MSA220 Statistical Learning for Big Data

Lecture 3

Rebecka Jörnsten

Mathematical Sciences
University of Gothenburg and Chalmers University of Technology

Prediction models

- y is the output, an $n \times 1$ vector
- X is the input (explanatories, independent variables....), an $n \times p$ matrix
- We want to build a predictive model or rule for y using X

What is a good model?

- Need to define a loss function to define "good"
- Usually squared error loss if y is continuous
- Usually 0-1 error loss if y is categorical

Predictive models

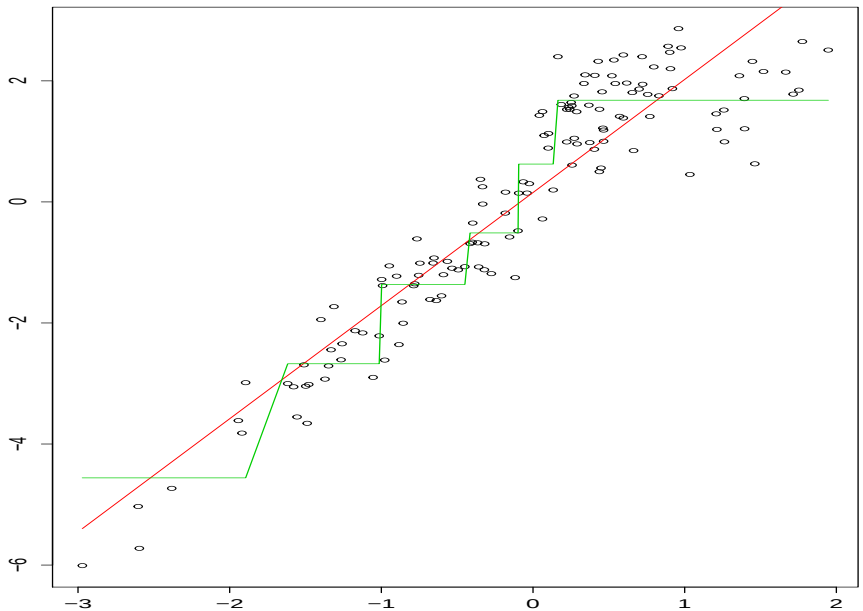
- $E(y - f(X))^2 = E_X[E_{y|X}[(y - f(X))^2 \mid X]]$
- We see that it is enough to minimize the above *pointwise* (for each $X = x$) so we focus on the inner expression
- $f(x) = \arg\min E_{y|X}[(y - f(x))^2 \mid X = x]$
- The minimizer is the *conditional mean* $\widehat{f(x)} = E(y \mid X = x)$
- For 0-1 loss the minimizer is the maximum probability class $\arg\max_c P(y = c \mid X = x)$.

The above conditional mean is called the *regression function*. It is the model that minimizes squared error loss.

How do we estimate $E(y \mid X = x)$?

- An intuitive approach is the approximate the conditional mean by the *local mean* defined by a neighborhood of observations $N(x)$ (e.g. k-nearest neighbors in X to observation x).
- Alternatively, we can parameterize the conditional mean via e.g. a linear model $E(y \mid X = x) = x'\beta$ or some other parametric form.

Predictive models



Red model: linear model. Green: local mean.

Bias-Variance Trade-off

How we choose to estimate the conditional mean determines how flexible/local or rigid/global we are in our modeling. Most methods allow us to choose from a spectrum of more or less local rules. We have already discussed that one needs to select tuning parameters for classification rules, e.g. the number of neighbors for kNN.

Depending on the tuning parameter value, a classification rule can be thought of as *local* or *global*.

local	global
use subset of data	use all data
flexible	more rigid
allow for complex boundaries or models	assume an underlying model for the data distribution
example: kNN with small k	example: discriminant analysis (multivariate normal data distribution)
example: Local average regression	example: linear regression model
need a lot of data to train on	requires less data in general

Discriminant Analysis

Discriminant analysis

CART has "problems" when the class boundaries are linear function in x-space. Such boundaries are poorly approximated by horizontal and vertical cuts. If your tree contains a lot of repeated splits on the same set of features (x1,x2,x1,x2,x1,x2) you can suspect that a linear boundary may be better. What CART is doing in this case is trying to approximate a linear boundary with a large number of small steps.

