LINGI2355: Mining Patterns in Data
Project 3: Classifying Graphs

Due 14th of May, 2021, 23:55

1 Tasks

In this project, you will use the *gSpan* graph mining algorithm to construct pattern-based classifiers for molecular data. You will have to use this implementation to train different models in order to accurately classify molecules between two classes. You are provided with

- an implementation of the *gSpan* algorithm in Python;
- a dataset of molecules, subdivided in two categories (positive and negative).

The project consists of a number of different tasks. You are strongly encouraged to finish these tasks in order, as the first tasks prepare for the later parts of the project.

All submissions for this project will have to be implemented in Python.

Phase 1: finding subgraphs This first task aims at familiarizing yourself with the gSpan implementation and the template provided. You will have to use the *gSpan* algorithm to find the top k most confident (on the positive class) frequent subgraphs, i.e. the subgraphs with the k highest positive confidence then frequency values. If several patterns have the same confidence, you should select the one with the highest frequency. Frequency is measured here over the sum of supports over both the positive and negative examples. If several pattern have both the same confidence and the same support, you should take all of them. Hence, you could find more than k patterns. Note that subgraphs with a same confidence but different frequencies count towards different entries in the topk subgraphs. Except minor variations, this task setting is similar to that of the previous project.

Phase 2: training a basic model In this second task, you will use the *Scikit-learn* library to train a model based on the top-k patterns found using *gSpan* with the algorithm in the previous phase. Here, we will use decision trees such as implemented in Scikit-learn. You will use cross-validation to evaluate the quality of your model. In order to do this, you will need to split your data in to a training set and a test set. *Scikit-learn* allows to tune the search tree using multiple parameters. Please leave these parameters to their default value and set the random seed of the tree to 1: `classifier = tree.DecisionTreeClassifier(random_state=1)`.

Phase 3: sequential covering for rule learning In this third task, you will implement a sequential covering algorithm to learn a rule list. Your sequential covering algorithm will have to use the top-k mining algorithm k times in order to repeatedly find a pattern with maximal confidence in either the positive or the negative class (you should thus call *gSpan* with a value of 1 for k). After having found a new pattern, do not forget to remove the transactions that are classified by it from your training data for the next iterations. The final list of rules can be used as a classifier. You can find more information about this task in the dedicated section.

Phase 4: another classifier For this last task, you will have to train and evaluate a model of your choice. You can also change the mining task in order to obtain better patterns. You are free to use the resources provided for the project or implement other algorithms. Your model has to be able to classify new graphs in one of the two classes. You will have to analyse the accuracy of your model. You are encouraged to use cross-validation to do so. Be wary of overfitting. The aim is to have a model as accurate as possible.

The next sections detail specific parts of the assignment.

Data

The data used for this project was obtained from this paper: `https://pubs.acs.org/doi/abs/10.1021/ci0503715`. The dataset contains molecules represented as graphs. It is separated into two files named `molecules.pos` and `molecules.neg` which correspond to the positive and negative class. The molecules are labelled based on their *mutagenicity* i.e. their potential for inducing mutations in DNA. The aim is to discover substructures in these molecules which are mutagens and based on this information, being able to predict whether or not a new molecule will cause mutations.

This molecular data is encoded as graphs using the following format: Each graph starts with a line

```
t # <graph id>
```

which indicates the start of a new graph. It is then followed by one line for each vertex of the graph under the following format:

```
v <vertex id> <vertex label>
```

Then, one line for each edge with the following format:

```
e <vertex 1 id> <vertex 2 id> <edge label>
```

The end of the graph is indicated by the line starting the next graph or by the end of the file. The end of the file is either a blank line or the line

```
t # -1
```

The template provided contains utility classes to read the dataset files, and provide these datasets as input to *gSpan*.

Top-k Mining with *gSpan*

The given implementation of *gSpan* solves the following problem: it finds *all* frequent subgraphs in a positive set of training graphs, and determines in which negative examples and test examples the patterns occur.

To familiarize yourself with the implementation, you need to use this implementation such that it finds subgraphs that are frequent in the complete set of positive and negative examples, and have high confidence with respect to the positive examples.

The implementation of *gSpan* is provided in the class `gSpan` (file `gspan.py`). An example of how to execute gSpan on a specific task is provided in the file `main.py` in the `example1` method. *gSpan* is initialized with a task object that defines the task to be executed. In the template, the task is defined in the `FrequentPositiveGraphs` class, which should inherit from the `PatternGraphs` class.

