

University of Bahrain

College of Science

Postgraduate Programs



Predicting Electricity Consumption Using Load Profiling Clustering Method

Thesis submitted in partial fulfilment of the requirements for the
degree of M.Sc. In Big Data Science and Analytics

Prepared by:

Esra Jasim Aljowder

ID 20021155

Supervised by:

Dr. Sawsan Hilal

Assistant Professor

University of Bahrain

Dr. Riadh Ksantini

Associate Professor

University of Bahrain

Kingdom of Bahrain

March 2023

Defense Committee's Approval

University of Bahrain
Deanship of Graduate Studies
And scientific Research



جامعة البحرين
عمادة الدراسات العليا والبحث العلمي

Thesis Discussion Committee Decision

Student Name: Esra Jasim Aljowder	Student ID: 20021155
Department: Choose an item.	College: Science
Program: M.Sc. in Big Data Science and Analytics	
Thesis Title: Predicting Electricity Consumption Using Load Profiling Clustering Method	
Supervisor Name: Dr. Sawsan Hilal + Dr. Riadh Ksantini	Student GPA: 3.33

The Discussion Committee met on 22-May-23 In S41-1002. Upon discussing the student whose details are stated above in public, the discussion committee recommends the following:

The student is awarded a grade Choose an item. in the thesis subject.

- ☐ Without making any changes to the thesis.
- ☒ After making the modifications attached with this decision.

Discussion Committee Members Approval:

Name

1. Dr. Sawsan Hilal + Dr. Riadh Ksantini
2. Prof. Nabil Hewahi
3. Prof. Ahmed Fawzi Otoom

Signature

(Advisor)

(Internal Examiner)

(External Examiner)

6 | Approved at the University Council 6th meeting, 2013
In accordance with Decision No. 736/2013 dated 14/3/2013
Modified by University Council Decision No. 2043/2013 dated 20/11/2013.

Abstract

Energy consumption is considered one of the essential topics of energy systems which gained high attention after the energy crisis in 1970's. It has been shown that energy consumption throughout the world is rapidly increasing. The field of smart meter data analytics is considered as a relatively recent field that grew recently due to the importance of the data generated from smart meters and with the progress in high-performance computing which made it possible to efficiently and effectively use deep neural networks to extract more beneficial knowledge and patterns from collected massive data within the smart grid. Hence it became crucial to analyse the energy consumption trends.

Feature Drift takes place when a neural network is trained to minimize a combination of two strongly conflicting loss functions which are the pseudo-supervised loss and the self-supervised one. Dynamic autoencoder model is used for the clustering so that the load profiling can handle time series data as well as feature randomness and feature drift.

This research aims to deploy a deep clustering algorithm embedded with an autoencoder and compare it with the dynamic autoencoder which can improve the prediction quality while applying dimensionality reduction and evaluate these models based on the performance using the silhouette score. The dynamic autoencoder outperformed the deep embedded clustering (DEC) and improved deep embedded clustering (IDEC) models and achieved a silhouette score of 0.9931 compared to DEC of 0.6128 and IDEC of 0.8829. This shows that positioning a dynamic autoencoder embedded with deep clustering model has worked effectively against the problem of feature drift and feature randomness.

Acknowledgments

I wish to extend my gratitude to University of Bahrain, BNET and Liverpool John Moores University for giving me the opportunity to join the Big Data Science and Analytics Master's Program.

I would like to extend my special acknowledgments to my supervisors:

Dr. Sawsan Hilal for her supervision, guidance and countless efforts throughout the programme and the thesis

Dr. Riadh Ksantini for his guidance and considerable support during the preparation of the thesis.

Dr. Nabil Hewahi, Dr. Hesham Al-Ammal, Dr. Abdulla AlQaddoumi and Dr. Ahmed Zeki for their efforts during the master's program

Table of Contents

Defense Committee’s Approval	A
Abstract.....	B
Acknowledgments	C
Table of Contents.....	D
List of Tables	F
List of Figures.....	G
List of Abbreviations	H
Chapter One Introduction to the Study	1
1.0. Introduction.....	2
1.1. Research Objectives and Questions.....	4
1.2. Research Problem	4
1.3. Research Significance.....	4
1.4. Thesis Structure	5
Chapter Two Literature Review.....	7
2. Approaches to Smart Meter Readings.....	8
2.1. Clustering Algorithms	10
2.1.1. Deep Clusterin	16
2.1.2. Deep Embedded Clustering.....	17
2.1.3. Deep Clustering with Dynamic Autoencoder	18
2.1.4. Deep Temporal Clustering.....	20
2.1.5. Survey of Clustering Algorithms.....	21
2.2. Summary of Related Studies	23
Chapter Three Methods and Procedures	25

3.0. Methodology	26
3.1. Temporal Autoencoder	27
3.2. Dynamic Autoencoder	28
3.3. Deep Embedded Clustering (DEC)	29
3.4. Improved Deep Embedded Clustering (IDEC)	29
Chapter Four Results and Discussion	31
4.0. Study Dataset	32
4.1. Experiment Protocol	34
4.2. Results.....	39
4.3. Discussion.....	55
Chapter Five Conclusion, Limitations and Future work	57
5.0. Conclusion	58
5.1. Limitations and Future work	59
References.....	60
Appendices.....	62
الملخص	143

List of Tables

Table	Page
Table (1) Summary of Related Work	23
Table (2) Irish dataset description	32
Table (3) Part of the dataset after the pre-processing	33
Table (4) Temporal Autoencoder parameters	35
Table (5) DynAE Parameters	36
Table (6) DEC Parameters	37
Table (7) IDEC Parameters	37
Table (8) MSE results for data dimension 700	39
Table (9) MSE results for data dimension 1000	40
Table (10) MSE results for data dimension 1152	40
Table (11) MSE results for SGD optimizer	40
Table (12) MSE results per dimension and per epoch	41
Table (13) MSE results for different data dimension	42
Table (14) SI Score for clusters 2 to10 Before training the model for the full dataset and reduced dataset	44
Table (15) SI score when using the full dataset	45
Table (16) SI score when using the reduced dataset	45
Table (17) SI score for 4 clusters	46
Table (18) SI and the MSE for the number of epochs and iteration tried.	46
Table (19) DEC highest and lowest SI score	47
Table (20) DEC highest and lowest MSE	47
Table (21) SI score for IDEC	47

List of Figures

Figure	Page
Fig. (1) Sample Load Profile for Electrical Power Consumption	9
Fig. (2) Experiment Protocol Flow Chart	26
Fig. (3) DTC Overview	27
Fig. (4) Clustering Phase of DynAE	28
Fig. (5) General Clustering Architecture of DynAE	29
Fig. (6) Irish Dataset Load Profile Graph	34
Fig. (7) MSE Pattern for Each Data Dimension	42
Fig. (8) MSE for Each Data Dimension	42
Fig. (9) Clusters Visualization	44
Fig. (10) Screen Plot of Principle Component	48
Fig. (11) Shape of Data Points for 3 Clusters (K=3)	48
Fig. (12) Shape of Data Points for 2 Clusters (K=2)	49
Fig. (13) Normalized and Non-Normalized Dataset	49
Fig. (14) Elbow Curve	50
Fig. (15) IDEC Dendogram	50
Fig. (16) Clusters Correlation	51
Fig. (17) Cluster 1 Trend Plot	51
Fig. (18) Cluster 2 Trend Plot	52
Fig. (19) Cluster 3 Trend Plot	52
Fig. (20) Cluster 4 Trend Plot	52
Fig. (21) Monthly Electricity Consumption	53
Fig. (22) Clusters Average Daily Consumption	53
Fig. (23) DynAE SI Score Comparison	56

List of Abbreviations

Abbreviation	Explanation
DTC	Deep Temporal Clustering
TAE	Temporal Autoencoder
DynAE	Dynamic Autoencoder
DEC	Deep Embedded Clustering
IDEK	Improved Deep Embedded Clustering
MSE	Mean Square Error
SI	Silhouette Index
TLP	Typical Load Profile
EMS	Energy Management Systems
DSM	Demand Side Management
DNN	Deep neural networks
PCA	Principal Component Analysis
ML	Machine Learning
BI-LSTM	Bi Directional Long Short-Term Memory
AUC	Area Under the Curve
ACC	Accuracy
NMI	Normalized Mutual Information
GMM	Gaussian Mixture Model
CKSN	Ciphering Key Sequence Number

Chapter One

Introduction to the Study

1.0. Introduction

1.1. Research Objectives and Questions

1.2. Research Problem

1.3. Research Significance

1.4. Thesis Structure

Chapter One

Introduction to the Study

1.0. Introduction

The revolution in power systems led to the introduction of smart grid, the communication technology is joined with conventional electricity meters, known as smart meters which measure electricity consumption, resulting in a very huge amount of data. These smart meters quantify electricity consumption along with other measurements at every small-time interval and communicate to energy suppliers and authorities. This large amount of data is being collected daily by the smart meters and big data analytics can aid in achieving insights for smart energy management. All the conventional energy meters will be replaced by smart meters soon which will result in power utilities receiving an overflow of data after the deployment of smart meters that will enable further analysis and more accurate prediction of energy consumption. (Mujeeb, et al, 2018).

Lately with the emphasis on energy consumption and sustainability, power utilities are dealing with the challenge of maintaining the security and reliability of the power supply while integrating new types of generation technologies and loads. (Haque, et al, 2012). It was estimated that, more than 800 million smart meters will be deployed world -wide until 2020. Smart grid systems that rely on utilizing smart meters enable the design of forecasting models that are more accurate on the distribution grid. Around 220 million smart meter measurements are recorded daily, in a large sized smart grid. In power system, an accurate and robust load forecasting is crucial for a reliable and secure operation as well as the optimization of the electric energy industry in a competitive electricity market environment. (Mujeeb, et al, 2018)

Load profiling is defined as clustering unlabelled high dimensional data, and its performance starts with data preparation where the dataset is preprocessed, and dimension reduction is performed to deal with the problem of high dimensionality and compress the size of the data. Load profiling aids in grouping the consumers based on similarities in their consumption behaviour, and as a result, produces the typical load profile (TLP) for each group. The load profile is a graph or a curve that represents the difference of electrical usage over time. A load profiling problem can be expressed as an unsupervised machine learning task, and hence, it is completely data-driven. Load profiling is a critical task in the knowledge discovery field of smart meter analytics and considered a critical input to other smart energy domains such as load dis-aggregation, load analysis, demand response, etc. Moreover, implementing load profiling plays a major role in the forecasting accuracy as it is a crucial step that precedes load forecasting. (Eskandarnia, et al, 2021)

In the recent years big data analytics are gaining popularity in the energy management systems (EMS) because it can help in achieving insights for smart energy management. (Mujeeb, et al, 2018)

For example, real world smart meter data normally lacks labels, so load profiling in the machine learning field can be considered as an unsupervised learning task that can mainly be attempted by clustering. Specifically clustering improves the prediction accuracy and the load forecasting by implementing autoencoder-based clustering that automatically converts smart meter data into more clustering-friendly representations that retain the original data characteristics. Deep embedded clustering performs feature learning and clustering assignment simultaneously, it not only outperforms standard clustering algorithms, but it can also be used with raw datasets, eliminating the requirement for hybrid clustering or dimension reduction. (Eskandarnia, et al, 2021)

1.1. Research Objectives and Questions

The main objective of this research is to improve the prediction for energy consumption by using clustering model which can tease out temporal information, while handling feature drift and feature drift problems.

The following questions can be extracted from the main objective of the research:

- How can load profiling be improved in predicting energy consumption by using time series data?
- How can clustering models tease out temporal information by handling feature drift problems?
- Which clustering approach can result in the best prediction?

1.2. Reasearch problem

There is a global trend to reduce the energy consumption and electricity is one of these sources, hence it became important to understand and extract electricity consumption patterns from smart meter data to identify potential saving opportunities by the electricity suppliers and authorities.

This reaserch will explor the expected electricity consumption for groups of consumers with the same consumption patterns using sequential data.

1.3. Research Significance

The main contribution of this research is proposing a novel and improved load profiling framework that implements an autoencoder-based clustering method where the result improves the quality of the obtained load profiles by representing data as a nested hierarchy of concepts within layers of the autoencoder, allowing data dimension reduction and highly

nonlinear decision segregation between clusters at the autoencoder bottleneck level. It automatically converts smart meter data into a more clustering-friendly representation which maintains all the original data's characteristics using clustering algorithm based on an autoencoder properties to automate the feature selection step and perform non-linear feature transformations.

Clustering techniques and their application on time series data remains an open problem which has left a gap for accurate unsupervised learning of sequential data. Real world time series data applications often have temporal gaps as well as high frequency noise due to the data acquisition method and/or the inherent nature of the data and exhibit substantial variations in important features and properties, temporal scales, and dimensionality. Deep Temporal Clustering (DTC) that will be applied in this research naturally integrates dimensionality reduction and temporal clustering into a single end-to-end learning framework which is fully unsupervised.

This research will help electricity authorities and suppliers on how to identify customer groups based on electricity usage patterns and hence will help in the prediction of high or low electricity consumption during the day which will aid in identifying electricity saving opportunities and the circumstances associated with high consumptions. In addition, this research will help set the basis for prediction using clustering algorithms embedded with autoencoders on time series data which can be applied on other time series data by other researchers.

1.4. Thesis Structure

The remainder of this report is structured as follows. Chapter Two summarizes the related work with an overview of the literature. Chapter Three details the methods and procedures applied, while Chapter Four

collects the results and provides a discussion and Chapter Five concludes the study and highlights directions for improvement.

Chapter Two

Background and Literature Review

2. Approaches to Smart Meter Readings

2.1. Clustering Algorithms

2.1.1. Deep Clustering

2.1.2. Deep Embedded Clustering

2.1.3. Deep Clustering with Dynamic Autoencoder

2.1.4. Deep Temporal Clustering

2.1.5. Survey of Clustering Algorithms

2.2. Summary of Related Studies

Chapter Two

Background and Literature Review

2.Approaches to Smart Meter Readings

With the electricity market liberalization and consequently the customers have the freedom to choose their electricity provider, the distribution and retail companies in addition to the authorities are looking for better market strategies based on adequate information based the consumption patterns of its electricity customers. A fair insight on the customers' behaviour will allow the definition of specific contract aspects based on the different consumption patterns. The consumption patterns are impacted by factors that can be external such as weather condition, the day of the week or the type of consumer (a domestic consumer or industrial consumer). The influence of these factors must be considered in any study related with the consumers' electrical consumption behaviour and to study the electricity consumption patterns the development of load research projects is essential (Rodrigues, et al, 2003).

Smart meters are extremely popular, which makes it possible to gather a huge quantity of precise data on electricity consumption. Globally, the deregulation of the power sector has been continuously progressing, especially on the delivery side. A critical problem is how to use vast amounts of smart meter data to support and improve the effectiveness and sustainability of the power grid. Significant research has been done so far on smart meter data processing (Chen, et al, 2019).

A load profile can be defined as a pattern of electricity demand for a consumer, or group of consumers, over a given period and used to generate realistic consumption data. A load profile will vary according to the customer type in terms of industrial, residential or commercial.

Power suppliers and authorities utilize the information generated from the load profile to plan and predict the electricity level they must make available at any given time. Each load profile should represent a relatively similar group of consumers and must be clearly different from the others. The number of profiles should be big enough to represent the different consumption patterns (Rodrigues, et al, 2003). An accurate classification of consumer classes or categories with the association of a load profile are essential to support marketing strategies. Understanding customer load curve shapes can help with demand side management (DSM) program development, load forecasting, marketing campaign effectiveness, and different tariff setting techniques. It can also help with understanding customer habits. Fig. (1) represents a sample of an hourly data for building load profile and electrical energy consumption, RT is the time of day rate and kW is the kilowatt (Viegas, et al, 2018).

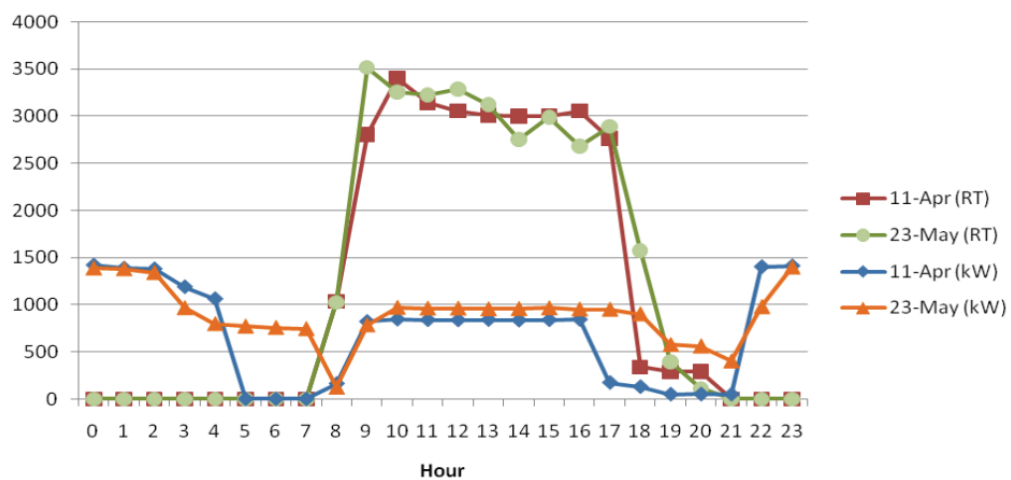


Fig. (1) Sample load profile for electrical power consumption (Khushairi, et al, 2011)

Prevalent Engineering Practice (PEP) was the early approach used for load profiling which was based on the yearly maximum demand. Due to the unrealistic resulted load curves, top-down and bottom-up approaches are used instead. Both approaches generate realistic load curves, but they

have some design limitations, so other options for load profiling were considered which include mathematical approaches, such as Gaussian distribution function. However, it is important to avoid over-fitting the Gaussian function to keep the model general (Tazi, et al, 2018).

To find the parameter formation that will achieve the best performance of a given algorithm, one can apply an optimization procedure but there are two major problems associated with this approach. First, adjusting parameters to a given dataset may lead to overfitting which means that specific values found to provide good performance may lead to worse results if new data is considered. Second, in some of the cases, parameter optimization can be unfeasible, given the time complexity of many algorithms, combined with their typically large number of parameters. Ultimately, many researchers apply the classifier or clustering algorithms using the default parameters provided by the software. Consequently, efforts are required for evaluating and comparing the performance of different clustering algorithms in the optimization and default situations (Rodriguez, et al, 2016).

2.1. Clustering Algorithms

Driver and Kroeber first introduced cluster analysis in the area of anthropology in 1932. Zubin and Tryon then introduced cluster analysis to psychology in 1938 and 1939, respectively. For the renowned trait theory classification in personality psychology, Cattell used it in early 1943 (Zhou, Wu, Luo & Abdel-Baset, 2018).

Clustering can be defined as division of the population or set of data points into several groups so that the data points within each group are more similar to one another and different from the data points within the other groups. It is essentially a grouping of objects based on how similar and unlike they are to one another. It is a kind of unsupervised learning

technique. The process of drawing references from datasets of input data without labelled is known as unsupervised learning. It is typically used as a method to identify the groups, generative qualities, and significant structures that are inherent in a set of instances. Clustering is a powerful data mining and analysis tool used in many fields, including machine learning, bioinformatics, robotics, pattern recognition, and image analysis (Ezugwu, et al, 2022). The most fundamental problem in cluster analysis is identifying the number of clusters apriori, specifying the correct number of clusters apriori can help obtain optimal solutions to many clustering problems (Rodriguez, et al, 2016). As a result, automatic clustering algorithms are replacing traditional clustering algorithms. Automated clustering algorithms are designed to perform clustering without any prior knowledge of the data, they also can determine the optimal number of clusters in noisy datasets as well.

In order to cluster data, there are five distinct methods: partitioning, hierarchical, density-based, grid-based, and model-based. The subspace clustering technique is an additional clustering method that attempts to eliminate the unnecessary dimensions from high-dimensional databases in order to produce high-quality clusters. The unnecessary dimensions in high-dimensional databases conceal the true facts (Ram, et al, 2010).

Clustering has been researched for a long time. Many approaches were proposed to divide the data according to similarities and without using significant supervisory signals. Nevertheless, the clustering problem remains an area of open question for three main reasons which are large-scale/big data which leads to high computational complexity and requires processing the excessive amounts within an acceptable time, high-dimensional data which means a high-dimensional space. Increasing the space volume leads to data sparsity, which is a problem and increased chances of overfitting due to high number of features and the last problem

is high-semantic data is one of the most challenging problems. Data with high-level semanticity (human level) such as speech, textual data, videos, and images, generally have complex similarity patterns, which are difficult to pin down mathematically (Mrabah, et al, 2020).

Cluster analysis is applied increasingly to smart meter electricity demand data to find patterns in electricity consumption in order to improve the demand side management (Yilmaz, et al, 2019).

Clustering load profiles is a difficult task because of the difficulty in explaining clusters without extra descriptive household data and the fact that clustering methods frequently fail to account for the temporal aspects of electricity usage. The use of cluster analysis to understand behavioural and other drivers of electricity consumption patterns is constrained by these issues (Meloy, Diakonova, & Grunewald, 2019).

The two main categories of clustering algorithms are partitional and hierarchical. One of the partitional methods that is most frequently used is the K-means clustering technique. Partitional clustering algorithm divides data into nested groups without using a hierarchical framework. The partitioning method can be used to solve clustering issues in large data set applications where creating a dendrogram would be computationally too expensive.

A hierarchical format is used to divide data objects into levels in hierarchical clustering algorithms. To create a dendrogram that shows the hierarchical structure of the formulated clusters, clusters are iteratively made in either a top-down approach which also know as divisive method or bottom-up approach which is also known as agglomerative method (Tazi, et al, 2018).

Deep neural network-based clustering techniques have demonstrated promise for clustering real-world data due to their great representational power (Aljalbout, et al, 2018). In order to learn clustering representations,

Deep learning clustering methods utilize deep neural networks (Ezugwu, et al, 2022). When clustering high-dimensional data, the similarity algorithms utilized in convectional data clustering methods work poorly. For generating data that existing classifiers can easily separate, methods of feature transformation and dimensionality reduction have been developed to translate raw data into new feature space. The efficiency of the current clustering approaches is still hampered by the tremendous complexity of the latent structure of the data. Due to deep learning algorithms' intrinsic extremely non-linear transformation feature, such as deep neural networks (DNN), data can be changed into a representation that is more conducive to clustering. Methods based on DNN have shown promise for accurate and effective grouping of real-world data. Novel deep learning-based clustering techniques already exist that combine deep neural networks with traditional clustering techniques. Two categories of approaches can be made: sequential methods that apply clustering to the acquired DNN representation and the unified approach that maximizes both the clustering objectives and deep representation learning. Deep neural networks are used in the deep embedding clustering (DEC) approach, which simultaneously trains feature representations and cluster assignments. While iteratively maximizing a clustering objective, its operations entail mapping from the data to a lower-dimensional feature space (Ezugwu, et al, 2022).

There have been several dimensionality reduction approaches proposed to avoid the curse of dimension. For a wide range of applications, linear transformations like Principal Component Analysis (PCA) and Factor Analysis are widely used (Mrabah, et al, 2020). Most of the current deep clustering methods connect an autoencoding architecture and other approaches depend on an encoding network without a decoder. The later approach consists of two alternative stages: grouping the embedded features based on a typical clustering algorithm and updating the encoding

weights using the subsequent pseudo-labels as a supervisory signal. The first strategy works as the encoder training is regularized by a reconstruction cost. Hence, the decoder is used to mitigate the effect of training with pseudo-labels. Eliminating the decoding network and training the encoder based on hypothetical similarities can easily corrupt the space topology. Thus, it is possible that the encoder generates random discriminative features, which do not reflect the real discriminative characteristics and this problem is called Feature Randomness (Mrabah, et al, 2020). To lessen the effect of Feature Randomness, autoencoders are provided with their reconstruction capability, which captures prominent features without forcing any randomness.

Dynamic Autoencoder is a deep clustering method for load profiling and a double-stage deep clustering model. Pseudo-labels are unreliable when extracted from random latent representations and it is reasonable to assume that completely random pseudo-labels are not a good to start with as a supervisory signal as it can leads to excessive Feature Randomness, which in turn misleads the whole learning process. Hereafter, the neural network can be pretrained to learn valuable information relying on some pretext task before starting to train using a pseudo-supervised objective function. After that, learning discriminative representations becomes more appropriate by introducing effective pseudo-supervision. Based on that, any self-supervised loss is a candidate for the pretraining phase (Mrabah, et al, 2020).

Using an autoencoder, DEC projects the data points into a lower-dimensional space. The encoder is then trained to enhance the embedded representations and the clustering centers while the decoder is discarded afterward. Before beginning to cluster the data, DEC requires a pretraining step. Improved Deep Embedded Clustering (IDEC) is fairly like DEC. The major contribution of IDEC is balancing the clustering cost with a

reconstruction one after the pretraining phase. Eliminating the reconstruction during the clustering phase hinders the network from conserving the local structures. Embedded clustering does not take into consideration non-discriminative patterns that do not contribute to data categorization, while reconstruction is mainly concerned with preserving all the information whether discriminative or not (Mrabah, et al, 2020).

Load profiling is used to determine a customer's energy demand trend over a given time frame, which is considered as the typical load curve of the group. Load profiling was implemented using engineering or statistical methods. However, in the smart grid time, load profiling is mostly performed via clustering-based methods. Cluster-driven load profiling efforts to group similar consumers into a single cluster based on similarities in consumption patterns (Eskandarnia, Al-Ammal, & Ksantini, 2021). Load profiling can be defined as clustering high dimensional data with no labels, and it is usually performed in three main stages which are: data preparation where the dataset is prepossessed, and dimension reduction is performed to deal with the problem of high dimensionality and compress the data. This stage aims to map raw data into a new feature space, where the generated data are easier to separate by existing classifiers. The second stage is load curve clustering: K-means, Fuzzy Clustering (FCM), Hierarchical Clustering Process, and Self Organize Maps (SOM) are some of the traditional and most widely used clustering algorithms in load profiling. The third and final stage is evaluation where calculating the optimum number of clusters and evaluating the accuracy of clustering outcomes is a challenging task as smart meter readings are unlabeled and no structural knowledge about the data set is available (Eskandarnia, Al-Ammal, & Ksantini, 2021). The most popular method for determining the optimum number of clusters is to run the clustering algorithm several times while attempting various numbers of clusters, and then choose the number

of clusters that produces the best results in a predefined evaluation criterion function. Clustering validity is usually evaluated with visualization, goodness of fit, as well as scalar measurements functions. The clustering quality also can be assessed with a clustering evaluation index such as the Silhouette Coefficient (SI), and Davies-Bouldin Index (Eskandarnia, Al-Ammal, & Ksantini, 2021).

2.1.1. Deep Clustering

Deep clustering has been outperforming more and more traditional shallow clustering algorithms and to attain this performance (Eskandarnia, Al-Ammal, & Ksantini, 2021). Deep clustering algorithms often integrate representation learning with deep neural networks while minimizing a clustering and non-clustering loss. In these situations, an autoencoder is often coupled with a clustering network, and both the autoencoder and the clustering network jointly learn the final clustering (Eskandarnia, Al-Ammal, & Ksantini, 2021).

Data are categorized into labelled (with outcome) that maybe continuous or distinct or unlabelled data (without outcome). Regression is a method of predicting for continuous outcome, and classification is a way of predicting for distinct outcome. Then clustering is applied to unlabelled data using the similarities of patterns between instances to group them into clusters and this statistical method depends on the characteristics of the data, that is the more similarities then the accuracy of the statistical method will be improved.

The performance of clustering algorithms can vary substantially for different types of data and application because they involve several parameters, often operate in high dimensional spaces, and must cope with noisy, incomplete and sampled data. More specifically, each clustering

algorithm depend on a set of parameters that needs to be attuned in order to achieve viable performance, which corresponds to an important point to be addressed while comparing clustering algorithms. Defining a proper procedure for setting the parameters values is a long-standing problem in ML.

Using neural networks, deep clustering develops deep feature representations that facilitate clustering tasks. By specifically defining a clustering-oriented loss, some ground-breaking work suggests learning embedded features and performing clustering at the same time. While promising performance has been shown in several applications, we find that these studies have neglected a crucial factor: the defined clustering loss may corrupt the feature space, creating non-representative meaningless features that negatively impact clustering performance (Guo, et al, 2017).

Different smart meter datasets and other datasets were used to apply different clustering algorithms to compare between the performance of traditional clustering models and deep clustering models by utilizing the load profiling which are detailed in the following sections of this chapter.

2.1.2. Deep Embedded Clustering

Eskandarnia, Al-Ammal and Ksantini (2021) used two-real world smart meter datasets by utilizing the load profiling framework and autoencoder allowing dimensionality reduction and highly nonlinear decision separation between clusters at the autoencoder bottleneck level by representing the data as a hierarchy within the layers of the deep network.

The framework was applied on Irish and UK datasets with three autoencoder clustering techniques and evaluated them with three ML clustering algorithms namely, DEC, IDEC and DynAE. All the three applied algorithms require that the number of clusters k is specified as input and to obtain the optimum number of clusters, the algorithm will perform

grid search for k and calculate the Silhouette Coefficient. To define the best number of clusters the Silhouette Index (SI) was recorded for k from 2 to 10 and the elbow method was then used to find the best number of clusters k . A comparison was done between the Silhouette Index for different clustering algorithms (classical and deep clustering) for different clusters k from 3 to 10 and resulted in having the highest silhouette index achieved by the DynAe and the DEC and IDEC respectively. While the classical clustering algorithms achieved lower silhouette index.

2.1.3. Deep Clustering with Dynamic Autoencoder

Dynamic Autoencoder is performed as a double stage deep-clustering model and the followed methodology consists of optimizing a dynamic objective function to steadily and smoothly reach a better trade-off between feature randomness and feature drift (Mrabah et al, 2020).

Feature randomness arises when a neural network is trained using pseudo-supervised labels and it is characterized by the deviation of the gradient of the real supervised objective function at every iteration. It can be expressed as the cosine of the angle between the gradient of the real supervised objective function and the gradient of the unsupervised objective function. Training with pseudo-labels enforces the neural network to learn features that stress resemblances between data points from different clusters. It also enforces learning to eliminate points from their natural clusters and map them to irrelevant clusters. These implications gave birth to features that contradict the data semanticity. A self-supervised objective function targets learning general-purpose features by solving a pretext task. Although the learned features are less discriminative than the features learned based on categorical labels which are labels that indicate the high semantic categories, for a self-supervised objective function, there are no pseudo-labels.

Feature drift happens when a neural network is trained to minimize a combination of two strongly conflicting loss function, and it is characterized by the level of competition between the gradient of the self-supervised objective function and the gradient of the pseudo-supervised objective function.

In order to validate the suitability of smooth dynamics in solving the stated problem, extensive experiments were conducted where DynAE was compared with several clustering algorithms on four datasets: MNIST-full, MNIST-test, USPS and Fashion-MNIST. For convolutional deep clustering models, the input data were not flattened. All samples were normalized and flattened before feeding them to the fully-connected network. In this study deep clustering methods outperformed the other clustering methods significantly which confirms the convenience of processing high-semantic, high-dimensional and large-scale data using deep neural networks. DynAE beats its deep baselines for all the datasets by a considerable margin comparing among the deep clustering strategies used (Mrabah et al, 2020).

