Intelligent Data Analysis
Lecture Notes on
Classification

Peter Tiňo

The next topic now takes us to the world of *supervised learning*. Such learning occurs in situations where we would like to associate a given input (vector) with a certain value. This value can be e.g. a class label (e.g. cancer, or non-cancer), or it can be a real value (e.g. given past values of a stock index in the form of input vector, what is the predicted value for that stock next day?).

We will now turn our attention to classification. For example, given a medical sample from a patient, we want to decide from previous analysis and observations on other subjects (both patients and healthy controls) whether that patient has cancer or not. This time, each data item will have two components - a data point (input) $\mathbf{x}^i \in \mathbb{R}^d$, and a 'label' $y^i \in \mathcal{Y}$, where $\mathcal{Y} = \{1, 2, \ldots, c\}$ is a label set. Each element of the label set represents a 'class', e.g. *cancer* ($y = 1$), *non-cancer* ($y = 2$). Sometimes, if there are only two classes, one is viewed as a 'positive' class ($y = +1$), the other as a 'negative' class ($y = -1$). In this case $\mathcal{Y} = \{-1, +1\}$. Our data set will now have the form: $\mathcal{D} = \{(\mathbf{x}^1, y^1), (\mathbf{x}^2, y^2), \ldots, (\mathbf{x}^N, y^N)\}$.

1 k-Nearest Neighbour classifier

There are many classification methods for classifying vectorial data. Even though we will study the simplest possible model, it will be sufficient to illustrate some of the core issues associated with performing classification in general.

We will use a simple two-dimensional example to demonstrate how we can classify future data assuming we already have a 'training set' $\mathcal{D}$ to work with. Suppose our data set is divided into two classes: +1 (plotted as blue circles) and -1 (plotted as yellow squares, see Figure 1).

Consider the scenario where we are given a 'test point' $\mathbf{x}$ and we want to decide what is its label $y(\mathbf{x})$. One natural way to do this is to look at the training points from $\mathcal{D}$ that are close neighbours to $\mathbf{x}$ and determine what is the 'majority label' among them. In particular, we may look at the k closest data points from $\mathcal{D}$ to $\mathbf{x}$ to decide its label. This type of classier is called the *k-nearest neighbour* (or k-NN) classifier. Figure 1 helps to illustrate this idea.

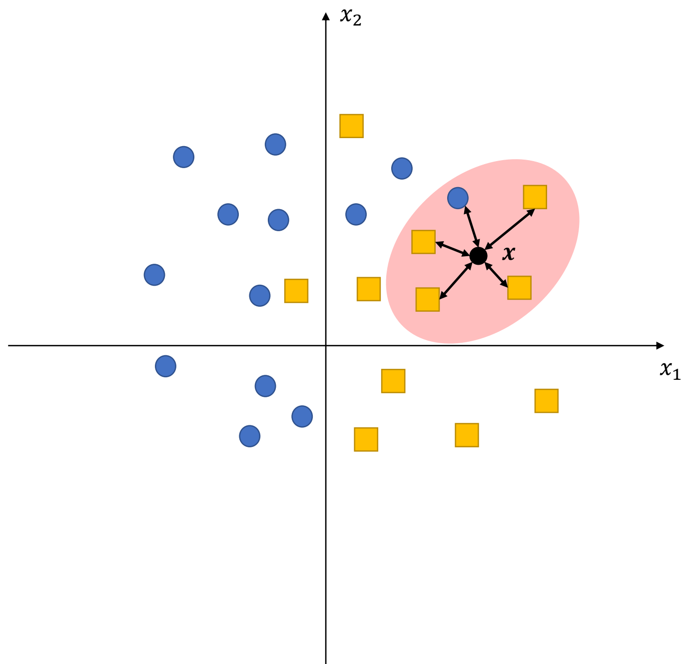

Figure 1: A plot of our data set, divided into two classes. We decide the classification of our new point $\mathbf{x}$ by looking at its k nearest neighbours. In this case $k = 5$.

The role of k, as a model parameter, is to determine the neighbourhood size around the test that should be consulted before making a prediction about the class label. We have two extremes for the value of k:

- $k = 1$ (we look for the closest training point to the test point) and
- $k = |\mathcal{D}| = N$ (extremely large neighbourhood of the test point - the whole training set $\mathcal{D}$). The latter case is a very stupid idea; if our training set has more cases of +1 than -1, then any future test point will immediately have a label of +1 (majority class)!

