Laboratory 6

In this laboratory we will focus on generative models for classification.

Gaussian models

In the first part of this laboratory we will solve the IRIS classification task using Gaussian classifiers. We have already introduced the IRIS dataset in Laboratory 2. We can load the dataset using the code that we already developed. Alternatively, the dataset is already available from the `sklearn` library (we need to transpose the data matrix since we work with column representations of feature vectors):

```
import sklearn.datasets
def load_iris():
    D,L = sklearn.datasets.load_iris()['data'].T, sklearn.datasets.load_iris()['target']
    return D, L
```

In the following examples we use the dataset as returned by the aforementioned `load_iris`. We split the datasets in two parts: the first part will be used for model training, the second for evaluation (validation set). You can try implementing yourself the split. The results in the following sections are based on the following code:

```
def split_db_2to1(D, L, seed=0):

    nTrain = int(D.shape[1]*2.0/3.0)
    numpy.random.seed(seed)
    idx = numpy.random.permutation(D.shape[1])
    idxTrain = idx[0:nTrain]
    idxTest = idx[nTrain:]

    DTR = D[:, idxTrain]
    DTE = D[:, idxTest]
    LTR = L[idxTrain]
    LTE = L[idxTest]

    return (DTR, LTR), (DTE, LTE)

D, L = load_iris()
# DTR and LTR are training data and labels, DTE and LTE are evaluation (or more
    precisely validation) data and labels
(DTR, LTR), (DTE, LTE) = split_db_2to1(D, L)
```

We use 100 samples for training and 50 samples for evaluation.

Multivariate Gaussian Classifier

The first model we implement is the Multivariate Gaussian Classifier (MVG). As we have seen, the classifier assumes that samples of each class $c \in \{0, 1, 2\}$ can be modeled as samples of a multivariate Gaussian distribution with class-dependent mean and covariance matrices

$$f_{\boldsymbol{X}|C}(\boldsymbol{x}|c) = \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

The ML solution for the parameters is given by the empirical mean and covariance matrix of each class

$$\mu_c^* = \frac{1}{N_c}\sum_i \boldsymbol{x}_{c,i} \ , \quad \boldsymbol{\Sigma}_c^* = \frac{1}{N_c}\sum_i (\boldsymbol{x}_{c,i} - \boldsymbol{\mu}_c^*)(\boldsymbol{x}_{c,i} - \boldsymbol{\mu}_c^*)^T$$

where $\boldsymbol{x}_{c,i}$ is the i-th sample of class c.

Compute the ML estimates for the classifier parameters $(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0), (\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1), (\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)$ (see also Laboratory

3 for how this can be done efficiently). You should obtain

$$\boldsymbol{\mu}_0 = \begin{bmatrix} 4.96129032 \\ 3.42903226 \\ 1.46451613 \\ 0.2483871 \end{bmatrix} \quad \boldsymbol{\Sigma}_0 = \begin{bmatrix} 0.13140479 & 0.11370447 & 0.02862643 & 0.01187305 \\ 0.11370447 & 0.16270552 & 0.01844953 & 0.01117586 \\ 0.02862643 & 0.01844953 & 0.03583767 & 0.00526535 \\ 0.01187305 & 0.01117586 & 0.00526535 & 0.0108845 \end{bmatrix}$$

$$\boldsymbol{\mu}_1 = \begin{bmatrix} 5.91212121 \\ 2.78484848 \\ 4.27272727 \\ 1.33939394 \end{bmatrix} \quad \boldsymbol{\Sigma}_1 = \begin{bmatrix} 0.26470156 & 0.09169881 & 0.18366391 & 0.05134068 \\ 0.09169881 & 0.10613407 & 0.08898072 & 0.04211203 \\ 0.18366391 & 0.08898072 & 0.21955923 & 0.06289256 \\ 0.05134068 & 0.04211203 & 0.06289256 & 0.03208448 \end{bmatrix}$$

$$\boldsymbol{\mu}_2 = \begin{bmatrix} 6.45555556 \\ 2.92777778 \\ 5.41944444 \\ 1.98888889 \end{bmatrix} \quad \boldsymbol{\Sigma}_2 = \begin{bmatrix} 0.30080247 & 0.08262346 & 0.18614198 & 0.04311728 \\ 0.08262346 & 0.08533951 & 0.06279321 & 0.05114198 \\ 0.18614198 & 0.06279321 & 0.18434414 & 0.04188272 \\ 0.04311728 & 0.05114198 & 0.04188272 & 0.0804321 \end{bmatrix}$$

