numpy functions. The use of Python/Numpy introduces some inefficiency as well (finding the `minindex` could of course be integrated into calculating the sum of squares of all elements for all vectors).

2.3.2 Nearest Neighbor Classification in Practice

In this section we try the NNb classifier on the Iris dataset. First we divide our set of examples into a learning set and a training set. For each of the three classes we have 50 examples. We select 30 for the learning set and 20 for the test set. The 20 examples of each Iris (60 in total) are then classified based on the learning set.

```
from pylab import loadtxt, arange, loadtxt, permutation, transpose,\
     zeros, sum, plot, subplot, array, scatter, logical_and, figure,\
     savefig


import sys
sys.path.append("./python")

from nnb import NNb

def cnvt(s):
    tab = {'Iris-setosa':1.0, 'Iris-versicolor':2.0, 'Iris-virginica':3.0}
    if tab.has_key(s):
        return tab[s]
    else:
        return -1.0

XC = loadtxt('data/iris.data', delimiter=',', dtype=float, converters={4: cnvt})

ind = arange(150) # indices into the dataset
ind = permutation(ind) # random permutation
L = ind[0:90] # learning set indices
T = ind[90:] # test set indices

# Learning Set
X = transpose(XC[L,0:4])
nnc = NNb(X, XC[L,-1])

# Classification of Test Set
c = zeros(len(T))
for i in arange(len(T)):
    c[i] = nnc.classify(XC[T[i],0:4])
```

```
# Confusion Matrix
CM = zeros((3,3))
for i in range(3):
    for j in range(3):
        CM[i,j] = sum( logical_and(XC[T,4]==(i+1),c==(j+1)) )

print(CM)

# Plot Test Set
figure(1)
color = array( [[1,0,0],[0,1,0],[0,0,1]] )
for i in range(4):
    for j in range(4):
        subplot(4,4,4*i+j+1)
        if i==j:
            continue
        scatter( XC[T,i], XC[T,j], s=100, marker='s',
                 edgecolor=color[XC[T,4].astype(int)-1],
                 facecolor=[1,1,1]*len(T))

        scatter( XC[T,i], XC[T,j], s=30, marker='+',
                 edgecolor=color[c.astype(int)-1])

savefig('figures/nnbtest.pdf')
```

```
 [[ 20.   0.   0.]
  [  0.  19.   0.]
  [  0.   2.  19.]]
```

2.3.3 k-Nearest Neighbor Classification

An often used extension to nearest neighbor classification is the k-nearest neighbor classifier. Instead of searching for the closest neighbor we search for the k closest neighbors. The class we assign to the unknown vector is the majority class in the k neighbors.

The advantage is that the k nnb classifier tends to make more smooth boundaries between the classes and thus hopefully in a more accurate way generalizes the learning set to the entire population.

The standard nearest neighbor classifier (a 1-NNb classifier) always achieves 100% correct classifications *on the learning set*. Every example in the learning set is thus taken as absolute truth despite the fact that noise or measurement errors could have introduced errors. The k-NNb classifier is capable of dealing with this situation. Consider a feature point of type A in the middle of a cluster of type B. The 1-NNb classifier would always classify feature vectors close to the A-point in the middle of the B-points as being of type A. A k-NNb classifier wouldn't be bothered by that one A point in the middle of B points, only one of k neighbors will be of type A and all points close to the A point will be classified as type B.

The disadvantage is that we have a parameter k that must be set. Often the optimal value is selected by learning as well. Note that the examples to learn this parameter must come from the learning set. It is wrong to test the classifier on the test set for different values of k and select the optimal one and say that that is the performance of your classifier. Then you have tuned (learned) parameters to result in best performance for the test set.

The need for extra samples needed to do this learning can be circumvented using a cross validation approach using only the learning set, while using the test only to test the classifier with optimal parameter.

