OPFython: A Python-Inspired Optimum-Path Forest Classifier

Gustavo H. de Rosa, João P. Papa
Department of Computing
São Paulo State University
Bauru, São Paulo - Brazil
gustavo.rosa@unesp.br, joao.papa@unesp.br

Alexandre X. Falcão
Institute of Computing
University of Campinas
Campinas, São Paulo - Brazil
afalcao@ic.unicamp.br

January 29, 2020

ABSTRACT

Machine learning techniques have been paramount throughout the last years, being applied in a wide range of tasks, such as classification, object recognition, person identification, image segmentation, among others. Nevertheless, conventional classification algorithms, e.g., Logistic Regression, Decision Trees, Bayesian classifiers, might lack complexity and diversity, not being suitable when dealing with real-world data. A recent graph-inspired classifier, known as the Optimum-Path Forest, has proven to be a state-of-the-art technique, comparable to Support Vector Machines and even surpassing it in some tasks. In this paper, we propose a Python-based Optimum-Path Forest framework, denoted as OPFython, where all of its functions and classes are based upon the original C language implementation. Additionally, as OPFython is a Python-based library, it provides a more friendly environment and a faster prototyping workspace than the C language.

Keywords Python · Machine Learning · Classifiers · Optimum-Path Forest

1 Introduction

Artificial Intelligence has become one of the most fostered research areas throughout the last years [1]. It is common to observe an increasing trend of automating tasks [2], which also fosters the development of minimal human interaction algorithms, denoted as Machine Learning.

Machine learning research consists of developing new types of algorithms that do not need explicit instructions, relying on patterns and inferences [3]. Also, they are designed in a way that humans can be assisted in decision-making tasks or even in daily activities automation, such as data retrieval [4], intelligent gadgets [5], self-driving cars [6], among others. In the past decades, most of machine learning-based algorithms were developed as symbolic- and knowledge-based models due to the difficulty in dealing with probabilistic models at that time [7]. Nevertheless, with the advent of computational power, probabilistic models were put in the spotlight as the availability of digitized information was no longer a problem [8]. Hence, most of today's algorithms rely on mathematical models and data sampling, i.e., models that are capable of learning occult patterns in training data and predicting unseen data, and are to a wide variety of tasks, such as computer vision [9] and natural language processing [10].

One can observe that it is possible to divide machine learning algorithms into two types of learning: supervised learning [11] and unsupervised learning [12], as depicted in Figure 1. Concerning the supervised learning ones, such as classification and regression tasks, the algorithms aim at building mathematical models from labeled data, i.e., data that contains the input features and the possible outputs (classes), and perform predictions on unseen data. Regarding the unsupervised learning, the algorithms aim at building mathematical models that are capable of aggregating sets of data with common characteristics, known as clusters. In other words, unsupervised learning is capable of discovering patterns in data and grouping them into categories without knowing their actual labels.
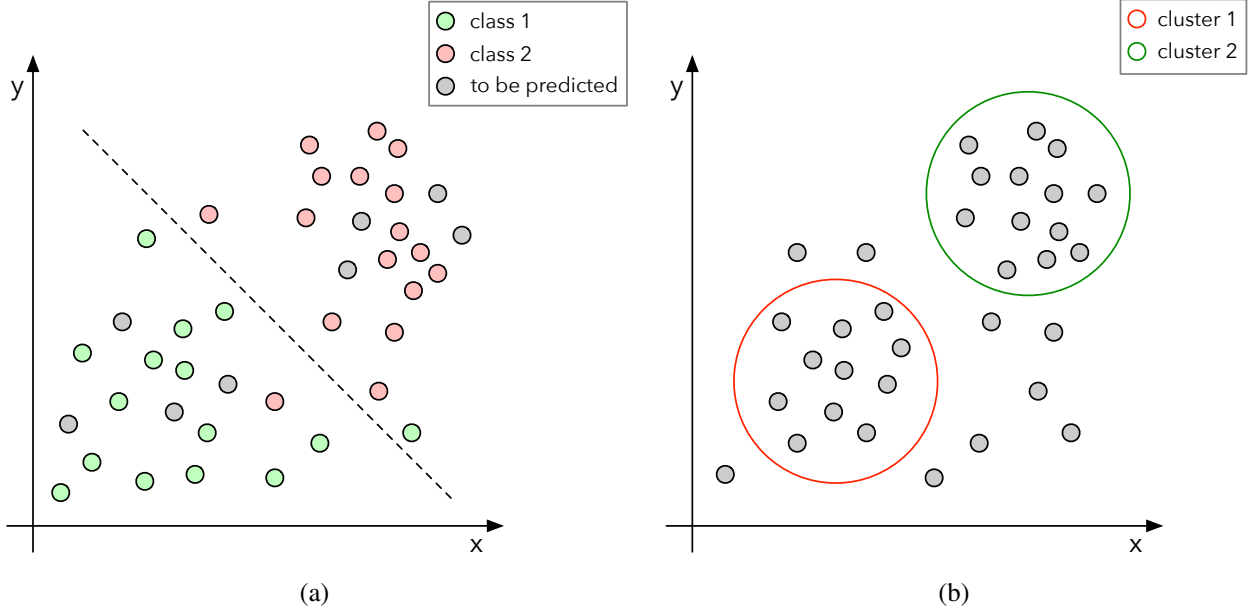
Figure 1: Illustration of the aforementioned machine learning types of learning: (a) supervised learning and (b) unsupervised learning.

