Implementation of Self-Organizing Maps with Python

Li Yuan
University of Rhode Island, li_yuan@uri.edu

IMPLEMENTATION OF SELF-ORGANIZING MAPS WITH PYTHON

BY

LI YUAN

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

2018

MASTER OF SCIENCE THESIS

OF

LI YUAN

APPROVED:

Thesis Committee:

Major Professor	Lutz Hamel
	Natallia Katenka
	Austin Humphries
	Nasser H. Zawia
	DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND
2018

ABSTRACT

As a member of Artificial Neural Networks, Self-Organizing Maps (SOMs) have been well researched since 1980s, and have been implemented in C, Fortran, R [1] and Python [2]. Python is an efficient high-level language widely used in the machine learning field for years, but most of the SOM-related packages which are written in Python only perform model construction and visualization. However, the POPSOM package, written in R, is capable of performing functionality beyond model construction and visualization, such as evaluating the model's quality with statistical methods and plotting marginal probability distributions of the neurons. In order to give the Python user the POPSOM package's advantages, it is important to migrate the POPSOM package to be Python-based. This study shows the details of this implementation.

There are three major tasks for the implementation: 1) Migrate the POPSOM package from R to Python; 2) Refactor the source code from procedural programming paradigm to object-oriented programming paradigm; 3) Improve the package by adding normalization options to the model construction function. In addition to constructing the model in Python, Fortran is also embedded to accelerate the speed of model construction significantly in this project.

The final program has been completed, and it is necessary to guarantee the correctness of the program. The best way to achieve this goal is to compare the output of the Python-based program to the output generated by the R-based program. For the model construction function, the SOM algorithm initializes the weight vector of the neurons randomly at the very beginning, and then selects the input vectors randomly

during the training. Due to these two random factors, one cannot expect the same input (data set) will result in exactly the same output (neurons). Instead, to give evidence that the Python program is working properly, there are two solutions that have been proposed and applied in this project: 1) measuring the average difference of vectors between two neurons which have been generated by the R and Python functions respectively; 2) measuring the ratio of the variances and the difference of features' mean for the two neurons. Besides the model construction, model visualization and other functions which take neurons as their input should return the same results by feeding the same input (neurons). The detail of above verification will be represented in the following chapters.

ACKNOWLEDGMENTS

I would like to acknowledge many individuals for helping me during my master's study at University of Rhode Island. Particularly, I would like to express my sincere gratitude to my advisor Dr. Lutz Hamel to provide the opportunity, knowledge and support that enabled this study. I would also like to thank him for his continuous support, patience and guidance throughout the past two years.

I would also like to thank my committee members Dr. Natallia Katenka, Dr. Austin Humphries and Dr. Orlando Merino for generously offering their time, suggestions and goodwill throughout the preparation and review of this document.

Thanks to Lorraine Berube, secretary of the Computer Science and Statistics department, for her never-ending encouragement and helpfulness.

A special thanks to my family – my wife Jieying, and my daughter Helena – for their love and support. I would also like to thank my parents and parents-in-law for their love and prayers.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Dimensionality reduction has been an important topic within the data analysis community for some time. Several solutions have been proposed by researchers, one of which is Principal Component Analysis (PCA), a statistical procedure based on orthogonal transformation. It has been used as a tool in exploratory data analysis and the creation of predictive models. In the 1980s, another approach for dimensionality reduction was proposed by T. Kohonen [3] known as Self-Organizing Maps (SOMs), a type of neural network for the visualization of high-dimensional data. Typically, the SOM graphic represents [4] the high-dimensional input data with a 2-D grid map. This type of map preserves the topology and neighborhood relationship of the input space [5]. Additionally, indicated by [3], the convergence of the model is guaranteed after a certain amount of iterations.

The SOM algorithm has been implemented by C, R, Fortran and Python [6]-[8]. To date, there are more than 100 packages available on the GitHub community. Although the number of packages is sufficient and continually increasing, the functionalities that are provided by these packages are quite similar. Most of these packages only focus on model construction and model visualization. Few of them touch on the aspect of evaluating the model's quality. POPSOM, an R package [8] developed and maintained by Dr. Lutz Hamel and his former students, not only provides the model construction and model visualization as other packages, but also provides a set of functions for evaluating the model's quality and visualizing the marginal probability distribution of each feature. The purpose of this project is to migrate the POPSOM package from R to Python so that researchers in the Python community may utilize it in their research.

A Self-Organizing Map (SOM) is a specific type of Artificial Neural Network whose purpose is to reduce the dimension of the input space. The resulting map is a graphical representation easily interpreted by the end user [4]. From a practical point of view, the SOM's program package should include at least the following three main functions: 1) model construction, 2) model evaluation, and 3) model visualization.

As the most important part of the SOM, the model construction algorithm has been proposed by [3]. The basic idea of this algorithm is described in the following two major steps:

1) Initiate the weight vector (or neuron) randomly.
2) Update the weight vector using the following formula with a certain number of iterations.

$$m_i(t+1) = m_i(t) + h_{ci}[(x(t) - m_i(t)] \tag{1}$$

$h_{ci}(t)$: neighborhood function, when $t \rightarrow 0$, $h_{ci}(t) \rightarrow 0$

$$h_{ci}(t) = \begin{cases} \alpha(t), & i \in Nc \\ 0, & i \notin Nc \end{cases} \tag{2}$$

N_c: neighborhood set

$\alpha(t)$: learning-rate factor which can be linear, exponential or inversely proportional.

Secondly, the model evaluation function is designed to help users determine the appropriateness of the model after each training. Many quality measures have been proposed to evaluate the quality of the resulting map [4]. Most of them either focus on one aspect of a SOM or on the computational expense [9]. In 2017, Dr. Lutz Hamel proposed an efficient statistical approach [9] to measure both the map embedding accuracy (or convergence) and the estimated topographic accuracy of the model. This approach has been since implemented in the R-based POPSOM package [8].

Most of packages available on GitHub only represent the resulting maps as heat maps, while the R-based POPSOM package provides users with three kinds of graphical reports: 1) the significance of each feature with respect to the self-organizing map model (Figure 1), 2) the starburst representation of the SOM model (Figure 2), and 3) the marginal probability distribution of the neurons and data (Figure 3).

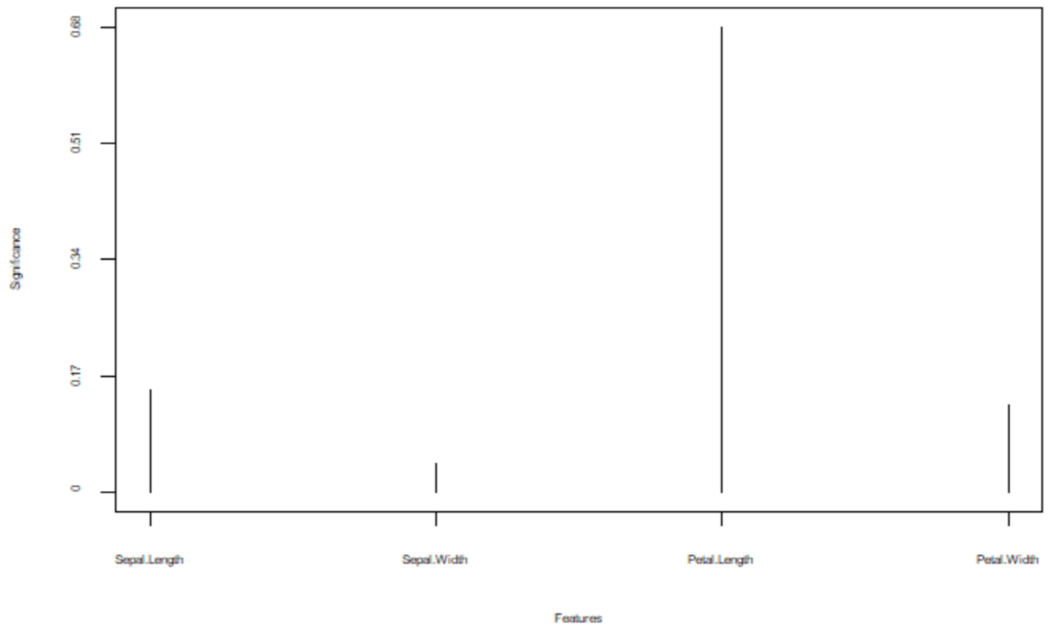


Figure 1. The significance of Iris dataset's individual features with respect to the self organizing map model.

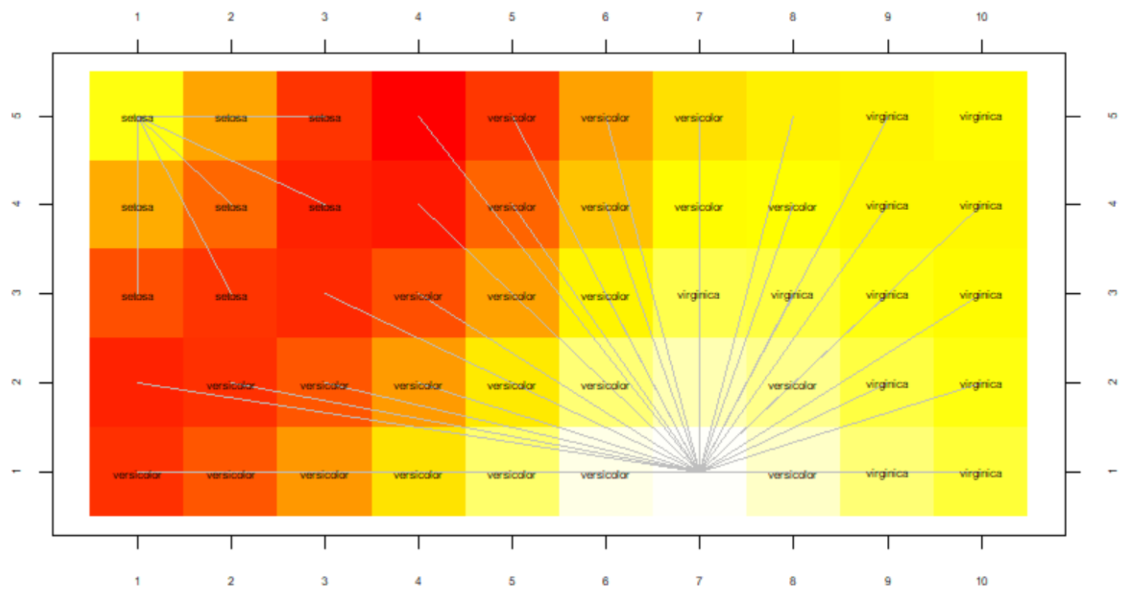


Figure 2. The starburst representation of the Self-Organizing Maps Model for Iris dataset. The centers of the starbursts are the centers of the clusters.

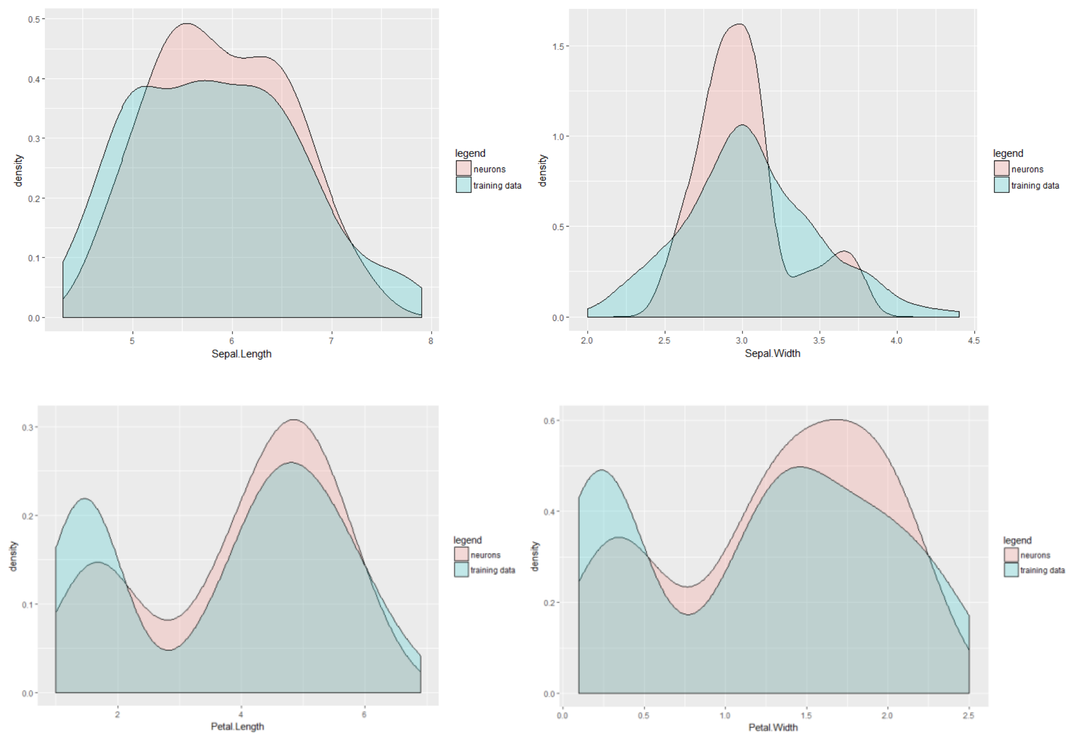


Figure 3. Density plots showing the marginal probability distributions of Iris data set's dimensions overlaid with the neuron density for each dimension respectively.

As a kind of pre-processing method, standard normalization is not necessary for the model training, but it may improve the map embedding accuracy [3] within the SOM algorithm.

The R-based POPSOM package has been developed and maintained since 2013. The latest version is 4.2, updated last on May, 31st 2017 [8]. R users are able to explore their data in a more detailed fashion using various aspects of the POPSOM package. Unfortunately, Python users are currently limited to model construction and visualization without the benefit of evaluating the model quality in regard to precise convergence characteristics. To benefit the researchers within the Python community, this project's purpose is to migrate the POPSOM package from R to Python.

Both R and Python are widely used in data analysis, and they have a few functions that share similar features. For this project, to successfully migrate the package from R to Python, three features need to be migrated: 1) naming rules, 2) mathematical and statistical functions and 3) data manipulation functions. Most of the functions in R already have counterparts in Python, but there are still some functions that are only available in R, such as *t.test* (produces a variety of t-tests), *var.test* (performs an F-test to compare the variances of two samples) and *smooth_2d* (performs kernel smoother for irregular 2-D data). To migrate these functions, they must be rewritten in Python from scratch.

Migrating all of the functionalities in the package from R to Python has been the basic goal of this project. Besides preserving all the functionalities of the R-based package, the following improvements have also been made: 1) refactoring the procedural programming paradigm to object-oriented programming paradigm, 2) addition of normalization as an optional argument for model initialization (or instantiation), and 3) Fortran embeddedness as another option for model training.

Finally, the Fisher/Anderson Iris data set [10] and Wheat Seed data set [11] from the UCI machine learning repository were utilized to evaluate the correctness of the Python-based package. The reasoning stands that if the same input data is imputed into both the R-based and Python-based packages and if the Python-based package is working correctly, then both packages will return the same outcome. Since the SOM algorithm initializes the weight vector randomly [3] at the beginning of model training and selects the vector randomly during the training, even the same input data set will return a different outcome (neurons) for a different training by one package. Thus, it is not feasible to evaluate the correctness of the Python program by measuring the difference in the two neurons directly. In order to achieve this goal, two statistical measurements have been proposed and applied in this project. 1) When measuring the average difference of vectors between the two neurons, the result should be closed to 0 at the end of the training if the two neurons are drawn from the same input data space. 2) The ratio of the variances and the difference of features' means for both neurons are evaluated. The ratio of the variances should be approximately equal to 1 and the difference of features' means should be close to 0. Each of these respective values should fall within the chosen computed confidence interval as appropriate if the two neurons are drawn from the same input data space. Besides the model construction, model visualization and other functions which take neurons as their input should return the same results as well by feeding the same neurons.

CHAPTER 2

Literature Review

2.1 Self-Organizing Maps

Since the dawn of the data era, more and more efficient data analysis technologies have been researched, proposed and applied at a very fast pace, especially tools for statistical analysis for high-dimensional data (data with multiple features). Self-Organizing Maps (SOMs) proposed by [3] are considered effective tools for the visualization of high-dimensional data [3]. The SOM algorithm is used to compress the information to produce a similarity graph while preserving the topologic relationship of the input data space. The convergence of the SOM has been previously discussed and guaranteed [3].

The basic SOM model construction algorithm can be interpreted as follows:

1) Create and initialize a matrix (weight vector) randomly to hold the neurons. If the matrix can be initialized with order and roughly compiles with the input density function, the map will converge quickly [3];

2) Read the input data space. For each observation (instance), use the optimum fit approach, which is based on the Euclidean distance

$$c = \underset{i}{arg\min} \, || \, x - m_i \, || \tag{3}$$

to find the neuron which best matches this observation. Let x denote the training vector from the observation and m_i denote a single neuron in the matrix. Update that neuron to resemble that observation using the following equation:

$$m_i(t+1) = m_i(t) + h_{ci}(t)[x(t) - m_i(t)] \tag{4}$$

$m_i(t)$: the weight vector before the neuron is updated.

$m_i(t+1)$: the weight vector after the neuron is updated.

$x(t)$: the training vector from the observation.

$h_{ci}(t)$: the neighborhood function (a smoothing kernel defined over the lattice points), defined though the following equation:

$$h_{ci}(t) = \begin{cases} \alpha(t), & i \in N_c \\ 0, & i \notin N_c \end{cases} \tag{5}$$

N_c: the neighborhood set, which decreases with time.

$\alpha(t)$: the learning-rate factor which can be linear, exponential or inversely proportional. It is a monotonically decreasing function of time (t).

3) Update the immediate neighborhood of that neuron accordingly (Figure 4).

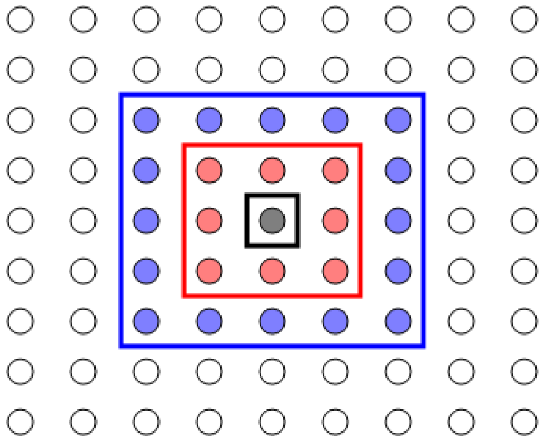


Figure 4. Neighborhoods (N_c) for a rectangular matrix of cluster units: $N_c = 0$ in black brackets, $N_c = 1$ in red, and $N_c = 2$ in blue.

As proposed by Cheng [12], after running this algorithm with a sufficient number of iterations, the map will ultimately converge. However, it is difficult for users to determine how many iterations are sufficient. Another practice measure is evaluating the map's quality, which can help users determine the optimal number of iterations.

2.2 Evaluation of the Quality of the Map

It is necessary to ensure that the model obtained from training is already well-converged and reliable. In other words, the quality of the SOMs need to be measured first before any further operation, such as visualization, is employed. Recently, many different quality measures of SOMs have been proposed and argued [13], [14]. However, most of them either measure only one aspect of a SOM or are computationally expensive. Some include both of these drawbacks [9]. Based on map embedding accuracy and estimated topographic accuracy, Dr. Hamel proposed a population-based [15] computationally efficient statistical approach [9] to evaluate the quality of a SOM model. This approach is based on two populations (one from the training data set and the other from the neuron of the map) and evaluates the quality of a SOM by the measure (or magnitude) of the convergence index, which is the linear combination of the map embedding accuracy (convergence) and the estimated topographic accuracy.

The map embedding accuracy, derived from the theory proposed by Yin and Allison [12], is limited in that the neurons of a SOM will converge on the probability distribution of the training data [12].

$$ea = \frac{1}{d}\sum_{i=1}^{d} \rho_i$$

where

$$\rho_i = \begin{cases} 1 & \text{if feature i is embeded,} \\ 0 & \text{otherwise,} \end{cases}$$

The computational complexity of the embedding accuracy is

$$O((n+m) \times d)$$

where *n* is the number of observations in the training data, *m* is the number of neurons, and *d* is the number of features in the training data. Without any exponential function, the above equation indicates this computation is efficient in most cases (where $d \ll n, and\ d \ \ll m$). Although the embedding accuracy measures the same thing as quantization error, it confers the advantage of indicating when statistically there is no difference between two populations (training data and neurons).

Topographic error [9] can be defined as:

$$te = \frac{1}{n}\sum_{i=0}^{n} err(x_i)$$

Where

$$err(x_i) = \begin{cases} 1 & \text{if } bmu(x_i) \text{ and } 2bmu(x_i) \text{ are not neighbors.} \\ 0 & \text{otherwise,} \end{cases}$$

where *n* is the number of observations in the training data, x_i is the *i*th observation in the training data, and $bmu(x_i)$ and $2bmu(x_i)$ (*bmu* stands for the best matching unit) represent the best-matching and second best-matching unit for the training vector x_i. Accordingly, the topographic accuracy could be defined as:

$$ta = 1 - te$$

Computing the topographic accuracy is a time-consuming task, especially for a large data set. To make this computation more efficient and practical, Dr. Hamel proposed utilizing a sample of the training data, a smaller subset of all the training data, to estimate the topographic error.

$$te' = \frac{1}{s}\sum_{i=0}^{s} err(x_i)$$

The estimated topographic accuracy [9] is defined as follows:

$$ta' = 1 - te'$$

The values of the map embedding accuracy and estimated topographic accuracy are numbers between 0 and 1. If the value is equal to 1, then one can interpret that the map has converged well or is fully organized. Dr. Hamel proposed to use the convergence index as defined by:

$$cix = \frac{1}{2}\,ea + \frac{1}{2}\,ta'$$

which is a linear combination of the map embedding accuracy and estimated topographic accuracy to evaluate the quality of a SOM model. This approach has been implemented in the R-based POPSOM package [8].

2.3 R-based POPSOM Package

The R-based POPSOM package [8] has been developed and maintained by Dr. Hamel and his former students since 2013. The latest version of this package is 4.2 updated most recently on 5/31/2017. This package involves model construction, model evaluation and model visualization. Logically, these three functions should be executed sequentially.

map.build is the entrance of the POPSOM package. The input of this function is the training data (or input space) in the form of a dataframe [16]. Each row of the training data is an unlabeled training observation (or instance), and each column presents a feature of the observation. After a round of training, this function will generate an object called “map” with the following structure:

Name	Description
Data	Input data space, in form of a dataframe
labels	Label for each observation of input data.
xdim	Dimension of the map. (default = 10)
ydim	Dimension of the map. (default = 5)
alpha	Learning rate, a positive real number. (default =0.3)
Train	Number of training iteration. (default = 1000)
algorithm	Selection of training engine. (default = “vsom”)
neurons	Neuron of the map, the outcome of training.
visual	The list of best match neuron for each observation.

Table 1. The structure of the “map” object.

