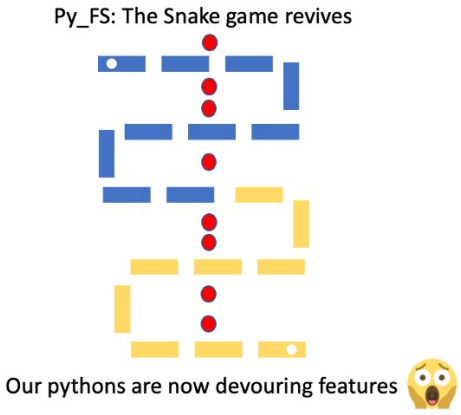
Py_FS: A Python Package for Feature Selection



Ritam Guha, Shameem Ahmed, Trinav Bhattacharyya, Bitanu Chatterjee, Ali Hussain Khan, Khalid Hassan Sk, Manosij Ghosh

supervised by
Dr. Ram Sarkar

An initiative by
Department of Computer Science and Engineering
Jadavpur University
India
November, 2020

User Manual for Py_FS: A Python Package for Feature Selection

November 2020

1 Introduction

Feature Selection (FS) is considered as one of the most important pre-processing steps in any machine learning or data mining algorithm. With the rapid advancement in science and technology, we are dealing with huge amount of data. The dimensions of these data are increasing exponentially, day-by-day. Now, for a normal machine learning or data mining algorithm, it becomes a tedious job to handle such a huge dataset. And needless to say, we have entered an era where data is considered as the new currency, which has the ability to change the way we think. So, handling data with utmost care is very important. FS makes the job easier for any such algorithm by reducing the dimensionality of the dataset. The rejected features are considered redundant/ irrelevant for any classification purpose. Hence, by reducing the number of features, FS helps to reduce the computational complexity as well as the storage requirement. Moreover, by removing the unwanted features which may act as noise during the classification process, the classification accuracy can be improved a lot. Reducing the number of redundant features drastically reduces the running time of a learning algorithm, which helps in obtaining a better comprehension into the basal intricacies of a practical classification problem. FS methods try to choose a subset of features that are relevant to the learning model. This would eventually help reduce the training time of the learning model and storage requirement.

Now-a-days, FS is extensively used in several fields for its effectiveness. As it helps in improving results of classification problems, the research community is investing a significant amount of time in it. We can divide FS methods broadly into two categories based on the evaluation criteria of the features: Filter method and Wrapper method.

Filter methods uses some statistical measure or intrinsic properties of the dataset and provides a rank-list of the features based on their relevance. This approach measures the relevance of features by their correlation with dependent variables.

Wrapper methods, on the other hand, proceed by introducing random subsets of features at the beginning. A learning algorithm (like a classifier) is then used to measure the quality of the subsets followed by a guided mechanism where the better subsets guide the worse ones to improve their own quality by discarding or selecting appropriate features. These methods are very effective for any kind of classification task. But, use of a learning algorithm makes wrapper methods computationally expensive and inappropriate for use when there are certain constraints on computational power and time requirements.

This user manual aims at providing a better understanding of our python package (Py_FS) to make the users comfortable at handling the different FS methods. The understanding of the APIs of Py_FS will help the users to make effective use of this package. The user manual will be updated continuously as per requirement. **Welcome to the world of *Py_FS*!!!**

2 Modules

Py_FS currently focuses on three different sectors of feature selection: nature-inspired wrappper methods, filter methods and evaluation metrics. These three sectors are described in detail:

2.1 Wrapper-based Nature-inspired Feature Selection

Wrapper-based Nature-inspired methods are very popular feature selection approaches due to their efficiency and simplicity. These methods progress by introducing random set of candidate solutions (agents which are natural elements like particles, whales, bats etc.) and improving these solutions gradually by using guidance mechanisms of fitter agents. In order to calculate the fitness of the candidate solutions, wrappers require some learning algorithm (like classifiers) to calculate the worth of a solution at every iteration. This makes wrapper methods extremely reliable but computationally expensive as well.