Furthermore, it is crucial to observe that the machine learning area is closely related to other fields, such as data mining, optimization, and statistics. Regarding data mining, while machine learning focus on predicting information based on already-known properties from the data, the data mining area focuses on finding unknown properties in the data and transforming unknown knowledge into real-knowledge, for further application in machine learning algorithms [13]. Concerning the optimization area, it is common to observe that most machine learning algorithms models are formulated as optimization problems, where some loss function is minimized over a set of training data. Essentially, the loss function is capable of expressing the discrepancy between the model's predictions and the actual samples, assisting the algorithm in learning the data's patterns and being capable of predicting unseen information [14]. Finally, regarding the statistics field, it is possible to perceive that statistics focus on drawing inferences from samples, while machine learning focuses on finding generalizable prediction patterns [15]. Additionally, due to their intimate relationship, some researchers combined machine learning and statistical methods in a new field of study, known as statistical learning [16].

Recently, a new graph-based classifier proposed by Papa et al. [17], known as Optimum-Path Forest (OPF), attempts to fulfill the literature with a parameterless classifier, which is effective during the learning step and efficiently when performing new predictions. Several works introduced the capacity of OPF and its state-of-the-art performance, being comparable to the well-known Support Vector Machines (SVM) [18] in supervised [19] and unsupervised learning [20] tasks. Additionally, it provides tools, such as graph-cutting and K-Nearest Neighbors (KNN) graphs [21], to reduce the training set size with negligible effects on the accuracy of the classification. Nevertheless, the problem arises with the fact that there is only one official implementation based on the C language, making it difficult to be integrated with other well-known machine learning frameworks. Furthermore, there is a Python-based trend in the machine learning community.

In this paper, we propose an open-source Python Optimum-Path Forest classification library, called OPFython[1]. Mainly, the idea is to provide a user-friendly environment to work with Optimum-Path Forest classifiers by creating high-level methods and classes, removing from the user the burden of programming at a mathematical level. The main contributions of this paper are threefold: (i) to introduce an Optimum-Path Forest classification library in the Python language, (ii) to provide an easy-to-go implementation and user-friendly framework, and (iii) to fill the lack of research regarding Optimum-Path Forest classifiers.

The remainder of this paper is organized as follows. Section 2 presents a literature review and related works concerning Optimum-Path Forest classifiers frameworks. Section 3 introduces a theoretical background concerning the supervised and unsupervised Optimum-Path Forest classifiers. Section 4 introduces thoughts of the OPFython library, such as its architecture, and an overview of the included packages. Section 5 provides more profound notions about the library,

---

[1]https://github.com/gugarosa/opfython

such as how to install, how to understand its documentation, some pre-included examples, and how to perform unitary tests. Furthermore, Section 6 presents vital knowledge about the usage of the library, i.e., how to run pre-defined examples, and how to model a new experiment. Finally, Section 7 states conclusions and future works.

2 Literature Review and Related Works

Optimum-Path Forest classifiers have arisen as a new approach to tackle supervised and unsupervised problems. They offer a parameterless graph-based implementation that is capable of executing an effective learning procedure while being extremely efficient when performing new predictions. It is possible to find its usage in a wide range of applications, such as feature selection [22], image segmentation [23, 24], signals classification [25, 26]. For instance, Iliev et al. applied an Optimum-Path Forest classification using glottal features for spoken emotion recognition, achieving state-of-the-art results comparable to the SVM classifier. Moreover, Ramos et al. [27] applied an OPF-based classification for detecting non-technical energy losses, achieving outstanding results comparable to state-of-the-art artificial intelligence techniques. Furthermore, Fernandes et al. [28] proposed a probabilistic-driven OPF classifier for detecting non-technical energy losses, improving the baselines obtained by the standard OPF.

Even though numerous works in the literature fosters the Optimum-Path Forests, there are some gaps in works regarding frameworks or open-sourced libraries. There is only an official implementation provided by Papa et al. [29], denoted as LibOPF, which does not provide straightforward tools for users to design new experiments or integrate with other frameworks. Additionally, it lacks documentation and test suites, which assist users in understanding the code and implementing new methods and classes. Moreover, the library is implemented in C language, making it extremely difficult to integrate with other frameworks or packages, primarily because the machine learning community is turning their attention to the Python language.

Therefore, OPFython attempts to fill the gaps concerning Optimum-Path Forest frameworks. It is purely implemented in Python and provides comprehensive documentation, test suites, and several pre-loaded examples. Furthermore, every line of code is commented, there are continuous integration tests for every new push to its repository, an astounding readme which teaches on how to get started with the library and full-time maintenance and support.

3 Theoretical Foundation

Before diving into OPFython's library, we present a theoretical foundation about the Optimum-Path Forest. In the next subsections, we mathematically explain how the supervised and unsupervised classifiers work.

3.1 Supervised Optimum-Path Forest

The Optimum-Path Forest is a multi-class classifier developed by Papa et al. [17], being efficient in the training step and effective in the testing stage. Its foremost ability is to segment the feature space without requiring massive volumes of data. Essentially, the OPF classifier is a graph, having two possible adjacent relations: a complete graph or a KNN graph. The difference between both methods is the adjacency relation, the methodology to estimate the prototypes[2], and the path cost function.

The principal idea behind the supervised OPF is to construct a complete graph, where any two samples are connected. In this case, the nodes represent the samples' features vector and the edges connect all nodes. Regarding the prototypes, the same are chosen throughout Minimum Spanning Trees (MST)[3] in order to find the nearest samples from different classes, namely, by selecting samples located in the classes frontiers[4]. After the prototypes definition, they compete with each other to conquer adjacent nodes while trying to find the best path (lowest cost) defined by the path cost function and create Optimum-Path Trees (OPT). Finally, during the testing phase, OPF inserts each new sample into the graph and finds the prototype, which offers the minimum cost path (class labeling).