Given the estimated model, we now turn our attention towards inference for a test sample x. As we have seen, the final goal is to compute class posterior probabilities $P(c|\boldsymbol{x})$. We split the process in three stages. The first step consists in computing, for each test sample, the likelihoods

$$f_{\boldsymbol{X}|C}(\boldsymbol{x}_t|c) = \mathcal{N}(\boldsymbol{x}_t|\boldsymbol{\mu}_c^*, \boldsymbol{\Sigma}_c^*)$$

We have seen how to compute the log-densities $\log f_{\boldsymbol{X}|C}(\boldsymbol{x}_t|c)$ in Laboratory 5. Compute the three likelihoods $f_{\boldsymbol{X}|C}(\boldsymbol{x}_t|0)$, $f_{\boldsymbol{X}|C}(\boldsymbol{x}_t|1)$ and $f_{\boldsymbol{X}|C}(\boldsymbol{x}_t|2)$ for all test samples (remember to exponentiate the log-densities).

Suggestion: store class-conditional probabilities in a *score* matrix `S`. `S[i, j]` should be the class conditional probability for sample `j` given class `i`. Each row of the score matrix corresponds to a class, and contains the conditional log-likelihoods for all the samples for that class.

We can now compute class posterior probabilities combining the score matrix with prior information. In the following we assume that the three classes have the same prior probability $P(c) = 1/3$. We can thus compute the joint distribution for samples and classes

$$f_{\boldsymbol{X},C}(\boldsymbol{x}_t, c) = f_{\boldsymbol{X}|C}(\boldsymbol{x}_t|c)P_C(c)$$

Compute the matrix of joint densities `SJoint`. This requires multiplying each row of `S` by the prior probability of the corresponding class ($1/3$). You can check your solution against the matrix contained in file `Solution/SJoint_MVG.npy`

Finally, we can compute class posterior probabilities as

$$P(C = c|\boldsymbol{X} = \boldsymbol{x}_t) = \frac{f_{\boldsymbol{X},C}(\boldsymbol{x}_t, c)}{\sum_{c'} f_{\boldsymbol{X},C}(\boldsymbol{x}_t, c')}$$

This requires summing over all classes the joint probability, which we have stored in matrix `SJoint`, to compute the marginal densities

$$f_{\boldsymbol{X}}(\boldsymbol{x}_t) = \sum_c f_{\boldsymbol{X},C}(\boldsymbol{x}_t, c) \ .$$

This can be achieved through

```
SMarginal = vrow(SJoint.sum(0))
```

Compute the array of class posterior probabilities `SPost`. The predicted label is obtained as the class that has maximum posterior probability. We can use the `argmax` method with `axis` keyword to compute the array of predicted labels.

Once you have computed predicted classes for each evaluation sample, you should compute the accuracy of the model. For the moment, we just measure the accuracy as the number of correctly labeled points over the number of evaluation samples:

$$acc = \frac{\#\text{ correct predictions}}{\#\text{ of samples}}$$

Alternatively, we can compute the error rate as

$$err = \frac{\#\text{ wrong predictions}}{\#\text{ of samples}} = 1 - acc$$

For the considered split, you should obtain an error rate of 4.0%.

Suggestion: you can compute an array of boolean values correspoding to whether predicted and real labels are equal or not. Summing the elements of a boolean array gives the number of elements that are `True`.

As we have already discussed, working directly with densities is often problematic, due to numerical issues. It's useful to implement the whole procedure directly in terms of log-densities (if we need, we can recover posterior probabilities at the end).