DynAE achieved the state-of-the-art results compared to deep clustering methods in terms of ACC and NMI. DynAE is significantly less sensitive to the setting of γ when compared to IDEC which confirms that the used model offers a better trade-off between Feature Randomness and Feature Drift compared to IDEC (Mrabah, et al ,2020).

The DynAE has achieved the highest ACC & NMI in all the datasets except in the Fashion-MNIST dataset where kSCN model has achieved the highest accuracy of all the models.

2.1.4. Deep Temporal Clustering

DTC which is based on application and requirement, the temporal clustering layer can be customized with any temporal similarity metric. Several similarity metrics and state-of-the-art algorithms were considered and compared. To gain more understanding into temporal features that the network has learned for its clustering, visualization method was applied that generates a region of interest heatmap for the time series (Madiraju, et al, 2018). The feasibility of the algorithm was demonstrated using time series data from different fields, ranging from spacecraft to earthquake sensor data. For both cases it was proved that the proposed algorithm outperforms traditional clustering methods. The superior performance was due to the fully integrated temporal dimensionality reduction and clustering criterion. The proposed DTC ensures that the input signal is encoded into a latent space by a convolutional autoencoder followed by a BI-LSTM.

The convolutional autoencoder and the BI-LSTM constitute a temporal autoencoder (TAE). The latent representation of the BI-LSTM is then fed to a temporal clustering layer generating the cluster assignments. Cluster labels generated by the DTC network were used to train a new supervised hierarchical convolutional network to classify the inputs. Activity in this new network allowed the generation of heatmaps showing the parts of spatio-temporal inputs that are most relevant for assigning inputs to clusters.

The performance of the algorithm as a classifier compared to the baseline metrics was determined based on using only the labels. The area under the curve (AUC) averaged over 5 trials and the baseline metrics AUC's were averaged over 10 runs using bootstrap sampling. The results proved that the performance of the DTC combined with the classical clustering methods used which are ACF, CID, COR and EUCL has outperformed these clustering methods when used by their own only. The

indicated agile dimensionality reduction from inputs with complex and extensive temporal structure to one or few dimensional space spanned by the cluster centroids. As most natural incentives are time-continuous and unlabelled, the approach promised to be of great utility in real-world applications (Madiraju, et al,2018).

2.1.5. Survey of Clustering Algorithms

A survey was soneon the various state-of-the-art clustering methods and algorithms and their application in different research domains (Ezugwu, et al, 2022).

The survey covered both the traditional and recently proposed clustering techniques to address the challenges faced during clustering and how various similarity measures been employed. Also, the research addressed the characteristic differences between the traditional and recently proposed clustering techniques that have been applied in different fields. The main contributions of this survey study were to provide an up-to-date comprehensive systematic review of the traditional and recently proposed clustering techniques that have been applied in different fields, provide a concise presentation of taxonomy, concepts and architecture of clustering algorithms, to present a discussion on open recent research issues relating to clustering problems and to identify possible future research trends and directions regarding the application and implementation of clustering algorithms in different research fields. After comparison and evaluation of the clustering algorithms in terms of strengths and weaknesses, the survey findings showed that the field of economics, bioinformatics, financial systems and economics, image processing, and segmentation attracts more research attention than other domains, such as data mining and object recognition. In addition, it showed

that the K-Means algorithm, Fuzzy C-Means, and Hierarchical clustering technique are among the most widely used clustering algorithms (Ezugwu, et al, 2022). Furthermore, the survey showed that many clustering algorithms (such as nature-inspired ones) have not been fully explored. Moreover, the survey illustrated that the application of newer or hybrid clustering algorithms to a field of choice depending on various trend.

2.2. Summary of Related Studies

Table (1) summarize the literature review

Table (1) Summary of related work

Author	Year	Methods used	Results	Dataset
Eskandarnia, Al-Ammal and Ksantini	2021	1. K-Means 2. GMM 3. Agglomerative 4. CKSC 5. DynAE	DynAE has outperformed all the methods by having the highest SI at cluster 5 & 6 of 0.5090 & 0.5167 respectively and for the other cluster sizes such as 2,3,4,6 & 7	1. Irish smart meter dataset 2. UK smart meter dataset
Mrabah, khan, Ksantini and Lachiri	2020	1. K-means 2. GMM 3. LSNMF 4. AC 5. SSC-OMP 6. EnSC 7. LMVSC 8. kSCN 9. SC 10. RBF K-Means 11. AE+ K-Means 12. AE+FINCH 13. DeepCluster 14. DCN 15. DEC 16. IDEC 17. SR-K- 18. Means 19. VaDE 20. JULE 21. DEPICT 22. DynAE	DynAE gave the highest ACC & NMI of 0.987, 0.964, 0.987, 0.963, 0.981, 0.948 and 0.642 respectively for all the datasets except for the Fashion-MNIST ACC kSCN model gave the highest ACC of 0.637	Several MNIST datasets used: 1. MNIST Full 2. MNIST test 3. USPS 4. Fashion MNIST
Madiraju, Sadat, Fisher and Karimabadi	2018	1. ACF 2. DTC & ACF 3. CID 4. DTC & CID 5. COR 6. DTC & COR 7. EUCL 8. DTC & EUCL 9. Kshape	The study resulted in proving that the performance of the DTC combined with the classical clustering methods used which are ACF, CID, COR and EUCL has outperformed these clustering methods when used by their own only.	Several datasets were used: 1. NASA MMS 2. BeetleFly 3. BirdChicken 4. Computers 5. Earthquakes 6. MoteStrain 7. Phalanges 8. ProximalPhalanx OutlineCorrect 9. ShapeletSim 10. SonyAIBO RobotSurfaceII 11. ItalyPower Demand 12. WormsTwoClass
Ezugwu, Ikotun, Oyelade, Abualigah, Agushaka, Eke and Akinyelu	2022	The Surveyed methods were: 1. Hierarchical clustering algorithms and its subdivisions 2. Partitional clustering algorithm and its subdivisions	1. The field of bioinformatics, financial systems and economics, image processing, and segmentation attracts more research interest such as data mining and object recognition. 2. K-Means algorithm, Fuzzy C-Means, and Hierarchical clustering technique are among the most widely used clustering algorithms. Many clustering algorithms have not been explored fully	Academic Databases which includes: 1. ACM Digital library 2. DBLM 3. IEEE Xplore 4. ISI Web of Science 5. Science Direct 6. Scopus 7. Springer Link 8. Wiley Online Library

This research will address the gap in the related work which is hybrid model based on deep autoencoder and deep temporal clustering for load profiling that can handle feature drift and feature randomness which can result in an accurate prediction of electricity consumption and hence can be used to manage energy demand. In addition, comparing the performance of different clustering algorithms such as DynAE, DEC and IDEC in terms of SI as a performance evaluation metric.

Chapter Three

Methods and Procedures

3.0. Methodology

3.1. Temporal Autoencoder (Madiraju, et al, 2018)

3.2 Dynamic Autoencoder (DynAE) (Eskandarnia, Al-Ammal, and Ksantini, 2021)

3.3. Deep Embedded Clustering (DEC) (Eskandarnia, Al-Ammal, and Ksantini, 2021)

3.4. Improved Deep Embedded Clustering (IDEC) (Eskandarnia, Al-Ammal, and Ksantini, 2021)

Chapter Three

Methods and Procedures

3.0. Methodology

This work proposes a framework based on an indirect clustering technique for load profiling that utilizes autoencoders for the dimensionality reduction step. Specifically, three deep autoencoder based clustering algorithms were used for the proposed load profiling framework, namely DEC, IDEC and DynAE which improves the trade-off between feature randomness and feature drift. The proposed algorithms project the input data into a lower dimensional feature space where separation and similarity are better stressed with respect to the problem's context by implementing embedded data representation and dimensionality reduction alongside.

Fig. (2) represents the experiments stages and the results generated from each algorithm applied and the final results for the performance evaluation. The flow chart is further explained in detail in this section.

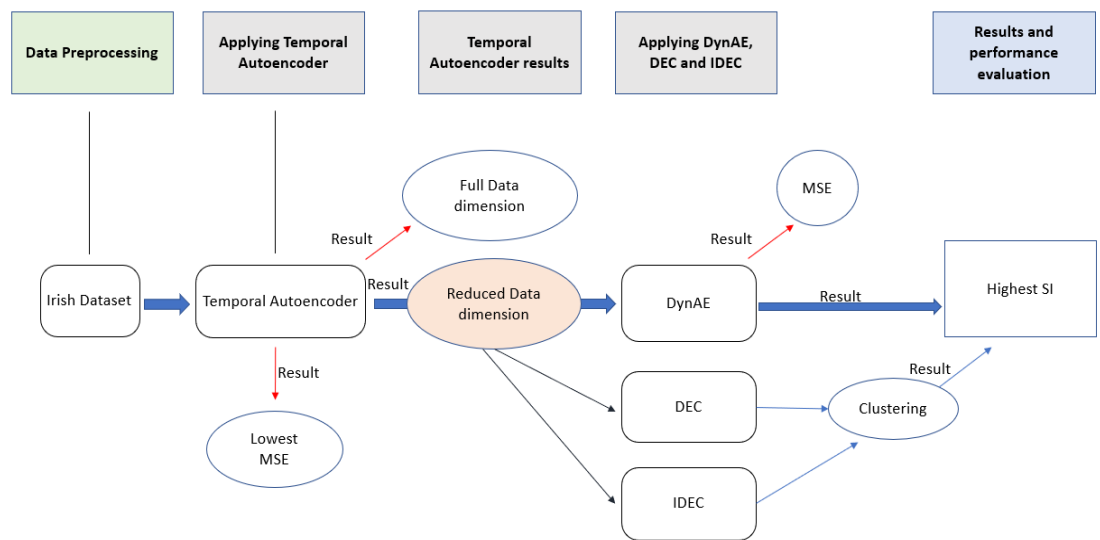


Fig. (2) Experiment protocol Flow chart

3.1. Temporal Autoencoder

First, by using an autoencoder the framework represents data as a hierarchy within the layers of the deep network, allowing for dimensionality reduction and highly nonlinear decision separation between clusters at the autoencoder bottleneck level. The TAE framework is tested on Irish smart meter dataset where the dataset was encoded into a latent space by a convolutional autoencoder followed by a BI-LSTM. The convolutional autoencoder and the BI-LSTM constitute a TAE. The latent representation of the BI-LSTM is then fed to a temporal clustering layer generating the cluster assignments. Several parameters were applied with different settings on the dull dataset dimension and changed the dataset dimension by reducing it to reach to the minimum MSE.

Fig. (3) presents the overview of deep temporal clustering.

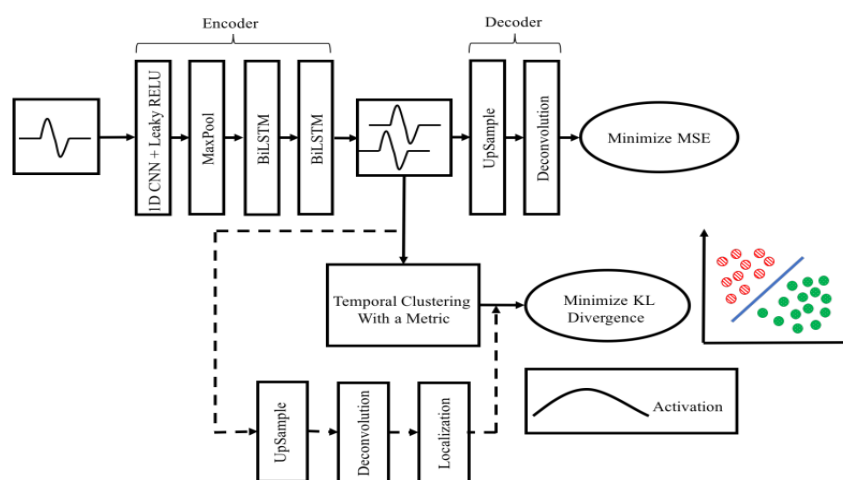


Fig. (3) DTC overview (Madiraju, et al, 2018)

As a result, the load profiling task is substantially enhanced. The suggested approach generates unique load curves for each cluster which can serve as the basis for other energy management functions.

3.2. Dynamic Autoencoder

DynAE addresses the trade-off between feature randomness and feature drift which improves the feature randomness. It is an improved way of obtaining a K-Means friendly latent space. Achieving state-of-the-art clustering performance on real benchmark datasets. Started the DynAE procedure with trials on the full dataset along with changing different parameters to generate different SI scores, then the reduced dataset of 700 which has achieved the lowest MSE in the temporal autoencoder experiment was fed in the DynAE and multiple trials were performed with different settings to generate different SI scores at different clusters, the clustering procedure starts, and the output is a vector of k with a size $n \times 1$ where k is from 3 to 10. The complete clustering framework is shown in Fig. (4)

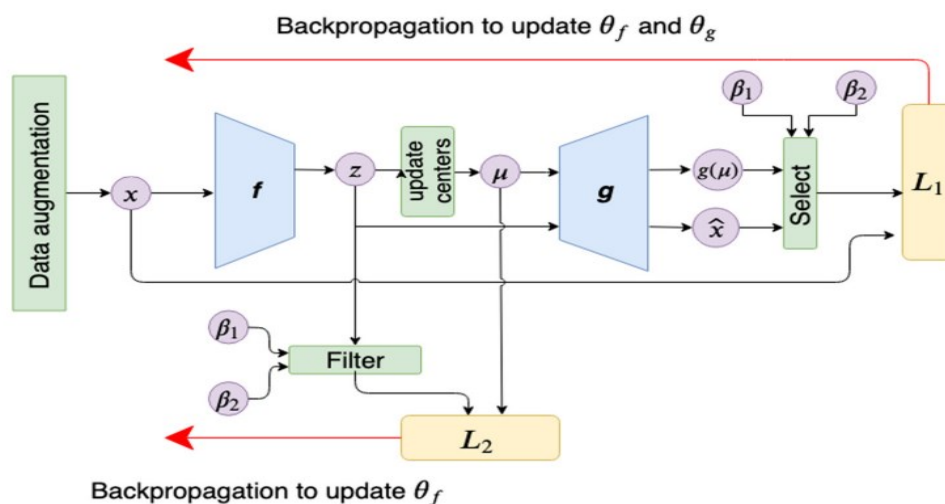


Fig. (4) Clustering phase of the DynAE (Mrabah, et al, 2020)

Illustrated in Fig. (5) the general clustering architecture of DynAE.

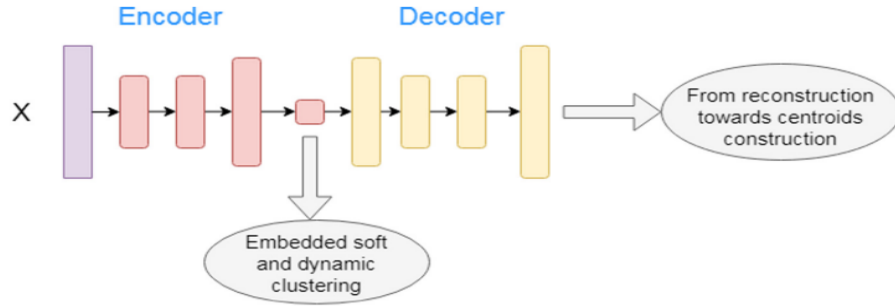


Fig (5) General clustering architecture of DynAE (Mrabah, et al, 2020)

3.3. Deep Embedded Clustering (DEC)

One of the used methods is DEC which performs feature learning and clustering assignment simultaneously and can be used with raw datasets. Deep Embedding for Clustering projects the data points in a lower-dimensional space using an autoencoder and followed by the decoder, it is discarded, and the encoder is trained to jointly improve the embedded representations and the clustering centres. The reduced dataset of 700 was fed into the model and different parameters were applied including epochs and number of iterations to reach at the highest SI score of all the trials.

3.4. Improved Deep Embedded Clustering (IDEC)

IDEC was also applied on the reduced dataset of dimension 700 by using different parameters such as the epochs and number of iterations to reach at the highest SI score for the specified k.

The DEC, IDEC and DynAE networks are trained by matching the soft assignment to the target distribution, so the clusters can be iteratively refined by learning from their high confidence assignment, with the assistant of an auxiliary target distribution

The clustering quality is assessed with a clustering evaluation index such as the Silhouette Index (SI). The Silhouette Index (SI) was adopted as an evaluation criterion, which is a famous measure for unlabelled data that has proven to be a good metric. SI values range between -1 and 1, and the higher the SI score the better the clustering results.

The performance of the DynAE, DEC and IDEC algorithms is evaluated for k by running the best clustering algorithm according to the SI with k and find the cluster of each house. Once the clustering procedure is finalized the load profile is obtained by merging two metrics then a resampling of time series to day-wise and visualize. Finally, evaluating the load profile task and select the best performance.

Chapter Four

Results and Discussion

4.0. Study Dataset

4.1. Experiment Protocol

4.2. Results

4.3. Discussion

Chapter Four

Results and Discussion

4.0. Study Dataset

The dataset used in this research was requested from the ISSDA Irish open smart meter dataset (CER Smart Metering Project Electricity Customer Behaviour Trial, 2009-2010) through the website (<https://www.ucd.ie/issda/>) which records the smart meter readings for 6345 homes from 2009 to 2010, having a dimension of 1152 which are the time series that starts from 1pm on the 14th of July. Table (2) summarize the Irish dataset descriptions.

Table (2) Irish dataset description

Data Source	ISSDA Irish open smart meter dataset (CER Smart Metering Project Electricity Customer Behaviour Trial, 2009-2010). The website (https://www.ucd.ie/issda/)
Data Dimension	1152 columns
Data Columns	Hourly Energy Consumption
Data Rows	6345 (Homes/Smart Meter IDs)
Time frame	2009 to 2010 and starts from 1 p.m.

The dataset was preprocessed before using it for the analysis. For example, zero reading for the hours in the dataset has been replaced with the average reading of the same month, while for the months where the whole month reading was zero, the value was kept as it is at zero. Moreover, the dataset was flipped from vertical presentation of the readings to horizontal presentation, i.e. the rows were the smart meter IDs and the columns were the hours and date combination and the energy consumption readings were presented for each smart meter ID (row) for

each hours and data combination (columns). Changing the presentation of the data from vertical to horizontal have reduced the number of the files from six csv files originally to one csv file that contains all the required information. The columns have specific numbering system such as 19501, 20337, 21484 and etc, these numbers have a meaning which is the first three digits for example 195 is the day code and 01 is the time code. Table (3) presents parts of the study dataset after it has been preprocessed. The first column on the left of Table (3) is the smart meter ID number, it starts with 1000 and continues as serial.

Table (3) Part of the dataset after the pre-processing

	19501	19502	19503	19504	19505	19506	19507	19508
1000	0.12	0.05	0.03	0.03	0.05	0.12	0.04	0.02
1001	0.29	0.09	0.09	0.15	0.09	0.09	0.15	0.09
1002	0.36	0.06	0.12	0.02	0.14	0.04	0.11	0.08
1003	0.69	0.38	0.38	0.38	0.35	0.27	0.28	0.35
1004	1.31	1.14	1.22	0.9	0.79	0.81	0.92	0.54
1005	0.18	0.17	0.15	0.16	0.16	0.15	0.17	0.14
1006	0.05	0.09	0.1	0.05	0.06	0.11	0.06	0.04
1007	0.32	0.4	0.23	0.24	0.22	0.23	0.27	0.2
1008	0.86	0.37	0.41	0.4	0.38	0.31	0.15	0.25
1009	0.61	0.55	0.59	0.62	0.52	0.38	0.46	0.46
1010	0.71	0.74	0.6	0.66	0.58	0.65	0.58	0.64
1011	0.47	0.43	0.34	0.22	0.32	0.33	0.26	0.19
1012	1.29	1.13	1.35	1.61	0.92	1.27	1.24	1.4
1013	0.21	0.21	0.14	0.13	0.13	0.12	0.05	0.05

It is noticed that some of the energy consumption have the same pattern and trend in terms of increasing at some point in time and decreasing at other times. This graphical presentation of the pattern is defined as the load profile and vary according to the customer type, holiday seasons and temperature. Fig. (6) illustrates the load profile graph for the used dataset where it starts at 1 a.m at a relatively high consumption and reduce drastically at 7 a.m and then increases again to reach the maximum consumption at 8 p.m. and starts to decline back after 9 p.m.

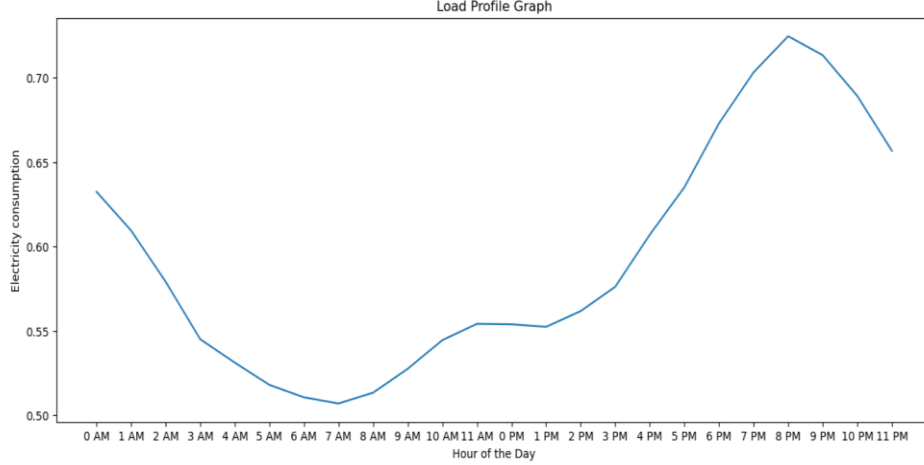


Fig (6) Irish dataset load profile graph

4.1. Experiment Protocol

Google Collab was used to perform the TAE on the preprocessed dataset. Python language was used, and all the necessary libraries were used such as pandas and numpy, in addition to tensorflow and keras libraries to train and interface the deep neural network.

Python package tslearn was used to perform the analysis of time series. Dataset was standardized and normalized by using Standard Scaler. Gaussian mixture model was also used to integrate information about to covariance structure of the data and the centre of latent.

The experiments started with the TAE where it was done several times on the full dataset dimension and the dataset with reduced dimension by reducing the dimension for different sizes each time. Furthermore, changing different parameters to find lowest MSE. The parameters that were changed are optimizers, epochs, batch size and filter size. For the optimizers, Adam and SGD were used. For the epochs 10, 50, 100, 200 and 300 were used. For the batch size, 100, 500 and 2000 were use and for the filters, 3, 5 and 7 were used. The dataset dimensions tried were 1152 which the full data dimension, 1000, 700, 600, 500 and 200.

Table (4) shows the parameters settings tried for the temporal autoencoder.

Table (4) Temporal Autoencoder parameters

Dataset Dimension	Full
Evaluation Method	MSE
<u>Parameters changes through out the experiment</u>	
Optimizers	Adam and SGD
Epochs	10,50, 100, 200 and 300
Batch size	100, 500 and 2000
Filters	3,5 and 7
Dataset dimensions opted	1152 which the full data dimension, 1000, 700, 600, 500 and 200

In most of the cases it was noticed that the mse was decreasing with the increase of epochs.

Based on the generated results using the TAE, the combination of the parameters used and the data dimension with the lowest MSE was selected to be experienced using the dynamic autoencoder.

The Mean Square Error (MSE) which is calculated by

$$MSE = \frac{\sum (P_i - O_i)^2}{n} \quad (1)$$

where $\sum$ means the sum

P_i is the predicted value of the i th observation

O_i is the observed value for the i th observation

n is the sample size

The DynAE was tested on both the full and reduced dataset. For the reduced dataset, the data dimension with the lowest MSE was selected to be tested which is 700. Started the model with generating the SI and cluster visualization for clusters from 2 to 10 for the full dataset and reduced dataset before applying the DynAE. The parameters that were used and

changed throughout training of the model are gamma, the bottleneck shape which is also referred to as the neural network shape, the number of output layers, the number of input layers, number of epochs and number of iterations. Gamma values were set to 1, 0.001, 0.01, 0.1, 1, 10, 100 and 1000. The neural network shape was set from 8 to 15. Output layers opted were 1, 2 and 3 and input layers were 1 and 2. Number of epochs used were 10, 50, 100, 150, 200, 250 and 300. Number of iterations tried were 10, 100, 200, 1000, 2000 and 10,000.

Table (5) shows the parameters settings tried for the DynAE.

Table (5) DynAE parameters

Dataset Dimension	Full and reduced
Evaluation Method	SI
<u>Parameters changes through out the experiment</u>	
Gamma	1, 0.001, 0.01, 0.1, 1, 10, 100 and 1000
Neural network shape	8 to 15
Input Layers	1 and 2
Output Layers	1, 2 and 3
Epochs	10, 50, 100, 150, 200, 250 ,300 and 350
Iterations	10, 100, 200, 1000, 2000 and 10,000

Then the experiment was focused on 4 clusters (K=4) while changing the other parameters since it was the cluster with the highest SI score when the data dimension was reduced.

The SI is calculated using the mean intra-cluster distance (a) and the mean nearest-cluster distance (b) for each sample. The SI can be expressed by

$$S(i) = \frac{(b(i)-a(i))}{\max\{a(i),b(i)\}} \quad (2)$$

Where a(i) - the mean distance between i and all other points in its own cluster and b(i) - the distance between i and its next nearest cluster centroid.

DEC algorithm was applied on the reduced dataset of dimension 700 and the evaluation method was the SI and the MSE which was recorded for the experiments done. The parameters that were changed while running the DEC algorithm were the number of epochs which were 10, 50, 100, 150, 200, 250 and 300 and iterations which were ranging from 0 to 27160.

Table (6) shows the parameters settings tried for the DEC.

Table (6) DEC parameters

Dataset Dimension	Reduced of 700
Evaluation Method	SI
<u>Parameters changes through out the experiment</u>	
Epochs	10, 50, 100, 150, 200, 250 and 300
Iterations	0 to 27160

The IDEC algorithm was applied on the reduced dataset of dimension 700 and the evaluation matrix is the SI. The parameters that were changed are the number of epochs which were 10, 50, 100, 150, 200 and 300 and iterations which were ranging from 0 to 27160. Table (7) shows the parameters settings tried for the DEC.

Table (7) IDEC parameters

Dataset Dimension	Reduced of 700
Evaluation Method	SI
<u>Parameters changes through out the experiment</u>	
Epochs	10, 50, 100, 150, 200, 250 and 300
Iterations	0 to 27160

After conducting all the required experiments of this research, the questions of the research can be answered based on the acquired results. The first question which is how can load profiling be improved in predicting energy consumption by using time series data? The load profiling is improved by clustering which also improves the prediction accuracy and the load forecasting by implementing autoencoder-based clustering which is the

temporal autoencoder and the DynAE. The temporal autoencoder indicated the data dimension reduction and the DynAE has improved the load profiling by producing four load profiles of customers groups having similar patterns of electricity consumption.

The second question which is how can clustering models tease out temporal information by handling feature drift problems? Clustering models can tease out temporal information by handling feature drift problems is by using the DynAE model so that the load profiling can handle time series data as well as feature randomness and feature drift.

The third question is which clustering approach can result in the best prediction? The clustering approach that resulted in the best prediction is the DynAE by scoring the highest SI between the DEC and IDEC.

4.2. Results

The experiments are done for all the models, starting with the TAE, followed by the DynAE, DEC and IDEC to arrive at the optimal cluster numbers that will give the highest SI score as the main evaluation matrix in this study. A secondary evaluation matrix was the MSE which was also recorder during the results generation.

The purpose of applying the TAE is to train the model with different data dimensions to arrive at the data dimension with the lowest MSE which is the evaluation matrix used in this model. Different parameters were changed throughout the experiment to arrive at the lowest MSE as stated in the Experiment protocol section with further details. It was noticed that the MSE is reducing with the increase number of epochs regardless of the data dimension or the optimizer which is due to training the neural network more when increasing the number of epochs.

The lowest MSE was generated for data dimension 700 while training the TAE. The MSE was 0.0052 which was lower than the original data dimension of 1152 where the MSE was 0.0076. It was also lower than the MSE for data dimension 1000 where the MSE was 0.0064. The same data dimension of 700 was tested with SGD optimizer but the MSE was one of the highest of 0.2168. Table (8) shows the MSE results generated for different epochs for dimension 700, filter size 5, batch size 100 and Adam optimizer, the lowest MSE is at 200 epochs which is 0.0052 in bold.

Table (8) MSE results for data dimension 700

input dimension	700				
Filter size	5				
batch size	100				
optimizer	Adam				
Epochs	10	50	100	200	300
MSE	0.1435	0.0341	0.0216	0.0052	0.0076

Table (9) shows the MSE results generated for different epochs for dimension 1000 as one of the best results generated that comes in second place after the reduced dimension of 700 mentioned in Table (3). The lowest MSE is 0.0064 at 200 epochs in bold.

Table (9) MSE results for data dimension 1000

input dimension	1000				
Filter size	5				
batch size	500				
optimizer	Adam				
Epochs	10	50	100	200	300
MSE	0.1919	0.0538	0.0121	0.0064	0.0157

Table (10) shows the results of the MSE for the original data dimension of 1152 and the lowest MSE was 0.0076 which is at 300 epochs in bold.

Table (10) MSE results for original data dimension 1152

input dimension	1152				
Filter size	3				
batch size	500				
optimizer	Adam				
Epochs	10	50	100	200	300
MSE	0.173	0.0398	0.0162	0.0254	0.0076

Table (11) shows the MSE results generated for different epochs using data dimension 700 and with optimizer SGD, the lowest MSE is 0.2168 at 100 epochs which is higher than the MSE's in the previous experiments.

Table (11) MSE results for SGD optimizer

input dimension	700				
Filter size	7				
batch size	2000				
optimizer	SGD				
Epochs	10	50	100	200	300
MSE	0.9000	0.2907	0.2168	0.3359	0.2236

In addition, when the SGD optimizer was used it gave higher MSE than the experiments using Adam optimizer which indicates that the Adam optimizer is the best to use in this case. Adam optimizer is known in the field of machine learning in achieving good results in a fast and efficient way and considered as a replacement optimization algorithm for SGD for training deep learning models. It combines the best settings of the RMSProp and AdaGrad algorithms to provide an optimized algorithm that can handle sparse gradients on noisy problems. Adam and SGD optimizers were used in this research.

It was noticed that reducing the data dimension has outperformed the other experiences with the original data dimension in terms of the MSE as an evaluation method. Based on that, dimension 700 is chosen to be trained on the DynAE, DEC and IDEC models.

Table (12) shows the summary of MSE results for the most well performing trials performed regardless of the parameters setting. The columns are the data dimension and the rows are the number of epochs. The low MSE indicates a better fit and is used to check how close estimates are to actual values and the lower the MSE, the closer is the estimate to actual.

Table (12) MSE results per dimension and per epoch

	Data Dimension					
Epochs	1152	1000	700	600	500	200
10	0.1731	0.1919	0.1435	0.2987	0.3091	0.3498
50	0.0398	0.0538	0.0341	0.1322	0.1348	0.0948
100	0.0162	0.0121	0.0216	0.0783	0.0806	0.0635
200	0.0254	0.0064	0.0052	0.0209	0.0214	0.0186
300	0.0076	0.0157	0.0071	0.0124	0.0121	0.0106

Fig. (7) illustrates the trend of the MSE against each data dimension applied which indicated a decrease in the MSE throughout changing the parameters.