Let $\mathcal{Z}$ be a dataset, where $\mathcal{Z} = \mathcal{Z}_1 \bigcup \mathcal{Z}_2$, and $\mathcal{Z}_1$ and $\mathcal{Z}_2$ represents the training and testing sets, respectively. Each sample $s \in \mathcal{Z}$ can be represented by its feature vector $\overrightarrow{v}(s) \in \Re^n$. OPF_{cpl} graph is represented by $\mathcal{G} = (\mathcal{V}, \mathcal{A})$, where $\mathcal{A}$ refers to the set of edges that connects all nodes pairs and $\mathcal{V}$ is the features vectors set $\overrightarrow{v}(s)$, $\forall s \in \mathcal{Z}$. In addition, let $\lambda(\cdot)$ be a function that assigns a real label for each sample in $\mathcal{Z}$.

---

[2]Prototypes are master nodes that represent a specific class and conquer other nodes.
[3]MSTs are subgraphs that connect all nodes within the same set using the minimum possible cost.
[4]Regions more likely to classification mistakes.

3.1.1 Training Step

Let the graph $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{A})$ be inducted from the training set, where $\mathcal{V}_1$ holds all feature vectors from samples belonging to the training set. The first objective of the training phase is to obtain a set of prototypes $\mathcal{S}$, onde $\mathcal{S} \subset \mathcal{Z}_1$.

Let a path π_s, with ending in $\mathbf{s}$, be in $\mathcal{G}$ and a function $f(\pi_s)$ that associates a value to this path. In order to a prototype conquer adjacent samples, the purpose is to minimize $f(\pi_s)$ through a path cost function given by the following equation:

$$
\begin{aligned}
f_{max}(\langle \mathbf{s} \rangle) &= \begin{cases} 0 & \text{se } \mathbf{s} \in S, \\ +\infty & \text{otherwise} \end{cases} \\
f_{max}(\pi \cdot \langle \mathbf{s}, \mathbf{t} \rangle) &= \max\{f_{max}(\pi), d(\mathbf{s}, \mathbf{t})\}.
\end{aligned}
\tag{1}
$$

where $f_{max}(\pi \cdot \langle \mathbf{s}, \mathbf{t} \rangle)$ computes the maximum distance between adjacent samples $\mathbf{s}$ and $\mathbf{t}$ along the path $\pi \cdot \langle \mathbf{s}, \mathbf{t} \rangle$. A path π_s is referred as optimum if $f(\pi_s) \leq f(\tau_s)$ for any other path τ_s.

The minimization of f_{max} assigns to each sample $\mathbf{t} \in \mathcal{Z}_1$ an optimum path $P^*(\mathbf{t})$, whose minimum cost $C(\mathbf{t})$ is given by the following equation:

$$
C(\mathbf{t}) = \min_{\forall \pi_t \in (\mathcal{Z}_1, \mathcal{A})} \{f_{max}(\pi_t)\}.
\tag{2}
$$

3.1.2 Testing Step

The testing set graph $\mathcal{G}_2 = (\mathcal{V}_2, \mathcal{A})$ is composed by samples $\mathbf{t} \in \mathcal{V}_2$. Each sample $\mathbf{t}$ is connected to a sample $\mathbf{s} \in \mathcal{V}_1$, making $\mathbf{t}$ as part of the original graph. The objective is to find an optimum path $P^*(\mathbf{t})$ from $\mathcal{S}$ to $\mathbf{t}$ with class $\lambda(R(\mathbf{t}))$ of its prototype $R(\mathbf{t}) \in \mathcal{S}$. Consequently, the sample $\mathbf{t}$ is removed from the graph. This path can be identified by evaluating the optimum cost value $C(\mathbf{t})$:

$$
C(\mathbf{t}) = \min\{\max\{C(\mathbf{s}), d(\mathbf{s}, \mathbf{t})\}\}, \forall \mathbf{s} \in \mathcal{Z}_1.
\tag{3}
$$

3.2 Unsupervised Optimum-Path Forest

Let $\mathcal{N}$ be a dataset such that for every sample $s \in \mathcal{N}$ there is a feature vector $\boldsymbol{v}(s)$. Additionally, let $d(s, t)$ be the distance between samples s and t in the feature space, which is described by $d(s, t) = \|\boldsymbol{v}(t) - \boldsymbol{v}(s)\|$.

A graph $(\mathcal{N}, \mathcal{A})$ is defined by arcs $(s, t) \in \mathcal{A}$ that connect k-nearest neighbors in the feature space. The arcs are weighted by $d(s, t)$ and the nodes $s \in \mathcal{N}$ are weighted by a density value $\rho(s)$, given by Equation 4:

$$
\rho(s) = \frac{1}{\sqrt{2\pi\sigma^2}|\mathcal{A}(s)|} \sum_{\forall t \in \mathcal{A}(s)} \exp\left(\frac{-d^2(s, t)}{2\sigma^2}\right),
\tag{4}
$$

where $|\mathcal{A}(s)| = k$, $\sigma = \frac{d_f}{3}$, and d_f is the maximum arc weight in $(\mathcal{N}, \mathcal{A})$. Using these parameters, all nodes are considered for density computation, since a Gaussian function covers most samples within $d(s, t) \in [0, 3\sigma]$.