Both *neurons* and *visual* fields are outcomes of the model training (*map.build* function), and the rest of the arguments are input parameters of this function. All of the input parameters are free to be adjusted within a reasonable scale by the end users.

map.convergence is the function utilized for evaluating the quality of a SOM model. The outcome of this function is the linear combination of map embedding accuracy and estimated topographic accuracy, or two convergence components separately. The input of this function is the “map” object that is generated by *map.build*. There are two measurement options for evaluating the map embedding accuracy in this approach. One is *ks-test* (Kolmogorov-Smirnov convergence test) and the other is a combination of the *variance* and *mean* test.

map.starburst is used to compute and display the starburst representation of the SOM model. The heat map and the connected component lines generated by this function help the end user visualize the clusters in the map and the relationships between grids.

map.significance computes the relative significance of each feature with respect to the SOM model and graphically reports it. The purpose of developing this function is to help the end user in making the decision as to whether or not it is necessary to normalize the original input data space before training.

map.marginal plots one single dimension's marginal probability distribution of the map's neurons and the input data space. That is another aspect of the convergence of two populations.

The remaining functions within the POPSOM package are follows:

Name	**Description**
map.embed	Evaluates how well the map models the underlying training data distribution.
map.embed.ks	Reports the embedding accuracy using Kolmogorov-Smirnov convergence test.
map.embed.vm	Reports the embedding accuracy using the variance and mean tests.
map.topo	Reports the estimated topographic accuracy.
map.projection	Generates a table with the association of the labels with map co-

	ordinates.
map.neuron	Returns the contents of a neuron at (x,y) on the map as a vector.
map.normalize	Normalizes the input data space.

Table 2. Description of functions in the R-based POPSOM package.

2.4 Other Python-based SOMs packages

As of the writing this paper, there are 113 Python-based self-organizing map related repositories available on GitHub.

Rank	Repository Name	Star	Folk
1	JustGlowing/minisom	129	43
2	spiglerg/Kohonen_SOM_Tensorflow	32	11
3	mpatacchiola/pyERA	22	12
4	erogol/RSOM	21	4
5	hamilton/SelfOrganizingMaps	20	1
6	stephantul/somber	17	4
7	ramarlina/som	14	8
8	jlauron/Kohonen	13	15
9	PragmaticLab/spark-som	7	1
10	jgabriellima/self_organization_map	5	0

Table 3. Top 10 most popular (most "Star") Python-based Self-Organizing Maps repositories on GitHub (as of 01/22/2018).

The top 6 packages with the highest Star value have been selected ("Star" indicates how many people keep track of this repository and reflects the popularity of the repository) for benchmark analysis.

1) minisom:

URL: https://github.com/JustGlowing/minisom

Pros: implements both stochastic training and batch training.

Provides multiple map visualization options for the users.

2) Kohonen_SOM_Tensorflow:

   URL:

   https://github.com/spiglerg/Kohonen_SOM_Tensorflow/blob/master/som.py

   Pros: exploits TensorFlow [17] (a package released by Google) for model training.

3) pyERA

   URL: https://github.com/mpatacchiola/pyERA

   Pros: Provides an example of a SOM in which the model been applied in the real world.

4) RSOM

   URL: https://github.com/erogol/RSOM

   Pros: Implements an extension version of SOM.

5) SelfOrganizingMaps

   URL: https://github.com/hamilton/SelfOrganizingMaps

   Pros: Provides 3-D visualization of the SOM model.

6) somber

URL: https://github.com/stephantul/somber

Pros: Evaluates the quality of a SOM model using topographic accuracy.

Beyond the above listed packages, the others available are either not up to date or not popular within the user community or both. All 6 of the aforementioned most popular packages analyzed provide model construction and visualization. The “somber” package is the only package that has a function to measure the quality of the SOM model using topographic accuracy. However, the approach is very time consuming [9]. None of the packages include map embedding accuracy. It should be mentioned that all of the source codes of these packages are organized in an object-oriented programming paradigm.

CHAPTER 3

Methodology

3.1 Migration of POPSOM Package from R to Python

Migrating the source code of the POPSOM package from R to Python is the first step in implementing the SOM in Python. The goal of migration is to preserve all the functionalities during the entire process. There are three kinds of objects that need to be taken into account: 1) naming rules, 2) mathematical and statistical functions, and 3) data manipulation functions.

3.1.1 Naming Rules

In R, period separated (.) is allowed as a part of a variable's or function's name which is unique to the R language. For Python, the period separated within the names of variables or functions needs to be changed into another acceptable sign such as underscore (_).

Both the left arrow sign (←) and the equals sign (=) are acceptable assignment operators in R. The left arrow sign, which is not an acceptable assignment operator in Python, has been applied widely within the R-based POPSOM package. Thus, each of these left arrow signs be substituted by the equals sign in Python.

Finally, both R and Python are case sensitive languages. Generally, the reserved words in Python [2] are in lower case except "True", "False", and "None", which are capitalized with their first letter. All letters of these three reserved words are upper case in R [16].

R	Python
NULL	None
TRUE	True
FALSE	False

Table 4. Three reserved words in R and Python.

3.1.2 Mathematical and Statistical Functions

Beyond the basic arithmetic operations addition (+), subtraction (-), multiplication (×), and division (÷), other mathematical and statistical operators in R and Python are coded differently. In R, most of the mathematical and statistical operators use either built-in functions [18] or a combination of operators (which start and end with the percentage sign (%)). In Python, most of these functions are supported by a third-party library such as *math* library or *numpy* library or both.

Function	R	Python
Matrix Multiplication	%*%	numpy.dot(x,y)
Outer product	%o%	numpy.outer(x,y)
Modulo	%%	%
Integer Division	%/%	//
Log	log(x)	math.log(x[,base])
Sum	sum(x)	math.fsum(x)
Square Root	sqrt(x)	math.sqrt(x)

Mean	mean(x)	numpy.mean(x)
Median	median(x)	numpy.median(x)
Ceiling	ceil(x)	numpy.ceil(x)

Table 5. Examples of mathematical and statistical functions in R and Python.

3.1.3 Data Manipulation Functions

Both R and Python provide powerful data manipulation functions for data analysis and research. R uses built-in functions to manipulate data, where as Python is powered by third-party libraries (e.g. *numpy*).

Functions	R	Python
Sorting Data	sort(x)	numpy.sort(x)
Ranked Position	order(x)	numpy.argsort(x)
Means of Column	colMeans(x)	numpy.true_divide(x)
Replicate Elements	rep(x,n)	numpy.linspace(x,x,n)

Table 6. Examples of data manipulation functions in R and Python.

3.2 Rewriting Functions

Although most of the mathematical and statistical functions in the R-based POPSOM package have counterparts in Python or third-party Python libraries, there are three specific statistical functions in the R-based POPSOM package that are not available in any Python built-in or third-party libraries. Hence, these functions needed to be constructed in Python from scratch.

3.2.1 Variety of T-test

“t.test” [19], which performs a variety of t-tests, has been applied in the R-based POPSOM package to test the difference between the means of two data sets. One of the data sets comes from an input data sample, while the other one comes from the map. In Python, t-tests are implemented by utilizing the *mean* function (returns arithmetic mean along specific axis) from the *numpy* [20] library and *DescrStatsW* (returns descriptive statistics and tests with weights) and *CompareMeans* (returns a class for two sample comparison) from the *statemodels* [21] library. The source code of this function is presented in the appendix.

3.2.2 F-test

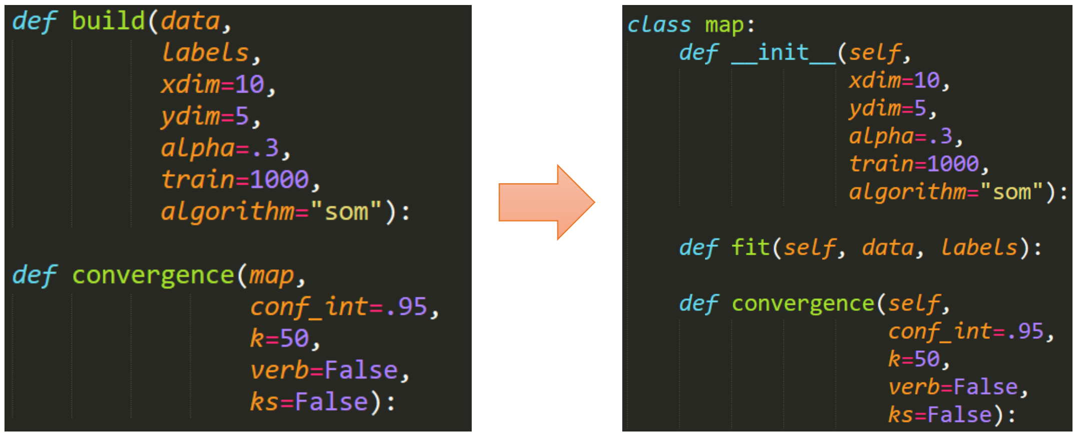
“var.test” [22] performs an F-test to test the ratio of variances of two data sets from an input data space and the map respectively in R. An F-test is implemented in Python with the help of the *variance* function (returns the sample variance of data) from the *statistics* library [23] and *ppf* (percent point function) from the *scipy* [24] library.

3.2.3 Kernel Smoother for Irregular 2-D Data

“smooth.2d” [25] is utilized to approximate the Nadaraya-Watson kernel smoother for irregular 2-D data. This function is implemented in Python by utilizing the *eculidean_distances* function from the *sklearn* [26] library and the *fft* (Discrete Fourier Transform) function from the *numpy* [20] library.

3.3 Programming Paradigm Refactoring

The formal object-oriented programming concept was introduced in the mid-1960s [27]. Many of the modern programming languages are multi-paradigm programming languages that support the object-oriented programming paradigm. Python is one of them. In the R-based POPSOM package, “map” has been defined as a class, and all the hyper-parameters, input arguments, and neurons are member variables of the class. However, all the methods are independent functions that take “map” as one of the arguments. In order to build a pure object-oriented programming package (convert all the independent methods into member methods of the class), the Python-based POPSOM package was refactored from a procedural programming paradigm to an object-oriented programming paradigm (Figure 5). After refactoring, the whole package was defined as a class. All of the input hyper-parameters, input arguments, and neurons are still member variables of the class, the same as in the R-based packages.



```
def build(data,
          labels,
          xdim=10,
          ydim=5,
          alpha=.3,
          train=1000,
          algorithm="som"):

def convergence(map,
                conf_int=.95,
                k=50,
                verb=False,
                ks=False):
```

```
class map:
    def __init__(self,
                 xdim=10,
                 ydim=5,
                 alpha=.3,
                 train=1000,
                 algorithm="som"):

    def fit(self, data, labels):

    def convergence(self,
                    conf_int=.95,
                    k=50,
                    verb=False,
                    ks=False):
```

Figure 5. Comparison of the source code appearance of a procedural programming paradigm and an object-oriented programming paradigm.

All independent methods become member methods of the “map” class. Python reserved method “__init__” was used as a constructor for the instance, and the *map.build* name was changed to *fit* as it has been used as a conventional function name for training the model (or instance) within the Python community.

3.4 Normalization

Different from PCA (Principle Component Analysis) [28], normalization is not necessary in the SOM algorithm, but it may improve numerical accuracy as proposed by [3]. A good rule of thumb, however, is for the end user to utilize the *significance* function to graphically report the significance of each feature in order to facilitate making a decision as to whether or not the original input data need to be normalized before training.

The normalization method has been developed and reserved in the R-based POP-SOM package, but works independently. In the Python-base package, not only is the normalization method implemented, but it is also applied to the model training by adding one more argument (option) in the model initialization function (“__init__”).

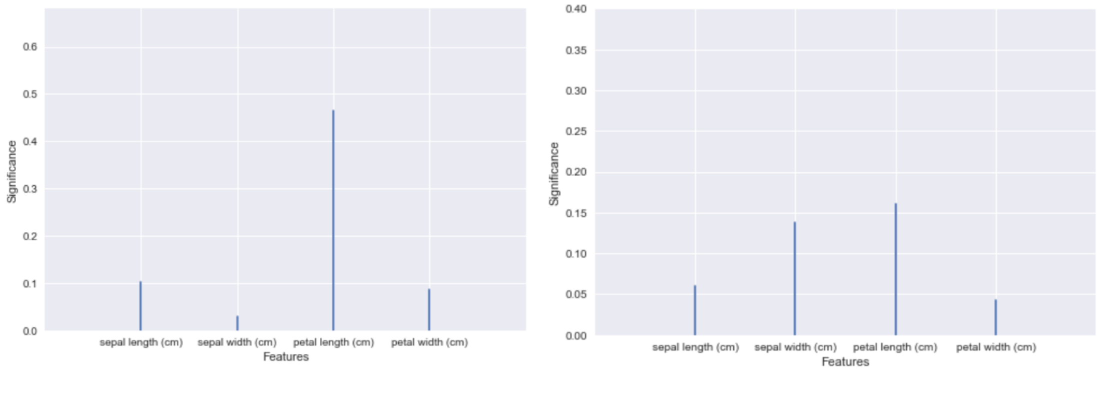


Figure 6 The significance levels of the Iris dataset features. Left figure plots the significance without data normalization. Right figure plots the significance with data normalization.

3.5 Embed Fortran for Training

There are three programming languages utilized for model training in the R-based POPSOM package: C, R and Fortran. As a kind of imperative programming language, Fortran is especially suited to numeric computation and scientific computing [28]. Hence, in addition to Python, one more training algorithm is also implemented in Fortran in this project.

Several solutions for embedding Fortran in Python have been proposed and discussed. Two of them are highly recommended. The first is to write an extension module, then to import into Python using the *import* command. The suffixes of extension modules are different in Windows (pyd) and Unix-like operation systems (so). The second is calling a shared-library subroutine directly from Python using the *ctypes* modules. It requires the code to be wrapped as a shared library. After comparison (more successful stories have been reported), the first solution is utilized in this project.

The laptop used for developing is running Windows 10 Home edition, x64-based processor, 8GB RAM. The version of Python is 3.6.0. The following discussions are all based on this development environment only.

Generally, there are five steps involved for the Fortran embedding:

1. Install MinGW-64

   MinGW (Minimalist GNU for Windows) is a software development environment for creating Microsoft Windows applications. MinGW-64 is an improved version of MinGW which supports both 32-bit and 64-bit processors. Installation is a little tricky. All of the following configurations are only applied to the specific version of Python 3.6.0.

Version	Architecture	Threads	Exception	Build Version
6.4	x86_64	posix	seh	0

Table 7. Configurations for installing MinGW-64 on Windows 10.

After successful installation, there are two more tasks:

1) Add the MinGW-64 bin path to the system path:

   c:\mingw\mingw64\bin

2) Create a configuration file (named "distutils.cfg") to connect Python with MinGW-64:

   [build]

   Compiler = mingw32

2. Install F2PY

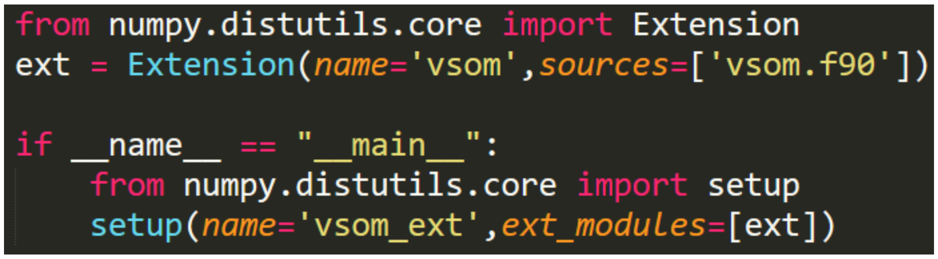
F2PY is a third-party Python library [29] which enables a Python script to call a compiled Fortran extension module. F2PY is part of the *numpy* library. Since *numpy* is already installed, there is no need to install F2PY for this project.

3. Install GFortran

GFortran (or GNU Fortran) is the abbreviation for the GNU Fortran Compiler. It is used to compile a source file (.f90) to an object file (extension module).

4. Compile the Fortran-extension module.

The standard Python build system *numpy.distutils* supports compiling Fortran-extensions (.f90 file to .pyd file). A small Python program (Figure 7) named "build.py" has been created to generate the extension module.



```
from numpy.distutils.core import Extension
ext = Extension(name='vsom',sources=['vsom.f90'])

if __name__ == "__main__":
    from numpy.distutils.core import setup
    setup(name='vsom_ext',ext_modules=[ext])
```

Figure 7. Source Code of "build.py" program

Next, execute the command: *python build.py build*. It will generate a file named "vsom.pyd" which can be loaded directly.

5. Test the extension file

Execute the following command in Python prompt:

>>>import vsom

If there is no error message prompt, then the extension module has been loaded successfully.

3.6 Speed Comparison between Python and Fortran

A comparison of the execution speeds of running the same model-training algorithm upon the same data set in Python and Fortran with the same number of iterations are of interest. The results are as follows:

Iteration	Python (in seconds)	Fortran (in seconds)
5000	2.158	0.055
10000	4.204	0.05
15000	6.632	0.06
20000	8.938	0.06
25000	11.811	0.062
30000	12.504	0.077
35000	14.787	0.076
40000	16.503	0.082
45000	17.645	0.122
50000	19.559	0.099

Table 8. Speed comparison of Python versus Fortran (as the number of iterations increases linearly).

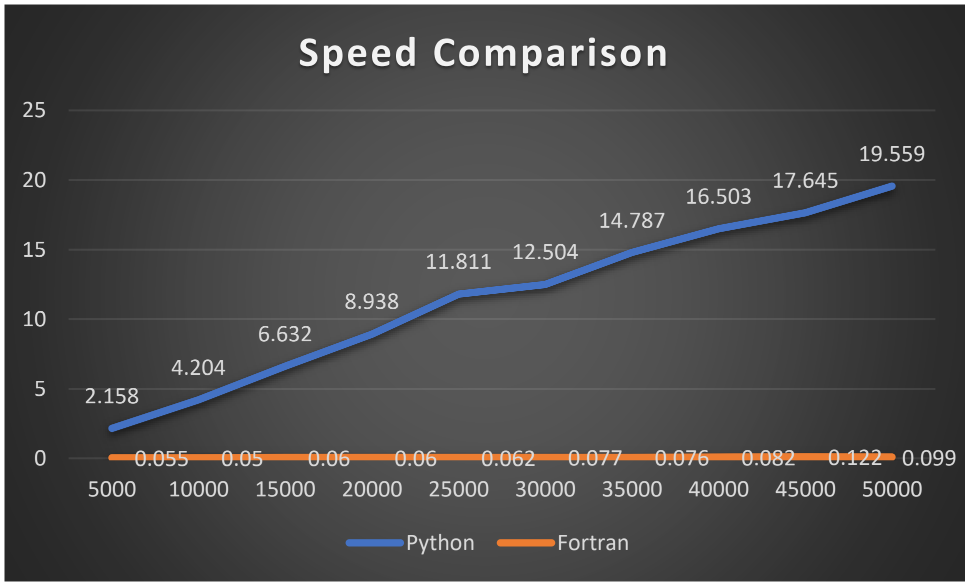


Figure 8. Speed comparison of Python versus Fortran (as the number of iterations increases linearly).

Iteration	Python (in seconds)	Fortran (in seconds)
1000	0.44	0.05
10000	3.91	0.06
100000	41.1	0.17
1000000	432.3	1.26
10000000	☹	12.4

Table 9. Speed comparison of Python versus Fortran (as the number of iterations increases exponentially).

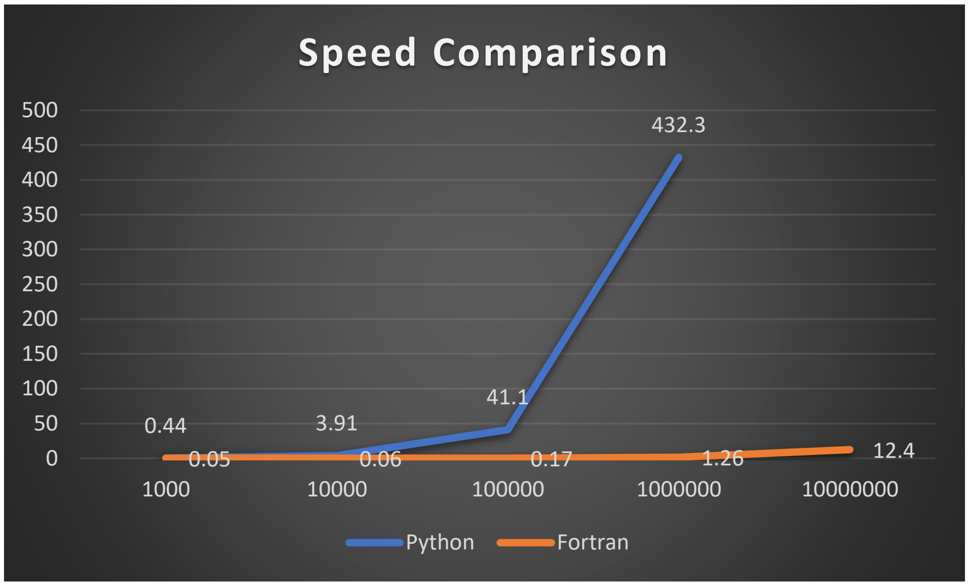


Figure 9. Speed comparison of Python versus Fortran (as the number of iterations increases exponentially).

Based on the above speed comparisons, it is obvious that Fortran is much more efficient than Python in numerical computations. Table 10 reports the processing time (in seconds) for training the iris flower data set [10] and wheat seed data set [11] until the maps fully converged using Python and Fortran respectively.

Data Set	Iris Flower		Wheat Seed	
Observation	150		210	
Feature	4		7	
Iteration	1000		2000	
Map Dimension	10×5		15×10	
	Python	**Fortran**	**Python**	**Fortran**
Convergence Index	0.939	0.959	0.97	0.89
Time (Seconds)	0.459	0.041	1.05	0.08

Table 10. The processing time (in seconds) for training Iris Flower data and Wheat Seed data using Python and Fortran.

CHAPTER 4

Results

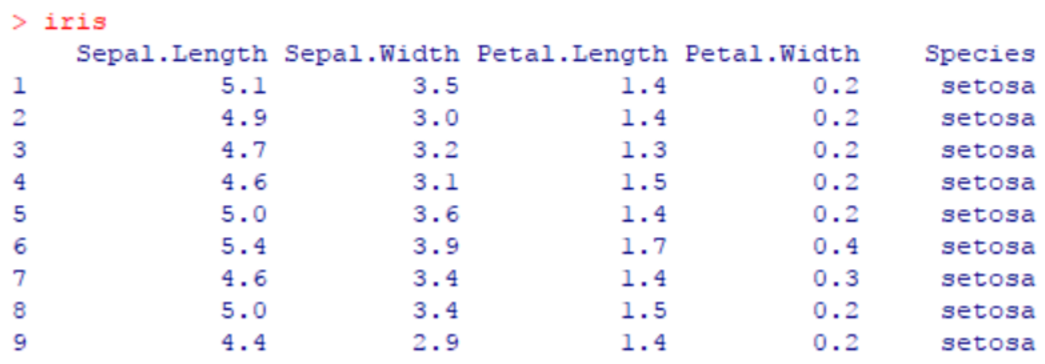
4.1 Experiment Design

Since this project is inspired and based on the R-based POPSOM package, the entire implementation of the Python-based package can be divided into migrating the R-based package to Python-based and refactoring the source code from a procedural programming paradigm to an object-oriented programming paradigm. After the Python-based package was complete, it was determined that the best way to evaluate the correctness and quality of the Python-based package was to compare the outcome of each function with the outcome from the R-base package. Most functions in the package run with a non-random algorithm. Hence, it is expected that the same input would generate the same outcome, such as reporting the significance of each feature and plotting the marginal probability distribution of neurons and input data. On the other hand, some algorithms run with random factors, in particular the model-training algorithm.