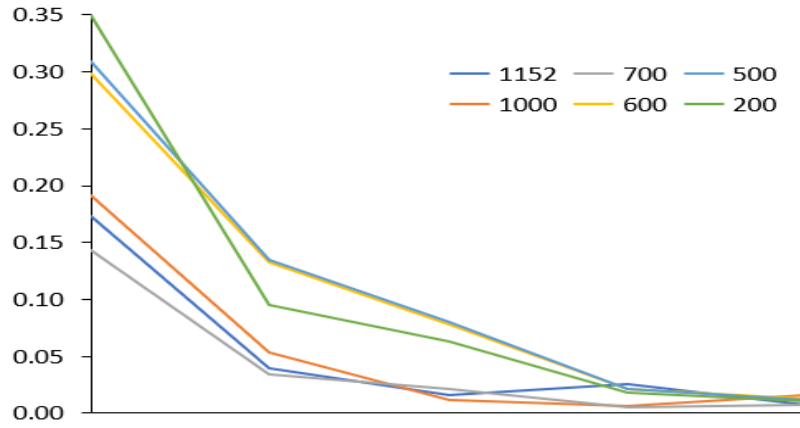


Fig. (7) MSE pattern for each data dimension

Table (13) represents the MSE results for different data dimensions and for data dimension 700 it is presented for two different optimizers, Adam and SGD. The same results are reflected in Fig. (8)

Table (13) MSE results for different data dimension

Data Dimension	1152	1000	700 (Adam)	700 (SGD)
MSE	0.0076	0.0064	0.0052	0.2168

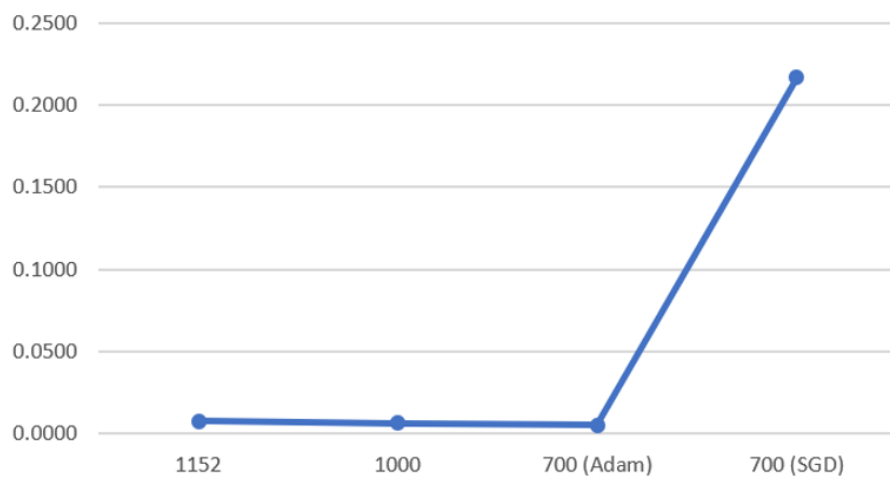


Fig. (8) MSE for each data dimension

Appendix A lists more results of the temporal autoencoder along with the parameters setting for more details.

The DynAE experiments were done on the full and reduced datasets to compare the SI score results between both datasets and to confirm that the reduced data dimension will result in the highest SI. SI was also used to confirm the optimal cluster numbers which is 4 by using the SI score as the indicator. When using the full dataset, the highest SI score achieved was 0.8697 at 2 clusters and for reduced dataset of 700 the highest SI score was 0.8715 at 4 clusters. The highest SI score achieved at these experiments was through extensive trials on reduced data and 4 clusters.

The characteristics and properties of the reduced dataset are: data dimension of 700, the sparsity of the data was reduced as the gaps in the data were reduced by using the monthly average at the preprocessing stage and reducing data dimension didn't increase the sparsity and finally the few outliers found in the dataset were removed.

For the DynAE the SI score was the evaluation matrix and the full dataset was used to run the results and the reduced dataset used based on the lowest MSE given using the temporal autoencoder which is data dimension 700. Cluster numbers was part of the experiments in this stage which was treated as a parameter that can be changed, also the neural network shape was part of the parameters as well, these two factors weren't part from the temporal autoencoder.

Fig. (9) presents a visualization of the clusters for clusters 2 and 5 for the reduced dataset.



Fig. (9) Clusters Visualization

It was noticed that the SI score for the full dataset was relatively low which has led to using the reduced data to arrive at one of the highest SI scores at this stage. The highest SI score was at 4 clusters, so extra experiments were done on 4 clusters specifically to ensure that it is the best cluster to use. Table (14) shows the SI for clusters from 2 to 10 for the full dataset and the reduced dataset before training the model which shows that the highest SI score for both data dimensions is at 2 clusters.

Table (14) SI score for cluster 2 to 10 before training the model for the full dataset and reduced dataset

Clusters	SI for the full dataset	SI for the reduced dataset
2	0.6249	0.6252
3	0.5874	0.5881
4	0.5681	0.5691
5	0.5551	0.5566
6	0.5461	0.5479
7	0.5382	0.5405
8	0.5325	0.5352
9	0.5279	0.5312
10	0.5226	0.5264

Table (15) shows the results for the highest SI achieved when using the full dataset and the parameters setting which is of 0.8697 at 2 clusters, 250 epochs and 5000 iterations. The MSE was also generated during the DynAE experiments but the focus was to evaluate the model based on SI score.

Table (15) SI score when using the full dataset

Clusters	2						
Gamma	1						
Shape	-1	9					
Input layers	1						
Output layers	1						
Epochs	10	50	100	150	200	250	300
Mse	0.3191	0.2331	0.2173	0.2125	0.2144	0.2267	0.2232
Maxiter	100	1000	2000	3000	4000	5000	6000
SI Score	0.7112	0.7845	0.8145	0.8565	0.8502	0.8697	0.8655

Table (16) shows the highest results for the SI when using the reduced dataset of dimension 700 and the parameters setting.

Table (16) SI score when using the reduced dataset

Clusters	4						
Gamma	0.1						
Shape	-1	10					
Input layers	1						
Output layers	1						
Epochs	10	50	100	150	200	250	300
Mse	0.3216	0.2162	0.2004	0.2121	0.2146	0.2217	0.2318
Maxiter	10000	10000	10000	10000	10000	10000	10000
SI Score	0.8715	0.8312	0.7258	0.7248	0.7557	0.7675	0.7769

Since cluster 4 has achieved the highest SI score in the DynAE for the reduced data, more experiments were done by fixing the number of clusters to 4 and changing the other parameters accordingly to arrive at the highest SI score.

Table (17) shows the highest SI achieved of 0.9931 with 4 clusters, 10 epochs and 700 iterations.

Table (17) SI score for 4 clusters

Clusters	4						
Gamma	1						
Shape	-1	8					
Input layers	1						
Output layers	1						
Batch size	128	128	128	128	128	128	128
Epochs	10	50	100	150	200	250	300
Mse	0.3743	0.2307	0.2128	0.1935	0.1649	0.1727	0.207
Maxiter	700	700	700	700	700	700	700
SI Score	0.9931	0.9888	0.9799	0.8223	0.8691	0.9285	0.9287

Extensive experiments were conducted on the DynAE using the full dataset and the reduced dataset which are presented in more details in Appendix B.

DEC was applied after the DynAE on the reduced dataset of dimension 700 and resulted in the highest silhouette score generated at 4 clusters and will be presented in the DEC results. Table (18) demonstrates the number of iterations and the SI score. The best SI for the DEC model was produced for 4 clusters at 27160 iterations and 250 epochs.

Table (18) SI and the MSE for the number of epochs and iterations tried.

Maxiter	Epochs	SI Score	MSE	Maxiter	Epochs	SI Score	MSE
0	10	0.5717	0.0000	1120	50	0.4848	0.1084
140	50	0.4408	1.1526	1260	100	0.4713	0.1136
280	100	0.5469	0.0367	1400	150	0.5161	0.1134
420	150	0.4282	0.0564	1540	200	0.5306	0.1185
560	200	0.4515	0.0254	1680	250	0.5551	0.1151
700	250	0.4429	0.0769	1820	300	0.5548	0.1020
840	300	0.4419	0.1033	1960	10	0.5577	0.1070
980	10	0.4713	0.1168	2100	50	0.5586	0.1075

Table (19) illustrates the settings for the highest and the lowest SI in the DEC along with the MSE of 0.0564 and 0.0787. The lowest MSE was

0 at the first run which was at 0 iterations and 10 epochs, while the highest MSE was of 1.1526 at 140 iterations and 50 epochs.

Table (19) DEC highest and lowest SI score

Maxiter	Epochs	SI Score	MSE
420	150	0.4282	0.0564
27160	250	0.6128	0.0787

Table (20) illustrates the settings for the highest and the lowest MSE in the DEC along with the SI score of 0.5717 and 0.4408.

Table (20) DEC highest and lowest MSE

Maxiter	Epochs	SI Score	MSE
0	10	0.5717	0.0000
140	50	0.4408	1.1526

More of the results for DEC are presented in more details in Appendix C.

The last model which is the IDEC has generated the second highest SI score after the DynAE of 0.8829 at 4 clusters which is confirming the number of clusters of 4 with the DynAE and the DEC.

First the SI was run for different clusters ranging from 2 to 10 and Table (21) shows the SI result for each cluster which indicated that the highest SI score is at 4 clusters of 0.8829

Table (21) SI score for IDEC

Clusters	SI Score
2	0.8111
3	0.7841
4	0.8829
5	0.6816
6	0.6925
7	0.5887
8	0.5643
9	0.5525
10	0.4236

Fig. (10) shows the principle component analysis for the reduced dataset. Fig. (11) and Fig. (12) represents the clusters 2 and 3 visualization for the reduced dataset.

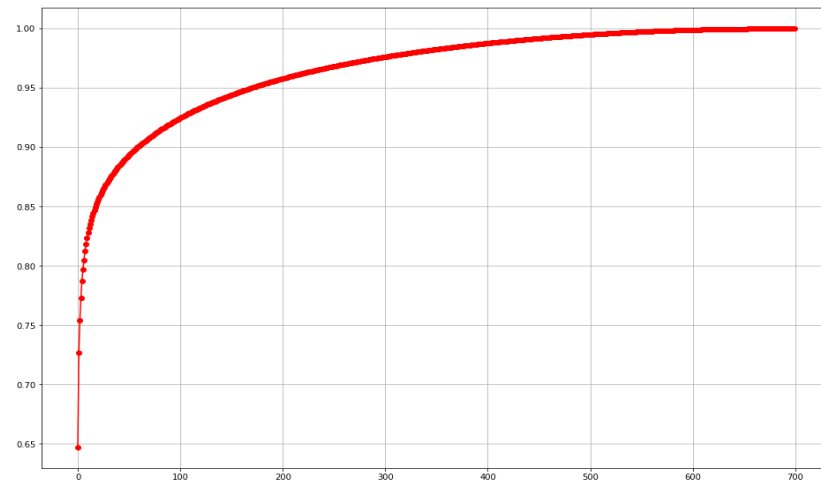


Fig. (10) screen plot of the principle component

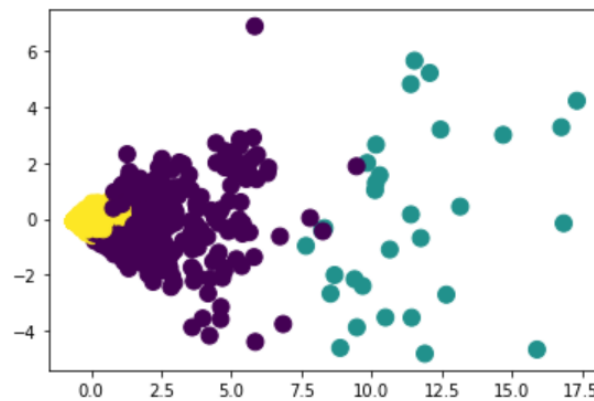


Fig (11) Shape of data points for 3 clusters (K=3)

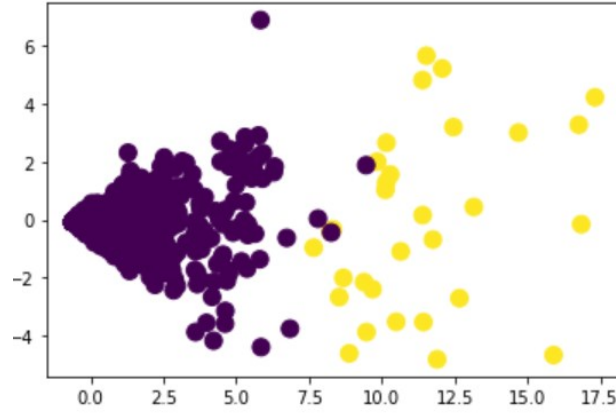


Fig (12) Shape of data points for 2 clusters (K=2)

Fig (13) shows the difference between the clusters of the subset is normalized and non-normalized.

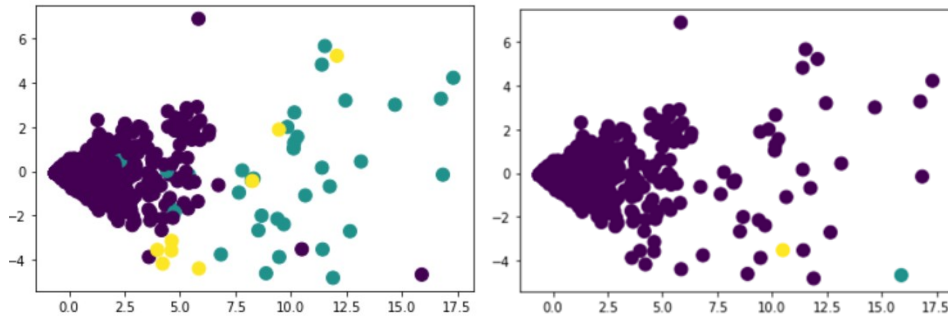


Fig (12) Normalized and non-normalized dataset

The normalized dataset shows more organized and clearer clusters where it is noticeable that there are 3 clusters while the non-normalized dataset shows 1 major cluster and 1 data point for the other two clusters. Normalizing the dataset help in having more clustered indexes.

Finally, the elbow method was performed to confirm that the best cluster to use is 3 clusters based on the lowest Sum of Squared Error (SSE) which is consistent with the results generated in the DynAE, DEC and IDEC. It also noticed SSE is not reducing significantly after cluster 4.

Fig. (14) presents the elbow curve for cluster from 2 to 10 and SSE.

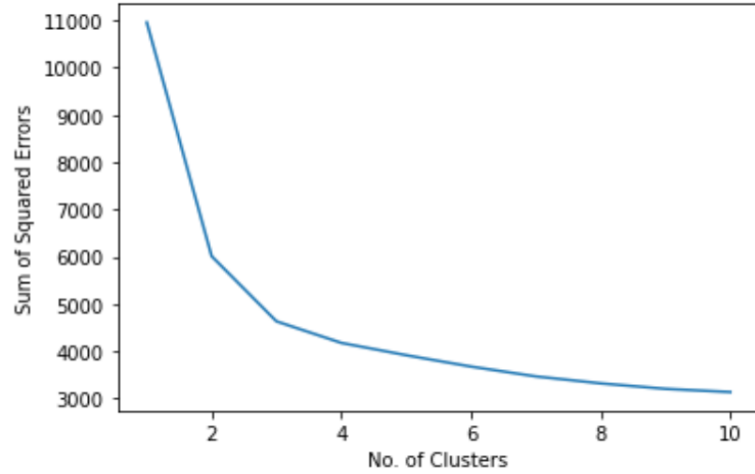


Fig (14) Elbow curve

Fig. (15) illustrates the dendrogram generated from the IDEC algorithm to decide the number of clusters which indicates that $K=3$ based on the highest vertical distance.

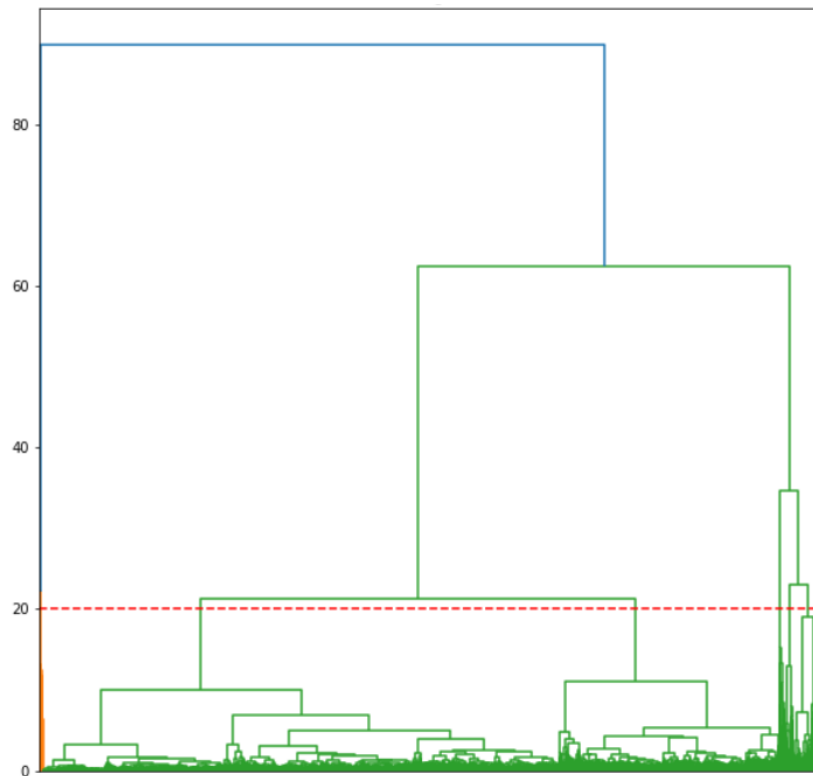


Fig (15) IDEC Dendrogram

The correlation between the resulted 4 clusters was measured and it was found that clusters 1 and 2 are correlated, clusters 1 and 4 are

correlated and clusters 2 & 4 are correlated which means that whenever there is an increase or a decrease in the electricity consumption in one cluster there is also increase or decrease in the electricity consumption in the other cluster at the same. Cluster 3 is not correlated with clusters 1,2 and 4 which means that the increase or decrease of electricity consumption in cluster 3 is not consistent with the movement in electricity consumption in the other clusters. Fig. (16) illustrates the correlation between the 4 clusters.

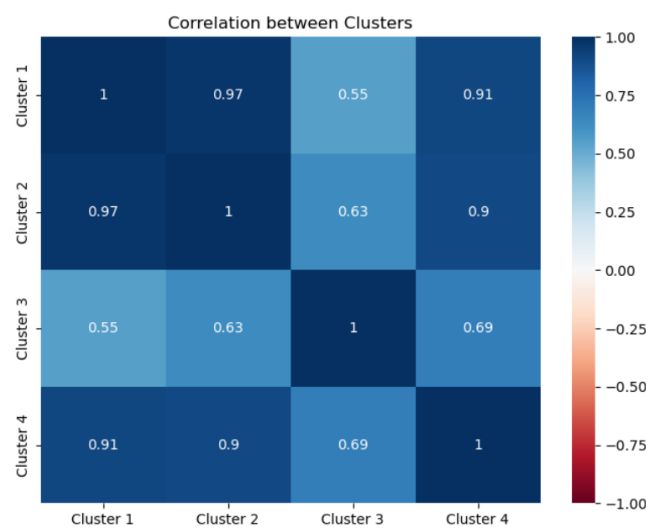


Fig (16) Clusters Correlation

To confirm the correlation between the 4 cluster, a visualization on the electricity consumption for all the clusters was produced to see the trend in the electricity consumption at different times of the day. Fig. (17) shows the trend plot for electricity consumption for cluster 1 over the hour in the day.

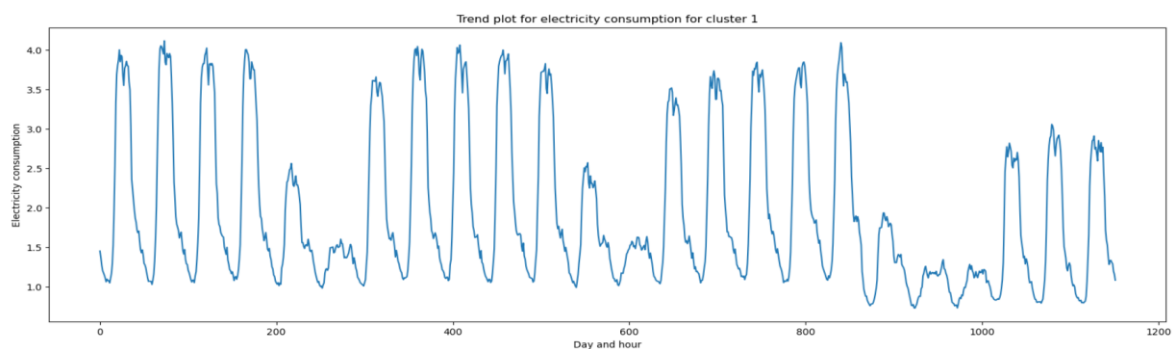


Fig (17) Cluster 1 trend plot

Fig. (18) shows the trend plot for electricity consumption for cluster 2 over the hour in the day.

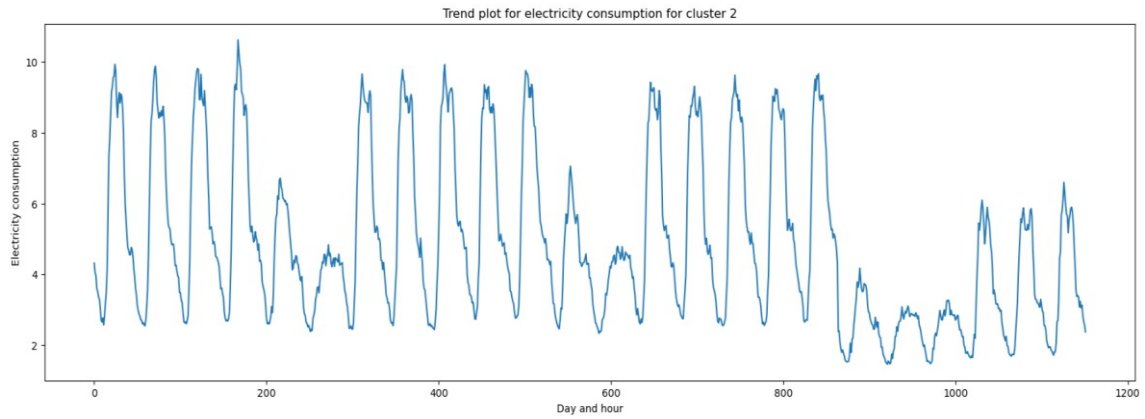


Fig (18) Cluster 2 trend plot

Fig. (19) shows the trend plot for electricity consumption for cluster 3 over the hour in the day.

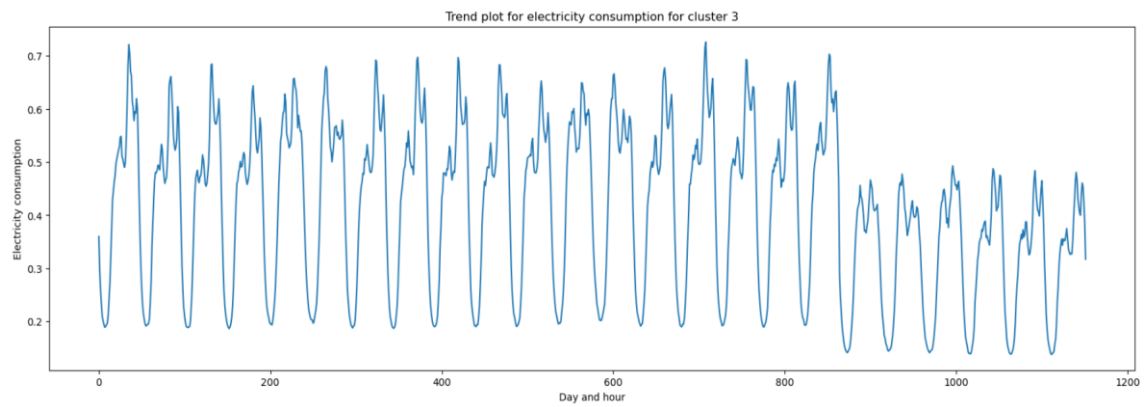


Fig (19) Cluster 3 trend plot

Fig. (20) shows the trend plot for electricity consumption for cluster 4 over the hour in the day.

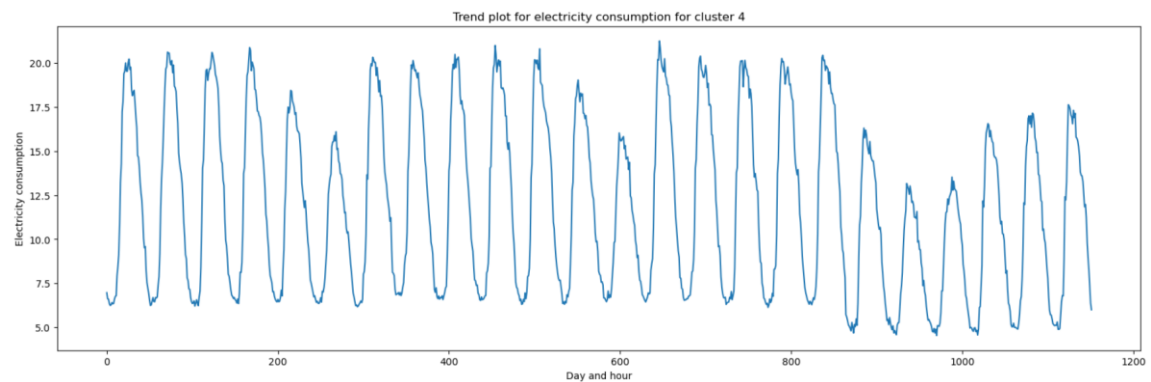


Fig (20) Cluster 4 trend plot

The monthly electricity consumption for the 4 clusters indicates that cluster 4 have the highest consumption of 62.2% of the total consumption, cluster 2 comes in second of 25.5%, then cluster 1 of 10.1% and finally cluster 3 of 2.2% of the totla consumption. The monthly consumption percentage is visualized in Fig. (21)

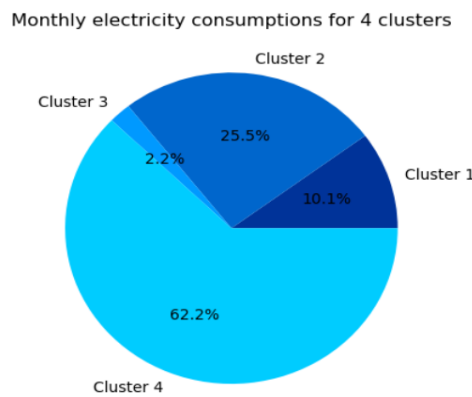


Fig (21) Monthly electricity consumption

To confirm the trend of the monthly electricity consumption between the clusters, a graph which is Fig. (22) for the average daily electricity consumptions was produced which further confirms that cluster 4 have the highest consumption, cluster 2 next, cluster 1 in third place and finally cluster 3 having the lowest average daily consumption.

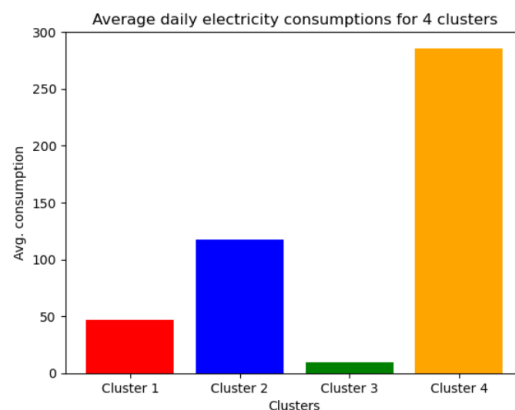


Fig (22) Clusters average daily consumption

The performance evaluation metrics in this research are the MSE and the SI. The temporal autoencoder evaluation matrix was the MSE where

the lowest MSE achieved for the reduced data was selected as the best performance and based on it the data dimension was selected.

The evaluation method for the DynAE is the SI which is a measure of how similar an object is to the cluster it belongs to (cohesion) compared to other clusters (separation) and it can be used to study the separation distance between the resulting clusters. In addition to the SI score the MSE was also recorded for the experiments done but the main focus as an evaluation metric is the SI score for the full and the reduced data. The same evaluation matrix which is the SI is used for the IDEC and DEC.

4.3. Discussion

The temporal autoencoder was applied to reduce that data noise (dimensionality reduction) by selecting the data dimension with the lowest MSE of 0.0052 which was 700. Compared with Madiraju, et al. (2018) study, it is consistent with it in terms of using time series datasets and the integrated temporal dimensionality reduction achieved where the resulted reduced data was later used to be further expirementented by the DynAE, DEC and IDEC. Madiraju, et al. (2018) have used Receiver Operating Characteristics (ROC) and area under the curve (AUC) as evaluation metrices which have proved the DTC to be robust and results in superior performance.

The DynAE has outperformed the DEC and IDEC in SI as an evaluation matrix. The DynAE has achieved SI of 0.9931 for clusters 4, DEC of 0.6128 and finally IDEC of 0.8829. The DynAE has also achieved the highest SI of 0.5090 and 0.5167 for clusters 5 and 6 respectively in a study done by Eskandarnia, Al-Ammal and Ksantinin (2021) which is consistent with this research in terms of having the highest SI score and this study has compared the DynAE with k-means, GMM, Agglomerative and CKSN algorithms. Another study by Mrabah, et al. (2020) has resulted in DynAE achieving the highest performance using NMI and ACC as an evaluation matrix which proves that the DynAE is outperforming other 21 models including k-means, DEC and IDEC even when using other performance evaluation measures. DynAE has performed better with the reduced dataset as the used data is a time series and thus sequential data and DynAE is applied to reduce the dimension of the data by selecting the important features of the data effectively. The DynAE has produced higher SI scores for the reduced data dimension used of 700 of 0.9931, 0.9787 and 0.8715 respectively than the SI score for the full data dimension of 1152

which is lower of 0.8697, 0.3144 and 0.2454 respectively, as Fig. (23) shows the DynAE comparison on the SI score for the full and reduced data dimension.

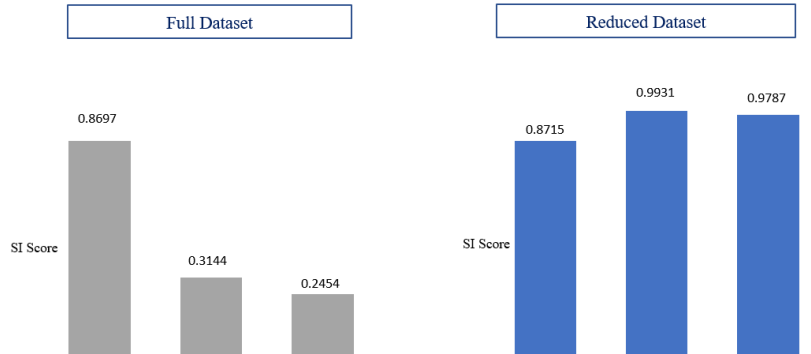


Fig (23) DynAE SI score comparison

The DEC model produced the lowest SI between the DynAE and IDEC of 0.6128 for clusters 4 and reduced dataset dimension of 700. The DEC results as the lowest performing model between the DynAE and the IDEC is inline with the study conducted by Mrabah, Khan, Ksantini and Lachiri (2020) where the 21 methods were applied including the DEC and the IDEC. The DEC performance was lower than the IDEC in most of the datasets used in terms of NMI and ACC.

