*Faculty of Mechanical Engineering and Robotics*

*Department of Robotics and Mechatronics*

# Python for Machine Learning and Data Science

*Course for Mechatronic Engineering*

## Instruction 4:

# **Regression** and **classification**

**You will learn:** basic implementation of the regression and classification models using scikit-learn.

**Additional materials:**

- Course lecture 3&4 [*obligatory*] http://galaxy.agh.edu.pl/~zdw/Materials/Python/LectureNotes/
- Matlab to Python handout http://galaxy.agh.edu.pl/~zdw/Materials/Python/Matlab%20-%20python%20handout.pdf

**Learning outcomes supported by this instruction:**
IMA1A_W07, IMA1A_U01, IMA1A_U05, IMA1A_U07, IMA1A_U14, IMA1A_K08

**Course supervisor:**
Ziemowit Dworakowski, zdw@agh.edu.pl

**Instruction author:**
Adam Machynia, machynia@agh.edu.pl

## Introduction

During this laboratory, you will learn the basic implementation and use of regression and classification models in Python. You will use linear regression and a multi-layer perceptron (MLP) for regression. For classification purposes, you will use logistic regression, support vector machines, and again the multi-layer perceptron. You are encouraged to also test weighted linear regression, polynomial regression, k-nearest neighbors classification, and decision trees.

At the very beginning, you should load the *wine quality* dataset and split it into subsets as you did during the previous laboratory.

## Regression

Regression aims to predict the target value based on certain features. We will use several models for this purpose, all of which are available in the scikit-learn library. To start, let's use a very simple linear regression. After importing *LinearRegression* from *sklearn.linear_model*, we create a model simply by calling *LinearRegression()*. To train our model, we use the *fit()* function, which takes as arguments the training data and their corresponding labels. Once the model is trained, we can use the *predict()* function to make predictions. Importing, creating, training models, and making predictions for training and validation sets are shown in the listing below. As you can see, with *scikit-learn*, it is very straightforward.

```python
from sklearn.linear_model import LinearRegression

reg = LinearRegression()
reg.fit(x_train, y_train)

y_pred_train = reg.predict(x_train)
y_pred = reg.predict(x_val)
```

For the evaluation of the obtained predictions, we will use the root mean squared error (RMSE), calculated according to the following equation:

$$RMSE = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(\hat{y}_n - y_n)^2},$$

where $\hat{y}_n$ is the predicted value, $y_n$ is the real value, and N is the number of samples. RMSE can be calculated from scratch as follows.

```python
rmse = np.sqrt(np.mean((y_pred_train - y_train)**2))
```

One can also use the *mean_squared_error* function from *sklearn.metrics.*

**Task 4.1:** Split the data into training, validation, and test sets. Use the quality as the target value – y and remember to drop it from the rest of the data. Implement linear regression and calculate the RMSE for the training and validation sets.

**Task 4.2:** A simple linear model can be easily extended to a weighted linear model by using an additional parameter, sample_weight. Check how to use this parameter and implement weighted linear regression.

Now, let's consider a more sophisticated model: a neural network. To use it, we need to import *MLPRegressor* from *sklearn.neural_network.*

```
from sklearn.neural_network import MLPRegressor

# simply with almost default parameters
reg = MLPRegressor(random_state=1, max_iter=500)
reg.fit(x_train, y_train)
y_pred_train = reg.predict(x_train)
y_pred = reg.predict(x_val)

# and then try some customization, for example:
reg = MLPRegressor(hidden_layer_sizes=(10, 5), random_state=1, max_iter=1000,
solver='lbfgs')
```

As neural networks are more complex, we should consider some parameters for the customization of our model.

- *hidden_layer_sizes:* A tuple where each number denotes the number of neurons in a specific hidden layer. For example, (15, 10, 5) indicates 15 neurons in the first hidden layer, 10 in the second, and 5 in the third.
- *random_state:* Allows for reproducibility.
- *max_iter:* The maximum number of epochs if convergence is not achieved beforehand.
- *Solver:* Sets the algorithm for weight optimization.

These are just selected parameters; you can find descriptions of all of them in the scikit-learn documentation.

**Task 4.3:** Implement neural network regressor. Make predictions and calculate RMSE for training and validation sets. Repeat this step while experimenting with your ideas for model parameter setup.

**Task 4.4:** Compare regression performance (with one selected model) using all features versus a selected subset of features. Start by choosing two features, then four, and suggest your concepts. Remember that the correlation matrix is useful for feature selection in regression.

