Assignment 1: AutoML for Stream k-Nearest Neighbours - Single-pass Self Parameter Tuning *

Emanuel Tomé †and Maria Ferreira ‡

DCC-FCUP, July 2021

1 Introduction

In this work we present our implementation of the algorithm Single-pass Self Parameter Tuning [1], which is an Auto Machine Learning (AutoML) algorithm for data streams. More specifically, we implement and apply the SSPT algorithm together with the k-Nearest Neighbours (kNN) algorithm for classification preceded by Random Projection (RP) in order to reduce the dimensionality of the input space. Therefore, the hyper-parameters to be tuned by the AutoML approach is the dimension of the reduced space (p) and the number of neighbours (k).

This work is organised as follows: we start with a brief review of core concepts of the implemented algorithms, namely the theoretical background on Random Projection, k-Nearest Neighbours algorithm, Nelder-Mead optimisation algorithm and Drift Dectection Method. Then, the SSPT algorithm is described as well as our implementation together with a user guide. Finally, the obtained results for demonstration of our implementation as well as the main conclusions of our work are presented.

2 Theoretical backgroud

2.1 Random Projection

Random Projection (RP) is a technique used to reduce the dimensionality. Comparing to the well-known Principal Component Analysis (PCA), RP has the advantage of lower time complexity (PCA: $O(k^2 \times n + k^2)$ on a matrix size $n \times k$ and RP $O(n \times k \times d)$) and is robust to outliers.

The RP algorithm can be summarised in four steps:

1. Take dataset K, of the dimension $M \times N$ (M - number of samples, N - original dimension/number of features).
2. Initialise a random 2d matriz R of size $N \times D$ where D is the new reduced dimension.
3. Normalise the columns of R making them unit length vectors.
4. Matrix multiplication $J = K \times R$. J is the final ouput with dimension $M \times D$.

The RP algorithm is grounded in the Johnson-Lindenstrauss Lemma that states high-dimensional data can be transformed to a lower dimension data nearly preserving the distance between any two points with litle to no distortion. RP is a better option for data streams than, for instance, PCA since in advance there is not data available to compute the transformation.

*This work was submitted on the framework of the course Data Stream Mining

†Emanuel Tomé is a student at the Faculty of Sciences of the University of Porto. Currently, he is enrolled in the 1^{st} year of the Master's in Data Science (e-mail: up200702634@edu.fc.up.pt).

‡Maria Ferreira is a student at the Faculty of Sciences of the University of Porto. Currently, she is enrolled in the 1^{st} year of the Master's in Data Science (e-mail: up202004113@edu.fc.up.pt).

2.2 k-Nearest Neighbours

The k-Nearest Neighbours (kNN) algorithm is based on learning by analogy, that is, by comparing a given test tuple with the training tuples that are similar to it. The training tuples are described by n attributes, representing a point in an n dimensional space. When a new tuple arrives, the kNN classifier searches the dimensional space for the k training tuples that are closed to the new tuple. Those k training tuples are the k nearest neighbours of the new tuple [2]. The kNN classifier algorithm can be divided in three steps:

Step 1 - Calculate Euclidean Distance

The first step is to calculate the distance between two rows in a dataset. Since rows of data are mostly made up of numbers, an easy way to calculate the distance between two rows or vectors of number is to draw a straight line. We can calculate the straight line distance between two vectors using the Euclidean distance measure. It is calculated as the square root of the sum of the squared differences between the two vectors:

$$d(x, y) = \sqrt{\sum_{i}^{N} (x_i - y_i)^2} \tag{1}$$

where x and y are two different rows and i refers to the column number. The smaller is the Euclidean distance $d(x, y)$, the more similar are the rows.

Step 2 - Get the Nearest Neighbours

To locate the neighbours for a new piece of data within a dataset we must first calculate the distance between each record in the dataset to the new piece of data. Once distances are calculated, we must sort all of the records in the training dataset by their distance to the new data. We can then select the top k to return as the most similar neighbours.

Step 3 - Make predictions

The most similar neighbours collected from training dataset can be used to make predictions. In case of classification, we can return the most represented class among the neighbours. We can achieve thus by determining the maximum from the list of output values from the neighbours.

2.3 Nelder-Mead Optimisation Algorithm

The **Nelder-Mead optimisation** algorithm is a widely used approach for non-differentiable objective functions. As such, it is generally referred to as a pattern search algorithm and is used as a local or global search procedure, challenging nonlinear and potentially noisy and multimodal function optimisation problems. The Nelder-Mead optimisation algorithm is a pattern search optimisation algorithm, which means it does not require or use function gradient information and is appropriate for optimisation problems where the gradient of the function is unknown or cannot be reasonably computed [3]. Although it is often used for multidimensional nonlinear optimisation problems, it can get stuck in a local optima and because of that it may benefit from multiple restarts with different staring points.

The algorithm works by using a shape structure (called simplex) composed of $n+1$ points (vertices), where n is the number of input dimensions of the function. For example, on a two-dimensional problem that may be plotted as a surface, the shape structure would be composed of three points represented as a triangle. The points of the simplex are evaluated and simple rules are used to decide how to move the points based on their relative evaluation. This includes operations such *reflection*, *expansion*, *contraction* and *shrinkage* of the simplex shape on the surface of the objective function. The search stops when the points converge on an optimum, when a minimum difference between evaluations is observed, or when a maximum number of function evaluations are performed [3].