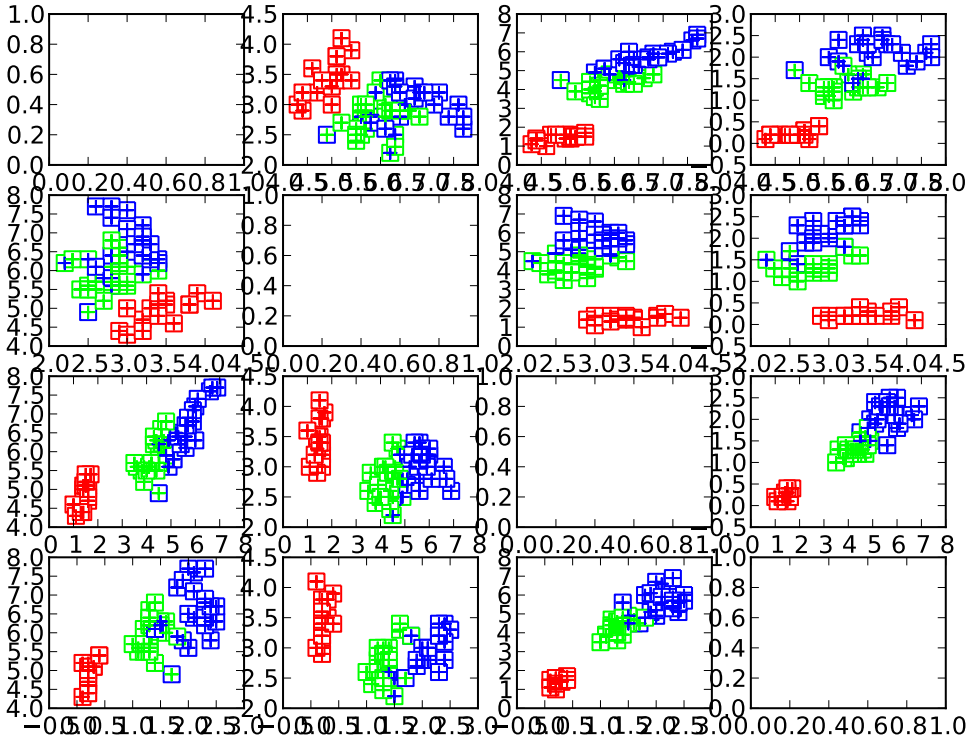

Figure 1: Nearest Neighbor Classification Results

2.3.4 Speeding Up

Speeding up nearest neighbor classification is not a trivial task. The Python code given above is very short and undoubtly very inefficient. Even on this level of abstraction using Python/Numpy it is possible to find ways to speed up nearest neighbor search.

Let X be a $n \times N$ matrix of the N prototype vectors $X = (\mathbf{x}_1 \cdots \mathbf{x}_N)$. Let $\mathbf{y}$ be a n dimensional vector then we need to calculate the distance to each of the prototype vectors $d_i = \|\mathbf{x}_i - \mathbf{y}\|$. Note that calculating the square is enough (that makes the square root not necessary). We have: $d_i^2 = \|\mathbf{x}_i - \mathbf{y}\|^2 = \mathbf{y}^\intercal\mathbf{y} - 2\mathbf{y}^\intercal\mathbf{x}_i + \mathbf{x}_i^\intercal\mathbf{x}_i$. Finding the i such that d_i^2 is minimal is not dependent on the constant $\mathbf{y}^\intercal\mathbf{y}$. Furthermore the contributions $\mathbf{x}_i^\intercal\mathbf{x}_i$ can be precalculated from the learning set.

But still the order of computation is linear in the number of samples in the learning set. The problem is that vectors cannot be sorted in a chain allowing for binary search techniques.

Several n dimensional index structures are proposed to make nearest neighbor search faster. Some are exact, some are approximative technique. A Google(Scholar) search will reveal a lot of the possibilities mentioned in the literature.

2.4 Minimum Error Classification

We now return to the question what makes a good classifier. In this section we consider the question from a statistical point of view. We will look for the classifier that minimizes the probability of making a wrong classification.

2.4.1 A simple two class case

Let's start with a simple 2 class problem and just one scalar feature x. Statistically we thus have two random variables X and C where X is a continuous rv and C is a discrete rv with just 2 possible values (1 or 2).

A classifier function classify assigns either 1 or 2 to a feature value x. Let R_1 be the subset of $\mathbb{R}$ such that classify$(x) = 1$ and let R_1 be the subset such that classify$(x) = 2$. We now search for a classifier such that the error of misclassification is minimal. This error is given by:

$$P(\text{error}) = \int_{R_2} p_{XC}(x, C = 1)dx + \int_{R_1} p_{XC}(x, C = 2)dx$$

Every possible x contributes to either the first or the second integral. Thus we can minimize the contribution of feature value x to the total error by assigning x to R_2 in case $p_{XC}(x, C = 1) < p_{XC}(x, C = 2)$, otherwise we assign it to R_1.

The minimal error classifier for a scalar 2 class problem thus is defined with:

$$\text{classify}(x) = \begin{cases} 1 & : p_{XC}(x, C = 1) \geq p_{XC}(x, C = 2) \\ 2 & : \text{elsewhere} \end{cases}$$

The condition in the above classifier can be rewritten as:

$$p_{X|C}(x|C = 1)P(C = 1) \geq p_{X|C}(x|C = 2)P(C = 2)$$

Classification is thus based on the class conditional probability density functions but we have to multiply with the a priori class probabilities.

The condition in the classifier can also be rewritten as:

$$\frac{p_{X|C}(x|C = 1)P(C = 1)}{p_X(x)} \geq \frac{p_{X|C}(x|C = 2)P(C = 2)}{p_X(x)}$$

where we recognize:

$$P(C = 1|x) \geq P(C = 2|x)$$

Thus the minimum error classifier is equivalent with a Maximum A Priori Classifier. A MAP classifier based on $P(C = i|x)$ has the added advantage that this probability tells us something about how sure we can be that the classification is correct. If $P(C = 1|x)$ is the maximal a posteriori probability (and thus the feature vector x is classified as being of type 1) but drops below some threshold value we could decide to use a classification procedure based on more knowledge or use a more elaborate procedure based on the same feature vector.

2.4.2 A numerical two class example

Again we consider a two class example of a minimum error aka maximum a posteriori classifier based on scalar feature. We assume both class conditional probability density functions are normal distributions:

$$X|C = 1 \sim \text{N}(\mu_1, \sigma_1), \qquad X|C = 2 \sim \text{N}(\mu_2, \sigma_2)$$

with given a priori probabilities for the classes $P(C = 1)$ and $P(C = 2)$.

For $\mu_1 = 4$, $\sigma_1 = 1$, $P(C = 1) = 0.3$, $\mu_2 = 7$, $\sigma_2 = 2$ and $P(C = 2) = 0.7$ the probabilities (and densities) involved are sketched in figure 2. Note that if we calculate the intersections of $p_{XC}(x, C = 1)$ and $p_{XC}(x, C = 1)$ we get *two* solutions one for x slightly larger then 5 and one for x about -2. The intersection for negative x is practically of no interest because for the x values in that range the probabilities for x are negligible small.