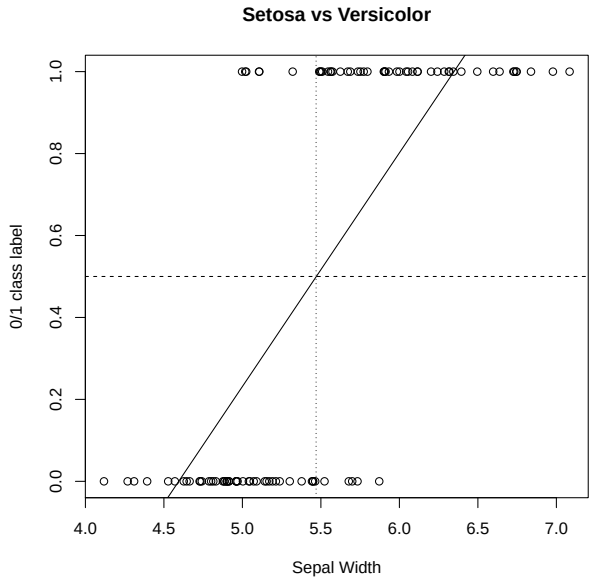
Discriminant analysis is a method that produces linear (and more complex) boundaries in x-space.

The underlying assumption that drives discriminant analysis methods is that the data distribution is *multivariate normal*.

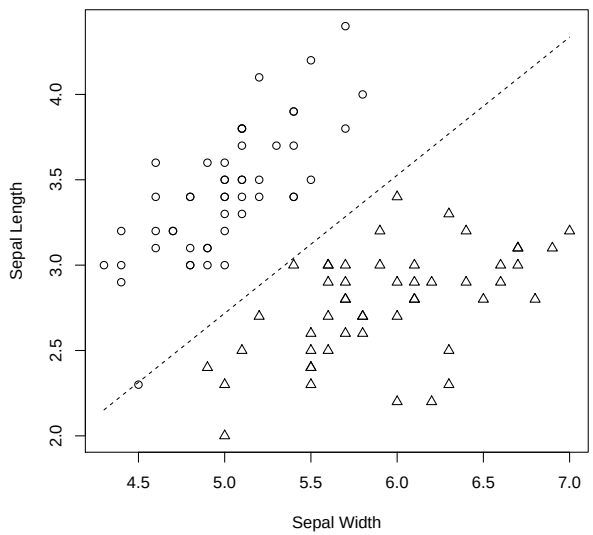
Before we dive into discriminant analysis we will look at a very simple rule based on regression.
If the data contains two classes, code these as a numerical variable y with values 0 and 1.
Let's consider the case with one x-variable. If you plot y versus x you can, hopefully, see a class separation in this scatter plot. Run regression of y on x and plot the fitted regression line. You can interpret this regression fit as an approximate estimate of the probability that the y equals 1 (the regression line is a linear model for $E(y \mid x)$ (conditional expectation of y given x) which is equal to $p(y = 1 \mid x)$ when y takes on only values 0 and 1.

0-1 REGRESSION

Using the maximum posterior probability as your rule, the decision boundary equals the value for x when the regression line crosses $y = .5$: $\{x : \hat{y} = x\hat{\beta} = .5\}$ (here I use the notation for the fitted regression line, and the estimated regression coefficient $\hat{\beta}$).



If you have more than one x-variable, you simply fit a multivariate regression model to the 0-1 data. The decision boundary can now be written as $\{x : x\hat{\beta} = .5\}$, where x and $\hat{\beta}$ are p-dimensional. This boundary expression is a linear equation system in x and can be solved for x-values such that the fitted value on the regression line equals 0.5.
Below is an example with 2 variables.

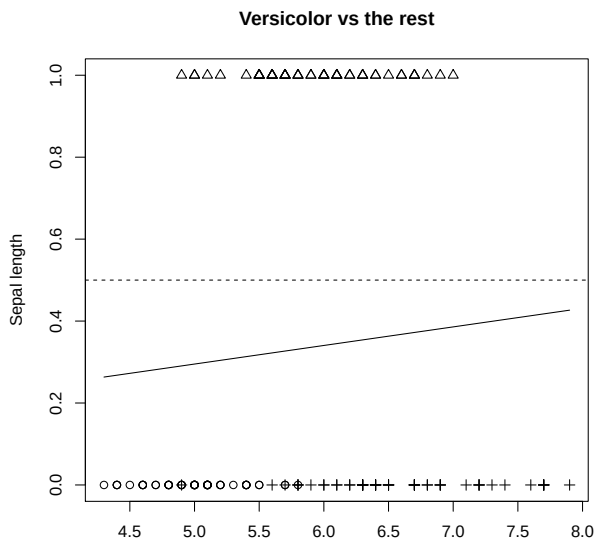


So you see that naive 0-1 regression does a good job at constructing linear decision boundaries.
Any problems?

- The 0-1 regression line can produce negative values and values exceeding 1 which means it's a strange estimate for a probability. *Logistic regression* addresses this issue and the fact that one may want to weigh observations differently depending on how close to the boundary they are. See the Linear Models class for more info on Logistic regression

Problems?

- Masking. 0-1 regression may not work if you have more than 2 classes in your data. The strategy here is to perform many 0-1 regressions where each class takes a turn to be the 1-class and all the other observations are labeled 0. This may, however, result in the 1-class being hidden inside a cloud of 0-s and the regression line won't produce a sensible boundary. See figure below.
  This problem can sometimes be alleviated by running a polynomial regression model instead of a linear model.



