A HYBRID METHODOLOGY FOR INVENTORY CLASSIFICATION WITH THE APPLICATION OF MACHINE LEARNING ALGORITHMS

Anik Roy	**201736002**
Ashiqur Rahman Khan	**201736004**
MD. Fahim Montasir	**201736022**

DEPARTMENT OF INDUSTRIAL AND PRODUCTION ENGINEERING MILITARY INSTITUTE OF SCIENCE AND TECHNOLOGY (MIST)

MIRPUR CANTONMENT, DHAKA 1216, BANGLADESH

Date of submission (13 August, 2020)

A Hybrid Methodology for Inventory Classification with the Application of Machine Learning Algorithms

by

Name	**Roll**
Anik Roy	**201736002**
Ashiqur Rahman Khan	**201736004**
MD. Fahim Montasir	**201736022**

A Thesis

Submitted to the

Department of Industrial and Production Engineering

In Partial Fulfillment of the

Requirements for the Degree

Of

BACHELOR OF SCIENCE IN INDUSTRIAL & PRODUCTION ENGINEERING

Department of Industrial and Production Engineering
Military Institute of Science and Technology (MIST)
Mirpur Cantonment, Mirpur, Dhaka-1216

This thesis work entitled **A Hybrid Methodology for Inventory Classification with the Application of Machine Learning Algorithms,** submitted by the following students, has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B. Sc. in Industrial and Production Engineering in **March 2021**.

Names and Roll number of the students

Student 1 Name: Anik Roy

Roll No: 201736002

Student 2 Name: Ashiqur Rahman Khan

Roll No: 201736004

Student 3 Name: MD. Fahim Montasir

Roll No: 201736022

Name of the supervisor: Dr. Kais Bin Zaman

Designation of supervisor: Professor

Department of Industrial and Production Engineering

Bangladesh University of Engineering and Technology (BUET)

Shahbag, Dhaka

Dhaka-1000, Bangladesh.

Declaration

We do hereby declare that we have done this thesis work, and neither this thesis nor any part of it has been submitted elsewhere for the award of any degree or diploma.

Signature of students

Signature of Student 1: ___________________________

Signature of Student 2: ___________________________

Signature of Student 3: ___________________________

____

Name of the supervisor: Dr. Kais Bin Zaman
Designation of supervisor: Professor
Department of Industrial and Production Engineering
Bangladesh University of Engineering and Technology (BUET)
Shahbag, Dhaka
Dhaka-1000, Bangladesh.

Contents

List of Tables

List of Figures

Acknowledgment

At first, we are thankful and would like to show our heartful gratitude to our esteemed supervisor Professor Kais Bin Zaman, Department of Industrial and Production Engineering, BUET, for his wholehearted supervision. He helped us to understand the necessity of the thesis, and his advice and instructions made us go for the development of the concepts, implement the ideas and theories, and gain some practical knowledge about inventory management and classifying the inventory items based on the priority of different aspects. He has given us his obvious and distinct information about Machine Learning Algorithm, Machine Learning Classification Techniques, Multi-Criteria Decision-Making Method, etc. His invaluable encouragement and special instructions from the very beginning to the report writing and entire work helped out a lot prevailed in us overcome all the obstacles and hurdles to complete the thesis.

Here, we like to extend our sincere gratitude to some people without whom it would not be possible for us to gain some emphatic ideas and detailed knowledge about the various Machine Learning techniques and also coding in PYTHON.

We are likewise extremely grateful to regarded personnel of the IPE department of MIST, who offered us their significant time and guidance.

Abstract

The motivation behind this study is to build up a hybrid approach that incorporates ABC analysis, Machine Learning (ML) algorithms with Multi-Criteria Decision-Making (MCDM) techniques to adequately direct multi-attribute inventory analysis. In our approached methodology, firstly, we will do ABC analysis, which is a famous and compelling strategy used to classify inventory items into explicit categories, based on Pareto distribution using different MCDM methods (i.e., Simple Additive Weighting, Analytical Hierarchy Process, and VIKOR) which are employed to determine the appropriate class for each of the inventory items. Succeeding this, Artificial Neural Network (ANN), Gaussian Process Classification (GPC), K-Nearest Neighbors (KNN), and Support Vector Machine (SVM) algorithms are actualized to anticipate classes of initially decided inventory items. Then, the accuracies of algorithms were determined for each method. The outcomes showed that AI-based methods exhibit overall exactness to MCDM. The results additionally uncovered that ANNs, GPCs, KNNs, and SVMs are, on the whole, prepared to effectively manage the unbalanced data issues related to Pareto distribution, and each of these algorithms performed well against all analyzed measures, along these lines approving the way that ML algorithms are exceptionally pertinent to inventory analysis issues. There are a few numbers of research where MCDM methods are combined with machine learning algorithms effectively and applied to various inventory settings. Moreover, this study gives an extension of the previous researches by introducing different machine learning algorithms and the classification of existing inventory items with the assistance of combining the classes of three MCDM methods into one and the accuracy comparison of machine learning models.

Chapter 1

Introduction

Efficient inventory management has assumed a significant job in the accomplishment of supply chain management. For industries that keep up a great number of inventory items, it is quite impossible to give equivalent thought to every inventory item. Managers are needed to classify these things to fittingly control each inventory class, as indicated by its significance rating.

ABC analysis is one of the most commonly utilized inventory classification procedures. Customary ABC classification was introduced for use by General Electric during the 1950s. The classification plot depends on the Pareto rule or the 80/20 principle, which utilizes the accompanying dependable guideline: "vital few and trivial many." The procedure of ABC analysis groups inventory items into A, B, or C classifications depending on the alleged annual dollar usage. Inventory items are then organized by the descending order of their annual dollar usage. Class A items are generally little in number, yet represent the best sum of annual dollar usage. Conversely, class C items are usually huge in number, yet make up a somewhat limited quantity of annual dollar usage. Items between classes A and C are arranged as class B.