4.1.1 Data Set Selection

As a kind of unsupervised learning algorithm, the major task of Self-Organizing Maps is clustering the input data. Thus, the label of the observation is not necessary in the algorithm, but it will help the end user to interpret the map. The ideal data for this project is intuitive, easily interpreted and clustered (or categorized) by human beings (although professional knowledge may be required in some cases). The data should have at least three-dimensional measurements (two-dimensional data can be presented by 2-D map without any learning). To evaluate the quality and capability of the Python-based package, two different data sets with different magnitudes of measurements and observations for the experiment were selected.

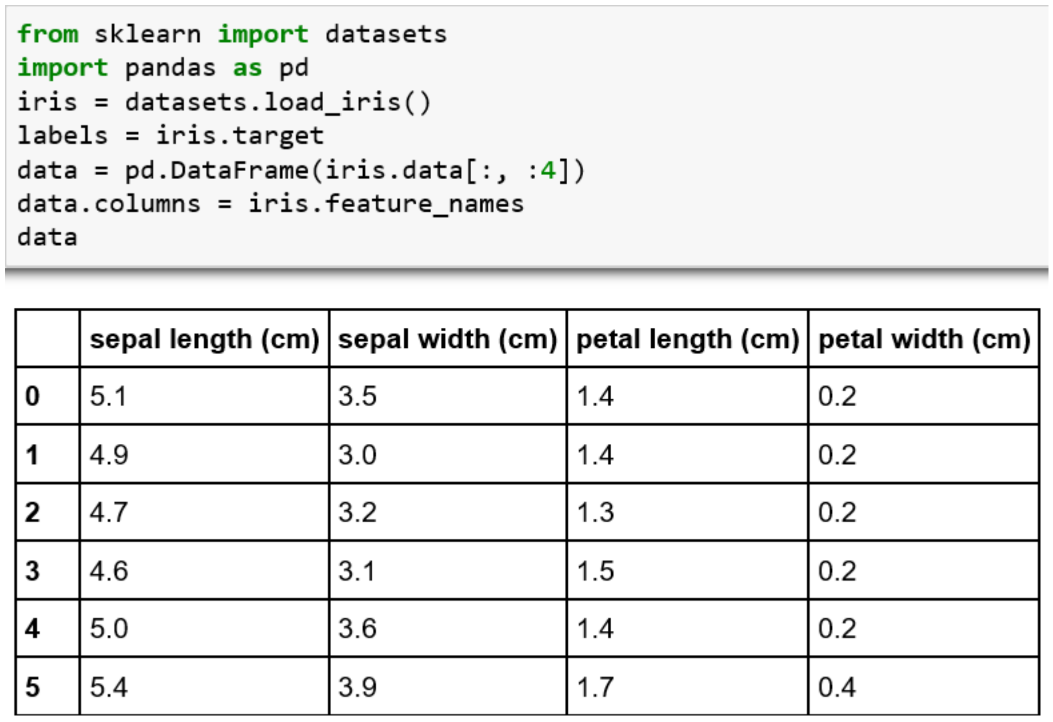
The Iris flower data set [10] (sometimes called Fisher's or Anderson's data set) introduced by Ronald Fisher in 1930s has been widely used as a "toy"/test data set within the machine learning and statistics communities. There are 150 observations (or instances) that are categorized into three species distributed evenly within the data set. This data set has four measurements (or attributes): the sepal length, the sepal width, the petal length and the petal width, all of which are measured on the same scale (in centimeters). The Iris data set has been embedded in R (Figure 10) and can be accessed directly.



```
> iris
  Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1          5.1         3.5          1.4         0.2     setosa
2          4.9         3.0          1.4         0.2     setosa
3          4.7         3.2          1.3         0.2     setosa
4          4.6         3.1          1.5         0.2     setosa
5          5.0         3.6          1.4         0.2     setosa
6          5.4         3.9          1.7         0.4     setosa
7          4.6         3.4          1.4         0.3     setosa
8          5.0         3.4          1.5         0.2     setosa
9          4.4         2.9          1.4         0.2     setosa
```

Figure 10. Accessing the Iris data set in R

The Iris data set has been embedded in the *scikit-learn* Python library. Before accessing this data set in Python, the *sklearn* library needs to be imported at the very beginning of the source code. In order to represent it as a data frame (Figure 11), which is friendlier to the end user, the *pandas* library should also be imported.



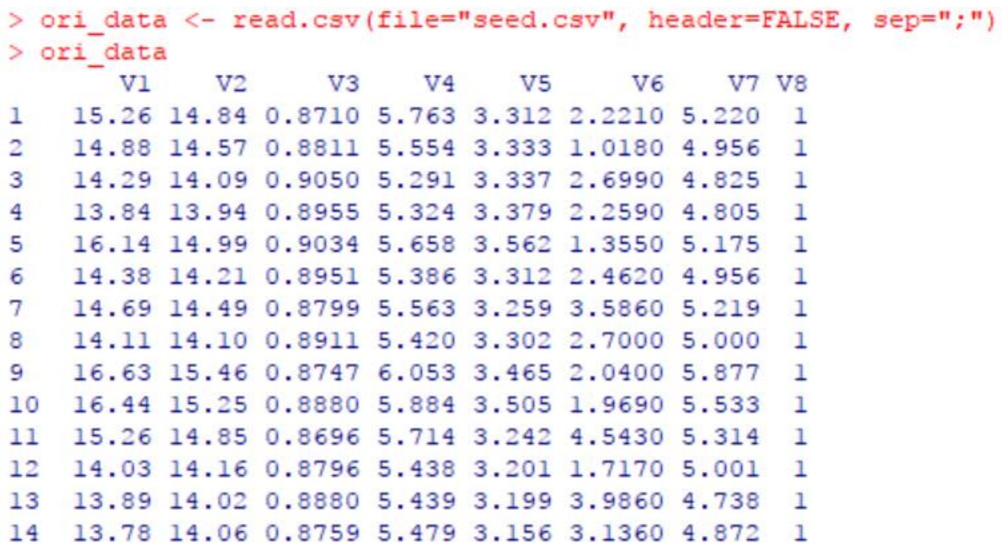
```
from sklearn import datasets
import pandas as pd
iris = datasets.load_iris()
labels = iris.target
data = pd.DataFrame(iris.data[:, :4])
data.columns = iris.feature_names
data
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2
5	5.4	3.9	1.7	0.4

Figure 11. Accessing and representing the Iris data set as a data frame in Python.

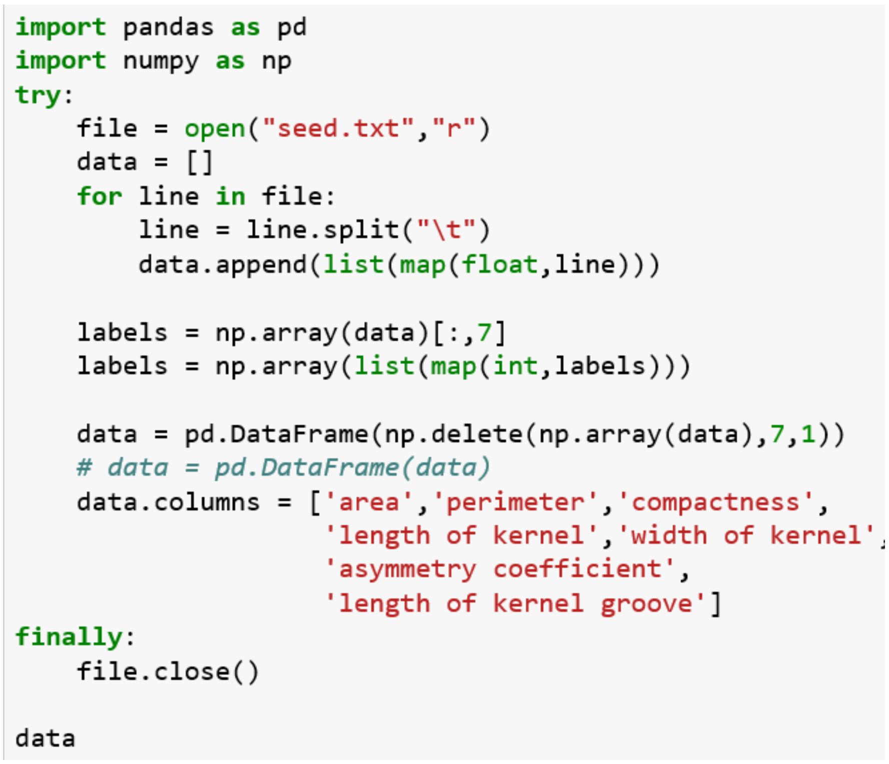
In addition to the Iris data set, the Wheat Seed data set [11] was also selected from UCI machine learning repository to evaluate the Python-based package. This data set of grain measurements, which was obtained from the real word, contains 210 observations clustered as 3 species (Kama, Rosa and Canadian), 70 elements for each. There are 7 measurements of main geometric features obtained by X-ray technique: area, perimeter, compactness, length of kernel, width of kernel, asymmetry coefficient, and length of kernel groove. All of them are scaled by either millimeters or square millimeters.

The original Wheat Seed data are stored in either a csv file or plain text file. The *read.csv* (Figure 12) command in R and the *open* (Figure 13) command in Python were utilized to access the Wheat Seed data set.



```
> ori_data <- read.csv(file="seed.csv", header=FALSE, sep=";")
> ori_data
      V1    V2     V3    V4    V5     V6    V7 V8
1  15.26 14.84 0.8710 5.763 3.312 2.2210 5.220  1
2  14.88 14.57 0.8811 5.554 3.333 1.0180 4.956  1
3  14.29 14.09 0.9050 5.291 3.337 2.6990 4.825  1
4  13.84 13.94 0.8955 5.324 3.379 2.2590 4.805  1
5  16.14 14.99 0.9034 5.658 3.562 1.3550 5.175  1
6  14.38 14.21 0.8951 5.386 3.312 2.4620 4.956  1
7  14.69 14.49 0.8799 5.563 3.259 3.5860 5.219  1
8  14.11 14.10 0.8911 5.420 3.302 2.7000 5.000  1
9  16.63 15.46 0.8747 6.053 3.465 2.0400 5.877  1
10 16.44 15.25 0.8880 5.884 3.505 1.9690 5.533  1
11 15.26 14.85 0.8696 5.714 3.242 4.5430 5.314  1
12 14.03 14.16 0.8796 5.438 3.201 1.7170 5.001  1
13 13.89 14.02 0.8880 5.439 3.199 3.9860 4.738  1
14 13.78 14.06 0.8759 5.479 3.156 3.1360 4.872  1
```

Figure 12. Accessing the Wheat Seed data set in R

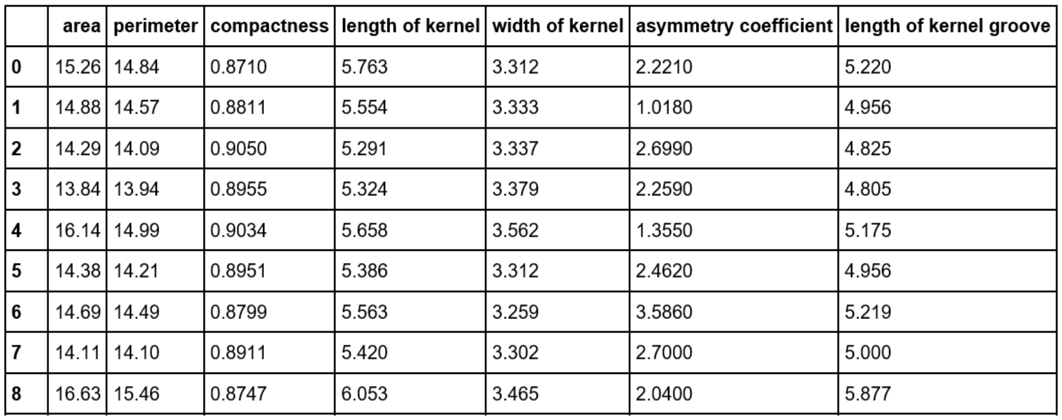


```python
import pandas as pd
import numpy as np
try:
    file = open("seed.txt","r")
    data = []
    for line in file:
        line = line.split("\t")
        data.append(list(map(float,line)))

    labels = np.array(data)[:,7]
    labels = np.array(list(map(int,labels)))

    data = pd.DataFrame(np.delete(np.array(data),7,1))
    # data = pd.DataFrame(data)
    data.columns = ['area','perimeter','compactness',
                    'length of kernel','width of kernel',
                    'asymmetry coefficient',
                    'length of kernel groove']
finally:
    file.close()

data
```



	area	perimeter	compactness	length of kernel	width of kernel	asymmetry coefficient	length of kernel groove
0	15.26	14.84	0.8710	5.763	3.312	2.2210	5.220
1	14.88	14.57	0.8811	5.554	3.333	1.0180	4.956
2	14.29	14.09	0.9050	5.291	3.337	2.6990	4.825
3	13.84	13.94	0.8955	5.324	3.379	2.2590	4.805
4	16.14	14.99	0.9034	5.658	3.562	1.3550	5.175
5	14.38	14.21	0.8951	5.386	3.312	2.4620	4.956
6	14.69	14.49	0.8799	5.563	3.259	3.5860	5.219
7	14.11	14.10	0.8911	5.420	3.302	2.7000	5.000
8	16.63	15.46	0.8747	6.053	3.465	2.0400	5.877

Figure 13. Accessing and representing the Wheat Seed data set as a data frame in Python

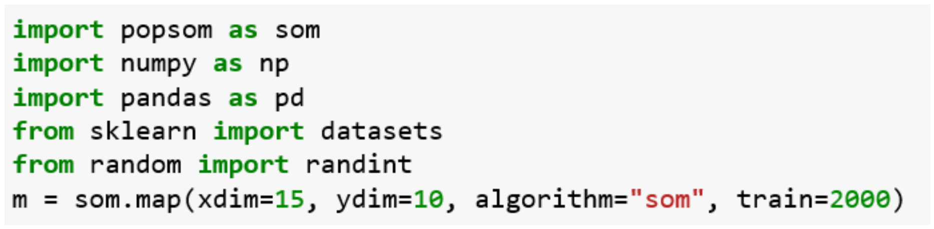
4.2 Iris Experiment Results

4.2.1 Initialize the Model (instantiate the Model)

In the R-based POPSOM package, there is no independent function for initializing the model. However, after refactoring the Python-based package, it allows the user to use the reserved function "__init__" to initialize the model (setup the hyper-parameters for model training):

#	Argument	Description	Default
1	Xdim	X-dimension of the map	10
2	Ydim	Y-dimension of the map	5
3	Alpha	Learning rate, should be a positive real number	0.3
4	Train	Number of training iterations	1000
5	Algorithm	Selection switch (Python or Fortran)	som
6	Norm	Switch, apply normalization to input data space	False

Table 11. Description of "__init__" function's arguments.



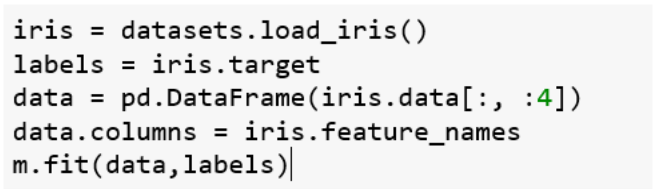
```python
import popsom as som
import numpy as np
import pandas as pd
from sklearn import datasets
from random import randint
m = som.map(xdim=15, ydim=10, algorithm="som", train=2000)
```

Figure 14. Example of initializing the model in Python.

Most of the arguments are easy to understand and setup. For the argument of "*train*", which indicates the number of iterations, there is no good rule of thumb recommended. Less iterations will result in insufficient converge, while more iterations will increase unnecessary computational expense. Examining the quality of the map after each training is the best way to determine the optimal number of iterations. For the Iris data set, 1000 iterations will return better than a 0.9 convergence index in most instances, which is acceptable for this project.

4.2.2 Fit the data

The R-based package merges the initialization and fitting the data into one function, called *map.build*, while the Python-based package has an independent fitting data function: *fit*. The *fit* function only has two arguments, data and labels. This label is different with the one in other supervised algorithms. In the SOM algorithm, labels are not involved in the training process. They are only used for labeling the grid of the map after training.

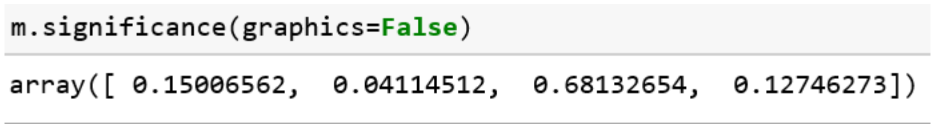


```
iris = datasets.load_iris()
labels = iris.target
data = pd.DataFrame(iris.data[:, :4])
data.columns = iris.feature_names
m.fit(data,labels)
```

Figure 15. Example of fitting the Iris data and labels to the SOM model.

4.2.3 Report the Significance of Each Feature

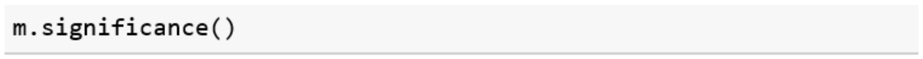
The significance of each feature can be reported in the form of either a vector (Figure 16) or a graph (Figure 17) by switching *graphics* to True (which is default) or False respectively.



```
m.significance(graphics=False)
```

```
array([ 0.15006562,  0.04114512,  0.68132654,  0.12746273])
```

Figure 16. Reporting the significance of each feature by vector for the Iris data.



```
m.significance()
```

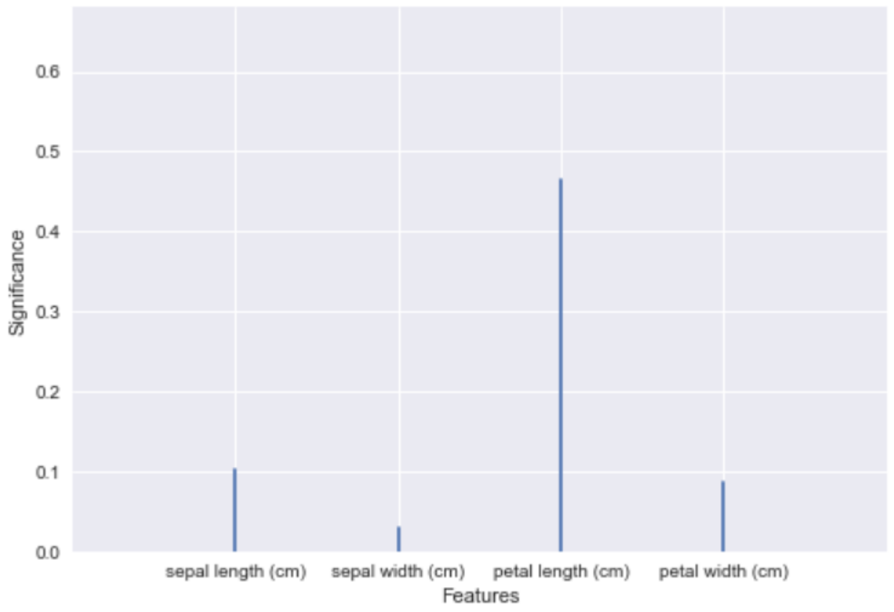


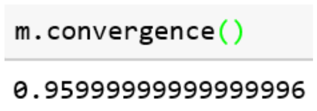
Figure 17. Graphically reporting the significance of each feature for the Iris data.

4.2.4 Report the map convergence index

The convergence index is a linear combination of the map embedding accuracy and the estimated topographic accuracy (Figure 18, 19, 20, 21). It is a criteria for evaluating the quality of the map [4] (or model). There are four arguments in this function:

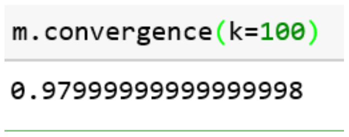
#	Argument	Description	Default
1	conf_int	Confidence interval of the quality assessment	0.95
2	K	Sample size used for the estimated topographic accuracy computation	50
3	verb	**True:** report the map embedding accuracy and estimated topographic accuracy separately; **False:** report the linear combination of map embedding accuracy and estimated topographic accuracy.	False
4	ks	**True:** use the ks-test to report the map embedding accuracy. **False:** use the variance and mean tests to report the map embedding accuracy.	False

Table 12. Description of convergence function's arguments.



```
m.convergence()
0.9599999999999996
```

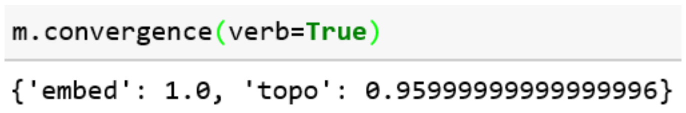
Figure 18. Reporting the convergence index of the map with default arguments.



```
m.convergence(k=100)

0.9799999999999998
```

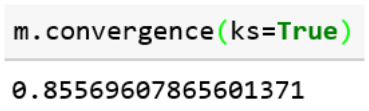
Figure 19. Reporting the convergence index of the map by selecting 100 samples for the estimated topographic accuracy.



```
m.convergence(verb=True)

{'embed': 1.0, 'topo': 0.9599999999999996}
```

Figure 20. Reporting the map embedding accuracy and the estimated topographic accuracy separately.



```
m.convergence(ks=True)

0.85569607865601371
```

Figure 21. Reporting the convergence index of the map with the *ks-test* approach for the map embedding accuracy.

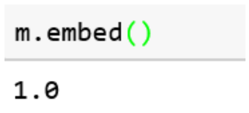
4.2.5 Report the Map Embedding Accuracy

Report the map embedding accuracy using either the *ks-test* or the *variance* and *mean* tests (Figure 22, 23).

#	Argument	Description	Default
1	conf_int	Confidence interval of the quality assessment	0.95
2	verb	**True:** report the map embedding accuracy and estimated topographic accuracy separately; **False:** report the linear combination of map embedding accuracy and estimated topographic accuracy.	False

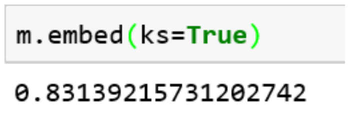
3	ks	**True:** use the ks-test to report the map embedding accuracy. **False:** use the variance and mean tests to report the map embedding accuracy.	False

Table 13. Description of *embed* function's arguments.



```
m.embed()
1.0
```

Figure 22. Reporting the map embedding accuracy using the *variance* and *mean* tests.