Nearest centroid classifier

Another simple rule that is related to discriminant analysis is the nearest centroid classifier.
We compute

$$\hat{\mu}_c = \frac{1}{N_c} \sum_{y_i=c} x_i$$

where N_c is the number of observations in class c. That is, we compute the mean, or centroid, of each class.
The rule is

$$\hat{c}(x) = \arg\min_c d(x, \hat{\mu}_c)$$

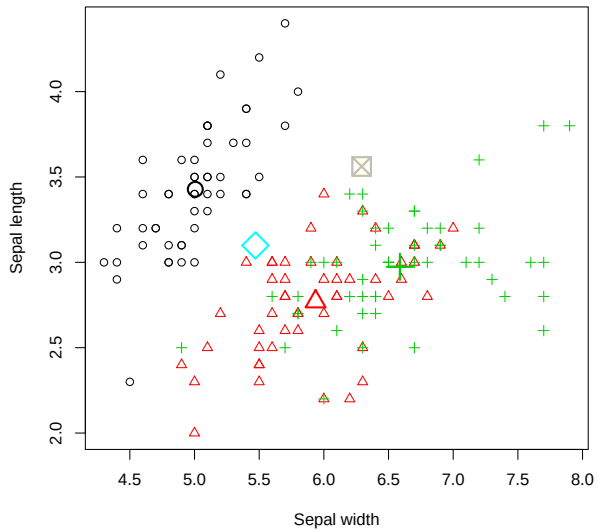
where $d(.,.)$ is the distance between observation location x and the centroid μ. This is usually the euclidean distance

$$d(x, \hat{\mu}_c) = || \, x - \hat{\mu}_c \, ||^2 = (x - \hat{\mu}_c)'(x - \hat{\mu}_c).$$

The rule is thus to allocate each observation to the class with the closest centroid.

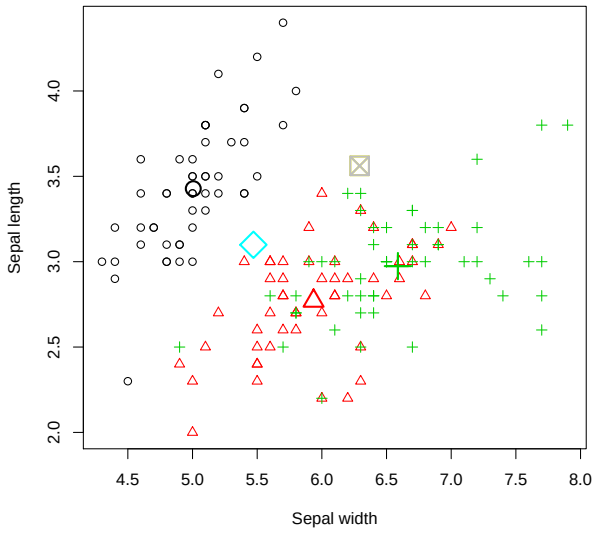
Nearest centroid classifier

Problems?
Nearest centroids ignores the variability of a class around its center and that this variability may be different for different classes and for different features.



For the iris data above, the black circle data (Setosa) exhibits much less variation than the other two classes. In addition, the black and red data sets are very tightly correlated in the two x-features, the green data less so.

Nearest centroid classifier



Consider the turqoise diamond. It is exactly halfway between the black and red class, but since the red class is much more spread out it seems more likely that this new observation belongs to the red class.

Consider the gray box. This is in fact closer to the green class center than the red class center. However, judging by the co-variation of the two x-features, the red class extends more in the direction of the gray box than the green class does so it is more likely this observation belongs to the red class.

From the above examples it is clear that one needs to consider both *spread/scale* of a distribution (the amount of spread around a centroid) and the *shape* of the distribution (the correlation structure between the features) to form a good classification rule. This is what discriminant analysis adds to the table.

General setup is the following;

- prior $\pi_c = p(y = c)$
- data distribution $p(x \mid y = c) \sim N(\mu_c, \Sigma_c)$ where μ_c is a p-dimensional vector and Σ_c is a p-by-p dimensional covariance matrix.

The multivariate normal assumption leads to the following simple, intuitive parameter estimates:

- $\hat{\pi}_c = N_c/N$, where $N_c = \sum_i 1\{y_i = c\}$ is the number of observations in class c.
- $\hat{\mu}_c = \frac{1}{N_c} \sum_{y_i=c} x_i$
- $\hat{\Sigma}_c = \sum_{y_i=c} (x_i - \hat{\mu}_c)(x_i - \hat{\mu}_c)'/(N_c - 1)$

This is quite a large number of parameters...: $(C-1)$ for $\hat{\pi}$ (not C since the πs add to 1), $p \times C$ mean parameters, and $p(p+1) \times C/2$ covariance parameters (since they're symmetric). As the dimensionality of the problem grows (p) the number of parameters grows quickly, especially due to the covariance matrices.

