Data Analysis and Model Classification
Guidesheet 3

Ruslan Aydarkhanov Bastien Orset Julien Rechenmann
Ricardo Chavarriaga José del R. Millán

With the help of this guidesheet, you will familiarize yourselves with Linear and Quadratic Discriminant Analysis classifiers (usually referred to as LDA and QDA, respectively) and the basics of estimating the performance of classifiers.

LDA/QDA classifiers

In this section we will train two classifiers for each database on all the samples of the training data. For both databases, the dimensionality of the original samples may be too large compared to the number of available samples. Therefore we will choose only a subset of the original features[1]. In the ERP database, you can select a subset of 60 features by taking every fifth feature, starting from the first feature and up to feature 300 by typing `features = features(:,1:5:300);`

Linear discriminant classifiers These classifiers assume an equal covariance matrix for both classes. The covariance matrix can be either *full*, in which case the covariance between features is taken into account, or *diagonal*, in which case only the variance of the features is taken into account. This method will yield a linear separating hyperplane.

Quadratic discriminant classifiers These classifiers assume that classes can have different covariance matrices. As before, the covariance matrices can be either full, in which case covariance between features is taken into account, or diagonal, in which case only the variance of the features is taken into account. The separating hyperplane is quadratic due to the additional quadratic term (obviously).

If you attended the course, you should already understand how to train those different classifiers (i.e. how to find the methods' parameters w).

The statistics toolbox from MATLAB has functions to train an LDA/QDA classifier, `classifier = fitcdiscr(features, labels, 'discrimtype', type);`, and classify a set of *test* features: `yhat = predict(classifier, features);` The outputs for the commands are, respectively, the trained classifier and the predicted classes for each sample in the `test_set` (in this case, the `yhat` is the result of the training set).

From the output `yhat` of the `predict`[2] function, we can compute the *classification error* and *class error* introduced last week.

Hands on

- Use your whole dataset to train a classifier using the `fitcdiscr` function and also use the same whole dataset (`features`) for prediction with the `predict()` function. Compute the classification

[1] Later in this course we will show some methods for selecting the best features

[2] You will find a lot of functions named predict in MATLAB help. In this special case, `which predict` will not lead you to the correct path. Locate the hyperlink of `predict` at the bottom of `fitcdiscr` in help browser (click "Reference page for fitcdiscr" if you type `help fitcdiscr`.)

error. Check the MATLAB help for `fitcdiscr()`. The default classifier is *linear*. Do the same using *diaglinear*, *diagquadratic* and *quadratic* classifiers. Based on the classification error, which classifier would you choose?

- Check the help function of fitcdiscr to find out how to specify prior probabilities for the two databases. Then, based on the chosen classifiers (for the two databases), repeat again by specifying uniform prior when calling `fitcdiscr()`. This time, look at both classification error and class error and compare the cases with and without uniform prior. What do you observe and how do you explain the values based on the prior. Also, regarding the classification error and the class error, would you argue that one is more useful than the other. Make some assumption if necessary, and choose one for the later part of this guide sheet.

Training and testing error

Now we will assess the *generalization* of the classifier on an *unseen* dataset or *test set*. We will be differentiating the **training** error (predict on the set of samples that are used to train a classifier) from the **testing** error, obtained using the unseen datasets.

Hands on

- Separate your dataset into two sets of equal size, *set1* and *set2*. Train a diaglinear classifier using the data from *set1* and compute the error on the same data (i.e. training error). Compare it to the **testing** error; i.e., the error on *set1* after training the classifier using the data from *set2*. What happened?

- Repeat the same using *linear*, *diagquadratic* and *quadratic* classifiers. Would you still choose the same classifier as in the last section? Why does the improvement on the **training** error not improve the **testing** error? Why can you not use the *quadratic* classifier?

- These classifiers use different types of models. Roughly, the complexity can be measured from the number of parameters that are needed to be estimated. How many parameters does each classifier have? Compare these numbers to the number of training samples you are using to train the classifier. Do you think these classifiers are robust, why?

- Compute the **training** and **testing** error again, but on *set2* and notice the variability of the performance.

Cross-validation

In the last section, we saw that it is vital to estimate the classification accuracy on an *unseen* dataset. However, when separating the dataset into training and testing sets, you reduce the amount of data used for training! On the other hand, can you think about a disadvantage of having too small testing sets?

If we had datasets containing millions of samples, we could afford "losing" samples for the test set, however, with a small database like ours, we want to use every single possible bit of data available, while still testing on unseen data. One approach to tackle this is k-fold cross-validation. In this case, you split your data into k subsets (i.e., folds) of the data of equal size[3]. Then use $k-1$ subsets to train a classifier and the remaining subset to test it, say the first subset. Then you iterate and select the second subset for testing and use subsets $\{1, 3, 4, \ldots\}$ for training, etc...

In MATLAB, you can use the `cvpartition` command to create a 10-fold cross-validation partition:

either `cp = cvpartition(N,'kfold',10);` or `cp = cvpartition(labels,'kfold',10);`, with `N` the total number of samples.

Use `cp.training(i)` and `cp.test(i)` to get the training and testing set.

When $k = N$, you have only one sample in each test subset. This special case is referred to as "leave-one-out" cross-validation.

[3]Obviously, the size can be different by one between folds, depending on the total number of samples and k.

Hands on

- Compare the partition for `cvpartition(N,'kfold',10)` and `cvpartition(labels_set,'kfold',10)`. How many samples do you have per class and per fold? Which one would you recommend to use?
- Implement a $k = 10$-fold cross-validation to estimate the cross-validation error of all classifiers tested before (if possible), use a `for` loop to compute the test error for each fold. Your cross-validation error is the mean of the test errors obtained over the 10 folds. Notice that you can also compute the standard deviation of the cross-validation error. This is an important measure of how stable your performances are.
- Change the parameters to do a leave-one out cross-validation.
- Repeat both cross-validations (10-fold and leave-one-out), changing the partition using `repartition(cp)` each time. Does the result change for both cross-validations? Why? Now you should know how to get a more stable estimation of accuracy. Is there any advantages of having classification performances that vary or on the contrary that are always the same?

Confusion matrix

You might have noticed that the amount of samples for each class of your dataset differs. In some applications (e.g. medical), misclassifying class 1 as class 0 is much worse than misclassifying class 0 as class 1. To get further insight into the performance of your classifier, we use *confusion matrices*. You can use the `confusionmat` command to compute *confusion matrices*. As usual, check the MATLAB help.

In our binary classification problem, the dimension of matrix is 2×2. The element in first row and column is called *True Positive*, which represents the amount of correctly predicted *positive* class (it is label 0 if you use `confusionmat`). Second column of first row is *False Negative* that shows the number of samples misclassified as *Negative* class; i.e., the number of samples should be predicted as *Positive* but wrongly predicted as *Negative*. For second row, first column is *False Positive*. The number of samples misclassified as *Positive*. The second column is *True Negative*; number of samples correctly predicted as *Negative* class.

Hands on

- Compute the confusion matrix of your final classifier and interpret the result.
- How are the confusion matrix and the classification error linked? How do they differ?
- Can you find a way of modifying your classifier to give more weight to a certain class?