Exercise 1

Deadline: 9.5.2023 16:00.
Ask questions in discord to #ask-your-tutor

In this exercise, you will implement and test linear classifiers.

Regulations

Please implement your solutions in form of *Jupyter notebooks* (`*.ipynb` files), which can mix executable code, figures and text in a single file. They can be created, edited, and executed in the web browser, in the stand-alone app JupyterLab, or in the cloud via Google Colab and similar services.

Create a Jupyter notebook `linear-classifiers.ipynb` for your solution and export the notebook to HTML as `linear-classifiers.html`. Zip all files into a single archive `ex01.zip` and upload this file to your assigned tutor on MaMPF before the given deadline.
Note: Each team creates only a single upload, and all team members must *join* it as described in the MaMPF documentation at `https://mampf.blog/zettelabgaben-fur-studierende/`.
Important: Make sure that your MaMPF name is the same as your name on Muesli. We now identify submissions purely from the MaMPF name. If we are unable to identify your name or if you forgot to join the submission you will not receive points for the exercise! Also note that joining a submission takes a while when you do it for the first time (later it will be just a single click), so don't wait until the last minute before the deadline.

Preliminaries

Create a Python environment containing the required packages using the `conda` package manager[1] by executing the following commands on the command line:

```
conda create --name ml_homework python   # create a virtual environment
conda activate ml_homework               # activate it (set paths etc.)
conda install scikit-learn matplotlib    # install packages into active environment
python                                   # run python
```

This brings up Python's interactive prompt. When everything got installed correctly, the following Python commands should load the respective modules without error:

```
import numpy        # matrices and multi-dimensional arrays, linear algebra
import sklearn      # machine learning
import matplotlib   # plotting
```

To install and run Jupyter, execute the following on the command line (note: this only works when environment `ml_homework` is active, see the `activate` command above)

```
conda install -c conda-forge jupyterlab    # install JupyterLab (only needed once)
jupyter lab                                # opens JupyterLab in your web browser
```

If your are not familiar with Python, Jupyter, and/or the numerics package `numpy`, check out our introductory video on MaMPF or work through these tutorials:

- `http://www.datacamp.com/community/tutorials/tutorial-jupyter-notebook`
- `https://www.physi.uni-heidelberg.de/Einrichtungen/AP/Python.php` (in German)
- `https://cs231n.github.io/python-numpy-tutorial/`

[1] You can download `conda` (specifically, its basic variant `miniconda`) from `https://conda.io/miniconda.html`. It is also part of the Anaconda software distribution, which you may already have installed.

1 Exploring the Data

Scikit-learn (usually abbreviated `sklearn`) provides a collection of standard datasets that are suitable for testing a classification algorithm (see `https://scikit-learn.org/stable/datasets.html` for a list of the available datasets and usage instructions). In this exercise, we want to recognize handwritten digits, which is a typical machine learning application. The dataset `digits` consists of 1797 small images with one digit per image.

Load the dataset from sklearn and extract the data:

```
from sklearn.datasets import load_digits

digits = load_digits()

print(digits.keys())

data          = digits["data"]
images        = digits["images"]
target        = digits["target"]
target_names  = digits["target_names"]
```

Note that `data` is a flattened (1-dimensional) version of `images`. What is the size of these images (the `numpy` attribute `shape` might come in handy)? Visualize one image of a **3** using the `imshow` function from `matplotlib.pyplot`, trying the two interpolation methods in the code:

```
import numpy as np
import matplotlib.pyplot as plt

img = ...

assert 2 == len(img.shape)

plt.figure()
plt.gray()
plt.imshow(img, interpolation="nearest") # also try interpolation="bicubic"
plt.show()
```

We will only work with digits "3" and "9" to get a two-class problem. Please filter the dataset such that only these two digits are left. Split this filtered dataset in a training and a test set (#train/#test = 3/2). Sklearn provides a convenient function to separate the data into a training and a test set:

```
from sklearn import model_selection

X_all = data
y_all = target

X_train, X_test, y_train, y_test =\
    model_selection.train_test_split(digits.data, digits.target,
                                     test_size = 0.4, random_state = 0)
```

2 Hand-crafted classifier

To better appreciate the achievements of machine learning, we will start with a hand-crafted classifier.