The solution to this problem is to try to simplify the modeling assumption somewhat: $\Sigma_c = \Sigma$, same correlation structure between the features for all classes.

- Realistic? Think about the heart disease data. Do you think ldl-level and bmi are correlated the same way for healthy patients and patients with heart disease?
- The assumption may not be realistic BUT in statistics you always have to worry about the flexible methods suffering from poor estimation and thus leading to a bad classifier. Here, the approximation of equal correlation may be "safer" than trying to build a very complex method with many parameters on noisy data or a data with a small sample size.
- Under this assumption you get $\hat{\Sigma}$ from a *pooled* estimate.

$$\hat{\Sigma} = \sum_{c=1}^{C} \sum_{y_i=c} (x_i - \hat{\mu}_c)(x_i - \hat{\mu}_c)'/(N - C) = \sum_{c=1}^{C} \hat{\Sigma}_c \frac{N_c - 1}{N - C},$$

a weighted average of covariance estimates from each individual class.

Other variants of discriminant analysis

You can make even more simplifying assumptions:

- $\Sigma_c = \Lambda_c$, diagonal matrix. You ignore the correlations between features. (DQDA) (Naive Bayes)
- $\Sigma_c = \Sigma = \Lambda$, diagonal matrix. You ignore correlations AND make the feature variance the same for all classes. (DLDA)
- $\Sigma_c = \Sigma = \sigma^2 I$, nearest centroid. Here you ignore all differences between classes and features in terms of variance and ignore feature correlations.

Building the classifier

We define the boundary between two classes, l and c, at the x-locations where the posterior probabilities are equal:

$$\{x : \pi_l p(x \mid y = l) = \pi_c p(x \mid y = c)\}$$

Equivalently, we can write this on a log-scale as

$$\{x : \log \frac{p(x \mid y = c)}{p(x \mid y = l)} + \log \frac{\pi_c}{\pi_l} = 0\}$$

Let's plug in the multivariate normal data distribution into this expression.

$$p(x \mid y = c) = \frac{1}{(2\pi)^{(}p/2) \mid \Sigma_c \mid} exp(-\frac{1}{2}(x - \mu_c)'\Sigma_c^{-1}(x - \mu_c))$$

Taking logs

$$\log p(x \mid y = c) = -\frac{1}{2}(x-\mu_c)'\Sigma_c^{-1}(x-\mu_c) - \frac{1}{2}\log \mid \Sigma_c \mid -\frac{p}{2}\log 2\pi$$

and taking the log-ratio of the data-distributions we get

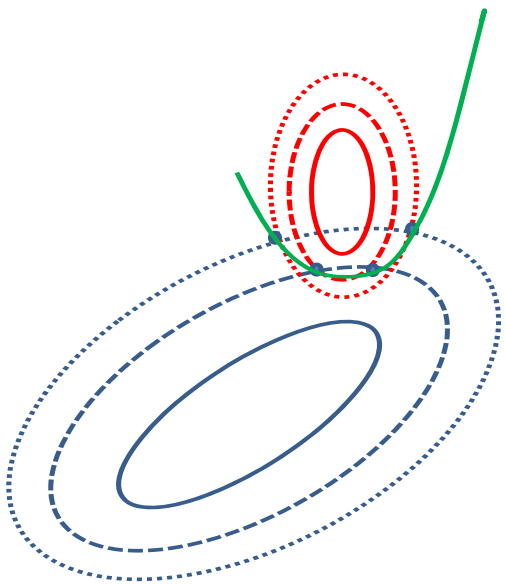
$$\log \frac{p(x \mid y = c)}{p(x \mid y = l)} = -\frac{1}{2}(x - \mu_c)'\Sigma_c(x - \mu_c) +$$

$$-\frac{1}{2}\log \mid \Sigma_c \mid +\frac{1}{2}(x - \mu_l)'\Sigma_c(x - \mu_l) + \frac{1}{2}\log \mid \Sigma_l \mid$$

Notice that this is *quadratic* in x, so class boundaries are quadratic curves in x-space.

Building the classifier

To draw these boundaries, simply look for points in x-space where the posterior distributions have the same value in two classes. I have illustrated that below with two classes and different line types corresponding to the contours for 90, 95 and 99 percent of the probability mass.