A standard method to compute a probability density function (p.d.f.) is the Parzen-window. Equation (4) provides a Parzen-window estimation based on an isotropic Gaussian kernel when the arcs $(s, t) \in \mathcal{A}$ if $d(s, t) \leq d_f$ are defined. Nevertheless, this approximation causes differences in scale and sample concentration.

An interesting way to solve this problem is to choose d_f based on a particular region of the feature space [?]. By considering the k-nearest neighbors, it is possible to handle different concentrations and transform the scale problem into finding the best k value within $[1, k_{\max}]$, for $1 \leq k_{\max} \leq |\mathcal{N}|$. An approach proposed by Rocha et al. [?] considers the minimum graph cut (best k) according to a measure suggested by Shi et al. [30].

Moreover, let a path π_t be the sequence of adjacent samples starting from a root $R(t)$ and ending at a sample t, being $\pi_t = \langle t \rangle$ a trivial path and $\pi_s \cdot \langle s, t \rangle$ the concatenation of π_s and arc (s, t). Among all possible paths π_t with roots on the maxima of the p.d.f., the problem lies in finding a path with the lowest density value. Each path defines an influence zone (cluster) by selecting strongly connected samples. Mathematically speaking, Equation 5 maximizes $f(\pi_t)$ for all $t \in \mathcal{N}$ where:

$$f(\langle t \rangle) \quad = \quad \left\{ \begin{array}{ll} \rho(t) & \text{if } t \in \mathcal{R} \\ \rho(t) - \delta & \text{otherwise} \end{array} \right.$$

and

$$f(\langle \pi_s \cdot \langle s, t \rangle \rangle) \quad = \quad \min\{f(\pi_s), \rho(t)\}, \tag{5}$$

where $\delta = \min_{\forall(s,t) \in \mathcal{A}|\rho(t) \neq \rho(s)} |\rho(t) - \rho(s)|$ and $\mathcal{R}$ is a root with one element set for each maximum of the p.d.f.. One can see that higher values of δ reduces the number of maxima. Additionally, in this library, we are using $\delta = 1.0$ and $\rho(t) \in [1, 1000]$.

Finally, the OPF algorithm maximizes $f(\pi_t)$ such that the optimum paths compose an Optimum-Path Forest, i.e., a predecessor no-cycling map P which assigns to each sample $t \notin \mathcal{R}$ its predecessor $P(t)$ from the optimum path $\mathcal{R}$ or a marker nil when $t \in \mathcal{R}$. Each p.d.f. maximum (prototype) is the root of an OPT, commonly known as a cluster. Furthermore, the collection of all OPTs is the so-called Optimum-Path Forest.

4 OPFython

OPFython is distributed among several packages, each one being accountable for particular classes and methods. Figure 2 represents a summary of OPFython's architecture, while the next sections present each one of its packages within more details.
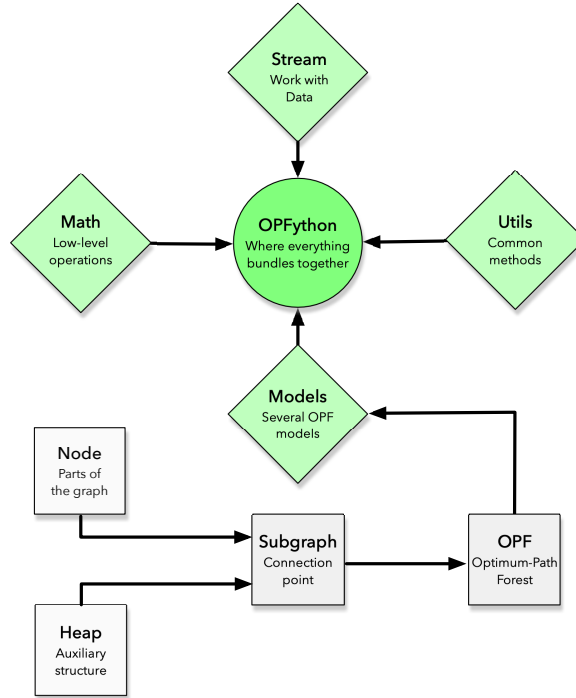
Figure 2: Flowchart of OPFython's architecture.

4.1 Core

The core package serves as the origin of all OPFython's sub-classes. It assists as a building base for implementing more appropriate structures that one may require when creating an Optimum-Path Forest-based classifier. As portrayed in Figure 3, four modules compose the core package, as follows:

- **Heap:** The heap assists OPF in stacking nodes' according to their costs and further unstacking them to build the subgraph;

- **Node:** When working with graph-based structures, each of their pieces is represented by a node. In OPFython, we use the node structure to store valuable information of a sample, such as its features, label and other information that OPF might need;

- **OPF:** The OPF class serves as the classifier itself. It implements some basic methods that are common to its children, as well as some methods that assist users in saving and loading pre-trained models;

- **Subgraph:** The subgraph is one of the most fundamental structures of the OPF classifier. As it is a graph-based classifier, it uses nodes and arcs to build the optimum-path costs and find the prototype nodes, which conquer the remaining samples and propagate their labels.
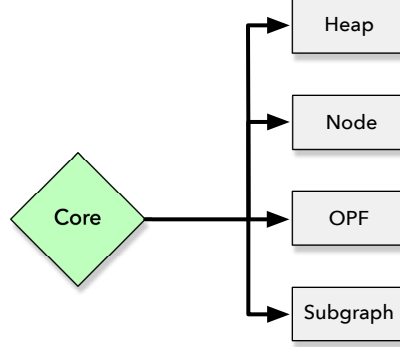
Figure 3: Flowchart of OPFython's core package.

