University Master's Degree in Intelligent Systems
SJK003 - MACHINE LEARNING

Laboratory 1
Machine Learning with Python

Objectives:

- Familiarization with scikit-learn.
- Experimenting with synthetic data sets.
- Reliable estimate of the classification error (from repeated measurements).
- Uncertainty of the classification error estimate (based on range, standard deviation).
- Familiarization with the concept of hyperparameter.
- Introduction to data visualisation.

Deliverable / Evaluation:

Report with results and analysis from the Exercise 3 (identified as Deliverable). The report should be sent through the virtual classroom before the deadline set.

Exercise 1: Warming up with a simple example of data classification and visualization.

Context. According to the official site, "Scikit-learn is an open-source machine learning library that supports supervised and unsupervised learning. It also provides various tools for model fitting, data preprocessing, model selection, model evaluation, and many other utilities". This first exercise is a simple introduction to a few basic operations.

Roadmap:

- Create a dataset of synthetic 2D samples.
- Divide the dataset into two subsets for training and test purposes, respectively.
- Visualise both subsets into a single plot, but differentiating their samples.
- Create a 1-NN classifier and evaluate its performance in the synthetic dataset.
- Play with the parameters of the data distribution; visualize and discuss results.

Instructions:

a. Create a Google Colab notebook: `File -> New notebook`.

b. Cell no. 1 (imports). Start by adding the following import statements:

```
from sklearn.datasets import make_circles
from sklearn.model_selection import train_test_split
```

c. Cell no. 2 (dataset). Create the dataset through the next sentence:

```
X, y = make_circles(n_samples=200, noise=0.2, factor=0.5)
```

It draws two concentric circles, a larger one containing a smaller one. The parameter 'noise' makes the circles imperfect (Gaussian noise added to the data), while 'factor' determines the distance between them (0 -> maximum separation, ~1 -> both overlap).

'X' is an ndarray of shape (200, 2) containing the generated samples; 'y', an ndarray of shape (200,) containing the corresponding labels (class membership) in [0,1].

- Check that X and y sizes match our expectations (numpy.ndarray.shape).

To take away. The use of synthetic data is particularly useful for the evaluation of models in pattern recognition. The researcher determines, and knows, precisely data complexity, which allows explaining, and fully understanding, the behaviour of the model.

d. Cell no. 3 (data partition). Add the following sentence to split data for training and testing:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

The generated data is divided into two subsets for training (70%) and testing (30%) purposes. The first ones remain in (X_train, y_train); the second ones, in (X_test, y_test).

- Check that X_train, X_test, y_train and y_test sizes match our expectations.

To take away. The amount of data available to create and evaluate models is usually scarce or insufficient. Thus, a majority is usually dedicated to training, while the complementary minority, to test.

e. Cell no. 4 (data visualisation). Try the following script to visualize the training and test data in the same plot. Are you able to distinguish them? Can you explain how it works?

Add these import statements into the cell no. 1:

```
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
```

Add the main script into the cell no. 4:

```
fig, ax = plt.subplots()
ax.set_title('Input data')
x_min, x_max = X[:,0].min() - 0.5, X[:,0].max() + 0.5
y_min, y_max = X[:,1].min() - 0.5, X[:,1].max() + 0.5
ax.set_xlim(x_min, x_max)
ax.set_ylim(y_min, y_max)
ax.set_xlabel('x1')
ax.set_ylabel('x2')
colors = ["green", "orange", "gold", "blue", "purple", "red"]
cmap = ListedColormap([colors[0], colors[1]])
ax.scatter(X_train[:,0], X_train[:,1], c=y_train, cmap=cmap)
ax.scatter(X_test[:,0], X_test[:,1], c=y_test, cmap=cmap, alpha=0.5)
plt.show()
```

- Run the current code (Ctrl+F9) repeatedly and observe the graphical result. Do you notice randomness? What are the sources of randomness?

In our experiments the random nature of the outcome is kept. However, both methods 'make_circles' and 'train_test_split' include a 'random_state' parameter, which allows for reproducible output on repeated method calls.

To take away. A rigorous scientific experimentation generally requires randomness when creating and selecting data, although sometimes it is necessary to reproduce a particular state to analyze some model behaviour in detail.

f. Cell no. 5 (classification). Next sentences create a 1-NN classifier from the training subset, and evaluate its performance on the test subset.