2.4 Drift Detection Method (DDM)

In this work concept drifts are detected with the Drift Detection Method (DDM) [4]. When an example becomes available, the decision model takes a decision and after the decision has been taken, it is compared with the ground truth, that is, the class label of the example. Supposing a sequence of examples in the form $< \overrightarrow{x_i}, y_i >$, the decision model classifies each example in the sequence. In the 0-1 loss function, predictions are either True ($\hat{y}_i = y_i$) or False ($\hat{y}_i \neq y_i$). For a set of examples, the error is a random variable from Bernoulli trials. The Binomial distribution gives the general form of the probability of observing a False. For each point i in the sequence, the error-rate is the probability of observe False, p_i, with standard deviation given by $s_i = \sqrt{p_i(1-p_i)/i}$.

According to the Probability Approximation Correct (PAC) Learning model, it is assumed that if the distribution of the examples is stationary, the error rate of the learning algorithm (p_i) will decrease when the number of examples i increases. Therefore, an increase in the error of the algorithm suggest a change in the class distribution and that the actual decision model is no longer appropriate. To sum up, for example j, the error of the learning algorithm will be:

- **In control** if $p_j + s_j < p_{min} + \beta \times s_{min}$
- **In warming level** if $p_{min} + \alpha \times s_{min} > p_j + s_j > p_{min} + \beta \times s_{min}$
- **In out-of-control** if $p_j + s_j > p_{min} + \alpha \times s_{min}$

β is usually considered equal to 2, which corresponds a confidence level of 95%, and α is usually considered equal to 3, which corresponds a confidence level of 99%.

3 AutoML for stream k-NN

The implemented algorithm in the scope of this work is the Single-pass Self Parameter Tuning (SSPT), recently proposed by Veloso, et al. [1]. The algorithm can be sumarised as follows:

1. Create $n+1$ learning models and train them.
2. Choose the Best, the Good and the Worst models from the $n+1$ previously trained models.
3. Create 7 experimental new models using the Nelder-Mead operators
4. Train the 10 models (the best, the good, the worst and the seven models created by the Nelder-Mead operators)
5. Compute the window size S, $S = \frac{16\sigma^2}{M^2}$, where σ represents the error standard deviation and M the confidence level (in this work, $C = 95\%$).
6. Repeat steps 2-5 until the convergence criteria is met.
7. Deploy the best model.
8. Use the Drift Detection Method (DDM) to react to concept drift. Whenever DDM detects a concept drift, we should go back to step 1. Note that the SSPT is an event-driven algorithm that continuously updates the current learning model.

A more detailed description of this algorithm can be found in the report of our assingnment 2.

4 Implementation and user guide

4.1 Brief description of the implementation

Our implementation of the SSPT algorithm was made in Python, using the package scikit-multiflow [5]. We created three Python classes:

- `RP_kNN_classifier` - In this class the RP-kNN algorithm is implemented. For each RP-kNN model, the user should call this class. This class creates an RP-kNN model, having as parameters the `stream`, `S`, `p`, `k`, `model_knn`, `data_window_X` and `data_window_Y`. Only the `stream` is compulsory. The other hyper-parameters can be obtained automatically internally by our implementation. Indeed, the other hyper-parameters are essentially needed when using the `SSPT` or `SSPT_par` classes.

  The class `RP_kNN_classifier` has the functions `partial_fit`, `predict_and_fit` and `predict`. `partial_fit` allows to add a new sample to the `data_window` used for fitting the model. `predict_and_fit` predicts using the RP-kNN model and re-fits it using the given datapoint. `predict` only predicts using the RP-kNN model, not using the new datapoint to refit the model.

- `SSPT` - In this class is implemented the Single-pass Self Parameter Tuning. This class creates an SSPT object, having as input parameters the `stream`, `n_hyper`, `S`, `models` and `exploration`. `stream` is the data stream, `n_hyper` is the number of hyper-parameters to be tunned by the SSPT algorithm, `models` is the list of models being considered in the SSPT algorithm (can be used to pass some previous tuned models to the SSPT) and `exploration` is a binary varible that indicates if the algorithm starts in the exploration phase or not.

- `SSPT_par` . This class is similar to the previous one. The difference is that it train the models in parallel. For that propose, the multiprocessing package was used.

The source code of the implemented classes can be found in the Appendix A. It should be noted that we used the kNN classifier from the scikit-learn [6] as well the DDM implementation from scikit-multiflow [5] since the core and objective of our work was the implementation of the SSPT algorithm.

4.2 User guide

This user guide is described tacking as principle that the user has the folder SSPT with developed code in the root of their working directory. Therefore, in order to import the developed classes, one should do the following:

```python
from SSPT.SSPT import SSPT           # Sequential version
from SSPT.SSPT_par import SSPT_par   # Parallel version
```

Then, the user should define the SSPT model as for instance:

```python
model_sspt = SSPT(stream=stream_drift,S=100)
```

To run the the SSPT algorithm, one should then call:

```python
while model_sspt_par.models[0].stream.n_remaining_samples()>0:
    model_sspt_par.run()
```

Together with this report, one can find our source code as well a Jupyter Notebook called SSPT_example.ipynb with the example described in the next section. An html version of the notebook is also made available.

