Spatial Pattern Recognition

Wesley Wei Qian on 07-20-15

Contents

Introduction

This package implments a probabilistic parametric model which can be trained to operate soatial pattern recognition task on ARGs, which comes from the idea of P. Hong & T.S. Huang.

The package is implmented in MATLAB using OOP and it is very easy to change the converging function, matching compatabiltiy function for different tasks (e.g., image/video retrieval, understand chemical comounds structure, discover gene regulation pattern, etc.)

Code Explanation

In this section, I will explain the basic structure of the code base and how to use them.

Basic Component

- **Class Components**
  - **sprMDL.m** which is the most important class representing the trained model.
    - The constructor will take a cell array of sample ARGs and the number of components and start the training.
    - The model is trained using an EM algorithm, whose converging condition and maximum iteration can be changed.
    - It has a number of model ARGs representing different component in the model and different weight associated with them.
    - Once the model is built, you can ask for the pattern that is being summarized in the model or if a new ARG has a similar pattern of the given sample ARGs, and if so, what is the pattern ARG?
  - **ARG.m** which represents the basic *sample* ARG with a cell array of Nodes and a symmetric cell matrix of Edges.
    - It is created with a full *NxN* matrix representation of the graph and a *N* length cell array of nodes attributes.
  - **node.m** which represents the node of a basic ARG.
    - It is created in the basic ARG construction, with an ID (construction order) and an attributes field.
  - **edge.m** which represents the edge of a basic ARG.

        - It is created in the basic ARG construction, with the edge attributes, two IDs of the edge endpoint and a cell array of Nodes (this constuction can be simplify).
    - **mdl_ARG.m** which is very similar to the ARG.m but represents an individual componennt in the probabilsitic parametric model and has more functionality.
        - It is created with a basic ARG as each component in the model is first initalized as a sample ARG.
        - Different than the basic ARG, model ARG can modify its structure, update nodes/edges accroding to the main model class.
    - **mdl_node.m** which inherits from the *node.m* class and represent the node in the model ARG.
        - It is created with a basic node during the construction of model ARG.
        - It has the ability to update its attributes vector and calcualte the inverse of its attributes.
    - **mdl_edge.m** which inherits from the *edge.m* class and represent the edge in the model ARG.
        - It is created with a basic edge during the construction of model ARG.
        - It has the ability to update its attributes vector and calcualte the inverse of its attributes.
- **Function Components**
    - **graph_matching.m** is another important component which return a matching matrix denoted the matching score between nodes of sample ARG and model ARG.
        - Graph Matching/Subgraph Isomorphism problem is a long-known NP-problem, so to solve the probelm we use God and Rangarajan's graduated assignment approach to get a good approxiamation efficently.
        - An individual implementation of such algorithm can be found in the `..\GraphMatching` folder, where the function will take two basic ARG instead of a basic ARG and a model ARG as required by the model training here.
    - **converege.m** is a function associate with *graph_matching.m* as it determines if the training result converges.
        - User can chagen this function with iteration variable `I_` and converging variable `beta_` in *graph_matching.m* to decide the accuracy of the matching score returned from *graph_matching.m*.
    - **heuristic.m** is another function associate with *graph_matching.m* as it is used to decorated the result from tha matching function.
        - For our purpose, we simply return what the algorithm returns, but for Graph Matching/Subgraph Isomorphism problem, such function can be used to pick the best matching between two ARGs, as written in `..\GraphMatching\heursitic.m`.
    - **edge_compatability.m** and **node_compatability.m** is the function calculate the matching score between nodes and edges according to their attributes and covariance matrix.
        - In our implementation, we assume both node and edge follow a Gaussian distribution, but this can be changed accroding to the application scenario.
    - **mdl_converge.m** is a fucntion associated with the main model class `sprMDL.m` and it determines if the model is converged.
        - By changing this function and iteration variable `iteration_EM` and converging variable `e_mdl_converge` in *graph_matching.m*, user can define the training accuracy of such model.
- **Test/Demo Components**
    - **test.m** is a simple setup to demo the model, where as you can notice, the nodes attributes can be vecotr.
    - **ModelTest.m** is a random test script to test the accurracy and efficiency of the model.