2.1 Feature construction

To facilitate visualization, you should construct a 2-dimensional feature space with any formula over the 64 original pixels you can come up with. You may, for example, choose two pixels that seem to have a big influence for the distinction between 3's and 9's. To identify suitable pixels, you may want to look at the average images for the two classes – pixels that tend to be bright in one

class and dark in the other are good candidates. You can also use some clever linear or non-linear combination of multiple pixels into 2 features, for example: $\tilde{f}_1 = 0.3f_{23} + 42\frac{f_{13}}{f_{64}}$ and $\tilde{f}_2 = f_{33} - f_{62}$. Of course, the quality of your features determines the achievable error and therefore is a limiting factor for the quality of your predictions. Your dimension reduction procedure should be callable through a function `features2d`:

```
features = features2d(x)
```

where `x` is a #instances $\times$ 64 matrix and `features` has shape #instances $\times$ 2.

2.2 Scatterplot

To decide whether your two features are suitable for the classification task, you should visually inspect the distribution of the instances in the new feature space by means of a *scatter plot*, which you can produce with the function `matplotlib.pyplot.scatter`. The plot axes are the two feature dimensions, and each training instance is placed at the appropriate location. To distinguish the classes, use two different markers. Check that the classes don't overlap too much and search for better features if they do. A good plot must be informative and easy to understand. Optimize your plot by choosing good values for the scatter function parameters (marker, color, size etc., see `https://matplotlib.org` for more options).

2.3 Decision rule

First define a very simple decision rule: Find the mean of the 2D feature vectors of each class in the training set and assign each test instance to the label of the nearest mean. The decision boundary of this method is the bisector between the two means. The signature of the function is supposed to look like this:

```
predicted_labels = nearest_mean(training_features, training_labels, test_features)
```

where `training_features` and `test_features` are the outputs of `features2d()` for training and test data respectively. The label vector should use label -1 to indicate digit "3" and label 1 to indicate digit "9" for this and all subsequent tasks. Compute and report the training and test errors of the classifier (i.e. the ratio $\frac{\#FP+\#FN}{N}$ on the respective dataset).

If your features are very informative, this classifier may already have good performance. Otherwise, you can define a more sophisticated decision boundary by picking good β and b in the formula $\widehat{y}_i = \text{sign}(x_i \cdot \beta + b)$. If this is still not satisfactory, improve your features. Again compute and report the test error of your formula.

2.4 Visualize the decision regions

An image of the decision regions can be quite helpful for analyzing classifier performance. The simplest way to do this is to create a grid (i.e. an image) where each pixel position represents a feature coordinate, and the pixel is colored with the corresponding predicted class label. Since you created the features on your own, you must find a sensible transformation (i.e translation and scaling) from feature coordinates to grid coordinates so that your image covers the interesting part of the feature domain. A 200×200 grid provides sufficient resolution. Plot this image to visualize the decision regions. Overlay the plot with two markers for the two class means and with a scatterplot of the test data, marked according to their ground truth labels as in task 2.2. Consult the matplotlib documentation to find out how several layers of information can be overlayed in the same plot. Plot the decision boundary of the two models from the previous subtask.

3 LDA

3.1 Implement LDA training

Implement the function:

`mu, covmat, p = fit_lda(training_features, training_labels)`

that accepts a $N \times D$ matrix `training_features` and a N-dimensional vector `training_labels`, where N is the total number of training instances used. Your code should work for arbitrary values of D, so that we can apply it to your features from task 2 as well as to the full vector of 64 pixels. Recall that the features x_j are the *rows* of matrix `training_features`. Use the algorithm that fits the cluster means and covariance matrix as explained in the lecture:

$$\mu_{-1} = \frac{1}{N_{-1}} \sum_{i:\, y_i^*=-1} x_i \qquad \mu_1 = \frac{1}{N_1} \sum_{i:\, y_i^*=1} x_i$$

$$\Sigma = \frac{1}{N} \left[\sum_{i:\, y_i^*=-1} (x_i - \mu_{-1})^T \cdot (x_i - \mu_{-1}) + \sum_{i:\, y_i^*=1} (x_i - \mu_1)^T \cdot (x_i - \mu_1) \right]$$

The output of `fit_lda` should be the $2 \times D$ matrix `mu` whose rows are the two class means, the $D \times D$ matrix `covmat` containing the covariance and the vector `p` containing the two priors. Apply the fit function to your feature data from task 2 and to the full pixel vector.
We shouldn't use all 64 pixel-dimensions though, as there are some 'dead' pixels i.e. they take the value of zero in every image. Filter them out first by calculating the variance of each pixel over all images (use `numpy.var()`) and masking out the pixels with a variance smaller than 0.001.

