Mining Patterns in Data

Classifying Graphs

Gilles Peiffer (23421600)
Université catholique de Louvain
gilles.peiffer@student.uclouvain.be

Liliya Semerikova (64811600)
Université catholique de Louvain
liliya.semerikova@student.uclouvain.be

ABSTRACT

The following paper contains some explanations concerning the third assignment for the "Mining Patterns in Data" class. Various tasks had to be implemented, with all of them being related to graph mining and classification using gSpan. Several interesting insights are given, as well as some implementation details for the various pieces of code. We also give some information about the performance tradeoffs with various models.

1. INTRODUCTION

Frequent substructure mining has been an important data mining problem for some time, as labeled graphs can be used to model complicated substructure patterns in data.

For the purpose of this paper, we consider the general task of assigning a label to elements of a dataset of molecules, using existing machine learning algorithms and techniques from pattern-based classification.

All algorithms and experiments in this paper were implemented in Python, using scikit-learn [5].

2. TASKS

Four tasks have to completed as part of this assignment.

2.1 Finding subgraphs

The first task is concerned with finding the top-k most confident frequent subgraphs in the positive dataset.

2.2 Training a basic model

In this second task, the result of the previous task is used to build a pattern-based classifier in order to predict the class of unknown molecules.

2.3 Sequential covering for rule learning

The third task builds a classification model using sequential covering, i.e. by successively removing classified transactions from the database. It uses the top-k mining algorithm in order to decide which pattern to prune the database with (by successively calling it with $k = 1$).

2.4 Another classifier

The final task is similar to the second one, as it is about finding a better classifier based on a graph mining algorithm.

3. ALGORITHMS & IMPLEMENTATIONS

Various algorithms and implementations were used to complete the tasks outlined in Section 2.

3.1 gSpan

The gSpan algorithm was proposed by Yan & Han [6] in order to efficiently mine frequent substructures without candidate generation. It significantly outperforms other algorithms, like the apriori-based FSG [4]. The gSpan algorithm was provided as a Python module by the teaching staff for the class.

3.2 Top-k mining with gSpan

In order to adapt the gSpan algorithm to mine only the top-k frequent patterns in the graph database, two functions needed to be implemented in `FrequentPositiveGraphs`:

- `store(self, dfs_code, gid_subsets)` is called whenever the gSpan implementation wants to store a pattern. To only store the top-k patterns, we thus have to determine whether the pattern that is being scheduled for storing has a higher confidence/frequency than the "worst" pattern currently being stored.

  To store the list of current patterns, we decided to use a sorted list, with the sortedness property being maintained thanks to the functions in Python's `bisect` module. Another possible implementation choice, the priority queue, does not have a much better performance, as was demonstrated in our timing benchmarks in the report for the second assignment.

- `prune(self, gid_subsets)` is called by gSpan when it tries to prune part of the search tree. In order to work with top-k pattern mining, it should return that the subtree can be pruned if either the confidence or the support is too low.

3.3 Sequential covering

Sequential covering is an iterative approach to pattern-based classification. Successively, the best pattern in the data is recognized and the transactions containing it are removed from the database. It uses the top-k version of gSpan, with $k = 1$, to generate this best pattern, hence the "best" pattern is the one with the highest confidence in our case.[1] Once there are a certain predefined number of patterns in the current pattern set, the unclassified transactions, if any, are assigned a default label.

[1]Other possibilities include using another criterion, such as χ^2 or the information gain.

3.4 Classification

Once the top-k best patterns have been found, and after some minor modifications to make them work with the scikit-learn library, it is possible to use standard machine learning classifiers to assign a class label to unknown transactions, based on a model built using the best patterns. In order to obtain the best possible model (according to some predefined performance metric, such as the classification accuracy), most classifiers have hyper-parameters which can be tuned. Other ways to improve performance include various means of preprocessing data, or combining predictions from multiple classifiers.

3.4.1 Decision trees

A decision tree algorithm is fairly straightforward, in that it builds a tree one can traverse according to the presence of certain patterns in the transaction being classified. A decision tree algorithm works by selecting, at each of its nodes, the best pattern to classify the data on.

3.4.2 Neural networks

Neural networks are another type of machine learning tool that can potentially be used to classify data. Due to the constraints on the software we are allowed to use, the scikit-learn implementation of multi-layer perceptrons (`MLPClassifier`) is the simplest way to implement a neural network for the classification task at hand. According to the universal approximation theorem as proposed in [3], any function can be represented by such an MLP, under certain mild assumptions, hence this type of classifier is in theory a good candidate for the task at hand.

