© Niko Krasowski and Ullrich Köthe
ullrich.koethe@iwr.uni-heidelberg.de

Exercise 1

Deadline: 7.11.2017, 15:00

In this exercise, you will implement the nearest-neighbor classifier and estimate its accuracy on unseen data via Monte Carlo simulation and cross-validation.

Regulations

Please implement your solutions in form of *jupyter notebooks* (`*.ipynb` files). Jupyter is a very elegant Python GUI, where executable code, figures and text can be embedded next to each other in nicely formatted notebook files. Installation instructions are given below, a tutorial can be found at `http://www.datacamp.com/community/tutorials/tutorial-jupyter-notebook`.

Hand-in `monte-carlo.ipynb` for problem 1 and `nearest-neighbor.ipynb` for problems 2 and 3. Also export your notebooks into PDF-files named `monte-carlo.pdf` and `nearest-neighbor.pdf`. Zip all files into a single archive (replace `name1` and `name2` with your names :-)

`name1_name2_exercise1.zip`

and upload this file to Moodle before the given deadline. Remember that you have to reach 50% of the homework points to be admitted to the final mini-research project.

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
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 work without error:

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

To install and run jupyter, execute on the command line

```
activate ml_homework     # activate ml_homework environment
conda install jupyter    # install jupyter (only needed once)
jupyter notebook         # opens jupyter in your web browser
```

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

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

[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.

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 these theoretical results experimentally.

1.1 Data Creation and Visualization (5 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 probabilities, and then uses the corresponding likelihood to sample the feature X. If no predefined random generator for the desired likelihood is available (as is the case here), uniformly distributed samples from a standard random number generator can be transformed to the desired distribution by means of "inverse transform sampling" (see `https://en.wikipedia.org/wiki/Inverse_transform_sampling`). Derive the required transformation formulas for our likelihoods. Then implement a function

```
features, labels = create_data(N)
```

that returns vectors containing the X-values and corresponding Y-labels for N data instances. Use module `numpy.random` to generate random numbers. Check that the data have the correct distribution with `numpy.histogram` and `matplotlib` (see `https://matplotlib.org/1.2.1/examples/pylab_examples/histogram_demo.html` for a demo).

1.2 Classification by Thresholding (4 points)

In the lecture, we defined two classification rules depending on a threshold $t \in [0, 1]$:

- Rule A: $\hat{Y} = f_A(X; t) = \begin{cases} 0 & \text{if } X \leq t \\ 1 & \text{else} \end{cases}$
- Rule B: $\hat{Y} = f_B(X; t) = \begin{cases} 1 & \text{if } X \leq t \\ 0 & \text{else} \end{cases}$

and claimed that their average error rates are

$$p(\text{error}|A, t) = \frac{1}{4} + \left(t - \frac{1}{2}\right)^2$$
$$p(\text{error}|B, t) = \frac{3}{4} - \left(t - \frac{1}{2}\right)^2$$

Check experimentally for $t \in \{0.2, 0.5, 0.6\}$ that these formulas predict the error rates correctly. In particular, verify that the minimum error of 25% is achieved when rule A is applied with threshold $t = 0.5$ (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 $M \in \{10, 100, 1000, 10000\}$. How does the error standard deviation decrease with increasing M?

1.3 Nearest Neighbor Classification (4 points)

Implement the nearest neighbor classifier for our toy problem. Create a training data set of size $N = 2$ (one instance of either class) and determine the error rate of the nearest neighbor classifier on a sufficiently large test set. Repeat this with 100 different training sets (all of size $N = 2$) and compute the average error. Verify that this average error is 35%.

2 Nearest Neighbor Classification on Real Data

2.1 Exploring the Data (2 points)

Scikit-learn (usually abbreviated `sklearn`) provides a collection of standard datasets that are suitable for testing a classification algorithm (see `http://scikit-learn.org/stable/datasets/` 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"]

import numpy as np
print np.dtype(data)
```

Note that `data` is a flattened version of the images. Explore the data further by testing its dimensionality (the function `numpy.shape()` might come in handy). Visualize one image of a '3'. One possibility to visualize an image is the `imshow` function from `matplotlib.pyplot`:

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

img = ...

assert 2 == np.size(np.shape(img))

plt.figure()
plt.gray()
plt.imshow(img, interpolation="nearest")
plt.show()
```

Moreover, sklearn provides a convenient function to separate the data into a training and a test set.

```
from sklearn import cross_validation

X_all = data
y_all = target

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

2.2 Distance function computation using loops (3 points)

A naive implementation of the nearest neighbor classifier uses loops to determine the required minimum distances. Implement this approach in a python function `dist_loop(training, test)`, which computes the Euclidean distance between all instances in the training and test set (in the feature space). The input should be the $N \times D$ and $M \times D$ training and test matrices with D pixels per image and N respectively M instances in the training and test set. The output should be a $N \times M$ distance matrix. For the calculation of the Euclidean distance you might want to use `numpy.square()`, `numpy.sum()` and `numpy.sqrt()`.

2.3 Distance function computation using vectorization (6 points)

Since loops are rather slow in python, and we will need efficient code later in the semester, write a second python function `dist_vec(training, test)` for computing the distance function which relies on vectorization and does not use `for` loops. Consult `https://www.safaribooksonline.com/library/view/python-for-data/9781449323592/ch04.html` and `https://softwareengineering.stackexchange.com/questions/254475/how-do-i-move-away-from-the-for-loop-school-of-thought` for information on how to do this. Verify that the new function returns the same distances as the loop-based version. Now compare the run times of the two implementations using jupyter's `%timeit` command (the vectorized version should be significantly faster).

2.4 Implement the nearest neighbor classifier (4 points)

Implement a nearest neighbor classifier and use it to distinguish the digit '1' from the digit '3'. To do so, filter the training and test data accordingly and use your function from subproblem 2.2 or 2.3 to compute distances. Use the test data to estimate the classifier's error rate for the two classes. Repeat the same experiment for digits '1' and '7'.

2.5 Generalize to k-nearest neighbors (4 points)

Extend your classifier to take the opinion of more than one neighbor into account: return the majority vote from the k-nearest neighbors of each test instance. Vary the value for k (try the values 1, 3, 5, 9, 17 and 33) and compute the error rate for the classification of ones against sevens. Describe the dependency of the classification performance on k.

3 Cross-validation (8 points)

In subproblem 2.4, we measured the performance of the nearest neighbor classifier on a predefined test set. To be able to do this, we had to put aside test data and thus reduced the size of the training set. This may lead to increased error because some relevant training instances might not end up in the training set. Another way to estimate whether the trained classifier is able to generalize to unseen data is cross-validation. Use cross-validation to compare your nearest neighbor classifier from 2.4 with `sklearn.neighbors.KNeighborsClassifier()`. The latter is trained with the `fit()` function and classifies new data via the `predict()` function (see `http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html` for more details).

Write a function that splits the given annotated data in n *folds* (parts of roughly equal size). Use each of the n subparts as test set once and the remaining parts as corresponding training set. Return the mean error rate as well as its standard deviation over the n repetitions. Cross-validate the two nearest neighbor classifiers on the full digits dataset for $n \in \{2, 5, 10\}$.