    - The test first build up a base pattern and buil some training samples with such pattern. The model is them trained with these samples and test against some test samples with such patterns and some samples genreated randomly.
    - User can show the pattern of the model and its componenets by making variable `view_pattern` to `1` instead of `0`.
    - User can also change other variables (e.g. pattern size, number of components, pattern connected rate) in the script for different testing purposes.

  - **..\GraphMatching\test.m** is a simple setup to demo the graph matching algorithm.

  - **..\GraphMatching\RandomGraphTest.m** is a random test script to test the accuracy and efficiency of the graph matching function.

    - The test first build a random graph, then create another graph by permutating the first graph (nodes order and attributes). The matching correcteness is then being calculated.
    - User can also change other variables (e.g. number of rounds, the size of the graph, noise rate) in the script for different testing purposes.

Usage

Despite all the complex components, the usage of such model is rather simple. Assume you have:

- a set of training sample ARGs listed in a cell array `sample_ARGs`;
- an ideal number of component for the model `number_of_component`;
- a test ARG that you want to know if has the similar pattern as the set of samples `test_ARG`;

Then to use the model, you can do the following:

```
% We first train a model by sending it the samples and the number_of_component,
% and a model will be returned.

    mdl = sprMDL(sample_ARGs, number_of_component);


% Then we can ask for the pattern that is summarized in the model,
% and a summarized pattern will be returned as an ARG,
% in the mean time, a biograph will show up to visualze the pattern.

    summarized_pattern = mdl.summarizedPattern();

% Besides showing the whole model, you can also ask to show a specific component (the ith),
% and the function will return a structure consisting the information of this model,
% including nodes attributes, nodes frequency, edges matrix
% as well as a visualization component bg that can be viewed as biograph.

    component_struct = mdl.mdl_ARGs{i}.showARG();
    view(component_struct.bg);


% We can also check if the test_ARG has a similar pattern,
% and a boolean value we tell us if the test_ARG has a simialr pattern.

    tf = mdl.checkSamePattern(test_ARG);


% If you are greeedy, you can also ask for which part of the test_ARG are similar to the sample ARG.
% If there is indeed a similar pattern, the pattern will be returned as an ARG.
% Otherwise, NaN will be return
```

```
    the_similar_pattern = mdl.getSamePattern(test_ARG);
```

Advanced Usage

Since the implementation is written in OOP, it can be modified easily. However, reading through the implementation detail in the following secation is highly recommended.

As I mentioned in the basic component section, user can modify the edge/node compatability functiona and the converging function for both the graph matching algorithm and EM algorithm during the model training. However, there are other properties that the user can tune to fit their application best.

For the spatial pattern recognition model, user can change the following constants in `sprMDL.m` :

```
    properties (Constant)
        % Maximum EM algorithm running rounds
            iteration_EM = 30;

        % Converging epsilon, the converging thredshold
         e_mdl_converge = 1e-4;

        % Structure Modification Setup
        % We don't want to delete nodes in early stage
        % so choose such number carefully
        % The thredshold is set up as e_delete_base - e_delete_iter^iter
            e_delete_base = 1;
            e_delete_iter = 0.5;
    end
```

Moreover, in the current implementation, the model use the lowest matching score from sample to set the thredshold for judging if a new ARG has the same pattern. Such thredshold setting system can also be changed according to user's application.

For the graph matching probelm, user can change the following variables `graph_matching.m` :

```
    % beta is the graduaded assignemnt update
        beta_0 = 0.5;
        beta_f = 10;
        beta_r = 1.075;

    % I control the iteration number for each round
        I_0 = 4;
        I_1 = 30;

    % e is the converging epsilon thredshold
        e_B = 0.5;
        e_C=0.05;

    % node attriubute compatability weight in the score
        alpha = 0.1;
```

Last but not least, even though some implementation is discussed in the following section, understand the underlying idea of the spatial pattern recognition model and the graph matching is very important, so pelase take a look at these two paper.

Some Implementation Detail

- **Null Edge/Node Representation**

  During the development of this project, we found that matrix operation on `NaN` value is much slower than `0` value, so we decide to use `0` value to represent a null value, or a non-existen edge or node. In the compatability functoin, if the function detect a zero, it will return a zero score so that a non-existen edge/node would not have any impact on the result.