The IDEC model resulted in the second highest SI of 0.8829 and has outperformed the DEC model which has achieved 0.6128. Both models were expirimented for 4 clusters and reduced data. The IDEC results are consistent with the study conducted by Eskandarnia, AL-Ammal and Ksantini (2021) in terms of having higher results in NMI and ACC rates than the DEC.

As an overall performance of all the algorithms applied which are DynAE, DEC and IDEC, the results are aligned with the studies presented in the literature review which are that the dimensionality reduction can be triggered by the temporal autoencoder and the DynAE has been outperforming other clustering models including the DEC and IDEC whether in terms of SI or other evaluation metrices such as NMI and ACC.

Chapter Five

Conclusion, Limitations and Future Work

5.0. Conclusion

5.1. Limitations and Future work

Chapter Five

Conclusion, Limitations and Future Work

5.0. Conclusion

The utility sector and the energy markets have undergone significant transformation and utilities are being forced to pursue innovation and new market-related insights due to the liberalization, increasing utility rivalry, technological advancements, and policy toward a sustainable use of resources. Utilities are increasingly investing in the study and implementation of new technologies to increase efficiency and reduce costs and to be able to track and store enormous amounts of data related to their operations thanks to technological developments in the areas of metering, communications, and computation. (Viegas, et al, 2019) The area of metering covers the smart meter analysis where energy consumption readings are provided for time intervals to the utilities and authorities and this data is converted to load profiles based on the groups formed based on consumption similarities. Many countries have established load research initiatives where the load profiles for various consumer classes are one of the key tools outlined in these projects. (Rodrigues, et al, 2003)

Applying clustering on load profiles is a challenging task because the clustering does not always capture the temporal aspects of electricity consumption (Meloy, Diakonova & Grunewald, 2019).

In this reaserch temporal autoencoder was applied first to reduce the dimensionality of the data which resulted in obtaining new data with a reduced dimension and temporal information, then the DynAE is applied at bottleneck level to efficiently perform the clustering which are the DEC and IDEC. The effectiveness of the DynAE algorithms was measured by comparing its performance to DEC and IDEC using SI. The temporal

autoencoder resulted in the reduced data mention of 700 with the lowest MSE of 0.0052. The DynAe has outperformed the other two algorithms as it yielded a score of 0.9931 compared to the DEC model of 0.6128 and the IDEC of 0.8829.

Overall, the DynAE model has outperformed the DEC and IDEC when used on Irish smart meter data and achieved great results despite the problem of feature drift and randomness. IDEC came in second place with a minor difference in the silhouette index than for the DynAE and DEC came last with the lowest Silhouette score.

5.1. Limitations and Future Work

The main challenge faced in this study was the lack of availability of real-world data for smart meter readings. Also, the available data were relatively old since 2009 and 2010 and need to be updated as the consumers behaviour might have changed over the coming years.

It is recommended to run the analysis on other real-world data from different countries for a better evaluation of the implemented method. Therefore, availability of an open source data is highly demanded.

In addition, other clustering algorithms whether classical or deep algorithms should be studied when applied to smart meter data other than the temporal, dynamic autoencoder, DEC and IDEC to explore the results that will be generated by those algorithms and compare their performance.

References

- Rodrigues, F., Jorge, J., Figueiredo, V., Vale, Z., & Cordeiro, M. (2003). A Comparative Analysis of Clustering Algorithms Applied to Load Profiling. 2.
- Aljalbout, E., Golkov, V., Siddiqui, Y., Strobel, M., & Cremers, D. (2018). Clustering with Deep Learning: Taxonomy and New Methods.
- Chen, Q., Hong, T., Wang, Y., & Kang, C. (2019). *Review of Smart Meter Data Analytics: Applications, Methodologies, and Challenges*.
- Diakonova, M., Grunewald, P., & Meloy, A. S. (2019). Cluster analysis and prediction of residential peak demand profiles using occupant activity data.
- Eskandarnia, E., Al-Ammal, H. M., & Ksantini, R. (2021). An Embedded Deep-Clustering-Based Load Profiling Framework. 33.
- Eskandarnia, E., Al-Ammal, H., Ksantini, R., & Hammad, M. (2021). Deep Learning Techniques for Smart Meter Data Analytics - A Review. 32.
- Ezugwa, A. E., Ikotun, A. M., Oyelade, O. O., Abualigah, L., Agushaka, J. O., Eke, C. I., & Akinyelu, A. A. (2022). Engineering Applications of Artificial Intelligence. *A comprehensive survey of clustering algorithms: State-of-the-art machine learning applications, taxonomy, challenges, and future research prospects*, 43.
- Guo, X., Gao, L., Liu, X., & Yin, J. (2017). Improved deep embedded clustering with local structure presentation.
- Haque, A. U., Mandal, P., Meng, J., & Pineda, R. L. (2012). Performance Evaluation of Different Optimization Algorithms for Power Demand Forecasting Applications in a Smart Grid Environment. 6.
- Khushairi, M., Abdullah, H., & Hazran, H. (n.d.). A study on the optimization of control strategy of a thermal energy storage system for building air-conditioning .

- Madiraju, N. S., Sadat, S. M., Fisher, D., & Karimabadi, H. (2018). Deep Temporal Clustering: Fully Unsupervised Learning of Time-Domain Features. 11.
- McLachlan, G., & Chang, S. (2004). Mixture modeling for cluster analysis.
- Mrabah, N., Khan, N. M., Ksantini, R., & Lachiri, Z. (2020). Neural Networks. *Deep clustering with a Dynamic Autoencoder: From reconstruction towards centroids construction*, 228.
- Mujeeb, S., Javaid, N., Javaid, J., Rafique, A., & Ilahi, M. (2018). Big Data Analytics for Load Forecasting in Smart Grids: A Survey. 10.
- Ram, A., Jalal , S., S. Jalal, A., & Kumar, M. (2010). A Density based Algorithm for Discovering Density Varied Clusters in Large Spatial Databases.
- Rodriguez, M. Z., Comin, C. H., Casanova, D., Bruno, O. M., Amancio, D. R., Rodrigues, F. A., & Costa, L. d. (2016). Clustering Algorithms: A Comparative Approach. 31.
- TAZI, K., ABBOU, F. M., & ABDI, F. (2018). Load Analysis and Consumption Profiling: An overview. 8.
- Viegas, J. L., Vieira, S. M., Sousa, J. M., Melicio, R., & Mendes , V. (n.d.). Electricity demand profile prediction based on household characteristics.
- Yilmaz, S., Chambers, J., Cozza, S., & Patel, M. K. (2019). Exploratory study on clustering methods to identify electricity use patterns in building sector. 6.
- Zhou, Y., Wu, H., Luo, Q., & Abdel-Baset, M. (2018). Automatic data clustering using nature inspired symbiotic organism search algorithm .

Appendices

Appendix A: Temporal Autoencoder Experiments Results

Table (A1) Temporal Autoencoder results

Input Dimension	1152				
Filter size	3				
Batch size	100				
Optimizer	Adam				
Epochs	10	50	100	200	300
MSE	0.1337	0.0245	0.0158	0.0129	0.0096

Table (A2) Temporal Autoencoder results

Input Dimension	500				
Filter size	7				
Batch size	500				
Optimizer	Adam				
Epochs	10	50	100	200	300
MSE	0.2033	0.0810	0.0278	0.0147	0.0083

Table (A3) Temporal Autoencoder results

Input Dimension	1000				
Filter size	7				
Batch size	1000				
Optimizer	Adam				
Epochs	10	50	100	200	300
MSE	0.2341	0.0445	0.0738	0.0336	0.0334

Table (A4) Temporal Autoencoder results

Input Dimension	200				
Filter size	7				
Batch size	500				
Optimizer	Adam				
Epochs	10	50	100	200	300
MSE	0.1900	0.0944	0.0548	0.0161	0.0071

Table (A5) Temporal Autoencoder results

Input Dimension	500				
Filter size	7				
Training sample	500				
Optimizer	Adam				
Epochs	10	50	100	200	300
MSE	0.3091	0.1348	0.0806	0.0214	0.0121

Table (A6) Temporal Autoencoder results

Input Dimension	200				
Filter size	7				
Training sample	500				
Optimizer	Adam				
Epochs	10	50	100	200	300
MSE	0.3498	0.0948	0.0635	0.0186	0.0106

Table (A7) Temporal Autoencoder results

Input Dimension	600				
Filter size	7				
Batch size	100				
Optimizer	Adam				
Epochs	10	50	100	200	300
MSE	0.2987	0.1322	0.0783	0.0209	0.0124

Appendix B: Dynamic Autoencoder Experiments Results

Table (1B) DynAE results

Clusters	4							
Gamma	0.1							
Shape	-1	10						
Input layers	1							
Output layers	1							
Epochs	10	50	100	150	200	250	300	350
MSE	0.3116	0.2062	0.2104	0.2121	0.2148	0.2205	0.2319	0.2181
Maxiter	200	500	550	600	700	1000	1500	2000
SI score	0.8705	0.8311	0.7887	0.7248	0.6981	0.6795	0.6221	0.5983

Table (2B) DynAE results

Clusters	7							
Gamma	10							
Shape	-1	13						
Input layers	2							
Output layers	2							
Epochs	10	50	100	150	200	250	300	350
MSE	0.3603	0.1975	0.2038	0.1944	0.2091	0.2031	0.1995	0.1918
Maxiter	200	500	550	600	700	1000	1500	2000
SI score	0.0372	0.0417	0.3115	0.3022	0.3144	0.2876	0.2659	0.2011

Table (3B) DynAE results

Clusters	8							
Gamma	10							
Shape	-1	15						
Input layers	2							
Output layers	3							
Epochs	10	50	100	150	200	250	300	350
MSE	0.2747	0.2022	0.1852	0.1915	0.1919	0.1961	0.1972	0.1881
Maxiter	200	800	1000	2000	3000	4000	5000	6000
SI score	0.0758	0.1711	0.2157	0.2349	0.1142	0.2321	0.2126	0.2454

In this experiment, there are SI scores in minus which are the lowest achieved in the DynAE.

Table (4B) DynAE results

Clusters	9							
Gamma	100							
Shape	-1	15						
Input layers	2							
Output layers	3							
Epochs	10	50	100	150	200	250	300	350
MSE	0.4242	0.1858	0.2218	0.2053	0.1875	0.1956	0.1883	0.1966
Maxiter	200	800	1000	2000	3000	4000	5000	6000
SI score	0.2604	0.0632	0.4455	0.4483	-0.388	-0.138	0.4139	0.0562

Table (5B) DynAE results

Clusters	2							
Gamma	1							
Shape	-1	9						
Input layers	1							
Output layers	1							
Epochs	10	50	100	150	200	250	300	300
MSE	0.322	0.3011	0.234	0.2152	0.2146	0.2367	0.2352	0.2227
Maxiter	10000	10000	10000	10000	10000	10000	10000	10000
SI score	0.521	0.5345	0.6098	0.6112	0.6543	0.6671	0.7098	0.7234

Table (5B) DynAE results

Clusters	7							
Gamma	10							
Shape	-1	13						
Input layers	2							
Output layers	2							
Epochs	10	50	100	150	200	250	300	350
MSE	0.3610	0.198	0.2235	0.2032	0.2081	0.203	0.1994	0.19
Maxiter	10000	10000	10000	10000	10000	10000	10000	10000
SI score	0.7152	0.7312	0.7358	0.7448	0.7484	0.7501	0.7557	0.7620

Table (6B) DynAE results

Clusters	8							
Gamma	10							
Shape	-1	15						
Input layers	2							
Output layers	3							
Epochs	10	50	100	150	200	250	300	350
MSE	0.2735	0.2019	0.185	0.1918	0.1925	0.1957	0.1978	0.187
Maxiter	10000	10000	10000	10000	10000	10000	10000	10000
SI score	0.7253	0.735	0.7362	0.748	0.7903	0.7789	0.789	0.7534

Table (7B) DynAE results

Clusters	9							
Gamma	100							
Shape	-1	15						
Input layers	2							
Output layers	3							
Epochs	10	50	100	150	200	250	300	350
MSE	0.4235	0.1867	0.2191	0.2201	0.1985	0.1956	0.1883	0.1978
Maxiter	100	500	1000	2000	5000	7000	10000	10000
SI score	0.7532	0.7576	0.7625	0.7685	0.7739	0.778	0.7921	0.7978

Table (8B) DynAE results

Clusters	4							
Gamma	1							
Shape	-1	8						
Input layers	1							
Output layers	1							
Batch size	32	32	32	32	32	32	32	32
Epochs	10	50	100	150	200	250	300	350
MSE	0.3509	0.1981	0.3212	0.2451	0.2536	0.2154	0.2183	0.2293
Maxiter	700	700	700	700	700	700	700	700
SI score	0.7748	0.7748	0.8014	0.8125	0.8403	0.8378	0.8334	0.8297

Table (9B) DynAE results

Clusters	4	
Gamma	1	
Shape	-1	8
Input layers	1	

Output layers	1							
Batch size	64	64	64	64	64	64	64	64
Epochs	10	50	100	150	200	250	300	350
MSE	0.3566	0.2244	0.2023	0.1915	0.1924	0.1935	0.1911	0.1918
Maxiter	700	700	700	700	700	700	1000	1000
SI score	0.8384	0.9617	0.8941	0.887	0.9118	0.9183	0.9617	0.9751

Table (10B) DynAE results

Clusters	4								
Gamma	1								
Shape	-1	9							
Input layers	1								
Output layers	1								
Batch size	256	256	256	256	250	250	250	350	350
Epochs	10	50	100	150	150	200	250	250	300
MSE	0.3582	0.2436	0.2073	0.2005	0.1786	0.1621	0.1609	0.1627	0.5876
Maxiter	700	1400	2000	2400	3000	3400	5000	5400	6000
SI score	0.8337	0.8726	0.9088	0.9342	0.9553	0.9731	0.9765	0.9787	0.9742

Appendix C: DEC results

Table (1C) DEC results

Maxiter	Epochs	SI Score	MSE
0	10	0.5717	0.0000
140	50	0.4408	1.1526
280	100	0.5469	0.0367
420	150	0.4282	0.0564
560	200	0.4515	0.0254
700	250	0.4429	0.0769
840	300	0.4419	0.1033
980	10	0.4713	0.1168
1120	50	0.4848	0.1084
1260	100	0.4713	0.1136
1400	150	0.5161	0.1134
1540	200	0.5306	0.1185
1680	250	0.5551	0.1151
1820	300	0.5548	0.1020
1960	10	0.5577	0.1070
2100	50	0.5586	0.1075
2240	100	0.5518	0.1020
2380	150	0.5619	0.1063
2520	200	0.5626	0.1005
2660	250	0.5641	0.0974
2800	300	0.5625	0.1002
2940	10	0.5605	0.0961
3080	50	0.5664	0.1004
3220	100	0.5672	0.0985
3360	150	0.5676	0.0950
3500	200	0.5665	0.0980
3640	250	0.5644	0.0939
3780	300	0.5682	0.0992
3920	10	0.5704	0.0992
4060	50	0.5717	0.0940
4200	100	0.5709	0.0966
4340	150	0.5670	0.0938
4480	200	0.5698	0.0980
4620	250	0.5716	0.0977
4760	300	0.5730	0.0948
4900	10	0.5720	0.0954
5040	50	0.5676	0.0925
5180	100	0.5735	0.0960
5320	150	0.5751	0.0961
5460	200	0.5757	0.0915

Maxiter	Epochs	SI Score	MSE
5740	300	0.5718	0.0918
5880	10	0.5770	0.0945
6020	50	0.5777	0.0941
6160	100	0.5777	0.0896
6300	150	0.5768	0.0943
6440	200	0.5733	0.0917
6580	250	0.5758	0.0963
6720	300	0.5763	0.0938
6860	10	0.5771	0.0891
7000	50	0.5777	0.0919
7140	100	0.5736	0.0896
7280	150	0.5763	0.0978
7420	200	0.5779	0.0938
7560	250	0.5784	0.0894
7700	300	0.5791	0.0920
7840	10	0.5775	0.0891
7980	50	0.5804	0.0901
8120	100	0.5808	0.0909
8260	150	0.5785	0.0891
8400	200	0.5764	0.0968
8540	250	0.5756	0.0884
8680	300	0.5810	0.0933
8820	10	0.5805	0.0915
8960	50	0.5819	0.0903
9100	100	0.5819	0.0917
9240	150	0.5787	0.0909
9380	200	0.5827	0.0898
9520	250	0.5830	0.0895
9660	300	0.5828	0.0855
9800	10	0.5847	0.0879
9940	50	0.5824	0.0861
10080	100	0.5838	0.0882
10220	150	0.5812	0.0904
10360	200	0.5810	0.0873
10500	250	0.5854	0.0878
10640	300	0.5840	0.0859
10780	10	0.5843	0.0900
10920	50	0.5851	0.0887
11060	100	0.5864	0.0854
11200	150	0.5872	0.0878

Table (2C) DEC results

Maxiter	Epochs	SI Score	MSE
11480	250	0.5872	0.0873
11620	300	0.5873	0.0875
11760	10	0.5821	0.0888
11900	50	0.5815	0.0968
12040	100	0.5833	0.0897
12180	150	0.5849	0.0904
12320	200	0.5864	0.0900
12460	250	0.5870	0.0846
12600	300	0.5879	0.0873
12740	10	0.5855	0.0863
12880	50	0.5870	0.0898
13020	100	0.5872	0.0876
13160	150	0.5857	0.0845
13300	200	0.5868	0.0872
13440	250	0.5857	0.0867
13580	300	0.5829	0.1003
13720	10	0.5875	0.0948
13860	50	0.5873	0.0852
14000	100	0.5891	0.0871
14140	150	0.5874	0.0854
14280	200	0.5903	0.0880
14420	250	0.5898	0.0868
14560	300	0.5888	0.0833
14700	10	0.5918	0.0853
14840	50	0.5897	0.0846
14980	100	0.5918	0.0892
15120	150	0.5914	0.0858
15260	200	0.5918	0.0818
15400	250	0.5924	0.0853
15540	300	0.5918	0.0847
15680	10	0.5932	0.0872
15820	50	0.5911	0.0881
15960	100	0.5925	0.0813
16100	150	0.5961	0.0843
16240	200	0.5933	0.0845
16380	250	0.5949	0.0859
16520	300	0.5922	0.0859
16660	10	0.5927	0.0830
16800	50	0.5899	0.1053
16940	100	0.5897	0.0877
17080	150	0.5932	0.0866

Maxiter	Epochs	SI Score	MSE
17220	200	0.5923	0.0872
17360	250	0.5908	0.0826
17500	300	0.5926	0.0929
17640	10	0.5933	0.0840
17780	50	0.5949	0.0851
17920	100	0.5926	0.0868
18060	150	0.5939	0.0866
18200	200	0.5974	0.0861
18340	250	0.5940	0.0838
18480	300	0.5930	0.0867
18620	10	0.5962	0.0861
18760	50	0.5961	0.0822
18900	100	0.5961	0.1012
19040	150	0.5920	0.0849
19180	200	0.5969	0.0866
19320	250	0.5970	0.0847
19460	300	0.5968	0.0829
19600	10	0.5975	0.0887
19740	50	0.5943	0.0835
19880	100	0.5980	0.0855
20020	150	0.5974	0.0894
20160	200	0.5990	0.0833
20300	250	0.5999	0.0861
20440	300	0.5971	0.0889
20580	10	0.6005	0.0870
20720	50	0.6015	0.0849
20860	100	0.6016	0.0819
21000	150	0.6019	0.0850
21140	200	0.5978	0.0830
21280	250	0.6006	0.0899
21420	300	0.6024	0.0837
21560	10	0.6032	0.0815
21700	50	0.6037	0.0843
21840	100	0.5994	0.0832
21980	150	0.5971	0.0877
22120	200	0.5945	0.0865
22260	250	0.6004	0.0838
22400	300	0.6046	0.0842
22540	10	0.6019	0.0822
22680	50	0.6056	0.0832
22820	100	0.6018	0.0876

Table (C3) DEC results

Maxiter	Epochs	SI Score	MSE
22960	150	0.6038	0.0878
23100	200	0.6034	0.0848
23240	250	0.5997	0.0817
23380	300	0.6043	0.0853
23520	10	0.6033	0.0852
23660	50	0.6040	0.0864
23800	100	0.6065	0.0836
23940	150	0.6040	0.0809
24080	200	0.6079	0.0820
24220	250	0.6079	0.0825
24360	300	0.6082	0.0798
24500	10	0.6074	0.0837
24640	50	0.6021	0.0828
24780	100	0.6065	0.0846
24920	150	0.6083	0.0832
25060	200	0.6094	0.0804
25200	250	0.6110	0.0829
25340	300	0.6074	0.0800
25480	10	0.6121	0.0813
25620	50	0.6099	0.0831
25760	100	0.6116	0.0795
25900	150	0.6125	0.0826
26040	200	0.6105	0.0800
26180	250	0.6120	0.0807
26320	300	0.6064	0.0927
26460	10	0.6100	0.0826
26600	50	0.6097	0.0838
26740	100	0.6005	0.0838
26880	150	0.6095	0.0849
27020	200	0.6097	0.0847
27160	250	0.6128	0.0787

Appendix D: Temporal Autoencoder Code (AE code)

```
from google.colab import drive
drive.mount('/content/drive')
!ls "/content/drive/My Drive"

import tensorflow as tf
tf.__version__

def load_data(data_path):
    #this part is a pre-processing of our data
    data1=pd.read_csv(data_path)
    data=data1.drop(['Unnamed: 0'],axis=1)
    x = np.concatenate((data[0:4064], data[4065:6345]))
    x1 = np.concatenate((x[0:1072], x[1073:5488], x[5489:6345]))
    X_train=x1[:, :700]
    # to add the time step beacause the auto-
encoder need three dimension (samples,feayures, time step)
    X_trainR=X_train.reshape(X_train.shape[0],X_train.shape[1],1)
    return X_train,X_trainR

import pandas as pd
import numpy as np
Data,DataR=load_data('/content/drive/MyDrive/Alllirishdata.csv')

import numpy as np
np.shape(Data)
#print(Data)

import tensorflow as tf
gpu_available = tf.test.is_gpu_available()
is_cuda_gpu_available = tf.test.is_gpu_available(cuda_only=True)
#is_cuda_gpu_min_3 = tf.test.is_gpu_available(True, (3,0))
print(is_cuda_gpu_available)

import tensorflow as tf
print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))

# Utilities
import os
import csv
import argparse
from time import time

# Keras
from keras.models import Model
from keras.layers import Dense, Reshape, UpSampling2D, Conv2DTranspose, GlobalAverageP
ooling1D, Softmax
```

```

from keras.losses import kullback_leibler_divergence
import keras.backend as K

pip install tslearn

# scikit-learn
from sklearn.cluster import AgglomerativeClustering, KMeans

# Dataset helper function

import os
os.chdir('/content/drive/MyDrive/DeepTemporalClustering-master')
# DTC components
from TSclusteringLayer import TSclusteringLayer
from TAE import temporal_autoencoder
from metrics import *
import tsdistances

class DTC:
    """
    Deep Temporal Clustering (DTC) model

    # Arguments
        n_clusters: number of clusters
        input_dim: input dimensionality
        timesteps: length of input sequences (can be None for variable length)
        n_filters: number of filters in convolutional layer
        kernel_size: size of kernel in convolutional layer
        strides: strides in convolutional layer
        pool_size: pooling size in max pooling layer, must divide the time series length

    th
        n_units: numbers of units in the two BiLSTM layers
        alpha: coefficient in Student's kernel
        dist_metric: distance metric between latent sequences
        cluster_init: cluster initialization method

    """

    def __init__(self, n_clusters, input_dim, timesteps,
                  n_filters=50, kernel_size=10, strides=1, pool_size=9, n_units=[300, 1
00],
                  alpha=1.0, dist_metric='eucl', cluster_init='kmeans', heatmap=False):

        assert(timesteps % pool_size == 0)
        self.n_clusters = n_clusters
        self.input_dim = input_dim
        self.timesteps = timesteps
        self.n_filters = n_filters

```

```

self.kernel_size = kernel_size
self.strides = strides
self.pool_size = pool_size
self.n_units = n_units
#self.latent_shape = (8, self.n_units[1])
self.latent_shape = (self.timesteps // self.pool_size, self.n_units[1])
self.alpha = alpha
self.dist_metric = dist_metric
self.cluster_init = cluster_init
self.heatmap = heatmap
self.pretrained = False
self.model = self.autoencoder = self.encoder = self.decoder = None
if self.heatmap:
    self.heatmap_model = None
    self.heatmap_loss_weight = None
    self.initial_heatmap_loss_weight = None
    self.final_heatmap_loss_weight = None
    self.finetune_heatmap_at_epoch = None

def initialize(self):
    """
    Create DTC model
    """
    # Create AE models
    self.autoencoder, self.encoder, self.decoder = temporal_autoencoder(input_dim=
self.input_dim,
                                                                    timesteps=
self.timesteps,
                                                                    n_filters=
self.n_filters,
                                                                    kernel_siz
e=self.kernel_size,
                                                                    strides=se
lf.strides,
                                                                    pool_size=
self.pool_size,
                                                                    n_units=se
lf.n_units)
    clustering_layer = TSClusteringLayer(self.n_clusters,
                                         alpha=self.alpha,
                                         dist_metric=self.dist_metric,
                                         name='TSClustering')(self.encoder.output)

    # Heatmap-generating network
    if self.heatmap:
        n_heatmap_filters = self.n_clusters # one heatmap (class activation map)
per cluster
        encoded = self.encoder.output

```

```

        heatmap_layer = Reshape((-1, 1, self.n_units[1]))(encoded)
        heatmap_layer = UpSampling2D((self.pool_size, 1))(heatmap_layer)
        heatmap_layer = Conv2DTranspose(n_heatmap_filters, (self.kernel_size, 1),
padding='same')(heatmap_layer)
        # The next one is the heatmap layer we will visualize
        heatmap_layer = Reshape((-
1, n_heatmap_filters), name='Heatmap')(heatmap_layer)
        heatmap_output_layer = GlobalAveragePooling1D()(heatmap_layer)
        # A dense layer must be added only if `n_heatmap_filters` is different fro
m `n_clusters`
        # heatmap_output_layer = Dense(self.n_clusters, activation='relu')(heatmap
_output_layer)
        heatmap_output_layer = Softmax()(heatmap_output_layer) # normalize activa
tions with softmax

    if self.heatmap:

        # Create DTC model
        self.model = Model(inputs=self.autoencoder.input,
                           outputs=[self.autoencoder.output, clustering_layer, hea
tmap_output_layer])
        # Create Heatmap model
        self.heatmap_model = Model(inputs=self.autoencoder.input,
                                   outputs=heatmap_layer)
    else:
        # Create DTC model
        self.model = Model(inputs=self.autoencoder.input,
                           outputs=[self.autoencoder.output, clustering_layer])

@property
def cluster_centers_(self):
    """
    Returns cluster centers
    """
    return self.model.get_layer(name='TSClustering').get_weights()[0]

@staticmethod
def weighted_kld(loss_weight):
    """
    Custom KL-divergence loss with a variable weight parameter
    """
    def loss(y_true, y_pred):
        return loss_weight * kullback leibler divergence(y true, y pred)
    return loss

def on_epoch_end(self, epoch):
    """
    Update heatmap loss weight on epoch end

```

```

"""
if epoch > self.finetune_heatmap_at_epoch:
    K.set_value(self.heatmap_loss_weight, self.final_heatmap_loss_weight)

def compile(self, gamma, optimizer, initial_heatmap_loss_weight=None, final_heatmap_loss_weight=None):
    """
    Compile DTC model

    # Arguments
        gamma: coefficient of TS clustering loss
        optimizer: optimization algorithm
        initial_heatmap_loss_weight (optional): initial weight of heatmap loss vs clustering loss
        final_heatmap_loss_weight (optional): final weight of heatmap loss vs clustering loss (heatmap finetuning)
    """
    if self.heatmap:
        self.initial_heatmap_loss_weight = initial_heatmap_loss_weight
        self.final_heatmap_loss_weight = final_heatmap_loss_weight
        self.heatmap_loss_weight = K.variable(self.initial_heatmap_loss_weight)
        self.model.compile(loss=['mse', DTC.weighted_kld(1.0 - self.heatmap_loss_weight), DTC.weighted_kld(self.heatmap_loss_weight)],
                           loss_weights=[1.0, gamma, gamma],
                           optimizer=optimizer)
    else:
        self.model.compile(loss=['mse', 'kld'],
                           loss_weights=[1.0, gamma],
                           optimizer=optimizer)

def load_weights(self, weights_path):
    """
    Load pre-trained weights of DTC model

    # Arguments
        weight_path: path to weights file (.h5)
    """
    self.model.load_weights(weights_path)
    self.pretrained = True

def load_ae_weights(self, ae_weights_path):
    """
    Load pre-trained weights of AE

    # Arguments
        ae_weight_path: path to weights file (.h5)
    """
    self.autoencoder.load_weights(ae_weights_path)

```

```

        self.pretrained = True

    def dist(self, x1, x2):
        """
        Compute distance between two multivariate time series using chosen distance me
        tric

        # Arguments
            x1: first input (np array)
            x2: second input (np array)
        # Return
            distance
        """
        if self.dist_metric == 'eucl':
            return tsdistances.eucl(x1, x2)
        elif self.dist_metric == 'cid':
            return tsdistances.cid(x1, x2)
        elif self.dist_metric == 'cor':
            return tsdistances.cor(x1, x2)
        elif self.dist_metric == 'acf':
            return tsdistances.acf(x1, x2)
        else:
            raise ValueError('Available distances are eucl, cid, cor and acf!')

    def init_cluster_weights(self, X):
        """
        Initialize with complete-linkage hierarchical clustering or k-means.

        # Arguments
            X: numpy array containing training set or batch
        """
        assert(self.cluster_init in ['hierarchical', 'kmeans'])
        print('Initializing cluster...')

        features = self.encode(X)

        if self.cluster_init == 'hierarchical':
            if self.dist_metric == 'eucl': # use AgglomerativeClustering off-the-
shelf
                hc = AgglomerativeClustering(n_clusters=self.n_clusters,
                                              affinity='euclidean',
                                              linkage='complete').fit(features.reshape(
features.shape[0], -1))
            else: # compute distance matrix using dist
                d = np.zeros((features.shape[0], features.shape[0]))
                for i in range(features.shape[0]):
                    for j in range(i):
                        d[i, j] = d[j, i] = self.dist(features[i], features[j])