3.4.3 Support vector machines

Another possible model for the classification task, that can easily be implemented with scikit-learn, is the support vector machine. SVMs try to maximize the classification margin between the positive and negative samples, possibly in some projected space (using a kernel transformation). The loss function which one tries to minimize can be adapted with a regularization constant, in order to control how badly misclassifications are punished.

3.4.4 k-nearest neighbors

k-nearest neighbor classification classifies test data according to the labels of its k-nearest neighbors in the training set, and the distance to each of them. k-NN is a lazy algorithm, hence it is efficient in situations where there is a large amount of data.

3.4.5 Data preprocessing

Data preprocessing is an important step in machine learning. Depending on the algorithm being used, several techniques can be considered to improve performance, such as principal component analysis or statistical selection of most important features of a model. This type of preprocessing serves two purposes: it can help to avoid overfitting, as well as reduce computation time significantly if the problem dimensionality is large.

4. PERFORMANCE EVALUATION & MODEL COMPARISON

4.1 Finding graphs

In the first task, no models are built: it simply consists in manipulating and modifying the `gSpan` algorithm to only return the top-k substructure patterns from the database. This can drastically reduce the computation time needed to mine a given database in some cases as the search tree can efficiently be pruned, without strongly negatively impacting the accuracy of the models built from the mined patterns, as using fewer patterns reduces overfitting. This is shown on Figure 1.

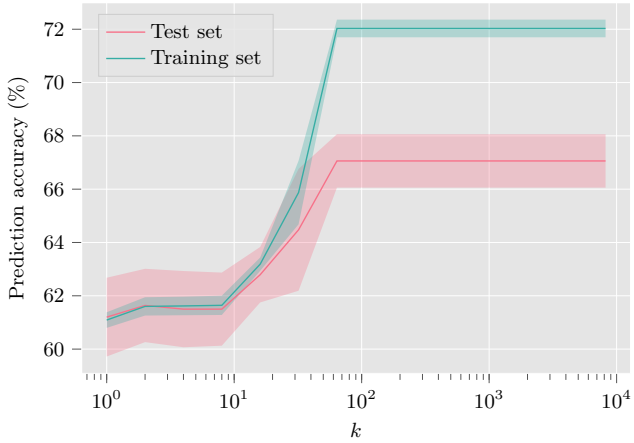

Figure 1: Influence of k on prediction accuracy of the model. For this experiment, the classifier was a `DecisionTreeClassifer(random_state=1)`, with accuracy being computed using 5-fold cross-validation and a minimum support value of 1000. The shaded region represents one standard deviation.

As one can see, the training set accuracy increases very quickly once the number of patterns used for classification increases, while the increase in test accuracy is much more minimal. We also observe that beyond a certain value of k, no further changes occur to the prediction accuracy. This is because there are only a limited number of patterns satisfying the minimum support constraint.

4.2 Training a model

This section groups the analysis for the final three tasks, as all three are concerned with building a machine learning model based on patterns that were mined.

4.2.1 Defining a performance metric

In order to compare various machine learning models, it is important to choose a good performance metric. Several choices exist for metric, each with its own properties. For the purpose of this assignment, we were tasked with using the accuracy, i.e. the proportion of correct classifications in the test set.

4.2.2 Estimating model quality

Now that we have decided which performance metric to use, we need to define an estimation procedure. The traditional choice is to use k-fold cross-validation. Cross-validation works by dividing the dataset into k distinct subsets, called "folds". k models are then trained on $k-1$ folds, and the accuracy on each left-over validation fold is averaged to es-

timate the true model accuracy.
In order to detect eventual overfitting, it can also be useful to look at the model accuracy on the data it was trained with. If there is a large disparity between both accuracy figures, the model does not generalize very well.

4.2.3 *Visualizing performance*

Once the metric and evaluation procedure have been defined, it is important to find a clear way to compare the models. For this purpose, one can use so-called "learning curves", which depict the evolution of the performance for a given model and metric as a function of training set size. Examples of such curves are given in Figures 2 to 6.