Despite the fact that ABC analysis is renowned for its convenience, it has been scrutinized for its restrictive spotlight on dollar usage. Other features such as manufacturing cost, fixed cost, selling price, raw material availability have likewise been perceived as essential for inventory management. In our research, we are going to classify the inventory items considering these issues using ABC Analysis. ABC analysis is an essential method in inventory management. By definition, ABC analysis does the categorization of items into three categories (A, B, and C) to determine levels of significance of an item. Thus, the inventory is assembled into three classifications (A, B, and C) arranged by their assessed significance. The primary utilization of ABC analysis is to improve the capacity to manage enormous and complex data indexes by breaking them down into three sections. These sections characterize the need for the data inside whatever zone we are utilizing them in. When the information is separated into sections, it is simpler to concentrate on the data and use it in a meaningful manner. Breaking down the data into these sections makes the data more self-evident. It additionally helps in organizing the

various segments. ABC analysis is, likewise, a phenomenal tool for inventory control. It is especially helpful for figuring out which of the inventory things sway your stock expense the most. It also gives a system to decide the ideal approaches to oversee and control the inventory.
In this report, MCDM (Multi-Criteria Decision-Making) methods will be applied to the multiple features (i.e., demand, manufacturing cost, raw material availability, selling price, fixed cost) of the inventory item dataset. Then ABC analysis for the existing inventory items will be conducted based on the performance of each item gained from MCDM models. Later, the future prediction will be done by Machine Learning Techniques where the features will be the input and the classes of ABC Analysis will be the label.
Most of the MCDM methods were focused on weight determination. It contains simple additive weighting (SAW), Analytic Hierarchy Process (AHP), VIšekriterijumsko KOmpromisno Rangiranje (VIKOR), fuzzy conjunctive/disjunctive methods, fuzzy outranking methods, and max-min methods. It appears that the methods of SAW, AHP, and VIKOR are the MCDM methods utilized for weight determinations and inclinations.
Simple Additive Weighting (SAW), which is also known as weighted linear combination or scoring methods, is a simple and most often used multi-attribute decision technique. The method is based on the weighted average.
According to Triantaphyllou and Mann[1], the excellent mathematical properties of AHP have attracted many researchers' interest. The Analytical Hierarchy Process (AHP), created under Multi-Criteria Decision Making (MCDM) method, is composed of suitable techniques for ranking critical management problems. The Analytic Hierarchy Process (AHP) was introduced by Saaty [2] in the 1970s. The AHP method is a ranking process that is used in making group decisions and is widely used around the world in a variety of fields such as business, inventory management, government, industry, education, health, and others.
The VIKOR method was introduced as an applicable technique to implement as an MCDM model. It focuses on ranking and selecting from a set of alternatives in the presence of conflicting criteria and to achieve a compromised solution. The compromise solution is a feasible solution that is the closest to the ideal solution, and a compromise means an agreement established by mutual concession. [3]
Artificial Neural Network is the combination of many mathematical models and research. It mimics the human brain through a set of algorithms. It is a supervised machine learning algorithm inspired by the connections of neurons of animal brains. An ANN is based on a collection of connected units or nodes called artificial neurons. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron

receives real number data as input and passes the data through a non-linear function known as node. It is used both in regression and classification problems. ANN consists of four components inputs, weights, a bias, & an output, and also multiple layers known as the input, hidden, and output layer. Each layer consists of multiple nodes named neurons and the summation of nodes of each layer is the input of the next layer. Backpropagation is done to determine the error of ANN. In this era, the use of ANN in industrial automation is increasing day by day. [4]

Support Vector Machine (SVM) is an ML algorithm with the capability of solving classification and regression problems. In SVM hyperplanes are used for a better understanding of the data points. A hyperplane with the largest distance to the nearest training data point of any class is considered as a good separation plane. This distance is also called the functional margin, it is considered that if the margin is large then the generalization error of the classifier will be low. SVM with a good separation plan gives higher accuracy.[5]

Later, models dependent on the Gaussian process (GP) priors have pulled in much consideration in the AI community. Gaussian processes are attractive models for probabilistic classification, but unfortunately, exact inference is analytically intractable. Though surmising in the GP regression model with Gaussian noise should be possible scientifically, probabilistic arrangement utilizing GPs is logically unmanageable. [6]

K-Nearest Neighbors (K-NN) algorithm is a non-parametric classification method. It was first developed by Evelyn Fix and Joseph Hodges in 1951,[7] later on expanded by Thomas Cover [8]. KNN is a supervised machine learning algorithm, can solve regression and classification problems. KNN classifies data points based on the points that are most similar to them. The similarities of data points are measured by the Euclidean distance from a point to point. Data points with similar properties or values are considered in the same class. This" K" or the number of classes is defined by the user.

The rest of the report is organized as follows. In Chapter 2, previous research work on different machine learning techniques, and MCDM methods are discussed. The techniques –ABC Analysis, Simple Additive Weighting (SAW), Analytical Hierarchy Process (AHP), VIKOR, Support Vector Machine (SVM), K-Nearest Neighbors (KNN), Artificial Neural Network (ANN), and Gaussian Process Classification (GPC) are explained in Chapter 3. The proposed methodology is described in Chapter 4. In Chapter 5, the sources, characteristics, and features of the inventory items datasets are described thoroughly. In Chapter 6, we examine the datasets with three different MCDM methods and four other machine learning techniques and determine the performance measures. The machine learning techniques that are better used in the multi-criteria inventory classification have been discussed in Chapter 7.

Chapter 2

Literature review

The application of MCDM methods with machine learning techniques related to classifying inventory items was discussed in several research papers. Various works have been done based on this topic. In this chapter, we will probably draw out all conditions of craftsmanship by multiple authors and analysts.