This class defines three important methods: the `__init__` method used for initializing objects, and the `store` and the `prune` method which will be called during the search. The `store` method is called by gSpan each time it has found a pattern. In the given example, all patterns are stored. Your first task is to modify this function such that it only stores the set of $k-$best patterns. The data structure used to store the patterns is a list. You might want to change this for another, more efficient, data structure in a topk mining context.

The `prune` function is called by *gSpan* to determine whether or not to prune a part of the search tree. In the example that is provided, a subtree is pruned if the support on the positive training examples is too small. For the first task you will need to change this function such that it prunes if the confidence or the support on all training examples is too low.

In order to use the `PatternGraphs` class for a specific task, you should create a new class that extends this class and implements the `store` and `prune` methods.

DFS codes

The implementation of *gSpan* uses dfs codes to represent the subgraphs found. The codes consist in a list of tuples, each corresponding to an edge, under the following format:

```
(frm=<first vertex>, to=<second vertex>, vevlb=(<first vertex label>, <edge label>,
<second vertex label>))
```

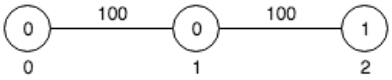

Figure 1: Example of subgraph

Where `frm` indicates the vertex from which the edge starts, `to` indicates the vertex at which the edge arrives and `vevlb` indicates the labels of the two vertices and the edge. The label values can be -1 if the value has already been given in a previous tuple. For example the subgraph in figure 1 will be displayed as:

```
[(frm=0, to=1, vevlb=('0', '100', '0')),(frm=1, to=2, vevlb=(-1, '100', '1'))]
```

Feature Set

In order for your top-k itemsets to be used to train a model, you need to build a representation of the data in terms of the patterns present in each of the transactions. The following illustrates this representation:

	Pattern 1	Pattern 2	$\cdots$
Transaction 1	1	0	$\cdots$
Transaction 2	1	1	$\cdots$
$\vdots$	$\vdots$	$\vdots$	

Here the bits indicate the presence or absence of the patterns. This format is required by the models of *Scikit-learn* to train them. An example of how to build such a matrix is provided in the is provided in the file `main.py`.

SciKit-learn

Scikit-learn is a python package providing multiple tools for data mining and machine learning tasks. In this project, it will be used to train classification models. Its documentation can be found at the following address: `https://scikit-learn.org/stable/index.html`. Note that *scikit-learn* requires `NumPy`, `SciPy` and `matplotlib` to work. It can be installed independently or as part of Anaconda (`https://www.anaconda.com/`).

Sequential Covering for Learning Rule Lists

We will use the sequential covering approach to identify a rule-based classifier, consisting of a set of rules.

The aim is to develop an algorithm that repeatedly adds the pattern that obtains the highest confidence on the remaining examples; hence, the expected approach is iterative: it searches repeatedly for patterns and does not post-process a set of patterns that was found earlier. The transactions that contain the pattern found are then removed and the next iteration searches a new pattern on the remaining ones.

To develop this algorithm, you can use the implementation that you used in the first task. The resulting algorithm is expected to be applied k times (with a topk size of 1) to find a new pattern to add to the set. If several patterns with the same confidence and frequency are found, you should take the lowest in the lexicographical order. After having k patterns in your pattern set, you should define a default class that will be assigned if none of the patterns in the pattern set is able to classify a transaction. This default class will be the one the most present in the remaining transactions or the positive class if both classes are equally present or if there is no transaction remaining.

The resulting list of rules can be used as a classifier without using SciKitLearn.

Cross-Validation

Cross-validated accuracy can be used as an evaluation criterion for the accuracy of a model. k-fold cross-validation assumes that the data is split into k folds; repeatedly one of these folds is removed, while the learning algorithm is applied on the remaining data. The resulting model is evaluated on the data that was left out.

In a pattern set mining context this means that the pattern mining algorithm needs to be applied k times to find a set of patterns. For each of these iterations, a model needs to be constructed from the patterns, and the resulting model needs to be applied on the remaining examples to evaluate the quality of the model.

You are encouraged to use this method in your performance evaluation. We limit the number of folds to 4 to reduce the run time of your experiments.

Implementing cross-validation requires that the occurrences of graph patterns are also determined in the test data. While we could implement a separate algorithm for subgraph isomorphism to accomplish this, we recommend a different approach in this project: for every pattern that gSpan generates, we immediately determine the occurrence in the test data as well. This allows to avoid creating a separate algorithm for evaluating subgraph isomorphism.

