Problem4

December 5, 2021

1 Problem 4

[1]:
```
import pandas as pd
import grakel
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
from sklearn.model_selection import KFold
from scipy.special import binom
```

1.1 Data Loading

First we will read in the files and convert them to numpy, in order to then build the graphs. The arrays that only have one column anyways, will be flattened, to avoid lists of length 1.

[2]:
```
edges = pd.read_csv("MUTAG/MUTAG_A.txt", header=None).to_numpy()
graph_indicator = pd.read_csv("MUTAG/MUTAG_graph_indicator.txt", header=None).
 ↪to_numpy().flatten()
labels = pd.read_csv("MUTAG/MUTAG_graph_labels.txt", header=None).to_numpy().
 ↪flatten()
node_labels = pd.read_csv("MUTAG/MUTAG_node_labels.txt", header=None).
 ↪to_numpy().flatten()
edges[:5]
```

[2]:
```
array([[2, 1],
       [1, 2],
       [3, 2],
       [2, 3],
       [4, 3]])
```

Next we will calculate, what nodes belong to what graph.

[3]:
```
node_lists = []
for graph in np.unique(graph_indicator):
    node_lists.append(np.asarray(graph_indicator == graph).nonzero()[0] + 1)
print(node_lists[0])
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17]
```

Next we will find the edges of each graph. This is not the most efficient way of doing it, since the list is sorted, but it is sufficiently fast.

[4]:
```
edges_lists = []
for node_list in node_lists:
    es = []
    for v, w in edges:
        if (v in node_list) and (w in node_list):
            es.append((v,w))
    edges_lists.append(es)
print(edges_lists[0])
```

```
[(2, 1), (1, 2), (3, 2), (2, 3), (4, 3), (3, 4), (5, 4), (4, 5), (6, 5), (5, 6),
(6, 1), (1, 6), (7, 5), (5, 7), (8, 7), (7, 8), (9, 8), (8, 9), (10, 9), (9,
10), (10, 4), (4, 10), (11, 10), (10, 11), (12, 11), (11, 12), (13, 12), (12,
13), (14, 13), (13, 14), (14, 9), (9, 14), (15, 13), (13, 15), (16, 15), (15,
16), (17, 15), (15, 17)]
```

Now we can create the `grakel.Graphs`.

[5]:
```
graphs = []
for es, ns in zip(edges_lists, node_lists):
    n_labels = {i: node_labels[i-1] for i in ns}
    g = grakel.Graph(es, node_labels=n_labels)
    graphs.append(g)
print(graphs[0])
print(len(graphs))
```

```
<grakel.graph.Graph object at 0x7f054c0d41d0>
188
```

Sanity check: do we have as many labels as graphs?

[6]:
```
print(len(graphs) == len(labels))
```

```
True
```

1.2 Classification and Cross validation for Graphlet Kernel

We can here use the crossvalidation algorithm from the last exercise (after slightly modifying it to take all Cs as an argument to avoid recomputing the kernel matrix unnecessarily). We will parametrize the function, such that it takes a function as one argument, which generates the kernel. This way we can easily reuse this function in the next task.

[7]:
```
def do_cv(Cs, X, Y, kernel):
    kf = KFold(n_splits=10, shuffle=True, random_state=42) # for reproducability
    X = np.asarray(X)
    Y = np.asarray(Y)

    # initialize output dict
```

```
    scores = {C: {
                "accuracy": [],
                "results": [],
                "truths": [],
                "classifiers": []
            } for C in Cs}

    # Crossvalidation-loop
    for train_indices, test_indices in kf.split(X):
        # generate training and test data
        Xtrain = X[train_indices]
        Ytrain = Y[train_indices]
        Xtest = X[test_indices]
        Ytest = Y[test_indices]

        # define kernel
        graphlet_kernel = kernel()
        # calculate kernels
        Ktrain = graphlet_kernel.fit_transform(Xtrain)
        Ktest = graphlet_kernel.transform(Xtest)

        for C in Cs:
            # initialize SV;
            e = svm.SVC(
                kernel="precomputed",
                C=C
            )

            # fit and evaluate
            e.fit(Ktrain, Ytrain)
            acc = e.score(Ktest, Ytest)
            scores[C]["accuracy"].append(acc)

            # save some more metadata in case we need it later
            results = e.decision_function(Ktest)
            scores[C]["results"].append(results)
            scores[C]["truths"].append(Ytest)

            scores[C]["classifiers"].append(e)

    for C in Cs:
        # calculate mean accuracy, save and print it
        scores[C]["accuracy"] = np.array(scores[C]["accuracy"]).mean()
        acc = scores[C]["accuracy"]
        print(f"Model (C={C}): accuracy = {acc:2.2%}")

    return scores
```

Define Cs

[8]:
```
Cs_graphlet = np.logspace(-4, 1, 6)
print(Cs_graphlet)
```

```
[1.e-04 1.e-03 1.e-02 1.e-01 1.e+00 1.e+01]
```

Define the function that will generate the graphlet kernel as described in the task

[9]:
```
def generate_grahplet_kernel():
    return grakel.GraphletSampling(
                random_state=42, # for reproducability
                k=3,
                sampling={
                    "n_samples": 1000
                }
            )
```