4.2.4 *Tuning the model*

Once all the previous points have been handled, one can tune the model by choosing hyper-parameters which perform the best. Other than trial-and-error, and occasionally using some domain-specific knowledge, there are no one-size-fits-all guidelines for this step. However, a good way of exploring the hyper-parameter space is to use exponentially-spaced values for each, and using scikit-learn's `GridSearchCV` function, which computes the cross-validated accuracy (as explained in Section *4.2.2*) for each possible combination of hyper-parameters. Other choices include using a specialized hyper-parameter optimization library, such as Hyperopt [2, 1].

4.2.5 *Results*

The following tests summarize the performance of the various models when trained on the full `molecules` dataset, with minsup $= 1000$ and $k = 64$. These values were chosen so as to provide a good tradeoff between accuracy of the generated models and execution time of the `gSpan` algorithm. No particular preprocessing was done, in order to stay in a pattern-based environment. However, especially in cases where overfitting occurs a lot, it would be a good idea to apply e.g. a principal components analysis transformation to the data, in order to reduce the problem dimensionality. Doing this correctly using scikit-learn can be done using the `pipeline` module.
All figures use shading to indicate one standard deviation, and the x-axis indicates the percentage of the total data set used as training set (when using k-fold cross-validation, the training set uses $k - 1$ folds, whereas the remaining fold is used as validation set).

4.2.5.1 *Learning curves.*

Figure 2 gives the performance of a decision tree classifier. Results indicate that there is quite a bit of overfitting with this model, though one should expect to have accuracy close to 70 % on an unknown test set.
Figure 3 gives the performance of a support vector machine classifier. The figure indicates there is a lot less overfitting than with the decision tree classifier, though results in general are disappointing, with an expected accuracy of about 65 % on an unknown test set but a large standard deviation on this estimate.
Figure 4 gives the performance of a nearest neighbors classifier. The k-nearest neighbors algorithm performs similarly to the support vector machine classifier, but has much lower variance on its estimated 65 % accuracy.
Figure 5 gives the performance of a multi-layer perceptron

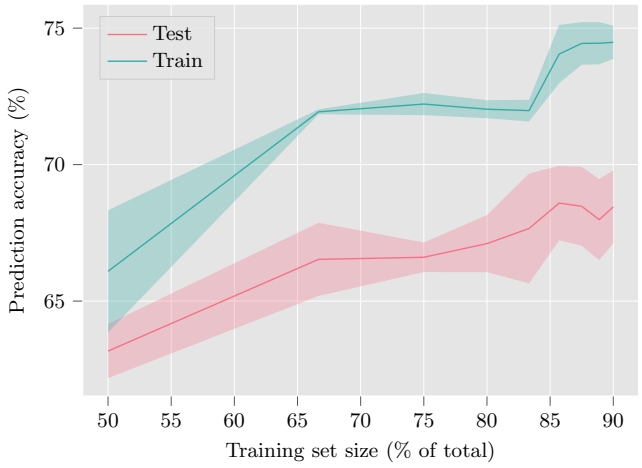

Figure 2: Learning curves on the train and test sets for the decision tree classifier.

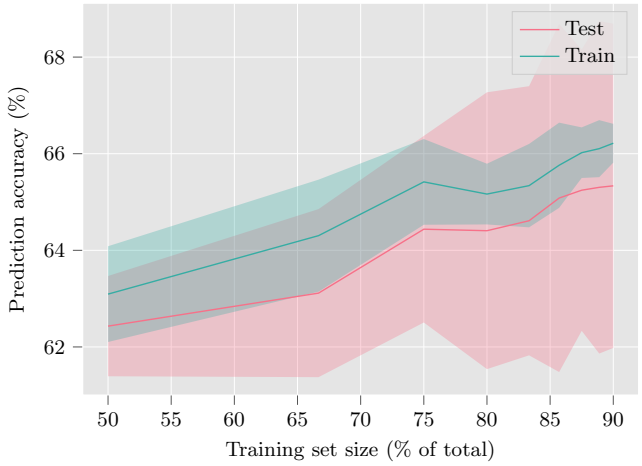

Figure 3: Learning curves on the train and test sets for the support vector machine classifier.

