Exercise 1

Deadline: 7.11.2018, 15:00

In this exercise, you will implement the k-nearest-neighbor classifier and estimate its accuracy on unseen test data and via cross-validation. Moreover, you learn how to speed-up Python code using "vectorization".

Regulations

Provide your comments for last week's homework in the files `monte-carlo-commented.ipynb` and `monte-carlo-commented.html`. Solutions for this week's tasks shall be handed in as `nearest-neighbor.ipynb` and exported to `nearest-neighbor.html`. Zip all files into a single archive with naming convention (sorted alphabetically by last names):

`lastname1-firstname1_lastname2-firstname2_exercise01b.zip`

or (if you work in a team of three)

`lastname1-firstname1_lastname2-firstname2_lastname3-firstname3_exercise01b.zip`

and upload this file to Moodle before the given deadline.

1 Comment on your solution to exercise 1

Study the sample solution `monte-carlo-solution.html` provided on Moodle and use it to comment on your own solution to this exercise. Specifically, copy your original notebook `monte-carlo.ipynb` to `monte-carlo-commented.ipynb` and insert comments as markdown cells starting with

```
<span style="color:green;font-weight:bold">Comment</span>
```

so that we can clearly distinguish your comments from the other cell types. Insert comment cells at the appropriate places according to the following rules:

- If your code is incorrect, identify the bugs and make brief suggestions for possible fixes (don't include a full corrected solution).
- If your solution is slow, identify inefficient code sections (e.g. Python loops) and suggest possible improvements.
- If your code is correct, but differs from the sample solution, briefly explain why your solution as a valid alternative and where either solution is more elegant.
- If your code is essentially equal to the sample solution, explicitly say so.

Export the commented notebook to `monte-carlo-commented.html` and hand in both files.

Note: If you fail to hand in `monte-carlo-commented.html`, we will deduct 50% of the points from your solution to exercise 1.

2 Asymptotic error of the nearest neighbor classifier for the toy example from exercise 1 (5 points)

Since many people had difficulties answering this question as an in-class quiz, we repeat it as a homework. We showed in the lecture, that the nearest-neighbor classifier's asymptotic local error (i.e. for a specific feature vector X) is given by the Gini impurity

$$p_\infty(\text{error}|X) = 1 - \sum_{k=1}^{C} p^*(Y = k \,|\, X)^2$$

where $p^*(Y = k \mid X)$ is the Bayesian posterior for class k. The asymptotic total error is then obtained from the local error by taking the expectation over all X

$$p_\infty(\text{error}) = \mathbb{E}_{X\sim p(X)}\left[p_\infty(\text{error} \mid X)\right] = \int p_\infty(\text{error} \mid X)\, p(X)\, dX$$

Derive the total error of the nearest neighbor classifier for the toy example from exercise 1 by means of these equations. Use this theoretical result to interpret your experimental findings in exercise 1.

3 Nearest Neighbor Classification on Real Data

3.1 Exploring the Data (3 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"]

print(data.dtype)
```

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`:

```
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()
```

Moreover, 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)
```

3.2 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()`.

3.3 Distance function computation using vectorization (8 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).

Note: It is absolutely critical that you understand vectorization, because the code for subsequent homeworks will otherwise be too slow for meaningful experimentation. We will therefore grade this task very strictly.

3.4 Implement the k-nearest neighbor classifier (6 points)

Revise your code from the previous homework to implement a k-nearest neighbor classifier. It should work for arbitrary k (number of neighbors to include in the majority vote), N (training set size) and D (number of features). Use your classifier to distinguish the digit **3** from the digit **9**. To do so, filter out these digits from the training and test sets and use your function from subproblems 3.2 or 3.3 to compute distances. Vary the value for k (try the values 1, 3, 5, 9, 17 and 33) and compute the error rates. Describe the dependency of the classification performance on k.

4 Cross-validation (8 points)

In subproblem 3.4, 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 (see `https://en.wikipedia.org/wiki/Cross-validation_(statistics)`).

Write a function

```
X_folds, y_folds = split_folds(data, target, L)
```

to randomly split the given data and labels into L *folds* (parts of roughly equal size). The numpy-functions `random.permutation()` and `array_split()` may be useful here. In each subsequent cross-validation iteration, you use one of the L subparts as test set and the remaining ones as corresponding training set.

Use the same splits to evaluate your implementation from subtask 3.4 and the pre-defined solution in `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).

Return the mean error rate as well as its standard deviation over the L repetitions. Cross-validate your k-nearest neighbor classifier with $k = 1$ and a suitable bigger k, as well as its `sklearn` counterparts, on the full digits dataset for $L \in \{2, 5, 10\}$. Compare the algorithms' performance. How do the results depend on L?