F. Lolli et al. [9] proposed that supervised machine learning classifiers could be applied in cutting-edge inventory classification systems, as their trial analysis of two large datasets indicated a brilliant precision. Multi-Criteria Inventory Classification (MCIC) methods, Deep Neural Network (DNN), and Support Vector Machine (SVM) were utilized as inventory classifiers. SVM with Gaussian kernel and DNN results showed significant improvements in the classification accuracy in comparison with a theoretical approach.

A pioneering contribution to Inventory Classification was provided by Flores, Olson, and Dorai (1992). [10] They applied the Analytical Hierarchy Process (AHP) to classify items in terms of average unit cost, criticality, annual usage value, and lead-time. This represented one of the early attempts to overcome the weakness shown by the mono-criterion classification on usage value in describing the whole criticality of an item. This paper proposed the utilization of the Analytic Hierarchy Process (AHP) to lessen this multi-criterion to a univariate and predictable measure to consider multiple inventory management focuses.

Ramanathan et al. proposed a simple classification scheme in their paper using weighted linear optimization. The proposed model is similar to linear programming models employed in data envelopment analysis (DEA). Later on, they compared the result of the proposed model result with the Conventional ABC Analysis outcome and MCDM (i.e., AHP) outcome. [11]

Sabaei et al. [12] proposed a critical comparative analysis of different MCDM methods (AHP, PROMETHEE (Preference Ranking Organization Method for Enrichment of Evaluations), ELECTRE(ELimination Et Choice Translating REality)) from the maintenance point of view. A set of criteria (Data type, Uncertainty, Outranking method) was proposed by the authors to evaluate the methods. Then the above MCDM methods were evaluated based on the criteria and hence provided a framework for the selection approaches to improve their decision efficiency and effectiveness.

Opricovic et al. [13] had shown a comparison analysis between VIKOR and TOPSIS (Technique for Order of Preference by Similarity to Ideal Solution) based on their approach to "closeness to ideal" behavior. Linear and vector normalization was used to eliminate the units of the criterion functions respectively for VIKOR and TOPSIS. A compromise solution was determined for the VIKOR method, providing a maximum 'group utility' for the 'majority and a minimum of individual regret of the 'opponent'. Thus, the normalized value in the VIKOR method did not depend on the evaluation unit of a criterion function.

Kartal and Cebi [14] investigated the SVMs (i.e., Poly kernel SVM and Normalised Poly kernel SVM) performance in inventory classification based on multi-attributes. They used the ABC analysis utilizing the Simple Additive Weighting (SAW) technique was used to decide inventory classes of items held in the inventory of an enormous automobile company.

Wan Lung Ng [15] discussed a model which is relative to the MCDM model 'SAW.' Later on, this model was compared to the DEA-like model and traditional ABC classification model. This model can be adopted directly if the categorical measures of the data are converted to continuous measures.

Partovi et al. [16] utilized ANNs for ABC classification of Stock Keeping Units(SKUs) in a pharmaceuticals company. . To classify inventory items, two learning methods were used, namely, BP (backpropagation) and GA (genetic algorithm). The reliability of the model was tested by two different datasets and also compared with the multiple discriminate analysis (MDA) technique. It was observed that the ANN had more accuracy than MDA. Besides, the results had shown us a very small difference between the two models (BP and GA).

Marcello Bragila et al. [17] proposed a methodology based on two techniques named after RCM Analysis and AHP model, which lead the methodology to an inventory management policy matrix. To start with, likewise to RCM analysis, a decision outline drives the researcher towards the best criticality classification of the items through a set of choice nodes. To help the researcher during the decision in each node, several (reduced) AHP models are proposed. Second, to join a proper inventory strategy for each class, an inventory management policy matrix is created.

Kumar et al. [18] have given us the progression of applying different MCDM techniques (SAW, AHP, VIKOR, TOPSIS, ELECTRE, PROMETHEE, etc.) towards sustainable renewable energy. Multiple indicators (social, technical, organizational, economical, environmental) were identified by the authors which were evaluated by the MCDM models.

Strengths and weaknesses were also identified in each MCDM model. But no proper technique was found in their analysis. A hybrid MCDM model was suggested to consider all the indicators.

Asadabadi et al. [19] focused on the argument of the efficiency of MCDM methods. They have done a detailed analysis of AHP and ANP (a version of the AHP) method. They have shown a real-life scenario and how to rank it with the help of AHP. After completing the analysis, they have found this method is less efficient. Hence, we want to apply machine learning algorithms to make this model more efficient and observe how this model fits into real-life problems.

Ali Jahan et al. [20] proposed a new version of the VIKOR method, which covers all types of criteria. They have focused on the material selection process. They eliminated some numerical difficulties of the traditional VIKOR method by using a novel normalization technique. They have demonstrated their theory on five different datasets and compared it with the conventional VIKOR way. From this paper, we get to understand the variation and difficulties of the VIKOR method and how it can be removed.

Zied Babai et al. [21] utilized five ML techniques to perform inventory classification and to decrease the class imbalance caused by six multi-criteria inventory classification techniques. The viability of these ML techniques was surveyed through three measures: class distribution, accuracy, and total inventory cost. They had tended to the imbalanced class issue that emerges in inventory classification by proposing another algorithm targeting adjusting ML classifiers to the specific attribute of inventory data. At the research work, They found that a good accuracy doesn't methodically mean a good class distribution, and a similar precision esteems can be related to various inventory costs.

Stelios H. Zanakis et al. [22] have shown us the comparative analysis of different MCDM methods on the same datasets. They have focused on the results of the various ways of analysis on the same datasets and the variation of the products. They have worked with eight types of plans: ELECTRE, TOPSIS, Multiplicative Exponential Weighting (MEW), Simple Additive Weighting (SAW), and four versions of AHP. From this paper, we have understood the results of different types of analysis may vary if we apply them to the same datasets.