If $k = 1$, then the decision boundary between the two classes in the input space will be very 'flexible'. This is illustrated in Figure 2. In a sense we feel that this decision boundary reflects the particular sample $\mathcal{D}$, but not the 'nature of the problem. What do we mean by that?

2 Learning beyond the training sample

We can think of our training data $\mathcal{D}$ as being generated from the *'nature'* represented by a joint probability distribution over inputs *and* labels, $p(\mathbf{x}, y)$. We can express the distribution using Bayes rule:

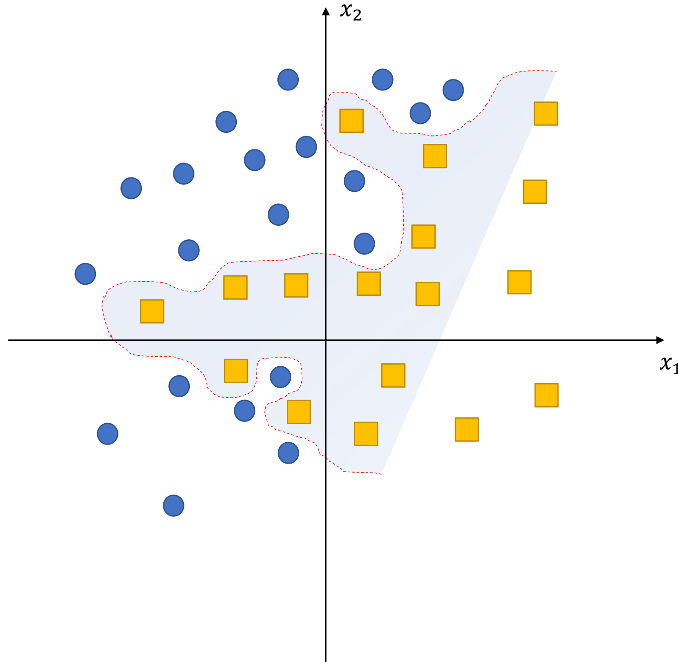

Figure 2: Overfitting the data.

$$p(\mathbf{x}, y) = p(y) \cdot p(\mathbf{x} \mid y). \tag{1}$$

This has a very intuitive interpretation: Each training item $(\mathbf{x}^i, y^i)$ is independently generated by first flipping a coin (drawing from the prior class distribution $p(y)$) and obtaining a class label y^i. Having fixed the class label y^i, we then draw from the class-conditional distribution over the inputs, $p(\mathbf{x} \mid y^i)$, an input $\mathbf{x}^i$ from the class y^i. This is illustrated in Figure 3.

Of course, if we knew the 'nature' $p(\mathbf{x}, y)$, we could find a theoretically optimal decision boundary that will minimize the average miss-classification error of future test inputs. Such a decision boundary is illustrated by the red dashed line in Figure 3. But, in general, we do not know the nature behind the data and only have a sample $\mathcal{D}$ from it. The role of the sample is to inform us about the 'nature' and help us to construct a decision boundary that resembles the optimal boundary as much as possible.

Intuitively, a good classifier should use the sample $\mathcal{D}$ to capture the essence of what is distinguishing the two classes in nature, irrespective of what particular points happened to be generated in $\mathcal{D}$. Let us perform the following mental experiment: Assume that (a-priori) class +1 is more probable than -1, i.e. $p(y = +1) > p(y = -1)$. Repeatedly generate a sample $\mathcal{D}$ of N training points by independently sampling from $p(\mathbf{x}, y)$. For each $\mathcal{D}$, using some value of k, construct the classifier. How much does the classifier change as we keep supplying *different* training sets $\mathcal{D}$ generated from *the same* 'nature' $p(\mathbf{x}, y)$?