```
m.embed(ks=True)
0.83139215731202742
```

Figure 23. Reporting the map embedding accuracy using *ks-test*.

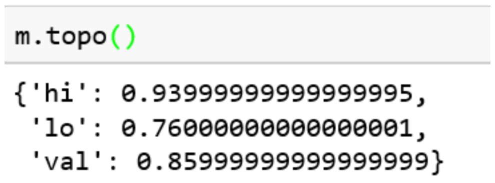
4.2.6 Report the Estimated Topographic Accuracy

Estimated topographic accuracy is a part of the convergence index. It also can be reported independently as well for this project (Figure 24, 25, 26, 27). As discussed in [9], evaluating the SOMs topographic accuracy by using random samples instead of all available input data is a reliably computational and efficient statistical approach.

#	Argument	Description	Default
1	conf_int	Confidence interval of the quality assessment	0.95
2	k	Sample size used for the estimated topographic accu-	50

		racy computation	
3	verb	**True:** report the map embedding accuracy and estimated topographic accuracy separately; **False:** report the linear combination of map embedding accuracy and estimated topographic accuracy.	False
4	interval	**True:** confidence interval is computed **False:** confidence interval is not computed	True

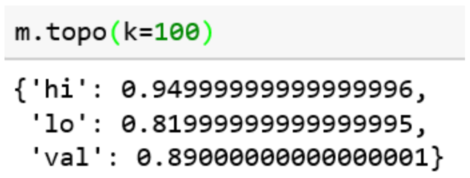
Table 14. Description of *topo* function's arguments.



```
m.topo()
```

```
{'hi': 0.93999999999999995,
 'lo': 0.76000000000000001,
 'val': 0.85999999999999999}
```

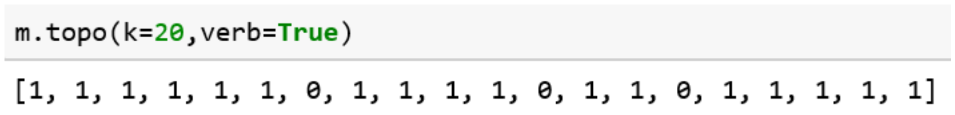
Figure 24. Reporting the estimated topographic accuracy with the default argument's value.



```
m.topo(k=100)
```

```
{'hi': 0.94999999999999996,
 'lo': 0.81999999999999995,
 'val': 0.89000000000000001}
```

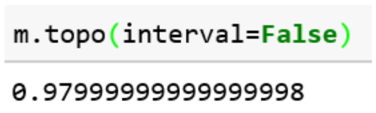
Figure 25. Reporting the estimated topographic accuracy with *k=100*.



```
m.topo(k=20,verb=True)
```

```
[1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1]
```

Figure 26. Reporting a vector of individual feature accuracies.



```
m.topo(interval=False)
```

```
0.97999999999999998
```

Figure 27. Reporting the estimated topographic accuracy without computing the confidence interval.

4.2.7 Starburst Visualization of the Model

Plotting the starburst representation of the SOM model is one of the most important functions in the POPSOM package. This function plots a 2-D heat map representation (Figure 28, 29) based on the model that satisfies the user. In addition, this function also plots the connected component lines over the heat map, which makes it easy for users to identify the center of clusters and the associated boundaries.

#	Argument	Description	Default
1	explicit	Control the shape of connected components True: show exact connected components. False: all nodes are connected to their centroid node.	False
2	smoothing	Control the smoothing lever of the U-Matrix	2
3	merge_clusters	Starburst clusters are merged together.	True
4	merge_range	The percentage of a certain distance in the code to determine whether components are closer to their centroids or instead centroids are closer to each other.	0.25

Table 15. Description of starburst function's arguments.

```
m.starburst()
```

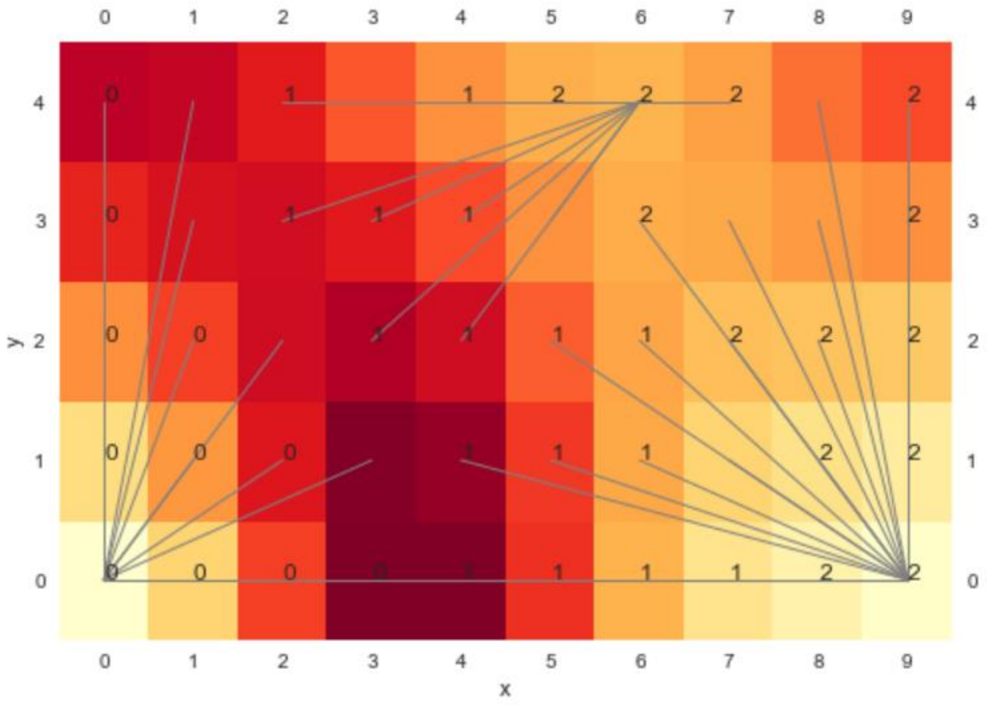


Figure 28. Starburst representation of the SOM model with default argument values. Connected component lines represent that all nodes are connected to the center node.

```
m.starburst(explicit=True)
```

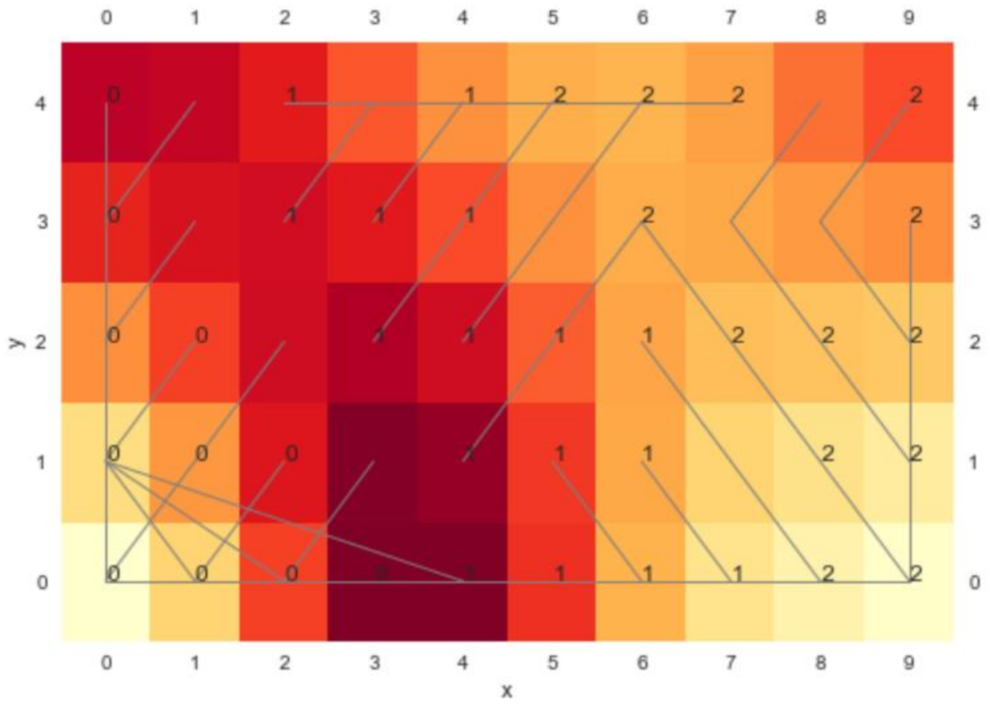


Figure 29. Starburst representation of the SOM model. Connected component lines represent the exact connected components.

4.2.8 Visualization of the Marginal Probability Distribution of the Feature

This function shows the marginal probability distribution of the neurons and the input data. The density of the training data frame and the neuron density of the same dimension (or index) are to be overlaid on the plot (Figure 31). The more overlaid these are indicates the higher quality of the model. The only argument of this function is either the index of measurement (such as 0) or the name of the measurement (such as "sepal length (cm)") as follows:

```
m.marginal(0)
m.marginal("sepal length (cm)")
```

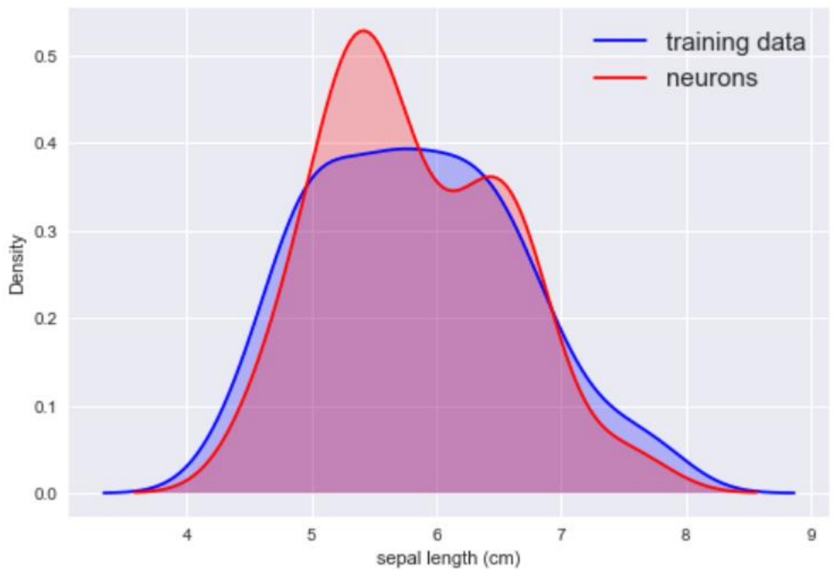


Figure 30. Reporting the marginal probability distribution of the first measurement by index or attribute name.

```
for i in range(4):
    m.marginal(i)
```

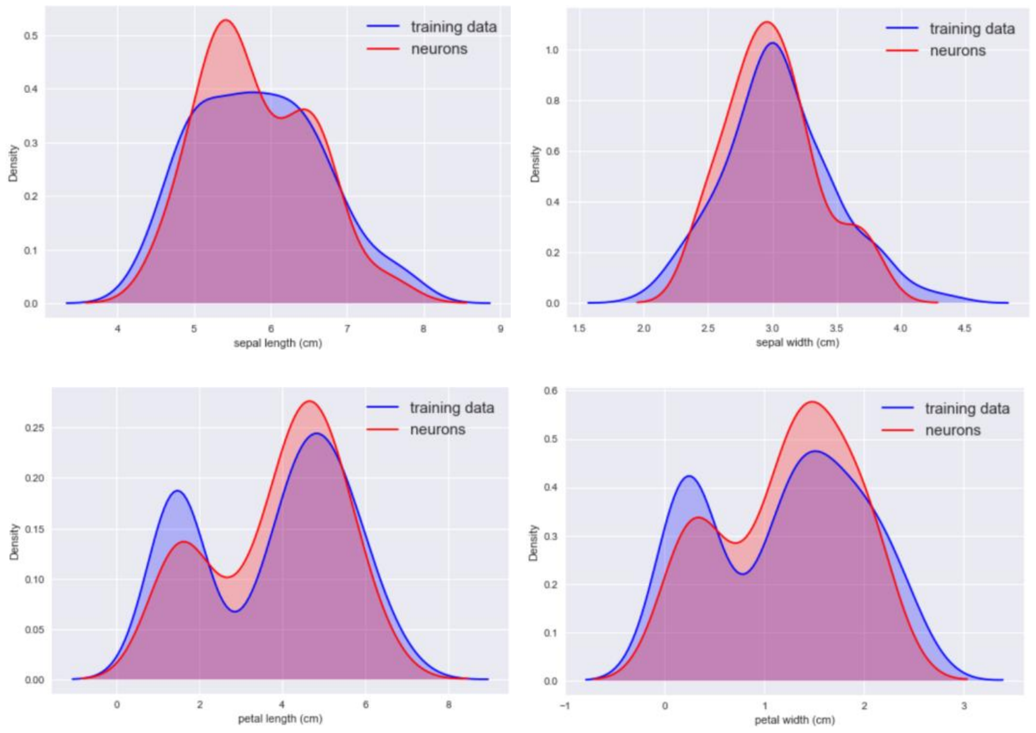
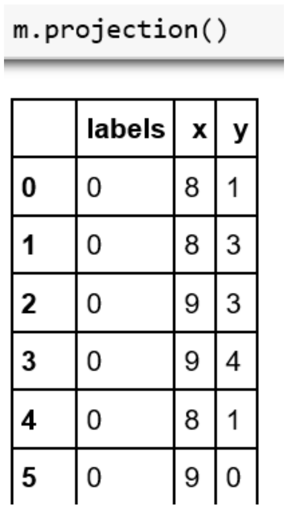


Figure 31. Marginal probability distributions of each attribute of the Iris data.

4.2.9 Projection

This function returns a table which reports the coordinate (location) of each observation on the map (Figure 32).



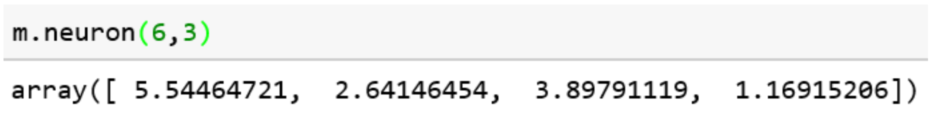
```
m.projection()
```

	labels	x	y
0	0	8	1
1	0	8	3
2	0	9	3
3	0	9	4
4	0	8	1
5	0	9	0

Figure 32. Reporting the coordinate of each observation on the map.

4.2.10 Neuron

This function returns the content of the observation by given coordinates (Figure 33).



```
m.neuron(6,3)
```

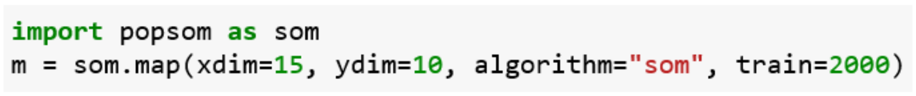
```
array([ 5.54464721,  2.64146454,  3.89791119,  1.16915206])
```

Figure 33. Reporting the content of the observation by given coordinates.

4.3 Wheat Seed Experiment Results

4.3.1 Initialize the Model (instantiate the Model)

Since there are 210 observations in the Wheat Seed data set, a larger (15 × 10) map is used to represent the model (Figure 34).

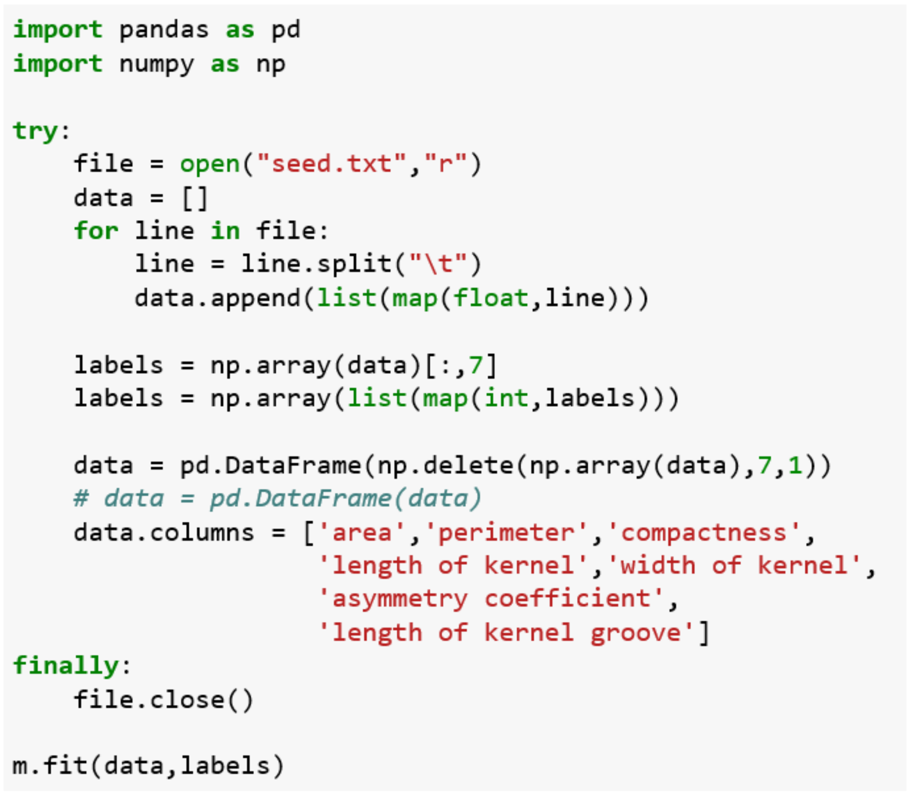


```
import popsom as som
m = som.map(xdim=15, ydim=10, algorithm="som", train=2000)
```

Figure 34. Example of initializing the model in Python.

4.3.2 Fit the data

The raw data of the Wheat Seed data set are stored in the text file without the header. The data was loaded from the text file and the attribute name was inserted manually (Figure 35).



```
import pandas as pd
import numpy as np

try:
    file = open("seed.txt","r")
    data = []
    for line in file:
        line = line.split("\t")
        data.append(list(map(float,line)))

    labels = np.array(data)[:,7]
    labels = np.array(list(map(int,labels)))

    data = pd.DataFrame(np.delete(np.array(data),7,1))
    # data = pd.DataFrame(data)
    data.columns = ['area','perimeter','compactness',
                    'length of kernel','width of kernel',
                    'asymmetry coefficient',
                    'length of kernel groove']
finally:
    file.close()

m.fit(data,labels)
```

Figure 35. Fitting the Wheat Seed data and labels to the model.

4.3.3 Report the Significance of Each Feature

```
m.significance(graphics=False)
```

```
array([  6.50574754e-01,   1.31056888e-01,   4.29049051e-05,
         1.50845671e-02,   1.09629677e-02,   1.73716399e-01,
         1.85615196e-02])
```

Figure 36. Reporting the significance of each feature by vector for the Wheat Seed data.

```
m.significance()
```

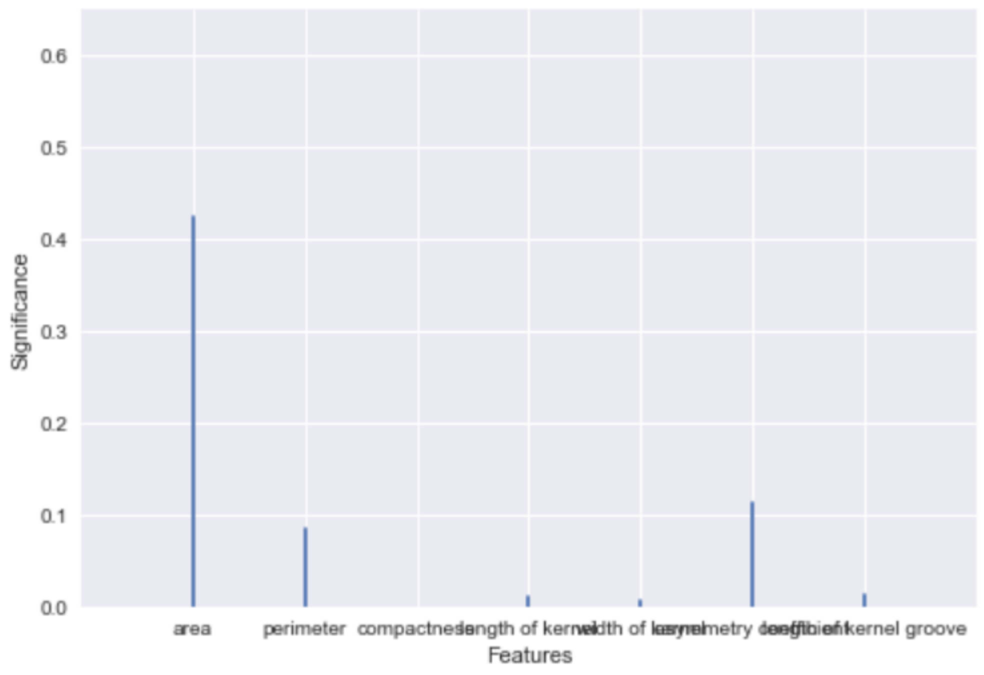


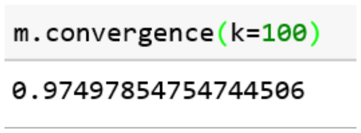
Figure 37. Graphically reporting the significance of each feature for the Wheat Seed data.

4.3.4 Report the map convergence index

```
m.convergence()
```

```
0.98997854754744508
```

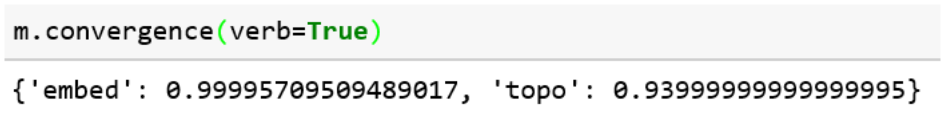
Figure 38. Reporting the convergence index of the map with arguments equal to default values.



```
m.convergence(k=100)
```

```
0.9749785475474450 6
```

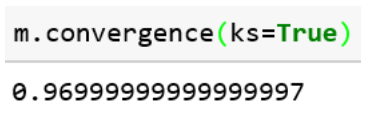
Figure 39. Reporting the convergence index of the map by selecting 100 samples for estimated topographic accuracy.