4.2 Math

In order to ease the user's life, OPFython offers a mathematical package, containing low-level math implementations, illustrated by Figure 4. Naturally, some repeated functions that are used throughout the library are represented in this package, as follows:

- **Distance:** A distance metric is used to calculate the cost between nodes. Hence, we offer a variety of distance metrics that fulfills every task needs;

- **General:** Common-use functions that do not have a special division are defined in this module;

- **Random:** Lastly, some methods might use random numbers for sampling or setting a heuristic. This module can generate uniform and Gaussian random numbers.
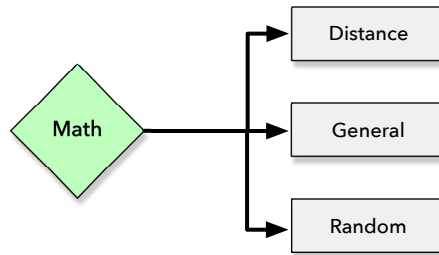
Figure 4: Flowchart of OPFython's math package.

4.3 Models

There are several approaches to be conducted when designing an optimum-path forest classifier, such as supervised, unsupervised, semi-supervised, among others. Therefore, the models' package provides classes and methods that compose these high-level abstractions and implement the classifying strategies. Currently, OPFython offers four types of classifiers, which are illustrated by Figure 5 and described as follows:

- **KNNSupervisedOPF [21]:** A supervised Optimum-Path Forest classifier that uses a KNN-based subgraph, providing a more effective way to build up the connectivity subgraph.
- **SemiSupervisedOPF [31]:** A semi-supervised Optimum-Path Forest classifier, which is extremely useful in labeling unknown samples.
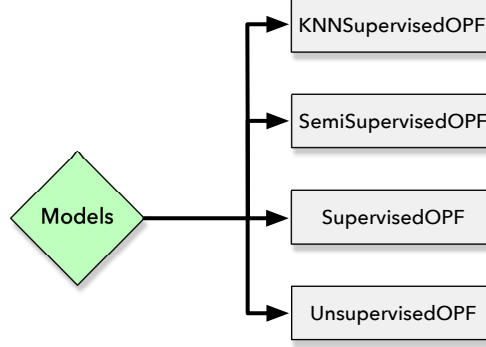- **SupervisedOPF [17]:** The classical supervised Optimum-Path Forest classifier, which is suitable for training on labeled datasets and performing new predictions;
- **UnsupervisedOPF [32]:** The standard unsupervised Optimum-Path Forest classifier, which is suitable for clustering unlabeled datasets.

Figure 5: Flowchart of OPFython's models package.

4.4 Stream

The stream package deals with every pre-processing step of the input data. Essentially, it is responsible for loading the data, parsing it into samples and labels, and splitting it into new sets, such as training, validation, and testing. Figure 6 depicts its modules, as well as we provide a brief description of them as follows:

- **Loader:** A loading module that assists users in pre-loading datasets. Currently, it is possible to load files in .txt, .csv and .json formats;
- **Parser:** After loading the files, it is necessary to parse the pre-loaded arrays into samples and labels;
- **Splitter:** Finally, if necessary, one can split the loaded and parsed dataset into new sets, such as training, validation, and testing.
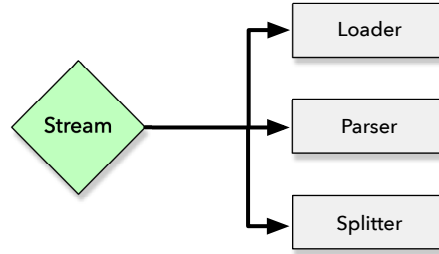
Figure 6: Flowchart of OPFython's stream package.

4.5 Subgraphs

As mentioned before, the subgraph is one of the essential structures of the classification process. Nevertheless, one can observe that distinct classifiers might need distinct subgraphs. Therefore, we are glad to offer additional subgraphs implementations as portrayed by Figure 7 and described as follows:

- **KNNSubgraph:** When dealing with KNN-based classifiers, it is crucial to use a KNN-based subgraph, as it implements some additional functions that the classifier might need.
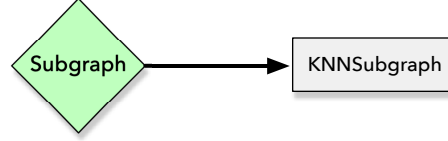
Figure 7: Flowchart of OPFython's subgraphs package.

4.6 Utils

A utility package implements standard tools that are shared over the library, as it is a better approach to implement once and re-use them across other modules, as shown in Figure 8. This package implements the subsequent modules:

- **Constants:** Constants are fixed numbers that do not alter throughout the code. For the sake of easiness, they are implemented in the same module;
- **Converter:** Most of OPF users are familiarized to the specific file format it uses. Hence, we implement an own module that is capable of converting .opf files into .txt, .csv, and .json.
- **Exception:** In order to assist users, the exception module implements common errors and exceptions that might happen when invalid arguments are used in OPFython classes and methods;
- **Logging:** Every method that is invoked in the library is logged onto a log file. One can watch the log in order to detect potential errors, essential warnings, or even success messages throughout the classification procedure.
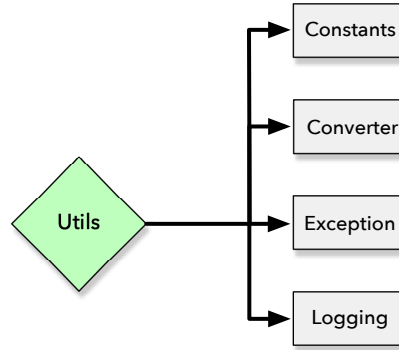
Figure 8: Flowchart of OPFython's utils package.