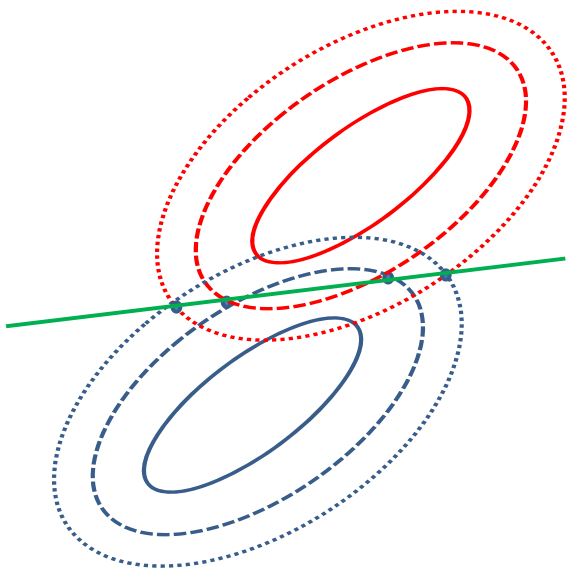
If you plug in the simplifying assumption that $\Sigma_c = \Sigma$ in the class boundary expression the quadratic terms in x cancel out and we get

$$\log \frac{p(x \mid y = c)}{p(x \mid y = l)} = -\frac{1}{2}(\mu_c + \mu_l)\Sigma^{-1}(\mu_c - \mu_l) + x\Sigma^{-1}(\mu_c - \mu_l) = 0$$

Notice that this is *linear* in x.

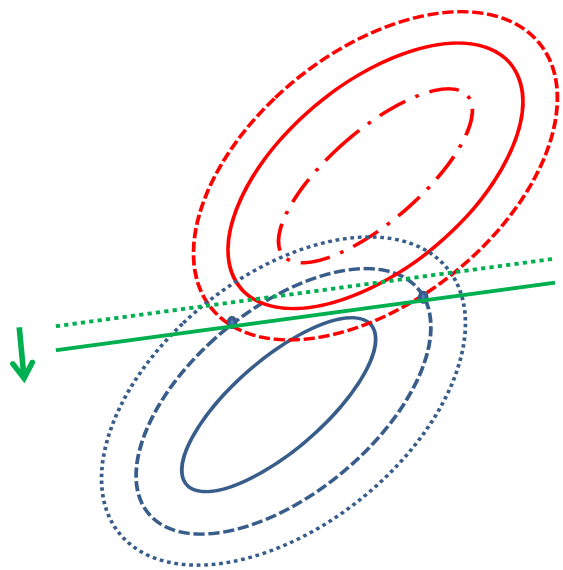
Building a linear classifier

To draw these boundaries, simply look for points in x-space where the posterior distributions have the same value in two classes. I have illustrated that below with two classes and different line types corresponding to the contours for 90, 95 and 99 percent of the probability mass.



Building a linear classifier

So what is the role of the prior? The prior will simply shift the contours of the data distribution center-outward if it increases, resulting in intersections with other class distribution contours further away from the distribution with a higher prior.



Let's focus on the rule instead of the boundary for a moment. The rule, as before, is the maximum posterior allocation. Here,

$$\hat{c}(x) = \arg\max_c \delta_c(x)$$

where

$$\delta_c(x) = \log \pi_c - \frac{1}{2}(x - \mu_c)'\Sigma_c^{-1}(x - \mu_c) - \frac{1}{2}\log \mid \Sigma_c \mid$$

If we consider the special case $\Sigma_c = \Sigma = \sigma^2 I$ (equal noise level for all features in all classes)

$$\delta_c(x) = \log \pi_c - \frac{1}{2\sigma^2}(x - \mu_c)'(x - \mu_c) + \textit{constant},$$

i.e., the nearest centroid classifier, adjusted for the prior.

If we consider the special case $\Sigma_c = \Sigma$ (equal noise level and feature correlation structure in all classes)

$$\delta_c(x) = \log \pi_c - \frac{1}{2\sigma^2}(x - \mu_c)'\Sigma^{-1}(x - \mu_c) + \textit{ constant} =$$

$$= \log \pi_c - \frac{1}{2}(\Sigma^{-1/2}(x - \mu_c))'(\Sigma^{-1/2}(x - \mu_c)) + \textit{ constant}$$

The matrix $\Sigma^{-1/2}$ is the "square root" matrix, meaning the $\Sigma^{-1/2}\Sigma^{-1/2} = \Sigma^{-1}$. Writing the expression this way has the following benefit: you know that $Cov(X) = \Sigma$. If you apply the transformation $\Sigma^{-1/2}$ to the data the resulting covariance is $V(\Sigma^{-1/2}\ X) = \Sigma^{-1/2}\Sigma\Sigma^{-1/2} = I$. That is, the transformation $\Sigma^{-1/2}$ serves the purpose of decorrelating and standardizing the data. This is called *sphering* the data and moves the classification problem into a new coordinate system.