3.2 Implement LDA prediction

Now, using the output of `fit_lda()` to implement a function:

`predicted_labels = predict_lda(mu, covmat, p, test_features)`

which uses LDA to predict the labels (a M dimensional vector) for a $M \times D$ matrix `test_features` of test instances. Apply the function separately to your training and test data and compute the training and test error rates respectively.

You can use the decision rule $\widehat{y}_i = \text{sign}(x_i \cdot \beta + b)$ to make predictions where β and b are:

$$\beta = \Sigma^{-1}(\mu_1 - \mu_{-1})^T \qquad b = -\frac{1}{2}(\mu_1 + \mu_{-1}) \cdot \beta + log\frac{N_1}{N_{-1}}$$

You can perform the calculation of the inverse of Σ with `numpy.linalg.inv()`.

3.3 Visualization

Create a grid for the 2-dimensional LDA decision regions and plot it as in task 2.4. Overlay this plot with a scatter plot of the *training data*. Then add another overlay that visualizes the cluster shape of the two Gaussian distributions in terms of representative isocontours (i.e. ellipses). Check out the documentation for matplotlib's `contour` function to learn how to do this.

Furthermore, perform an eigenvalue/eigenvector decomposition of the covariance matrix using `numpy.linalg.eig()`. Recall that the eigenvectors indicate the directions of the principal cluster axes, and the eigenvalues' square roots are the corresponding standard deviations, i.e. they indicate the cluster radii along each axis. Add another overlay to your plot that draws these axes (centered at the cluster mean) in form of lines whose length equals the standard deviation.

Use this plot to comment on the quality of the LDA results separately on both the training and test data.

3.4 Quantitative performance evaluation

Now we want to get an idea of how well the LDA classifier generalizes to unseen data on the full pixel feature space. In order to estimate the test error, apply 10-fold cross validation to the LDA classifier.

Again, use only the digits "3" and "9", but perform cross validation on all the available data (i.e. ignore the test/train split from earlier in the exercise). Take advantage of sklearn's cross validation functions (`https://scikit-learn.org/stable/modules/cross_validation.html`) for splitting your data.

In addition, compare your results with the results from the official LDA implementation in the `sklearn` package.

4 SVM

Repeat all tasks of assignment 3 for the support vector machine (SVM), i.e. implement functions `fit_svm()` and `predict_smv()`, apply them to your data and visualize the decision boundary. Recall that the SVM training objective function consists of a regularization term and a data term, whose relative importance can be adjusted by the hyperparameter λ:

$$\mathcal{LOSS}(\beta, b) = \frac{1}{2}\beta^T \cdot \beta + \frac{\lambda}{N}\sum_{i=1}^{N} \text{ReLU}\big(1 - y_i^* \cdot (x_i \cdot \beta + b)\big)$$

The gradients of the objective w.r.t. the learnable parameters are:

$$\frac{\partial \mathcal{LOSS}(\beta, b)}{\partial \beta} = \beta + \frac{\lambda}{N} \sum_{i:\ y_i^*(x_i \cdot \beta + b) < 1} (-y_i^* \cdot x_i^T)$$

$$\frac{\partial \mathcal{LOSS}(\beta, b)}{\partial b} = \frac{\lambda}{N} \sum_{i:\ y_i^*(x_i \cdot \beta + b) < 1} (-y_i^*)$$

Use the gradient descent algorithm explained in the lecture to optimize these parameters, using initial gues $\beta^{(0)} \sim \mathcal{N}(0, 1)$ (i.e. a standard normal random vector created with `numpy.random.normal()`) and $b^{(0)} = 0$. With learning rate τ, the iterations become

$$\beta^{(t)} = \beta^{(t-1)} - \tau \cdot \frac{\partial \mathcal{LOSS}(\beta^{(t-1)}, b^{(t-1)})}{\partial \beta}$$

$$b^{(t)} = b^{(t-1)} - \tau \cdot \frac{\partial \mathcal{LOSS}(\beta^{(t-1)}, b^{(t-1)})}{\partial b}$$

For good convergence, it may be neccessary to use a *learning rate schedule*: When the training error stagnates, reduce the learning rate to $\tau/10$ and keep optimizing. This often leads to much better results than a fixed learning rate. Stop the optimization after a suitably chosen maximum number of steps T. Visualize in a diagram how the Loss and the training error change in the course of iteration.

Use cross-validation on the training set to find a good value for λ (i.e. pick several candidate values and then keep the one minimizing the cross-validation error). In addition, compare your results with the results from the official linear support vector classification in the `sklearn` package. How does the prediction quality change relative to LDA and the nearest mean classifier?