5 Demonstration and benchmark comparison

5.1 Used dataset

In order to demonstrate our implementation of the SSPT algorithm for the RP-kNN, we generated a dataset using scikit-multiflow [5]. Namely, we used the two generators: the `ConceptDriftStream` and the `MultilabelGenerator`. The first is a stream generator that adds concept drift or change by joining two streams. This is done by building a weighted combination of two pure distributions that characterizes the target concepts before and after the change [5]. Therefore, the `MultilabelGenerator` is used to generate the streams given to the `ConceptDriftStream`. We then created two streams using the `MultilabelGenerator` both with 8000 samples, 50 features and 1 target variable. The difference between the two are the number of labels (average number of labels per instance, see [5]) and the random_state. We introduced the drift in sample 4000 with a width of 25 datapoint. A snippets of the code used to generate the stream can be seen below:

```python
from skmultiflow.data import MultilabelGenerator, ConceptDriftStream

stream_drift = ConceptDriftStream(stream=MultilabelGenerator(n_samples=8000,
    n_features=50, n_targets=1, n_labels=6, random_state=0), drift_stream=
    MultilabelGenerator(n_samples=8000, n_features=50, n_targets=1, n_labels=2,
    random_state=1), position=4000, width=25,random_state=0)
```

5.2 Considered algorithms for benchmark comparison

The data stream described previously is now used to test our implementation of the SSPT-RP-kNN algorithm. The obtained results using that algorithm are presented as well as the results using the RP-kNN algorithm and the kNN algorithm. These last two are mainly implemented in order to compare their performance with the one obtained using the SSPT-RP-kNN approach. All algorithms are initialised with a window (S) of 100 datapoints. The number of neighbours (k) in kNN and RP-kNN is set to 5 and the dimension p of the lower dimensional space given by the RP was set to 40 in the RP-kNN algorithm. In the SSPT-RP-kNN k is allowed to vary between 3 and 15 and p between 5 and 50.

5.3 Results

In Figure 1 are presented the errors obtained suing the SSPT-RP-KNN approach as well as the detected warnings and change points. Note that the algorithm was able to correctly detect the change point around index 4000. It should be also mentioned that the SSPT-RP-kNN algorithm only converged in index 3164, which justifies the peaks observed in the first 3000 datapoints of the stream. After the change detection, the algorithm converged at index 4561. If we compare the errors obtained using the SSPT-RP-kNN algorithm with the RP-kNN and kNN algorithms, we can noticed that similar performances were obtained. However, it should be said that this may depend on the chosen hyper-parameters, namely the size of the window. Even though, the obtained results using the SSPT-RP-kNN approach has in general slightly better stable errors than the RP-kNN algorithm.

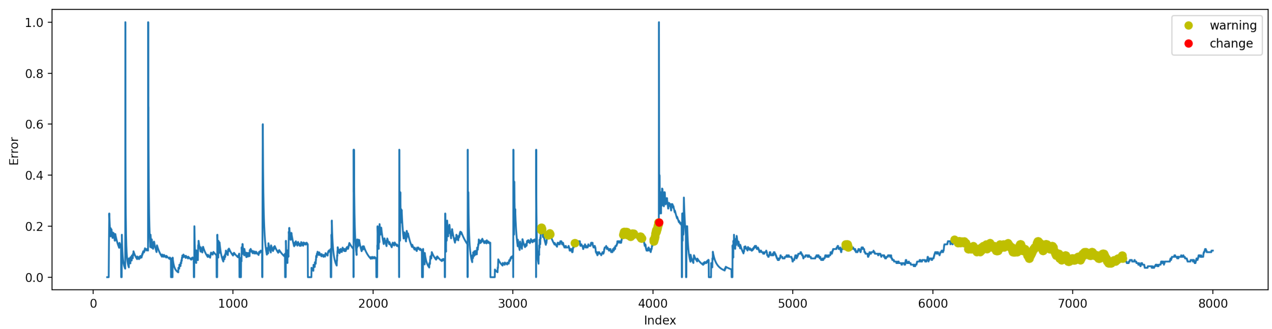

Figure 1: Errors of the SSPT-RP-kNN algorithm (with warnings and change detection).

6 Conclusions

In this work we implemented the SSPT algorithm [1] using Python classes. Namely, three classes were developed, namely a class with the implementation of the RP-kNN and two with the SSPT algorithm, a sequential and a parallel version. Note that different results can be obtained for each run of the SSPT due to the random generation of the hyper-parameters. We also presented a brief theoretical background necessary to understand and follow the implemented algorithm as well as a description of the developed implementation. Finally, we presented a demonstration of the implemented SSPT using a synthetic data stream. The results shown that it was possible to detect the instant of the concept drift.

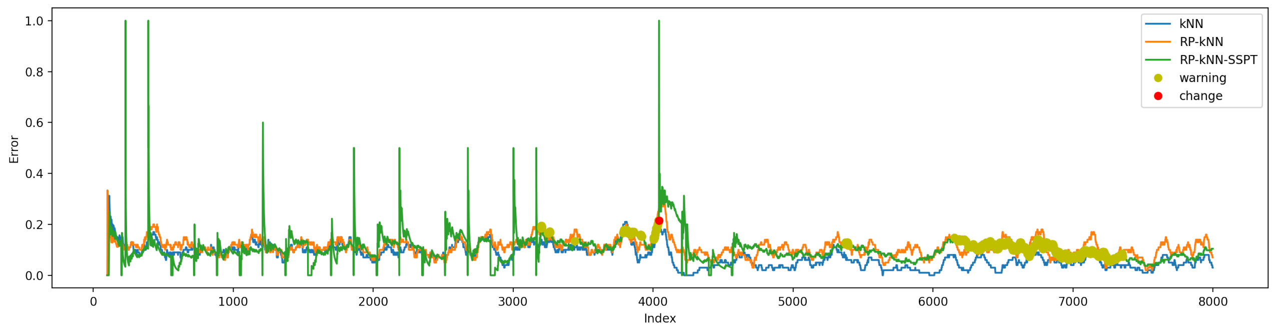

Figure 2: Errors of the SSPT-RP-kNN, RP-kNN and kNN algorithms (with warnings and change detection).