2.4.3 Multi Class, Multivalued Case

The extension to multivalued features is conceptually quite simple (but hard to visualize). The partioning of n dimensional space can become quite complex, even for two classes. It all depends on the probability density functions involved.

In case the class conditional pdf's are Normal with equal covariance matrix, it can be shown that the boundaries of the partioning are straight lines. For general covariance matrices we end up with quadratic curves defining the class boundaries.

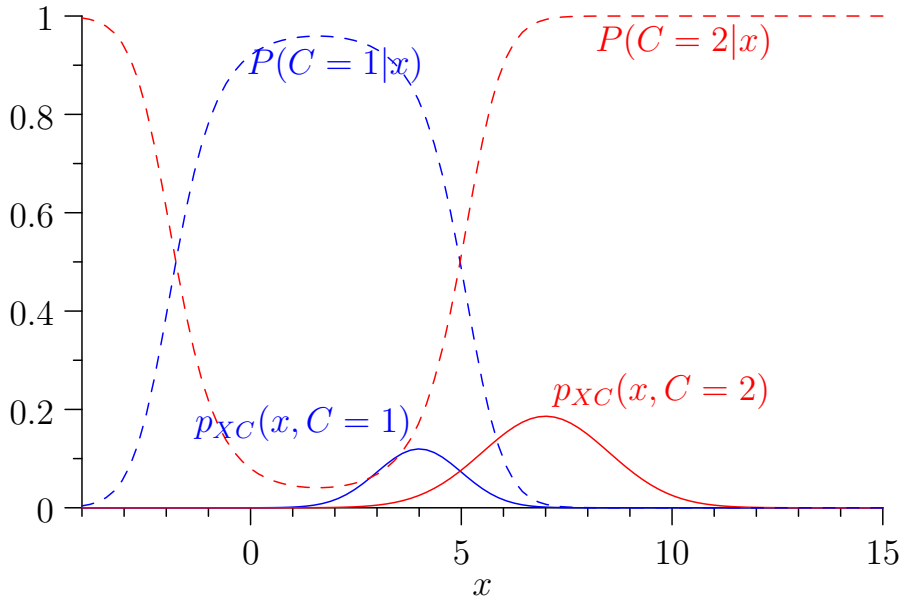

Figure 2: Minimum Error Classification

The extension to the multiclass problem is equally simple. Without proof (you can find that in any decent textbook on pattern recognition) we state that a feature vector $\mathbf{x}$ is classified to belong to class k in case $P(C = k|\mathbf{x}) \geq P(C = l|\mathbf{x})$ for all $l \neq k$. Or equivalently we can define the classify function as:

$$\text{classify}(\mathbf{x}) = \arg\max_k P(C = k|\mathbf{x})$$

2.5 Fisher Linear Discriminant

Let $\mathbf{X}$ and $\mathbf{Y}$ be two multivariate random variables. Our goal is to project the rv's onto a one dimensional subspace such that the resulting scalar rv's can be separated easily. Let $\mathbf{w}$ be the direction vector ($\|\mathbf{w}\| = 1$) that characterizes the one dimensional subspace. The projected scalar rv's are $X_\mathbf{w} = \mathbf{w}^\intercal\mathbf{X}$ and $Y_\mathbf{w} = \mathbf{w}^\intercal\mathbf{Y}$. For these scalar rv's we have $\mu_{X_\mathbf{w}} = \mathbf{w}^\intercal\mu_\mathbf{X}$ and $\mu_{Y_\mathbf{w}} = \mathbf{w}^\intercal\mu_\mathbf{Y}$. The variances are given by $\text{Var}(X_\mathbf{w}) = \text{Cov}(\mathbf{w}^\intercal\mathbf{X}) = \mathbf{w}^\intercal\,\text{Cov}(\mathbf{X})\mathbf{w} = \mathbf{w}^\intercal\Sigma_\mathbf{X}\mathbf{w}$ and $\text{Var}(Y_\mathbf{w}) = \text{Cov}(\mathbf{w}^\intercal\mathbf{Y}) = \mathbf{w}^\intercal\,\text{Cov}(\mathbf{Y})\mathbf{w} = \mathbf{w}^\intercal\Sigma_\mathbf{Y}\mathbf{w}$.

Fisher now defined the optimal $\mathbf{w}$ to be that direction that maximizes the ratio

$$J(\mathbf{w}) = \frac{(\mu_{X_\mathbf{w}} - \mu_{Y_\mathbf{w}})^2}{\text{Var}(X_\mathbf{w}) + \text{Var}(Y_\mathbf{w})}$$

Using the expressions in the previous paragraph this can be rewritten as:

$$J(\mathbf{w}) = \frac{\mathbf{w}^\intercal\left((\mu_\mathbf{X} - \mu_\mathbf{Y})(\mu_\mathbf{X} - \mu_\mathbf{Y})^\intercal\right)\mathbf{w}}{\mathbf{w}^\intercal\left(\Sigma_\mathbf{X} + \Sigma_\mathbf{Y}\right)\mathbf{w}} = \frac{\mathbf{w}^\intercal\Sigma_B\mathbf{w}}{\mathbf{w}^\intercal\Sigma_W\mathbf{w}}$$

where Σ_B is called the *between scatter matrix* and Σ_W is called the *within scatter matrix*.