As you can see, while we have created the model, training and making predictions remain consistent across different models. This consistency helps in ensuring that the code is organized and facilitates the examination of various models. However, not all regression methods have direct implementations. Still, they can often be implemented easily with scikit-learn. If you would like to learn how to cleverly implement polynomial regression using a linear regression model, take a look here: https://scikit-learn.org/stable/modules/linear_model.html#polynomial-regression-extending-linear-models-with-basis-functions. Generally, handling nonlinear models as linear models operating on nonlinear functions can be beneficial, as it maintains fast performance while solving more complex problems.

**Task 4.5:** Based on provided link, implement polynomial regression.

## Classification

As classification aims to assign a class label to each sample, we will modify our task. Suppose we want to predict whether a wine is "good" or not. Let's assume that a wine is considered good if its quality score is higher than 5. Therefore, we can prepare labels for training as follows. Don't forget to apply the same process to other subsets or perform thresholding in advance.

```
y_train = data_train['quality'] > 5.5
```

Once the labels are arranged, we can choose our first classification model. Let's start with the logistic regression classifier. Similar to the regression case, after creating the model, we use the *fit()* function for training and the *predict()* function for prediction.

```
from sklearn.linear_model import LogisticRegression

clf = LogisticRegression()
clf.fit(x_train, y_train)

y_pred_train = clf.predict(x_train)
y_pred = clf.predict(x_val)
```

For now, we'll use only the accuracy metric to evaluate the performance of the models. Generally, this is insufficient, and we will discuss this issue in Lab 6.

```
from sklearn.metrics import accuracy_score

acc_train = accuracy_score(y_train, y_pred_train)
acc_val = accuracy_score(y_val, y_pred)
```

**Task 4.6:** Implement the logistic regression model, train it, and make predictions on training and validation sets. Check its accuracy.

Hint: If an error occurs, read the message carefully; it should indicate possible solutions.

**Task 4.7:** Use the logistic regression model, but only with selected features. Compare the accuracy of two scenarios by selecting four different features in each:

1) *'alcohol', 'volatile acidity', 'total sulfur dioxide', 'sulphates',*
2) *'pH', 'free sulfur dioxide', 'residual sugar', 'fixed acidity'.*

Compare the accuracy between the two cases.

Other models can also be created and used in a similar way. Each model has specific parameters that should be tuned, such as the kernel for SVM, the number of neighbors for KNN, or the maximum depth for decision trees.

```
from sklearn import svm
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn import tree

clf = svm.SVC(kernel='rbf')
clf = MLPClassifier()
clf = KNeighborsClassifier(n_neighbors=5)
clf = tree.DecisionTreeClassifier(max_depth=3)
```

**Task 4.8:** Implement the SVM model. Try different kernels using the *kernel* parameter.

**Task 4.9:** Implement multi-layer perceptron. Set your own configuration for the parameters. Some of these were described in the section covering regression.

**Task 4.10:** Implement other classifiers, such as KNN, decision tree, or others.

**Task 4.11:** You've probably achieved around 70-75% accuracy so far. Try improving the performance of the models by tuning their parameters.

**Task 4.12:** Select the model (with tuned parameters) that performs best for you. Now, train it on both the training and validation sets and check its final performance on the test subset.

## Pipelines

The *Scikit-learn* module implements *pipelines*, which facilitate creating consistent processing routines for different datasets, such as training and validation sets. This may include tasks like outlier removal or feature scaling. The pipeline takes a list of tuples as an argument, where each tuple consists of a name and a transformer. The name is a string, and the transformer must implement the *fit()* and *transform()* methods. The final step, which serves as the estimator, only requires the *fit()* method.

Let's look at an example. Here, we put standardization and the MLP into a single pipeline so that the data input into the *clf.fit()* method is first standardized before the neural network is trained.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

clf = Pipeline([("scaler", StandardScaler()),
("mlp", MLPClassifier(max_iter=500))])

clf.fit(x_train, y_train)
```

**Task 4.13:** Implement a pipeline with two steps: standardization and a neural network. Choose a reasonable size for the network and compare its performance with and without feature scaling, using the stochastic gradient descent ('sgd') algorithm for learning.

**Task 4.14:** Examine how different solvers and the maximum number of iterations affect the results. For three solvers ('lbfgs', 'sgd', 'adam'), try at least three different iteration counts that trigger warnings about failure to converge, as well as one that satisfies the required number of iterations. Do not change other parameters of the network. Collect the accuracy score for each case in a table for discussion.

## Additional tasks

**Task 4.15:** Implement linear regression and neural network regressor for pH feature as a target.

**Task 4.16:** Implement KNN and decision tree classifiers. Adjust their parameters.