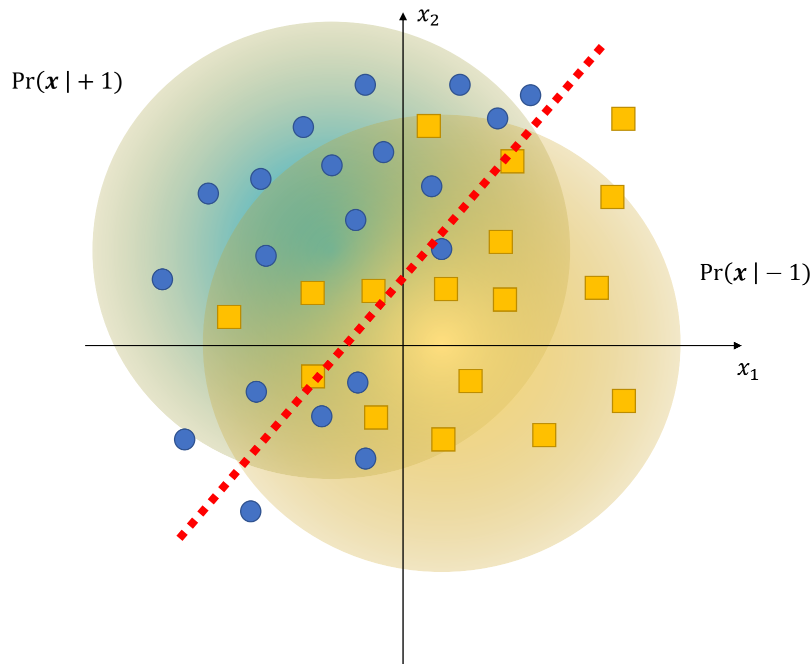

Figure 3: Class-conditional distributions.

We can summarise the two extreme cases as follows:

i) $k = N$: This is a very 'rigid' model. We almost always have the same decision boundary! Whatever the test point is, its predicted class will always be the majority class. This is a very stable (low variance), but pretty bad 'decision boundary' - all minority class points from $\mathcal{D}$ will be miss-classified (high model bias). We say that this classifier has large bias, but low variance.

i) $k = 1$: This corresponds to a very 'flexible' model. As illustrated in Figure 2, every time we get a new sample $\mathcal{D}$ from the same 'nature', we will construct a different decision boundary that closely follows the distribution of points *in that particular sample* (high variance). Of course, this classifier will commit a very small error on training inputs from $\mathcal{D}$ (low bias). The low bias is achieved by following the training sample too closely. We say that we "read too much in the data sample". The widely varying decision boundaries across different samples $\mathcal{D}$ do not reflect the 'nature' very well. In this case we talk about *overfitting the training data.*

We have described a very general issue present when *non-linear models* are involved. How much 'flexibility' (or complexity) should we allow in the model? Too simple a model ($k = N$) will be too rigid and have high bias. Too complex a model ($k = 1$) will be too flexible and will overfit the particular data we have. We must find a way of selecting the right model complexity by finding the right *compromise between the bias and variance.* But we need to do this using a data sample $\mathcal{D}$, because this is the only information about the nature we have.

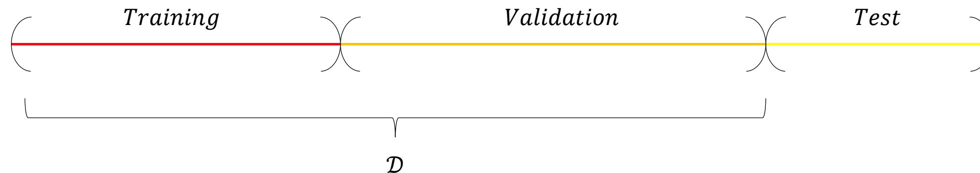

Figure 4: Partition of the data.

3 Training, Validation and test Data

We continue our discussion of classification by looking at how we can use data to build up our k-NN classifier model, including how to decide on the complexity i.e. what is an appropriate value for k.

The data we use are divided into three types:

i) *Training set* - This is used to train or construct our classifier model.

ii) *Validation set* - This set is used to decide on the model complexity. In the case of k-NN classifier, we have already shown that this boils down to selecting an appropriate neighborhood size k. Remember that we cannot simply use the training set for this task, as the optimal value of k would obviously be 1! In other words, *the decision about the desired model complexity cannot be made on the data sample used to construct/train the model itself*. We must have an 'objective' view enabled by a different, independent data set.

iii) *Test set* - This is data we do not touch until we built the full model successfully (i.e. including setting the right model complexity!). This set is useful in estimating the inherent error the model has, or in other words, to estimate how well the model generalizes the training data to future unseen inputs.