To maximize this expression we have to look for the $\mathbf{w}$ such that $\partial J(\mathbf{w})/\partial\mathbf{w} = 0$. Note that $J(\mathbf{w})$ is a quotient function and thus:

$$\frac{\partial J(\mathbf{w})}{\partial\mathbf{w}} = \frac{(\mathbf{w}^\intercal\Sigma_W\mathbf{w})\Sigma_B\mathbf{w} - (\mathbf{w}^\intercal\Sigma_B\mathbf{w})\Sigma_W\mathbf{w}}{(\mathbf{w}^\intercal\Sigma_W\mathbf{w})^2}$$

Setting this equal to zero we find:

$$(\mathbf{w}^\intercal\Sigma_W\mathbf{w})\Sigma_B\mathbf{w} = (\mathbf{w}^\intercal\Sigma_B\mathbf{w})\Sigma_W\mathbf{w}$$

Note that $\Sigma_B\mathbf{w} \propto \mu_\mathbf{X} - \mu_\mathbf{Y}$ and thus

$$\mathbf{w} \propto \Sigma_W^{-1}(\mu_\mathbf{X} - \mu_\mathbf{Y})$$

Note that the above expression will not lead to $\|\mathbf{w}\| = 1$ and therefore the use of the $\propto$ (proportional to) symbol. In practice you calculate the right hand side and normalize the vector by dividing all elements by its norm.

The Fisher discriminant optimization defined above thus finds the direction vector and thus the 1D subspace such that the projection onto that subspace gives us a good chance to classify the two classes. Thusfar we haven't used the exact shape of the pdf's (only the means and (co)variances) but to make a classification we could assume that the scalar projected rv's are normally distributed and find a classification rule based on a MAP classifier.

Nowadays there are generalizations of Fishers discriminant to deal with more then two classes. We could also project on a space that is d dimensional where $d < n$ with n the dimension of the original feature space. Then it serves as a projection on a lower dimensional subspace while preserving as much as possible of the separability of the classes.

2.6 Support Vector Machines

The Support Vector Machine is often suggested to be the best 'off-the-shelves' classification method. The math needed to fully understand the SVM is beyond the scope of this lecture series. But we have more then enough math knowledge to understand what the SVM does and in what way it achieves its goals. This understanding will enable us to use an SVM in practice and get the most out of it.

The discussion on SVM's will be broken up in several parts:

1. First we discuss the simplest *linear* support vector machine. This is a classifier that tries so separate 2 classes using a linear hyperplane. We search for the hyperplane with a *maximum margin* (as we will explain later). This linear SVM can only find the maximum margin separating hyperplane in case the dataset is indeed linearly separable.

2. Then we discuss how to change the SVM so we can deal with classes which do not allow perfect linear separation. We do so by allowing some points to fall 'on the wrong side of the hyperplane'.

3. Next we give some simple examples showing that not all datasets of interest allow for a linear separation (not even by being tolerant as described in the previous part). SVM's deal with this problem in particular elegant way. Conceptually the feature vector is embedded in a (much) higher dimensional space and in that higher dimensional space there is better chance to find a separating hyperplane. In practice, due to the so-called *kernel trick*, actual computations are always done in the low dimensional space of the original feature vectors. So conceptually we are doing the (linear) separation in a high(er) dimensional space while all computations are done in the original (lower dimensional) space.

4. Finally we describe the way in which we generalize the 2-class SVM to deal with an arbitrary number of classes.

2.6.1 Linear Support Vector Machines

A linear support vector machine finds a separating hyperplane. So let's first look at (hyper)planes and the (directed) distance from points to the plane. Your intuition in 2D, where a hyperplane is a line, can help out here. A hyperplane $\pi_{\mathbf{w},b}$ is defined with:

$$\mathbf{w}^\intercal \mathbf{x} + b = 0$$

where $\mathbf{w}$ is a vector normal to the plane and the scalar b is proportional to the distance from the plane to the origin. The above equation is true for all points $\mathbf{x}$ on the plane $\pi_{\mathbf{w},b}$.

As an example consider the hyperplane in 2D (a line) defined with $\mathbf{w} = (2\ 1)^\intercal$ and $b = 2$. In components we have:

$$w_1 x_1 + w_2 x_2 + b = 2x_1 + x_2 + 2 = 0$$

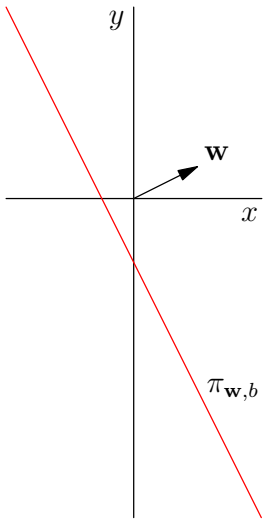

Figure 3: Hyperplane (a line in 2D)

or equivalently (provided $w_2 \neq 0$):

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{b}{w_2} = -2x_1 - 2$$

clearly the equation for a straight line.