  However, if you do have a zero attributes for your existen edge/node, a good get around can be adding a small constatn to all your edge/node attributes and then set tha `NaN` value to `0`.

- **Edge Attributes' Vector Representation**

  During the development, we assume that the edge attribute is just a weight, so we assume the edge attribute will always be a weight so the connection is represented by a simple matrix.

  However, in you real life, the edge attributes can be vector. So to modify the implementation, I think the major modification you might need to do is changing the constructon in `ARG.m` and `mdl_ARG.m`.

  If you think this modification is too much, we are very likely to solve this probelm during our next update.

- **Edge Compatability Calculation during Graph Matching**

  As you might notice, we implemented two methods to calculating the edge compatability depends on the connected rate of the graph.

  The first way is we go nodes by nodes in graph to pariing four nodes (two nodes for each graph). If we have a graph with `A` nodes and another graph with `I` nodes, this operation can be as expensive as `O(A^2*I^2)`. Thanks to the effecient matrix operation in MATLAB, we can convert this potential four `for` loop operation into simple matrix operation.

  However, a lot of time will be wasted if the graph has a very low connected rate and has very little edges. So we discvoer the second way, where we only calculate the matching score for existen edges using a sparse matrix.

  Now you might think the second method can definitely saves time, but it is not true. Since we need to use some nested for loop in the second method, if the connected rate is high enough, the first method can actually beat the second method. That's why we only use the second method when the connected rate is less than `0.4` in the code.

- **Node Compatability Weight**

  In the code for the graph matching function, there is a variable `alpha` that can weight the node compatabiltiy score and you can see it as a control to how much the node compatability impact the algorithm against the edge compatabilyty.

  For example, if you have a lot of nodes that has the same attribute while there are only a few number of edges. The edge compatability will be much more important during your graph matching task. In this case, you would like to lower the `alpha` so that the algorithm can focus on the edges.

  However, if your graph has a lot of similar edges while the variance for node attributes are very high, you would like the algorithm to focus more on the node as against the edges. In this case, you would like to make `alpha` larger.

- **Model Modification During the Training**

  As I mentioned above, when we initialize our pattern recognition model, we initialzied our components by randomly picking samples. Therefore if we didn't do any strutrue modification on our components, our components can have much more nodes than it needed to represent the summarized pattern, which means more background noise is introduced.

  To deal with this case, we add a structure modification step so that we can cut the unnecessary node, or less weighted node in each EM iteration. To do so, we set up an thredshold score and change this thredshold score at each iteration so that the cutting rule is more ane more strict after each iteration.

However, we don't want to cut nodes at the beginning of each training since our initialization can be wrong and the node we cut can be an important representation of the model.

- **Component Weight and Component Nodes Number**

  During the development, I found out that the model tends to give a higher weight for the component with more nodes, even though the pattern they need to summarized is far less than it. It makes sens because a componennt with more nodes simply has a better chance to match somthing from our sample patterns.

  However, in this case, the model might say yes to some random pattern instead of the one they summarized. Therefore, we introduce a normalization in the calculation of sample-component matching score, where we divdie the score by the `log` of the component's size.

  Therefore, we can make the components more fair to the components having less weight and taking a `log` can make such change smoother.

- **Implementation Efficiency**

  As you can imagine, EM algorithm training and graph matching problem can both be very expensive task in terms of time efficeincy. In the graph matching function, we use a lot of matrix operation so that MATLAB can optimize the runtime for us. However, during the development of the spatial pattren recognition model, we found it hard to make matrix representation of our information so there are some huge nested `for` loop in the code, which can slow the running time.

  So this is definitely an improvement we can continue to work on.

Result

Even though more advanced test might be needed, here are some of the result we currently have:

For the spatial pattern recognition model, it takes about **5 hours** to train a model with a 10-node-pattern, fifty 20-node-sample, five components, 4% connected rate on an average commercial machine. Then the model is test on fifty testing samples with the patterns and fifty random samples may or maynot have the patterns. The recognition rate is **92%** compared to a **56%** recognition rate for the random samples.

For the graph matching function, it takes about **30 minutes** to run a match on two ARGs with 100 nodes and a 5% connected rate permutating in 10% noise. We run 10 rounds of such match and the average correct rate is about **99%**.