```

```

        hc = AgglomerativeClustering(n_clusters=self.n_clusters,
                                     affinity='precomputed',
                                     linkage='complete').fit(d)

        # compute centroid
        cluster_centers = np.array([features[hc.labels_ == c].mean(axis=0) for c in
                                     range(self.n_clusters)])

    elif self.cluster_init == 'kmeans':
        # fit k-means on flattened features
        km = KMeans(n_clusters=self.n_clusters, n_init=10).fit(features.reshape(fe
atures.shape[0], -1))
        cluster_centers = km.cluster_centers_.reshape(self.n_clusters, features.sh
ape[1], features.shape[2])

    self.model.get_layer(name='TSClustering').set_weights([cluster_centers])
    print('Done!')

def encode(self, x):
    """
    Encoding function. Extract latent features from hidden layer

    # Arguments
        x: data point
    # Return
        encoded (latent) data point
    """
    return self.encoder.predict(x)

def decode(self, x):
    """
    Decoding function. Decodes encoded sequence from latent space.

    # Arguments
        x: encoded (latent) data point
    # Return
        decoded data point
    """
    return self.decoder.predict(x)

def predict(self, x):
    """
    Predict cluster assignment.

    """
    q = self.model.predict(x, verbose=0)[1]
    return q.argmax(axis=1)

@staticmethod

```

```

def target_distribution(q): # target distribution p which enhances the discrimina
tion of soft label q
    weight = q ** 2 / q.sum(0)
    return (weight.T / weight.sum(1)).T

def predict_heatmap(self, x):
    """
    Produces TS clustering heatmap from input sequence.

    # Arguments
        x: data point
    # Return
        heatmap
    """
    return self.heatmap_model.predict(x, verbose=0)

def pretrain(self, X,
              optimizer='SGD',
              epochs=10,
              batch_size=64,
              save_dir='results/tmp',
              verbose=1):
    """
    Pre-train the autoencoder using only MSE reconstruction loss
    Saves weights in h5 format.

    # Arguments
        X: training set
        optimizer: optimization algorithm
        epochs: number of pre-training epochs
        batch_size: training batch size
        save_dir: path to existing directory where weights will be saved
    """
    print('Pretraining...')
    self.autoencoder.compile(optimizer=optimizer, loss='mse')

    # Begin pretraining
    t0 = time()
    self.autoencoder.fit(X, X, batch_size=batch_size, epochs=epochs, verbose=verbo
se)

    print('Pretraining time: ', time() - t0)
    self.autoencoder.save_weights('{}ae_weights-
epoch{}.h5'.format(save_dir, epochs))
    print('Pretrained weights are saved to {}ae_weights-
epoch{}.h5'.format(save_dir, epochs))
    self.pretrained = True

def fit(self, X_train, y_train=None,

```

```

        X_val=None, y_val=None,
        epochs=10,
        eval_epochs=100,
        save_epochs=10,
        batch_size=64,
        tol=0.001,
        patience=5,
        finetune_heatmap_at_epoch=8,
        save_dir='results/tmp'):
    """
    Training procedure

    # Arguments
        X_train: training set
        y_train: (optional) training labels
        X_val: (optional) validation set
        y_val: (optional) validation labels
        epochs: number of training epochs
        eval_epochs: evaluate metrics on train/val set every eval_epochs epochs
        save_epochs: save model weights every save_epochs epochs
        batch_size: training batch size
        tol: tolerance for stopping criterion
        patience: patience for stopping criterion
        finetune_heatmap_at_epoch: epoch number where heatmap finetuning will start
    . Heatmap loss weight will
        switch from `self.initial_heatmap_loss_weight` to
    o `self.final_heatmap_loss_weight`
        save_dir: path to existing directory where weights and logs are saved
    """
    if not self.pretrained:
        print('Autoencoder was not pre-trained!')

    if self.heatmap:
        self.finetune_heatmap_at_epoch = finetune_heatmap_at_epoch

    # Logging file
    logfile = open(save_dir + '/dte_log.csv', 'w')
    fieldnames = ['epoch', 'T', 'L', 'Lr', 'Lc']
    if X_val is not None:
        fieldnames += ['L_val', 'Lr_val', 'Lc_val']
    if y_train is not None:
        fieldnames += ['acc', 'pur', 'nmi', 'ari']
    if y_val is not None:
        fieldnames += ['acc_val', 'pur_val', 'nmi_val', 'ari_val']
    logwriter = csv.DictWriter(logfile, fieldnames)
    logwriter.writeheader()

    y_pred_last = None

```

```

patience_cnt = 0

print('Training for {} epochs.\nEvaluating every {} and saving model every {}
epochs.'.format(epochs, eval_epochs, save_epochs))

for epoch in range(epochs):

    # Compute cluster assignments for training set
    q = self.model.predict(X_train)[1]
    p = DTC.target_distribution(q)

    # Evaluate losses and metrics on training set
    if epoch % eval_epochs == 0:

        # Initialize log dictionary
        logdict = dict(epoch=epoch)

        y_pred = q.argmax(axis=1)
        if X_val is not None:
            q_val = self.model.predict(X_val)[1]
            p_val = DTC.target_distribution(q_val)
            y_val_pred = q_val.argmax(axis=1)

        print('epoch {}'.format(epoch))
        if self.heatmap:
            loss = self.model.evaluate(X_train, [X_train, p, p], batch_size=batch_size, verbose=False)
        else:
            loss = self.model.evaluate(X_train, [X_train, p], batch_size=batch_size, verbose=False)

        logdict['L'] = loss[0]
        logdict['Lr'] = loss[1]
        logdict['Lc'] = loss[2]
        print('[Train] - Lr={:f}, Lc={:f} - total loss={:f}'.format(logdict['Lr'], logdict['Lc'], logdict['L']))
        if X_val is not None:
            val_loss = self.model.evaluate(X_val, [X_val, p_val], batch_size=batch_size, verbose=False)
            logdict['L_val'] = val_loss[0]
            logdict['Lr_val'] = val_loss[1]
            logdict['Lc_val'] = val_loss[2]
            print('[Val] - Lr={:f}, Lc={:f} - total loss={:f}'.format(logdict['Lr_val'], logdict['Lc_val'], logdict['L_val']))

    # Evaluate the clustering performance using labels
    if y_train is not None:
        logdict['acc'] = cluster_acc(y_train, y_pred)
        logdict['pur'] = cluster_purity(y_train, y_pred)

```

```

        logdict['nmi'] = metrics.normalized_mutual_info_score(y_train, y_p
red)

        logdict['ari'] = metrics.adjusted_rand_score(y_train, y_pred)
        print('[Train] - Acc={:f}, Pur={:f}, NMI={:f}, ARI={:f}'.format(lo
gdict['acc'], logdict['pur'],

                                                                 lo
gdict['nmi'], logdict['ari']))
        if y_val is not None:
            logdict['acc_val'] = cluster_acc(y_val, y_val_pred)
            logdict['pur_val'] = cluster_purity(y_val, y_val_pred)
            logdict['nmi_val'] = metrics.normalized_mutual_info_score(y_val, y
_val_pred)

            logdict['ari_val'] = metrics.adjusted_rand_score(y_val, y_val_pred
)

            print('[Val] - Acc={:f}, Pur={:f}, NMI={:f}, ARI={:f}'.format(logd
ict['acc_val'], logdict['pur_val'],

                                                                 logd
ict['nmi_val'], logdict['ari_val']))

        logwriter.writerow(logdict)

        # check stop criterion
        if y_pred_last is not None:
            assignment_changes = np.sum(y_pred != y_pred_last).astype(np.float
32) / y_pred.shape[0]
            y_pred_last = y_pred
            if epoch > 0 and assignment_changes < tol:
                patience_cnt += 1
                print('Assignment changes {} < {} tolerance threshold. Patience: {
}/{}}'.format(assignment_changes, tol, patience_cnt, patience))
                if patience_cnt >= patience:
                    print('Reached max patience. Stopping training.')
                    logfile.close()
                    break
            else:
                patience_cnt = 0

        # Save intermediate model and plots
        if epoch % save_epochs == 0:
            self.model.save_weights(save_dir + '/DTC_model_' + str(epoch) + '.h5')
            print('Saved model to:', save_dir + '/DTC_model_' + str(epoch) + '.h5'
)

        # Train for one epoch
        if self.heatmap:
            self.model.fit(X_train, [X_train, p, p], epochs=1, batch_size=batch_si
ze, verbose=False)
            self.on_epoch_end(epoch)

```

```

        else:
            self.model.fit(X_train, [X_train, p], epochs=1, batch_size=batch_size,
                           verbose=False)

            # Save the final model
            logfile.close()
            print('Saving model to:', save_dir + '/DTC_model_final.h5')
            self.model.save_weights(save_dir + '/DTC_model_final.h5')

X_train=DataR
np.shape(X_train)

X_train.shape[1]

pretrain_optimizer = 'Adam'
optimizer = 'Adam'

n_clusters=4

dtc = DTC(n_clusters=n_clusters,
          input_dim=X_train.shape[-1],
          timesteps=X_train.shape[1],
          n_filters=7,
          kernel_size=10,
          strides=1,
          pool_size=4,
          n_units=[300,100],
          alpha=1.0,
          dist_metric='eucl',
          cluster_init='kmeans',
          heatmap=None)

optimizer = 'Adam'

dtc.initialize()

dtc.compile(gamma=1.0, optimizer=optimizer, initial_heatmap_loss_weight=1.0,
            final_heatmap_loss_weight=0.9)

dtc.pretrain(X=X_train, optimizer='SGD',
             epochs=10, batch_size=2000,
             save_dir='/content/drive/MyDrive/Work.Esra/Work.Esra/DeepTemporal
Clustering-master/results/tmp')

```

Appendix D: Dynamic Autoencoder Code (DynAE code)

```
pip uninstall h5py

pip install h5py==2.10.0

from yellowbrick.cluster import SilhouetteVisualizer
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples, silhouette_score
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

data = pd.read_csv("/content/drive/MyDrive/Allirishdata.csv");
data.info();
X = data;
range_n_clusters = [2, 3, 4, 5, 6, 7, 8, 9, 10];
for n_clusters in range_n_clusters:
    #Initialize the KMeans clusterer n_clusters value and a random generator seed of 1
    0 for reproducibility.
    clusterer = KMeans(n_clusters=n_clusters, random_state=42)
    cluster_labels = clusterer.fit_predict(X)
    sample_labels = pd.DataFrame(cluster_labels)

silhouette_avg = silhouette_score(X, cluster_labels)
    print("For n_clusters =", n_clusters, "The average silhouette_score is: ", silhoue
tte_avg)

sample_silhouette_values = silhouette_samples(X, cluster_labels)
    sample_silhouette_scores = pd.DataFrame(sample_silhouette_values)
for i in range(n_clusters):
    ith_cluster_silhouette_values = sample_silhouette_values[cluster_labels == i]
    ith_cluster_silhouette_values.sort()
    size_cluster_i = ith_cluster_silhouette_values.shape[0]
    print ("For clusters = ", i, "The number of samples is: ", size_cluster_i)
samples_results = pd.concat([sample_labels,sample_silhouette_scores],axis=1)

    samples_results.columns = ['Labels','Silhouette Score']
file_path = '/content/drive/MyDrive/Allirishdata.csv_'+ str(n_clusters) +'.csv'
    print (file_path)
    samples_results.to_csv(file_path)

visualizer = SilhouetteVisualizer(clusterer, colors='yellowbrick')
visualizer.fit(X)          # Fit the data to the visualizer
visualizer.show()          # Finalize and render the figure

#2D visualization of clusters
```

```

pca = PCA(2)
df = pca.fit_transform(X)
for i in range(n_clusters):
plt.scatter(df[cluster_labels == i, 0] , df[cluster_labels == i, 1])
plt.legend()
plt.show()

import tensorflow as tf
gpu_available = tf.test.is_gpu_available()
is_cuda_gpu_available = tf.test.is_gpu_available(cuda_only=True)
#is_cuda_gpu_min_3 = tf.test.is_gpu_available(True, (3,0))
print(is_cuda_gpu_available)

#Import the required libraries and modules to execute the code

#Import TensorFlow version 1.x
try:
    %tensorflow_version 1.x
except Exception:
    pass

import sys
#Insert list of directories that the interpreter will search in for the required modules
#(DynAE.py and datasets.py) and the TON_IoT subsets
sys.path.insert(0, '/content/drive/MyDrive/DynAE/DynAE_Copy')
sys.path.insert(0, '/content/drive/MyDrive/Dataset DynAe/data')

#TensorFlow is a library for training and inference of deep neural networks
import tensorflow as tf
#Import optimizers that implement both Adam and SGD (Gradient Descent (with momentum) optimizer)
from tensorflow.keras.optimizers import SGD, Adam

#os is a library for handling directories
import os

#NumPy is a library for adding support for large, multi-dimensional arrays and matrices,
#along with a large collection of high-level mathematical functions to operate on these arrays and matrices
import numpy as np

#time is a library for time-related functions
from time import time

#Import an instance from DynAE class in the DynAE.py module
#from DynAE import DynAE

```

```

from DynAEel import DynAE

#Import load_data and load_data_conv functions from datasets.py module to load the TON
_IoT dataset
#from datasets import load_data
from datasets import load_data, load_data_conv

#Import the metrics.py module
import metrics

#Import the StandardScaler library to standardize features by removing the mean and sc
aling to unit variance
#The standard score of a sample x is calculated as:  $z = (x - \mu(\text{mean})) / \sigma(\text{standard deviation})$ 
from sklearn.preprocessing import StandardScaler as ss

#Assign the required variables and hyperparameters

dataset='irish'
data_path='/content/drive/MyDrive/Dataset DynAe/data/' + dataset

#Save weights and logs for both pre-training and clustering phases
save_dir='/content/drive/My Drive/DynAE_Copy/results'

#Save visualizations for both pre-training and clustering phases
visualisation_dir='/content/drive/My Drive/DynAE_Copy/visualisation'


#Number of clusters in the dataset
n_clusters=4

#Hyperparameters
#advweight parameter for aci_ae_constructor function (refer to Eq(6) in article)
#Used when creating DynAE model
loss_weight_lambda=0.5

#Batch size for the mini-batch gradient descent
#Used for both pre-training and clustering phases
#batch_size=256
batch_size=350

#Maximum number of iterations the pre-training phase
#maxiter_pretraining=130e3
#maxiter_pretraining=50

#Maximum number of iterations for autoencoder to reach convergence in clustering phase
#maxiter_clustering=1e5
#maxiter_clustering=100

```

```

#Convergence threshold for the clustering phase
#tol=0.01
tol=0.001

optimizer0=SGD(1.0, 0.9)

#SGD optimizer to update learning parameters of clustering phase
optimizer1=SGD(0.001, 0.9)

#Adam optimizer to update learning parameters of pre-trainig phase
#optimizer2=tf.train.AdamOptimizer(0.0001)
optimizer2=tf.keras.optimizers.Adam(0.001)

#Confidence threshold for the clustering phase (refer to Eq(10) in article)
kappa=3

#Balancing hyperparameter for dynAE_constructor function
gamma=1

#Data augmentation parameters for image datasets only
ws=0.1
hs=0.1
rot=10
scale=0.

x, y = load_data(dataset, data_path)
x1 = x
sc = ss()
x = sc.fit_transform(x)
dims=[x.shape[-1], 14]

#model = DynAE(batch_size=batch_size, dataset=dataset, dims=[x.shape[-
1], 500, 500, 2000, 10], loss_weight=loss_weight_lambda, gamma=gamma, n_clusters=n_clu
sters, visualisation_dir=visualisation_dir, ws=ws, hs=hs, rot=rot, scale=scale)
model = DynAE(batch_size=batch_size, dataset=dataset, dims=dims, loss_weight=loss_weig
ht_lambda, gamma=gamma, n_clusters=n_clusters, visualisation_dir=visualisation_dir, ws
=ws, hs=hs, rot=rot, scale=scale)
#Configure model for training using SGD optimizer
model.compile_dynAE(optimizer=optimizer1)
model.compile_disc(optimizer=optimizer2)
model.compile_aci_ae(optimizer=optimizer2)
pretrain_optimizer = Adam(0.001)
model.train_ae(x, None, optimizer=pretrain_optimizer, epochs=250, batch_size=batch_siz
e, save_dir=save_dir, aug_train=False)

print (x.shape[0], x.shape[1])
print(np.arange(x.shape[0]))

```

```

#Retrieve pre-training weights after training autoencoder
import h5py

#Open h5 file for reading
f = h5py.File('/content/drive/MyDrive/DynAE_Copy/results/irish/pretrain/ae_weights.h5'
, 'r')
list(f.keys())

#Retrieve weights for each layer
input_autencoder_data = f['input_autencoder']
print (input_autencoder_data)

#Encoder
encoder_data_1 = f['encoder/encoder_0/kernel:0']
print (encoder_data_1)
print ('Weights of encoder/encoder_0/kernel:0 dataset: ', encoder_data_1[:, '\n'])
encoder_data_2 = f['encoder/encoder_0/bias:0']
print (encoder_data_2)
print ('Weights of encoder/encoder_0/bias:0 dataset:', encoder_data_2[:, '\n'])

#Decoder
decoder_data_1 = f['decoder/decoder_0/kernel:0']
print (decoder_data_1)
print ('Weights of decoder/decoder_0/kernel:0 dataset:', decoder_data_1[:, '\n'])
decoder_data_2 = f['decoder/decoder_0/bias:0']
print (decoder_data_2)
print ('Weights of decoder/decoder_0/bias:0 dataset:', decoder_data_2[:, '\n'])

#Create DynAE model and cluster
maxiter_clustering=1000

model = DynAE(batch_size=batch_size, dataset=dataset, dims=dims, loss_weight=loss_weight_lambda, gamma=gamma, n_clusters=n_clusters, visualisation_dir=visualisation_dir, ws=ws, hs=hs, rot=rot, scale=scale)
model.compile_dynAE(optimizer=optimizer1)

#Load weights of the pre-trained autoencoder model
model.ae.load_weights(save_dir + '/' + dataset + '/pretrain/ae_weights.h5')

y1 = model.train_dynAE(x=x, y=None, kappa=kappa, n_clusters=n_clusters, maxiter=maxiter_clustering, batch_size=batch_size, tol=tol, validate_interval=5, show_interval=None, save_interval=maxiter_clustering, save_dir=save_dir, aug_train=False)
#Load the pretraining weights if you have already pretrained your network
#model.ae.load_weights(save_dir + '/' + dataset + '/pretrain/ae_weights.h5')
#model.critic.load_weights(save_dir + '/' + dataset + '/pretrain/critic_weights.h5')

```

Appendix E: Deep Embedded Clustering Code (DEC code)

```
import os
os.chdir('/content/drive/MyDrive/DEC/DEC_Sarra')
# Install TensorFlow
try:
    %tensorflow_version 1.x
except Exception:
    pass
import sys
sys.path.insert(0, '/content/drive/MyDrive/DEC/DEC_Sarra')
sys.path.insert(0, '/content/drive/MyDrive/DatasetDec/DatasetDec/data')
import tensorflow as tf
from tensorflow.keras.optimizers import SGD
# from tensorflow.keras.optimizers.legacy import Adam
from tensorflow.keras.optimizers import Adam
import os
import numpy as np
import time
from FcDEC import FcDEC, show_double
from datasets import load_data, load_data_conv
import metrics

class Args():
    def __init__(self,
                  method='FcDEC',
                  dataset='irish',
                  save_dir= '/content/drive/My Drive/DEC/DEC_Sarra/results2',
                  visualisation_dir= '/content/drive/My Drive/DEC/DEC Sarra/visualisation
',
                  aug_pretrain=True,
                  pretrained_weights=None,
                  pretrain_epochs=500,
                  verbose=1,
                  testing=False,
                  weights=None,
                  aug_cluster=False,
                  optimizer='adam',
                  lr=0.0001,
                  pretrain_lr=1.0,
                  batch_size=256,
                  maxiter_dec=100000,
                  maxiter_decoder_v1 = 3e5,
                  maxiter_invertor = 3e5,
                  update_interval=140,
                  tol=0.001):
        self.method = method
        self.dataset = dataset
```

```

self.save_dir = save_dir
self.visualisation_dir = visualisation_dir
self.aug_pretrain = aug_pretrain
self.pretrained_weights = pretrained_weights
self.pretrain_epochs = pretrain_epochs
self.verbose = verbose
self.testing = testing
self.weights = weights
self.aug_cluster = aug_cluster
self.optimizer = optimizer
self.lr = lr
self.pretrain_lr = pretrain_lr
self.batch_size = batch_size
self.maxiter_dec = maxiter_dec
self.update_interval = update_interval
self.tol = tol
self.data_path = '/content/drive/MyDrive/DatasetDec/DatasetDec/data/' + dataset
self.maxiter_decoder_v1 = maxiter_decoder_v1
self.maxiter_invertor = maxiter_invertor

args = Args()

x, y = load_data(args.dataset, args.data_path)
n_clusters = 4 #len(np.unique(y))
model = FcDEC(dataset=args.dataset, dims=[x.shape[-1], 500, 500, 2000, n_clusters], n_clusters=n_clusters)
model.compile_dec(optimizer=SGD(args.lr, 0.9), loss='kld')
model.compile_dec(optimizer=Adam(), loss='kld')

pretrain_optimizer = SGD(1., 0.9)
start = time.time()
pretrain_optimizer = Adam(0.001)
model.train_ae(x, y, optimizer=pretrain_optimizer, epochs=args.pretrain_epochs, batch_size=args.batch_size, save_dir=args.save_dir, verbose=args.verbose, aug_pretrain=args.aug_pretrain)
exec_time = time.time() - start
print(exec_time)

model.autoencoder.load_weights(args.save_dir + '/' + args.dataset + "/pretrain/ae_weights.h5")

y_pred = model.train_dec(x, y, maxiter=args.maxiter_dec, batch_size=args.batch_size, tol=args.tol, update_interval=args.update_interval, save_dir=args.save_dir, aug_cluster=args.aug_cluster)

```

Appendix F: Improved Deep Embedded Clustering Code (IDEC code)

```
# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib.pyplot as mp
import seaborn as sb
import scipy.stats as scs
df=pd.read_csv("/content/Allirishdata.csv")
df.head()

df.isnull().sum()
df1=df.drop(['S1_No'],axis=1)
df1

# Standardizing the data

#from sklearn.preprocessing import StandardScaler
#df_scaled=StandardScaler().fit_transform(df1)
#df=pd.DataFrame(df_scaled,columns=df1.columns)

#Normalized Data set:
df_min_max_scaled = df1.copy()

# apply normalization techniques
for column in df_min_max_scaled.columns:
    df_min_max_scaled[column] = (df_min_max_scaled[column] - df_min_max_scaled[column].min()) / (df_min_max_scaled[column].max() - df_min_max_scaled[column].min())

# view normalized data
print(df_min_max_scaled)

df = df_min_max_scaled
df

# Importing PCA
from sklearn.decomposition import PCA

pca = PCA()
df_pca = pca.fit_transform(df)

#plot a Scree plot of the Principal Components
plt.figure(figsize=(16,11))
plt.plot(np.cumsum(pca.explained_variance_ratio_), 'ro-')
```

```

plt.grid()

#from the graph, first 17 components describe the maximum variance(more than 90% of the data). We shall use them for our subsequent analysis.
new_pca = PCA(n_components=180)
new_pca

df2 = new_pca.fit_transform(df)

from sklearn.cluster import KMeans
SSE=[]
for i in range(1,11):
    Kmeans=KMeans(n_clusters=i,init='k-means++',random_state=42)
    Kmeans.fit(df2)
    SSE.append(Kmeans.inertia_)
plt.plot(range(1,11),SSE)
plt.xlabel("No. of Clusters")
plt.ylabel("Sum of Squared Errors")
plt.show()

kmeans_2=KMeans(n_clusters=2,init='k-means++',random_state=42).fit(df2)
kmeans_3=KMeans(n_clusters=3,init='k-means++',random_state=42).fit(df2)
kmeans_4=KMeans(n_clusters=4,init='k-means++',random_state=42).fit(df2)
kmeans_5=KMeans(n_clusters=5,init='k-means++',random_state=42).fit(df2)
kmeans_6=KMeans(n_clusters=6,init='k-means++',random_state=42).fit(df2)
kmeans_7=KMeans(n_clusters=7,init='k-means++',random_state=42).fit(df2)
kmeans_8=KMeans(n_clusters=8,init='k-means++',random_state=42).fit(df2)
kmeans_9=KMeans(n_clusters=9,init='k-means++',random_state=42).fit(df2)
kmeans_10=KMeans(n_clusters=10,init='k-means++',random_state=42).fit(df2)

y_kmeans_2=kmeans_2.fit_predict(df2)
y_kmeans_3=kmeans_3.fit_predict(df2)
y_kmeans_4=kmeans_4.fit_predict(df2)
y_kmeans_5=kmeans_5.fit_predict(df2)
y_kmeans_6=kmeans_6.fit_predict(df2)
y_kmeans_7=kmeans_7.fit_predict(df2)
y_kmeans_8=kmeans_8.fit_predict(df2)
y_kmeans_9=kmeans_9.fit_predict(df2)
y_kmeans_10=kmeans_10.fit_predict(df2)

from sklearn.metrics import silhouette_score

## clusters? .Let's decide it using Silhouette index
print(f'Silhouette Score(n=2): {silhouette_score(df2, y_kmeans_2)}')
print(f'Silhouette Score(n=3): {silhouette_score(df2, y_kmeans_3)}')
print(f'Silhouette Score(n=4): {silhouette_score(df2, y_kmeans_4)}')
print(f'Silhouette Score(n=5): {silhouette_score(df2, y_kmeans_5)}')

```

```

print(f'Silhouette Score(n=6): {silhouette_score(df2, y_kmeans_6)}')
print(f'Silhouette Score(n=7): {silhouette_score(df2, y_kmeans_7)}')
print(f'Silhouette Score(n=8): {silhouette_score(df2, y_kmeans_8)}')
print(f'Silhouette Score(n=9): {silhouette_score(df2, y_kmeans_9)}')
print(f'Silhouette Score(n=10): {silhouette_score(df2, y_kmeans_10)}')

kmeans=KMeans(n_clusters=2).fit(df2)
labels=kmeans.labels_
labels

unique, counts = np.unique(labels, return_counts=True)
print(np.asarray((unique, counts)).T)

pip install scikit-learn-extra

# To decide the no of clusters, lets again use Silhouette score method

from sklearn.mixture import GaussianMixture

# Model fitting with 2 components
gmm_2 = GaussianMixture(n_components=2)
gmm_2.fit(df2)
y_gmm_2=gmm_2.predict(df2)

#Model fitting with 3 components
gmm_3 = GaussianMixture(n_components=3)
gmm_3.fit(df2)
y_gmm_3=gmm_3.predict(df2)

print(f'Silhouette Score("n_clusters=2"): {silhouette_score(df2, y_gmm_2)}')
print(f'Silhouette Score("n_clusters=3"): {silhouette_score(df2, y_gmm_3)}')
print(f'Silhouette Score("n_clusters=4"): {silhouette_score(df2, y_gmm_4)}')
print(f'Silhouette Score("n_clusters=5"): {silhouette_score(df2, y_gmm_5)}')
print(f'Silhouette Score("n_clusters=6"): {silhouette_score(df2, y_gmm_6)}')
print(f'Silhouette Score("n_clusters=7"): {silhouette_score(df2, y_gmm_7)}')
print(f'Silhouette Score("n_clusters=8"): {silhouette_score(df2, y_gmm_8)}')
print(f'Silhouette Score("n_clusters=9"): {silhouette_score(df2, y_gmm_9)}')
print(f'Silhouette Score("n_clusters=10"): {silhouette_score(df2, y_gmm_10)}')

#Hierarchical Clustering uses Dendrogram to decide the no. of clusters

import scipy.cluster.hierarchy as shc
plt.figure(figsize=(10,10))
plt.title('Dendrogram')
dend=shc.dendrogram(shc.linkage(df2,method='ward'))

# We have the longest vertical line in the range around 15-
25,so we set the threshold to be 20

```

```

plt.axhline(y=20,color='red',linestyle='--')

from sklearn.cluster import AgglomerativeClustering
cluster= AgglomerativeClustering(n_clusters=3,affinity='euclidean',linkage='ward')
y_hr=cluster.fit_predict(df2)
plt.scatter(df2[:,0],df2[:,1],s=100,c=cluster.labels_)

from sklearn.cluster import AgglomerativeClustering
cluster= AgglomerativeClustering(n_clusters=2,affinity='euclidean',linkage='ward')
y_hr1=cluster.fit_predict(df2)
plt.scatter(df2[:,0],df2[:,1],s=100,c=cluster.labels_)

print(f'Silhouette Score("n_clusters=2"): {silhouette_score(df2, y_hr1)}')
print(f'Silhouette Score("n_clusters=3"): {silhouette_score(df2, y_hr)}')
print(f'Silhouette Score("n_clusters=4"): {silhouette_score(df2, y_hr1)}')
print(f'Silhouette Score("n_clusters=5"): {silhouette_score(df2, y_hr)}')
print(f'Silhouette Score("n_clusters=6"): {silhouette_score(df2, y_hr1)}')
print(f'Silhouette Score("n_clusters=7"): {silhouette_score(df2, y_hr)}')
print(f'Silhouette Score("n_clusters=8"): {silhouette_score(df2, y_hr1)}')
print(f'Silhouette Score("n_clusters=9"): {silhouette_score(df2, y_hr)}')
print(f'Silhouette Score("n_clusters=10"): {silhouette_score(df2, y_hr)}')

from scipy.spatial import distance
rbf_param =1
# K is a Adjacency Matrix(Square Matrix)
K= np.exp(-rbf_param * distance.cdist(df2,df2,metric='sqeuclidean'))
K.shape

#D is a diagonal matrix whose diagonal elements are the sum of respective rows of matr
ix K
D= K.sum(axis=1)
D= np.sqrt(1/D)
M= np.multiply(D[np.newaxis,:],np.multiply(K,D[:,np.newaxis]))
M.shape

from scipy import linalg
from sklearn.preprocessing import normalize
from sklearn.cluster import KMeans

U,sigma,_ = linalg.svd(M,full_matrices=False,lapack_driver='gesvd')
# first-three eigen vectors are used to form clusters
U_subset=U[:,0:3]
#This requires U_subset to be normalized
y_sc_1=KMeans(n_clusters=3).fit_predict(normalize(U_subset))
y_sc_1

plt.scatter(df2[:,0],df2[:,1],c=y_sc_1,s=100)

```

```

plt.show()

#If we don't normalize the U_subset
U,sigma,_ = linalg.svd(M,full_matrices=False,lapack_driver='gesvd')
# first-three eigen vectors are used to form clusters
U_subset=U[:,0:3]
y_sc_2=KMeans(n_clusters=3).fit_predict((U_subset))
y_sc_2

plt.scatter(df2[:,0],df2[:,1],c=y_sc_2,s=100)
plt.show()

print(f'Silhouette Score("Normalized"): {silhouette_score(df2, y_sc_1)}')
print(f'Silhouette Score("Non-Normalized"): {silhouette_score(df2, y_sc_2)}')

