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_compone
nt,
% 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 compo
nent (the ith),
% and the function will return a structure consisting the information of thi
s 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 si
milar to the sample ARG.
% If there is indeed a similar pattern, the pattern will be returned as an A
RG.
% 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
```

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
- Edge/Node attributes' vector representation
- Model Modification During the Training
- Edge Compatability Calculation during Graph Matching

More to come.

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%**.