Hedi Cherif et al. [23] presented another methodology for the ABC Multi-Criteria Inventory Classification issue dependent on the Artificial Bee Colony (ABC) algorithm with the Multi-Criteria Decision-Making technique, specifically VIKOR. The principal commitment of the proposed work was to exploit the effectiveness of the ABC algorithm and the strategy VIKOR on a half and half to group the inventory items dependent on target weights and to diminish the inventory cost. In the end, they just compared their outcomes with other researchers' proposed models' outcomes who worked with the same data set but using a different multi-criteria inventory classification model.

Cover and Hart [24] presented that K-NN is a non-parametric procedure for classifying perceptions. Calculations of the measure of distance or similitude between perceptions are directed prior to classification. A recently presented thing is at that point classified to the group where most of the k-NNs have a place. The utilization of k-NN requires a fitting value of k.

Loftgaarden and Queensberry [25] recommended that a sensible k could be gotten by ascertaining the square root of the number of perceptions in a group. A trial and error technique may be more viable in distinguishing the value of k that acquires the lowest classification blunder. [26] A sensitivity analysis can be led to analyze the characterization precision among different estimations of k to reason that an estimation of 3 gives that most noteworthy correct classification.

Amir Atiya et al. [27] have proposed a new Monte Carlo-based algorithm for the Gaussian process classification problem. When GPC is applied to classification problems, intractable integrals are encountered which can only be solved by some approximations or using lengthy algorithms. This problem has shrunk the opportunity to apply the GPC on a large-scale dataset despite having very good results. Based on some bootstrap sampling and acceptance-rejection they developed a new Monte Carlo algorithm that converted the problem into a ratio of two orthant integrals of a multivariate Gaussian density.

Chapter 3

Techniques

There are several techniques to classify inventory items. ABC Analysis is a traditional way to approach the classification of inventory items. ABC analysis is a methodology for classifying inventory items depending on the item's annual costs. ABC analysis is a straightforward structure to work out which inventory items are the most significant and should subsequently devour a large portion of time regarding stock control and management. Basically, ABC Analysis works with a single feature (annual cost). Therefore, it has been an issue for modern inventory management. Here come the MCDM Methods, which functions with multi-attributes to identify the classification of inventory items. Some of the additional MCDM techniques have been developed as a method for supporting ABC analysis. They have become mainstream as the immediate result of their capacity to consolidate additional criteria into inventory management, including manufacturing cost, selling cost, demand, fixed cost, raw material availability, etc. MCDM techniques usually generate priority performance for every single item. Though ABC Analysis can work with a single attribute, so in this report, the priority performance has been used as that single attribute to classify the items. Then for future class prediction, Machine learning techniques have been implemented. As of now, we live in the crude period of machines while the future of machines is vast and is beyond the scope of imagination, it is the need of time for the machines to begin learning the pattern of the work from their experience. Machine learning is a technique for data analysis that automates the analytical model structure. It is a part of artificial intelligence that frameworks can learn from data, recognize patterns, and make choices with insignificant human inclusion. The center of machine learning and the way to Artificial Intelligence lies in perceiving and separating any pattern by the program itself with no human assistance. As machine learning offers the accuracy performance of the future prediction so the machine learning models of this research were evaluated to find out which model fits the data and predicts the newly added item class most accurately.

In our study, we have used two different inventory item datasets and analyzed two datasets with three MCDM techniques, such as Simple Additive Weighting (SAW), Analytic Hierarchy Process (AHP), and VIšekriterijumsko KOmpromisno Rangiranje (VIKOR), and four machine learning techniques such as Support Vector Machine (SVM), Artificial Neural Network (ANN), K-nearest neighbors (KNN) and Gaussian Process Classification (GPC).

3.1 Multi-Criteria Decision-Making Methods (MCDM)

3.1.1 Simple Additive Weighting (SAW)

The Simple Additive Weighting (SAW) technique, otherwise called the weighted sum technique, is the most popular and most broadly utilized MCDM technique. The algorithm of SAW is described below,

1. Firstly, a Multi-criteria dataset is taken as input, and beneficial and non-beneficial criteria are identified by the user
2. Secondly, the beneficial and non-beneficial criteria are normalized by the equation (01) or, (02). Where x is denoted as criteria, and x_{ij} is the value of the ith row and jth column.
3. Then, user-defined weights of each criterion will be multiplied to the respective criteria of the dataset. Weight is denoted by w and w_j is the weight of the jth column.
4. Next, the row-wise summation of the dataset will be measured by the equation (03). This row-wise summation is known as the performance of the respective item.
5. Finally, the performance of each item is sorted in descending order. The item with the highest performance value is considered as a high priority item and the rest items have respective priority.

$$\text{Linear Normalization, } k_{ij} = \begin{cases} \frac{x_{ij}}{\max_i x_{ij}}, & \text{if j is a beneficial attribute} \\ \frac{\min_i x_{ij}}{x_{ij}}, & \text{if j is a non} - \text{beneficial attribute} \end{cases} \qquad (01)$$

i =1, 2……., m; j =1, 2…. n.

Or,

Max-Min Normalization,

$$k_{ij} = \begin{cases} \frac{x_{ij} - \min_i x_j}{\max_i x_j - \min_i x_j}, & \text{if j is a beneficial attribute} \\ \frac{\max_i x_j - xj}{\max_i x_j - \min_i x_j}, & \text{if j is a non} - \text{beneficial attribute} \end{cases} \qquad (02)$$

i =1,2,…….,m ; j =1,2,….n.

where k_{ij} ($0 \le k_{ij} \le 1$) is characterized as the normalized performance rating. This normalization process changes all the evaluations in a linear (proportional) or maximum-minimum normalization way so that the relative order of magnitude of the ratings stays equivalent. The general preference value of every alternative V_i is gained by-

$$V_i = \sum_{j=1}^{n} w_j k_{ij} \text{ ; } j = 1, 2, \ldots m. \qquad (03)$$

The alternative with the maximum value of V will be recognized as the best alternative. Exploration results have indicated that the linear or max-min form of compromises between attributes utilized by the SAW technique creates amazingly close approximations to muddled non-linear structures while being far simpler to use and comprehend. [28] Both of the normalization equations were utilized based on the data characteristics.