Working with log-densities, we need to compute

$$\log f_{\boldsymbol{X}|C}(\boldsymbol{x}_t|c) = \log \mathcal{N}(\boldsymbol{x}_t|\boldsymbol{\mu}_c^*, \boldsymbol{\Sigma}_c^*)$$

The joint log-density is given

$$l_c = \log f_{\boldsymbol{X},C}(\boldsymbol{x}_t, c) = \log f_{\boldsymbol{X}|C}(\boldsymbol{x}_t|c) + \log P_C(c)$$

We now need to compute the marginal log-density $\log f_{\boldsymbol{X}}(\boldsymbol{x}_t) = \log \sum_c f_{\boldsymbol{X},C}(\boldsymbol{x}_t, c)$. We can rewrite the expression as

$$\log f_{\boldsymbol{X}}(\boldsymbol{x}_t) = \log \sum_c e^{l_c}$$

However, we need to take care that computing the exponential terms may result again in numerical errors. A robust method to compute $\log \sum_c e^{l_c}$ consists in rewriting the sum as

$$\log \sum_c e^{l_c} = l + \log \sum_c e^{l_c - l}$$

where $l = \max_c l_c$. We can then safely compute the exponentials $e^{l_c - l}$, since at least one of them will be equal to 1 (we may still have numerical issues of all the remaining terms are very small, but the effects are much less dramatic). This is known as the log-sum-exp trick, and is already implemented in `scipy` as `scipy.special.logsumexp`. We can thus use `scipy.special.logsumexp(s)`, where `s` is the array that contains the joint log-probabilities for a given sample, to compute $\log f_{\boldsymbol{X}}(\boldsymbol{x}_t)$. `scipy.special.logsumexp` also allows specifying an axis, thus we can directly compute the array of marginals for all samples directly from the matrix of joint log-densities as we did before.

Let `logS` be the matrix of class-conditional log-likelihoods, and `logSJoint` be the matrix containing the log-densities for each class and each sample (in practice, the logarithm of the entries of `SJoint`, computed directly using the log-density function)

```
logSJoint = logS + vcol(numpy.log(prior))
```

The log-marginal corresponds to

```
logSMarginal = vrow(scipy.special.logsumexp(logSJoint, axis=0))
```

Finally, we can compute log-posteriors as

$$\log P(C = c|\boldsymbol{X} = \boldsymbol{x}_t) = \log f_{\boldsymbol{X},C}(\boldsymbol{x}_t, c) - \log f_{\boldsymbol{X}}(\boldsymbol{x}_t)$$

corresponding to

```
logSPost = logSJoint - logSMarginal
SPost = numpy.exp(logSPost)
```

Implement the inference chain in the log-domain, and check that the posterior probabilities are the same as with the previous approach (up to numerical precision errors). You can also check intermediate results against the matrices provided in the `Solution` folder `logSJoint_MVG.npy` , `logMarginal_MVG.npy` and `logPosterior_MVG.npy`

Naive Bayes Gaussian Classifier

We now consider the Naive Bayes version of the classifier. As we have seen, the Naive Bayes version of the MVG is simply a Gaussian classifier where the covariance matrices are diagonal. The ML solution for the mean parameters is the same, whereas the ML solution for the covariance matrices is

$$\text{diag}(\mathbf{\Sigma}_c^*) = \text{diag}\left[\frac{1}{N_c}\sum_i (\boldsymbol{x}_{c,i} - \boldsymbol{\mu}_c^*)(\boldsymbol{x}_{c,i} - \boldsymbol{\mu}_c^*)^T\right]$$

i.e., the diagonal of the ML solution for the MVG model. Implement the Naive Bayes classifier.

NOTE: since the number of features is small, we can adapt the MVG code by simply zeroing the out-of-diagonal elements of the MVG ML solution. This can be done, for example, multiplying element-wise the MVG ML solution with the identity matrix. The rest of the code remains unchanged. If we have large dimensional data, it may be advisable to implement ad-hoc functions to work directly with just the diagonal of the covariance matrices (we won't do this in this course).

The accuracy for the Naive Bayes classifier should be again 4.0% for this dataset. The `Solution` folder contains all the intermediate results, both in the likelihood and in the log-likelihood domain.