It is standard practice to split our given data into the training, validation and test sets. Obviously, we would like all three sets to be as large as possible to guarantee stable model construction and error estimation.

In the following we will assume that our dataset is rather small, but we have enough data for at least a reasonable split into two parts. As the test part needs to be untouched in the model construction phase, we will need to use the other part for *both* training and validation purposes. As we do not have access to the test data during the model building, we will refer to this joint training/validation part as our data $\mathcal{D}$.

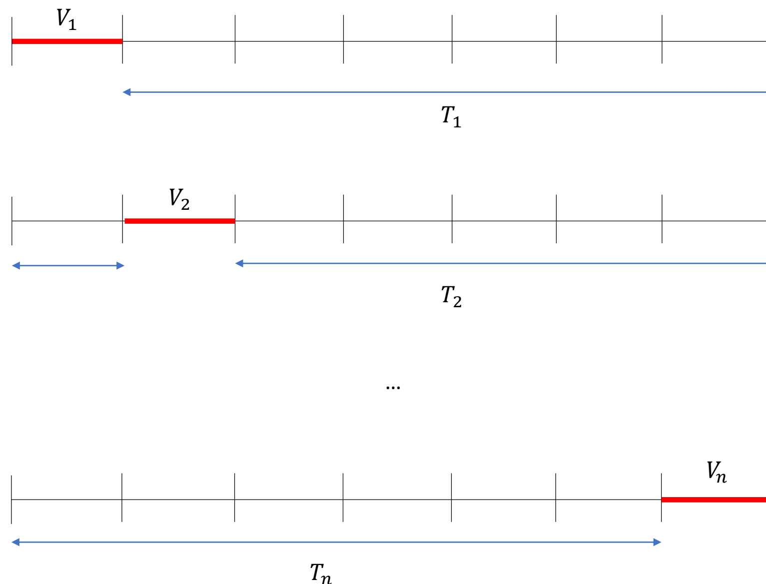

Figure 5: Schematic illustration of data partition in cross-validation.

4 Cross-Validation

As explained above, consider now a situation where $|\mathcal{D}|$ is not large enough for a single split into a training and validation sets. We need a large training set to avoid building a bad model, but this risks the validation set being very small. We need to work with $\mathcal{D}$ in a clever way to build and tune up our model. Given our data set $\mathcal{D}$, we divide it up into n disjoint data sets called *folds* (we can take n to be e.g. 10). For each fold $i = 1, 2, ..., n$, we declare it a validation set V_i, leaving the rest of the data $\mathcal{D} \backslash V_i = T_i$ to be our training set. Training a model on T_i yields a validation error $\mathcal{E}_i$ on V_i. The following figure illustrates how we use this technique, called *n-fold cross validation.*

4.1 Error Analysis

An advantage to this method is that every point in our data is eventually used for both training and validation, producing validation errors $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$. We will take an estimation of our overall validation error to be

$$\bar{\mathcal{E}} = \frac{1}{n} \sum_{i=1}^{n} \mathcal{E}_i.$$

Note that the individual validation sets V_i are rather small at the expense of larger training sets T_i. This will cause stable model construction on T_i, but unstable validation error estimation on V_i. Fortunately, by taking the mean $\bar{\mathcal{E}}$ of $\mathcal{E}_i$, we reduce their variability, while

preserving their mean. What do we mean by this?

Let X be a random variable representing validation error on validation folds. We can produce n equally distributed 'copies' of X, $X_1, X_2, \dots, X_n$, one for each fold. An analogy to this is that rolling a die n times produces n random variables on the face the die lands on. Using these copies, we define the random variable Y to be the value of the error estimator of our model i.e.