References

[1] Bruno Veloso et al. "Hyperparameter self-tuning for data streams". In: *Information Fusion* (2021). ISSN: 1566-2535. DOI: https://doi.org/10.1016/j.inffus.2021.04.011. URL: https://www.sciencedirect.com/science/article/pii/S1566253521000841.

[2] J. Han, J. Pei, and M. Kamber. *Data Mining: Concepts and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2011. ISBN: 9780123814807. URL: https://books.google.pt/books?id=pQws07tdpjoC.

[3] Jason Brownlee. *How to Use Nelder-Mead Optimization in Python*. Jan. 2021. URL: https://machinelearningmastery.com/how-to-use-nelder-mead-optimization-in-python/.

[4] João Gama et al. "Learning with Drift Detection". In: *Advances in Artificial Intelligence – SBIA 2004*. Ed. by Ana L. C. Bazzan and Sofiane Labidi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 286–295. ISBN: 978-3-540-28645-5.

[5] Jacob Montiel et al. "Scikit-Multiflow: A Multi-output Streaming Framework". In: *Journal of Machine Learning Research* 19.72 (2018), pp. 1–5. URL: http://jmlr.org/papers/v19/18-251.html.

[6] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

Appendixes

A Source Code

A.1 RP kNN classifier

```python
from sklearn import random_projection
from sklearn.neighbors import KNeighborsClassifier as KNN
from sklearn.metrics import accuracy_score
import numpy as np
import random

class RP_kNN_classifier():
    '''
    RP-kNN classifer:
    stream - data stream.
    S - Window size (optional)
    p - Random Projection dimension (optional)
    k - Number of neighbours (optional)
    model_knn - kNN model (optional)
    data_window_X - data window of the predictors (optional)
    data_window_Y - data window of the targert (optional)
    '''
```

```
    def __init__(self, stream, S=30, p=None, k=None, model_knn=None, data_window_X=np.empty([0]), data_window_Y=np.empty([0])): #, max_samples=1000
        self.stream = stream # Data stream
        self.S = S # window size
        self.p = p if p!=None else random.randint(5, min(50,self.stream.n_features)) # Random Projection parameter
        self.k = k if k!=None else random.randint(3,15) # Number of neighbours
        self.model_knn = model_knn if model_knn!=None else KNN(n_neighbors=self.k, n_jobs=1)
        self.data_window_X = data_window_X if (data_window_X!=None).any() else np.empty([0,self.stream.n_features])
        self.data_window_Y = data_window_Y if (data_window_Y!=None).any() else np.empty([0])
        self.l_predict = [] # list with the predicted labels
        self.l_gt = []      # list with the ground truth of the predicted labels
        self.his_error = [] # list with the error history (prequential)

    def __str__(self):
        return 'Stream: ' + str(self.stream) + '\nS: ' + str(self.S) + '\np: ' + str(self.p) + '\nk: ' + str(self.k) + '\nmodel_knn: ' + str(self.model_knn) #+ '\nmax_samples: ' + str(self.max_samples)

    def partial_fit(self, X, y): # Add a new sample to the data_window
        self.data_window_X = np.concatenate((self.data_window_X, X))
        self.data_window_Y = np.concatenate((self.data_window_Y, y))

        # Remove old sample
        self.data_window_X = self.data_window_X[-self.S:,:]
        self.data_window_Y = self.data_window_Y[-self.S:]

    def predict_and_fit(self, max_samples=None):
        max_samples = max_samples if max_samples!=None else self.S
        n_samples = 0

        if self.data_window_X.shape[0]==0: # In case there is not samples in the window
            Xi, yi= self.stream.next_sample(self.S)
            self.partial_fit(Xi, yi)

        elif self.data_window_X.shape[0]<15:
            Xi, yi = self.stream.next_sample(self.S-self.data_window_X.shape[0])
            self.partial_fit(Xi, yi)


        # Random Projection
        transformer = random_projection.GaussianRandomProjection(n_components=self.p)

        # Predict and refit model
        while (self.stream.has_more_samples() and n_samples<max_samples):
            # Transform data
            transformer = transformer.fit(self.data_window_X[-self.S:])
            X_transformed = transformer.transform(self.data_window_X[-self.S:])

            # Fit kNN
            self.model_knn.fit(X_transformed,self.data_window_Y[-self.S:])

            # Take a sample from the stream and transform it:
            Xi, yi= self.stream.next_sample(1)
            Xi_transformed = transformer.transform(Xi)

            # Predict a value/label
            y_pred = self.model_knn.predict(Xi_transformed)

            # Append y_pred to l_predicted
            self.l_predict.append(y_pred)
            self.l_predict = self.l_predict[-self.S:] # list with the predicted
```

```
labels

            # Append y to l_gt
            self.l_gt.append(yi)
            self.l_gt = self.l_gt[-self.S:] # list with the ground truth of the predicted labels

            # Compute error
            self.his_error.append(1-accuracy_score(self.l_gt, self.l_predict))

            # Add sample to fit in the next prediction
            self.partial_fit(Xi, yi)

            # Augment variable n_samples
            n_samples+=1


    def predict(self, max_samples=None):
        max_samples = max_samples if max_samples!=None else self.S

        # Random Projection
        transformer = random_projection.GaussianRandomProjection(n_components=self.p)

        # Predict and refit model
        n_samples = 0
        while (self.stream.has_more_samples() and n_samples<max_samples):
            # Transform data
            transformer = transformer.fit(self.data_window_X[-self.S:])
            X_transformed = transformer.transform(self.data_window_X[-self.S:])

            # Fit kNN
            self.model_knn.fit(X_transformed,self.data_window_Y[-self.S:])

            # Take a sample from the stream and transform it:
            Xi, yi= self.stream.next_sample(1)
            Xi_transformed = transformer.transform(Xi)

            # Predict a value/label
            y_pred = self.model_knn.predict(Xi_transformed)

            # Append y_pred to l_predicted
            self.l_predict.append(y_pred)
            self.l_predict = self.l_predict[-self.S:] # list with the predicted labels

            # Append y to l_gt
            self.l_gt.append(yi)
            self.l_gt = self.l_gt[-self.S:] # list with the ground truth of the predicted labels

            # Compute error
            self.his_error.append(1-accuracy_score(self.l_gt, self.l_predict))

            # Augment variable n_samples
            n_samples+=1
```