3.1.2 Analytic Hierarchy Process (AHP)

AHP is a supervised MCDM technique. The essential thoughts of this technique are inclined towards the pair-wise correlation dependent on the eigenvector. Marcus and Minc[29] characterized eigenvector as: " eigenvectors are a unique set of vectors related to a linear system of equations (i.e., a matrix equation) that are some of the time otherwise called characteristic vectors, proper vectors, or latent vectors. It is broadly being used for subjective evaluations by specialists and academics [30]. As a piece of structuring the problem in this methodology, the decision problem should structure into a hierarchical model. The model must represent the connection between the goal, criteria, and alternatives [31]. AHP methodology can be described in some steps as below –

1. Formation of pair-wise comparison matrix of criteria with the help of Saaty’s scale (illustrated in Table - 3.1)
2. Evaluation of the pair-wise comparison matrix as it is consistent or not. If not consistent then the input of the pair-wise matrix must be changed for making the matrix consistent.
3. If the matrix is consistent, then determined weights of the criteria (which are the byproduct of the consistency evaluation calculation) will be applied to the inventory item dataset (and the dataset must have the same criteria as the pair-wise matrix has).
4. Finally, the performance of each item will be measured after some calculations.

And the implementation of the AHP method is shown in Chapter-4 section-4.3.2.2.

Table 3.1: Saaty's Scale for AHP

Rank	Description
1.00	Equally Important
3.00	Moderately Important
5.00	Strongly Important
7.00	Significantly Important
9.00	Extremely Important

3.1.3 VIšekriterijumsko KOmpromisno Rangiranje (VIKOR)

VIKOR is a multicriteria decision-making technique developed by Serafim Opricovic in 1998 [13]. VIKOR means Multicriteria Optimization and Compromise Solution. VIKOR solves the problem between conflicting criteria by an agreement established by mutual concession means a compromise solution. In VIKOR compromise solution is the ranking of the alternatives close to the ideal solution. Lp-matric (equation 04) is the equation that is the mathematical representation of the compromise ranking.

The procedure for VIKOR is described below –

Let assume,

f_{ij} = value of jth alternative and ith criterion

w_i = weight of ith criterion

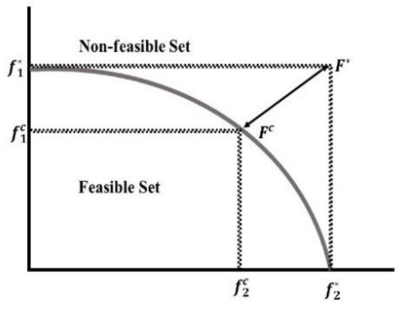
$$w_j = \sum_{i=1}^{n} w_i f_{ij}$$

And the VIKOR method utilizes the accompanying L_p- metric as a full function:

$$L_{p,j} = \left\{\sum_{i=1}^{n} [w_j (f_i^* - f_{ij})/(f_i^* - f_i^-)]^p\right\}^{\frac{1}{p}}, \qquad (04)$$

$$1 \leq p \leq \infty;\ j = 1, 2, \ldots, J.$$

The compromise solution F^c is an attainable solution that is the 'closest' to the ideal F^*, and compromise implies an arrangement built up by standard concessions, as is outlined in Fig 3.1 by-



0

Figure 3.1: Ideal and compromise solutions

$\Delta f_1 = f_1^* - f_1^c$ and $\Delta f_2 = f_2^* - f_2^c$.

The steps of the compromise ranking algorithm VIKOR:

1. The f_i^* and f_i^- respectively presents the best and the worst values of all criteria. B and C represent beneficial and non-beneficial criteria respectively by equations (05) & (06). [23]

$$f_i^* = \{(\max_j\{f_{ij}\}|j \in B, \min_j\{f_{ij}\}j \in C)\} \quad (05)$$

$$f_i^- = \{(\min_j\{f_{ij}\}|j \in B, \max_j\{f_{ij}\}j \in C)\} \quad (06)$$

2. The next step is to calculate S_j, R_j, j =1, 2…., J, by the equations (07) & (08) respectively,

$$S_j = \sum_{i=1}^{n} w_i(f_i^* - f_{ij})/(f_i^* - f_i^-) \quad (07)$$

$$R_j = \max_i[w_j(f_i^* - f_{ij})/(f_i^* - f_i^-)] \quad (08)$$

3. Later on, Q_j is calculated which is the VIKOR Index by the equation (09) below,

$$Q_j = \frac{v(S_j - S^*)}{(S^- - S^*)} + (1 - v)\frac{(R_j - R^*)}{(R^- - R^*)} \quad (09)$$

Where

$v =$ maximum group utility

$(1 - v) =$ weight of the individual regret,

In is research value of v is 0.5

$S^* = \min_j S_j$, $S^- = \max_j S_j$,

$R^* = \min_j R_j$, $R^* = \max_j R_j$.

4. The alternatives are ranked, arranged by the values S, R, and Q, in descendent order. The outcomes are three ranking lists**.**

The best alternative has the minimum value of Q.[13]

3.2 Machine Learning Algorithms

3.2.1 Support Vector Machine (SVM)

SVM, a set of interrelated machine learning methods, basically depends on the statical learning theory of Vladimir Vapnik. The algorithms utilized for SVMs began developing with more noteworthy processing power accessibility, preparing for various useful applications. The conventional SVM manages two-class issues, yet it tends to be produced for multi-class arrangement. But now SVM deals with multi-class problems. SVM is now a supervised AI method for both regression and classification problems and is generally utilized for classification issues. An SVM classifier takes a set of data as input, which has a place with various classes, manufactures a model that predicts any provided instance has a place with which class. It measures an ideal hyperplane to isolate multiple classes in the data set. The hyperplane is put at the maximum distances from the closest points of a given data set.[14]

An SVM model proposes the training models into isolated classes in a mapped space as specific points. By utilizing Lagrange multipliers, identifying the hyperplane needs to tackle a quadratic optimization problem in that space. Those hyperplane determining points are called Support Vectors. In this way, the vectors are essential components to train the classifying algorithm. It decides an optimal hyperplane to isolate various classes in a data set.