5 Library Usage

In this section, we describe how to install the OPFython library, as well as the first steps to start playing with it. Essentially, one can study its documentation or make use of the already-included examples. Besides, there are implemented methods that conduct unitary tests and verify if everything is operating as presumed.

5.1 Installation

First of all, we understand that everything has to be smooth without being tricky or daunting. Therefore, OPFython will always be the one-to-go package, from the very first installation to its further usage. Just execute the following command under the most preferred Python environment (standard, conda, virtualenv):

```
pip install opfython
```

Alternatively, it is possible to use the bleeding-edge version by cloning its repository and installing it:

```
git clone https://github.com/gugarosa/opfython.git
```

```
pip install .
```

Note that there is no other requirement to use OPFython. As its single dependency is the Numpy package, it can be installed everywhere, despite the machine's operational system.

5.2 Documentation

One might have an enthusiasm for mastering the concepts and strategies behind OPFython. Hence, we provide a fully documented reference[5] containing everything that the library offers. From elementary classes to more complex methods, OPFython's documentation is the perfect reference to learn how the library was developed or even improve it with contributions.

5.3 Classes and Methods Examples

Additionally, in the `examples/` folder, we provide example scripts for all packages that the library implements, such as:

- **Core:** `create_heap.py`, `create_node.py`, `create_opf.py`, `create_subgraph.py`;
- **Math:** `calculate_distances.py`, `general_purpose.py`, `generate_random_numbers.py`;
- **Models:** `create_knn_supervised_opf.py`, `create_semi_supervised_opf.py`, `create_supervised_opf.py`, `create_unsupervised_opf.py`;
- **Stream:** `load_file.py`, `parse_loaded_file.py`, `split_data.py`;
- **Subgraphs:** `create_knn_subgraph.py`.
- **Utils:** `convert_from_opf.py`;

Each example is constituted of high-level explanations of how to use predefined classes and methods. One can observe that it provides a standard description of how to instantiate each class and decide which arguments should be employed.

5.4 Test Suites

OPFython is prepared with tests to give a more in-depth analysis of the code. Also, the intention behind any test is to check whether everything is running as demanded or not. Thus, there are two main methods in order to execute the tests:

- **PyTest:** The first method is running the solo command `pytest tests/`, as depicted by Figure 9. It will fulfill all the implemented tests and return an output indicating whether they succeeded or failed;
- **Coverage:**, An interesting extension to PyTest is the coverage module. Despite granting the same outputs from PyTest, it will also present a report that states how much the tests cover the code, as illustrated by Figure 10. Its usage is also straightforward: `coverage run -m pytest tests/` and `coverage report -m`.

6 Applications

In this section, we explain how to perform a classification task with OPFython, as well as briefly describe the seven pre-loaded applications that are included with the library.

6.1 Getting Started

After installing the OPFython library, it is straightforward to use its packages, where seven elementary examples show key features that the library implements. One can refer to the `examples/applications` folder and examine the following files:

- **KNN-based supervised OPF training:** `knn_supervised_opf_training.py`;
- **Semi-supervised OPF:** `semi_supervised_opf_training.py`;
- **Supervised OPF agglomerative learning:** `supervised_opf_agglomerative.py`;
- **Supervised OPF learning:** `supervised_opf_learning.py`;
- **Supervised OPF pruning:** `supervised_opf_pruning.py`;
- **Supervised OPF training:** `supervised_opf_training.py`;
- **Unsupervised OPF clustering:** `unsupervised_opf_clustering.py`.

---

[5]https://opfython.readthedocs.io

Figure 9: Example of running tests with PyTest.

Each example is comprised of the following pipeline: loading the dataset, parsing the dataset, splitting the dataset, instantiating a classifier, fitting the training data, predicting the validation/test data, and calculating the classifier's performance. Finally, after performing the classification process, it is possible to save the model in a disk-file for further inspection. Figure 11 illustrates the output logs generated by an OPFython classification.

The difference between the provided scripts consists of the type of classifier. While supervised classifications attempt to learn a set of classes from particular samples that represent them, the unsupervised classification tries to aggregate samples into clusters, i.e., dense regions where samples share some similar traits. As for now, we offer three supervised classifications, e.g., KNN-based supervised OPF, semi-supervised OPF and supervised OPF, as well as, an unsupervised classification, the unsupervised OPF. Additionally, we offer three extensions of the supervised OPF, e.g., supervised OPF agglomerative learning (learns from mistakes over the validation set), supervised OPF learning (learns the best classifier over a validation set), and supervised OPF pruning (prunes nodes while maintaining the accuracy).