A.2 Single-pass Self Parameter Tuning (SSPT)

```
# SSPT
from copy import deepcopy
import math
import statistics
from skmultiflow.drift_detection import DDM

class SSPT():
```

```python
    def __init__(self, stream, n_hyper=2, S=30, models=None, exploration=True):#, S=1000, p=random.randint(5,50), k=random.randint(3,15), max_samples=1000):
        self.stream = stream # Stream
        self.n_hyper = n_hyper # Number of hyper-parameters
        self.S = S             # Window size
        if models == None:
            self.models = {}
            self.models[0] = RP_kNN_classifier(stream=deepcopy(self.stream), S=self.S, p=min(40,self.stream.n_features), k=5)
            for k in range(n_hyper):
                self.models[k+1] = RP_kNN_classifier(stream=deepcopy(self.stream), S=self.S)

        self.exploration = exploration # Binary variable. True if in the exploration phase
        self.hist_performance = []     # History of performance
        self.ddm = DDM()               # Drift detection method
        self.ddm_warming_zone = False  # Binary variable: True if in warming zone, False if not in warming zone.
        self.his_error = []            # Historical of the model errors
        self.his_warming = []          # Indexes of the stream where an warming was detected (used for ploting proposes)
        self.his_change = []           # Indexes of the stream where a change was detected (used for ploting proposes)
        self.warming_X = np.empty([0,self.stream.n_features]) # Data not used for training in the warming zone (X)
        self.warming_Y = np.empty([0])                        # Data not used for training in the warming zone (Y)

    def run(self):
        if self.exploration == True: # Exploration phase
            print('=================================================================================')
            print(f'[INFO] Exploration phase')
            print(f'[INFO] Training the models')

            # Training of the models
            self.vec_error = {}

            for k in range(len(self.models)):
                self.models[k].predict_and_fit(self.S)
                print(f'[INFO] Model {k} trained, S:{self.models[k].S}, p:{self.models[k].p}, k:{self.models[k].k}')
                self.vec_error[k] = self.models[k].his_error[-1]

            print(f'[INFO] All models are trained\n')

            # Order of the learning models
            self.list_ord_learn = [k for k, v in sorted(self.vec_error.items(), key=lambda item: item[1])]# if k<=self.n_hyper]#, reverse=True)]

            # Copy stream from model 0 to this class
            self.stream = deepcopy(self.models[0].stream)

            # Put errors of the best model in the the list his_error
            self.his_error += self.models[self.list_ord_learn[0]].his_error[-self.S:]

            # Computation of S
            self.S = round(max(16*statistics.stdev(self.vec_error)**2/(0.95**2),30))
            print(f'[INFO] S={self.S}')

            # Substitution of models
            ## Temporary copy of the models
            temporary_models_copy = {}
            for k1 in range(self.n_hyper+1):
```

```
                temporary_models_copy[k1] = deepcopy(self.models[self.list_ord_learn[k1]])

            ## Replacement of the three best models
            for k1 in range(self.n_hyper+1):
                self.models[k1] = deepcopy(temporary_models_copy[k1])
                self.models[k1].S = self.S
                self.models[k1].l_predict = []
                self.models[k1].l_gt = []

            # Delete temporary variables
            del temporary_models_copy, k1

            # Compute the experimental models (Nelder Mead Operators)
            self._experimental_models()

            if (self._convergence_criteria()<=1 or self.models[0].his_error[-1]==0):
                self.exploration=False
                print('[INFO] End of exploration phase')
                print('[INFO] Best Model is:')
                print(self.models[0])
                self.print_control = True

                # delete models 1 to 9 (all except best)
                for k in range(1,len(self.models)):
                    self.models.pop(k, None)


        else: # Deployment phase
            if self.print_control:
                print('================================================================================')
                print(f'[INFO] Deployment phase, started at index {self.models[0].stream.sample_idx}')
                self.print_control = False

            if self.ddm_warming_zone==False:
                self.models[0].predict_and_fit(1)
                self.his_error.append(self.models[0].his_error[-1])
                self.ddm.add_element(int(self.models[0].l_predict[-1]!=self.models[0].l_gt[-1]))
                self.ddm_warming_zone = self.ddm.detected_warning_zone()

            elif self.ddm.detected_warning_zone():
                print('================================================================================')
                print(f"[INFO] Warning zone has been detected in data at index {self.models[0].stream.sample_idx}.")
                self.models[0].predict(1)
                self.his_error.append(self.models[0].his_error[-1])
                self.ddm.add_element(int(self.models[0].l_predict[-1]!=self.models[0].l_gt[-1]))
                self.ddm_warming_zone = self.ddm.detected_warning_zone()
                self.his_warming.append(self.models[0].stream.sample_idx)
                last_X, last_y = self.models[0].stream.last_sample()
                self.warming_X = np.concatenate((self.warming_X, last_X))
                self.warming_Y = np.concatenate((self.warming_Y, last_y[0]))