```

Appendix G: Libraries that were used in DynAE, DEC and IDEC codes

Dataset Library code

```
import numpy as np
from scipy import ndimage, misc
import re
import matplotlib.image as mpimg
import os
import math
import pandas as pd

def rgb2gray(rgb):
    #convert image from rgg to grayscale
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])

def extract_vgg16_features(x):
    from keras.preprocessing.image import img_to_array, array_to_img
    from keras.applications.vgg16 import preprocess_input, VGG16
    from keras.models import Model

    # im_h = x.shape[1]
    im_h = 224
    model = VGG16(include_top=True, weights='imagenet', input_shape=(im_h, im_h, 3))
    # if flatten:
    #     add_layer = Flatten()
    # else:
    #     add_layer = GlobalMaxPool2D()
    # feature_model = Model(model.input, add_layer(model.output))
    feature_model = Model(model.input, model.get_layer('fc1').output)
    print('extracting features...')
    x1 = np.asarray([img_to_array(array_to_img(im, scale=False).resize((im_h,im_h)))
    for im in x[0:1000]])
    x1 = preprocess_input(x1)
    features = feature_model.predict(x1)
    for i in range(int(x.shape[0] / 1000)-1):
        x1 = np.asarray([img_to_array(array_to_img(im, scale=False).resize((im_h,im_h)))
        for im in x[1000*(i+1):1000*(i+2)]]])
        x1 = preprocess_input(x1) # data - 127. #data/255.#
        features1 = feature_model.predict(x1)
        features = np.concatenate((features, features1)).astype(float)
    print('Features shape = ', features.shape)
    return features
```

```

def make_reuters_data(data_dir):
    np.random.seed(1234)
    from sklearn.feature_extraction.text import CountVectorizer
    from os.path import join
    did_to_cat = {}
    cat_list = ['CCAT', 'GCAT', 'MCAT', 'ECAT']
    with open(join(data_dir, 'rcv1-v2.topics.qrels')) as fin:
        for line in fin.readlines():
            line = line.strip().split(' ')
            cat = line[0]
            did = int(line[1])
            if cat in cat_list:
                did_to_cat[cat] = did_to_cat.get(did, []) + [cat]
            # did_to_cat = {k: did_to_cat[k] for k in list(did_to_cat.keys()) if
len(did_to_cat[k]) > 1}
        for did in list(did_to_cat.keys()):
            if len(did_to_cat[cat]) > 1:
                del did_to_cat[cat]

    dat_list = ['lyrl2004_tokens_test_pt0.dat',
                'lyrl2004_tokens_test_pt1.dat',
                'lyrl2004_tokens_test_pt2.dat',
                'lyrl2004_tokens_test_pt3.dat',
                'lyrl2004_tokens_train.dat']

    data = []
    target = []
    cat_to_cid = {'CCAT': 0, 'GCAT': 1, 'MCAT': 2, 'ECAT': 3}
    del did
    for dat in dat_list:
        with open(join(data_dir, dat)) as fin:
            for line in fin.readlines():
                if line.startswith('.I'):
                    if 'did' in locals():
                        assert doc != ''
                        if did in did_to_cat:
                            data.append(doc)
                            target.append(cat_to_cid[did_to_cat[cat][0]])
                        did = int(line.strip().split(' ')[1])
                        doc = ''
                elif line.startswith('.W'):
                    assert doc == ''
                else:
                    doc += line

    print((len(data), 'and', len(did_to_cat)))
    assert len(data) == len(did_to_cat)

    x = CountVectorizer(dtype=np.float64, max_features=2000).fit_transform(data)

```

```

y = np.asarray(target)

from sklearn.feature_extraction.text import TfidfTransformer
x = TfidfTransformer(norm='l2', sublinear_tf=True).fit_transform(x)
x = x[:10000].astype(np.float32)
print(x.dtype, x.size)
y = y[:10000]
x = np.asarray(x.todense()) * np.sqrt(x.shape[1])
print('todense succeed')

p = np.random.permutation(x.shape[0])
x = x[p]
y = y[p]
print('permutation finished')

assert x.shape[0] == y.shape[0]
x = x.reshape((x.shape[0], -1))
np.save(join(data_dir, 'reutersidf10k.npy'), {'data': x, 'label': y}

)

def load_mice_protein(data_path='/content/drive/My Drive/Colab/ADEC/data/mice-
protein'):
    mouse_data = pd.read_csv(data_path + '/Data_Cortex_Nuclear.csv')
    x = mouse_data.to_numpy()[:, 1:78]
    y = mouse_data['class'].to_numpy()
    for i in range(1080):
        if y[i] == 'c-CS-s':
            y[i] = 0
        elif y[i] == 'c-CS-m':
            y[i] = 1
        elif y[i] == 't-CS-s':
            y[i] = 2
        elif y[i] == 't-CS-m':
            y[i] = 3
        elif y[i] == 'c-SC-s':
            y[i] = 4
        elif y[i] == 'c-SC-m':
            y[i] = 5
        elif y[i] == 't-SC-s':
            y[i] = 6
        elif y[i] == 't-SC-m':
            y[i] = 7
    for j in range(77):
        if math.isnan(x[i, j]):
            x[i, j] = 0.
    p = np.random.permutation(x.shape[0])
    x = x[p,:]

```

```

y = y[p]
x = x.reshape((x.shape[0], -1)).astype('float64')
y = y.reshape((y.size,))
print(('Mice-protein samples ', x.shape))
return x, y

#Load the Irish dataset
def load_irish(data_path='/content/drive/MyDrive/Dataset DynAe/data/irish'):
    #Load TON_IoT Subset-1
    #TON_IoT_data = pd.read_csv(data_path + '/TargetDataSet_1.csv')
    #Load TON_IoT Subset-2
    #TON_IoT_data = pd.read_csv(data_path + '/TargetDataSet_2.csv')
    #Load TON_IoT Subset-3
    #TON_IoT_data = pd.read_csv(data_path + '/TargetDataSet_3.csv')
    #Load TON_IoT Subset-4
    #TON_IoT_data = pd.read_csv(data_path + '/TargetDataSet_4.csv')
    #Load TON_IoT Subset-5
    irish_data = pd.read_csv(data_path + '/reducedirishdata.csv')
    x = irish_data.to_numpy()
    y = None
    print(('irish samples: ', x.shape))
    return x, y

def load_mnist():
    # the data, shuffled and split between train and test sets
    from tensorflow.keras.datasets import mnist
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    x = np.concatenate((x_train, x_test))
    y = np.concatenate((y_train, y_test))
    x = x.reshape([-1, 28, 28, 1]) / 255.0
    print('MNIST samples', x.shape)
    return x, y

def load_coil(data_path='/content/drive/My Drive/Colab/ADEC/data/COIL'):
    if not os.path.exists(data_path + '/data.npy'):
        X = np.zeros((1440, 128, 128))
        Y = np.zeros((1440, ), dtype=int)
        i = 0
        for filename in os.listdir(data_path):
            image = mpimg.imread(data_path + "/" + filename)
            X[i, :] = image
            Y[i] = int(filename[3: filename.rfind('_')-1]) - 1
            i = i + 1
            print(i)
        p = np.random.permutation(X.shape[0])
        X = X.reshape([-1, 128, 128, 1]) / 255.0
        X = X[p]
        Y = Y[p]

```

```

        data = {'X': X, 'Y': Y}
        file_name = data_path + '/data.npy'
        np.save(file_name, data)
    else:
        file_name = data_path + '/data.npy'
        data = np.load(file_name, allow_pickle=True) [()]
        X = data['X']
        Y = data['Y']
    print('COIL samples', X.shape)
    return X, Y

def load_coil_100_color(data_path='/content/drive/My Drive/Colab/ADEC/data/coil-100'):
    if not os.path.exists(data_path + '/data_color.npy'):
        X = np.zeros((7200, 128, 128, 3))
        Y = np.zeros((7200, ), dtype=int)
        i = 0
        for filename in os.listdir(data_path):
            #if re.search("\.(jpg|gif|jpeg|png|bmp|tiff)$", filename):
            image = mpimg.imread(data_path + "/" + filename)
            X[i, :] = image
            Y[i] = int(filename[3: filename.rfind('_')-1]) - 1
            i = i + 1
            print(i)
        p = np.random.permutation(X.shape[0])
        X = X.reshape([-1, 128, 128, 3]) / 255.0
        X = X[p]
        Y = Y[p]
        data = {'X': X, 'Y': Y}
        file_name = data_path + '/data_color.npy'
        np.save(file_name, data)
    else:
        file_name = data_path + '/data_color.npy'
        data = np.load(file_name, allow_pickle=True) [()]
        X = data['X']
        Y = data['Y']
    print('COIL-100 samples', X.shape)
    return X, Y

def load_coil_100_grayscale(data_path='/content/drive/My Drive/Colab/ADEC/data/coil-100'):
    if not os.path.exists(data_path + '/data.npy'):
        X = np.zeros((7200, 128, 128))
        Y = np.zeros((7200, ), dtype=int)
        i = 0
        for filename in os.listdir(data_path):
            image = mpimg.imread(data_path + "/" + filename)
            image = rgb2gray(image)
            X[i, :] = image

```

```

        Y[i] = int(filename[3: filename.rfind('_')-1]) - 1
        print(i)
        i = i + 1
    p = np.random.permutation(X.shape[0])
    X = X.reshape([-1, 128, 128, 1]) / 255.0
    X = X[p]
    Y = Y[p]
    data = {'X': X, 'Y': Y}
    file_name = data_path + '/data.npy'
    np.save(file_name, data)
else:
    file_name = data_path + '/data.npy'
    data = np.load(file_name, allow_pickle=True) [()]
    X = data['X']
    Y = data['Y']
print('COIL-100 samples', X.shape)
return X, Y

def load_mnist_test():
    # the data, shuffled and split between train and test sets
    from tensorflow.keras.datasets import mnist
    _, (x, y) = mnist.load_data()
    x = x.reshape([-1, 28, 28, 1]) / 255.0
    print('MNIST samples', x.shape)
    return x, y

def load_fashion_mnist():
    from tensorflow.keras.datasets import fashion_mnist # this requires keras>=2.0.9
    (x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
    x = np.concatenate((x_train, x_test))
    y = np.concatenate((y_train, y_test))
    x = x.reshape([-1, 28, 28, 1]) / 255.0
    print('Fashion MNIST samples', x.shape)
    return x, y

def load_pendigits(data_path='/content/drive/My Drive/Colab/DAE/data/pendigits'):
    import os
    if not os.path.exists(data_path + '/pendigits.tra'):
        os.system('wget http://mllearn.ics.uci.edu/databases/pendigits/pendigits.tra -P %s' % data_path)
        os.system('wget http://mllearn.ics.uci.edu/databases/pendigits/pendigits.tes -P %s' % data_path)
        os.system('wget http://mllearn.ics.uci.edu/databases/pendigits/pendigits.names -P %s' % data_path)

    # load training data
    with open(data_path + '/pendigits.tra') as file:
        data = file.readlines()

```

```

data = [list(map(float, line.split(','))) for line in data]
data = np.array(data).astype(np.float32)
data_train, labels_train = data[:, :-1], data[:, -1]
print('data_train shape=', data_train.shape)

# load testing data
with open(data_path + '/pendigits.tes') as file:
    data = file.readlines()
data = [list(map(float, line.split(','))) for line in data]
data = np.array(data).astype(np.float32)
data_test, labels_test = data[:, :-1], data[:, -1]
print('data_test shape=', data_test.shape)

x = np.concatenate((data_train, data_test)).astype('float32')
y = np.concatenate((labels_train, labels_test))
x /= 100.
print('pendigits samples:', x.shape)
return x, y

def load_usps(data_path='/content/drive/My Drive/Colab/DAE/data/usps'):
    import os
    if not os.path.exists(data_path+'/usps_train.jf'):
        if not os.path.exists(data_path+'/usps_train.jf.gz'):
            os.system('wget http://www-i6.informatik.rwth-
aachen.de/~keyzers/usps_train.jf.gz -P %s' % data_path)
            os.system('wget http://www-i6.informatik.rwth-
aachen.de/~keyzers/usps_test.jf.gz -P %s' % data_path)
            os.system('gunzip %s/usps_train.jf.gz' % data_path)
            os.system('gunzip %s/usps_test.jf.gz' % data_path)

    with open(data_path + '/usps_train.jf') as f:
        data = f.readlines()
    data = data[1:-1]
    data = [list(map(float, line.split())) for line in data]
    data = np.array(data)
    data_train, labels_train = data[:, 1:], data[:, 0]

    with open(data_path + '/usps_test.jf') as f:
        data = f.readlines()
    data = data[1:-1]
    data = [list(map(float, line.split())) for line in data]
    data = np.array(data)
    data_test, labels_test = data[:, 1:], data[:, 0]

    x = np.concatenate((data_train, data_test)).astype('float64') / 2.
    y = np.concatenate((labels_train, labels_test))
    x = x.reshape([-1, 16, 16, 1])
    print('USPS samples', x.shape)

```

```

return x, y

def load_reuters(data_path='/content/drive/My Drive/Colab/DAE/data/reuters'):
    import os
    if not os.path.exists(os.path.join(data_path, 'reutersidf10k.npy')):
        print('making reuters idf features')
        make_reuters_data(data_path)
        print(('reutersidf saved to ' + data_path))
    data = np.load(os.path.join(data_path, 'reutersidf10k.npy'),
allow_pickle=True).item()
    # has been shuffled
    x = data['data']
    y = data['label']
    x = x.reshape((x.shape[0], -1)).astype('float64')
    y = y.reshape((y.size,))
    print(('REUTERSIDF10K samples', x.shape))
    return x, y

def load_cifar10(data_path='/content/drive/My Drive/Colab/DAE/data/cifar10'):
    from keras.datasets import cifar10
    (train_x, train_y), (test_x, test_y) = cifar10.load_data()
    x = np.concatenate((train_x, test_x))
    y = np.concatenate((train_y, test_y)).reshape((60000,))

    # if features are ready, return them
    import os.path
    if os.path.exists(data_path + '/cifar10_features.npy'):
        return np.load(data_path + '/cifar10_features.npy', allow_pickle=True), y

    # extract features
    features = np.zeros((60000, 4096))
    for i in range(6):
        idx = range(i*10000, (i+1)*10000)
        print("The %dth 10000 samples" % i)
        features[idx] = extract_vgg16_features(x[idx])

    # scale to [0,1]
    from sklearn.preprocessing import MinMaxScaler
    features = MinMaxScaler().fit_transform(features)

    # save features
    np.save(data_path + '/cifar10_features.npy', features)
    print('features saved to ' + data_path + '/cifar10_features.npy')

    return features, y

def load_stl(data_path='/content/drive/My Drive/Colab/DAE/data/stl'):
    import os

```

```

    assert os.path.exists(data_path + '/stl_features.npy') or os.path.exists(data_path
+ '/train_X.bin'), \
        "No data! Use %s/get_data.sh to get data ready, then come back" % data_path

# get labels
y1 = np.fromfile(data_path + '/train_y.bin', dtype=np.uint8) - 1
y2 = np.fromfile(data_path + '/test_y.bin', dtype=np.uint8) - 1
y = np.concatenate((y1, y2))
y1 = None
y2 = None

# if features are ready, return them
if os.path.exists(data_path + '/stl_features.npy'):
    return np.load(data_path + '/stl_features.npy', allow_pickle=True), y

# get data
x1 = np.fromfile(data_path + '/train_X.bin', dtype=np.uint8)
x1 = x1.reshape((int(x1.size/3/96/96), 3, 96, 96)).transpose((0, 3, 2, 1))
x2 = np.fromfile(data_path + '/test_X.bin', dtype=np.uint8)
x2 = x2.reshape((int(x2.size/3/96/96), 3, 96, 96)).transpose((0, 3, 2, 1))
x = np.concatenate((x1, x2)).astype(float)
x1 = None
x2 = None

# extract features
features = extract_vgg16_features(x)

# scale to [0,1]
from sklearn.preprocessing import MinMaxScaler
features = MinMaxScaler().fit_transform(features)

# save features
np.save(data_path + '/stl_features.npy', features)
print('features saved to ' + data_path + '/stl_features.npy')

return features, y

def load_data_conv(dataset, datapath):
    if dataset == 'mnist':
        return load_mnist()
    elif dataset == 'mnist-test':
        return load_mnist_test()
    elif dataset == 'fmnist':
        return load_fashion_mnist()
    elif dataset == 'usps':
        return load_usps(datapath)
    elif dataset == 'pendigits':
        return load_pendigits(datapath)

```

```

elif dataset == 'reuters10k' or dataset== 'reuters':
    return load_reuters(datapath)
elif dataset == 'stl':
    return load_stl(datapath)
elif dataset == 'cifar10':
    return load_cifar10(datapath)
elif dataset == 'coil':
    return load_coil(datapath)
elif dataset == 'coil-100':
    return load_coil_100_grayscale(datapath)
elif dataset == 'coil-100-color':
    return load_coil_100_color(datapath)
elif dataset == 'mice-protein':
    return load_mice_protein(datapath)
elif dataset == 'irish':
    return load_irish(datapath)
else:
    raise ValueError('Not defined for loading %s' % dataset)

def load_data(dataset, datapath):
    x, y = load_data_conv(dataset, datapath)
    return x.reshape([x.shape[0], -1]), y

def generate_data_batch(x, y=None, batch_size=256):
    index_array = np.arange(x.shape[0])
    index = 0
    while True:
        idx = index_array[index * batch_size: min((index+1) * batch_size, x.shape[0])]
        index = index + 1 if (index + 1) * batch_size <= x.shape[0] else 0
        if y is None:
            yield x[idx]
        else:
            yield x[idx], y[idx]

def generate_transformed_batch(x, datagen, batch_size=256):
    if len(x.shape) > 2: # image
        gen0 = datagen.flow(x, shuffle=False, batch_size=batch_size)
        while True:
            batch_x = next(gen0)
            yield batch_x
    else:
        width = int(np.sqrt(x.shape[-1]))
        if width * width == x.shape[-1]: # gray
            im_shape = [-1, width, width, 1]
        else: # RGB
            width = int(np.sqrt(x.shape[-1] / 3.0))
            im_shape = [-1, width, width, 3]

```

```

        gen0 = datagen.flow(np.reshape(x, im_shape), shuffle=False,
batch_size=batch_size)
        while True:
            batch_x = next(gen0)
            batch_x = np.reshape(batch_x, [batch_x.shape[0], x.shape[-1]])
            yield batch_x

def generate(x, datagen, batch_size=256):
    gen1 = generate_data_batch(x, batch_size=batch_size)
    if len(x.shape) > 2: # image
        gen0 = datagen.flow(x, shuffle=False, batch_size=batch_size)
        while True:
            batch_x1 = next(gen0)
            batch_x2 = next(gen1)
            yield (batch_x1, batch_x2)
    else:
        width = int(np.sqrt(x.shape[-1]))
        if width * width == x.shape[-1]: # gray
            im_shape = [-1, width, width, 1]
        else: # RGB
            width = int(np.sqrt(x.shape[-1] / 3.0))
            im_shape = [-1, width, width, 3]
        gen0 = datagen.flow(np.reshape(x, im_shape), shuffle=False,
batch_size=batch_size)
        while True:
            batch_x1 = next(gen0)
            batch_x1 = np.reshape(batch_x1, [batch_x1.shape[0], x.shape[-1]])
            batch_x2 = next(gen1)
            yield (batch_x1, batch_x2)

def convert_shape_fc_to_conv(image_shape):
    width = int(np.sqrt(image_shape[-1]))
    if width * width == image_shape[-1]: # Gray
        im_shape = (image_shape[0], width, width, 1)
    else: # RGB
        width = int(np.sqrt(image_shape[-1] / 3.0))
        im_shape = (image_shape[0], width, width, 3)
    return im_shape

def convert_shape_conv_to_fc(image_shape):
    im_shape = (image_shape[0], np.prod(image_shape[1:]))
    return im_shape

```

DynAE code

```

import tensorflow as tf
import metrics

```

```

import math
import numpy as np
import matplotlib.pyplot as plt
import os

from sklearn.metrics import silhouette_score
from time import time

from tensorflow.keras.models import Model
from tensorflow.keras.losses import mean_squared_error
from datasets import generate
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Layer, Flatten, Lambda, InputSpec, Input, Dense
from tensorflow.keras.initializers import VarianceScaling
from tensorflow.keras import callbacks
from tensorflow.keras.utils import plot_model
from tensorflow.keras import backend as K
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.neighbors import NearestNeighbors
from datasets import generate_data_batch, generate_transformed_batch

#PATH_RESULT = '/content/drive/My Drive/Colab/DynAE/results'
#PATH_VIS = '/content/drive/My Drive/Colab/DynAE/visualisation'

PATH_RESULT = '/content/drive/My Drive/Colab Notebooks/Colab/DynAE/results'
PATH_VIS = '/content/drive/My Drive/Colab Notebooks/Colab/DynAE/visualisation'

def map_vector_to_clusters(y_true, y_pred):
    y_true = y_true.astype(np.int64)
    D = max(y_pred.max(), y_true.max()) + 1
    w = np.zeros((D, D), dtype=np.int64)
    for i in range(y_pred.size):
        w[y_true[i], y_pred[i]] += 1
    from scipy.optimize import linear_sum_assignment
    row_ind, col_ind = linear_sum_assignment(w.max() - w)

    y_true_mapped = np.zeros(y_pred.shape)
    for i in range(y_pred.shape[0]):
        y_true_mapped[i] = col_ind[y_true[i]]
    return y_true_mapped.astype(int)

#Generate clustering assignment matrix of eq(4) in article
def q_mat(X, centers, alpha=1.0):
    if X.size == 0:
        q = np.array([])
    else:
        q = 1.0 / (1.0 + (np.sum(np.square(np.expand_dims(X, 1) - centers), axis=2) /
alpha))
        q = q**((alpha+1.0)/2.0)

```

```

        q = np.transpose(np.transpose(q)/np.sum(q, axis=1))
    return q

def kl(x, y):
    X = tf.distributions.Categorical(probs=x)
    Y = tf.distributions.Categorical(probs=y)
    return tf.distributions.kl_divergence(X, Y)

def generate_supervisory_signals(x_emb, x_img, centers_emb_fixed, centers_img_fixed,
    beta1, beta2):
    q = q_mat(x_emb, centers_emb_fixed, alpha=1.0)
    y_pred = q.argmax(1)
    confidence1 = q.max(1)
    confidence2 = np.zeros((q.shape[0],))
    ind = np.argsort(q, axis=1)[:, -2]
    Y_encoder = []
    Y_autoencoder = []
    for i in range(x_img.shape[0]):
        confidence2[i] = q[i, ind[i]]
        if (confidence1[i]) > beta1 and (confidence1[i] - confidence2[i]) > beta2:
            Y_encoder.append(centers_emb_fixed[y_pred[i]])
            Y_autoencoder.append(centers_img_fixed[y_pred[i]])
        else:
            Y_encoder.append(x_emb[i])
            Y_autoencoder.append(x_img[i])
    Y_autoencoder = np.asarray(Y_autoencoder)
    Y_encoder = np.asarray(Y_encoder)
    return Y_encoder, Y_autoencoder

def draw_centers(n_clusters, centers_img, img_h=28, img_w=28):
    #plt.figure(figsize=(n_clusters, 4))
    #for i in range(n_clusters):
        #ax = plt.subplot(1, n_clusters, i + 1)
        #plt.imshow(centers_img[i].reshape(img_h, img_w))
        #plt.gray()
        #ax.get_xaxis().set_visible(False)
        #ax.get_yaxis().set_visible(False)
    #plt.show()
    print("can not draw ")

def total_loss(y_true, y_pred):
    #
    return y_pred

#Construction of Encoder
def encoder_constructor(dims, visualisation_dir, act='relu'):
    n_stacks = len(dims) - 1
    #Layer Weight Initializer

```

```

init = VarianceScaling(scale=1. / 3., mode='fan_in', distribution='uniform')
# input
x = Input(shape=(dims[0],), name='input_encoder')
h = x
# internal layers in encoder
for i in range(n_stacks-1):
    h = Dense(dims[i + 1], activation=act, kernel_initializer=init,
name='encoder_%d' % i)(h)
    # hidden layer
    h = Dense(dims[-1], kernel_initializer=init, name='encoder_%d' % (n_stacks -
1))(h) # hidden layer, features are extracted from here
    encoder = Model(inputs=x, outputs=h, name='encoder')
    plot_model(encoder, show_shapes=True, show_layer_names=True,
to_file=visualisation_dir + '/graph/FcEncoder.png')
    return encoder

#Construction of Decoder
def decoder_constructor(dims, visualisation_dir, act='relu'):
    n_stacks = len(dims) - 1
    #Layer Weight Initializer
    init = VarianceScaling(scale=1. / 3., mode='fan_in', distribution='uniform')
    # input
    z = Input(shape=(dims[-1],), name='input_decoder')
    y = z
    # internal layers in decoder
    for i in range(n_stacks-1, 0, -1):
        y = Dense(dims[i], activation=act, kernel_initializer=init, name='decoder_%d'
% i)(y)
    # output
    y = Dense(dims[0], kernel_initializer=init, name='decoder_0')(y)
    decoder = Model(inputs=z, outputs=y, name='decoder')
    plot_model(decoder, show_shapes=True, show_layer_names=True,
to_file=visualisation_dir + '/graph/FcDecoder.png')
    return decoder

#Construction of Autoencoder
def ae_constructor(encoder, decoder, dims, visualisation_dir):
    x = Input(shape=(dims[0],), name='input_autencoder')
    autoencoder = Model(inputs=x, outputs=decoder(encoder(x)), name='autoencoder')
    plot_model(autoencoder, show_shapes=True, show_layer_names=True,
to_file=visualisation_dir + '/graph/FcAutoencoder.png')
    return autoencoder

#Construction of Dynamic Autoencoder
def dynAE_constructor(encoder, ae, dims, gamma, visualisation_dir):
    input1 = Input(shape=(dims[0],), name='input_dynAE')
    target1 = Input(shape=(dims[-1],), name='target1_dynAE')
    target2 = Input(shape=(dims[0],), name='target2_dynAE')

```

```

target3 = Input(shape=(dims[-1],), name='target3_dynAE')

def loss_dynAE(x):
    encoder_output = x[0]
    ae_output = x[1]
    target1 = x[2]
    target2 = x[3]
    loss1 = tf.losses.mean_squared_error(encoder_output, target1)
    loss2 = tf.losses.mean_squared_error(ae_output, target2)
    return loss1 + gamma * loss2

x1 = encoder(input1)
x2 = ae(input1)
out1 = Lambda(lambda x: loss_dynAE(x), name="output1_dynAE")((x1, x2, target1,
target2))
dynAE = Model(inputs=[input1, target1, target2], outputs=[out1], name='dynAE')
plot_model(dynAE, show_shapes=True, show_layer_names=True,
to_file=visualisation_dir + '/graph/FcDynAE.png')
return dynAE

# Computing Feature Randomness and Feature Drift coefficients
#loss_pseudo_supervised = tf.losses.mean_squared_error(x1, target1)
#loss_self_supervised = tf.losses.mean_squared_error(x2, target2)
#loss_supervised = tf.losses.mean_squared_error(x1, target3)
#loss_dynAE = loss_pseudo_supervised + gamma * loss_self_supervised
#grad_loss_pseudo_supervised = []
#grad_loss_self_supervised = []
#grad_loss_supervised = []
#grad_loss_dynAE = []
#for i in range((len(dims) - 1) * 2):
#    grad_loss_pseudo_supervised.append(tf.gradients(loss_pseudo_supervised,
dynAE.trainable_weights[i]))
#    grad_loss_self_supervised.append(tf.gradients(loss_self_supervised,
dynAE.trainable_weights[i]))
#    grad_loss_supervised.append(tf.gradients(loss_supervised,
dynAE.trainable_weights[i]))
#    grad_loss_dynAE.append(tf.gradients(loss_dynAE, dynAE.trainable_weights[i]))
#return dynAE, grad_loss_pseudo_supervised, grad_loss_self_supervised,
grad_loss_supervised, grad_loss_dynAE

#Construction of Interpolation Autoencoder
def i_ae_constructor(encoder, decoder, dims, visualisation_dir):
    input1 = Input(shape=(dims[0],), name='input_i_ae')
    alpha1 = Input(shape=(1,), name='alpha i ae')

    def mix_layer(x):
        encode = x[0]
        alpha1 = x[1]
        encode_mix = alpha1 * encode + (1 - alpha1) * encode[::-1]

```

```

        return encode_mix

    encode = encoder(input1)
    encode_mix = Lambda(mix_layer, name="mix_layer")((encode, alpha1))
    decode_mix = decoder(encode_mix)
    i_ae = Model(inputs=[input1, alpha1], outputs=decode_mix,
name='interpolation_autoencoder')
    plot_model(i_ae, show_shapes=True, show_layer_names=True,
to_file=visualisation_dir + '/graph/FcI_AE.png')
    return i_ae

#Construction of Critic Network
def critic_constructor(dims, visualisation_dir, act='relu'):
    n_stacks = len(dims) - 1
    #Layer Weight Initializer
    init = VarianceScaling(scale=1. / 3., mode='fan_in', distribution='uniform')
    # input
    x = Input(shape=(dims[0],), name='input_critic')
    h = x
    # internal layers in encoder
    for i in range(n_stacks-1):
        h = Dense(dims[i + 1], activation=act, kernel_initializer=init,
name='critic_%d' % i)(h)
    # hidden layer
    h = Dense(dims[-1], kernel_initializer=init, name='critic_%d' % (n_stacks - 1))(h)
    # hidden layer, features are extracted from here
    h = Lambda(lambda x: tf.reduce_mean(x, [1]), name="reduce_mean_layer")(h)
    critic = Model(inputs=x, outputs=h, name='critic')
    plot_model(critic, show_shapes=True, show_layer_names=True,
to_file=visualisation_dir + '/graph/FcCritic.png')
    return critic

#Construction of Discriminator Network
def disc_constructor(critic, dims, visualisation_dir):
    input1 = Input(shape=(dims[0],), name='input1_disc')
    input2 = Input(shape=(dims[0],), name='input2_disc')
    alpha1 = Input(shape=(1,), name='alpha1_disc')
    alpha2 = Input(shape=(1,), name='alpha2_disc')

    def loss_disc(x):
        x1 = x[0]
        x2 = x[1]
        alpha1 = x[2]
        alpha2 = x[3]
        loss_within_cluster_interpol = tf.reduce_mean(tf.square(x1 - alpha1))
        loss_between_cluster_interpol = tf.reduce_mean(tf.square(x2 - alpha2))
        return loss_within_cluster_interpol + loss_between_cluster_interpol

```

```

x1 = critic(input1)
x2 = critic(input2)
out1 = Lambda(loss_disc, name="output1_disc")((x1, x2, alpha1, alpha2))

disc = Model(inputs=[input1, input2, alpha1, alpha2], outputs=[out1],
name='discriminator')

plot_model(disc, show_shapes=True, show_layer_names=True,
to_file=visualisation_dir + '/graph/FcDiscriminator.png')
return disc

#Construction of Adversarially Constrained Interpolation Autoencoder
def aci_ae_constructor(critic, ae, i_ae, advweight, dims, visualisation_dir):
    critic.trainable = False
    input1 = Input(shape=(dims[0],), name='input_aci_ae')
    alpha1 = Input(shape=(1,), name='alpha_aci_ae')

    def loss_aci_ae(x):
        input1 = x[0]
        x1 = x[1]
        x2 = x[2]
        loss_ae_critic = tf.reduce_mean(tf.square(x2))
        loss_ae = tf.losses.mean_squared_error(input1, x1)
        return loss_ae + advweight * loss_ae_critic

    x1 = ae(input1)
    x2 = critic(i_ae([input1, alpha1]))

    out1 = Lambda(lambda x: loss_aci_ae(x), name="output1_aci_ae")((input1, x1, x2))
    aci_ae = Model(inputs=[input1, alpha1], outputs=[out1], name='aci_ae')
    plot_model(aci_ae, show_shapes=True, show_layer_names=True,
to_file=visualisation_dir + '/graph/FcACI_AE.png')
    return aci_ae

class DynAE:

    def __init__(self, batch_size, dataset, dims, loss_weight, gamma=1.0,
n_clusters=10, alpha=1.0, visualisation_dir=PATH_VIS, ws=0.1, hs=0.1, rot=10,
scale=0.0):
        self.batch_size = batch_size
        self.dataset = dataset
        self.dims = dims
        self.visualisation_dir = visualisation_dir
        self.n_clusters = n_clusters
        self.alpha = alpha
        self.loss_weight = loss_weight
        self.datagen = ImageDataGenerator(width shift range=ws, height shift range=hs,
rotation_range=rot, zoom_range=scale)