Py_FS currently supports the following 12 wrapper-based FS methods:

- Binary Bat Algorithm (BBA)
- Cuckoo Search Algorithm (CS)
- Equilibrium Optimizer (EO)
- Genetic Algorithm (GA)
- Gravitational Search Algorithm (GSA)
- Grey Wolf Optimizer (GWO)
- Harmony Search (HS)

- Mayfly Algorithm (MA)
- Particle Swarm Optimization (PSO)
- Red Deer Algorithm (RDA)
- Sine Cosine Algorithm (SCA)
- Whale Optimization Algorithm (WOA)

2.2 Filter-based Feature Selection

Filter methods do not use any intermediate learning algorithm to verify the strength of the generated solutions. Instead, they use statistical measures to identify the importance of different features in the context. So, finally every feature gets a rank according to their relevance in the dataset. The top features can then be used for classification.

Py_FS currently supports the following 4 filter-based FS methods:

- Pearson Correlation Coefficient (PCC)
- Spearman Correlation Coefficient (SCC)
- Relief
- Mutual Information (MI)

2.3 Evaluation Metrics

The package comes with tools to evaluate features before or after FS. This helps to easily compare and analyze performances of different FS procedures.

Py_FS currently supports the following evaluation metrics:

- classification accuracy
- average recall
- average precision
- average f1 score
- confusion matrix
- confusion graph

3 Function Descriptions

In this section, the functions available in Py_FS are described along with their necessary and optional parameter configurations.

3.1 Wrappers

General Parameters:

Some parameters are used very frequently. We need to discuss these parameters before proceeding with the function prototypes.

- **num_agents (integer)**: number of search agents
- **max_iter (integer)**: maximum number of generations
- **train_data (numpy array)**: training samples of data
- **train_label (numpy vector)**: class labels for the training samples
- **obj_function (function)**: the objective function for feature

  format: obj_function(*agent, train_X, test_X, train_Y, test_Y*)
  When the user wants to provide an objective function, this format needs to be followed. The parameters for the function are:

  agent (numpy vector): an alias of a search agent in the process
  train_X (numpy array): set of training data
  test_X (numpy array): set of test data
  train_Y (numpy vector): set of training labels
  test_Y (numpy vector): set of test labels

  default: compute_fitness(*agent, train_X, test_X, train_Y, test_Y*) compute_fitness computes the fitness of an agent using the following equation:

$$fitness = 0.7 \times accuracy(agent) + 0.3 \times \frac{tot\_feat - sel\_feat}{tot\_feat} \quad (1)$$

  where $accuracy(agent)$ refers to the classification accuracy provided by the agent, tot_feat is the total number of features present in the agent and sel_feat is the number of features selected by the agent. It can be seen that 70% importance is provided to accuracy, while 30% is provided to the number of unselected features.

- **trans_function_shape (character)**: the shape of the function to map real values to binary values for feature selection

  Currently, Py_FS provides three different shapes of transfer functions: 's', 'v', 'u'
  default: *'s'*

- **save_conv_graph (boolean)**: boolean value stating if the user wants to save the convergence graphs produced during the process
  default: *False*

Function Prototypes:

1. **Py_FS.wrapper.nature_inspired.BBA**(*num_agents, max_iter, train_data, train_label, obj_function=compute_fitness, trans_function_shape='s', constantLoudness=True, save_conv_graph=False*)

   Unique Parameters:

   - **constantLoudness (boolean)**: boolean value to ensure if the user wants to keep the loudness constant throughout the generations
     default: *True*

2. **Py_FS.wrapper.nature_inspired.CS**(*num_agents, max_iter, train_data, train_label, obj_function=compute_fitness, trans_function_shape='s', save_conv_graph=False*)

3. **Py_FS.wrapper.nature_inspired.EO**(*num_agents, max_iter, train_data, train_label, obj_function=compute_fitness, trans_function_shape='s', save_conv_graph=False*

4. **Py_FS.wrapper.nature_inspired.GA**(*num_agents, max_iter, train_data, train_label, obj_function=compute_fitness, trans_function_shape='s', prob_cross=0.4, prob_mut=0.3, save_conv_graph=False*)

   Unique Parameters:

   - **prob_cross (range: [0, 1])**: probability of crossover
     default: *0.4*

   - **prob_mut (range: [0, 1])**: probability of mutation
     default: *0.3*

5. **Py_FS.wrapper.nature_inspired.GSA**(*num_agents, max_iter, train_data, train_label, obj_function=compute_fitness, trans_function_shape='s', save_conv_graph=False*)