            if self.ddm.detected_change():
                print('================================================================================')
                print(f"[INFO] Change has been detected in data at index {self.models[0].stream.sample_idx}.")
```

```
                self.ddm_warming_zone=False
                self.exploration=True
                self.ddm.reset()
                self.his_change.append(self.models[0].stream.sample_idx)
                self.stream = deepcopy(self.models[0].stream)

                # Create new models:
                for k in range(self.n_hyper):
                    self.models[k+1] = RP_kNN_classifier(stream=deepcopy(self.stream), S=self.S,
                                                         data_window_X=deepcopy(self.warming_X), #models[0].data_window_X),
                                                         data_window_Y=deepcopy(self.warming_Y)) #models[0].data_window_Y))
                    self.models[k+1].his_error = deepcopy(self.models[0].his_error)

                self.models[0].l_predict = []
                self.models[0].l_gt = []
                self.warming_X = np.empty([0,self.stream.n_features])
                self.warming_Y = np.empty([0])


    def _experimental_models(self):
        '''Defines the new 7 experimental models M(3), R(4), E(5), C1(6), C2(7), S1(8) and S2(9)'''
        # Midpoint Model (M) - 3
        self.models[3] = RP_kNN_classifier(stream=deepcopy(self.stream), S=self.S,
                                           p=max(min(round((self.models[self.list_ord_learn[0]].p + self.models[self.list_ord_learn[1]].p)/2), min(50,self.stream.n_features)), 5),
                                           k=max(min(round((self.models[self.list_ord_learn[0]].p + self.models[self.list_ord_learn[1]].p)/2), 15), 3),
                                          data_window_X=deepcopy(self.models[0].data_window_X),data_window_Y=deepcopy(self.models[0].data_window_Y))
        # Reflection Model (R) - 4
        self.models[4] = RP_kNN_classifier(stream=deepcopy(self.stream), S=self.S,
                                           p=max(min(2*self.models[3].p - self.models[self.list_ord_learn[2]].p, min(50,self.stream.n_features)), 5),
                                           k=max(min(2*self.models[3].k - self.models[self.list_ord_learn[2]].k, 15), 3),
                                          data_window_X=deepcopy(self.models[0].data_window_X),data_window_Y=deepcopy(self.models[0].data_window_Y))
        # Expantion Model (E) - 5
        self.models[5] = RP_kNN_classifier(stream=deepcopy(self.stream), S=self.S,
                                           p=max(min(2*self.models[4].p - self.models[3].p, min(50,self.stream.n_features)), 5),
                                           k=max(min(2*self.models[4].k - self.models[3].k, 15), 3),
                                          data_window_X=deepcopy(self.models[0].data_window_X),data_window_Y=deepcopy(self.models[0].data_window_Y))
        # Contraction Model 1 (C1) - 6
        self.models[6] = RP_kNN_classifier(stream=deepcopy(self.stream), S=self.S,
                                           p=max(min(round((self.models[4].p + self.models[3].p)/2), min(50,self.stream.n_features)), 5),
                                           k=max(min(round((self.models[4].k + self.models[3].k)/2), 15), 3),
                                          data_window_X=deepcopy(self.models[0].data_window_X),data_window_Y=deepcopy(self.models[0].data_window_Y))
        # Contraction Model 2 (C2) - 7
        self.models[7] = RP_kNN_classifier(stream=deepcopy(self.stream), S=self.S,
                                           p=max(min(round((self.models[self.list_ord_learn[2]].p + self.models[3].p)/2), min(50,self.stream.n_features)), 5),
                                           k=max(min(round((self.models[self.list_ord_learn[2]].k + self.models[3].k)/2), 15), 3),
                                          data_window_X=deepcopy(self.models[0].data_window_X),data_window_Y=deepcopy(self.models[0].data_window_Y))
```

```python
        # Shrinkage Model 1 (S1) - 8
        self.models[8] = RP_kNN_classifier(stream=deepcopy(self.stream), S=self.S,
                                           p=max(min(round((self.models[self.list_ord_learn[0]].p + self.models[4].p)/2), min(50,self.stream.n_features)), 5),
                                           k=max(min(round((self.models[self.list_ord_learn[0]].k + self.models[4].k)/2), 15), 3),
                                          data_window_X=deepcopy(self.models[0].data_window_X),data_window_Y=deepcopy(self.models[0].data_window_Y))
        # Shrinkage Model 2 (S2) - 9
        self.models[9] = RP_kNN_classifier(stream=deepcopy(self.stream), S=self.S,
                                           p=max(min(round((self.models[self.list_ord_learn[0]].p + self.models[self.list_ord_learn[2]].p)/2), min(50,self.stream.n_features)), 5),
                                           k=max(min(round((self.models[self.list_ord_learn[0]].k + self.models[self.list_ord_learn[2]].k)/2), 15), 3),
                                          data_window_X=deepcopy(self.models[0].data_window_X),data_window_Y=deepcopy(self.models[0].data_window_Y))

        for k1 in range(3,10):
            self.models[k1].his_error = deepcopy(self.models[0].his_error)


    def _convergence_criteria(self):
        d = 0
        for k1 in range(self.n_hyper+1):
            for k2 in range(self.n_hyper+1):
                d = max(d, math.sqrt((self.models[k1].p-self.models[k2].p)**2 + (self.models[k1].k-self.models[k2].k)**2))
        # r - radius
        r = d * math.sqrt(self.n_hyper/(2*(self.n_hyper+1)))
        return r
```