The conditions for the SVM classifications are represented by the equation (10) & (11),

$$wx_i + b \geq +1 \text{ for } y_i = +1 \tag{10}$$

$$wx_i + b \leq -1 \text{ for } y_i = -1 \tag{11}$$

Where,

x_i is the feature of the ith example	y_i is the binary output of the ith example
w is the given weight	b is the bias

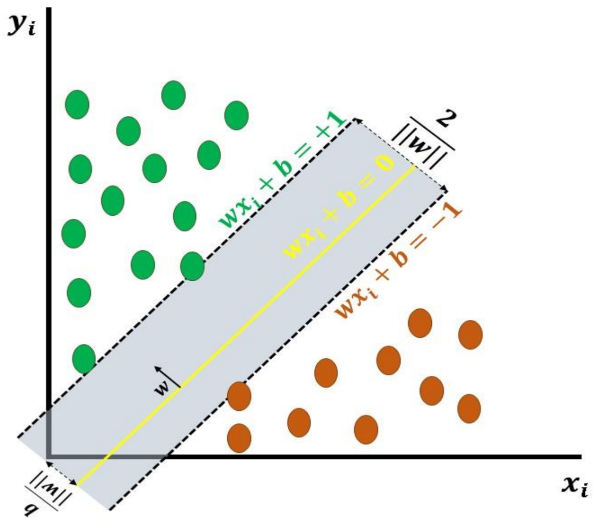


Figure 3.2: Maximum-margin hyperplane and margins for an SVM

On the off chance that these conditions in (10) and (11) are considered for each set of (x_i, y_i) while i =1, 2… m, we can frame it as in (12)

$$\{(x_i, y_i)|x_i \in R^N, y_i \in \{-1,1\}\}_{i=1}^{m} \tag{12}$$

So as to decide the hyperplane, which was set up as a long way from the support vectors as could reasonably be expected, the margin expected to be maximized. Hence maximization of the margin is proportional to the depreciation of ||w||. We may get the least ||w|| by quadratic programming, as demonstrated in equation (13) & (14),

$$\min \frac{1}{2} w^T w \tag{13}$$

$$\text{Subject to} \quad y_i(w^T x_i + b) \geq 1 \quad \text{and i =1, 2..., m} \tag{14}$$

Presenting Lagrange multipliers utilizing the Karush-Kuhn-Tucker hypothesis can settle this quadratic issue in equation (15). C in equation (16) is a penalty parameter. [32]

$$\max \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{i=1}^{m} \alpha_i \alpha_j y_i y_j x_i^T x_j \tag{15}$$

Subject to i =1, 2…., m

$$\sum_{i=1}^{m} \alpha_i y_i = 0, 0 \le \alpha_i \le C \tag{16}$$

To decrease the multifaceted nature of issues, SVM algorithms use, likewise kernels as mapping them in a high dimensional space[33]. A kernel function makes it simpler to classify the inputs utilizing the kernel trick as the following equation (17),

$$K(x_i, x_j) = \varphi(x_i)^T \varphi(x_j) \tag{17}$$

We can change the issue to locate the optimal hyperplane in another quadratic model by following equation (18) & (19),

$$\max \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{i=1}^{m} \alpha_i \alpha_j y_i y_j \varphi(x_i)^T \varphi(x_j) \tag{18}$$

$$\text{Subject to } \sum_{i=1}^{m} \alpha_i y_i = 0, 0 \le \alpha_i \le C \tag{19}$$

and i =1, 2….., m

For each of the n support vectors, the decision function D(x) in equation (20), which isn't an extent, yet the sign becomes as in (11).

$$D(x) = \text{sign}(\sum_{k=1}^{n} a_k\, y_k K(x, x_k) + b) \tag{20}$$

For example, SVM's different kernel functions outspread essential capacity, sigmoid capacity, and polynomial work. More data on SVM calculations and applications can be found in[34], and the measurable learning hypothesis's subtleties are accessible in[33].

3.2.2 Artificial Neural Network (ANN)

Being an equal, dynamic arrangement of profoundly interconnected and collaborating parts dependent on neurobiological models, an ANN is a supervised machine learning algorithm. It is a mathematical model that integrates neurons. The neurobiological sensory or nervous system comprises individuals however profoundly interconnected nerve cells called neurons. These neurons usually get data or stimuli from the external condition like how the neuron in an organic eye enlists the intensity of light in a particular room. These stimuli go through the network created by neurons delivering synapses to the neighboring neurons. The associations between the neurons are also called neurotransmitters. The stimuli can either energize the neuron or restrain it. It will fire when it gets the input and give the data to other neighboring neurons if its data energize the accepting neuron. The data's information is hosed and not passed on because the neuron is repressed. As such, the neurons process the data and pass it on just in the event that it is viewed as significant.