Tied Covariance Gaussian Classifier

We now consider the Tied covariance version of the classifier. In this case, the class covariance matrices are tied, with $\mathbf{\Sigma}_c = \mathbf{\Sigma}$. We have seen that the ML solution for the class means is again the same. The ML solution for the covariance matrix is given by the empirical within-class covariance matrix

$$\mathbf{\Sigma}^* = \frac{1}{N}\sum_c \sum_i (\boldsymbol{x}_{c,i} - \mu_c^*)(\boldsymbol{x}_{c,i} - \mu_c^*)^T$$

Compute the ML solution for the model. Remember that we have already computed within-class covariance matrices when we implemented LDA. Alternatively, we can observe that $\mathbf{\Sigma}^* = \frac{1}{N}\sum_c N_c \mathbf{\Sigma}_c^*$, where $\mathbf{\Sigma}_c^*$ is the ML solution for class c for the MVG classifier.

You should obtain

$$\mathbf{\Sigma}^* = \begin{bmatrix} 0.23637589, & 0.09525344, & 0.1364944, & 0.03614529 \\ 0.09525344, & 0.11618517, & 0.05768855, & 0.0357726, \\ 0.1364944, & 0.05768855, & 0.14992811, & 0.03746458 \\ 0.03614529, & 0.0357726, & 0.03746458, & 0.04291763 \end{bmatrix}$$

The accuracy for the tied covariance classifier should be 2.0% for this dataset. Again, the `Solution` folder contains all the intermediate results, both in the likelihood and in the log-likelihood domain.

Binary tasks: log-likelihood ratios and MVG

We now focus on the same binary task we employed for LDA (see Laboratory 3), which requries classifying only two kinds of flowers, iris versicolor and iris virginica. You can refer to Laboratory 2 to build the 2-class dataset.

Although we could proceed in the same way as for the multiclass iris problem, for binary tasks we have seen that we can cast the classification as a comparison of a score, the *log-likelihood ratio*, with a threshold t that depends on class priors.

Assuming that class 2 is the *true* class and class 1 is the *false* class (i.e., we are testing the hypothesis that a flower is from the virginica class), the log-likelihood ratio is defined as

$$s(\boldsymbol{x}_t) = llr(\boldsymbol{x}_t) = \log\frac{f_{\boldsymbol{X}|C}(\boldsymbol{x}_t|2)}{f_{\boldsymbol{X}|C}(\boldsymbol{x}_t|1)} = \log\frac{\mathcal{N}(\boldsymbol{x}_t|\boldsymbol{\mu}_2, \mathbf{\Sigma}_2)}{\mathcal{N}(\boldsymbol{x}_t|\boldsymbol{\mu}_1, \mathbf{\Sigma}_1)} = \log\mathcal{N}(\boldsymbol{x}_t|\boldsymbol{\mu}_2, \mathbf{\Sigma}_2) - \log\mathcal{N}(\boldsymbol{x}_t|\boldsymbol{\mu}_1, \mathbf{\Sigma}_1)$$

Compute the MVG ML solution for the two-class problem, and then compute the 1-D array of log-likelihood ratios (LLRs) for all the validation samples for the MVG model from the arrays of log-densities. You can compare your results with the contents of array `'Solution/llr_MVG.npy'`).

To compute our predictions we now compare the LLRs with a threshold. For the moment, we assume uniform priors $P(C=2) = P(C=1) = \frac{1}{2}$. Therefore, the threshold becomes

$$t = -\log \frac{P(C=2)}{P(C=1)} = 0$$

The predictions are thus obtained by assigning label 2 to samples whose LLR is greater or equal to 0, and label 1 to the other samples. Compute the error rate. You should obtain 8.8%
Now try again with the tied model. As we have seen, the tied model provides the same classification rule of LDA, and the two model are equivalent when the prior is uniform. Verify that the tied Gaussian model provides the same labels as LDA (Laboratory 2).