$$Y = \frac{1}{n}\sum_{i=1}^{n} X_i.$$

Notice that as each X_i are copies of X, $\mathbb{E}[X_i] = \mathbb{E}[X]$ and $Var[X_i] = Var[X]$, for every $i \in \{1, 2, \dots, n\}$. Provided we draw from each X_i independently, we have:

$$\mathbb{E}[Y] = \mathbb{E}[\left[\frac{1}{n}\sum_{i=1}^{n} X_i\right] = \frac{1}{n}\sum_{i=1}^{n} \mathbb{E}[X_i] = \mathbb{E}[X],$$

$$Var[Y] = Var\left[\frac{1}{n}\sum_{i=1}^{n} X_i\right] = \frac{1}{n^2}\sum_{i=1}^{n} Var[X_i] = \frac{Var[X]}{n}.$$

Hence, by averaging over our error estimators, we do not change the expected value of $\bar{\mathcal{E}}$, but the (square) fluctuations in our estimator are n times smaller.

However, there is a problem with the argument above - X_i are *not* independent. In our case, even though for each fold the validation set V_i is disjoint from the training set T_i, the error estimates $\mathcal{E}_i$, $i = 1, 2, ..., n$, *are dependent*, since the training sets T_i and T_j used to build the models validated on V_i and V_j overlap: $V_i \cap V_j = \emptyset$, but $T_i \cap T_j \neq \emptyset$! Therefore, the *cross-validated errors usually tend to underestimate the true generalization error.*

To argue this, note that in general

$$\begin{aligned} Var[Y] &= Var\left[\frac{1}{n}\sum_{i=1}^{n} X_i\right] \\ &= \frac{1}{n^2}\left[Var[X_i] + \sum_{i,j=1, i\neq j}^{n} Cov[X_i, X_j]\right] \\ &= \frac{Var[X]}{n} + \frac{n(n-1)}{n^2}\gamma \\ &= \frac{Var[X]}{n} + \frac{n-1}{n}\gamma, \end{aligned}$$

where, since the covariances $Cov[X_i, X_j]$ for all $i \neq j$ are the same, we denote the common covariance value by γ. The problem is that $\gamma \neq 0$, in fact $\gamma > 0$ because of the large overlapping portion of the training data for the i-th and j-th folds. It is thus obvious that

by considering only $Var[X]/n$ we underestimate the true uncertainty (fluctuations) $Var[Y]$ of the cross-validation error estimate $\bar{\mathcal{E}}$. Note that for larger n, $(n-1)/n \approx 1$, and so

$$Var[Y] \approx \frac{Var[X]}{n} + \gamma,$$

meaning that if we have too many cross-validation folds n, even though we reduce the variance of individual fold error estimates by a factor of $1/n$, the fold estimates themselves will be more unstable (larger $Var[X]$) and their covariance γ will be larger, since larger training parts will be shared between any two folds. Setting n is a trade-off that needs to be carefully balanced. In practice $n = 5$ or $n = 10$ are the most used choices.

5 Metric learning

In many machine learning and data analysis systems the notion a 'distance' or 'similarity' between data items is crucial. Somewhere in the model formulation there must be something that in some way looks at the (training) data given to us and 'compares' each new previously unseen test input with that data to somehow produce the corresponding output. The strategy is that the output associated with the new test input should in some way follow the known outputs associated with previously seen data that are 'similar' or 'close' to the test input. This is made plainly clear in k-NN classification, where it is commonly automatically assumed that the distance is the usual Euclidean metric. However, such notion of *continuity* - 'similar' inputs will often lead to 'similar' outputs (whatever the notion of 'similarity' used) - is a very general principle behind many machine learning algorithms. It is either expressed somehow implicitly in the model formulation, or quite explicitly, as in the case of k-NN.

Consider a data set $\mathcal{D} = \{(\mathbf{x}^1, y^1), \ldots, (\mathbf{x}^N, y^N)\}$, where each $\mathbf{x}^i \in \mathbb{R}^2$ is a data point and $y^i \in \{+1, -1\}$ is a binary class label. Plotting the data using class markers results in:

It appears that certain directions in $\mathbb{R}^2$ carry more 'weight' or impact on the classification of the data, meaning that we need a new model for measuring 'distance'. For example, when inside the classes, moving in the direction of red line makes no difference in the class label. In other words, *from the standpoint of this particular classification problem*, moving in the red line direction causes very small change in the distance, since nothing much changes in the label structure. In contrast, moving in the direction perpendicular to the red line leads to radical change in the class labelling, so even small steps in that direction would be viewed as moving a large distance. When applying k-NN it would be better to use this notion of a distance.