A.3 Single-pass Self Parameter Tuning - Parallel (SSPT_par)

```python
# SSPT parallel
from copy import deepcopy
import math
import statistics
from skmultiflow.drift_detection import DDM
import multiprocessing as mp

class SSPT_par():
    def __init__(self, stream, n_hyper=2, S=30, models=None, exploration=True):#, S=1000, p=random.randint(5,50), k=random.randint(3,15), max_samples=1000):
        self.stream = stream
        self.n_hyper = n_hyper
        self.S = S
        if models == None:
            self.models = {}
            self.models[0] = RP_kNN_classifier(stream=deepcopy(self.stream), S=self.S, p=40, k=5)
            for k in range(n_hyper):
                self.models[k+1] = RP_kNN_classifier(stream=deepcopy(self.stream), S=self.S)

        self.exploration = exploration # Binary variable. True if in the exploration phase
        self.hist_performance = []     # History of performance
        self.ddm = DDM()               # Drift detection method
        self.ddm_warming_zone = False  # Binary variable: True if in warming zone, False if not in warming zone.
        self.his_error = []            # Historical of the model errors
        self.his_warming = []          # Indexes of the stream where an warming was detected (used for ploting proposes)
```

```python
        self.his_change = []            # Indexes of the stream where a change was detected (used for ploting proposes)
        self.warming_X = np.empty([0,self.stream.n_features]) # Data not used for training in the warming zone (X)
        self.warming_Y = np.empty([0])                        # Data not used for training in the warming zone (Y)
        self.Q = mp.Queue()

    def _parallel(self,k):
        self.models[k].predict_and_fit()
        print(f'[INFO] Model {k} trained')
        self.Q.put(self.models[k])

    def run(self):
        if self.exploration == True: # Exploration phase
            print('================================================================================')
            print(f'[INFO] Exploration phase')
            print(f'[INFO] Training the models')

            # Training of the models
            self.vec_error = {}
            processes = []

            for i2 in range(len(self.models)):
                p = mp.Process(target=self._parallel, args=([i2]))#i,i+1))
                processes.append(p)
                p.start()

            for k,process in enumerate(processes):
                self.models[k] = self.Q.get()
#                process.join()


            for k in range(len(self.models)):
                self.vec_error[k] = self.models[k].his_error[-1]

            print(f'[INFO] All models are trained\n')

            # Order of the learning models
            self.list_ord_learn = [k for k, v in sorted(self.vec_error.items(), key=lambda item: item[1])]# if k<=self.n_hyper]#, reverse=True)]

            # Copy stream from model 0 to this class
            self.stream = deepcopy(self.models[0].stream)

            # Put errors of the best model in the the list his_error
            self.his_error += self.models[self.list_ord_learn[0]].his_error[-self.S:]

            # Computation of S
            self.S = round(max(16*statistics.stdev(self.vec_error)**2/(0.95**2),30))
            print(f'[INFO] S={self.S}')

            # Substitution of models
            ## Temporary copy of the models
            temporary_models_copy = {}
            for k1 in range(self.n_hyper+1):
                temporary_models_copy[k1] = deepcopy(self.models[self.list_ord_learn[k1]])

            ## Replacement of the three best models
            for k1 in range(self.n_hyper+1):
                self.models[k1] = deepcopy(temporary_models_copy[k1])
                self.models[k1].S = self.S
                self.models[k1].l_predict = []
```

```python
                self.models[k1].l_gt = []

            # Delete temporary variables
            del temporary_models_copy, k1

            # Compute the experimental models (Nelder Mead Operators)
            self._experimental_models()

            if (self._convergence_criteria()<=1 or self.models[0].his_error[-1]==0):
                self.exploration=False
                print('[INFO] End of exploration phase')
                print('[INFO] Best Model is:')
                print(self.models[0])
                self.print_control = True

                # delete models 1 to 9 (all except best)
                for k in range(1,len(self.models)):
                    self.models.pop(k, None)


        else: # Deployment phase
            if self.print_control:
                print('=================================================================================')
                print(f'[INFO] Deployment phase')
                self.print_control = False

            if self.ddm_warming_zone==False:
                self.models[0].predict_and_fit(1)
                self.his_error.append(self.models[0].his_error[-1])
                self.ddm.add_element(int(self.models[0].l_predict[-1]!=self.models[0].l_gt[-1]))
                self.ddm_warming_zone = self.ddm.detected_warning_zone()

            elif self.ddm.detected_warning_zone():
                print('=================================================================================')
                print(f"[INFO] Warning zone has been detected in data at index {self.models[0].stream.sample_idx}.")
                self.models[0].predict(1)
                self.his_error.append(self.models[0].his_error[-1])
                self.ddm.add_element(int(self.models[0].l_predict[-1]!=self.models[0].l_gt[-1]))
                self.ddm_warming_zone = self.ddm.detected_warning_zone()
                self.his_warming.append(self.models[0].stream.sample_idx)
