TD N° 2 : Kernels for ML

- Exercise 1: Graph kernels for graphs classification -

In this exercise we will have access to a dataset of graphs $(G_1, y_1), \cdots, (G_n, y_n)$ where y_i are classes and we want to learn a function that takes a graph as input and output a label/class. This a problem of classification on the space of graphs. We will use the framework of graph kernels and the `GraKel`, `networkx` libraries. Make sure you have installed `networkx` version $\geq$ 3.2.1 and `GraKel` $\geq$ 0.1.8. You can find the data here https://chrsmrrs.github.io/datasets/docs/datasets/ and a description of it here https://ls11-www.cs.tu-dortmund.de/staff/morris/graphkerneldatasets.

(i) First download the MUTAG dataset in here https://chrsmrrs.github.io/datasets/docs/datasets/ in .zip format. You can load the data afterwards using the following code.

```
import numpy as np #we will need the following libraries
from grakel.datasets import fetch_dataset
from grakel.datasets.base import read_data
from grakel.kernels import WeisfeilerLehman, VertexHistogram
import zipfile

name = "MUTAG"
verbose = True
path = './data' # where data is stored
with zipfile.ZipFile(path+"/" + str(name) + '.zip', "r") as zip_ref:
    if verbose:
        print("Extracting dataset ", str(name) + "..")
    zip_ref.extractall()

if verbose:
    print("Parsing dataset ", str(name) + "..")

dataset = read_data(name,
                    with_classes=True,
                    prefer_attr_nodes=False,
                    prefer_attr_edges=False,
                    produce_labels_nodes=False,
                    is_symmetric=False,
                    as_graphs=True
                    )
G = dataset.data
y = dataset.target
```

(ii) Do a quick inspection of the dataset and labels: what do the data represent? how many classes? what can you conclude? What do you need to consider?

(iii) Using the following code based on the `networkx` library plot different training points

```python
import networkx as nx
import matplotlib.pyplot as plt
m = 12
nx_g = nx.from_numpy_array(G[m].get_adjacency_matrix())
i = 0
for _, v in G[m].node_labels.items():
    nx.set_node_attributes(nx_g, {i: {'label': v}})
    i += 1
cols = {0: 'r', 1: 'b', 2: 'g'}
pos = nx.layout.kamada_kawai_layout(nx_g)
nx.draw_networkx(nx_g,
                 with_labels=True,
                 labels=nx.get_node_attributes(nx_g, 'label'),
                 node_color=[cols[v]
                             for k, v in nx.get_node_attributes(nx_g, 'label').items()]
                 )
```

(iv) With `scikit-learn` splits the dataset into a training and a test set (you can use `from sklearn.model_selection import train_test_split`).

(v) The following code consider a Weisfeiler-Lehman graph kernel and train/test the model on the splits defined before. What is the kernel considered here? What do the parameters correspond to? What is the accuracy on the test set? Compare it to a simple naive baseline (you may use `from sklearn.dummy import DummyClassifier`).

```python
from sklearn.svm import SVC
wl_kernel = WeisfeilerLehman(n_iter=5, normalize=True,
                             base_graph_kernel=VertexHistogram)
K_train = wl_kernel.fit_transform(G_train)
K_test = wl_kernel.transform(G_test)
clf = SVC(kernel='precomputed')
clf.fit(K_train, y_train)
y_pred = clf.predict(K_test)
```

(vi) Implement a cross validation (CV) procedure with `StratifiedKFold` and compute the average CV score. Compare with the previous error.

(vii) Compare the Weisfeiler-Lehman kernel with the graphlet kernel (`grakel.kernels.GraphletSampling`), the shortest-path kernel (`grakel.kernels.ShortestPath`) and the vertex histogram kernel (`grakel.kernels.VertexHistogram`).

(viii) Do the same procedure by changing some hyperparameters (e.g. the normalization of the kernels).

(ix) Based on this comparison can we say that one model is better than the others ?

(x) Implement a nested CV procedure (explanations on the board). You can use the following class and code.

(xi) Do the same procedure with another dataset (e.g. PTC MR).

```python
class GK_classifier():
    def __init__(self, C=1, n_iter=5, normalize=True):
        self.C=C
        self.n_iter=n_iter
        self.normalize=normalize
        wl_kernel = WeisfeilerLehman(n_iter=self.n_iter,
                                    normalize=self.normalize,
                                    base_graph_kernel=VertexHistogram)
        self.graphkernel = wl_kernel
        self.svc=SVC(kernel='precomputed', C=self.C)

    def fit(self, X, y=None):
        # to complete

    def predict(self,X):
        # to complete
```

```python
def frozendict(d: dict):
    keys = sorted(d.keys())
    return tuple((k, d[k]) for k in keys)

def do_cv(X, y, model, n_splits=5, random_state=0, shuffle=True):
    skf = StratifiedKFold(
        n_splits=n_splits, random_state=random_state, shuffle=shuffle)
    errors_ = []
    for i, (train_index, test_index) in enumerate(skf.split(X, y)):
        X_train, X_test = [X[i] for i in train_index], [X[i]
                                                        for i in test_index]
        y_train, y_test = y[train_index], y[test_index]
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        errors_.append(accuracy_score(y_test, y_pred))
    return errors_

def do_nested_cv(X, y, class_model, param_grid, inner_cv=5, outer_cv=5,
                 higher_the_better=True, random_state=0, shuffle=True):
    # estimate the generalization error of the whole model
    outer_skf = StratifiedKFold(
        n_splits=outer_cv, random_state=random_state, shuffle=shuffle)
    errors = []
    chosen_parameters = []
    for i, (train_index, test_index) in enumerate(outer_skf.split(X, y)):
        X_train, X_test = [X[i] for i in train_index], [X[i]
                                                        for i in test_index]
        y_train, y_test = y[train_index], y[test_index]

        res_of_param = {}
        # loop for selecting the best hyperparameter
        for param in param_grid:
            instantiated_model = class_model(**param)
            res = # to fill
            res_of_param[frozendict(param)] = np.mean(res)

        if higher_the_better:
            # the higher accuracy the better
            best_param = max(res_of_param, key=res_of_param.get)
        else:
            best_param = min(res_of_param, key=res_of_param.get)
        chosen_parameters.append(best_param)
        best_model = class_model(**{elt[0]: elt[1] for elt in best_param})

        best_model.fit(X_train, y_train)
        y_pred = best_model.predict(X_test)
        score = accuracy_score(y_test, y_pred)
        errors.append(score)
        print('--- Outer index {} done ---'.format(i))
        print('Best hyperparam chosen are {}'.format(best_param))
        print('Score is {:.2f}'.format(score))

    return errors, chosen_parameters
```