We can write

$$\log \pi_c - \frac{1}{2}(\Sigma^{-1/2}(x - \mu_c))'(\Sigma^{-1/2}(x - \mu_c)) + \ \textit{constant} =$$

$$= \log \pi_c - \frac{1}{2}(\tilde{x} - \tilde{\mu}_c)'(\tilde{x} - \tilde{\mu}_c) + \ \textit{constant} =$$

$$= \log \pi_c - \frac{1}{2}\sum_{j=1}^{p}(\tilde{x}_j - \tilde{\mu}_{cj})^2 + \ \textit{constant}$$

where we define $\tilde{x} = \Sigma^{-1/2}x$ and $\tilde{\mu}_c = \Sigma^{-1/2}\mu_c$.
Notice that this is just the nearest centroid classifier! So LDA is nearest centroid in a new coordinate system formed by rotating and scaling the data x with respect to the covariance structure of x within each class.

Cautionary remarks

One take-home message from this lecture is that a key component in LDA is the inverse of Σ, the within-class covariance matrix. Problem?

- Σ may be very difficult to invert, numerically unstable, if the sample size n is not much bigger than the data dimension p.
- If some of the x-features are highly correlated, then the matrix Σ is also difficult to invert since it is near singular.
- Special case when $n < p$ or some x-s are perfectly correlated, the inverse of Σ does not exist. This is less of a worry since most programs will warn you about this. When we are *near* singular, that's when you need to pay attention. So correlated xs, high-dimensional data and small sample sizes are all situations when LDA can fail due to the poor performance of the inverse of Σ.

Cautionary remarks

There are some fixes we can consider. *Penalized discriminant analysis, PDA* is a numerical fix you may recognize from linear algebra class: we use $(\Sigma + \lambda I)^{-1}$ instead of Σ^{-1}. When Σ is near singular, adding a small values λ to the diagonal stabilizes the inverse operation.

In the high-dimensional case we will also consider *sparse* matrix inversion techniques, essentially restricting which values in the inverse are non-zero.

You can of course always reduce the number of features to consider, via manual selection, pre-screening or using principal components. We will come back to this in the high-dimensional part of the course.

More on Discriminant analysis

More on Discriminant analysis

As mentioned in the previous lecture, one problem with LDA stems from the instability of the estimate $\hat{\Sigma}$ when n is small and/or p is large and/or x-features are correlated.
Let's look at the source of this problem in more detail.

Inverse of Σ

Consider the eigendecomposition of $\hat{\Sigma} = UDU'$, where U are the eigenvectors and $U'U = I$ and D is a diagonal matrix containing the eigenvalues

$$\begin{pmatrix} d_1^2 & 0 & \cdots \\ 0 & d_2^2 & \cdots \\ \cdots & \cdots & \cdots \\ \cdots & 0 & d_p^2 \end{pmatrix}$$

where $d_1^2 > d_2^2 > \cdots$

Inverse of Σ

We can then write the inverse as

$$\hat{\Sigma}^{-1} = UD^{-1}U'$$

and plugging this into the expression for the DA rule

$$\hat{c}(x) = \arg\min_c (x - \hat{\mu}_c)' \hat{\Sigma}^{-1} (x - \hat{\mu}_c) =$$

$$= (x - \hat{\mu}_c)' UD^{-1}U' (x - \hat{\mu}_c) =$$

$$= [U'(x - \hat{\mu}_c)]' D^{-1} [U'(x - \hat{\mu}_c)] =$$

$$= (\tilde{x} - \tilde{\mu})' D^{-1} (\tilde{x} - \tilde{\mu}) =$$

$$= \sum_{j=1}^{p} \frac{(\tilde{x}_j - \tilde{\mu}_j)^2}{d_j^2}$$

which is a weighted euclidean distance between x and μ in the new coordinate system corresponding to the principal component directions U of $\hat{\Sigma}$.

Inverse of Σ

So LDA is really just nearest centroids in the new coordinate system that you get by rotating the data by U and scaling it by D. Writing $\hat{\Sigma} = UDU' = UD^{1/2}D^{1/2}U'$, we have that

$$\hat{\Sigma}^{-1} = UD^{-1/2}D^{-1/2}U' = \hat{\Sigma}^{-1/2}\hat{\Sigma}^{-1/2}$$

where we define $\hat{\Sigma}^{-1/2} = D^{-1/2}U'$ (square root of a diagonal matrix is just the square root of the elements).
Therefore we can write

$$(x - \hat{\mu}_c)'\hat{\Sigma}^{-1}(x - \hat{\mu}_c) = [\hat{\Sigma}^{-1/2}(x - \hat{\mu}_c)]'[\hat{\Sigma}^{-1/2}(x - \hat{\mu}_c)].$$

Inverse of Σ

The operation $\hat{\Sigma}^{-1/2}$ on x is called *sphering* the data. Why?

$$Cov(\hat{\Sigma}^{-1/2}X) = E[\hat{\Sigma}^{-1/2}X(\hat{\Sigma}^{-1/2}X)'] =$$

$$= E[\hat{\Sigma}^{-1/2}XX'\hat{\Sigma}^{-1/2}] = \hat{\Sigma}^{-1/2}E[XX']\hat{\Sigma}^{-1/2} = \hat{\Sigma}^{-1/2}\hat{\Sigma}\hat{\Sigma}^{-1/2} = I$$

I.e., in the new coordinate system Xs are uncorrelated and all features have variance 1.