```

```

        self.ws = ws
        self.hs = hs
        self.rot = rot
        self.scale = scale
        self.gamma = gamma
        #models
        self.encoder = encoder_constructor(self.dims, self.visualisation_dir)
        self.decoder = decoder_constructor(self.dims, self.visualisation_dir)
        self.ae = ae_constructor(self.encoder, self.decoder, self.dims,
self.visualisation_dir)
        self.i_ae = i_ae_constructor(self.encoder, self.decoder, self.dims,
self.visualisation_dir)
        ##self.dynAE, self.grad_loss_pseudo_supervised,
self.grad_loss_self_supervised, self.grad_loss_supervised, self.grad_loss_dynAE =
dynAE_constructor(self.encoder, self.ae, self.dims, self.gamma,
self.visualisation_dir)
        self.dynAE = dynAE_constructor(self.encoder, self.ae, self.dims, self.gamma,
self.visualisation_dir)
        self.critic = critic_constructor(self.dims, self.visualisation_dir)
        self.disc = disc_constructor(self.critic, self.dims, self.visualisation_dir)
        self.aci_ae = aci_ae_constructor(self.critic, self.ae, self.i_ae,
self.loss_weight, self.dims, self.visualisation_dir)

def predict_ae(self, x):
    x_recons = self.ae.predict(x, verbose=0)
    return x_recons

#Generate output predictions for input samples
#x:input samples
#verbose:Verbosity mode
def predict_encoder(self, x):
    x_encode = self.encoder.predict(x, verbose=0)
    return x_encode

def predict_i_ae(self, x, alpha):
    x_inter_recons = self.i_ae.predict([x, alpha], verbose=0)
    return x_inter_recons

def compile_ae(self, optimizer='sgd'):
    #
    self.ae.compile(optimizer=optimizer, loss='mse')

#Configure dynAE model for training:
#optimizer:String or optimizer instance
#loss:Loss function
def compile_dynAE(self, optimizer='sgd'):
    #
    self.dynAE.compile(optimizer=optimizer, loss=total_loss)

```

```

def compile_critic(self, optimizer='sgd'):
    #
    self.critic.compile(optimizer=optimizer, loss='mse')

def compile_disc(self, optimizer='sgd'):
    #
    self.disc.compile(optimizer=optimizer, loss='total_loss')

def compile_aci_ae(self, optimizer='sgd'):
    #
    self.aci_ae.compile(optimizer=optimizer, loss='total_loss')

def train_on_batch_ae(self, x, y):
    #
    return self.ae.train_on_batch(x, y)

def train_on_batch_dynAE(self, x, y1, y2):
    #
    y = np.zeros((x.shape[0],))
    return self.dynAE.train_on_batch([x, y1, y2], y)

def train_on_batch_disc(self, x1, x2, y1, y2):
    y = np.zeros((x1.shape[0],))
    return self.disc.train_on_batch([x1, x2, y1, y2], y)

def train_on_batch_aci_ae(self, x1, x2):
    y = np.zeros((x1.shape[0],))
    return self.aci_ae.train_on_batch([x1, x2], y)

#Pre-train Autoencoder
def train_ae(self, x, y=None, optimizer='adam', epochs=500, batch_size=256,
save_dir=PATH_RESULT, verbose=1, aug_train=True):
    print('Begin pretraining: ', '-' * 60)
    #Configure autoencoder for training
    self.ae.compile(optimizer=optimizer, loss='mse')
    #Record pretraining results to a CSV file (loss per epoch)
    csv_logger = callbacks.CSVLogger(save_dir + '/' + self.dataset +
'/pretrain/log_ae.csv')
    cb = [csv_logger]

    #####
    if y is not None and verbose > 0:
        class PrintACC(callbacks.Callback):
            def __init__(self, x, y, encoder):
                self.x = x
                self.y = y
                self.encoder = encoder

```

```

        super(PrintACC, self).__init__()

    def on_epoch_end(self, epoch, logs=None):
        if epoch % 20 != 0:
            return

        features = self.encoder.predict(self.x)
        km = KMeans(n_clusters=len(np.unique(y)), n_init=20)
        y_pred = km.fit_predict(features)
        print(' '*8 + '|==> acc: %.4f, nmi: %.4f <==|'% (metrics.acc(y,
y_pred), metrics.nmi(y, y_pred)))
        cb.append(PrintACC(x, y, self.encoder))
        #####

    # begin pre-training
    #If cluster labels are unknown for training samples, i.e. y is None
    #Start recording pre-training time
    t0 = time()
    #If there is no data augmentation
    #Train the autoencoder for a fixed number of epochs on the training dataset
    #x:input data
    #y(passed as x again):target data
    #batch_size:Number of samples per gradient update (Can be an Interger or None,
default = 32)
    #epochs:Number of epochs to train the model (Must be a Integer)
    #callbacks:List of callbacks to apply during model training (Record loss per
epoch to a CSV log file)
    #verbose:Verbosity mode (1: Progress Bar)
    if not aug_train:
        self.ae.fit(x, x, batch_size=batch_size, epochs=epochs, callbacks=cb,
verbose=verbose)

    #####
    else:
        print('-=='*20)
        print('Using augmentation for ae')
        print('-=='*20)
        def gen(x, batch_size):
            if len(x.shape) > 2: # image
                gen0 = self.datagen.flow(x, shuffle=True, batch_size=batch_size)
                while True:
                    batch_x = gen0.next()
                    yield (batch_x, batch_x)
            else:
                width = int(np.sqrt(x.shape[-1]))
                if width * width == x.shape[-1]: # gray
                    im_shape = [-1, width, width, 1]
                else: # RGB
                    width = int(np.sqrt(x.shape[-1] / 3.0))

```

```

        im_shape = [-1, width, width, 3]
        gen0 = self.datagen.flow(np.reshape(x, im_shape), shuffle=True,
batch_size=batch_size)

        while True:
            batch_x = gen0.next()
            batch_x = np.reshape(batch_x, [batch_x.shape[0], x.shape[-1]])
            yield (batch_x, batch_x)

        self.ae.fit_generator(gen(x, batch_size),
steps_per_epoch=int(x.shape[0]/batch_size), epochs=epochs, callbacks=cb,
verbose=verbose)

#####

#Record overall autoencoder training time
exec_time = time() - t0
print('Pretraining time: %ds' % (exec_time))

#Save all layer weights of pre-trained autoencoder model in HDF5 format
self.ae.save_weights(save_dir + '/' + self.dataset +
'/pretrain/ae_weights.h5')
print('Pretrained weights are saved to %s/%s/pretrain/ae_weights.h5' %
(save_dir, self.dataset))

#####

if y is not None:
    features = self.encoder.predict(x)
    km = KMeans(n_clusters=len(np.unique(y)), n_init=20)
    y_pred = km.fit_predict(features)
    f = open(save_dir + '/' + self.dataset + '/pretrain/results_train_ae.txt',
'w+')

    f.write('|==> ACC: %.4f, NMI: %.4f <==| \n' % (metrics.acc(y, y_pred),
metrics.nmi(y, y_pred)))
    f.write('|==> Execution time: %.5f <==| \n' % (exec_time))
    f.close()

#####

print('End pretraining: ', '-' * 60)

def train_aci_ae(self, x, y=None, maxiter=120e3, batch_size=256,
validate_interval=2800, save_interval=2800, save_dir=PATH_RESULT, verbose=1,
aug_train=True):
    print('Begin aci_ae training: ', '-' * 60)

    #Prepare log file
    import csv, os
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)

    logfile = open(save_dir + '/' + self.dataset +
'/pretrain/train_aci_ae_log.csv', 'w')

```

```

        logwriter = csv.DictWriter(logfile, fieldnames=['iter', 'acc', 'nmi',
'loss_aci_ae', 'loss_disc'])
        logwriter.writeheader()

#Initialization
t0 = time()
loss_aci_ae = 0
loss_disc = 0
index = 0
index_array = np.arange(x.shape[0])

#Training loop
for ite in range(int(maxiter)):
    #Validation interval
    if ite % validate_interval == 0:
        if y is not None and verbose > 0:
            avg_loss_aci_ae = loss_aci_ae / validate_interval
            avg_loss_disc = loss_disc / validate_interval
            loss_aci_ae = 0.
            loss_disc = 0.
            features = self.predict_encoder(x)
            km = KMeans(n_clusters=self.n_clusters, n_init=20)
            y_pred = km.fit_predict(features)
            acc = np.round(metrics.acc(y, y_pred), 5)
            nmi = np.round(metrics.nmi(y, y_pred), 5)
            ari = np.round(metrics.ari(y, y_pred), 5)
            print('Iter %d: acc=%.5f, nmi=%.5f, loss_aci_ae=%.5f,
loss_disc=%.5f' % (ite, acc, nmi, avg_loss_aci_ae, avg_loss_disc))
            logdict = dict(iter=ite, acc=acc, nmi=nmi,
loss_aci_ae=avg_loss_aci_ae, loss_disc=avg_loss_disc)
            logwriter.writerow(logdict)
            logfile.flush()

    #Save interval
    if ite % save_interval == 0:
        self.ae.save_weights(save_dir + '/' + self.dataset +
'/pretrain/ae_weights.h5')
        self.critic.save_weights(save_dir + '/' + self.dataset +
'/pretrain/critic_weights.h5')

    #Train on batch
    idx = index_array[index * batch_size: min((index+1) * batch_size,
x.shape[0])]
    x_batch = x[idx]
    np.random.shuffle(x_batch)
    x_batch = self.random_transform(x_batch, ws=self.ws, hs=self.hs,
rot=self.rot, scale=self.scale) if aug_train else x_batch

```

```

        alpha_interp = np.random.uniform(low=0.0, high=1.0,
size=[x_batch.shape[0]])
        beta_interp = np.random.uniform(low=0.0, high=1.0, size=None)
        beta_interp = 0.5 - np.abs(beta_interp - 0.5)

        x_batch_recons = self.predict_ae(x_batch)
        x_batch_recons_interp = self.predict_i_ae(x_batch, alpha_interp)
        x_batch_mix = np.multiply(beta_interp, x_batch) + np.multiply(1 -
beta_interp, x_batch_recons)

        loss1 = self.train_on_batch_aci_ae(x_batch, alpha_interp)
        loss2 = self.train_on_batch_disc(x_batch_recons_interp, x_batch_mix,
alpha_interp, np.zeros((x_batch.shape[0],)))

        loss_aci_ae = loss_aci_ae + loss1
        loss_disc = loss_disc + loss2
        index = index + 1 if (index + 1) * batch_size <= x.shape[0] else 0

    logfile.close()
    print('training time: ', time() - t0)
    self.ae.save_weights(save_dir + '/' + self.dataset +
'/pretrain/ae_weights.h5')
    self.critic.save_weights(save_dir + '/' + self.dataset +
'/pretrain/critic_weights.h5')
    print('trained weights are saved to %s/%s/pretrain/ae_weights.h5' % (save_dir,
self.dataset))
    print('trained weights are saved to %s/%s/critic_weights.h5' % (save_dir,
self.dataset))
    print('training: ', '-' * 60)

    #Train DynAE
    def train_dynAE(self, x, y=None, kappa=3, n_clusters=10, maxiter=1e5,
batch_size=256, tol=1e-2, validate_interval=140, show_interval=None,
save_interval=2800, save_dir=PATH_RESULT, aug_train=True):
        #init
        number_of_samples = x.shape[0]

        #####
        img_h = int(math.sqrt(x.shape[1]))
        img_w = int(math.sqrt(x.shape[1]))
        #####

        #Logging file
        import csv, os
        #Check if save_dir directory exists, else create directories that do no exist
        if not os.path.exists(save_dir):
            os.makedirs(save_dir)
        #Open log file for recording results of clustering phase

```

```

        logfile = open(save_dir + '/' + self.dataset + '/cluster/train_dynAE_gamma=' +
str(self.gamma) + '_log.csv', 'w')

        #logwriter = csv.DictWriter(logfile, fieldnames=['iter', 'acc', 'nmi', 'ari',
'acc_unconf', 'nmi_unconf', 'acc_conf', 'nmi_conf', 'nb_unconf', 'nb_conf', 'fr',
'fd', 'loss'])

        #csv.DictWriter operates like a regular writer but maps Python dictionaries
into CSV rows

        #logfile:csv file
        #fieldnames:csv file column headers
        logwriter = csv.DictWriter(logfile, fieldnames=['iter', 'acc', 'nmi', 'ari',
'acc_unconf', 'nmi_unconf', 'acc_conf', 'nmi_conf', 'nb_unconf', 'nb_conf', 'loss'])

        #Write a row in the logfile with the fieldnames
        logwriter.writeheader()

#intervals config
print('Begin clustering:', '-' * 60)
if save_interval is None:
    save_interval = int(maxiter) # only save the initial and final model
print('Save interval ', save_interval)
if show_interval is None:
    show_interval = int(np.ceil(number_of_samples/batch_size))*20
print('show interval ', show_interval)

# Step 1: initialize (embedded) cluster centers using k-means
#centers_emb:Embedded cluster centers generated by k-means
#centers_img:Centers' images generated by decoder from embedded cluster
centers

#y_pred:Predicted labels generated by k-means
#Start recording clustering time
t1 = time()
print('Initializing cluster centers with k-means. \n')
centers_emb, centers_img, y_pred, _ = self.generate_centers(x, n_clusters)

# Step 2: beta1 and beta2
#beta1: Minimal confidence threshold under which a data point is considered
#to be conflicted
#beta2: Minimal difference between the highest and
#the second-highest assignment probabilities
beta1, beta2 = self.generate_beta(kappa, n_clusters)

# Step 3: deep clustering
loss = 0
index = 0
#Initialize number of conflicted points to total number of samples in the
dataset

nb_conf_prev = x.shape[0]
index array = np.arange(x.shape[0])
delta_kappa = 0.3 * kappa

```

```

sess = tf.compat.v1.keras.backend.get_session()
first_time = 1
for ite in range(int(maxiter)):

    ##
    print('Current Iteration: ', ite)
    print('-' * 30)
    print('Current Loss: ', loss)
    ##

    #For every (validate_interval) iterations
    if ite % validate_interval == 0:

        ##
        print('IF statement (1)')
        ##

        x_emb = self.encoder.predict(x)
        q = q_mat(x_emb, centers_emb)
        y_pred = q.argmax(1)
        avg_loss = loss / validate_interval
        loss = 0.

        ##
        print('Average Loss: ', avg_loss)
        ##

        if ite > 0:
            print('IF statement (2)')
            nb_conf_prev = nb_conf

        nb_unconf, nb_conf = self.compute_nb_conflicted_data(x, centers_emb,
beta1, beta2)

        ##
        print ('Current number of unconflicted data points:', nb_unconf)
        print ('Current number of conflicted data points:', nb_conf)
        ##

        #update centers
        if nb_conf >= nb_conf_prev and first_time == 1:
            print('IF statement (3)')
            centers_emb, centers_img, , = self.generate_centers(x,
n_clusters)

            #print("update centers")
            beta1 = beta1 - (delta_kappa / n_clusters)
            print ("New beta1 = ", beta1)
            beta2 = beta2 - (delta_kappa / n_clusters)

```

```

        print ("New beta2 = ", beta2)
        delta_kappa = 0.3 * kappa
        print ("New delta_kappa = ", delta_kappa)
        kappa = delta_kappa
        print ("New kappa = ", kappa)
        first_time = 0
        #print("update confidences")

    ###
    if y is not None:
        y_mapped = map_vector_to_clusters(y, y_pred)
        x_emb = self.predict_encoder(x)
        y_encoder, y_autoencoder = generate_supervisory_signals(x_emb, x,
centers_emb, centers_img, beta1, beta2)
        y_encoder_true = centers_emb[y_mapped]
        #grad_loss_dynAE = sess.run(self.grad_loss_dynAE,
feed_dict={'input_dynAE:0': x, 'target1_dynAE:0': y_encoder, 'target2_dynAE:0':
y_autoencoder})

        #grad_loss_pseudo_supervised =
sess.run(self.grad_loss_pseudo_supervised, feed_dict={'input_dynAE:0': x,
'target1_dynAE:0': y_encoder})
        #grad_loss_self_supervised =
sess.run(self.grad_loss_self_supervised, feed_dict={'input_dynAE:0': x,
'target2_dynAE:0': y_autoencoder})
        #grad_loss_supervised = sess.run(self.grad_loss_supervised,
feed_dict={'input_dynAE:0': x, 'target3_dynAE:0': y_encoder_true})
        acc = np.round(metrics.acc(y, y_pred), 5)
        nmi = np.round(metrics.nmi(y, y_pred), 5)
        ari = np.round(metrics.ari(y, y_pred), 5)
        #fr = np.round(metrics.cos_grad(grad_loss_supervised,
grad_loss_dynAE), 5)

        #fd = np.round(metrics.cos_grad(grad_loss_self_supervised,
grad_loss_pseudo_supervised), 5)

        acc_unconf, nmi_unconf, acc_conf, nmi_conf =
self.compute_acc_and_nmi_conflicted_data(x, y, centers_emb, beta1, beta2)
        #logdict = dict(iter=ite, acc=acc, nmi=nmi, ari=ari,
acc_unconf=acc_unconf, nmi_unconf=nmi_unconf, acc_conf=acc_conf, nmi_conf=nmi_conf,
nb_unconf=nb_unconf, nb_conf=nb_conf, fr=fr, fd=fd, loss=avg_loss)
        logdict = dict(iter=ite, acc=acc, nmi=nmi, ari=ari,
acc_unconf=acc_unconf, nmi_unconf=nmi_unconf, acc_conf=acc_conf, nmi_conf=nmi_conf,
nb_unconf=nb_unconf, nb_conf=nb_conf, loss=avg_loss)
        logwriter.writerow(logdict)
        logfile.flush()

        #print('Iter %d: acc=%.5f, nmi=%.5f, ari=%.5f, acc_unconf=%.5f,
nmi_unconf=%.5f, acc_conf=%.5f, nmi_conf=%.5f, nb_unconf=%d, nb_conf=%d, fr=%.5f,
fd=%.5f, loss=%.5f' % (ite, acc, nmi, ari, acc_unconf, nmi_unconf, acc_conf, nmi_conf,
nb_unconf, nb_conf, fr, fd, avg_loss))

```

```

        print('Iter %d: acc=%.5f, nmi=%.5f, ari=%.5f, acc_unconf=%.5f,
nmi_unconf=%.5f, acc_conf=%.5f, nmi_conf=%.5f, nb_unconf=%d, nb_conf=%d, loss=%.5f' %
(ite, acc, nmi, ari, acc_unconf, nmi_unconf, acc_conf, nmi_conf, nb_unconf, nb_conf,
avg_loss))

        print("The number of unconflicted data points is : " +
str(nb_unconf))

        print("The number of conflicted data points is : " + str(nb_conf))
        ###

    else:
        print('ELSE statement (1)')
        x_embed = self.encoder.predict(x)

        ##
        #print('The shape of embedded samples array predicted by
encoder:', x_embed.shape)
        ##

        #The silhouette_score gives the average value for all the samples
in the dataset

        #This gives a perspective into the density and separation of the
formed clusters

        silhouette_avg = silhouette_score(x_embed, y_pred)
        print("For number of clusters = " + str(self.n_clusters) + " The
average silhouette_score is: ", silhouette_avg)
        #logdict = dict(iter=ite, silhouette_avg=silhouette_avg,
loss=avg_loss)

        #logwriter.writerow(logdict)
        #logfile.flush()

        #If convergence threshold reached end clustering phase
        if(nb_conf / x.shape[0]) < tol:
            print('IF statement (4)')
            logfile.close()
            break

    #if ite % show_interval == 0 and ite!=0:
        #print("")
        #print("-----")
        #print("-----")
        #print("Centroids : ")
        #print("-----")
        #print("-----")
        #draw_centers(n_clusters, centers_img, img_h=img_h, img_w=img_w)
        #print ("draw")

    # save intermediate model
    if ite % save_interval == 0:

```

```

        print('IF statement (5)')
        print("")
        print("-----")
    -----")
        print("Save embeddings for visualization : ")
        print("-----")
    -----")
        z = self.predict_encoder(x)
        q1 = q_mat(z, centers_emb)
        y1_pred = q1.argmax(1)

        pca = PCA(n_components=2).fit(z)
        z_2d = pca.transform(z)
        centers_2d = pca.transform(centers_emb)

        # save states for visualization
        #np.save(self.visualisation_dir + '/embeddings/' + self.dataset +
        '/vis_' + str(ite) + '.numpy', {'z_2d': z_2d, 'centers_2d': centers_2d, 'y_pred':
        y1_pred})

        print('saving model to: ', save_dir + '/' + self.dataset +
        '/cluster/ae_' + str(ite) + '_weights.h5')
        self.ae.save_weights(save_dir + '/' + self.dataset + '/cluster/ae_' +
        str(ite) + '_weights.h5')

        # train on batch
        idx = index_array[index * batch_size: min((index+1) * batch_size,
        x.shape[0])]

        X_img = x[idx]
        X_emb = self.predict_encoder(X_img)

        Y_encoder, Y_autoencoder = generate_supervisory_signals(X_emb, X_img,
        centers_emb, centers_img, beta1, beta2)
        X_transformed = self.random_transform(X_img, ws=self.ws, hs=self.hs,
        rot=self.rot, scale=self.scale) if aug_train else X_img

        losses = self.train_on_batch_dynAE(X_transformed, Y_encoder,
        Y_autoencoder)
        losses = np.nan_to_num(losses)
        print ('Train on batch loss = ', losses)
        loss = loss + losses
        index = index + 1 if (index + 1) * batch_size <= x.shape[0] else 0
        print ('Current batch index : ', index)
        print('End of iteration', ite, '\n')

    # save the trained model
    logfile.close()

```

```

        print('saving model to:', save_dir + '/' + self.dataset +
              '/cluster/ae_weights.h5')
        self.ae.save_weights(save_dir + '/' + self.dataset + '/cluster/ae_weights.h5')
        print('Clustering time: %ds' % (time() - t1))
        print('End clustering:', '-' * 60)

    return y_pred

def generate_centers(self, x, n_clusters):
    features = self.predict_encoder(x)

    ###
    #print('The shape of features array predicted by encoder:', features.shape)
    ###

    #Generate predicted label of each sample using k-means
    kmeans = KMeans(n_clusters=n_clusters, n_init=10)
    y_pred = kmeans.fit_predict(features)

    #Calculate the clustering assignment matrix (refer to eq(4) in article)
    q = q_mat(features, kmeans.cluster_centers_, alpha=1.0)

    ###
    #print ('The soft clustering assignments matrix is:', q.shape)
    ###

    #Find nearest neighboring points to predicted k-means cluster centers
    nn = NearestNeighbors(n_neighbors=1, algorithm='ball_tree').fit(features)
    _, indices = nn.kneighbors(kmeans.cluster_centers_)

    #Generate the embedded cluster centers and their images from decoder
    centers_emb = np.reshape(features[indices], (-1,
self.encoder.output.shape[1]))
    centers_img = self.decoder.predict(centers_emb)
    return centers_emb, centers_img, y_pred, q

#Generate the indices of both unconflicted and conflicted data points
def generate_unconflicted_data_index(self, x_img, centers_emb, betal, beta2):
    x_emb = self.encoder.predict(x_img)
    unconf_indices = []
    conf_indices = []
    q = q_mat(x_emb, centers_emb, alpha=1.0)
    confidence1 = q.max(1)
    confidence2 = np.zeros((q.shape[0],))
    a = np.argsort(q, axis=1)[:, -2]
    for i in range(x_img.shape[0]):
        confidence2[i] = q[i, a[i]]
        if (confidence1[i] > betal and (confidence1[i] - confidence2[i]) > beta2:

```

```

        unconf_indices.append(i)
    else:
        conf_indices.append(i)
    unconf_indices = np.asarray(unconf_indices, dtype=int)
    conf_indices = np.asarray(conf_indices, dtype=int)
    return unconf_indices, conf_indices

def compute_acc_and_nmi_conflicted_data(self, x, y, centers_emb, beta1, beta2):
    features = self.predict_encoder(x)
    unconf_indices, conf_indices = self.generate_unconflicted_data_index(x,
centers_emb, beta1, beta2)

    if unconf_indices.size == 0:
        print(' '*8 + "Empty list of unconflicted data")
        acc_unconf = 0
        nmi_unconf = 0
    else:
        x_emb_unconf = self.predict_encoder(x[unconf_indices])
        y_unconf = y[unconf_indices]
        y_pred_unconf = q_mat(x_emb_unconf, centers_emb, alpha=1.0).argmax(axis=1)
        acc_unconf = metrics.acc(y_unconf, y_pred_unconf)
        nmi_unconf = metrics.nmi(y_unconf, y_pred_unconf)
        print(' '*8 + '|==> acc unconflicted data: %.4f, nmi unconflicted data:
%.4f <==|'% (acc_unconf, nmi_unconf))

    if conf_indices.size == 0:
        print(' '*8 + "Empty list of conflicted data")
        acc_conf = 0
        nmi_conf = 0
    else:
        x_emb_conf = self.predict_encoder(x[conf_indices])
        y_conf = y[conf_indices]
        y_pred_conf = q_mat(x_emb_conf, centers_emb, alpha=1.0).argmax(axis=1)
        acc_conf = metrics.acc(y_conf, y_pred_conf)
        nmi_conf = metrics.nmi(y_conf, y_pred_conf)
        print(' '*8 + '|==> acc conflicted data: %.4f, nmi conflicted data: %.4f
<==|'% (metrics.acc(y_conf, y_pred_conf), metrics.nmi(y_conf, y_pred_conf)))
    return acc_unconf, nmi_unconf, acc_conf, nmi_conf

def compute_acc_and_nmi(self, x, y):
    features = self.predict_encoder(x)
    km = KMeans(n_clusters=len(np.unique(y)), n_init=20)
    y_pred = km.fit_predict(features)
    print(' '*8 + '|==> acc: %.4f, nmi: %.4f <==|'% (metrics.acc(y, y_pred),
metrics.nmi(y, y_pred)))

#Compute the number of both unconflicted and conflicted data points
def compute_nb_conflicted_data(self, x, centers_emb, beta1, beta2):

```

```

        unconf_indices, conf_indices = self.generate_unconflicted_data_index(x,
centers_emb, beta1, beta2)
        return unconf_indices.shape[0], conf_indices.shape[0]

def random_transform(self, x, ws=0.1, hs=0.1, rot=10, scale=0.0):
    self.datagen = ImageDataGenerator(width_shift_range=ws, height_shift_range=hs,
rotation_range=rot, zoom_range=scale)
    if len(x.shape) > 2: # image
        return self.datagen.flow(x, shuffle=False, batch_size=x.shape[0]).next()

    # if input a flattened vector, reshape to image before transform
    width = int(np.sqrt(x.shape[-1]))
    if width * width == x.shape[-1]: # gray
        im_shape = [-1, width, width, 1]
    else: # RGB
        width = int(np.sqrt(x.shape[-1] / 3.0))
        im_shape = [-1, width, width, 3]
    gen = self.datagen.flow(np.reshape(x, im_shape), shuffle=False,
batch_size=x.shape[0])
    return np.reshape(gen.next(), x.shape)

#Refer Eq.(10) in the article
def generate_beta(self, kappa, n_clusters):
    beta1 = kappa / n_clusters
    beta2 = beta1 / 2
    print("Beta1 = " + str(beta1) + " and Beta2 = " + str(beta2))
    return beta1, beta2

def show_hcImages(self, x, n_hCIndex, image_shape=(28, 28)):
    clusters_hCIndex, _, _ = self.get_clusters_hCIndex(x, n_hCIndex)
    f, axarr = plt.subplots(self.n_clusters, n_hCIndex, figsize=(10, 10))
    for i in range(self.n_clusters):
        for j in range(n_hCIndex):
            image = x[int(clusters_hCIndex[i, j]),:].reshape(image_shape)
            axarr[i, j].imshow(image, cmap="gray", aspect="auto")
            axarr[i, j].get_xaxis().set_visible(False)
            axarr[i, j].get_yaxis().set_visible(False)
    plt.subplots_adjust(wspace=0., hspace=0.)
    plt.show()

def get_clusters_hCIndex(self, x, n_hCIndex):
    x_emb = self.encoder.predict(x)
    centers_emb, , , = self.generate_centers(x, self.n_clusters)
    q = q_mat(x_emb, centers_emb)
    conf = q.max(1)
    y_pred = q.argmax(1)
    clusters_hCIndex = np.zeros((self.n_clusters, n_hCIndex))
    clusters_index = [[] for i in range(self.n_clusters)]

```

```

clusters_conf = [[] for i in range(self.n_clusters)]
for i in range(x.shape[0]):
    clusters_index[y_pred[i]].append(i)
    clusters_conf[y_pred[i]].append(conf[i])
for i in range(self.n_clusters):
    clusters_index[i] = np.asarray(clusters_index[i])
    clusters_conf[i] = np.asarray(clusters_conf[i])
    for j in range(n_hCIndex):
        l = np.argsort(clusters_conf[i], axis=0)[- (j+1)]
        index = clusters_index[i][l]
        clusters_hCIndex[i][j] = index
return clusters_hCIndex, clusters_index, clusters_conf

```

Appendix F: Metrics Code

```
import numpy as np
from sklearn.metrics import normalized_mutual_info_score, adjusted_rand_score

nmi = normalized_mutual_info_score
ari = adjusted_rand_score

def cos_grad(grad1, grad2):
    grad1_list = []
    grad2_list = []
    for i in range(len(grad1)):
        grad1_list.append(grad1[i][0].flatten())
        grad2_list.append(grad2[i][0].flatten())
    grad1_vector = np.concatenate(grad1_list)
    grad2_vector = np.concatenate(grad2_list)
    return np.matmul(grad1_vector, grad2_vector) / ((np.linalg.norm(grad1_vector)) *
    (np.linalg.norm(grad2_vector)))

def acc(y_true, y_pred):
    """
    Calculate clustering accuracy. Require scikit-learn installed

    # Arguments
        y: true labels, numpy.array with shape `(n_samples,)`
        y_pred: predicted labels, numpy.array with shape `(n samples,)`

    # Return
        accuracy, in [0,1]
    """
    y_true = y_true.astype(np.int64)
    assert y_pred.size == y_true.size
    D = max(y_pred.max(), y_true.max()) + 1
    w = np.zeros((D, D), dtype=np.int64)
    for i in range(y_pred.size):
        w[y_pred[i], y_true[i]] += 1
    #from sklearn.utils.linear_assignment_ import linear_assignment
    from scipy.optimize import linear_sum_assignment
    row_ind, col_ind = linear_sum_assignment(w.max() - w)
    return sum([w[i, j] for i, j in zip(row_ind, col_ind)]) * 1.0 / y_pred.size
```