5.1 "Weights" and Metric Tensors

For convenience, we will work in the two-dimensional space $\mathbb{R}^2$, but generalization to higher dimensions is obvious. For points $\mathbf{x} = [x_1, x_2]^T$ and $\mathbf{y} = [y_1, y_2]^T$, square of the usual Euclidean distance between them, $d^2(\mathbf{x}, \mathbf{y})$, is:

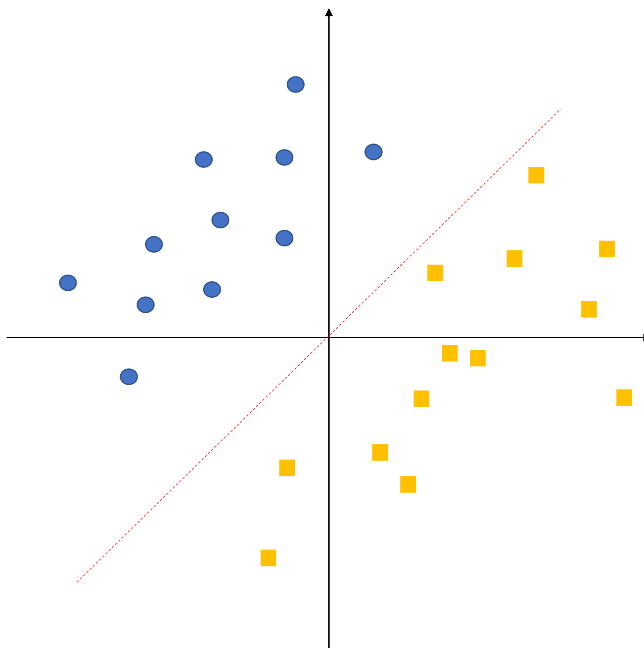

Figure 6: Schematic illustration of the need of Non-Euclidean distance in classification.

$$
\begin{aligned}
d^2(\mathbf{x}, \mathbf{y}) &= \|\mathbf{x} - \mathbf{y}\|^2 \\
&= (x_1 - y_1)^2 + (x_2 - y_2)^2 \\
&= (\mathbf{x} - \mathbf{y})^T \, (\mathbf{x} - \mathbf{y}) \\
&= (\mathbf{x} - \mathbf{y})^T \, \mathbf{I} \, (\mathbf{x} - \mathbf{y}),
\end{aligned}
$$

where $\mathbf{I}$ is the identity matrix.

Let us suppose that the vertical direction contributes more to the distance (is "more important") than the horizontal component. An intuitive idea to express this is adding 'weights' $a_1, a_2 \in \mathbb{R}_{\geq 0}$, where $a_1 < a_2$:

$$d'^2(\mathbf{x}, \mathbf{y}) = a_1(x_1 - y_1)^2 + a_2(x_2 - y_2)^2$$

To write this in a matrix form, if we let $\mathbf{A} = \begin{bmatrix} a_1 & 0 \\ 0 & a_2 \end{bmatrix}$ be a diagonal matrix, then

$$d'^2(\mathbf{x}, \mathbf{y}) = d_{\mathbf{A}}(\mathbf{x}, \mathbf{y})^2 = (\mathbf{x} - \mathbf{y})^T \, \mathbf{A} \, (\mathbf{x} - \mathbf{y}).$$

Our weights indicate that in the horizontal direction, things do not change much i.e. $(x_1 - y_1)^2$ may be large, $d(\mathbf{x}, \mathbf{y})$ is still relatively small. However, $(x_2 - y_2)^2$ creates a significant impact. One way to think about this is the hill analogy; in one direction, we slowly

