Exercise 4

Deadline: 29.11.2017, 10:00

This time, we will implement various generative models for an enlarged digits dataset, and use these models to generate new digits.

Regulations

Please hand-in your solution as a jupyter notebook `generative-models.ipynb`, accompanied with exported PDF `generative-models.pdf`. Zip all files into a single archive with naming convention (sorted alphabetically by last names)

`lastname1-firstname1_lastname2-firstname2_exercise04.zip`

or (if you work in a team of three)

`lastname1-firstname1_lastname2-firstname2_lastname3-firstname3_exercise04.zip`

and upload it to Moodle before the given deadline. We will give zero points if your zip-file does not conform to the naming convention.

Preliminaries

Generative modeling requires more training data than classification. Therefore, we created a new dataset containing 120000 digits of size 9×9. The data are provided in the HDF5 file `'digits.h5'`, which you can download from Moodle. To extract data from a HDF5 file, you need the Python module h5py (install via `'conda install h5py'`). HDF5 is powerful (multiple data arrays can be stored in a single file), user-friendly and increasingly accepted as the standard in data analysis. In our case, the file holds the array `'images'` containing the image data (size $120000 \times 9 \times 9$) and the vector `'labels'` with the corresponding labels. To access the data, you must first open the file and then refer to each array by its name, similar to a normal Python dictionary:

```
import h5py

f = h5py.File("/path/to/digits.h5")
images = f["images"].value
labels = f["labels"].value
f.close()

print(images.shape)
print(labels.shape)
```

1 Data Generation with Naive Bayes (8 points)

Choose one digit and visualize 8 images from the training set. Train a naive Bayes model (i.e. one histogram for each of the 81 pixels) with your function `fit_naive_bayes()` from exercise 3. Now create new digits by sampling from this distribution. In each feature dimension (i.e. for each pixel), perform two steps: (1) select a bin, (2) select a location (=final pixel value) between the bin's lower and upper bounds.

The probability for selecting bin l is defined by $P_l = \frac{N_l}{N}$. To sample from this discrete distribution, you should again apply "inverse transform sampling" (see exercise 1), which is especially simple in this case: First, compute the cumulative histogram Q_l of the P_l using the Python function `numpy.cumsum()`. Then, create a uniformly distributed random number $t \sim \text{uniform}(0, 1)$ and choose the *smallest* l with $Q_l \geq t$. The location within this bin is then sampled from a uniform distribution between the bin's bounds: $x \sim \text{uniform}(X_{\min,l}, X_{\max,l})$. Implement sampling by a function

```
sampled_image = sample_naive_bayes(histograms, binning)
```

You may play with the bin count to get better results, but don't expect too much from this method: The naive Bayes assumption (all features are independent given the class label) is clearly not fulfilled for the pixels in an image. Generate and plot 8 new digits.

2 Data Generation and Classification with Density Trees (26 points)

2.1 Train Tree by Error Minimization

We learned in the lecture that density trees can be trained by best-first recursive subdivision. Each leaf of the resulting tree represents a bin of an *adaptive* multi-dimensional histogram. To create the tree, split the most promising leaf according to some score function and iterate this process until a pre-specified total number of leaves is reached (e.g. $L = \tau\sqrt[3]{N}$ for some constant τ). Here, the score of a split is the expected reduction in approximation error:

$$\text{score} = \text{error}_{\text{before split}} - \text{error}_{\text{after split}} = -p_l^2 V_l + p_\lambda^2 V_\lambda + p_\rho^2 V_\rho$$

where V_l and $p_l = \frac{N_l}{N V_l}$ denote volume and probability density of leaf l before executing the proposed split, and V_λ, p_λ resp. V_ρ, p_ρ are the corresponding quantities for the resulting left and right child. For each leaf, the best split is selected by exhaustive search over all possible features and thresholds. (Since there are many candidate thresholds in a big dataset like ours, you may restrict the search to 10 equally spaced thresholds per feature.) In each iteration, only the globally best split is executed, and new scores are determined for the resulting children. A priority queue data structure (Python module `'heapq'`) is helpful for efficient bookkeeping of the current leaf ranking according to scores. You also have to keep track of each bin's bounding box to compute its volume and perform sampling later. Implement model creation by a function

```
density_tree = fit_density_tree1(features, bincount)
```

and implement a suitable data structure for your tree (look at `https://stackoverflow.com/questions/2358045/how-can-i-implement-a-tree-in-python-are-there-any-built-in-data-structures-in` for inspiration or use Python's `'networkx'` module).

Sampling from this model is again carried out in two steps: (1) select a bin, (2) select a location in the bin. To select a bin, you can take advantage of the tree structure: Each leaf has probability $P_l = p_l V_l = \frac{N_l}{N}$ (note: capital P_l is a probability, lower case p_l is a probability density). These probabilities can be propagated upwards to an interior node k by the simple formula:

$$P_k = P_\lambda + P_\rho$$

where P_λ and P_ρ are the probabilities of k's left and right child. The root node should have $P_{\text{root}} = 1$ (otherwise, there is a bug). To select the leaf, create a uniformly distributed random number $t \sim \text{uniform}(0, 1)$. In every non-leaf node (incuding the root), check if $t \leq P_\lambda$. If so, descent into the left subtree. Otherwise, update $t \leftarrow t - P_\lambda$ and descent into the right subtree. Upon arrival in a leaf, sample uniformely within its bounding box. Implement sampling via a function

```
sampled_image = sample_density_tree(density_tree)
```

Play around with τ to find a good bin count. Then generate and plot 8 new digits.

2.2 Train Tree by Maximizing Non-Uniformity

This task differs from the previous one only by the scoring criterion: Instead of maximizing the error gain, we seek the split whose children are maximally different, i.e. deviate the most from a

uniform distribution. This can be measured by the χ^2 criterion:

$$\text{score} = \chi^2 = N_l \left[\frac{(N_\lambda - E_\lambda)^2}{E_\lambda} + \frac{(N_\rho - E_\rho)^2}{E_\rho} \right]$$

where $E_\lambda = V_\lambda \frac{N_l}{V_l}$ and $E_\rho = V_\rho \frac{N_l}{V_l}$ are the expected counts if the children were in fact equal, i.e. their probabilities were simply proportional to the volume. This formula can be simplified into

$$\text{score} = \chi^2 = \frac{(N_\lambda V_l - N_l V_\lambda)^2}{V_\lambda (V_l - V_\lambda)}$$

Implement tree creation by a function

```
density_tree = fit_density_tree2(features, bincount)
```

and generate and plot 8 new digits as in task 2.1.

2.3 Tree-Based Classification

Select one of the two tree types and train models for the two digits "3" and "9". Then implement a generative classifier

```
predicted_labels = predict_density_trees(test_features, tree_for_3, tree_for_9)
```

This function is similar to `predict_naive_bayes()` from exercise 3, but the product of 1-dimensional histograms is replaced with tree-based probability estimation. Use the test data from HDF5 file `'digits_test.h5'` available on Moodle and report the confusion matrix. Does the density tree work better than the naive Bayes classifier?

3 Data Generation with a Multi-dimensional Gaussian (6 points)

Finally, we use the QDA model to fit a generative model. Simplify your function `fit_qda()` from exercise 2 so that it only receives the features of a single class:

```
mu, covmat = fit_gaussian(features)
```

where `mu` is the mean vector of size 81, and `covmat` is the 81×81 covariance matrix.

Use the function `numpy.random.multivariate_normal()` to generate 8 new digits and visualize them as in the other tasks.