```
m.convergence(verb=True)
```

```
{'embed': 0.99995709509489017, 'topo': 0.939999999999999995}
```

Figure 40. Reporting the map embedding accuracy and estimated topographic accuracy separately.

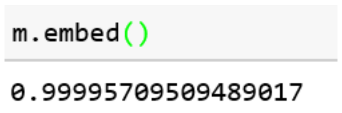


```
m.convergence(ks=True)
```

```
0.96999999999999997
```

Figure 41. Reporting the convergence index of the map with the *ks-test* approach for embedding accuracy.

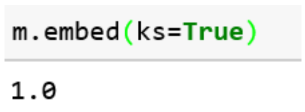
4.3.5 Report the Map Embedding Accuracy



```
m.embed()
```

```
0.99995709509489017
```

Figure 42. Reporting the map embedding accuracy using the *variance* and *mean tests*.

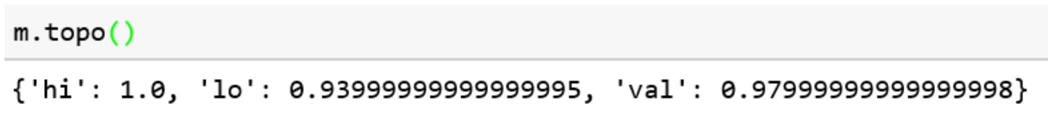


```
m.embed(ks=True)
```

```
1.0
```

Figure 43. Reporting the map embedding accuracy using the *ks-test*.

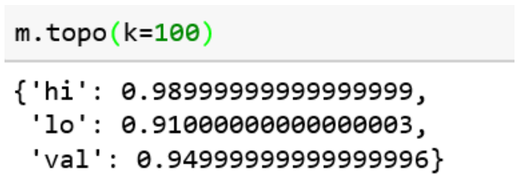
4.3.6 Report the Estimated Topographic Accuracy



```
m.topo()
```

```
{'hi': 1.0, 'lo': 0.93999999999999995, 'val': 0.97999999999999998}
```

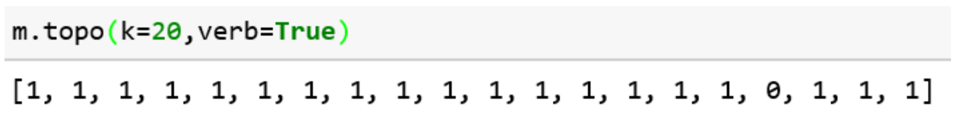
Figure 44. Reporting the estimated topographic accuracy with arguments equal to the default value.



```
m.topo(k=100)
```

```
{'hi': 0.98999999999999999,
 'lo': 0.91000000000000003,
 'val': 0.94999999999999996}
```

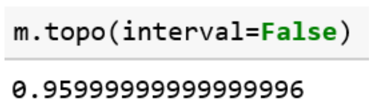
Figure 45. Reporting the estimated topographic accuracy with *k=100*.



```
m.topo(k=20,verb=True)
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1]
```

Figure 46. Reporting a vector of individual feature accuracies.



```
m.topo(interval=False)
```

```
0.95999999999999996
```

Figure 47. Reporting the estimated topographic accuracy without computing the confidence interval.

4.3.7 Starburst visualization of the model

```
m.starburst()
```

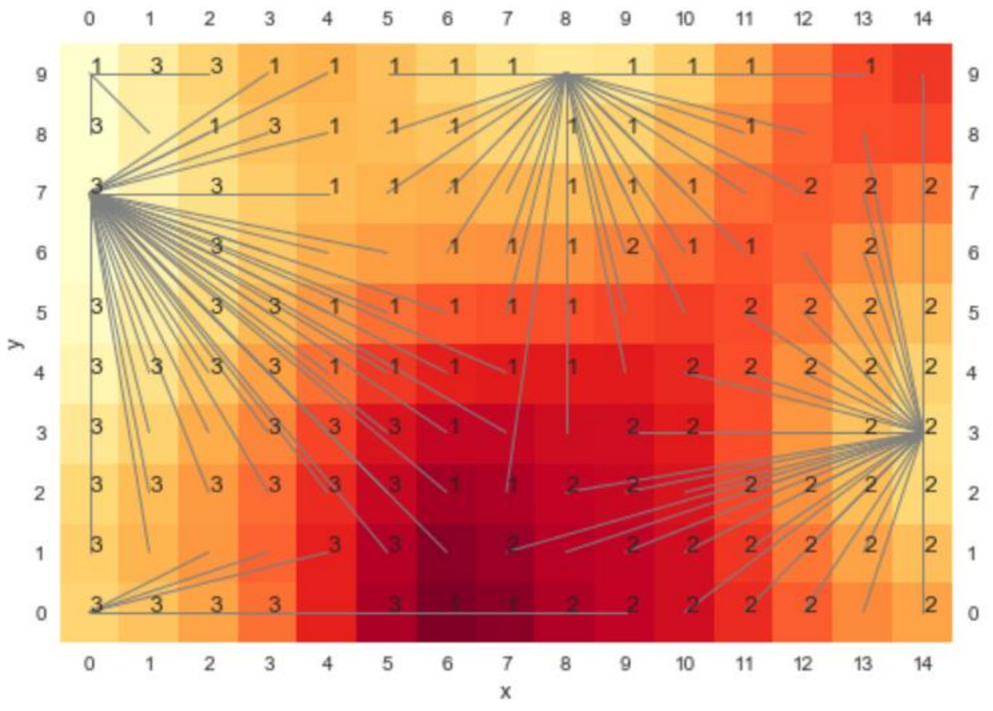


Figure 48. Starburst representation of the SOM model with arguments equal to default values. Connected component lines represent that all nodes are connected to the center node.

```
m.starburst(explicit=True)
```

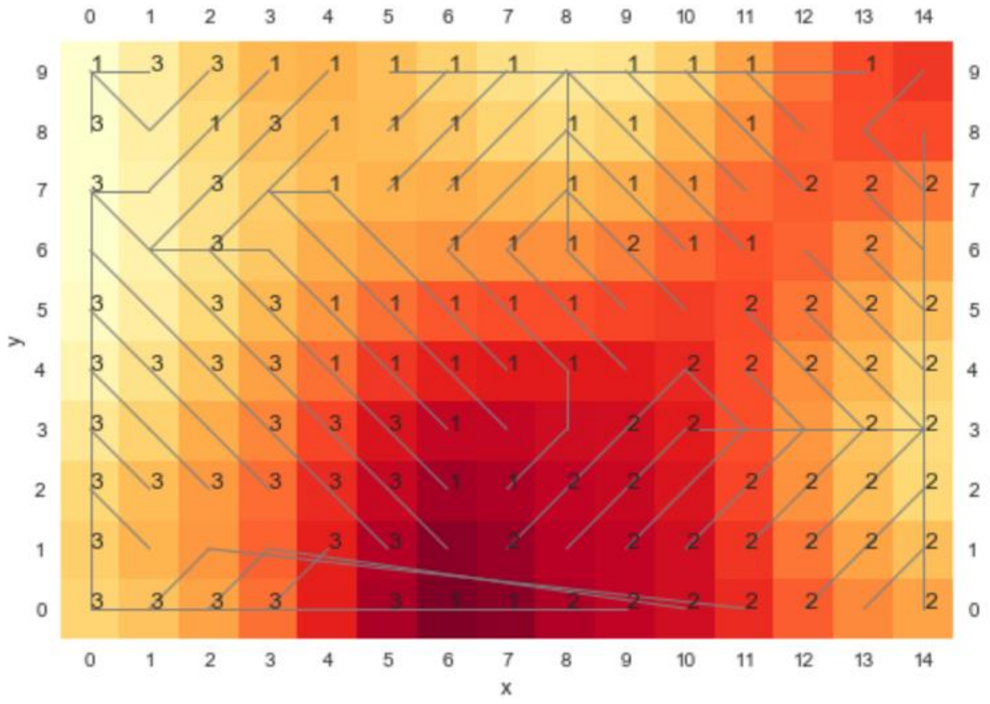


Figure 49. Starburst representation of the SOM model. Connected component lines represent exact connected components.

4.3.8 Visualization of the Marginal Probability Distribution of the Feature

```
m.marginal(1)
m.marginal("perimeter")
```

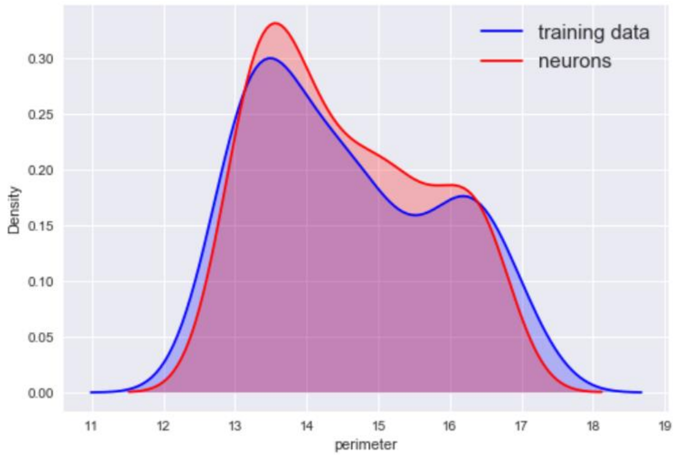


Figure 50. Reporting the marginal probability distribution of the second measurement by the index or attribute name.

```python
for i in range(7):
    m.marginal(i)
```

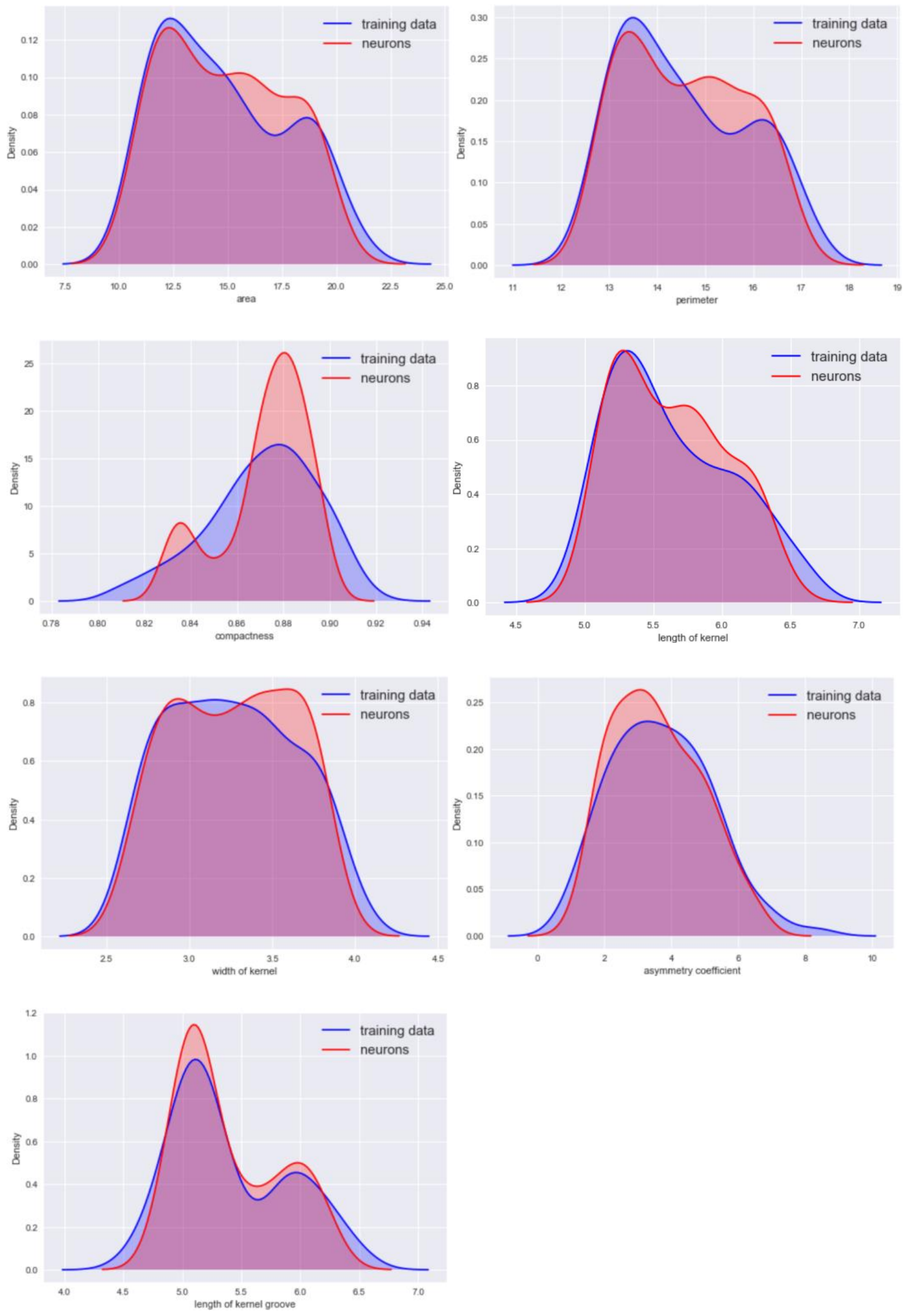
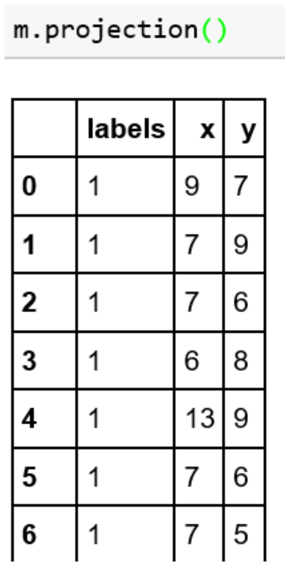


Figure 51. Marginal probability distribution of each attribute of the Wheat Seed data.

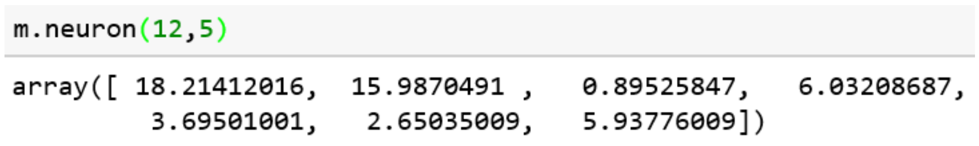
4.3.9 Projection


```
m.projection()
```

	labels	x	y
0	1	9	7
1	1	7	9
2	1	7	6
3	1	6	8
4	1	13	9
5	1	7	6
6	1	7	5

Figure 52. Reporting the location of each observation on the map.

4.3.10 Neuron


```
m.neuron(12,5)
```

```
array([ 18.21412016,  15.9870491 ,   0.89525847,   6.03208687,
         3.69501001,   2.65035009,   5.93776009])
```

Figure 53. Reporting the content of the observation by given coordinates.

4.4 Evaluating the Correctness of Python-based Package.

The R-based POPSOM package has been developed and verified, and this Python-based package was derived from it. Thus, the best way to evaluate the correctness of the Python-based package is to measure the distance between the R-based package and the Python-based package results. Three sophisticated functions from the package were utilized to demonstrate this comparison: model training, starburst representation, and visualization of marginal probability distribution.

4.4.1 Evaluating the Model Training Function

The R-based package used *map.build* to train the model, while the Python-based package used *__init__* and *fit* functions to complete this same task. There are two random factors within the algorithm: 1) randomly initialized the neurons at the beginning and 2) random selection of a vector from the input data space for training. Due to these two random factors, it is not feasible to expect that the same training data will generate exactly the same neurons from both the R and Python functions. In order to evaluate the correctness of the Python program based on the two neurons generated, three statistical approaches have been proposed and applied in this project.

1) Measuring the average difference of vectors between the two neurons with following formula,

$$ave = \frac{\sum |W_R - W_P|}{D_N * F_N}$$

W_R: the single weight vector from R function.

W_P: the single weight vector from Python function.

D_N: the dimension of the neurons.

F_N: the number of features.

The average difference should be approximately equal to 0 if the two neurons are drawn from the same input data space. The following chart represents the average differences of vectors between two neurons during the training.

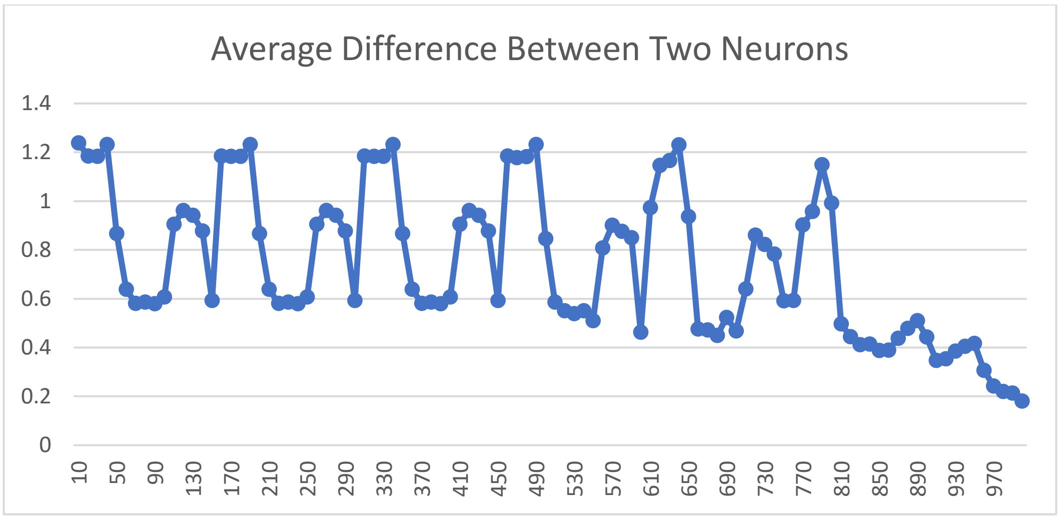


Figure 54. The average difference of vectors between two neurons.

As can be clearly seen, the average difference descends to 0.2 at the end of the training. This result fulfills the expectation.

2) Measuring the ratio of the variances from the two neurons. [9], [30]

$$\frac{S_1^2}{S_2^2} \cdot \frac{1}{f_{\frac{\alpha}{2},n_1-1,n_2-1}} < \frac{\sigma_1^2}{\sigma_2^2} < \frac{S_1^2}{S_2^2} \cdot f_{\frac{\alpha}{2},n_1-1,n_2-1}$$

S_1^2, S_2^2: The values of the variance from each neuron that have been generated by R and Python function respectively.

$f_{\frac{\alpha}{2},n_1-1,n_2-1}$: The F distribution with n_1 - 1 and n_2 – 1 degrees of freedom.

n_1, n_2: the number of neurons generated by R and Python functions.

The ratio should be approximately equal to 1 if the two neurons are drawn from the same input space. The following ratio of each features' variances and corresponding 95% confidence intervals were obtained from the Iris data experiments:

Lower bound	0.579149	0.582138	0.573727	0.582895
Ratio	1.020570	1.025837	1.011014	1.027170
Upper bound	1.798437	1.807719	1.781599	1.810068

Table 16. The ratio of the variance between two neurons

This result reveals that the two neurons have very similar distributions since all 4 ratios reported are not significantly different from 1.

3) Measuring the difference of the means of two neurons.

$$\boldsymbol{\mu_1 - \mu_2 > (\overline{x_1} - \overline{x_2}) - z_{\frac{\alpha}{2}} \cdot \sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}$$

$$\boldsymbol{\mu_1 - \mu_2 < (\overline{x_1} - \overline{x_2}) + z_{\frac{\alpha}{2}} \cdot \sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}$$

The following mean differences were obtained from Iris data experiments reported with the corresponding 95% confidence intervals ($\alpha = 0.05$):

Lower bound	-0.305903	-0.117114	-0.579569	-0.248382
Difference	-0.060282	-0.007240	-0.044441	-0.005211
Upper bound	0.185339	0.102633	0.490687	0.237959

Table 17. The difference of means between two neurons

These results show that 0 falls within each confidence interval obtained by applying the above formulas. This indicates that there is no statistical difference in the means; the means of the two neurons are the same.

Based on the above statistical analysis, it is apparent that each feature in the two neurons share the same distribution and the same means, and the average difference between them is closed to 0. Since all the criteria are fulfilled, this evidence supports the hypothesis that the Python package is working in the same way as the R package.

4.4.2 Evaluating the Starburst Representation of the SOM Model

Based on the same *neurons* and *visual* (the outcome from model training function), and with the same value of arguments, the starburst function from the R-based and Python-based packages expect to report the same heat map and connected components as well. In order to guarantee the input consistency, the model is trained in the R-based POPSOM package, and then the *neurons* and *visual* result are shared with the Python-based package. Results are plotted as starburst representations of the model by the R-based package (Figure 57) and the Python-based package (Figure 58) respectively as follow.

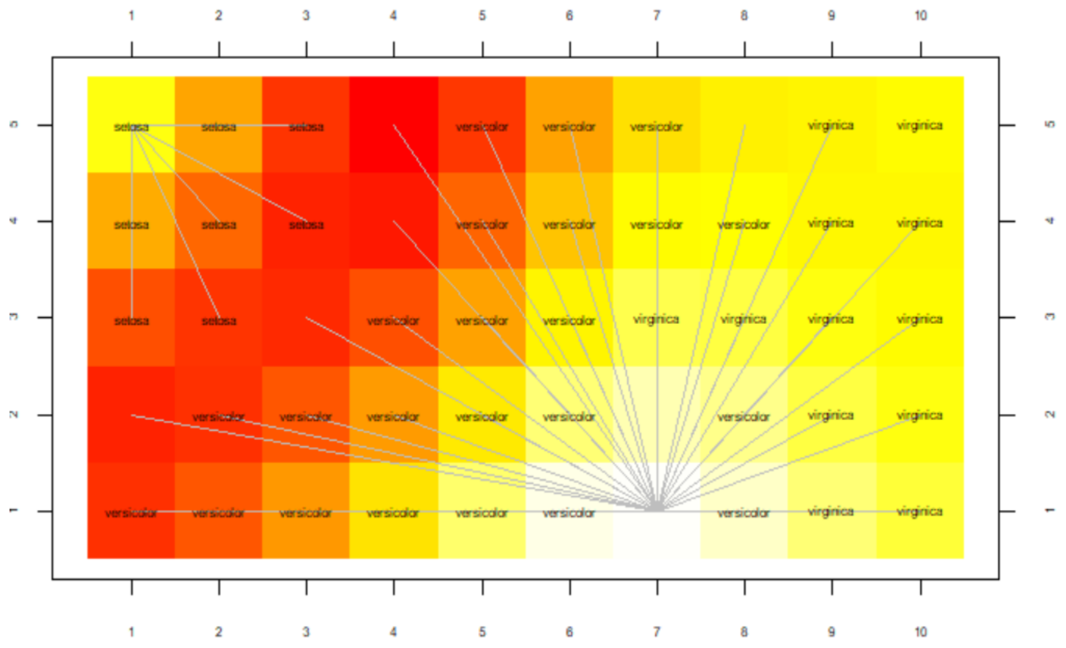


Figure 55. Starburst Representation of the SOM model in R.

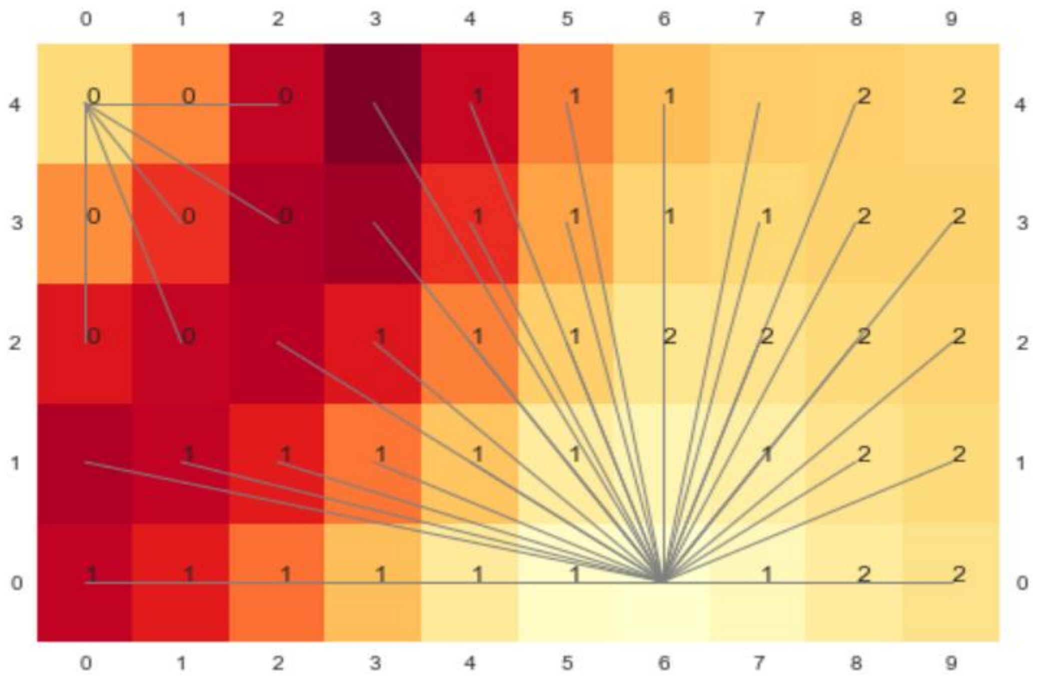


Figure 56. Starburst Representation of the SOM model in Python.

It is clear by visual comparison of the starburst representations that the heat map and connected components generated by R and Python are exactly the same.

4.4.3 Evaluating the Density Plot Function.

Plotting the density of training data overlaid with the neurons density for the same features is easy for the user to interpret the quality of the map. Plotting the same density representation by different programs (R-based and Python-based) is the best way to reveal any differences between the two programs. The following are side-by-side displays of the density plots from both the R-based and Python-based packages for each feature of the Iris data set (Figure 59).

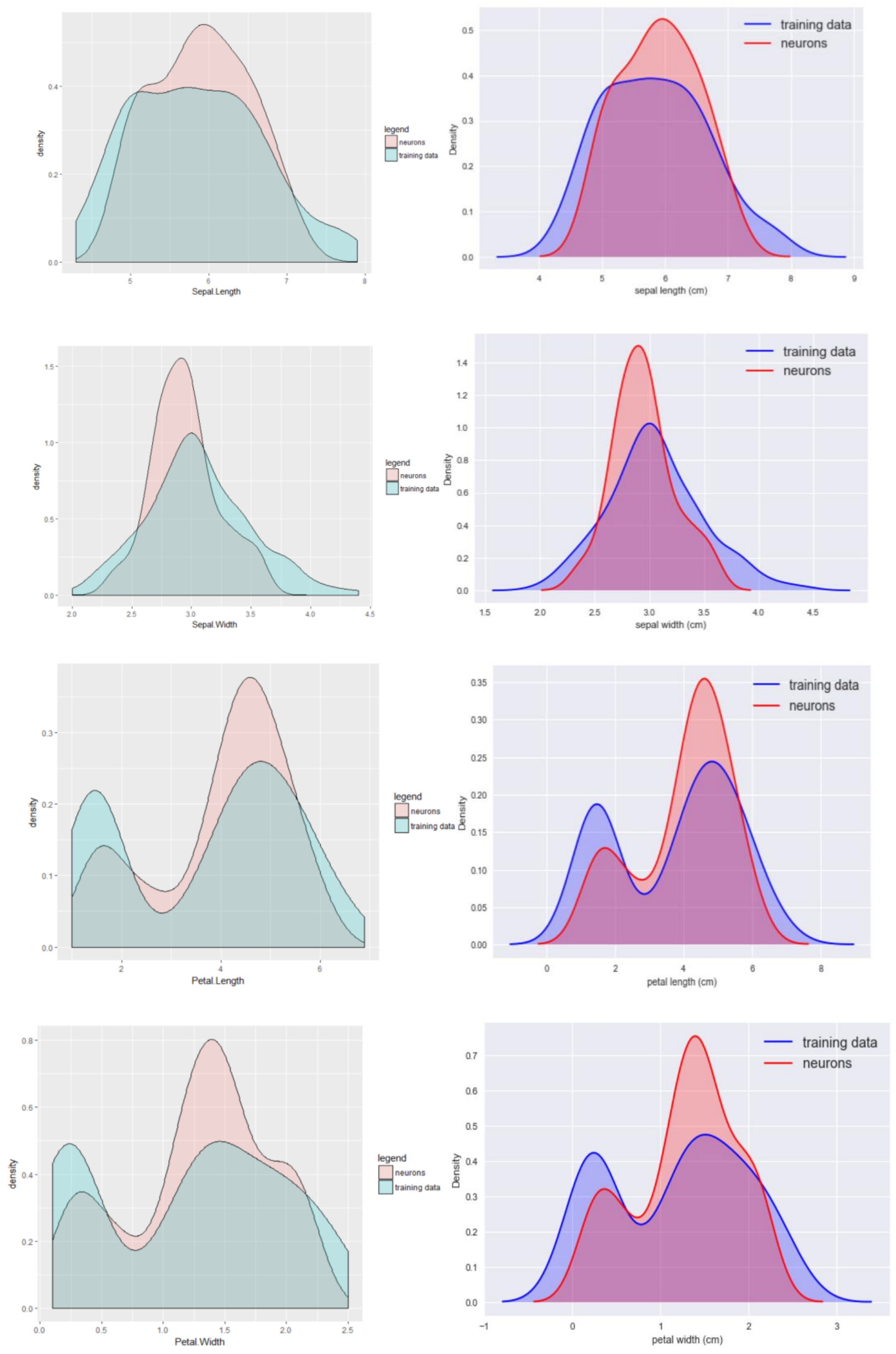


Figure 57. Plots of the marginal probability distribution of each feature of the Iris data set. R-based results are on the left, and Python-based results are on the right.

These four groups of density plots are evidence that there is no difference between the R-based package and the Python-based package for this function. And, this also indicates the Python-based package is working properly as the R-based package is validated as reliable and consistent.

CHAPTER 5

Conclusion

5.1 Conclusions

Compared with other existing SOM packages in the public repository, the R-based POPSOM package can be thought of as a one-stop solution for the Self-Organizing Maps. Besides the basic model construction and visualization function, it also provides a computationally efficient model evaluation function. This function evaluates the quality of the model in regards to the map embedding accuracy and estimated topographic accuracy at the same time. With the purpose of sharing the benefit of the R-based POPSOM package with Python users, this project successfully migrates the entire R-based POPSOM package into the Python-based package. Two different data sets (the Iris data set and the Wheat Seed data set) with different features from the UCI machine learning repository were utilized to demonstrate the function of the Python-based package. In addition to migrating the functions within the package, the entire package was refactored into an object-oriented programming paradigm and required splitting the *map.build* function into *__init__* and *fit* functions, which are both conventional function names in machine learning community. Furthermore, to guarantee the correctness and reliability of the Python-based package, the Python-based package was verified by utilizing three statistical approaches that are based on specific aforementioned comparisons of the outcomes from the R-based package and the Python-based package. The result of this benchmark is convincing evidence that the Python-based package is as trustable as the R-based package.

5.2 Future Works

5.2.1 Submit the Python-based Package to Public Repository

With the purpose of benefiting Python users with the SOM algorithm, the R-based POPSOM package has been distributed as free software, so that everyone can use, modify and redistribute it under the terms of the GNU General Public License (published by the Free Software Foundation). PyPI (the Python Package Index) is a public repository of software for the Python programming language. It contains 128882 packages as of 2/7/2018. Submitting the Python-based package used here to the PyPI community will allow Python users to utilize these findings in their research.

In addition, the Python-based package can be published on GitHub as open-source software to increase exposure to it as the R-based POPSOM package has been added for R users.

5.2.2 Using animation to simulate the formation of the model.

Currently, what is obtained from the POPSOM package is a graphical report (starburst representation) of a SOM model. Although the algorithm is not difficult to understand, the end users without basic quantitative knowledge still view this algorithm as a black box and might wonder what is happening during the training. Presenting the process of the heat map formation may give the user knowledge of the internal mechanisms and make the magical box more transparent. This, in turn, gives evidence to the end user that the algorithm is not a magic trick, but rather a reliable, predicable and replicable process.

APPENDICES

Source Code (popsom.py)