Inverse of Σ

When $\hat{\Sigma}$ is near singular, $\hat{\Sigma}^{-1}$ behaves poorly (or may not even exist). The estimate is numerically unstable and small changes to the data can lead to big change for the inverse (and thus how you rotate the data before applying nearest centroids $\rightarrow$ poor classification performance.

The source of the problems lie in the direction u_j corresponding to small eigenvalues d_j since d_j appears in the denominator in the weighted euclidean distance computation. Small ds "blows up" the distance computation.

How do we fix this? The solution is to stabilize the inverse by reducing the influence of these small eigenvalues. This is done quite easily by simply adding something to the diagonal of $\hat{\Sigma}$ before you take the inverse.

Inverse of Σ

Use $\tilde{\Sigma} = (\hat{\Sigma} + \lambda I)$ and its inverse $\tilde{\Sigma}^{-1} = (\hat{\Sigma} + \lambda I)^{-1}$.
The impact of this is mainly limited to the small eigenvalues as we can see from the following

$$\hat{\Sigma} + \lambda I = UDU' + \lambda I = UDU' + \lambda UU' = U(D + \lambda I)U'$$

For large d_j the contribution λ is negligible.
Using $\tilde{\Sigma}^{-1}$ in your DA rule is called *penalized DA* (or regularized DA). When $\lambda = 0$ PDA is the same as LDA. If you make λ really big it starts to dominate the $d_j, \forall j$ which essentially means you start ignoring the correlation and scale structure in the data (get closer and closer to nearest centroids).

Flexible DA methods

Penalized DA addresses one problem with LDA, poor performance due to unstable estimates of $\hat{\Sigma}^{-1}$ (high variance). We also need to be concerned with potential BIAS, meaning the linear boundaries that LDA implicitly assumes are too simplistic to separate the classes from eachother.
One extension is then to use QDA (quadratic DA) we already looked at. This assumes that each class has its own correlation and scale structure. It leads to quadratic boundaries in x-space and is quite costly in terms of the number of parameters you need to estimate. This can reduce BIAS but lead to a large increase in VARIANCE so the end result is little or no improvement over LDA (or even worse performance if VARIANCE grows quickly as would be the case for very large p).

Mixture DA

A very nice alternative to QDA that generalizes LDA to more flexible boundaries is *mixture discriminant analysis* (MDA), introduced by Hastie and Tibshirani in the mid-90s.
We make the classifier more complex by allowing each class to be made up of many, simple components (as opposed to one complex component as in QDA). By combining many simple shapes we can build up quite complex shapes in x-space! Example: you can build a donut shape in x-space with 5-6 spherical distributions.

Mixture DA

We assume the following model for each class

$$p(x \mid y = c) = \sum_{r=1}^{R_c} \pi_{cr} N(x; \mu_{cr}, \Sigma)$$

Notice

- There are R_c components for class c and this may differ from class to class
- Each component has a different contribution or "weight" in the class distribution, π_{cr}
- Each component within and between the classes have the same shape, Σ.

For large p, the last bullet constitutes a large savings in terms of the number of parameters compared to QDA.

Mixture DA

As we saw before, estimating parameters in DA was relatively easy (just computing means, proportions and covariances). Here, the situation is more complex since we don't actually know which component r within class c that an observation belongs to. The components are artificial constructs that allows to generate complex data distribution shapes, but we don't know anything about them a-priori.

Mixture DA

For models like these we start by working out what we would do if we did know about the component memberships. Then things are just as easy as in standard DA. We would take all the observations in each component and compute the parameter estimates:

- For all classes c, compute for each component r within this class: $\hat{\mu}_{cr} = \sum_{i \in cr} x_i / N_{cr}, \quad N_{cr} = \sum_{i \in cr} 1$
- $\hat{\pi}_{cr} = N_{cr}/N_c, \quad N_c = \sum_{y_i = c} 1$
- $\hat{\Sigma} = \sum_{c=1}^{C} \sum_{r=1}^{R_{cr}} (x_i - \hat{\mu}_{cr})(x_i - \hat{\mu}_{cr})' / (N - C)$

So now we know what we would do if we knew which observations belonged to which component. Let's ask the other hypothetical question: what would we do if we knew all the parameters of the class components? If we knew those we could just apply the maximum posterior principle to classify observations at the *component* level rather than the class level:

$$\arg \max_{r \in \textit{class } c} p(i \in \textit{ component } r \textit{ of class } c \mid y_i = c, \mu_{cr}, \pi_{cr}, \Sigma)$$

To get something we can work with for the posterior we apply Bayes theorem:

$$\arg \max_{r \in \textit{class } c} p(i \in \textit{ component } r \textit{ of class } c \mid y_i = c, \mu_{cr}, \pi_{cr}, \Sigma) =$$

$$= \frac{\pi_{cr} N(x_i; \mu_{cr}, \Sigma)}{\sum_{l=1}^{R_c} \pi_{cl} N(x_i; \mu_{cl}, \Sigma)}$$

We usually denote this posterior by $\eta_{cr}(x_i)$.