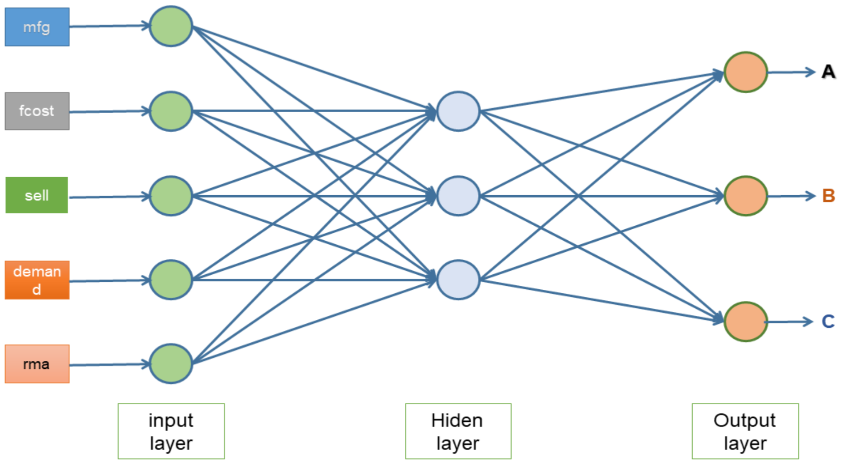


Figure 3.3: The model of Artificial Neural Network (ANN)

An ANN developed to mirror the neurons in the human cerebrum comprises lots of neurons, which are disseminated in various hierarchical or progressive layers. One of the most generally executed neural network designs is the multilayer perceptron's (MLP) model. This network has a three-layer, hierarchical, feedforward structure. The absolute number of neurons, the number of neurons on each layer, just as the number of layers decides the network model's precision. An ordinary MLP has appeared in Fig 3.3. The neurons in the input layer speak to the properties of stimuli in a data set. These inputs (x_1, x_2, ..., x_n) start the actuations into the network. As outlined in Fig 3.3, these inputs are consolidated in the lower part of the neuron. [16]The upper part of the neuron takes this total and computes how much the total is significantly utilizing a transfer function (f) in equation (21), delivering an individual output,

$$f\left(\sum_{i=1}^{n} w_i x_i\right) \tag{21}$$

Where w is weight vector $w \equiv [w_1, w_2, ..., w_n]$, and x is the input vector $x \equiv [x_1, x_2, ..., x_n]$ for a specific neuron.

The transfer function fills in as a faint switch for turning on and off, depending on the neurons' input. The transfer function's choice ordinarily relies upon the type of the network's output. In such a manner, there are various other options, including the step function, Softsign function, Rectified Linear Unit (ReLU) function sigmoid function, hyperbolic tangent function, and linear function, among others. As this study has a multi-class output and the output ranges between 0 and 1, this study uses the Softmax function in equation (22) as the output activation function. This Softmax activation function is mostly utilized in multi-class models where it returns probabilities of each class, with the objective class having the most elevated probability.

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{22}$$

In this study, the Rectified Linear Unit (ReLU) function represented in equation (23) was used as the activation function for hidden layers. Being one of the most popular activation functions, the ReLU function ensures quicker calculation – it doesn't process exponentials and divisions, accordingly boosting the general calculation speed.

$$f(x) = \max(0, x) = \begin{cases} x_i, \text{if } x_i \geq 0 \\ 0, \text{if } x_i < 0 \end{cases} \tag{23}$$

Since the inventory classification issues are naturally non-linear, it is necessary to make an ANN, which can rough complex non-linear functions. This is accomplished by including hidden layers (for example, a few layers of sigmoidal capacities), which comprise neurons that get inputs from the neighboring cells and give outputs to resulting cell layers. Even though a solitary hidden layer is enough to tackle any function approximation issue in principle, a few problems might be simpler to tackle utilizing more than a solitary hidden layer. [16]

In rundown, every connection in the ANN has a weight created from the input values and changed over to an output value by a transfer function. The output value of a neuron is a function of the weighted sum of its inputs. The weights speak to both the quality and nature of the connection between neurons. A substantial positive value will affect the following neuron to enact, while an enormous negative value will repress the activation of the following neuron. The assurance of these weights is an essential segment of the learning cycle. It is produced by an iterative training measure where case models with known decision outputs are over and again presented to the network[16]. An optimization algorithm Adam (Adaptive Moment Estimation) can be utilized for optimizing the value of weights (w) and bias (b) in ANN.

3.2.3 K-Nearest Neighbors (KNN)

The k-nearest neighbors' algorithm (k-NN) is a non-parametric technique proposed by Thomas Cover utilized for classification and regression.[1] In the two cases, the input comprises the k nearest training models in the component space. k-NN is a kind of occasion-based learning, or lazy learning, where the function is just approximated locally, and all calculation is conceded until function assessment. Since this algorithm depends on distance for classification, normalizing the training data can develop its accuracy drastically. Despite the fact that there have been a few learning algorithms uniquely intended for multi-label learning, creating lazy learning approach for multi-label issues is an unsolved issue. [35]

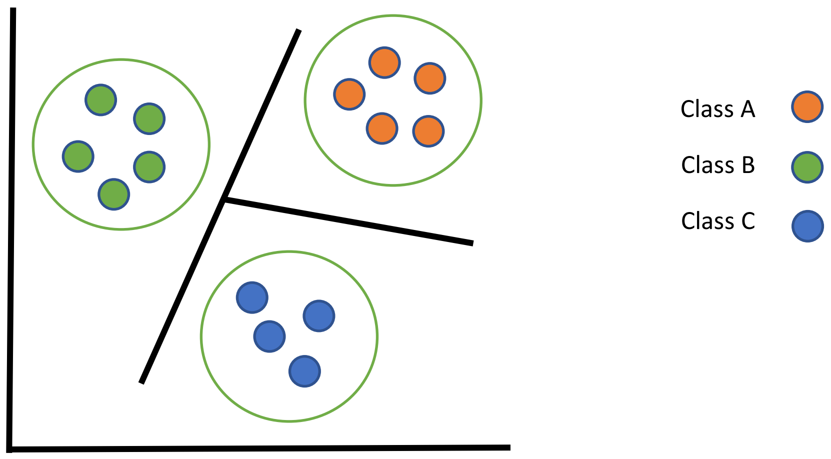


Figure 3.4: K-nearest neighbors visual representation

K-nearest neighbors algorithm has a place with a specific family called instance-based learnings (the system is called instance-based learning). It varies from different methodologies since it doesn't function with a real numerical model. The induction is actually performed by a direct comparison of new examples with existing ones (characterized as instances). KNN is a methodology that can be handily utilized to tackle clustering, grouping, classification, and regression issues. The fundamental thought behind the clustering algorithm is straightforward. [36] Now we consider a data generating process P_{data} and specify a dataset in equation (24) drawn from this appropriation:

$$X = \{\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n\} \text{ where } \bar{x}_i \in \mathbb{R}^N \tag{24}$$