```
import sys
import numpy as np
import pandas as pd
import seaborn as sns
import vsom            # Call vsom.f90 (Fortran package)
from random import randint
from sklearn.metrics.pairwise import euclidean_distances
import statsmodels.stats.api as sms     # t-test
import statistics as stat               # F-test
from scipy import stats                 # KS Test
from scipy.stats import f               # F-test
from itertools import combinations
import matplotlib.pyplot as plt
import time              # Calculate the training time

class map:
     def __init__(self, xdim=10, ydim=5, alpha=.3,
train=1000, algorithm="som", norm=False):
          """ __init__ -- Initialize the Model

               parameters:
               - xdim,ydim - the dimensions of the map
               - alpha - the learning rate, should be a
positive non-zero real number
               - train - number of training iterations
               - algorithm - selection switch (som and
som_f)
               - norm - normalize the input data space
     """
          self.xdim = xdim
          self.ydim = ydim
          self.alpha = alpha
          self.train = train
          self.algorithm = algorithm
          self.norm = norm

     def fit(self, data, labels):
          """ fit -- Train the Model with Python or
Fortran

               parameters:
                - data - a dataframe where each row con-
tains an unlabeled training instance
                - labels - a vector or dataframe with one
label for each observation in data
     """

          start_time = time.time()
          if self.norm:
               data = data.div(data.sum(axis=1), axis=0)

          self.data = data
          self.labels = labels

          algorithms = ["som", "som_f"]

         # check if the dims are reasonable
          if (self.xdim < 3 or self.ydim < 3):
               sys.exit("build: map is too small.")

          try:
               _index_algorithm = algo-
rithms.index(self.algorithm)
          except ValueError:
               sys.exit("build only supports
'som','som_f'")

          if _index_algorithm == 0:        # train the
model with python
               self.vsom_p()

          elif _index_algorithm == 1:      # train the
model with Fortran
               self.vsom_f()

          else:
               sys.exit("build only supports
'som','som_f'")

          visual = []

          for i in range(self.data.shape[0]):
               b = self.best_match(self.data.iloc[[i]])
               visual.extend([b])

          self.visual = visual
          # print the number of iteration and training
time
          # print(str(self.train) + "  Execute %s seconds
" % (time.time() - start_time))

     def marginal(self, marginal):
          """ marginal -- plot that shows the marginal
probability distribution of the neurons and data

               parameters:
               - marginal is the name of a training data
frame dimension or index
          """

          # check if the second argument is of type char-
acter
          if type(marginal) == str and marginal in
list(self.data):

               f_ind = list(self.data).index(marginal)
               f_name = marginal
               train = np.matrix(self.data)[:, f_ind]
               neurons = self.neurons[:, f_ind]
               plt.ylabel('Density')
               plt.xlabel(f_name)
               sns.kdeplot(np.ravel(train),
                           label="training data",
                             shade=True,
                             color="b")
               sns.kdeplot(neurons, label="neurons",
shade=True, color="r")
               plt.legend(fontsize=15)
               plt.show()

          elif (type(marginal) == int and marginal <
len(list(self.data)) and marginal >= 0):

               f_ind = marginal
               f_name = list(self.data)[marginal]
               train = np.matrix(self.data)[:, f_ind]
               neurons = self.neurons[:, f_ind]
               plt.ylabel('Density')
               plt.xlabel(f_name)
               sns.kdeplot(np.ravel(train),
                             label="training data",
                             shade=True,
                             color="b")
               sns.kdeplot(neurons, label="neurons",
shade=True, color="r")
               plt.legend(fontsize=15)
               plt.show()

          else:
               sys.exit("marginal: second argument is not
the name of a training \
                              data frame dimension or in-
dex")

     def vsom_p(self):
          """ vsom_p -- vectorized, unoptimized version
of the stochastic SOM
                      training algorithm written entirely
in python
     """
     # some constants
          dr = self.data.shape[0]
          dc = self.data.shape[1]
          nr = self.xdim*self.ydim
          nc = dc  # dim of data and neurons is the same

         # build and initialize the matrix holding the
neurons
          cells = nr * nc  # No. of neurons times number
of data dimensions

         # vector with small init values for all neurons
          v = np.random.uniform(-1, 1, cells)

         # NOTE: each row represents a neuron, each col-
umn represents a dimension.
          neurons = np.transpose(np.reshape(v, (nc, nr)))
# rearrange the vector as matrix

          # neurons = np.reshape(v, (nr, nc)) # Another
option to reshape

         # compute the initial neighborhood size and step
          nsize = max(self.xdim, self.ydim) + 1
          nsize_step = np.ceil(self.train/nsize)
          step_counter = 0  # counts the number of epochs
per nsize_step

         # convert a 1D rowindex into a 2D map coordinate
          def coord2D(rowix):
               x = np.array(rowix) % self.xdim
               y = np.array(rowix) // self.xdim

               return np.concatenate((x, y))

         # constants for the Gamma function
          m = [i for i in range(nr)]  # a vector with all
neuron 1D addresses

         # x-y coordinate of ith neuron: m2Ds[i,] = c(xi,
yi)
          m2Ds =
np.matrix.transpose(coord2D(m).reshape(2, nr))

         # neighborhood function
          def Gamma(c):

             # lookup the 2D map coordinate for c
               c2D = m2Ds[c, ]
             # a matrix with each row equal to c2D
               c2Ds = np.outer(np.linspace(1, 1, nr),
c2D)
             # distance vector of each neuron from c in
terms of map coords!
               d = np.sqrt(np.dot((c2Ds - m2Ds)**2, [1,
1]))
             # if m on the grid is in neigh then alpha
else 0.0
               hood = np.where(d < nsize*1.5, self.alpha,
0.0)

               return hood

         # training #
         # the epochs loop

          for epoch in range(self.train):

             # hood size decreases in disrete nsize.steps
               step_counter = step_counter + 1
               if step_counter == nsize_step:

                    step_counter = 0
                    nsize = nsize - 1

             # create a sample training vector
```

```
               ix = randint(0, dr-1)
               # ix = (epoch+1) % dr   # For Debugging
               xk = self.data.iloc[[ix]]

             # competitive step
               xk_m = np.outer(np.linspace(1, 1, nr), xk)

               diff = neurons - xk_m
               squ = diff * diff
               s = np.dot(squ, np.linspace(1, 1, nc))
               o = np.argsort(s)
               c = o[0]

             # update step
               gamma_m = np.outer(Gamma(c),
np.linspace(1, 1, nc))
               neurons = neurons - diff * gamma_m

          self.neurons = neurons

     def vsom_f(self):
          """ vsom_f -- vectorized and optimized version
of the stochastic
                         SOM training algorithm written
in Fortran90
          """

         # some constants
          dr = self.data.shape[0]
          dc = self.data.shape[1]
          nr = self.xdim*self.ydim
          nc = dc  # dim of data and neurons is the same

         # build and initialize the matrix holding the
neurons
          cells = nr * nc        # no. of neurons times
number of data dimensions

         # vector with small init values for all neurons
          v = np.random.uniform(-1, 1, cells)

         # NOTE: each row represents a neuron, each col-
umn represents a dimension.
          neurons = np.transpose(np.reshape(v, (nc, nr)))
# rearrange the vector as matrix

          neurons = vsom.vsom(neurons,
                              np.array(self.data),
                              self.xdim,
                              self.ydim,
                              self.alpha,
                              self.train,
                              dr,
                              dc)

          self.neurons = neurons

     def convergence(self, conf_int=.95, k=50,
verb=False, ks=False):
          """ convergence -- the convergence index of a
map

               Parameters:
               - conf_int - the confidence interval of
the quality assessment (default 95%)
               - k - the number of samples used for the
estimated topographic accuracy computation
               - verb - if true reports the two conver-
gence components separately, otherwise it will
                        report the linear combination of
the two
               - ks - a switch, true for ks-test, false
for standard var and means test

               Return
               - return value is the convergence index
          """

          if ks:
               embed = self.embed_ks(conf_int,
verb=False)
          else:
               embed = self.embed_vm(conf_int,
verb=False)

          topo_ = self.topo(k, conf_int, verb=False, in-
terval=False)

          if verb:
               return {"embed": embed, "topo": topo_}
          else:
               return (0.5*embed + 0.5*topo_)
     def starburst(self, explicit=False, smoothing=2,
merge_clusters=True,  merge_range=.25):
          """ starburst -- compute and display the star-
burst representation of clusters

               parameters:
               - explicit - controls the shape of the
connected components
               - smoothing - controls the smoothing level
of the umat (NULL,0,>0)
               - merge_clusters - a switch that controls
if the starburst clusters are merged together
               - merge_range - a range that is used as a
percentage of a certain distance in the code
                               to determine whether com-
ponents are closer to their centroids or
                               centroids closer to each
other.
          """

          umat = self.compute_umat(smoothing=smoothing)
          self.plot_heat(umat,
                              explicit=explicit,
                              comp=True,
                              merge=merge_clusters,
                              merge_range=merge_range)

     def compute_umat(self, smoothing=None):
          """ compute_umat -- compute the unified dis-
tance matrix

               parameters:
               - smoothing - is either NULL, 0, or a pos-
itive floating point value controlling the
                              smoothing of the umat repre-
sentation
               return:
               - a matrix with the same x-y dims as the
original map containing the umat values
          """

          d = euclidean_distances(self.neurons,
self.neurons)
          umat = self.compute_heat(d, smoothing)

          return umat
     def compute_heat(self, d, smoothing=None):
          """ compute_heat -- compute a heat value map 
representation of the given distance matrix

               parameters:
               - d - a distance matrix computed via the 
'dist' function
               - smoothing - is either NULL, 0, or a pos-
itive floating point value controlling the
                            smoothing of the umat repre-
sentation

               return:
               - a matrix with the same x-y dims as the 
original map containing the heat
          """

          x = self.xdim
          y = self.ydim
          heat = np.matrix([[0.0] * y for _ in range(x)])

          if x == 1 or y == 1:
               sys.exit("compute_heat: heat map can not 
be computed for a map \
                      with a dimension of 1")

          # this function translates our 2-dim map coor-
dinates
          # into the 1-dim coordinates of the neurons
          def xl(ix, iy):

               return ix + iy * x

          # check if the map is larger than 2 x 2 (other-
wise it is only corners)
          if x > 2 and y > 2:
               # iterate over the inner nodes and compute 
their umat values
               for ix in range(1, x-1):
                    for iy in range(1, y-1):
                         sum = (d[xl(ix, iy), xl(ix-1, 
iy-1)] +
                                d[xl(ix, iy), xl(ix, iy-
1)] +
                              d[xl(ix, iy), xl(ix+1, iy-1)] 
+
                              d[xl(ix, iy), xl(ix+1, iy)] +
                            d[xl(ix, iy), xl(ix+1, iy+1)]
+
                            d[xl(ix, iy), xl(ix, iy+1)] +
                            d[xl(ix, iy), xl(ix-1, iy+1)]
+
                            d[xl(ix, iy), xl(ix-1, iy)])

                         heat[ix, iy] = sum/8

               # iterate over bottom x axis
               for ix in range(1, x-1):
                    iy = 0
                    sum = (d[xl(ix, iy), xl(ix+1, iy)] +
                        d[xl(ix, iy), xl(ix+1, iy+1)] +
                        d[xl(ix, iy), xl(ix, iy+1)] +
                        d[xl(ix, iy), xl(ix-1, iy+1)] +
                        d[xl(ix, iy), xl(ix-1, iy)])

                    heat[ix, iy] = sum/5

               # iterate over top x axis
               for ix in range(1, x-1):
                    iy = y-1
                    sum = (d[xl(ix, iy), xl(ix-1, iy-1)]
+
                        d[xl(ix, iy), xl(ix, iy-1)] +
                        d[xl(ix, iy), xl(ix+1, iy-1)] +
                        d[xl(ix, iy), xl(ix+1, iy)] +
                        d[xl(ix, iy), xl(ix-1, iy)])

                    heat[ix, iy] = sum/5

               # iterate over the left y-axis
               for iy in range(1, y-1):
                    ix = 0
                    sum = (d[xl(ix, iy), xl(ix, iy-1)] +
                        d[xl(ix, iy), xl(ix+1, iy-1)] +
                        d[xl(ix, iy), xl(ix+1, iy)] +
                        d[xl(ix, iy), xl(ix+1, iy+1)] +
                        d[xl(ix, iy), xl(ix, iy+1)])

                    heat[ix, iy] = sum/5

               # iterate over the right y-axis
               for iy in range(1, y-1):
                    ix = x-1
                    sum = (d[xl(ix, iy), xl(ix-1, iy-1)]
+
                        d[xl(ix, iy), xl(ix, iy-1)] +
                        d[xl(ix, iy), xl(ix, iy+1)] +
                        d[xl(ix, iy), xl(ix-1, iy+1)] +
                        d[xl(ix, iy), xl(ix-1, iy)])

                    heat[ix, iy] = sum/5

          # compute umat values for corners
          if x >= 2 and y >= 2:
               # bottom left corner
               ix = 0
               iy = 0
               sum = (d[xl(ix, iy), xl(ix+1, iy)] +
                    d[xl(ix, iy), xl(ix+1, iy+1)] +
                    d[xl(ix, iy), xl(ix, iy+1)])

               heat[ix, iy] = sum/3

               # bottom right corner
               ix = x-1
               iy = 0
               sum = (d[xl(ix, iy), xl(ix, iy+1)] +
                    d[xl(ix, iy), xl(ix-1, iy+1)] +
                    d[xl(ix, iy), xl(ix-1, iy)])
               heat[ix, iy] = sum/3

               # top left corner
               ix = 0
               iy = y-1
               sum = (d[xl(ix, iy), xl(ix, iy-1)] +
                    d[xl(ix, iy), xl(ix+1, iy-1)] +
                    d[xl(ix, iy), xl(ix+1, iy)])
               heat[ix, iy] = sum/3

               # top right corner
               ix = x-1
               iy = y-1
               sum = (d[xl(ix, iy), xl(ix-1, iy-1)] +
                    d[xl(ix, iy), xl(ix, iy-1)] +
                    d[xl(ix, iy), xl(ix-1, iy)])
               heat[ix, iy] = sum/3

          # smooth the heat map
          pts = []
          for i in range(y):
               for j in range(x):
                    pts.extend([[j, i]])

          if smoothing is not None:
               if smoothing == 0:
                    heat = self.smooth_2d(heat,
                                               nrow=x,
                                               ncol=y,
                                               sur-
face=False)
               elif smoothing > 0:
                    heat = self.smooth_2d(heat,
                                               nrow=x,
                                               ncol=y,
                                               sur-
face=False,
                                               the-
ta=smoothing)
               else:
                    sys.exit("compute_heat: bad value for
smoothing parameter")

          return heat

     def plot_heat(self, heat, explicit=False, comp=True,
merge=False, merge_range=0.25):
          """ plot_heat -- plot a heat map based on a
'map', this plot also contains the connected
                           components of the map based on
the landscape of the heat map

               parameters:
               - heat - is a 2D heat map of the map re-
turned by 'map'
               - explicit - controls the shape of the
connected components
               - comp - controls whether we plot the con-
nected components on the heat map
               - merge - controls whether we merge the
starbursts together.
               - merge_range - a range that is used as a
percentage of a certain distance in the code
                               to determine whether com-
ponents are closer to their centroids or
                               centroids closer to each
other.
```