6. **Py_FS.wrapper.nature_inspired.GWO**(*num_agents, max_iter, train_data, train_label, obj_function=compute_fitness, trans_function_shape='s', save_conv_graph=False*)

7. **Py_FS.wrapper.nature_inspired.HS**(*num_agents, max_iter, train_data, train_label, obj_function=compute_fitness, trans_function_shape='s', save_conv_graph=False*)

8. **Py_FS.wrapper.nature_inspired.MA**(*num_agents, max_iter, train_data, train_label, obj_function=compute_fitness, trans_function_shape='s', prob_mut=0.2, save_conv_graph=False*)

   Unique Parameters:

   - **prob_mut (range: [0, 1])**: probability of mutation
     default: *0.2*

9. **Py_FS.wrapper.nature_inspired.PSO**(*num_agents, max_iter, train_data, train_label, obj_function=compute_fitness, trans_function_shape='s', save_conv_graph=False*)
10. **Py_FS.wrapper.nature_inspired.RDA**(*num_agents, max_iter, train_data, train_label, obj_function=compute_fitness, trans_function_shape='s', save_conv_graph=False*)
11. **Py_FS.wrapper.nature_inspired.SCA**(*num_agents, max_iter, train_data, train_label, obj_function=compute_fitness, trans_function_shape='s', save_conv_graph=False*)
12. **Py_FS.wrapper.nature_inspired.WOA**(*num_agents, max_iter, train_data, train_label, obj_function=compute_fitness, trans_function_shape='s', save_conv_graph=False*)

Function returns:

Each one of the functions return an object of class. This Solution class has following member variables:

Solution

- **best_agent**: best feature vector over all the iterations
- **best_fitness**: fitness value of the best_agent
- **best_accuracy**: classification accuracy of the best_agent
- **final_population**: final population of agents
- **final_fitness**: fitness value of the final_population
- **final_accuracy**: classification accuracy of the final_population
- **convergence_curve**: record of fitness and number of features over the course of iteration
- **execution_time**: time required to execute the piece of code

3.2 Filters

Every filter method in Py_FS uses two general parameters:

- **data (numpy array)**: training data
- **target (numpy vector)**: class labels for training data

Function Prototypes

1. **Py_FS.filter.MI**(*data, target*)
2. **Py_FS.filter.PCC**(*data, target*)
3. **Py_FS.filter.Relief**(*data, target*)

4. **Py_FS.filter.SCC**(*data, target*)

Function returns:

Each one of the functions return an object of a class. This class has following member variables:

Solution

- **scores**: scores provided to the features
- **ranks**: ranks of the features
- **ranked_features**: the feature values after ordering them according to ranks

3.3 Evaluation Metrics

The third utility of Py_FS is the evaluation metrics for the classification.

Function Prototype:

Py_FS.evaluation.evaluate(*train_X, test_X, train_Y, test_Y, agent=None, classifier='knn', save_conf_mat=False*)

Parameters:

- **train_X (numpy array)**: set of training data
- **test_X (numpy array)**: set of test data
- **train_Y (numpy vector)**: set of training labels
- **test_Y (numpy vector)**: set of test labels
- **agent (numpy vector)**: an alias of a search agent in the process
  default:*None*
- **classifier (string)**: name of the classifier to use. Py_FS currently supports three classifiers:
  'knn': K-Nearest Neighbors
  'rf': Random Forest
  'svm': support vector machine
  default:*'knn'*
- **save_conf_mat (boolean)**: boolean value stating if the user wants to save the confusion matrix produced during the process

Function returns:

The evaluate function return an object of a Result class. This class contains the following member variables:

Result

- **predictions**: prediction labels generated for test data (test_X)
- **accuracy**: classification accuracy provided by the features in agent. If agent is None, all the features are used.
- **recall**: recall value for the predictions
- **precision**: precision value for the predictions
- f_1 **score**: f_1 score for the predictions
- **confusion_matrix**: confusion matrix of classification

4 Quick Demonstration

Please dive into the following link for a quick demonstration of Py_FS:
Py_FS: Demonstration