Add this import statement into the cell no. 1:

```
from sklearn.neighbors import KNeighborsClassifier
```

Add the main script into the cell no. 5:

```
classifier = KNeighborsClassifier(1)
classifier.fit(X_train, y_train)
score = classifier.score(X_test, y_test)
print(score)
```

The statements create a generic 1-NN classifier, fit it to the task of interest, score its performance on the test set, and print the scoring result.

- What do you think will happen if you score the performance of the classifier on the training set itself? Check it.
- Run the current code (Ctrl+F9) repeatedly and observe the scoring result. Can you explain the observations? Can we trust an individual result? Why?

Analysis. An individual result depends on the data randomly generated by the statistical law of the generating process, generally unknown, and on the arbitrary data partitioning performed in the current run. These random events could have been very favourable for the purposes of the task or, conversely, very unfavourable. That is, the result of a single run could have been very optimistic or pessimistic, but in no case, realistic. Neither of the two cases would reflect the most probable scenario, nor would it help us to establish an objective measure of the performance of the classifier in the task of interest.

How can we get a reliable measure of the performance of a classifier? How can we measure the confidence of the scoring result?

g. Try other combinations of the 'noise' (e.g., between 0 and 0.5) and the 'factor' (e.g., between 0 and 0.99) in the 'make_circles' function, and see how it affects the complexity of the data distributions and the classification result. Try to explain each case. Keep in mind that the random classifier will be roughly right in half of the decisions (~0.5), like when you flip a coin, a reference that helps to judge the score in the different scenarios.

h. Before leaving this exercise, return to the original parameter values: noise=0.2, factor=0.5.

Exercise 2: Getting a reliable classification score and uncertainty measures.

Context. As experienced in the previous exercise, a single classification score is uncertain. Our main sources of uncertainties are data instability (insufficient data affected by noise), and classifier biases and limitations (most classifiers suffer from both). Next you will learn how to calculate more confident classification scores, and how to measure uncertainty of the result.

Both indices tell us ***to what extent we can trust*** *the results of the model (classifier).*

Roadmap:

- Calculate a reliable classification score (the average score over a number of repetitions).
- Calculate an uncertainty measure of individual scores (standard deviation).
- Estimate an uncertainty measure of the average score (standard error of the mean – SEM).
- Calculate an uncertainty measure of the average score (standard deviation of a mean distr).
- How many individual measurements are necessary to calculate a confident score?

Instructions:

a. Calculating a reliable classification score.

Adapt the code from Exercise 1 to repeat the experiment n times, including all sources of randomness, and to calculate the average classification score. It is recommended to work on a copy of the code from exercise 1 (`File -> Save a copy in Drive`).

Test the code for different values of n.

b. Run the code 5 times for $n = 1$, $n = 10$, $n = 100$, $n = 1000$, and fill in the table below:

	run 1	run 2	run 3	run 4	run 5
$n = 1$					
$n = 10$					
$n = 100$					
$n = 1000$					

Check the variability of the 4 series (rows) of 5 means (columns). What do you observe?

Analysis. The average of multiple measurements is expected to be more stable (less uncertain) than individual measurements (row 1): the greater the number of repetitions, the greater the confidence of the average.

c. Calculating an uncertainty measure of individual scores via the standard deviation.

In the code above, insert the calculation of the standard deviation of the n scores. Consider using `numpy.std(y, ddof=1)`, or implement the corresponding equation:

$\boldsymbol{\sigma_y} = \sqrt{\frac{\sum_{i=1}^{n}(y_i-\bar{y})^2}{n-1}}$, where $\bar{y}$ denotes the mean of the scores y_i.

Run the code for $n = 10$, $n = 100$, $n = 1000$, and check results. Do you notice any important difference between the three results?

To take away. *The standard deviation is a measure of the variability of a sample, often used as a spread or uncertainty measure. It is calculated as the square root of an (almost) average squared deviation. Since both the numerator and the denominator depend on the number of measurements ($\boldsymbol{n}$), the result does not necessarily depend on $\boldsymbol{n}$. Besides, the standard deviation is expressed in the same original units; so, it is easy to interpret.*

d. Estimate an uncertainty measure of the average score.