Now let $\mathbf{x}$ be an arbitrary point in space and thus $\mathbf{w}^\intercal\mathbf{x} + b$ is not necessarily equal to zero. In fact it is not too difficult to prove that the value of $|\mathbf{w}^\intercal\mathbf{x} + b|$ is proportional to the distance of point $\mathbf{x}$ from the plane. We have:

$$d(\mathbf{x}, \pi_{\mathbf{w},b}) = \frac{|\mathbf{w}^\intercal\mathbf{x} + b|}{\|\mathbf{w}\|}$$

The proof is simple. We start with an arbitrary point $\mathbf{x}_O$ on the plane. The projection of the vector $\mathbf{x} - \mathbf{x}_0$ on $\mathbf{w}$ leads to the (signed) distance:

$$\frac{(\mathbf{x} - \mathbf{x}_0)^\intercal\mathbf{w}}{\|\mathbf{w}\|} = \frac{x^\intercal\mathbf{w} - \mathbf{x}_0^\intercal\mathbf{w}}{\|\mathbf{w}\|} = \frac{x^\intercal\mathbf{w} + b}{\|\mathbf{w}\|}$$

where in the last equality we have used the fact that $\mathbf{x}_0$ is on the plane.

The measure $\gamma(\mathbf{x}) = \mathbf{w}^\intercal\mathbf{x} + b$ is thus proportional to the *signed distance* from the point $\mathbf{x}$ to the plane $\pi_{\mathbf{w},b}$. The sign of $\gamma(\mathbf{x})$ indicates on which side of the plane the point $\mathbf{x}$ is.

Now consider a set of labeled points $\mathbf{x}_i$ with label y_i for $i = 1, \ldots, N$. We assume that the labeling is such that points from the first class are labeled $y_i = +1$ and points from the second class are labeled $y_i = -1$.

The task of a SVM is to find a plane $\pi_{\mathbf{w},b}$ that separates the two classes. We assume for now that such a plane does exists (we then say that the point sets are linearly separable), later we will consider the situation that we allow a few points to be wrongly classified (i.e. are on the wrong side of the plane).

Note that $(\mathbf{w}, b)$ and $(\alpha\mathbf{w}, \alpha b)$ (with α a real scalar) indicate the same plane. With the sign of α we can determine on which side of the plane γ is positive. We assume that the sign of $\mathbf{w}$ is chosen such that $\gamma(\mathbf{x}_i) \geq 1$ for points $\mathbf{x}_i$ for which $y_i = +1$ and $\gamma(\mathbf{x}_i) \leq -1$ for points $\mathbf{x}_i$ for which $y_i = -1$. Saying that the closest points on both sides of the plane have $|\gamma(\mathbf{x})| = 1$ defines the scaling of the normal $\mathbf{w}$ completely. Note that the scaling does depend on the dataset!

For each point $\mathbf{x}_i$ we have $y_i\gamma(\mathbf{x}_i) = y_i(\mathbf{w}^\intercal\mathbf{x}_i + b) \geq 1$. For misclassified points we have $y_i\gamma(\mathbf{x}_i) < 1$. Now define $\pi^+_{\mathbf{w},b}$ as the plane $\mathbf{w}^\intercal\mathbf{x} + b = +1$ and equivalenty $\pi^-_{\mathbf{w},b}$ as the plane $\mathbf{w}^\intercal\mathbf{x} + b = -1$. By definition the margin between planes π^- and π^+ is free of points in the learning set. The width of the margin depends on the direction of the plane.

Let $\mathbf{x}_-$ be an arbitrary point on the π^- plane and let $\mathbf{x}_+$ be a point on the π^+ plane. The vector $\mathbf{x}_+ - \mathbf{x}_-$ connects these points and the distance between the planes (both with norm $\mathbf{w}$) is given by:

$$m = \frac{|(\mathbf{x}_+ - \mathbf{x}_-)^\intercal \mathbf{w}|}{\|\mathbf{w}\|} = \frac{|\mathbf{x}_-^\intercal \mathbf{w} - \mathbf{x}_+^\intercal \mathbf{w}|}{\|\mathbf{w}\|}$$

Because $\mathbf{x}_-$ is on π^-, i.e. $\mathbf{w}^\intercal \mathbf{x}_- + b = -1$ and equivalently $\mathbf{w}^\intercal \mathbf{x}_+ + b = +1$ we have:

$$m = \frac{|(-1-b) - (1-b)|}{\|\mathbf{w}\|} = \frac{|-2|}{\|\mathbf{w}\|} = \frac{2}{\|w\|}$$

Note that the margin separating the two point sets is dependent on the direction $\mathbf{w}$. In our search for a separating plane we would like the margin to be maximal. This leads to the following optimization problem:

$$\max_{\mathbf{w},b} \frac{2}{\|\mathbf{w}\|}$$
$$\text{s.t. } y_i(\mathbf{w}^\intercal \mathbf{x}_i + b) \geq 1,\ i = 1, .., n$$

In words this reads: find me the plane $\pi_{\mathbf{w},b}$ such that the margin $m = 2/\|vw\|$ is maximal, and at the same time all points $\mathbf{x}_i$ for wich $y_i = -1$ are on the correct side of the π^- plane and all the $y_i = +1$ points are on the correct side of the π^+ plane.

The above optimization problem is equivalent with:

$$\min_{\mathbf{w},b} \tfrac{1}{2}\mathbf{w}^\intercal \mathbf{w}$$
$$\text{s.t. } y_i(\mathbf{w}^\intercal \mathbf{x}_i + b) \geq 1,\ i = 1, .., n$$

This is a /quadratic programming problem/ that can be solved using 'off-the-shelves' QP libraries (for Python you might look at `OpenOpt` or `cvxopt`). For a mathematical introduction look at http://www.stanford.edu/~boyd/cvxbook/.

