Laboratory 5

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):

```python
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. You can try implementing yourself the split. The results in the following sections are based on the following code:

```python
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 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

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

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

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

We have seen how to compute these terms in Laboratory 4. Compute the the three likelihoods $f_{\boldsymbol{X}|C}(\boldsymbol{x}|0)$, $f_{\boldsymbol{X}|C}(\boldsymbol{x}|1)$ and $f_{\boldsymbol{X}|C}(\boldsymbol{x}|2)$ for all test samples.

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`.

We can now compute class 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

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

Compute the matrix of joint probabilities `SJoint`. This requires multiplying each row of `S` by the prior probability of the corresponding class $(1/3)$.

Finally, we can compute class posterior probabilities as

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

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

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

Compute the array of class posterior probabilities `SPost` (remember that we can sum all rows of a matrix providing the `axis` keyword to method `sum`). The predicted label is obtained as the class that has maximum posterior probability (again, we can use `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 and error rate of 4.0%.

Suggestion: you can compute an array of boolean values correspoding to whether predicted and real label 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

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

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

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

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 $\log f_{\boldsymbol{X}}(\boldsymbol{x})$. `scipy.special.logsumexp(s)` 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.

Finally, we can compute log-posteriors as

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

Implement the inference chain in the log-domain.

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

$$\mathrm{diag}(\boldsymbol{\Sigma}_c^*) = \mathrm{diag}(\boldsymbol{x}_{c,i} - \boldsymbol{\mu}_c^*)(\boldsymbol{x}_{c,i} - \boldsymbol{\mu}_c^*)^T$$

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.

Tied Covariance Gaussian Classifier

We now consider the Tied covariance version of the classifier. In this case, the class covariance matrices are tied, with $\boldsymbol{\Sigma}_c = \boldsymbol{\Sigma}$. 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

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

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 $\boldsymbol{\Sigma}^* = \frac{1}{N} \sum_c N_c \boldsymbol{\Sigma}_c^*$, where $\boldsymbol{\Sigma}_c^*$ is the ML solution for class c for the MVG classifier.

You should obtain

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

The accuracy for the Naive Bayes classifier should be again 2.0% for this dataset.

K-fold cross validation

Given the limited number of evaluation samples, we turn our attention to cross-validation approaches. The K-fold cross validation method can be employed to split the dataset in K, non overlapping, subsets. We then iteratively consider one subset as evaluation, and the remaining $K - 1$ as training set. We can then accumulate the number of errors and compute a global accuracy.

The leave-one-out approach brings the tchnique to the limiting case where the number of folds is equal to the number of sa,ples. Thus, for each sample, we estimate a model using all the remaining points and we compute the predicted label for the considered sample using such model.

Implement the K-fold and leave-one-out mechanisms, and test the four different classifiers. For the leave-one-out method you should get the following error rates:

- Multivariate Gaussian model: 2.7%

- Naive Bayes model: 4.7%

- Tied Covariance model: 2.0%