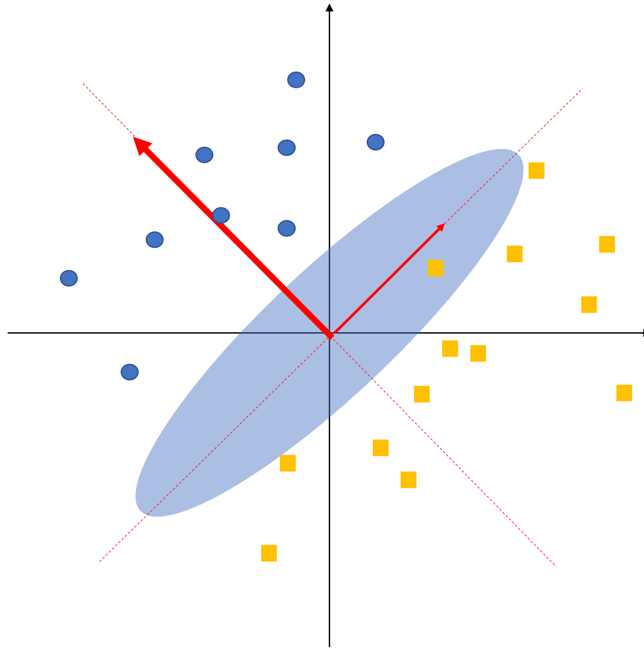

Figure 7: A 'rotation' of the original axes. The thickness of the red arrows indicate the significance of each direction when calculating the distance, each quantified by weights. A neighborhood of a point would then be an ellipse!

traverse down the hill, but in the other direction the hill gets very steep.

Recall now how in PCA we had a symmetric positive semi-definite[1] covariance matrix and by appropriate rotation of co-ordinate axis we were able to express it purely as a diagonal matrix with non-negative terms along the diagonal. In other words, given a squared, symmetric and positive-definite matrix which is non-diagonal, we can always change the axes to get a diagonal matrix. We can apply the same arguments here; if our $\mathbf{A}$ was non-diagonal, we can turn it into a diagonal matrix by rotating the axes in which our data is expressed.

Figure 7 gives an example of a lot can change in the distance if we move in the direction of one diagonal (thick red arrow) and small change in the direction of the other (normal red arrow). The landscape metaphor is illustrated by the superimposed eclipse. The square, symmetric and positive semi-definite matrix $\mathbf{A}$ we have been using is called a *metric tensor*. This matrix governs our notion of distance between any two points, denoted as $d_{\mathbf{A}}$.

More precisely, given a metric tensor $\mathbf{A}$, we can understand what kind of distance it represents by diagonalizing $\mathbf{A}$ through its eigenvectors $\mathbf{v}_i$ (stored as columns of matrix $\mathbf{V}$)

[1] A positive-definite $d \times d$ matrix $\mathbf{A}$ just means that $\mathbf{x}^T\mathbf{A}\mathbf{x}$ is non-negative for every non-zero vector $\mathbf{x}$ of d real numbers.

and the associated eigenvalues λ_i (stored as diagonal elements of a diagonal matrix $\mathbf{S}$):

$$\mathbf{A} = \mathbf{V}\mathbf{S}\mathbf{V}^T.$$

The metric $d_{\mathbf{A}}(\mathbf{x}, \mathbf{y})$ represented by the metric tensor $\mathbf{A}$ enhances and suppresses differences between $\mathbf{x}, \mathbf{y}$ in the direction of eigenvectors $\mathbf{v}_i$ with large and small eigenvalues λ_i, respectively.

5.2 Metrics and Classification

Going back to classifying data, it may be that certain features are more important than others. It may also be that certain directions of input features (combinations of input features) are more important for classification than others. Those directions can be expressed by unit direction vectors $\mathbf{v}_i$. In our example above, the metric tensor in the original co-ordinates can be expressed as diagonal metric tensor in the new co-ordinates indicated by $\mathbf{v}_1$ and $\mathbf{v}_2$:

$$\mathbf{A} = [\mathbf{v}_1, \mathbf{v}_2] \cdot \begin{bmatrix} a_1 & 0 \\ 0 & a_2 \end{bmatrix} \cdot [\mathbf{v}_1, \mathbf{v}_2]^T.$$

$a_1, a_2 \geq 0$ are the weights for $\mathbf{v}_1$ and $\mathbf{v}_2$, respectively, indicating their significance in our new metric.

There are several techniques to automatically learn the appropriate (global) metric tensor $\mathbf{A}$ from the data, together with the classifier. Such techniques fall within the general category of *metric learning*. One can also allow the metric tensor to change smoothly over the input space (thus creating Riemannian metric structure), but such extensions are well beyond the scope of this course.