2.6.2 Soft Support Vector Machines

Thusfar we have assumed that the learning set *is* linearly separable. This need not be the case in general of course. Most probable your dataset that you need to classify will not be linearly separable. Due to noise, uncertainty and natural variations some feature points are bound to be on the 'wrong side of the plane'. Fortunately we can extend the method from the previous section to deal with those 'wrong points'.

This is done by considering the following optimization problem:

$$\min_{\mathbf{w},b} \tfrac{1}{2}\mathbf{w}^\intercal \mathbf{w} + C\sum_{i=1}^{n} \xi_i$$
$$\text{s.t.}$$
$$y_i(\mathbf{w}^\intercal \mathbf{x}_i + b) \geq 1 - \xi_i,\ i = 1, \ldots, n$$
$$\xi_i \geq 0,\ i = 1, \ldots, n$$

Observe that in case $\xi_i > 1$ the point $\mathbf{x}_i$ will be on the wrong side of the plane and thus misclassified. In case $0 \leq \xi \leq 1$ the point ξ_i is correctly classified but it will be in the margin. So we allow for misclassified feature points. But we don't like them because we try to minimize the sum of all ξ_i's. The variable C balances the two requirements. The optimization problem formulated here still falls within the category of problems that can be readily solved using the standard QP libraries.

2.6.3 What's in a Name

Why are these machines called *support* vector machines? Look at figure **??**. It is evident that the separating plane and it's margin is determined by just a few samples on the margin planes π^- and π^+ in case we look at the 'hard' maximal margin classifier. But even for the soft margin classifier there are only a few points in the learning set that determine the optimal solution. These feature vectors are called the *support vectors*. These support vectors will play an important role when we will generalize the SVM to a non-linear SVM.

2.6.4 Solving your problem in higher dimensions

Even a permissive linear classifier as introduced in the previous section is not capable of dealing with all classification problems. Consider the problem depicted in figure **??**. No way you can come up with a linear separating plane for this case.

The important idea is to try to represent the original problem in a space of higher dimension where we *are* able to find a separating hyperplane. In figure **??** we show that an embedding of the original (2D) space into a higher dimensional (3D) space indeed allows for a separating hyperplane. The embedding in this case is:

$$\phi((x\ y)^\intercal) = (x\ y\ x^2 + y^2)^\intercal$$

and the separating plane in 3D corresponds with a separating circle in 2D.

Evidently we can still use the QP technique to solve the problem. Instead of working with the vectors $\mathbf{x}_i$ we use the vectors $\phi(\mathbf{x}_i)$. Note that also the vector $\mathbf{w}$ now is a vector in the higher dimensional space.

Although feasible, this approach suffers from the fact that choosing the correct embedding into a higher dimensional space is not an easy task. And furthermore the dimensional of the new space might become quite large making the representation and runtime of the algorithm problematic.

Fortunately there is a way to conceptually solve the classification problem in high dimensions but still represent the vectors in our original lower dimensional space and also do all calculations in that space. This may sound like a miracle but it is what is nowadays known as the 'kernel trick'.

2.6.5 Dual Optimization Problem

The kernel trick is dependent on what is known as the *dual optimization problem*. In a previous section we have given the optimization problem leading to a soft linear classification with maximal margin:

$$\min_{\mathbf{w},b,\xi} \tfrac{1}{2}\mathbf{w}^\intercal\mathbf{w} + C\sum_{i=1}^{n}\xi_i$$

s.t.

$$y_i(\mathbf{w}^\intercal\mathbf{x}_i + b) \geq 1 - \xi_i,\ i = 1,\ldots,n$$
$$\xi_i \geq 0,\ i = 1,\ldots,n$$

Without proof we state that this optimization problem is equivalent with the following optimization problem:

$$\max_{\alpha} W(\alpha) = \sum_{i=1}^{m}\alpha_i - \tfrac{1}{2}\sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \mathbf{x}_i^\intercal \mathbf{x}_j$$

s.t.

$$0 \leq \alpha_i \leq C, \quad i = 1,\ldots,n$$
$$\sum_{i=1}^{n}\alpha_i y_i = 0$$

Solving the first optimization problem for $\mathbf{w}, b, \xi$ (note ξ is a vector) or solving the second problem for α (also a vector with elements α_i) is the same.

The optimal vector $\mathbf{w}$ is given by

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i$$

where the sum is over all vectors in the training set. Fortunately this is not as computationally expensive as it seems because only for the support vectors $\alpha_i \neq 0$. In general the number of support vectors is relatively small.

Calculating the value of b is a bit more complex. Without proof we state that for for the support vectors (i.e. $0 < \alpha_i \leq C$) we have:

$$b = 1 - \mathbf{w}^\intercal \mathbf{x}_i$$

So given one support vector we may calculate b, but it is numerically advisable to take an average over all support vectors.

2.6.6 Kernel Support Vector Machines

The 'kernel trick' is a clever way of lifting a classification problem into a higher dimension—assuming it will be linearly separable in higher dimensions—without doing the actual calculations in the higher dimensional.

The kernel trick mathematically depends on the following observation. For very special transforms ϕ the inner product of $\phi(\mathbf{x})$ and $\phi(\mathbf{y})$ is equal to $K(\mathbf{x}, \mathbf{y})$ where K is called the kernel function:

$$\langle \phi(\mathbf{x}), \phi(\mathbf{y}) \rangle = K(\mathbf{x}, \mathbf{y})$$

In the above we have switched to the notation $\langle \mathbf{a}, \mathbf{b} \rangle$ for the inner product of vectors $\mathbf{a}$ and $\mathbf{b}$ (because strictly the notation $\mathbf{a}^\intercal \mathbf{b}$ is not valid for infinite dimensional vectors).