Project

Apply the MVG model to the project data. Split the dataset in model training and validation subsets (important: use the same splits for all models, including those presented in other laboratories), train the model parameters on the model training portion of the dataset and compute LLRs

$$s(\boldsymbol{x}_t) = llr(\boldsymbol{x}_t) = \frac{f_{\boldsymbol{X}|C}(\boldsymbol{x}_t|1)}{f_{\boldsymbol{X}|C}(\boldsymbol{x}_t|0)} \tag{1}$$

(i.e., with class True, label 1 on top of the ratio) for the validation subset. Obtain predictions from LLRs assuming uniform class priors $P(C=1) = P(C=0) = 1/2$. Compute the corresponding error rate (*suggestion:* in the next laboratories we will modify the way we compute predictions from LLRs, we therefore recommend that you keep separated the functions that compute LLRs, those that compute predictions from LLRs and those that compute error rate from predictions).

Apply now the tied Gaussian model, and compare the results with MVG and LDA. Which model seems to perform better?

Finally, test the Naive Bayes Gaussian model. How does it compare with the previous two?

Let's now analyze the results in light of the characteristics of the features that we observed in previous laboratories. Start by printing the covariance matrix of each class (you can extract this from the MVG model parameters). The covariance matrices contain, on the diagonal, the variances for the different features, whereas the elements outside of the diagonal are the feature co-variances. For each class, compare the covariance of different feature pairs with the respective variances. What do you observe? Are co-variance values large or small compared to variances? To better visualize the strength of co-variances with respect to variances we can compute, for a pair of features i, j, the Pearson correlation coefficient, defined as

$$Corr(i,j) = \frac{Cov(i,j)}{\sqrt{Var(i)}\sqrt{Var(j)}}$$

or, directly matrix form,

```
Corr = C / ( vcol(C.diagonal()**0.5) * vrow(C.diagonal()**0.5) )
```

where `C` is a covariance matrix. The correlation matrix has diagonal elements equal to 1, whereas out-of-diagonal elements correspond to the correlation coefficients for all feature pairs, with $-1 \leq Corr(i,j) \leq 1$. When $Corr(i,j) = 0$ the features i, j are uncorrelated, whereas values close to ± 1 denote strong correlation.

Compute the correlation matrices for the two classes. What can you conclude on the features? Are the features strongly or weakly correlated? How is this related to the Naive Bayes results?

The Gaussian model assumes that features can be jointly modeled by Gaussian distributions. The goodness of the model is therefore strongly affected by the accuracy of this assumption. Although visualizing 6-dimensional distributions is unfeasible, we can analyze how well the assumption holds for single (or pairs) of features. In Laboratory 5 we separately fitted a Gaussian density over each feature for each class. This corresponds to the Naive Bayes model. What can you conclude on the goodness of the Gaussian assumption? Is it accurate for all the 6 features? Are there features for which the assumptions do not look good?

To analyze if indeed the last set of features negatively affects our classifier because of poor modeling assumptions, we can try repeating the classification using only feature 1 to 4 (i.e., discarding the last 2 features). Repeat the analysis for the three models. What do you obtain? What can we conclude on discarding the last two features? Despite the inaccuracy of the assumption for these two features, are the Gaussian models still able to extract some useful information to improve classification accuracy?

In Laboratory 2 and 5 we analyzed the distribution of features 1-2 and of features 3-4, finding that for features 1 and 2 means are similar but variances are not, whereas for features 3 and 4 the two classes mainly differ for the feature mean, but show similar variance. Furthermore, the features also show limited correlation for both classes. We can analyze how these characteristics of the features distribution affect the performance of the different approaches. Repeat the classification using only features 1-2 (jointly), and then do the same using only features 3-4 (jointly), and compare the results of the MVG and tied MVG models. In the first case, which model is better? And in the second case? How is this related to the characteristics of the two classifiers? Is the tied model effective at all for the first two features? Why? And the MVG? And for the second pair of features?

Finally, we can analyze the effects of PCA as pre-processing. Use PCA to reduce the dimensionality of the feature space, and apply the three classification approaches. What do you observe? Is PCA effective for this dataset with the Gaussian models? Overall, which is the model that provided the best accuracy on the validation set?