6.2 Modeling a New Classification

In order to model a new classification, some conventional rules need to be comprehended. First of all, the data should be loaded and parsed, which in this case, we will be loading a common dataset known as Boat:

```python
import opfython.stream.loader as l
import opfython.stream.parser as p

# Loading a .txt file to a numpy array
txt = l.load_txt('data/boat.txt')

# Parsing a pre-loaded numpy array
X, Y = p.parse_loader(txt)
```
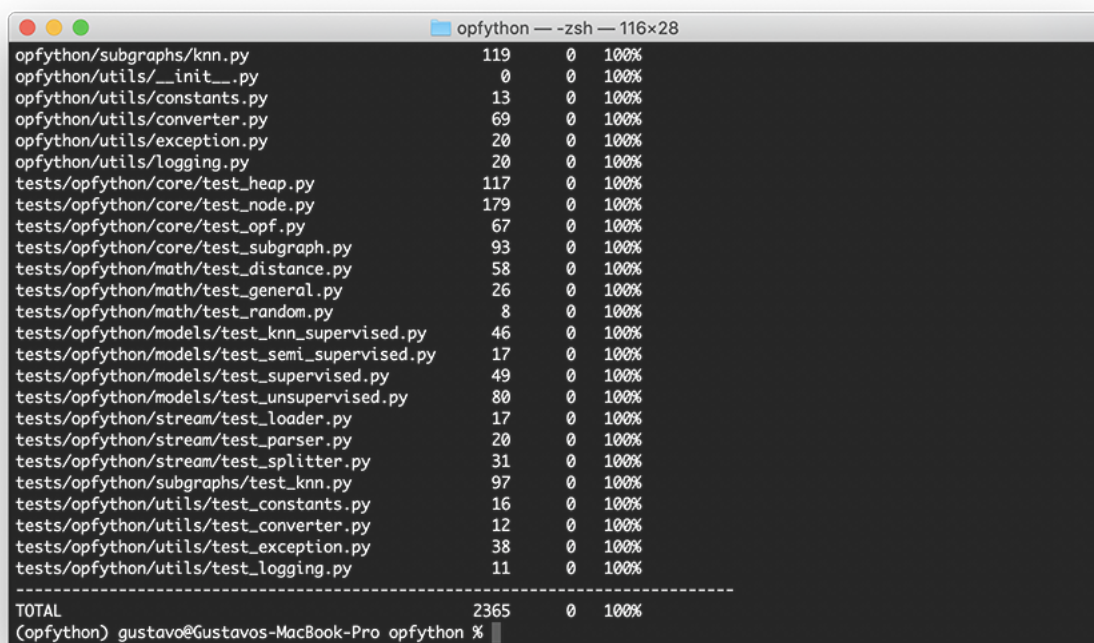
Furthermore, if necessary, we can split the data into new sets, such as training and testing, as follows:

```python
import opfython.stream.splitter as s
```

10

```
● ● ●                           ▢ opfython — -zsh — 116×28
opfython/subgraphs/knn.py                           119      0     100%
opfython/utils/__init__.py                            0      0     100%
opfython/utils/constants.py                          13      0     100%
opfython/utils/converter.py                          69      0     100%
opfython/utils/exception.py                          20      0     100%
opfython/utils/logging.py                            20      0     100%
tests/opfython/core/test_heap.py                    117      0     100%
tests/opfython/core/test_node.py                    179      0     100%
tests/opfython/core/test_opf.py                      67      0     100%
tests/opfython/core/test_subgraph.py                 93      0     100%
tests/opfython/math/test_distance.py                 58      0     100%
tests/opfython/math/test_general.py                  26      0     100%
tests/opfython/math/test_random.py                    8      0     100%
tests/opfython/models/test_knn_supervised.py         46      0     100%
tests/opfython/models/test_semi_supervised.py        17      0     100%
tests/opfython/models/test_supervised.py             49      0     100%
tests/opfython/models/test_unsupervised.py           80      0     100%
tests/opfython/stream/test_loader.py                 17      0     100%
tests/opfython/stream/test_parser.py                 20      0     100%
tests/opfython/stream/test_splitter.py               31      0     100%
tests/opfython/subgraphs/test_knn.py                 97      0     100%
tests/opfython/utils/test_constants.py               16      0     100%
tests/opfython/utils/test_converter.py               12      0     100%
tests/opfython/utils/test_exception.py               38      0     100%
tests/opfython/utils/test_logging.py                 11      0     100%
-----------------------------------------------------------------------
TOTAL                                              2365      0     100%
(opfython) gustavo@Gustavos-MacBook-Pro opfython %
```

Figure 10: Example of code coverage tests with Coverage.

```python
# Splitting data into training and testing sets
X_train, X_test, Y_train, Y_test = s.split(X, Y, percentage=0.5,
    random_state=1)
```

Afterward, we can instantiate an OPF classifier:

```python
from opfython.models.supervised import SupervisedOPF

# Creates a SupervisedOPF instance
opf = SupervisedOPF(distance='log_squared_euclidean',
    pre_computed_distance=None)
```

Finally, we can fit the classifier and perform new predictions:

```python
# Fits training data into the classifier
opf.fit(X_train, Y_train)

# Predicts new data
preds = opf.predict(X_test)
```

After predicting new samples, it is possible to evaluate the classifier's performance:

```python
import opfython.math.general as g

# Calculating accuracy
acc = g.opf_accuracy(Y_test, preds)
```

11

Figure 11: Output logs generated by executing an OPFython classification.

7   Conclusions

In this article, we introduce an open-source Python-inspired library for handling Optimum-Path Forest classifiers, known as OPFython. Based upon an object-oriented paradigm, OPFython provides a modern yet straightforward implementation, allowing users to prototype new OPF-based classifiers swiftly.

The library implements a wide variety of Optimum-Path Forest classifiers, such as supervised, semi-supervised, and unsupervised ones, as well as auxiliary functions that assist the classifiers' workflow, i.e., distance functions, classification metrics, data processing, errors logging. Additionally, as the original LibOPF thoroughly inspires OPFython's library, it is possible to use the same loading format (OPF file format) and methods that are available in the original package. Furthermore, OPFython provides a model-saving method, which can be used to pre-train classifiers and retrieve insightful information about the classification procedure.

Regarding future works, we intend to make available more OPF-based classifiers, as well as a visualization package, which will allow users to feed their saved models and furnish charts. Furthermore, we aim at improving our implementations by distributing the calculations, i.e., employ a parallel computing concept, which will hopefully reduce our computational burden.