[10]:
```
scores_graphlet = do_cv(Cs_graphlet, graphs, labels, generate_grahplet_kernel)
```

```
Model (C=0.0001): accuracy = 81.93%
Model (C=0.001): accuracy = 81.93%
Model (C=0.01): accuracy = 82.49%
Model (C=0.1): accuracy = 82.49%
Model (C=1.0): accuracy = 82.49%
Model (C=10.0): accuracy = 82.49%
```

Find best accuracy:

[11]:
```
def print_best_accuracy(scores, Cs):
    accuracies = []
    for C in Cs:
        accuracies.append((C, scores[C]["accuracy"]))
    best_C, best_acc = max(accuracies, key=lambda x: x[1])
    print(f"The best accuracy is {best_acc:2.2%} which was achieved by␣
 ↪C={best_C}")
```

[12]:
```
print_best_accuracy(scores_graphlet, Cs_graphlet)
```

```
The best accuracy is 82.49% which was achieved by C=0.01
```

1.2.1 How many samples would we need?

For this we can use Theorem 6 from the script (graph kernels, page 20). For this we need to know, how many size 3 graphlets are there are in the largest graph (since the samples must be enough for all graphs, including the biggest graph). We can easily find the number of nodes in the largest graph:

[13]:
```
n = len(max(node_lists, key=len))
print(f"Number of nodes in the largest graph: n = {n}")
```

```
Number of nodes in the largest graph: n = 28
```

By page 18 of the same scipt, we know, that a graph with n nodes has $\begin{pmatrix} n \\ k \end{pmatrix}$ graphlets of size k. Since we are using $k = 3$ we get: $a = \begin{pmatrix} n \\ k \end{pmatrix} = \begin{pmatrix} 28 \\ 3 \end{pmatrix}$. What that is as a number, we can also calculate:

[14]:
```
k = 3
a = binom(n, k)
print(a)
```

```
3276.0
```

So we have $a = 3276$ graphlets in the biggest graph.

Now we can use Theorem 6:

We have $\mathcal{A} = \{1, ..., a\}$ as the set of graphlets in the largest graph. We then need:

$$m = \left\lceil \frac{2(\log 2 \cdot a + \log\left(\frac{1}{\delta}\right)}{\epsilon^2} \right\rceil$$

samples, in order to restrict the chance of more than ϵ deviation to a probability of δ.

Using the values from the task we get: $\epsilon = 0.05$ and $\delta = 1 - 0.9 = 0.1$ which leads us to:

[15]:
```
epsilon = 0.05
delta = 0.1

m = np.ceil(
    (2 * (np.log(2) * a + np.log(1/delta))) /
    (epsilon**2)
)
print(f"m = {m}")
```

```
m = 1818443.0
```

So we would need to sample 1818443 times in order to have a distribution of less, than 0.05 with probability 0.9. This number might be a stong overestimation since a lot of the graphlets might be isomorhic (see the paper cited in the scipt *Nino Shervashidze et al., Efficient graphlet kernels for large graph comparison* Chapter 3.2).

But as seen from the paper it is enough, to use the number of *different* graphlets of size k. For $k = 3$ there are only 4 possible graphlets:

```
- everything connected,
- everything unconnected,
- one edge (where that is does not matter),
- two edges (i.e. one edge missing, again, where that is does not matter).
```

So now we have $a = 4$ and can calculate a much tighter bound:

[16]:
```
a = 4
m = np.ceil(
    (2 * (np.log(2) * a + np.log(1/delta))) /
    (epsilon**2)
)
print(f"m = {m}")
```

```
m = 4061.0
```

Now we have a bound of $m = 4064$ samples which is much more reasonable.

1.3 Classification and Cross validation for Weisfeiler-Lehman subtree kernel

Given the way we have parametrized the funcion, we can now very simply do that second task:

[17]:
```
Cs_wlsk = np.logspace(-4, 2, 7)
print(Cs_wlsk)
```

```
[1.e-04 1.e-03 1.e-02 1.e-01 1.e+00 1.e+01 1.e+02]
```

Define the new kernel generating function

[18]:
```
# https://ysig.github.io/GraKeL/0.1a8/auto_examples/weisfeiler_lehman_subtree.
↪html?highlight=subtree
def generate_weisfeiler_lehman_subtree_kernel():
    return grakel.WeisfeilerLehman(
            n_iter=4,
            base_graph_kernel=grakel.kernels.VertexHistogram,
            normalize=True
        )
```

[19]:
```
scores_wlsk = do_cv(Cs_wlsk, graphs, labels,␣
↪generate_weisfeiler_lehman_subtree_kernel)
```

```
Model (C=0.0001): accuracy = 66.49%
Model (C=0.001): accuracy = 66.49%
Model (C=0.01): accuracy = 66.49%
Model (C=0.1): accuracy = 66.49%
Model (C=1.0): accuracy = 77.72%
Model (C=10.0): accuracy = 86.75%
Model (C=100.0): accuracy = 83.01%
```

[20]:
```
print_best_accuracy(scores_wlsk, scores_wlsk)
```

```
The best accuracy is 86.75% which was achieved by C=10.0
```