It should be carefully noted that *not* each function K corresponds with an embedding operator ψ such that $K(\cdot, \cdot) = \langle \phi(\cdot), \phi(\cdot) \rangle$. The mathematical theory on creating kernels with this property is rather involved and we refer to the literature. Here we give a few examples of kernels that are also in use in the `libsvm` library.

- **Linear Kernel**.

  $$K(\mathbf{x}, \mathbf{y}) = \mathbf{x}^\intercal \mathbf{y}$$

  Here the embedding is the identity operator. This trivial example of a Kernel only shows that the kernel formulation of the SVM encompasses the 'original' problem (no embedding in a higher dimensional space) as well. If your problem is simple enough then this might be the most efficient choice (also with regard to the runtime).

- **Polynomial Kernel**.

  $$K(\mathbf{x}, \mathbf{y}) = (\gamma \mathbf{x}^\intercal \mathbf{y} + c)^d$$

  To see the embedding function ϕ we consider the case $\gamma = 1$ and $d = 2$. Then we may write:

  $$\begin{aligned} K(\mathbf{x}, \mathbf{y}) &= (\mathbf{x}^\intercal \mathbf{y} + c)^2 \\ &= \sum_{i,j=1}^{n} (x_i x_j)(y_i y_j) + \sum_{i=1}^{n} (x_i \sqrt{2c})(y_i \sqrt{2c}) + c^2 \end{aligned}$$

For $n = 3$ we can write:

$$\phi(\mathbf{x}) = \begin{pmatrix} x_1x_1 \\ x_1x_2 \\ x_1x_3 \\ x_2x_1 \\ x_2x_2 \\ x_2x_3 \\ x_3x_1 \\ x_3x_2 \\ x_3x_3 \\ x_1\sqrt{2c} \\ x_2\sqrt{2c} \\ x_3\sqrt{2c} \\ c \end{pmatrix}$$

If we would start with a 256 dimensional feature space (not that big really: it is just one spectrum, or the encoding of a small 16×16 image) and take $d = 4$ then we end up with an embedding in a 183181376 dimensional space (and still doing the calculations in 256 dimensional vector space).

- **Radial Basis Function**.

$$K(\mathbf{x}, \mathbf{y}) = \exp(-\gamma \|\mathbf{x} - \mathbf{y}\|^2)$$

For the radial basis function it can be shown that the finite dimensional vector space of the feature vectors is mapped onto an infinite dimensional vector space (a Hilbert space). These radial basis functions turn out to be a very generic choice leading to good classification results.

2.6.7 Multiclass Support Vector Machines

There are several ways to extend the basic two class SVM classification to deal with multiple classes (say K classes). One approach is 'one-versus-rest' where you train K two class classifiers. One for each class 'against' all other classes. Then while classifying it is easy to select a class in case only one of the trained classifiers indicate a succes for a class. In other situations you have to decide which class is more probable (e.g. you might look at how far away the point is from the decision boundaries).

Another approach (used in `libsvm`) is the 'one-versus-one' approach. A SVM classifier is trained for each class i agains another class j. Then when classifying an unknown vector we just count the number of classifiers that are in favour for a class. The class with the most votes wins (in case of equal votes we just pick one).

3 Clustering

Thusfar we have discussed supervised learning methods in (statistical) pattern recognition in the sense that for all features (observation) vectors the class (a semantic label) is known. In many applications such a semantic labeling is not known. In these situations we could be looking for a method to find *clusters* in the data.

Clustering, according to Wikipedia, is the assignment of a set of observations into subsets (called clusters) so that observations in the same cluster are similar in some sense. Clustering is a method of unsupervised learning, and a common technique for statistical data analysis used in many fields, including machine learning, data mining, pattern recognition, image analysis, information retrieval, and bioinformatics.

In this short introduction we introduce only one simple but often used clustering method: k-means clustering.

3.1 k-Means Clustering

k-means clustering is nicely described in the corresponding Wikipedia article and is considered to be part of this handout (in the sense that you have to study it).

Also study the k-means++ variant. This extension to the original k-means clustering algorithm describes a clever way to make initial estimates for the k clusters (but i am not quite sure about its effectiveness).

Clustering based on the 2-means algorithm (assuming two classes) leads to a linear boundary between the two classes. As such it is related to linear classification algorithms. Also note that the feature vectors in the k-means algorithm are compared based on their distance, i.e. an innerproduct. All ingredients for the *kernel trick* that we have seen for the SVM's in a previous section. Indeed you can find several *kernel k-means* algorithms and again we see that a *kernel* 2-*means* algorithm is capable of finding clusters that are not linearly separable.

4 Exercises

4.1 Exercise: Nearest Neighbor Classification

- What is the order of computation for the straightforward implementation of the nearest neighbor classification.

4.2 LabExercise: k-NNb classifier

- Adapt the code for the 1-NNb classifier given in this handout and implement a k-NNb classifier. Try to avoid the loops in Python/Numpy.
- Experiment with different values of k. First of all, with $k = 1$ the results should be the same as with the standard NNb classifier. Also try values $k = 3, 5, 7, 9$ and evaluate the performance of the classifier.