Acknowledgments

References

[1] S. J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[2] D. Acemoglu and P. Restrepo. Artificial intelligence, automation and work. Technical report, National Bureau of Economic Research, 2018.

[3] C. M. Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[4] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge university press, 2008.

[5] S. Li, L. Da Xu, and S. Zhao. The internet of things: a survey. *Information Systems Frontiers*, 17(2):243–259, 2015.

[6] S. Shalev-Shwartz, S. Shammah, and A. Shashua. On a formal model of safe and scalable self-driving cars. *arXiv preprint arXiv:1708.06374*, 2017.

[7] J. W. Shavlik, R. J. Mooney, and G. G. Towell. Symbolic and neural learning algorithms: An experimental comparison. *Machine learning*, 6(2):111–143, 1991.

[8] P. Langley. The changing science of machine learning. *Machine Learning*, 82(3):275–279, 2011.

[9] N. Sebe, I. Cohen, A. Garg, and T. S. Huang. *Machine learning in computer vision*, volume 29. Springer Science & Business Media, 2005.

[10] N. Indurkhya and F. J. Damerau. *Handbook of natural language processing*. Chapman and Hall/CRC, 2010.

[11] S. B. Kotsiantis. *Supervised machine learning: A review of classification techniques*. 2007.

[12] T. Hastie, R. Tibshirani, and J. Friedman. Unsupervised learning. In *The elements of statistical learning*, pages 485–585. Springer, 2009.

[13] J. H. Friedman. Data mining and statistics: What's the connection? *Computing Science and Statistics*, 29(1):3–9, 1998.

[14] S. Sra, S. Nowozin, and S. J. Wright. *Optimization for machine learning*. Mit Press, 2012.

[15] D. Bzdok, N. Altman, and M. Krzywinski. Points of significance: statistics versus machine learning, 2018.

[16] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[17] J. P. Papa, A. X. Falcão, and C. T. N. Suzuki. Supervised pattern classification based on optimum-path forest. *International Journal of Imaging Systems and Technology*, 19(2):120–131, 2009.

[18] C.-C. Chang and C.-J. Lin. Libsvm: A library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):1–27, 2011.

[19] J. P. Papa, A. X. Falcão, V. H. C. Albuquerque, and J. M. R. S. Tavares. Efficient supervised optimum-path forest classification for large datasets. *Pattern Recognition*, 45(1):512–520, 2012.

[20] G. H. Rosa, K. A. P. Costa, L. A. P. Júnior, J. P. Papa, A. X. Falcão, and J. M. R. S. Tavares. On the training of artificial neural networks with radial basis function using optimum-path forest clustering. In *2014 22nd International Conference on Pattern Recognition*, pages 1472–1477, Aug 2014.

[21] J. P. Papa and A. X. Falcão. A learning algorithm for the optimum-path forest classifier. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 195–204. Springer, 2009.

[22] D. Rodrigues, L. A. M. Pereira, R. Y. M. Nakamura, K. A. P. Costa, X.-S. Yang, A. N. Souza, and J. P. Papa. A wrapper approach for feature selection based on bat algorithm and optimum-path forest. *Expert Systems with Applications*, 41(5):2250 – 2258, 2014.

[23] P. A. V. Miranda and A. X. Falcão. Links between image segmentation based on optimum-path forest and minimum cut in graph. *Journal of Mathematical Imaging and Vision*, 35(2):128–142, Oct 2009.

[24] F. A. M. Cappabianco, A. X. Falcão, C. L. Yasuda, and J. K. Udupa. Brain tissue mr-image segmentation via optimum-path forest clustering. *Computer Vision and Image Understanding*, 116(10):1047 – 1059, 2012.

[25] T. M. Nunes, A. L. V. Coelho, C. A. M. Lima, J. P. Papa, and V. H. C. de Albuquerque. Eeg signal classification for epilepsy diagnosis via optimum path forest – a systematic assessment. *Neurocomputing*, 136:103 – 123, 2014.

[26] E. J. da S. Luz, T. M. Nunes, V. H. C. de Albuquerque, J. P. Papa, and D. Menotti. Ecg arrhythmia classification based on optimum-path forest. *Expert Systems with Applications*, 40(9):3561 – 3573, 2013.

[27] C. C. O. Ramos, A. N. de Sousa, J. P. Papa, and A. X. Falcão. A new approach for nontechnical losses detection based on optimum-path forest. *IEEE Transactions on Power Systems*, 26(1):181–189, Feb 2011.

[28] S. E. N. Fernandes, D. R. Pereira, C. C. O. Ramos, A. N. Souza, D. S. Gastaldello, and J. P. Papa. A probabilistic optimum-path forest classifier for non-technical losses detection. *IEEE Transactions on Smart Grid*, 10(3):3226–3235, May 2019.

13

[29] J. P. Papa, A. X. Falcão, and C. T. N. Suzuki. *LibOPF: A library for the design of optimum-path forest classifiers*, 2015. Software version 3.1 available at http://www.ic.unicamp.br/ afalcao/.

[30] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[31] W. P. Amorim, A. X. Falcão, and M. H. de Carvalho. Semi-supervised pattern classification using optimum-path forest. In *2014 27th SIBGRAPI Conference on Graphics, Patterns and Images*, pages 111–118. IEEE, 2014.

[32] L. M. Rocha, F. A. M. Cappabianco, and A. X. Falcão. Data clustering as an optimum-path forest problem with applications in image analysis. *International Journal of Imaging Systems and Technology*, 19(2):50–68, 2009.