Appendix G: DEC Code

```
from time import time
import tensorflow as tf
tf.compat.v1.disable_eager_execution()
import matplotlib.pyplot as plt
import numpy as np
import tensorflow_probability as tfp
from sklearn.metrics import silhouette_score
from datasets import generate_data_batch, generate_transformed_batch, generate
import numpy as np
import tensorflow.keras.backend as K
from tensorflow.keras.layers import Layer, InputSpec, Input, Dense, Flatten, Lambda
from tensorflow.keras.models import Model
from tensorflow.keras import callbacks
from tensorflow.keras.initializers import VarianceScaling
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow_probability.python import distributions
from sklearn.cluster import KMeans
import metrics
import platform

PATH_RESULT = '/content/drive/My Drive/DEC/DEC_Sarra/results2'

def map_vector_to_clusters(y_true, y_pred):
    y_true = y_true.astype(np.int64)
    D = max(y_pred.max(), y_true.max()) + 1
    w = np.zeros((D, D), dtype=np.int64)
    for i in range(y_pred.size):
        w[y_true[i], y_pred[i]] += 1
    from scipy.optimize import linear_sum_assignment
    row_ind, col_ind = linear_sum_assignment(w.max() - w)

    y_true_mapped = np.zeros(y_pred.shape)
    for i in range(y_pred.shape[0]):
        y_true_mapped[i] = col_ind[y_true[i]]
    return y_true_mapped

def convert_to_sparse(y, n_clusters):
    y_sparse = np.zeros((y.shape[0], n_clusters))
    for i in range(y.shape[0]):
        for j in range(n_clusters):
            if j==y[i]:
                y_sparse[i, j] = 0.99
            else:
                y_sparse[i, j] = 0.01 / n_clusters
    return y_sparse
```

```

def kl(x, y):
    X = distributions.Categorical(probs=x)
    Y = distributions.Categorical(probs=y)
    return distributions.kl_divergence(X, Y)

def show_double(x1, x2, y, image_shape=(28, 28), nb_images=50):
    plt.figure(figsize=(40, 4))
    for i in range(nb_images):
        ax = plt.subplot(2, nb_images, i + 1)
        plt.imshow(x1[i].reshape(image_shape))
        plt.gray()
        plt.title(y[i])
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

        ax = plt.subplot(2, nb_images, i + 1 + nb_images)
        plt.imshow(x2[i].reshape(image_shape))
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
    plt.show()

def reset_weights(model):
    session = K.get_session()
    for layer in model.layers:
        if isinstance(layer, keras.engine.network.Network):
            reset_weights(layer)
            continue
        for v in layer.__dict__.values():
            if hasattr(v, 'initializer'):
                v.initializer.run(session=session)

def q_mat(X, centers, alpha=1.0):
    if X.size == 0:
        q = np.array([])
    else:
        q = 1.0 / (1.0 + (np.sum(np.square(np.expand_dims(X, 1) - centers), axis=2) /
alpha))
        q = q**((alpha+1.0)/2.0)
        q = np.transpose(np.transpose(q)/np.sum(q, axis=1))
    return q

class ClusteringLayer(Layer):
    def __init__(self, n_clusters, weights=None, alpha=1.0, **kwargs):
        if 'input_shape' not in kwargs and 'input_dim' in kwargs:
            kwargs['input_shape'] = (kwargs.pop('input_dim'),)
        super(ClusteringLayer, self).init(**kwargs)
        self.n_clusters = n_clusters

```

```

        self.alpha = alpha
        self.initial_weights = weights
        self.input_spec = InputSpec(ndim=2)

    def build(self, input_shape):
        assert len(input_shape) == 2
        input_dim = input_shape.as_list()[1]
        self.input_spec = InputSpec(dtype=K.floatx(), shape=(None, input_dim))
        self.clusters = self.add_weight(shape=(self.n_clusters, input_dim),
initializer='glorot_uniform', name='clusters')
        if self.initial_weights is not None:
            self.set_weights(self.initial_weights)
            del self.initial_weights
        self.built = True

    def call(self, inputs, **kwargs):
        """ student t-distribution, as same as used in t-SNE algorithm.
             $q_{ij} = 1/(1+dist(x_i, u_j)^2)$ , then normalize it.
        Arguments:
            inputs: the variable containing data, shape=(n_samples, n_features)
        Return:
            q: student's t-distribution, or soft labels for each sample.
        shape=(n_samples, n_clusters)
        """
        q = 1.0 / (1.0 + (K.sum(K.square(K.expand_dims(inputs, axis=1) -
self.clusters), axis=2) / self.alpha))
        q **= (self.alpha + 1.0) / 2.0
        q = K.transpose(K.transpose(q) / K.sum(q, axis=1))
        return q

    def compute_output_shape(self, input_shape):
        assert input_shape and len(input_shape) == 2
        return input_shape[0], self.n_clusters

    def get_config(self):
        config = {'n_clusters': self.n_clusters}
        base_config = super(ClusteringLayer, self).get_config()
        return dict(list(base_config.items()) + list(config.items()))

    def encoder_constructor(dims, act='relu'):
        n_stacks = len(dims) - 1
        init = VarianceScaling(scale=1. / 3., mode='fan_in', distribution='uniform')
        # input
        x = Input(shape=(dims[0],), name='input_encoder')
        h = x
        # internal layers in encoder
        for i in range(n_stacks-1):

```

```

        h = Dense(dims[i + 1], activation=act, kernel_initializer=init,
name='encoder_%d' % i)(h)
        # hidden layer
        h = Dense(dims[-1], kernel_initializer=init, name='encoder_%d' % (n_stacks -
1))(h) # hidden layer, features are extracted from here
        encoder = Model(inputs=x, outputs=h, name='encoder')
        return encoder

def decoder_constructor(dims, act='relu'):
    n_stacks = len(dims) - 1
    init = VarianceScaling(scale=1. / 3., mode='fan_in', distribution='uniform')
    # input
    z = Input(shape=(dims[-1],), name='input_decoder')
    y = z
    # internal layers in decoder
    for i in range(n_stacks-1, 0, -1):
        y = Dense(dims[i], activation=act, kernel_initializer=init, name='decoder_%d'
% i)(y)
    # output
    y = Dense(dims[0], kernel_initializer=init, name='decoder_0')(y)
    decoder = Model(inputs=z, outputs=y, name='decoder')
    return decoder

def ae_constructor(encoder, decoder, dims):
    x = Input(shape=(dims[0],), name='input_autencoder')
    autoencoder = Model(inputs=x, outputs=decoder(encoder(x)), name='autoencoder')
    return autoencoder

def dec_constructor(encoder, dims, n_clusters, alpha):
    out1 = ClusteringLayer(n_clusters, name='clustering', alpha=alpha)(encoder.output)
    dec = Model(inputs=encoder.input, outputs=out1, name='DEC')
    # Computing Feature Randomness and Feature Drift coefficients
    target1 = Input(shape=(n_clusters,), name='target1_dec')
    target2 = Input(shape=(n_clusters,), name='target2_dec')
    loss_pseudo_supervised = kl(out1, target1)
    loss_supervised = kl(out1, target2)
    grad_loss_pseudo_supervised = []
    grad_loss_supervised = []
    for i in range((len(dims) - 1) * 2):
        grad_loss_pseudo_supervised.append(tf.gradients(loss_pseudo_supervised,
dec.trainable_weights[i]))
        grad_loss_supervised.append(tf.gradients(loss_supervised,
dec.trainable_weights[i]))
    return dec, grad_loss_pseudo_supervised, grad_loss_supervised

def decoder_vl_constructor(encoder, decoder, dims):
    encoder.trainable = False
    decoder.trainable = True

```

```

input1 = Input(shape=(dims[0],), name='input_decoder_v1')

def loss_decoder_v1(x):
    input1 = x[0]
    x2 = x[1]
    loss_ae = tf.losses.mean_squared_error(input1, x2)
    return loss_ae

x1 = encoder(input1)
x2 = decoder(x1)
out1 = Lambda(loss_decoder_v1, name="output_decoder_v1")((input1, x2))
decoder_v1 = Model(inputs=[input1], outputs=[out1], name='decoder_v1')
return decoder_v1

def total_loss(y_true, y_pred):
    #Pass all filter
    return y_pred

class FcDEC(object):

    def __init__(self, dataset, dims, n_clusters=10, alpha=1.0):
        super(FcDEC, self).__init__()
        self.dataset = dataset
        self.dims = dims
        self.input_dim = dims[0]
        print('input_dim', self.input_dim)
        self.n_stacks = len(self.dims) - 1
        self.n_clusters = n_clusters
        self.alpha = alpha
        self.pretrained = False
        self.datagen = ImageDataGenerator(width_shift_range=0.1,
height_shift_range=0.1, rotation_range=10)
        #self.autoencoder, self.encoder = autoencoder(self.dims)
        print('dims', self.dims)
        self.encoder = encoder_constructor(self.dims)
        self.decoder = decoder_constructor(self.dims)
        self.autoencoder = ae_constructor(self.encoder, self.decoder, self.dims)
        self.dec, self.grad_loss_pseudo_supervised, self.grad_loss_supervised =
dec_constructor(self.encoder, self.dims, self.n_clusters, self.alpha)
        # prepare FcDEC model
        #clustering_layer = ClusteringLayer(self.n_clusters,
name='clustering')(self.encoder.output)
        #self.model = Model(inputs=self.encoder.input, outputs=clustering_layer)

    def create_decoder_v1(self, optimizer_decoder_v1):
        self.decoder_v1 = decoder_v1_constructor(self.encoder, self.decoder,
self.dims)
        self.compile_decoder_v1(optimizer=optimizer_decoder_v1)

```

```

def create_invertor(self, optimizer_invertor):
    self.invertor = invertor_constructor(self.encoder, image_shape=(1,
self.dims[0]))
    self.compile_invertor(optimizer=optimizer_invertor)

def load_weights(self, weights):
    # load weights of FcDEC model
    self.dec.load_weights(weights)

def extract_features(self, x):
    #
    return self.encoder.predict(x)

def predict(self, x):
    q = self.dec.predict(x, verbose=0)
    return q

def predict_labels(self, x):
    # predict cluster labels using the output of clustering layer
    return np.argmax(self.predict(x), 1)

@staticmethod
def target_distribution(q):
    weight = q ** 2 / q.sum(0)
    return (weight.T / weight.sum(1)).T

def random_transform(self, x, ws=0.1, hs=0.1, rot=10, scale=0.0):
    self.datagen = ImageDataGenerator(width_shift_range=ws, height_shift_range=hs,
rotation_range=rot, zoom_range=scale)
    if len(x.shape) > 2: # image
        return self.datagen.flow(x, shuffle=False, batch_size=x.shape[0]).next()

    # if input a flattened vector, reshape to image before transform
    width = int(np.sqrt(x.shape[-1]))
    if width * width == x.shape[-1]: # gray
        im_shape = [-1, width, width, 1]
    else: # RGB
        width = int(np.sqrt(x.shape[-1] / 3.0))
        im_shape = [-1, width, width, 3]
        #im_shape = [-1, 3, 3, 3]
    gen = self.datagen.flow(np.reshape(x, im_shape), shuffle=False,
batch size=x.shape[0])
    return np.reshape(gen.next(), x.shape)

def compile_dec(self, optimizer='sgd', loss='kld'):
    #
    self.dec.compile(optimizer=optimizer, loss=loss)

```

```

def compile_decoder_v1(self, optimizer='sgd'):
    #compile decoder_v1
    self.decoder_v1.compile(optimizer=optimizer, loss=total_loss)

def train_on_batch_dec(self, x, y, sample_weight=None):
    #
    return self.dec.train_on_batch(x, y, sample_weight)

def train_on_batch_decoder_v1(self, x):
    #train the decoder while the encoder weights are frozen
    y = np.zeros((x.shape[0],))
    return self.decoder_v1.train_on_batch(x, y)

def train_ae(self, x, y=None, optimizer='adam', epochs=200, batch_size=256,
save_dir=PATH_RESULT, verbose=1, aug_pretrain=False):
    print('Begin pretraining: ', '-' * 60)
    self.autoencoder.compile(optimizer=optimizer, loss='mse')

    csv_logger = callbacks.CSVLogger(save_dir + '/' + self.dataset +
'/pretrain/log.csv')
    # csv_logger = callbacks.CSVLogger(save_dir + '/log.csv')
    cb = [csv_logger]
    if y is not None and verbose > 0:
        class PrintACC(callbacks.Callback):
            def __init__(self, x, y, encoder):
                self.x = x
                self.y = y
                self.encoder = encoder
                super(PrintACC, self).__init__()

            def on_epoch_end(self, epoch, logs=None):
                if epoch % 20 != 0:
                    return
                features = self.encoder.predict(self.x)
                km = KMeans(n_clusters=len(np.unique(self.y)), n_init=20)
                y_pred = km.fit_predict(features)
                print(' '*8 + '|==>  acc: %.4f,  nmi: %.4f  <==|'%
(metrics.acc(self.y, y_pred), metrics.nmi(self.y, y_pred)))

        cb.append(PrintACC(x, y, self.encoder))

    # begin pretraining
    t0 = time()
    if not aug_pretrain:
        self.autoencoder.fit(x, x, batch_size=batch_size, epochs=epochs,
callbacks=cb, verbose=verbose)
    else:

```

```

print('--'*20)
print('Using augmentation for ae')
print('--'*20)
def gen(x, batch_size):
    if len(x.shape) > 2: # image
        gen0 = self.datagen.flow(x, shuffle=True, batch_size=batch_size)
        while True:
            batch_x = gen0.next()
            yield (batch_x, batch_x)
    else:
        width = int(np.sqrt(x.shape[-1]))
        if width * width == x.shape[-1]: # gray
            print("Hello-")
            im_shape = [-1, width, width, 1]
            print(im_shape)
        else: # RGB
            width = int(np.sqrt(x.shape[-1] / 3.0))
            im_shape = [-1, width, width, 3]
            print('hellow')
            #im_shape = [-1, 3, 3, 3]
        gen0 = self.datagen.flow(np.reshape(x, im_shape), shuffle=True,
batch_size=batch_size)
        print("Hello1")
        while True:
            batch_x = gen0.next()
            batch_x = np.reshape(batch_x, [batch_x.shape[0], x.shape[-1]])
            yield (batch_x, batch_x)

        self.autoencoder.fit_generator(gen(x, batch_size),
steps_per_epoch=int(x.shape[0]/batch_size),
                                epochs=epochs, callbacks=cb,
verbose=verbose)

    print('Pretraining time: ', time() - t0)
    self.autoencoder.save_weights(save_dir + '/' + self.dataset +
'/pretrain/ae_weights.h5')
    # self.autoencoder.save_weights(save_dir + '/ae_weights.h5')
    self.pretrained = True
    print('End pretraining: ', '-' * 60)

def train_dec(self, x, y=None, maxiter=2e4, batch_size=256, tol=1e-3,
update_interval=140, save_dir=PATH_RESULT, aug_cluster=False):
    print('Begin clustering:', '-' * 60)
    print('Update interval', update_interval)
    save_interval = int(maxiter) # only save the initial and final model
    print('Save interval', save_interval)

    # Step 1: initialize cluster centers using k-means
    t1 = time()

```

```

print('Initializing cluster centers with k-means.')
print(self.n_clusters)
kmeans = KMeans(n_clusters=self.n_clusters, n_init=20)
print("Helloo1")
features = self.encoder.predict(x)
print("Helloo2")
y_pred = kmeans.fit_predict(features)
print("Helloo3")
y_pred_last = np.copy(y_pred)
self.dec.get_layer(name='clustering').set_weights([kmeans.cluster_centers_])
print("Helloo4")
# Step 2: deep clustering
# logging file
import csv, os
if not os.path.exists(save_dir):
    os.makedirs(save_dir)
if y is not None:
    logfile = open(save_dir + '/' + self.dataset + '/cluster/log.csv', 'w')
    # logfile = open(save_dir + '/log.csv', 'w')
    #logwriter = csv.DictWriter(logfile, fieldnames=['iter', 'acc', 'nmi',
'ari', 'fr_dec', 'loss'])
    logwriter = csv.DictWriter(logfile, fieldnames=['iter', 'acc', 'nmi',
'ari', 'loss'])
else:
    logfile = open(save_dir + '/' + self.dataset + '/cluster/log_' +
str(self.n_clusters) + '.csv', 'w')
    # logfile = open(save_dir + '/log_' + str(self.n_clusters) + '.csv', 'w')
    logwriter = csv.DictWriter(logfile, fieldnames=['iter', 'silhouette_avg',
'loss'])

logwriter.writeheader()
loss = 0
index = 0
index_array = np.arange(x.shape[0])
#sess = tf.keras.backend.get_session()
for ite in range(int(maxiter)):
    if ite % update_interval == 0:
        q = self.predict(x)
        p = self.target_distribution(q) # update the auxiliary target
distribution p

        # evaluate the clustering performance
        y_pred = q.argmax(1)
        avg_loss = loss / update_interval
        loss = 0.
        #nb_unconf, nb_conf = self.compute_nb_conflicted_data(x, beta1[0],
beta2[0])

    if y is not None:
        y_mapped = map_vector_to_clusters(y, y_pred)

```

```

        y_mapped_sparse = convert_to_sparse(y_mapped, self.n_clusters)
        #grad_loss_pseudo_supervised =
sess.run(self.grad_loss_pseudo_supervised, feed_dict={'input_encoder:0': x,
'target1_dec:0': p})
        #grad_loss_supervised = sess.run(self.grad_loss_supervised,
feed_dict={'input_encoder:0': x, 'target2_dec:0': y_mapped_sparse})
        acc = np.round(metrics.acc(y, y_pred), 5)
        nmi = np.round(metrics.nmi(y, y_pred), 5)
        ari = np.round(metrics.ari(y, y_pred), 5)
        #FR_dec = np.round(metrics.cos_grad(grad_loss_supervised,
grad_loss_pseudo_supervised), 5)
        #logdict = dict(iter=ite, acc=acc, nmi=nmi, ari=ari,
fr_dec=FR_dec, loss=avg_loss)
        logdict = dict(iter=ite, acc=acc, nmi=nmi, ari=ari, loss=avg_loss)
        logwriter.writerow(logdict)
        logfile.flush()
        #print('Iter %d: acc=%.5f, nmi=%.5f, ari=%.5f, loss=%.5f,
nb_conf=%d' % (ite, acc, nmi, ari, avg_loss, nb_conf))
        print('Iter %d: acc=%.5f, nmi=%.5f, ari=%.5f, loss=%.5f' % (ite,
acc, nmi, ari, avg_loss))
        #print("|==> DEC Feature Randomness: %.5f <==|" % (FR_dec))
    else:
        x_embed = self.encoder.predict(x)
        silhouette_avg = silhouette_score(x_embed, y_pred)
        print("For number of clusters = " + str(self.n_clusters) + " The
average silhouette_score is :", silhouette_avg)
        logdict = dict(iter=ite, silhouette_avg=silhouette_avg,
loss=avg_loss)
        logwriter.writerow(logdict)
        logfile.flush()
        # check stop criterion
        delta_label = np.sum(y_pred != y_pred_last).astype(np.float32) /
y_pred.shape[0]
        y_pred_last = np.copy(y_pred)
        if ite > 0 and delta_label < tol:
            print('delta_label ', delta_label, '< tol ', tol)
            print('Reached tolerance threshold. Stopping training.')
            logfile.close()
            break
        #if (nb_conf / x.shape[0]) < tol:
        #    logfile.close()
        #    break

    # save intermediate model
    if ite % save_interval == 0:
        print('saving model to:', save_dir + '/cluster/dec_weights.h5')
        self.dec.save_weights(save_dir + '/' + self.dataset +
'/cluster/dec_weights.h5')

```

```

        self.autoencoder.save_weights(save_dir + '/' + self.dataset +
'/cluster/ae_weights.h5')
        # self.dec.save_weights(save_dir + '/dec_weights.h5')
        # self.autoencoder.save_weights(save_dir + '/ae_weights.h5')
        # train on batch
        idx = index_array[index * batch_size: min((index+1) * batch_size,
x.shape[0])]
        x_batch = self.random_transform(x[idx]) if aug_cluster else x[idx]
        loss += self.train_on_batch_dec(x=x_batch, y=p[idx])
        index = index + 1 if (index + 1) * batch_size <= x.shape[0] else 0
        ite += 1

    # save the trained model
    logfile.close()
    print('saving model to:', save_dir + '/' + self.dataset +
'/cluster/ae_weights.h5')
    self.autoencoder.save_weights(save_dir + '/' + self.dataset +
'/cluster/ae_weights.h5')
    self.dec.save_weights(save_dir + '/' + self.dataset +
'/cluster/dec_weights.h5')
    # print('saving model to:', save_dir + '/ae_weights.h5')
    # self.autoencoder.save_weights(save_dir + '/ae_weights.h5')
    # self.dec.save_weights(save_dir + '/dec_weights.h5')
    exec_time = time() - t1
    print('Clustering time: %ds' % (exec_time))
    #f = open(save_dir + '/' + self.dataset + '/cluster/results_train_adec.txt',
'w+')
    #f.write('|==> ACC: %.4f, NMI: %.4f, FR_DEC: %.4f <==| \n' % (metrics.acc(y,
y_pred), metrics.nmi(y, y_pred), FR_dec))
    #f.write('|==> ACC: %.4f, NMI: %.4f <==| \n' % (metrics.acc(y, y_pred),
metrics.nmi(y, y_pred)))
    #f.write('|==> Execution time: %.5f <==| \n' % (exec_time))
    #f.close()
    print('End clustering:', '-' * 60)
    return y_pred

def train_decoder_v1(self, x, y=None, n_clusters=10, maxiter=2e4, batch_size=256,
image_shape=(28, 28), save_dir=PATH_RESULT, update_interval=140, save_interval=None,
show_interval=None, aug_train=False):
    if save_interval is None:
        save_interval = int(maxiter) # only save the initial and final model
        print('Save interval', save_interval)

    if show_interval is None:
        show_interval = int(np.ceil(x.shape[0]/batch_size))*10
        print('show interval ', show_interval)

```

```

        show = self.show_images(x, y, image_shape=image_shape, nb_images=50,
aug=aug_train)
        t1 = time()

        # Create the logging file
        import csv, os
        if not os.path.exists(save_dir):
            os.makedirs(save_dir)
        logfile = open(save_dir + '/' + self.dataset + '/train_decoder/log.csv', 'w')
        # logfile = open(save_dir + '/log.csv', 'w')
        logwriter = csv.DictWriter(logfile, fieldnames=['iter', 'loss'])
        logwriter.writeheader()

        loss = 0
        index = 0
        index_array = np.arange(x.shape[0])
        for ite in range(int(maxiter)):
            if ite % update_interval == 0:
                # evaluate the reconstruction loss
                avg_loss = loss / update_interval
                loss = 0
                print('Iter= %d, loss=%.5f' % (ite, avg_loss))

            # save the final result
            if ite % save_interval == 0:
                print('saving model to:', save_dir + '/' + self.dataset +
'/train_decoder/decoder.h5')
                self.decoder.save_weights(save_dir + '/' + self.dataset +
'/train_decoder/decoder.h5')
                # print('saving model to:', save_dir + '/decoder.h5')
                # self.decoder.save_weights(save_dir + '/decoder.h5')

            if ite % show_interval == 0 and ite!=0:
                print("")
                print("-----")
                print("Images Construction : ")
                print("-----")
                next(show)
                print("")

            # train on batch
            idx = index_array[index * batch_size: min((index+1) * batch_size,
x.shape[0])]
            x_batch = self.random_transform(x[idx]) if aug_train else x[idx]
            loss += self.train_on_batch_decoder vl(x=x_batch)
            index = index + 1 if (index + 1) * batch_size <= x.shape[0] else 0

```

```

        ite += 1

    # save the trained model
    logfile.close()
    print('saving model to:', save_dir + '/' + self.dataset +
          '/train_decoder/decoder.h5')
    self.decoder.save_weights(save_dir + '/' + self.dataset +
                              '/train_decoder/decoder.h5')
    # print('saving model to:', save_dir + '/decoder.h5')
    # self.decoder.save_weights(save_dir + '/decoder.h5')
    print('The time requiried to train the decoder: %ds' % (time() - t1))
    print('End training:', '-' * 60)

def train_invertor(self, x, maxiter=10e5, initial_shape=(1, 784),
save_dir=PATH_RESULT, show_interval=10e4, sigma_tv=5e-7, ws=0.1, hs=0.1, rot=10,
scale=0.0):
    self.encoder.trainable = False
    self.encoder.compile(optimizer='sgd', loss='mse')
    input1 = self.encoder.input
    self.encoder.load_weights(save_dir + '/' + self.dataset +
                              '/cluster/encoder_weights.h5')
    # self.encoder.load_weights(save_dir + '/encoder_weights.h5')
    X = tf.Variable(tf.zeros(initial_shape))
    z1 = self.encoder(input1)
    l2_loss = tf.reduce_sum(tf.norm(self.encoder(X) - z1, 'euclidean', axis=1) /
tf.norm(z1, 'euclidean', axis=1))
    conv_shape = convert_shape_fc_to_conv(initial_shape)
    tv_loss =
tf.reduce_sum(tf.image.total_variation(tf.reshape(tf.convert_to_tensor(X),
conv_shape)))
    loss = l2_loss + sigma_tv * tv_loss
    train_step = tf.train.AdamOptimizer(learning_rate=0.01).minimize(loss)
    x = self.random_transform(x, ws, hs, rot, scale)
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        for i in range(maxiter):
            _, loss_value = sess.run([train_step, loss],
feed_dict={self.encoder.input: np.reshape(x, initial_shape)})
            if i % show_interval == 0:
                print("Loss: ", loss_value)
        O = sess.run(X)
    return O

def compute_nb_conflicted_data(self, x, beta1, beta2):
    centers_emb = self.dec.get_layer(name='clustering').get_weights()[0]
    unconf_indices, conf_indices = self.generate_unconflicted_data_index_local(x,
centers_emb, beta1, beta2)
    return unconf_indices.shape[0], conf_indices.shape[0]

```

```

def generate_unconflicted_data_index_local(self, x_img, centers_emb, beta1,
beta2):
    x_emb = self.encoder.predict(x_img)
    unconf_indices = []
    conf_indices = []
    q = q_mat(x_emb, centers_emb, alpha=1.0)
    confidancel = q.max(1)
    confidence2 = np.zeros((q.shape[0],))
    a = np.argsort(q, axis=1)[:,-2]
    for i in range(x_img.shape[0]):
        confidence2[i] = q[i,a[i]]
        if (confidancel[i] > beta1 and (confidancel[i] - confidence2[i]) > beta2:
            unconf_indices.append(i)
        else:
            conf_indices.append(i)
    unconf_indices = np.asarray(unconf_indices, dtype=int)
    conf_indices = np.asarray(conf_indices, dtype=int)
    return unconf_indices, conf_indices

def show_images(self, x, y, image_shape=(28, 28), nb_images=50, aug=True):
    gen0 = generate_data_batch(x=x, y=y)
    gen1 = generate_transformed_batch(x=x, datagen=self.datagen)
    np.random.seed(0)
    while True:
        X1, Y = next(gen0)
        X2 = next(gen1)
        p = np.random.permutation(X1.shape[0])
        X1 = X1[p]
        X2 = X2[p]
        Y = Y[p]
        imgs_recons = self.autoencoder.predict(X1)
        plt.figure(figsize=(40, 4))
        for i in range(nb_images):
            if aug is False:
                # display original
                ax = plt.subplot(2, nb_images, i + 1)
                plt.imshow(X1[i].reshape(image_shape))
                plt.gray()
                plt.title(Y[i])
                ax.get_xaxis().set_visible(False)
                ax.get_yaxis().set_visible(False)

                # display reconstruction
                ax = plt.subplot(2, nb_images, i + 1 + nb_images)
                plt.imshow(imgs_recons[i].reshape(image_shape))
                plt.gray()
                ax.get_xaxis().set_visible(False)

```

```

        ax.get_yaxis().set_visible(False)
    else:
        # display original
        ax = plt.subplot(3, nb_images, i + 1)
        plt.imshow(X1[i].reshape(image_shape))
        plt.gray()
        plt.title(Y[i])
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

        # display transformed
        ax = plt.subplot(3, nb_images, i + 1 + nb_images)
        plt.imshow(X2[i].reshape(image_shape))
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

        # display reconstruction
        ax = plt.subplot(3, nb_images, i + 1 + 2*nb_images)
        plt.imshow(imgs_recons[i].reshape(image_shape))
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
    plt.show()
    yield

def compute_acc_and_nmi(self, x, y):
    features = self.encoder.predict(x)
    km = KMeans(n_clusters=len(np.unique(y)), n_init=20)
    y_pred = km.fit_predict(features)
    print(' '*8 + '|==>  acc: %.4f,  nmi: %.4f  <==|'% (metrics.acc(y, y_pred),
metrics.nmi(y, y_pred)))

```

الملخص

يعتبر استهلاك الطاقة أحد الموضوعات الأساسية لأنظمة الطاقة التي اكتسبت اهتماما كبيرا بعد أزمة الطاقة في عام 1970. لقد ثبت أن استهلاك الطاقة في جميع أنحاء العالم يتزايد بسرعة كما أن مصادر الطاقة المتجددة في أنظمة طاقة زيادة الاختراق يزيد من الحاجة إلى تعزيز إدارة جانب الطلب. من ثم حظيت أبحاث ملفات الطلب على الكهرباء باهتمام متزايد من سلطات المرافق والمنظمات والباحثين لإدارة جانب الطلب مثل برامج الاستجابة للطلب.

يعتبر مجال تحليل البيانات للعدادات الذكية مجالا حديثا نسبيا نما مؤخرا نظرا لأهمية البيانات الناتجة عن العدادات الذكية. في الآونة الأخيرة ، أتاح التقدم في الحوسبة عالية الأداء استخدام الشبكات العصبية العميقة بكفاءة وفعالية لاستخراج المزيد من المعرفة والأنماط المفيدة من البيانات الضخمة التي تم جمعها داخل الشبكة الذكية. ومن ثم أصبح من الأهمية بمكان تحليل اتجاهات استهلاك الطاقة.

يهدف هذا البحث إلى نشر خوارزمية تجميع عميقة مضمنة مع مشفر تلقائي ومقارنتها بالتشفير التلقائي الديناميكي أثناء تطبيق تقليل الأبعاد وتقييم تلك النماذج بناء على الأداء باستخدام درجة الصورة الظلية. تفوق جهاز التشفير التلقائي الديناميكي على نماذج التجميع DEC و IDEC وحقق درجة صورة ظلية تبلغ 0.9931 مقارنة ب DEC تبلغ 0.6128 و IDEC تبلغ 0.8829. يشير هذا إلى أن نشر نموذج تجميع عميق مضمن في برنامج تشفير تلقائي ديناميكي قد عمل بشكل فعال ضد مشكلة انحراف الميزة والعشوائية.

جامعة البحرين

برامج الدراسات العليا

كلية العلوم



التنبؤ باستهلاك الكهرباء باستخدام طريقة تجميع توصيف الأحمال

إطروحة مقدمة كجزء من متطلبات الحصول على درجة الماجستير
في علوم وتحليل البيانات الضخمة

إعداد

اسراء جاسم الجودر

الرقم الجامعي

20021155

إشراف

د . رياض قسطيني
أستاذ مشارك

جامعة البحرين

د . سوسن هلال
أستاذ مساعد

جامعة البحرين

مملكة البحرين
مارس 2023