4.3 LabExercise: Minimum Error Classification I

In a previous section enough information is given to make the figure illustrating the minimum error classification. But not all formula's are given there... Write a program to draw the curves that are shown in figure 2. In case you don't feel like programming then at least give all formula's needed to draw the curves (hint: how to calculate the evidence pdf $p_{\mathbf{X}}(\mathbf{x})$?)

4.4 LabExercise: Minimum Error Classification II

- In this exercise you have to implement a MAP classifier (minimum error classifier) for the Iris dataset.
- Assume that the feature vectors are class conditionally distributed according to a 4 dimensional normal distribution (characterized with mean vector μ and covariance matrix Σ. The parameters for these distributions can be estimated from the learning set.
- Test your classifier on the test set.

4.5 LabExercise: Minimum Error Classification III

In this exercise we make use of the results (and program) for the previous exercise.

A MAP classifier is bound to make mistakes. In some cases these mistakes can be quite costly. E.g. classifying a patient with cancer as being healthy can have very negative effect on your patient. In such cases it is wiser to be very carefull with saying a patient is healthy. The a posteriori probability $P(C = \text{healthy}|\mathbf{x})$ can help us here. In case this probabilty is not high enough we can still treat the patient as being not healthy (and then we have to look further to see what is really the problem).

In this exercise you are asked to build a classifier that calculates the a posteriori probability using the Iris dataset. For all misclassified samples in the test set this probability should be reported. Compare this with the a posteriori probability of the samples that were correctly classified. Are those with low probability indeed near the borders of the classes (and therefore likely to be misclassified)?

4.6 LabExercise: Support Vector Machine Classification

In this exercise you have to use a support vector machine classifier to classify the Iris dataset. You are not supposed to implement the SVM classifier yourself, instead you may use a well known implementation known as it's library name in Linux: `libsvm`.

Download the `libsvm` library and compile it (or copy the entire compiled directory from someone else). Be sure to read the "A Practical Guide to Support Vector Classification" (on the homepage of `libsvm`), the README files in the top directory of the libsvm tree (for the parameters of the learning program) and the README file in the python directory (for the pythonic interface to libsvm).

For the more mathematically inclined students i refer to the official reference for the `libsvm` library that can also be found on the home page.

4.7 LabExercise: SVM for Color Assignment

In the dataset for natural spectra there are comments for each of the spectra. In the comment the associated color is mentioned. Your task now is to make a classifier that returns the color given the spectrum.

You can do it directly: feeding the SVM with the high dimensial spectral vector or you could use the PCA basis to represent spectra first in a lower dimensional space (a 3d space seems a logical choice) and then feed these vectors to the SVM.

You should at least learn and test an SVM classifier based on one of these approaches.

The following snippet of Python code can be used to read the data set of natural spectra and also read in the color classes.

```
# read the natural spectra and make it into a set for classification


# the colors found in the dataset
Colors = ['black', 'blue', 'brown', 'gray',
          'green', 'orange', 'pink', 'red',
          'violet', 'white', 'yellow']

# does a line of text contains a color name?
def containsColor( line ):
    for c in Colors:
        if line.find(c)>=0:
            return Colors.index(c), c
    return None, None

# read the file and store spectra in matrix D (rows are the spectra)
# and the classes in vector y

fp = open("../data/natural400_700_5.asc")
lines = fp.readlines()

D = zeros((0,61))
y = array([])
for i in range(0,len(lines),2):
    ind, c = containsColor(lines[i])
    print ind
    if ind is not None:
        d = fromstring(lines[i+1],dtype=int,sep=" ")
        D = append(D,array([d]),axis=0)
        y=append(y,ind)
```

Hints:

- When using the `libsvm` Python functions in `svmutil.py` be aware that these functions require 'normal' Python lists and not `numpy` arrays. In my opinion it is best to use arrays and only convert to lists when using libsvm function. The method `tolist()` can be used to convert to a standard list. Please be aware that in case your array, say `m` is 2 dimensional, the list `m.tolist()` is a list of lists where the 'inner' lists are the *rows* of the matrix `m`.
- Don't expect high accuracy on this task. Around 75% accurate classification is what i could achieve.
- In case your accuracy is way below 75% check whether you have scaled your data. In this application it is necessary to do so (at least for the radial basis kernel).

Remarks:

- I was surprised to see 'brown' as the 'name' for the spectra in the database. I have learned that 'brown' really is darkish orange, or darkish reddish but that the (bright) colors surrounding it leads the human brain to classify it as brown. So when looking at just a single color (a light beam for instance) we will never use the name brown to classify the color. To quote Wikipedia:

  "Brown exists as a color perception only in the presence of a brighter color contrast: yellow, orange, red, or rose objects are still perceived as such if the general illumination level is low, despite reflecting the same amount of red or orange light as a brown object would in normal lighting conditions."

  In this experiment we assign names to spectra, i.e. one color beam in isolation. So we should have a hard time classifying the spectrum as being brown.

  Therefore it should be no surprise that indeed there is some confusion in the classification of brown, it is more of a surprise that so many 'brown' colors are correctly classified without any context.

4.8 LabExercise: k-Means Clustering

- Write a Python/Numpy program implementing k-means clustering.
- Also implement the extension known as k-means++ to make a choice for initial cluster means.
- Test your algorithm on a test set (look for one on the web or make your own data set). A 2D set is easiest for visual inspection of course. The `libsvm` library contains a nice program to use mouse clicks to make 2D data sets with several classes. Look in the directory `svm_toy`.