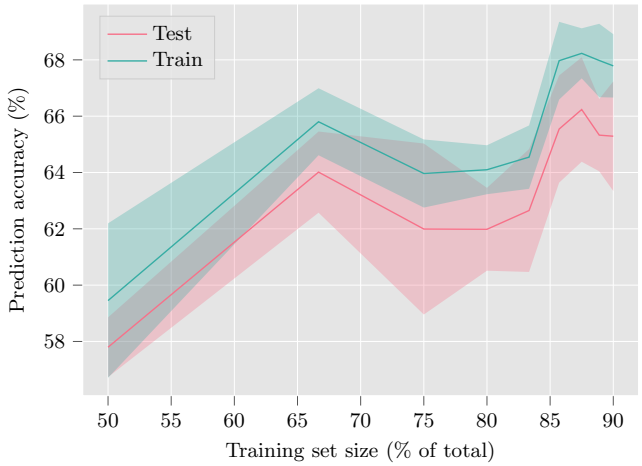

Figure 4: Learning curves on the train and test sets for the nearest neighbors classifier.

classifier. The multi-layer perceptron performs very well,

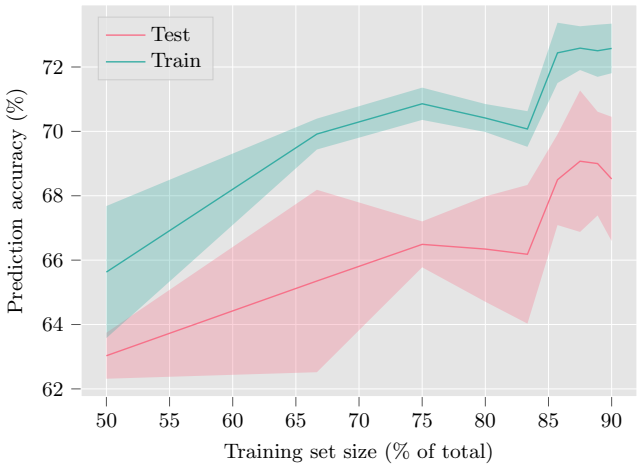

Figure 5: Learning curves on the train and test sets for the multi-layer perceptron classifier.

with an expected accuracy near 70 %, like the decision tree classifier, but with much less overfitting.
Figure 6 gives the performance of a classifier using sequential covering. Sequential covering performs the worst out of

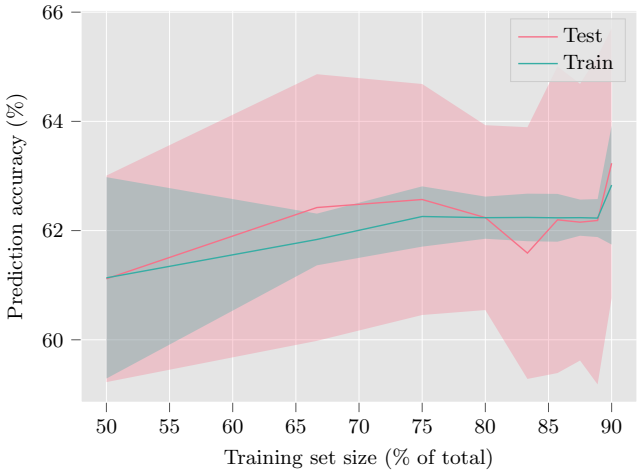

Figure 6: Learning curves on the train and test sets for the classifier using sequential covering.

all the algorithms we tested, while also having the highest variance. Its estimated accuracy is between 60 % and 65 %. The above figures thus indicate that in order to obtain good performance, it is preferable to use either a decision tree, or a multi-layer perceptron.

5. CONCLUSION

Many situations can be more accurately modelled as problems on graphs. When one needs to label graphs as being part of a given class, it is very important to have efficient algorithms, both for the mining part and for the classification part. The former can e.g. be done using gSpan, one of the fastest graph mining algorithms that are currently known. The latter can be done using a myriad of classifiers, and it is often impossible to predict ahead of time which will perform the best. For this, several benchmarking tools exist, such as cross-validation, which allow an end-user to build a performant solution to their problem.

References

[1] J. Bergstra. Hyperopt: Distributed asynchronous hyperparameter optimization in Python, 2013.

[2] J. Bergstra, D. Yamins, and D. D. Cox. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, I–115–I–123, Atlanta, GA, USA. JMLR.org, 2013.

[3] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec. 1989. ISSN: 1435-568X. DOI: 10.1007/BF02551274. URL: https://doi.org/10.1007/BF02551274.

[4] M. Kuramochi and G. Karypis. An efficient algorithm for discovering frequent subgraphs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1038–1051, 2004.

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[6] Xifeng Yan and Jiawei Han. gSpan: graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* Pages 721–724, 2002.