```
          """

          umat = heat

          x = self.xdim
          y = self.ydim
          nobs = self.data.shape[0]
          count = np.matrix([[0]*y]*x)

          # need to make sure the map doesn't have a di-
mension of 1
          if (x <= 1 or y <= 1):
               sys.exit("plot_heat: map dimensions too
small")

          tmp = pd.cut(heat, bins=100, labels=False)

          tmp_1 = np.array(np.matrix.transpose(tmp))
          fig, ax = plt.subplots()
          ax.pcolor(tmp_1, cmap=plt.cm.YlOrRd)
          ax.set_xticks(np.arange(x)+0.5, minor=False)
          ax.set_yticks(np.arange(y)+0.5, minor=False)
          plt.xlabel("x")
          plt.ylabel("y")
          ax.set_xticklabels(np.arange(x), minor=False)
          ax.set_yticklabels(np.arange(y), minor=False)
          ax.xaxis.set_tick_params(labeltop='on')
          ax.yaxis.set_tick_params(labelright='on')

          # put the connected component lines on the map
          if comp:
               if not merge:
                    # find the centroid for each neuron
on the map
                    centroids =
self.compute_centroids(heat, explicit)
               else:
                    # find the unique centroids for the
neurons on the map
                    centroids =
self.compute_combined_clusters(umat, explicit,
merge_range)

               # connect each neuron to its centroid
               for ix in range(x):
                    for iy in range(y):
                         cx = centroids['centroid_x'][ix,
iy]
                         cy = centroids['centroid_y'][ix,
iy]
                         plt.plot([ix+0.5, cx+0.5],
                              [iy+0.5, cy+0.5],
                              color='grey',
                              linestyle='-',
                              linewidth=1.0)

          # put the labels on the map if available
          if not (self.labels is None) and
(len(self.labels) != 0):

               # count the labels in each map cell
               for i in range(nobs):

                    nix = self.visual[i]
                    c = self.coordinate(nix)
                    ix = c[0]
                    iy = c[1]

                    count[ix-1, iy-1] = count[ix-1, iy-
1]+1

               for i in range(nobs):

                    c = self.coordinate(self.visual[i])
                    ix = c[0]
                    iy = c[1]

                    # we only print one label per cell
                    if count[ix-1, iy-1] > 0:

                         count[ix-1, iy-1] = 0
                         ix = ix - .5
                         iy = iy - .5
                         l = self.labels[i]
                         plt.text(ix+1, iy+1, l)

          plt.show()

     def compute_centroids(self, heat, explicit=False):
          """ compute_centroids -- compute the centroid
for each point on the map

               parameters:
```

```
               - heat - is a matrix representing the heat
map representation
               - explicit - controls the shape of the
connected component

               return value:
               - a list containing the matrices with the
same x-y dims as the original map containing the centroid
x-y coordinates
          """

          xdim = self.xdim
          ydim = self.ydim
          centroid_x = np.matrix([[-1] * ydim for _ in
range(xdim)])
          centroid_y = np.matrix([[-1] * ydim for _ in
range(xdim)])

          heat = np.matrix(heat)

          def compute_centroid(ix, iy):
               # recursive function to find the centroid
of a point on the map

               if (centroid_x[ix, iy] > -1) and (cen-
troid_y[ix, iy] > -1):
                    return {"bestx": centroid_x[ix, iy],
"besty": centroid_y[ix, iy]}

               min_val = heat[ix, iy]
               min_x = ix
               min_y = iy

               # (ix, iy) is an inner map element
               if ix > 0 and ix < xdim-1 and iy > 0 and
iy < ydim-1:

                    if heat[ix-1, iy-1] < min_val:
                         min_val = heat[ix-1, iy-1]
                         min_x = ix-1
                         min_y = iy-1

                    if heat[ix, iy-1] < min_val:
                         min_val = heat[ix, iy-1]
                         min_x = ix
                         min_y = iy-1
```

```
     if heat[ix+1, iy-1] < min_val:
          min_val = heat[ix+1, iy-1]
          min_x = ix+1
          min_y = iy-1

     if heat[ix+1, iy] < min_val:
          min_val = heat[ix+1, iy]
          min_x = ix+1
          min_y = iy

     if heat[ix+1, iy+1] < min_val:
          min_val = heat[ix+1, iy+1]
          min_x = ix+1
          min_y = iy+1

     if heat[ix, iy+1] < min_val:
          min_val = heat[ix, iy+1]
          min_x = ix
          min_y = iy+1

     if heat[ix-1, iy+1] < min_val:
          min_val = heat[ix-1, iy+1]
          min_x = ix-1
          min_y = iy+1

     if heat[ix-1, iy] < min_val:
          min_val = heat[ix-1, iy]
          min_x = ix-1
          min_y = iy

# (ix, iy) is bottom left corner
elif ix == 0 and iy == 0:

     if heat[ix+1, iy] < min_val:
          min_val = heat[ix+1, iy]
          min_x = ix+1
          min_y = iy

     if heat[ix+1, iy+1] < min_val:
          min_val = heat[ix+1, iy+1]
          min_x = ix+1
          min_y = iy+1

     if heat[ix, iy+1] < min_val:
          min_val = heat[ix, iy+1]
          min_x = ix
          min_y = iy+1
```

```
# (ix, iy) is bottom right corner
elif ix == xdim-1 and iy == 0:

     if heat[ix, iy+1] < min_val:
          min_val = heat[ix, iy+1]
          min_x = ix
          min_y = iy+1

     if heat[ix-1, iy+1] < min_val:
          min_val = heat[ix-1, iy+1]
          min_x = ix-1
          min_y = iy+1

     if heat[ix-1, iy] < min_val:
          min_val = heat[ix-1, iy]
          min_x = ix-1
          min_y = iy

# (ix, iy) is top right corner
elif ix == xdim-1 and iy == ydim-1:

     if heat[ix-1, iy-1] < min_val:
          min_val = heat[ix-1, iy-1]
          min_x = ix-1
          min_y = iy-1

     if heat[ix, iy-1] < min_val:
          min_val = heat[ix, iy-1]
          min_x = ix
          min_y = iy-1

     if heat[ix-1, iy] < min_val:
          min_val = heat[ix-1, iy]
          min_x = ix-1
          min_y = iy

# (ix, iy) is top left corner
elif ix == 0 and iy == ydim-1:

     if heat[ix, iy-1] < min_val:
          min_val = heat[ix, iy-1]
          min_x = ix
          min_y = iy-1

     if heat[ix+1, iy-1] < min_val:
          min_val = heat[ix+1, iy-1]
          min_x = ix+1
          min_y = iy-1

     if heat[ix+1, iy] < min_val:
          min_val = heat[ix+1, iy]
          min_x = ix+1
          min_y = iy

# (ix, iy) is a left side element
elif ix == 0 and iy > 0 and iy < ydim-1:

     if heat[ix, iy-1] < min_val:
          min_val = heat[ix, iy-1]
          min_x = ix
          min_y = iy-1

     if heat[ix+1, iy-1] < min_val:
          min_val = heat[ix+1, iy-1]
          min_x = ix+1
          min_y = iy-1

     if heat[ix+1, iy] < min_val:
          min_val = heat[ix+1, iy]
          min_x = ix+1
          min_y = iy

     if heat[ix+1, iy+1] < min_val:
          min_val = heat[ix+1, iy+1]
          min_x = ix+1
          min_y = iy+1

     if heat[ix, iy+1] < min_val:
          min_val = heat[ix, iy+1]
          min_x = ix
          min_y = iy+1

# (ix, iy) is a bottom side element
elif ix > 0 and ix < xdim-1 and iy == 0:

     if heat[ix+1, iy] < min_val:
          min_val = heat[ix+1, iy]
          min_x = ix+1
          min_y = iy

     if heat[ix+1, iy+1] < min_val:
          min_val = heat[ix+1, iy+1]
          min_x = ix+1
```

```
                         min_y = iy+1

                    if heat[ix, iy+1] < min_val:
                         min_val = heat[ix, iy+1]
                         min_x = ix
                         min_y = iy+1

                    if heat[ix-1, iy+1] < min_val:
                         min_val = heat[ix-1, iy+1]
                         min_x = ix-1
                         min_y = iy+1

                    if heat[ix-1, iy] < min_val:
                         min_val = heat[ix-1, iy]
                         min_x = ix-1
                         min_y = iy

               # (ix, iy) is a right side element
               elif ix == xdim-1 and iy > 0 and iy <
ydim-1:

                    if heat[ix-1, iy-1] < min_val:
                         min_val = heat[ix-1, iy-1]
                         min_x = ix-1
                         min_y = iy-1

                    if heat[ix, iy-1] < min_val:
                         min_val = heat[ix, iy-1]
                         min_x = ix
                         min_y = iy-1

                    if heat[ix, iy+1] < min_val:
                         min_val = heat[ix, iy+1]
                         min_x = ix
                         min_y = iy+1

                    if heat[ix-1, iy+1] < min_val:
                         min_val = heat[ix-1, iy+1]
                         min_x = ix-1
                         min_y = iy+1

                    if heat[ix-1, iy] < min_val:
                         min_val = heat[ix-1, iy]
                         min_x = ix-1
                         min_y = iy

               # (ix, iy) is a top side element
               elif ix > 0 and ix < xdim-1 and iy ==
ydim-1:

                    if heat[ix-1, iy-1] < min_val:
                         min_val = heat[ix-1, iy-1]
                         min_x = ix-1
                         min_y = iy-1

                    if heat[ix, iy-1] < min_val:
                         min_val = heat[ix, iy-1]
                         min_x = ix
                         min_y = iy-1

                    if heat[ix+1, iy-1] < min_val:
                         min_val = heat[ix+1, iy-1]
                         min_x = ix+1
                         min_y = iy-1

                    if heat[ix+1, iy] < min_val:
                         min_val = heat[ix+1, iy]
                         min_x = ix+1
                         min_y = iy

                    if heat[ix-1, iy] < min_val:
                         min_val = heat[ix-1, iy]
                         min_x = ix-1
                         min_y = iy

             # if successful
             # move to the square with the smaller value,
i_e_, call
             # compute_centroid on this new square
             # note the RETURNED x-y coords in the cen-
troid_x and
             # centroid_y matrix at the current location
             # return the RETURNED x-y coordinates

               if min_x != ix or min_y != iy:
                    r_val = compute_centroid(min_x,
min_y)

                 # if explicit is set show the exact con-
nected component
                 # otherwise construct a connected compo-
nenent where all
                 # nodes are connected to a centrol node
                    if explicit:
                         centroid_x[ix, iy] = min_x
                         centroid_y[ix, iy] = min_y
                         return {"bestx": min_x, "besty":
min_y}

                    else:
                         centroid_x[ix, iy] =
r_val['bestx']
                         centroid_y[ix, iy] =
r_val['besty']
                         return r_val

               else:
                    centroid_x[ix, iy] = ix
                    centroid_y[ix, iy] = iy
                    return {"bestx": ix, "besty": iy}

          for i in range(xdim):
               for j in range(ydim):
                    compute_centroid(i, j)

          return {"centroid_x": centroid_x, "centroid_y":
centroid_y}

     def compute_combined_clusters(self, heat, explicit,
rang):

          # compute the connected components
          centroids = self.compute_centroids(heat, ex-
plicit)
          # Get unique centroids
          unique_centroids =
self.get_unique_centroids(centroids)
          # Get distance from centroid to cluster ele-
ments for all centroids
          within_cluster_dist =
self.distance_from_centroids(centroids,

                  unique_centroids,

                  heat)
          # Get average pairwise distance between clus-
ters
          between_cluster_dist =
self.distance_between_clusters(centroids,
                      unique_centroids,

                      heat)
          # Get a boolean matrix of whether two compo-
nents should be combined
          combine_cluster_bools =
self.combine_decision(within_cluster_dist,

            between_cluster_dist,

            rang)
          # Create the modified connected components grid
          ne_centroid =
self.new_centroid(combine_cluster_bools,
                                                    cen-
troids,

     unique_centroids)

          return ne_centroid

     def get_unique_centroids(self, centroids):
          """ get_unique_centroids -- a function that
computes a list of unique centroids from
                                      a matrix of cen-
troid locations.

               parameters:
               - centroids - a matrix of the centroid lo-
cations in the map
          """

          # get the dimensions of the map
          xdim = self.xdim
          ydim = self.ydim
          xlist = []
          ylist = []
          x_centroid = centroids['centroid_x']
          y_centroid = centroids['centroid_y']

          for ix in range(xdim):
               for iy in range(ydim):
                    cx = x_centroid[ix, iy]
                    cy = y_centroid[ix, iy]
          # Check if the x or y of the current centroid
is not in the list
          # and if not
          # append both the x and y coordinates to the
respective lists
                    if not(cx in xlist) or not(cy in
ylist):
                         xlist.append(cx)
                         ylist.append(cy)

          # return a list of unique centroid positions
          return {"position_x": xlist, "position_y":
ylist}

     def distance_from_centroids(self, centroids,
unique_centroids, heat):
          """ distance_from_centroids -- A function to
get the average distance from
                                        centroid by
cluster.
               parameters:
               - centroids - a matrix of the centroid lo-
cations in the map
               - unique_centroids - a list of unique cen-
troid locations
               - heat - a unified distance matrix
          """

          centroids_x_positions =
unique_centroids['position_x']
          centroids_y_positions =
unique_centroids['position_y']
          within = []

          for i in range(len(centroids_x_positions)):
               cx = centroids_x_positions[i]
               cy = centroids_y_positions[i]

               # compute the average distance
               distance = self.cluster_spread(cx, cy,
np.matrix(heat), centroids)

               # append the computed distance to the list
of distances
               within.append(distance)

          return within
     def cluster_spread(self, x, y, umat, centroids):
          """ cluster_spread -- Function to calculate the
average distance in
                                one cluster given one
centroid.

               parameters:
               - x - x position of a unique centroid
               - y - y position of a unique centroid
               - umat - a unified distance matrix
               - centroids - a matrix of the centroid lo-
cations in the map
          """

          centroid_x = x
          centroid_y = y
          sum = 0
          elements = 0
          xdim = self.xdim
          ydim = self.ydim
          centroid_weight = umat[centroid_x, centroid_y]

          for xi in range(xdim):
               for yi in range(ydim):
                    cx = centroids['centroid_x'][xi, yi]
                    cy = centroids['centroid_y'][xi, yi]

                    if(cx == centroid_x and cy == cen-
troid_y):
                         cweight = umat[xi, yi]
                         sum = sum + abs(cweight - cen-
troid_weight)
                         elements = elements + 1

          average = sum / elements

          return average

     def distance_between_clusters(self, centroids,
unique_centroids, umat):
          """ distance_between_clusters -- A function to
compute the average pairwise
                                             distance be-
tween clusters.

               parameters:
                - centroids - a matrix of the centroid lo-
cations in the map
                - unique_centroids - a list of unique cen-
troid locations
                - umat - a unified distance matrix
          """

          cluster_elements =
self.list_clusters(centroids, unique_centroids, umat)

          tmp_1 =
np.zeros(shape=(max([len(cluster_elements[i]) for i in
range(
                    len(cluster_elements))]),
len(cluster_elements)))

          for i in range(len(cluster_elements)):
               for j in range(len(cluster_elements[i])):
                    tmp_1[j, i] = cluster_elements[i][j]

          columns = tmp_1.shape[1]

          tmp =
np.matrix.transpose(np.array(list(combinations([i for i
in range(columns)], 2))))

          tmp_3 = np.zeros(shape=(tmp_1.shape[0],
tmp.shape[1]))

          for i in range(tmp.shape[1]):
               tmp_3[:, i] = np.where(tmp_1[:, tmp[1,
i]]*tmp_1[:, tmp[0, i]] != 0,
                                             abs(tmp_1[:,
tmp[0, i]]-tmp_1[:, tmp[1, i]]), 0)
             # both are not equals 0

          mean = np.true_divide(tmp_3.sum(0), (tmp_3 !=
0).sum(0))
          index = 0
          mat = np.zeros((columns, columns))

          for xi in range(columns-1):
               for yi in range(xi, columns-1):
                    mat[xi, yi + 1] = mean[index]
                    mat[yi + 1, xi] = mean[index]
                    index = index + 1
          return mat

     def list_clusters(self, centroids, unique_centroids,
umat):
          """ list_clusters -- A function to get the
clusters as a list of lists.

               parameters:
               - centroids - a matrix of the centroid lo-
cations in the map
               - unique_centroids - a list of unique cen-
troid locations
               - umat - a unified distance matrix
          """

          centroids_x_positions =
unique_centroids['position_x']
          centroids_y_positions =
unique_centroids['position_y']
          cluster_list = []

          for i in range(len(centroids_x_positions)):
               cx = centroids_x_positions[i]
               cy = centroids_y_positions[i]

         # get the clusters associated with a unique cen-
troid and store it in a list
               clus-
ter_list.append(self.list_from_centroid(cx, cy, cen-
troids, umat))

          return cluster_list

     def list_from_centroid(self, x, y, centroids, umat):
          """ list_from_centroid -- A funtion to get all
cluster elements
                                          associated to one
centroid.

               parameters:
               - x - the x position of a centroid
               - y - the y position of a centroid
               - centroids - a matrix of the centroid lo-
cations in the map
               - umat - a unified distance matrix
          """
          centroid_x = x
          centroid_y = y
          xdim = self.xdim
          ydim = self.ydim

          cluster_list = []
          for xi in range(xdim):
               for yi in range(ydim):
                    cx = centroids['centroid_x'][xi, yi]
                    cy = centroids['centroid_y'][xi, yi]

                    if(cx == centroid_x and cy == cen-
troid_y):
                         cweight = np.matrix(umat)[xi,
yi]
                         cluster_list.append(cweight)

          return cluster_list

     def combine_decision(self, within_cluster_dist, dis-
tance_between_clusters, rang):
          """ combine_decision -- A function that produc-
es a boolean matrix
                                  representing which
clusters should be combined.

               parameters:
               - within_cluster_dist - A list of the dis-
tances from centroid to cluster elements for all cen-
troids
               - distance_between_clusters - A list of
the average pairwise distance between clusters
               - range - The distance where the clusters
are merged together.
          """

          inter_cluster = distance_between_clusters
          centroid_dist = within_cluster_dist
          dim = inter_cluster.shape[1]
          to_combine = np.matrix([[False]*dim]*dim)

          for xi in range(dim):
               for yi in range(dim):
                    cdist = inter_cluster[xi, yi]
                    if cdist != 0:
                         rx = centroid_dist[xi] * rang
                         ry = centroid_dist[yi] * rang
                         if (cdist < centroid_dist[xi] +
rx or
                              cdist < centroid_dist[yi] +
ry):
                              to_combine[xi, yi] = True

          return to_combine

     def new_centroid(self, bmat, centroids,
unique_centroids):
          """ new_centroid -- A function to combine cen-
troids based on matrix of booleans.

               parameters:
               - bmat - a boolean matrix containing the
centroids to merge
               - centroids - a matrix of the centroid lo-
cations in the map
               - unique_centroids - a list of unique cen-
troid locations
          """

          bmat_rows = bmat.shape[0]
          bmat_columns = bmat.shape[1]
          centroids_x = unique_centroids['position_x']
          centroids_y = unique_centroids['position_y']
          components = centroids

          for xi in range(bmat_rows):
               for yi in range(bmat_columns):
                    if bmat[xi, yi]:
                         x1 = centroids_x[xi]
                         y1 = centroids_y[xi]
                         x2 = centroids_x[yi]
                         y2 = centroids_y[yi]
                         components =
self.swap_centroids(x1, y1, x2, y2, components)

          return components

     def swap_centroids(self, x1, y1, x2, y2, centroids):
          """ swap_centroids -- A function that changes
every instance of a centroid to
                                one that it should be
combined with.
               parameters:
                - centroids - a matrix of the centroid lo-
cations in the map
          """

          xdim = self.xdim
          ydim = self.ydim
          compn_x = centroids['centroid_x']
          compn_y = centroids['centroid_y']
          for xi in range(xdim):
               for yi in range(ydim):
                    if compn_x[xi, 0] == x1 and
compn_y[yi, 0] == y1:
                         compn_x[xi, 0] = x2
                         compn_y[yi, 0] = y2

          return {"centroid_x": compn_x, "centroid_y":
compn_y}

     def embed(self, conf_int=.95, verb=False, ks=False):
          """ embed -- evaluate the embedding of a map
using the F-test and
                         a Bayesian estimate of the vari-
ance in the training data.

               parameters:
               - conf_int - the confidence interval of
the convergence test (default 95%)
               - verb - switch that governs the return
value false: single convergence value
                          is returned, true: a vector of
individual feature congences is returned.

