Exercise 1a

Deadline: 5.11.2021, 16:00.
Ask questions to #ask-your-tutor-felix

In this exercise, you will familiarize yourself with Python, Jupyter, Numpy and Matplotlib and verify some theoretical findings from the lecture via Monte Carlo simulation.

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 are created in the browser app JupyterLab or its legacy version Jupyter Notebook, or in the cloud via Google Colab or a similar service.

Create a Jupyter notebook `monte-carlo.ipynb` for your solution and export the notebook to HTML as `monte-carlo.html`. Zip all files into a single archive with naming convention (sorted alphabetically by first names)

`firstname1-lastname1_firstname2-lastname2_ex01a.zip`

or (if you work in a team of three)

`firstname1-lastname1_firstname2-lastname2_firstname3-lastname3_ex01a.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/` (setting up a team is a bit tedious the first time, but gets easier from the second assignment). Remember that you have to reach 50% of the homework points to be admitted to the final mini-research project.

Preliminaries (not graded)

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`

[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.

- `https://www.physi.uni-heidelberg.de/Einrichtungen/AP/Python.php` (in German)
- `https://cs231n.github.io/python-numpy-tutorial/`

1 Monte-Carlo Simulation

In the lecture, we considered the following toy problem: The feature variable $X \in [0, 1]$ is real-valued and 1-dimensional, and the response $Y \in \{0, 1\}$ is discrete with two classes. The prior probabilities and likelihoods are given by

$$\begin{aligned} p(Y = 0) = p(Y = 1) &= \tfrac{1}{2} \\ p(X = x|Y = 0) &= 2 - 2x \\ p(X = x|Y = 1) &= 2x \end{aligned}$$

We also derived theoretical error rates of the Bayes and nearest neighbor classifiers for this problem. Monte Carlo simulation is a powerful method to verify the correctness of theoretical results experimentally.

1.1 Data Creation and Visualization (7 points)

Since the given model is generative, one can create data using a random number generator. Specifically, one first samples an instance label Y according to the prior probability $p(Y)$, and then uses the corresponding likelihood (that is, $p(X|Y = 0)$ or $p(X|Y = 1)$, depending on which label we got in the first step) to sample the feature X. Since no predefined random generator is available for these likelihoods, we need to transform uniformly distributed samples from a standard random number generator to the desired distribution by means of "inverse transform sampling" (see `https://en.wikipedia.org/wiki/Inverse_transform_sampling`).

Work out the required transformation formulas for our likelihoods and show your derivation in a Markdown cell of your notebook. Then implement a function

```
def create_data(N):
    return ... # data X and labels Y
```

that returns the X-values and corresponding Y-labels for N data instances. Use the module `numpy.random` to generate random numbers. Check that the data have the correct distribution with `matplotlib` (see `https://matplotlib.org/gallery/statistics/hist.html` for a demo).

1.2 Classification by Thresholding (6 points)

In the lecture, we defined two classification rules deciding according to a threshold $x_t \in [0, 1]$:

- Rule A (threshold classifier): $\hat{Y} = f_A(X; x_t) = \begin{cases} 0 & \text{if } X < x_t \\ 1 & \text{if } X \geq x_t \end{cases}$
- Rule B (threshold anti-classifier): $\hat{Y} = f_B(X; x_t) = \begin{cases} 0 & \text{if } X \geq x_t \\ 1 & \text{if } X < x_t \end{cases}$

  (It always predicts the opposite of rule A – imagine that a programmer tried to implement rule A, but messed up by swapping the greater and less than signs.)

The corresponding error rates are:

$$p(\text{error}|A; x_t) = \frac{1}{4} + \left(x_t - \frac{1}{2}\right)^2$$

$$p(\text{error}|B; x_t) = \frac{3}{4} - \left(x_t - \frac{1}{2}\right)^2 = 1 - p(\text{error}|A; x_t).$$

Confirm experimentally for $x_t \in \{0.0, 0.2, 0.5, 0.6\}$ that the predicted error rates are correct. In particular, verify that the minimum overall error of 25% is achieved with rule A at threshold $x_t = 0.5$ (optimal Bayes classifier). Repeat each test with 10 different test datasets of the same size M and compute mean and standard deviation of the error. Use test set sizes $M \in \{10, 100, 1000, 10000\}$ and plot the results (i.e. mean error with corresponding error bars as a function of test set size). How does the error standard deviation decrease with increasing M?

1.3 Baseline Classifiers (2 points)

We now compare the above results to two rules that entirely ignore the features:

- Rule C (guessing): $\hat{Y} = f_C(X) = \begin{cases} 0 & \text{with probability } \frac{1}{2} \\ 1 & \text{otherwise} \end{cases}$
- Rule D (constant): $\hat{Y} = f_D(X) = 1$ (it always predicts class 1)

Both result in an error rate of 1/2. Confirm this like in the previous exercise: Plot the error and its standard deviation as a function of test set sizes $M \in \{10, 100, 1000, 10000\}$ for both new rules.

1.4 Nearest Neighbor Classification (6 points)

Implement the nearest neighbour classifier for our toy problem. That is, the predicted label of a test instance X_{test} is copied from its nearest training instance:

$$\begin{aligned} \hat{i} &= \arg\min_i |X_{\text{test}} - X_i| && \text{(find index of nearest training instance)} \\ \hat{Y}_{\text{test}} &= Y_{\hat{i}} && \text{(copy label of nearest training instance)} \end{aligned}$$

Create a function `toy_nearest_neighbor(Xtest, Xtrain, Ytrain)` predicting $\hat{Y}_{\text{text}}$ according to this rule, implemented such that it can handle training sets of arbitrary sizes.

First try this classifier with training sets of size $N = 2$, the smallest possible. Make sure that each training set contains one instance of either class (so your function `create_data()` from task 1.1 may need a slight modification). Determine the error rate of the nearest neighbour classifier on a sufficiently large test set. Repeat this with 100 different *training* sets (all of size $N = 2$) and compute the average error on the test set. Verify that this average error is around 35%, as derived in the lecture.

Repeat the experiment with training sets of size $N = 100$ and report the average error.