Note however that for a reliable evaluation of a machine learning algorithm, it is absolutely necessary that occurrences calculates on the test data are **not** used while building the pattern-based model; it should only be used for evaluating the quality of the model.

An example of k-fold cross-validation is provided in the second example of the template.

Provided template

For this project, you are provided with a python implementation of the *gSpan* algorithm that has been modified to work in a supervised context. The template is available in the following github repository: `https://github.com/cftmthomas/LINGI2364-Project3_template`. This templates includes a package named `gspan_mining` which contains the *gSpan* implementation as well as the classes it needs and a separated file called `main.py` which contains two examples of usage of the template. **The contents of the package `gspan_mining` should not be changed**. This package will be provided on INGInious, you should thus not include it in your submission.

The package contains three files:

- `graph.py` which contains the classes used to represent edges, vertices and graphs.
- `graphdatabase.py` which contains the class used to read graph database files and manipulate them in the search.
- `gspan.py` which contains the implementation of the gspan algorithm and the classes used to represent dfs codes.

The `main.py` file contains two examples of how to use gSpan along with the task classes used. The first example defines a simple search on the positive class with a minimum support constraint. The second example extends the first example by using the result of the search to train a basic naive bayes gaussian classifier model using *Scikit-learn* and performs a k-fold cross validation.

2 Directives

- The project will be done by groups of at most two students.
- The project will be done in Python. You should not use any additional package besides *Scikit-learn*, `NumPy`, `SciPy` and their dependencies.
- Here are the inputs and outputs for each task:

  Phase 1: finding subgraphs Your program will receive four parameters: the paths to the positive and negative database files, a parameter k which is the size of top$k$ and a parameter f which indicates the minimum frequency (defined as the sum of the positive and negative supports) that your subgraphs should have. All the subgraphs found by your program should be displayed on the standard output, one per line following this format:

  ```
  <dfs code> <confidence> <total support>
  ```

  Use the first example provided in the template as base of your program.

  Phase 2: training a basic model Your program will receive five parameters: the paths to the positive and negative database files, a parameter k which is the size of top$k$, a parameter f which is the minimum frequency and a parameter indicating the number of folds to use for the k-fold cross-validation. For each fold, you should display the following: A single line `fold <fold number>` where fold number (1 to number of folds) indicates which fold is excluded. All the subgraphs found by the program using the same format as in the first task (one per line). A line with the prediction results of the test set using

the model as a list of integers (1 for the positive class, -1 for the negative class). A line indicating the accuracy of the prediction on the test set under the following format: `accuracy: <accuracy value>`. Each fold should be separated by a blank line. Use the second example provided in the template as base of your program.

Phase 3: sequential covering for rule learning Your program will receive the same parameters than the second task and use the same output format.

Phase 4: another classifier Your program will receive three parameters indicating the path to the positive and negative files and the number of folds to use in the cross-validation. It should use the same output format as the other classification tasks (it is not needed to display the confidence and frequency for the patterns found.)

- As with the previous projects your code will be graded using INGInious. The INGInious tasks will be made available later and will contain more details about the submission modalities. **Please make sure that your program works by testing it on your machine before uploading it**. INGInious is a grading tool not a debugger and its resources are limited. Make sure that your program works without bugs and provides the expected output for multiple test cases. You will be provided with several outputs to compare with.
- Your report must not exceed 4 pages. It has to contain a short description of your implementations. You have to explain and justify your implementation choices. The report must also contain an experimental comparison in which you discuss the results obtained using the different techniques. This last part should be the main focus of your report, particularly the model used in the last task.

  Optionally, you can briefly discuss the difficulties that you encountered during the project.

  Your report must contain the number of your group as well as the names and NOMAs of each member. It should be written in correct English. Be precise and concise. Using tables, schemas or graphics to illustrate is a good idea.
- You will be graded based on several criteria:
  - The correctness and performances of your implementations (12/20).
  - The quality and relevance of your report, justifications and performance analysis (8/20).
- The deadline for this project is **December 21th at 23:55**. Your submission should be uploaded on INGInious according to the modalities specified for each task.

3 Tips

- Make full use of the time allocated. Do not start the project just before the deadline!
- Do not forget to comment your code.
- Test your code before submitting it.
- In case of difficulties, you can always use the moodle forum to ask questions.
- Plagiarism is forbidden and will be checked against! Do not share code between groups. If you use online resources, cite them.