               - return value:
               - return is the cembedding of the map
(variance captured by the map so far)

               Hint:
                      the embedding index is the variance
of the training data captured by the map;
                     maps with convergence of less than
90% are typically not trustworthy.  Of course,
                     the precise cut-off depends on the
noise level in your training data.
          """

          if ks:
               return self.embed_ks(conf_int, verb)
          else:
               return self.embed_vm(conf_int, verb)

     def embed_ks(self, conf_int=0.95, verb=False):
          """ embed_ks -- using the kolgomorov-smirnov
test """

          # map_df is a dataframe that contains the neu-
rons
          map_df = self.neurons

          # data_df is a dataframe that contain the
training data
          data_df = np.array(self.data)

          nfeatures = map_df.shape[1]

          # use the Kolmogorov-Smirnov Test to test
whether the neurons and training
          # data appear
          # to come from the same distribution
          ks_vector = []
          for i in range(nfeatures):

     ks_vector.append(stats.mstats.ks_2samp(map_df[:, i],
data_df[:, i]))

          prob_v = self.significance(graphics=False)
          var_sum = 0

          # compute the variance captured by the map
          for i in range(nfeatures):

               # the second entry contains the p-value
               if ks_vector[i][1] > (1 - conf_int):
                    var_sum = var_sum + prob_v[i]
               else:
                    # not converged - zero out the proba-
bility
                    prob_v[i] = 0

          # return the variance captured by converged
features
          if verb:
               return prob_v
          else:
               return var_sum
     def embed_vm(self, conf_int=.95, verb=False):
          """ embed_vm -- using variance and mean tests
"""

          # map_df is a dataframe that contains the neu-
rons
          map_df = self.neurons

          # data_df is a dataframe that contain the
training data
          data_df = np.array(self.data)

          def df_var_test(df1, df2, conf=.95):

               if df1.shape[1] != df2.shape[1]:
                    sys.exit("df_var_test: cannot compare
variances of data frames")

               # init our working arrays
               var_ratio_v = [randint(1, 1) for _ in
range(df1.shape[1])]
               var_confintlo_v = [randint(1, 1) for _ in
range(df1.shape[1])]
               var_confinthi_v = [randint(1, 1) for _ in
range(df1.shape[1])]

               def var_test(x, y, ratio=1,
conf_level=0.95):

                    DF_x = len(x) - 1
                    DF_y = len(y) - 1
                    V_x = stat.variance(x.tolist())
                    V_y = stat.variance(y.tolist())

                    ESTIMATE = V_x / V_y

                    BETA = (1 - conf_level) / 2
                    CINT = [ESTIMATE / f.ppf(1 - BETA,
DF_x, DF_y),
                              ESTIMATE / f.ppf(BETA,
DF_x, DF_y)]

                    return {"estimate": ESTIMATE,
"conf_int": CINT}
```

```
              # compute the F-test on each feature in our
populations
               for i in range(df1.shape[1]):

                    t = var_test(df1[:, i], df2[:, i],
conf_level=conf)
                    var_ratio_v[i] = t['estimate']
                    var_confintlo_v[i] = t['conf_int'][0]
                    var_confinthi_v[i] = t['conf_int'][1]

               # return a list with the ratios and conf
intervals for each feature
               return {"ratio": var_ratio_v,
                         "conf_int_lo": var_confintlo_v,
                         "conf_int_hi": var_confinthi_v}

          def df_mean_test(df1, df2, conf=0.95):

               if df1.shape[1] != df2.shape[1]:
                    sys.exit("df_mean_test: cannot com-
pare means of data frames")

               # init our working arrays
               mean_diff_v = [randint(1, 1) for _ in
range(df1.shape[1])]
               mean_confintlo_v = [randint(1, 1) for _ in
range(df1.shape[1])]
               mean_confinthi_v = [randint(1, 1) for _ in
range(df1.shape[1])]

               def t_test(x, y, conf_level=0.95):
                    estimate_x = np.mean(x)
                    estimate_y = np.mean(y)
                    cm =
sms.CompareMeans(sms.DescrStatsW(x), sms.DescrStatsW(y))
                    conf_int_lo =
cm.tconfint_diff(alpha=1-conf_level, usevar='unequal')[0]
                    conf_int_hi =
cm.tconfint_diff(alpha=1-conf_level, usevar='unequal')[1]

                    return {"estimate": [estimate_x, es-
timate_y],
                              "conf_int": [conf_int_lo,
conf_int_hi]}

               # compute the F-test on each feature in
our populations
```

```
                for i in range(df1.shape[1]):
                     t = t_test(x=df1[:, i], y=df2[:, i],
conf_level=conf)
                     mean_diff_v[i] = t['estimate'][0] -
t['estimate'][1]
                     mean_confintlo_v[i] =
t['conf_int'][0]
                     mean_confinthi_v[i] =
t['conf_int'][1]

                # return a list with the ratios and conf
intervals for each feature
                return {"diff": mean_diff_v,
                          "conf_int_lo": mean_confintlo_v,
                          "conf_int_hi": mean_confinthi_v}
           # do the F-test on a pair of datasets
           vl = df_var_test(map_df, data_df, conf_int)

           # do the t-test on a pair of datasets
           ml = df_mean_test(map_df, data_df,
conf=conf_int)

           # compute the variance captured by the map --
           # but only if the means have converged as well.
           nfeatures = map_df.shape[1]
           prob_v = self.significance(graphics=False)
           var_sum = 0

           for i in range(nfeatures):

                if (vl['conf_int_lo'][i] <= 1.0 and
vl['conf_int_hi'][i] >= 1.0 and
                     ml['conf_int_lo'][i] <= 0.0 and
ml['conf_int_hi'][i] >= 0.0):

                     var_sum = var_sum + prob_v[i]
                else:
                     # not converged - zero out the proba-
bility
                     prob_v[i] = 0

           # return the variance captured by converged
features
           if verb:
                return prob_v
           else:
                return var_sum
```

```
     def topo(self, k=50, conf_int=.95, verb=False, in-
terval=True):
          """ topo -- measure the topographic accuracy of
the map using sampling

               parameters:
               - k - the number of samples used for the
accuracy computation
               - conf_int - the confidence interval of
the accuracy test (default 95%)
               - verb - switch that governs the return
value, false: single accuracy value
                          is returned, true: a vector of
individual feature accuracies is returned.
               - interval - a switch that controls wheth-
er the confidence interval is computed.

               - return value is the estimated topograph-
ic accuracy
          """


          # data.df is a matrix that contains the train-
ing data
          data_df = self.data

          if (k > data_df.shape[0]):
               sys.exit("topo: sample larger than train-
ing data.")

          data_sample_ix = [randint(1, data_df.shape[0])
for _ in range(k)]

          # compute the sum topographic accuracy - the
accuracy of a single sample
          # is 1 if the best matching unit is a neighbor
otherwise it is 0
          acc_v = []
          for i in range(k):

     acc_v.append(self.accuracy(data_df.iloc[data_sample_
ix[i]-1], data_sample_ix[i]))

          # compute the confidence interval values using
the bootstrap
          if interval:
               bval = self.bootstrap(conf_int, data_df,
k, acc_v)

          # the sum topographic accuracy is scaled by the
number of samples -
          # estimated
          # topographic accuracy
          if verb:
               return acc_v
          else:
               val = np.sum(acc_v)/k
               if interval:
                    return {'val': val, 'lo': bval['lo'],
'hi': bval['hi']}
               else:
                    return val

     def bootstrap(self, conf_int, data_df, k, sam-
ple_acc_v):
          """ bootstrap -- compute the topographic accu-
racies for the given confidence interval """

          ix = int(100 - conf_int*100)
          bn = 200

          bootstrap_acc_v = [np.sum(sample_acc_v)/k]

          for i in range(2, bn+1):

               bs_v = np.array([randint(1, k) for _ in
range(k)])-1
               a =
np.sum(list(np.array(sample_acc_v)[list(bs_v)]))/k
               bootstrap_acc_v.append(a)

          bootstrap_acc_sort_v = np.sort(bootstrap_acc_v)

          lo_val = bootstrap_acc_sort_v[ix-1]
          hi_val = bootstrap_acc_sort_v[bn-ix-1]

          return {'lo': lo_val, 'hi': hi_val}

     def accuracy(self, sample, data_ix):
          """ accuracy -- the topographic accuracy of a
single sample is 1 is the best matching unit
                         and the second best matching
unit are are neighbors otherwise it is 0
```

```
          """

          o = self.best_match(sample, full=True)
          best_ix = o[0]
          second_best_ix = o[1]

          # sanity check
          coord = self.coordinate(best_ix)
          coord_x = coord[0]
          coord_y = coord[1]

          map_ix = self.visual[data_ix-1]
          coord = self.coordinate(map_ix)
          map_x = coord[0]
          map_y = coord[1]

          if (coord_x != map_x or coord_y != map_y or
best_ix != map_ix):
               print("Error: best_ix: ", best_ix, "
map_ix: ", map_ix, "\n")

          # determine if the best and second best are
neighbors on the map
          best_xy = self.coordinate(best_ix)
          second_best_xy =
self.coordinate(second_best_ix)
          diff_map = np.array(best_xy) -
np.array(second_best_xy)
          diff_map_sq = diff_map * diff_map
          sum_map = np.sum(diff_map_sq)
          dist_map = np.sqrt(sum_map)

          # it is a neighbor if the distance on the map
          # between the bmu and 2bmu is less than 2,
should be 1 or 1.414
          if dist_map < 2:
               return 1
          else:
               return 0

     def best_match(self, obs, full=False):
          """ best_match -- given observation obs, return
the best matching neuron """

          # NOTE: replicate obs so that there are nr rows
of obs
```

```
          obs_m = np.tile(obs, (self.neurons.shape[0],
1))
          diff = self.neurons - obs_m
          squ = diff * diff
          s = np.sum(squ, axis=1)
          d = np.sqrt(s)
          o = np.argsort(d)

          if full:
               return o
          else:
               return o[0]

     def significance(self, graphics=True, fea-
ture_labels=False):
          """ significance -- compute the relative sig-
nificance of each feature and plot it

               parameters:
               - graphics - a switch that controls wheth-
er a plot is generated or not
               - feature_labels - a switch to allow the
plotting of feature names vs feature indices

               return value:
               - a vector containing the significance for
each feature
          """

          data_df = self.data
          nfeatures = data_df.shape[1]

         # Compute the variance of each feature on the
map
          var_v = [randint(1, 1) for _ in
range(nfeatures)]

          for i in range(nfeatures):
               var_v[i] = np.var(np.array(data_df)[:, i])

         # we use the variance of a feature as likelihood
of
         # being an important feature, compute the Bayes-
ian
         # probability of significance using uniform pri-
ors
```

```
          var_sum = np.sum(var_v)
          prob_v = var_v/var_sum

         # plot the significance
          if graphics:
               y = max(prob_v)

               plt.axis([0, nfeatures+1, 0, y])

               x = np.arange(1, nfeatures+1)
               tag = list(data_df)

               plt.xticks(x, tag)
               plt.yticks = np.linspace(0, y, 5)

               i = 1
               for xc in prob_v:
                    plt.axvline(x=i, ymin=0, ymax=xc)
                    i += 1

               plt.xlabel('Features')
               plt.ylabel('Significance')
               plt.show()
          else:
               return prob_v

     def projection(self):
          """ projection -- print the association of la-
bels with map elements

               parameters:

               return values:
               - a dataframe containing the projection
onto the map for each observation
          """

          labels_v = self.labels
          x_v = []
          y_v = []

          for i in range(len(labels_v)):

               ix = self.visual[i]
               coord = self.coordinate(ix)
               x_v.append(coord[0])
               y_v.append(coord[1])
          return pd.DataFrame({'labels': labels_v, 'x':
x_v, 'y': y_v})

     def neuron(self, x, y):
          """ neuron -- returns the contents of a neuron
at (x,y) on the map as a vector

               parameters:
                - x - map x-coordinate of neuron
                - y - map y-coordinate of neuron

               return value:
                - a vector representing the neuron
          """

          ix = self.rowix(x, y)
          return self.neurons[ix]

     def coordinate(self, rowix):
          """ coordinate -- convert from a row index to a
map xy-coordinate  """

          x = (rowix) % self.xdim
          y = (rowix) // self.xdim
          return [x, y]

     def rowix(self, x, y):
          """ rowix -- convert from a map xy-coordinate
to a row index  """

          rix = x + y*self.xdim
          return rix

     def smooth_2d(self, Y, ind=None, weight_obj=None,
grid=None, nrow=64, ncol=64, surface=True, theta=None):
          """ smooth_2d -- Kernel Smoother For Irregular
2-D Data """

          def exp_cov(x1, x2, theta=2, p=2, distMat=0):
               x1 = x1*(1/theta)
               x2 = x2*(1/theta)
               distMat = euclidean_distances(x1, x2)
               distMat = distMat**p
               return np.exp(-distMat)

          NN = [[1]*ncol] * nrow
          grid = {'x': [i for i in range(nrow)], "y": [i
for i in range(ncol)]}

          if weight_obj is None:
               dx = grid['x'][1] - grid['x'][0]
               dy = grid['y'][1] - grid['y'][0]
               m = len(grid['x'])
               n = len(grid['y'])
               M = 2 * m
               N = 2 * n
               xg = []

               for i in range(N):
                    for j in range(M):
                         xg.extend([[j, i]])

               xg = np.matrix(xg)

               center = []
               center.append([int(dx * M/2-1), int((dy *
N)/2-1)])

               out = exp_cov(xg,
np.matrix(center),theta=theta)
               out = np.matrix.transpose(np.reshape(out,
(N, M)))
               temp = np.zeros((M, N))
               temp[int(M/2-1)][int(N/2-1)] = 1

               wght = np.fft.fft2(out)/(np.fft.fft2(temp)
* M * N)
               weight_obj = {"m": m, "n": n, "N": N, "M":
M, "wght": wght}

          temp = np.zeros((weight_obj['M'],
weight_obj['N']))
          temp[0:m, 0:n] = Y
          temp2 = np.fft.ifft2(np.fft.fft2(temp) *

weight_obj['wght']).real[0:weight_obj['m'],

            0:weight_obj['n']]

          temp = np.zeros((weight_obj['M'],
weight_obj['N']))
          temp[0:m, 0:n] = NN
          temp3 = np.fft.ifft2(np.fft.fft2(temp) *
weight_obj['wght']).real[0:weight_obj['m'],
            0:weight_obj['n']]
          return temp2/temp3
```

LIST OF REFERENCE

[1] R Core Team, *R: A Language and Environment for Statistical Computing*. Vienna, Austria, R Foundation for Statistical Computing, 2017.

[2] Guido van Rossum, *Python Programming Language*, 2014.

[3] T. Kohonen, *Self-organizing maps*, Third edition.. ed. Berlin ; New York, Berlin ; New York : Springer, 2001.

[4] G.T. Breard, "Evaluating self-organizing map quality measures as convergence criteria," *Evaluating SOM quality measures*Thesis (M.S.)--University of Rhode Island, 2017 2017.

[5] B.H. Ott, "A convergence criterion for self-organizing maps,"DigitalCommons@URI 2012.

[6] V. Moosavi, S. Packmann and I. Vallés, "A python library for self organizing map (SOM),", 1/17/ 2018.

[7] Peter Wittek, Shi Chao Gao, Ik Soo Lim and Li Zhao, "Somoclu: An ecient parallel library for self-organizing maps,", 5/7/ 2013.

[8] Lutz Hamel, Benjamin Ott, Greg Breard,Robert Tatoian, Vishakh Gopu, "Popsom,", vol. 4.2 2017.

[9] L. Hamel, *SOM Quality Measures An Efficient Statistical Approach*, 2017.

[10] Anonymous *UCI machine learning repository: Iris data set* [Online]. available: http://archive.ics.uci.edu/ml/datasets/Iris. 2018

[11] Anonymous *UCI machine learning repository: Seeds Data Set* [Online]. available: http://archive.ics.uci.edu/ml/datasets/seeds. 2018

[12] H. Yin and N.M. Allinson, "On the distribution and convergence of feature space in self-organizing maps," *Neural Comput.*, vol. 7, no. 6, pp. 1178-1187 1995.

[13] R. Mayer, R. Neumayer, D. Baum and A. Rauber, "Analytic comparison of self-organising maps," *Advances in Self-Organizing Maps*, pp. 182-190 2009.

[14] G. Pölzlbauer, *Survey and comparison of quality measures for self-organizing maps*, na, 2004.

[15] L. Hamel and B. Ott, "A population based convergence criterion for self-organizing maps," in *Proceedings of the International Conference on Data Mining (DMIN)*, 2012, pp. 1.

[16] M.L. Rizzo, *Statistical computing with R*. Boca Raton [u.a.], Chapman & Hall/CRC, 2008.

[17] P. Goldsborough, "A tour of TensorFlow," 2016.

[18] A. Ohri, *Python for R Users*. Somerset, John Wiley & Sons, Incorporated, 2017.

[19] Anonymous *Student's T-Test* [Online]. available: https://www.rdocumentation.org/packages/stats/versions/3.4.3/topics/t.test. 2018

[20] Eric Jones, Travis Oliphant and Pearu Peterson. (-). *Open source scientific tools for Python* [Online]. available: http://www.scipy.org/. 2018

[21] Anonymous *Python statsmodels* [Online]. available: http://www.statsmodels.org/stable/index.html. 2018

[22] Anonymous *Var Test* [Online]. available: https://www.rdocumentation.org/packages/stats/versions/3.4.3/topics/var.test. 2018

[23] Anonymous *Python Statistics* [Online]. available: https://docs.python.org/3/library/statistics.html. 2018

[24] Anonymous *Python Scipy* [Online]. available: https://www.scipy.org/. 2018

[25] Anonymous *Kernel Smoother For Irregular 2-D Data* [Online]. available: https://www.rdocumentation.org/packages/fields/versions/9.0/topics/smooth.2d. 2018

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

[27] M. Abadi and Luca Cardelli, *A Theory of Objects*, Springer Verlag, 1998.

[28] K. Pearson. On Lines and Planes of Closest Fit to Systems of Points in Space. *Philosophical Magazine*. pp. 559–572.

[29] Magnus H-S Dahle, "A quick guide on how to use the fortran-to-python (F2PY) module,", Feb. 2015.

[30] I. Miller, *John E. Freund's mathematical statistics with applications*, 7th ed. / Irwin Miller, Marylees Miller.. ed. Upper Saddle River, NJ, Upper Saddle River, NJ : Prentice Hall, 2004.

BIBLIOGRAPHY

"Discrete Fourier Transform," , accessed 1/31/, 2018, https://docs.scipy.org/doc/numpy/reference/routines.fft.html.

"Fortran Pgoramming Language," , accessed 1/31/, 2018, https://en.wikipedia.org/wiki/Fortran.

"Kernel Smoother For Irregular 2-D Data," , accessed 1/30/, 2018, https://www.rdocumentation.org/packages/fields/versions/9.0/topics/smooth.2d.

"Math — Mathematical functions," , accessed 1/29/, 2018, https://docs.python.org/3/library/math.html.

"MinGW," , accessed 2/1/, 2018, https://en.wikipedia.org/wiki/MinGW#MinGW-w64.

"numpy.mean," , accessed 1/30/, 2018, https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.mean.html.

"PYPI Python Package Index," , accessed 2/7/, 2018, https://pypi.python.org/pypi.

"Python Keywords and Identifier," , accessed 1/29/, 2018, https://www.programiz.com/python-programming/keywords-identifier.

"Python Scipy," , accessed 1/30/, 2018, https://www.scipy.org/.

"Python Statistics," , accessed 1/30/, 2018, https://docs.python.org/3/library/statistics.html.

"Python statsmodels," , accessed 1/30/, 2018, http://www.statsmodels.org/stable/index.html.

"Python Statsmodels CompareMeans," , accessed 1/30/, 2018, http://www.statsmodels.org/dev/generated/statsmodels.stats.weightstats.CompareMeans.html.

"Python Statsmodels DescrStatsW," , accessed 1/30/, 2018, http://www.statsmodels.org/dev/generated/statsmodels.stats.weightstats.DescrStatsW.html.

"Reserved Words in R," , accessed 1/29/, 2018, https://stat.ethz.ch/R-manual/R-devel/library/base/html/Reserved.html.

"Scikit-learn," , accessed 1/31/, 2018, http://scikit-learn.org/stable/.

"Student's t-test," , accessed 1/30/, 2018, https://en.wikipedia.org/wiki/Student%27s_t-test.

"Student's T-Test," , accessed 1/30/, 2018, https://www.rdocumentation.org/packages/stats/versions/3.4.3/topics/t.test.

"UCI machine learning repository: Iris data set," , accessed Jan,16th, 2018, http://archive.ics.uci.edu/ml/datasets/Iris.

"UCI machine learning repository: Seeds Data Set," , accessed 2/3/, 2018, http://archive.ics.uci.edu/ml/datasets/seeds.

"UCI machine learning repository: Wine data set," , accessed Jan,16th, 2018, http://archive.ics.uci.edu/ml/datasets/Wine.

"UCI machine learning repository: Wine Quality Data Set&nbsp;" , accessed 2/2/, 2018, http://archive.ics.uci.edu/ml/datasets/Wine+Quality.

"Var Test," , accessed 1/30/, 2018, https://www.rdocumentation.org/packages/stats/versions/3.4.3/topics/var.test.

M. Abadi and Luca Cardelli, *A Theory of Objects*:Springer Verlag, 1998.

G. T. Breard, "Evaluating self-organizing map quality measures as convergence criteria,"Thesis (M.S.)--University of Rhode Island, 2017, , 2017.

D. H. Brown, *Cartogram data projection for self-organizing maps*:DigitalCommons@URI, 2012.

Y. Cheng, *Neural Computation* 1997.

P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis, *Modeling wine preferences by data mining from physicochemical properties* 2009.

Eric Jones, Travis Oliphant and Pearu Peterson, "Open source scientific tools for Python," , accessed 1/30/, 2018, http://www.scipy.org/.

P. Goldsborough. "A Tour of TensorFlow." . 2016.

Guido van Rossum, *Python Programming Language* 2014.

L. Hamel, *SOM Quality Measures An Efficient Statistical Approach* 2017.

L. Hamel and C. Brown, *Practical Tools for Self-Organizing Maps*.

L. Hamel and B. Ott, "A population based convergence criterion for self-organizing maps," in Proceedings of the International Conference on Data Mining (DMIN): The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp)2012, pp. 1.

T. Kohonen, *Self-organizing maps*, Third edition.. ed. Berlin ; New York:Berlin ; New York : Springer, 2001.

E. Loh. "The Ideal HPC Programming Language." *Queue*, vol. 8, no. 6 , pp. 30-38, Jun 1,. 2010. doi:10.1145/1810226.1820518. http://dl.acm.org/citation.cfm?id=1820518.

Magnus H-S Dahle, *A Quick Guide on How to Use the Fortran-to-Python (F2PY) Module*.

Małgorzata Charytanowicz, Jerzy Niewczas, Piotr A. Kowalski, Piotr Kulczycki, Szymon Łukasik, and Sławomir Z. ak, *A Complete Gradient Clustering Algorithm for Features Analysis of X-ray Images*.

R. Mayer, R. Neumayer, D. Baum, and A. Rauber. "Analytic comparison of self-organising maps." *Advances in Self-Organizing Maps*, pp. 182-190. 2009.

I. Miller, *John E. Freund's mathematical statistics with applications*, 7th ed. / Irwin Miller, Marylees Miller.. ed. Upper Saddle River, NJ:Upper Saddle River, NJ : Prentice Hall, 2004.

V. Moosavi, S. Packmann, and I. Vallés, *A Python Library for Self Organizing Map (SOM)*.

A. Ohri, *Python for R Users*. Somerset:John Wiley & Sons, Incorporated, 2017.

K. Pearson. "On Lines and Planes of Closest Fit to Systems of Points in Space." *Philosophical Magazine*, pp. 559–572. 1908.

Pearu Peterson, "F2PY Users Guide and Reference Manual," , accessed 2/1/, 2018, https://docs.scipy.org/doc/numpy-dev/f2py/.

Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V. and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P. and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E. "Scikit-learn: Machine Learning in Py-thon." *Journal of Machine Learning Research*, vol. 12, pp. 2830. 2011.

Peter Wittek, Shi Chao Gao, Ik Soo Lim, and Li Zhao, *Somoclu: An Ecient Parallel Library for Self-Organizing Maps*.

G. Pölzlbauer, *Survey and comparison of quality measures for self-organizing maps*:na, 2004.

R Core Team, *R: A Language and Environment for Statistical Computing*. Vienna, Austria:R Foundation for Statistical Computing, 2017.

M. L. Rizzo, *Statistical computing with R*. Boca Raton [u.a.]:Chapman & Hall/CRC, 2008.

N. J. Salkind, "Encyclopedia of research design," Sage.

Yan Jun, *R-based SOM Package*. https://github.com/cran/som.

H. Yin and N. M. Allinson. "On the distribution and convergence of feature space in self-organizing maps." *Neural Comput.*, vol. 7, no. 6 , pp. 1178-1187. 1995.