Mixture DA

Let's put the two things together. Given a component classification we know how to estimate the parameters, given the parameters we know how to classify observations into the components. We simply iterate these two steps until the results converge.
How do you start off? Usually by a random selection of center points for each component and a nearest centroid classification. Then you start estimating the parameters etc. You may need to try a couple of different starting points in order to ensure convergence to the best fitting component distribution.

Mixture DA

The above iterative scheme is a variant of the so-called EM (Expectation-Maximization) algorithm which is a very important tool for dealing with models that have this additional complication - a bit of information is missing (component labels). The above algorithm is a variant called *classification* EM since I classify the observations into only one of the components.
Since the components are an artificial construct they are rarely well separated and then one might argue that using observations only for one component is an inefficient use of data if the components overlap and blend into each other.

The remedy for this is to use *weighs* in the parameter estimation and let those weights be the component posterior probabilities for each observation. This way, an observation can contribute to all components within a class, just more to the component it fits best to.
The weights are thus the $\eta_{cr}(x_i)$ we defined above.

Mixture DA

We use the weights to come up with another form for the parameter estimates:

- For all classes c, compute for each component r within this class: $\hat{\mu}_{cr} = \sum_{y_i=c} \eta_{cr}(x_i) x_i / N_{cr}, \quad N_{cr} = \sum_{y_i=c} \eta_{cr}(x_i)$
- $\hat{\pi}_{cr} = N_{cr}/N_c, \quad N_c = \sum_{y_i=c} \sum_{r=1}^{R_c} \eta_{cr}(x_i)$
- $\hat{\Sigma} = \sum_{c=1}^{C} \sum_{y_i=c} \frac{\sum_{r=1}^{R_{cr}} \eta_{cr}(x_i)(x_i - \hat{\mu}_{cr})(x_i - \hat{\mu}_{cr})'}{\sum_{y_i=c} \eta_{cr}(x_i)} / (N - C)$

EM algorithm

The EM-algorithm is something we will come back to when we do clustering.
Here, the E-step is the computation of the posterior probabilities that an observation belongs to a certain component within a class. This produces the weights $\eta_{cr}(x)$.
The M-step is the parameter estimation step where the weights are used to allow for observations within a class to contribute to the estimation of all the class components.

Validation

Both PDA and MDA can be tuned to be more or less flexible/local.
For PDA, you have to choose λ just big enough that the instability of the matrix inverse operation is surpressed (variance reduced) without affecting the rotation of the data in the leading principal component directions (would lead to bias).
For MDA, you have to choose the number of components for each class just big enough so that the class shapes are adapting to the data shape but not so big that you are adapting to random noise in the data or don't have enough observations to train the component parameters on.
As always in statistics - we have to consider the bias-variance trade-off!

Validation

Both PDA and MDA tuning parameter selection can be done via cross-validation.
PDA:

1. Split the data into B parts
2. For $b = 1, \cdots, B$
   For $\lambda = 0, \cdots, \lambda_{max}$
   1. Apply PDA with value λ to all data except the b-th test data
   2. Predict class labels on the b-th test data
   3. Compute the test error rate TE_λ^b
3. Compute the average error rate across folds b for each λ: $TE_\lambda = \sum_b TE_\lambda^b / B$
4. Choose the λ that minimizes this test error: $\lambda^* = \arg\min TE_\lambda$

Validation

For MDA it's slightly more complicated since you have to consider different number of components for each class. Let's denote a set of component numbers by $R = (R_1, R_2, \cdots, R_C)$, e.g. $R = (3, 4)$ if you use 3 components for class 1 and 4 components for class 4. Let's enumerate all sets R to consider as $R^m, m = 1, \cdots, M$

1. Split the data into B parts
2. For $b = 1, \cdots, B$
   For $m = 1, \cdots, M$
   1. Apply MDA, with component set $R^m = (R_1^m, R_2^m, \cdots, R_C^m)$, to all data except the b-th test data
   2. Predict class labels on the b-th test data
   3. Compute the test error rate TE_m^b
3. Compute the average error rate across folds b for each m: $TE_m = \sum_b TE_m^b / B$
4. Choose the m that minimizes this test error: $m^* = \arg\min TE_m$

The space of component sets to consider is quite large, but it's usually a safe strategy to start out small and search forward by adding one component at a time to the class where the improvement is the biggest.