                last_X, last_y = self.models[0].stream.last_sample()
                self.warming_X = np.concatenate((self.warming_X, last_X))
                self.warming_Y = np.concatenate((self.warming_Y, last_y[0]))

            if self.ddm.detected_change():
                print('=================================================================================')
                print(f"[INFO] Change has been detected in data at index {self.models[0].stream.sample_idx}.")
                self.ddm_warming_zone=False
                self.exploration=True
                self.ddm.reset()
                self.his_change.append(self.models[0].stream.sample_idx)
                self.stream = deepcopy(self.models[0].stream)

                # Create new models:
                for k in range(self.n_hyper):
                    self.models[k+1] = RP_kNN_classifier(stream=deepcopy(self.stream)
```

```
, S=self.S,
                                                            data_window_X=deepcopy(self.models[0].data_window_X),
                                                            data_window_Y=deepcopy(self.models[0].data_window_Y))
                    self.models[k+1].his_error = deepcopy(self.models[0].his_error)

                self.models[0].l_predict = []
                self.models[0].l_gt = []
                self.warming_X = np.empty([0,self.stream.n_features])
                self.warming_Y = np.empty([0])


    def _experimental_models(self):
        '''Defines the new 7 experimental models M(3), R(4), E(5), C1(6), C2(7), S1(8) and S2(9)'''
        # Midpoint Model (M) - 3
        self.models[3] = RP_kNN_classifier(stream=deepcopy(self.stream), S=self.S,
                                            p=max(min(round((self.models[self.list_ord_learn[0]].p + self.models[self.list_ord_learn[1]].p)/2), 50), 5),
                                            k=max(min(round((self.models[self.list_ord_learn[0]].p + self.models[self.list_ord_learn[1]].p)/2), 15), 3),
                                            data_window_X=deepcopy(self.models[0].data_window_X),
                                            data_window_Y=deepcopy(self.models[0].data_window_Y))
        # Reflection Model (R) - 4
        self.models[4] = RP_kNN_classifier(stream=deepcopy(self.stream), S=self.S,
                                            p=max(min(2*self.models[3].p - self.models[self.list_ord_learn[2]].p, 50), 5),
                                            k=max(min(2*self.models[3].k - self.models[self.list_ord_learn[2]].k, 15), 3),
                                           data_window_X=deepcopy(self.models[0].data_window_X),data_window_Y=deepcopy(self.models[0].data_window_Y))
        # Expantion Model (E) - 5
        self.models[5] = RP_kNN_classifier(stream=deepcopy(self.stream), S=self.S,
                                            p=max(min(2*self.models[4].p - self.models[3].p, 50), 5),
                                            k=max(min(2*self.models[4].k - self.models[3].k, 15), 3),
                                           data_window_X=deepcopy(self.models[0].data_window_X),data_window_Y=deepcopy(self.models[0].data_window_Y))
        # Contraction Model 1 (C1) - 6
        self.models[6] = RP_kNN_classifier(stream=deepcopy(self.stream), S=self.S,
                                            p=max(min(round((self.models[4].p + self.models[3].p)/2), 50), 5),
                                            k=max(min(round((self.models[4].k + self.models[3].k)/2), 15), 3),
                                           data_window_X=deepcopy(self.models[0].data_window_X),data_window_Y=deepcopy(self.models[0].data_window_Y))
        # Contraction Model 2 (C2) - 7
        self.models[7] = RP_kNN_classifier(stream=deepcopy(self.stream), S=self.S,
                                            p=max(min(round((self.models[self.list_ord_learn[2]].p + self.models[3].p)/2), 50), 5),
                                            k=max(min(round((self.models[self.list_ord_learn[2]].k + self.models[3].k)/2), 15), 3),
                                           data_window_X=deepcopy(self.models[0].data_window_X),data_window_Y=deepcopy(self.models[0].data_window_Y))
        # Shrinkage Model 1 (S1) - 8
        self.models[8] = RP_kNN_classifier(stream=deepcopy(self.stream), S=self.S,
                                            p=max(min(round((self.models[self.list_ord_learn[0]].p + self.models[4].p)/2), 50), 5),
                                            k=max(min(round((self.models[self.list_ord_learn[0]].k + self.models[4].k)/2), 15), 3),
                                           data_window_X=deepcopy(self.models[0].data_window_X),data_window_Y=deepcopy(self.models[0].data_window_Y))
```

```
        # Shrinkage Model 2 (S2) - 9
        self.models[9] = RP_kNN_classifier(stream=deepcopy(self.stream), S=self.S,
                                            p=max(min(round((self.models[self.
    list_ord_learn[0]].p + self.models[self.list_ord_learn[2]].p)/2), 50), 5),
                                            k=max(min(round((self.models[self.
    list_ord_learn[0]].k + self.models[self.list_ord_learn[2]].k)/2), 15), 3),
                                           data_window_X=deepcopy(self.models[0].
    data_window_X),data_window_Y=deepcopy(self.models[0].data_window_Y))

        for k1 in range(3,10):
            self.models[k1].his_error = deepcopy(self.models[0].his_error)


    def _convergence_criteria(self):
        d = 0
        for k1 in range(self.n_hyper+1):
            for k2 in range(self.n_hyper+1):
                d = max(d, math.sqrt((self.models[k1].p-self.models[k2].p)**2 + (self
    .models[k1].k-self.models[k2].k)**2))
        # r - radius
        r = d * math.sqrt(self.n_hyper/(2*(self.n_hyper+1)))
        return r
```