In the previous item, an uncertainty measure (standard deviation) of individual scores was calculated. Considering that the mean of multiple measurements is a more reliable score estimate, would it be possible to estimate the uncertainty of the mean?

In the code above, insert the formulation of the *Standard Error of the Mean* (SEM), what can be considered a measure of the precision of the mean:

$$SEM = \frac{\sigma_y}{\sqrt{n}}$$

The SEM is also denoted as $\boldsymbol{\sigma_{\bar{y}}}$, to represent the standard deviation of the mean $\bar{y}$. For its implementation in Python, you will need `math.sqrt()`.

Analysis. *Suppose we run the current code $\boldsymbol{m}$ times, each iteration repeating the current experiment $\boldsymbol{n}$ times (including sources of uncertainty). As a result, we would have $\boldsymbol{m}$ average scores (more reliable than individual scores). The standard deviation of the series of $\boldsymbol{m}$ means, each calculated from $\boldsymbol{n}$ repetitions, is expected to approximate the estimate obtained from the SEM (which only requires a single run of the code).*

Exercise 3: Model tuning with HyperOpt (Deliverable).

Context. In the k-NN model, k is a hyperparameter. In AI, a hyperparameter is a parameter that determines the capacity of a model to solve a task or controls the learning process. Virtually all classification models have at least one hyperparameter. They allow the model to adapt to the complexity of a problem. Hyperparameter tuning is one of the most important (and often neglected) stages in an intelligent system training process.

HyperOpt is an open-source python package to find model hyperparameters which optimize a user-defined objective function, within previously defined domains for each hyperparameter. Let's look at the following example: finding the best value of k for the Iris data set.

Instructions:

a. Study the following code in detail. Copy it to a Python environment and run it.

```
from sklearn import datasets
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.neighbors import KNeighborsClassifier
from hyperopt import fmin, tpe, hp, STATUS_OK, space_eval
from sklearn.metrics import accuracy_score

iris = datasets.load_iris()
X, y = iris.data, iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y)

def hyperopt_train_test(params):
  clf = KNeighborsClassifier(n_neighbors=params['k'])
  return cross_val_score(clf, X_train, y_train).mean()

def f(params):
  acc = hyperopt_train_test(params)
  return {'loss': -acc, 'status': STATUS_OK}

hp_domains = {'k': hp.choice('k', range(1,25))}
best_hp = fmin(f, hp_domains, algo=tpe.suggest, max_evals=100)

# evaluate the best k-NN rule
best_k = space_eval(hp_domains, best_hp)['k']
clf = KNeighborsClassifier(n_neighbors=best_k)
clf.fit(X_train, y_train)
y_train_pred = clf.predict(X_train)
y_test_pred = clf.predict(X_test)
train_acc_score = accuracy_score(y_train, y_train_pred)
test_acc_score = accuracy_score(y_test, y_test_pred)
print('Train accuracy (best k={}): {:.3f}'.format(best_k, train_acc_score))
print('Test accuracy (best k={}): {:.3f}'.format(best_k, test_acc_score))
```

What is the goal of the code? How does the code work?

Analysis. *The above code covers a single classification result from a single partition of the data set into training and test subsets. And, as you already know, a single result is a matter of chance, it could be noisy, and it's generally unreliable. So, let's iterate!*

b. Truth is in the mean (from as many repetitions as possible).

Create a copy of the previous code. Modify the code to repeat the above process 100 times, and obtain the mean, standard deviation, and SEM of the series of 100 accuracy results on test data. As a matter of curiosity, also calculate…

- the mean, the standard deviation, and the SEM of the set of 100 accuracy results on training data (for comparison purposes with test results).
- the mean and standard deviation of the series of 100 best *k* values (note that you might get a different better *k* in each iteration).
- the histogram of the distribution of the 100 best *k* values.

Perform a critical analysis.

c. What if we need to tune two hyperparameters?

Many learning algorithms take advantage of having standardized features, that is, normalized features with mean 0 and standard deviation 1. Therefore, another design decision for our learning solution could be whether to use the features in their original domains or standardize them. Add to the original code the necessary statements to include this analysis in HyperOpt. You will need the following tools:

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
clf = KNeighborsClassifier(n_neighbors=k)
clf = make_pipeline(StandardScaler(), clf)
```