Each point has a dimensionality equivalent to N. Here; it would now be able to present a distance function $\mathrm{d}(\bar{\mathrm{x}}_1, \bar{\mathrm{x}}_2)$, which, as a rule, can be summed up with the Minkowski distance in equation (25),

$$\mathrm{d}_\mathrm{p}(\bar{\mathrm{x}}_1, \bar{\mathrm{x}}_2) = \left(\sum_{\mathrm{j}=1}^{\mathrm{N}} |\bar{\mathrm{x}}_1{}^{(\mathrm{j})} - \bar{\mathrm{x}}_2{}^{(\mathrm{j})}|^\mathrm{p} \right)^{\frac{1}{\mathrm{p}}} \tag{25}$$

At the point when p = 2, $\mathrm{d}_\mathrm{p}(\bar{\mathrm{x}}_1, \bar{\mathrm{x}}_2)$ speaks to the convectional Euclidean distance, which is regularly the default decision in practically any situation. In specific cases, it tends to be useful to utilize different variations, for example, p = 1 (which is otherwise called Manhattan distance) or p > 2. Regardless of whether all the properties of a measurement matrix function stay unaltered, various p final results can be semantically assorted.

The distance diminishes monotonically with p and merges to the most prominent part supreme distinction $|\bar{\mathrm{x}}_1{}^{(\mathrm{j})} - \bar{\mathrm{x}}_2{}^{(\mathrm{j})}|$ when p → ∞. This way, at whatever point, it's critical to weigh all the parts similarly so as to have a predictable metric, little values of p are best (for instance, p = 1 or 2). This outcome has additionally been contemplated and formalized by Aggarwal, Hinneburg, and Keim. [37]

On the off chance that we consider a conventional distribution G of M focuses $\bar{\mathrm{x}}_1 \in (0,1)^\mathrm{d}$, a distance function dependent on the L_p standard, and the largest $\mathrm{D}^\mathrm{p}_{\mathrm{max}}$ and least $\mathrm{D}^\mathrm{p}_{\mathrm{min}}$ distance (calculated utilizing the L_p standard) between two points, $\bar{\mathrm{x}}_\mathrm{j}$ and $\bar{\mathrm{x}}_\mathrm{k}$ drawn from G and the origin $0 \in \mathbb{R}^\mathrm{N}$, the accompanying disparity holds in equation (26):

$$\mathrm{C}_\mathrm{p} \leq \lim_{\mathrm{d}\to\infty} \mathrm{E}\left[\frac{\mathrm{D}^\mathrm{p}_{\mathrm{max}} - \mathrm{D}^\mathrm{p}_{\mathrm{min}}}{\mathrm{d}^{\frac{1}{\mathrm{p}}-\frac{1}{2}}} \right] \leq (\mathrm{M}-1)\mathrm{C}_\mathrm{p} \text{ where } \mathrm{C}_\mathrm{p} \geq 0 \tag{26}$$

Obviously, when the input data dimensionality is exceptionally high and $\mathrm{p} = 2$, the desired value $\mathrm{E}\left[\mathrm{D}^\mathrm{p}_{\mathrm{max}} - \mathrm{D}^\mathrm{p}_{\mathrm{min}}\right]$ gets limited between two constants, $\mathrm{k}_1(\mathrm{C}_\mathrm{p}\mathrm{d}^{\frac{1}{\mathrm{p}}-\frac{1}{2}})$ and $(\mathrm{M}-1)\mathrm{k}_1\left(\mathrm{C}_\mathrm{p}\mathrm{d}^{\frac{1}{\mathrm{p}}-\frac{1}{2}}\right) > 0$, decreasing the real impact of any particular distance. Actually, given two nonexclusive couples of points $(\bar{\mathrm{x}}_1, \bar{\mathrm{x}}_2)$ and $(\bar{\mathrm{x}}_3, \bar{\mathrm{x}}_4)$ drawn from G, the accompanying disparity's general result is when p → ∞, autonomously of their relative positions. This significant outcome confirms the dataset's dimensionality indicates the significance of picking the correct metric and that $\mathrm{p} = 1$ is the most ideal decision when $\mathrm{d} = 1$, while $\mathrm{p} = 1$ can create conflicting outcomes due to the incapability of the metric. [36]

3.2.4 Gaussian Process Classification (GPC)

Gaussian Processes (GP) can be defined as a generic supervised learning method designed to solve regression and probabilistic classification problems.

Suppose, we have a data set D of data points $\mathrm{x_i}$ with binary class labels $\mathrm{y_i}$ with this data set, we want to find the correct class label for a new data point $\mathrm{x_i}$. We can do this by computing the class probability $\mathrm{p(\tilde{y}|\tilde{x}, D)}$. Here, $\mathrm{y_i \in \{-1,1\}: D = \{(x_i, y_i)|i = 1,2, \dots, n\}}$,

$$\mathrm{X = \{x_i|i = 1,2, \dots, n\}},$$

$$\mathrm{Y = \{y_i|i = 1,2, \dots, n\}}$$

Let us assume that the class label is obtained by transforming some real-valued latent variables $\tilde{f}$, which is the value of some latent function f(˙) evaluated at $\tilde{x}$. We will put a Gaussian process prior to this function which indicates that any number of points evaluated from the function have a multivariate Gaussian density. [38] We can assume that this Gaussian process prior is parameterized by Θ which we will call this the hyperparameters. We can find the probability of interest given Θ as in equation (27)

$$p(\tilde{y}|\tilde{x}, D, \Theta) = \int p(\tilde{y}|\tilde{f}, \Theta) p(\tilde{f}|D, \tilde{x}, \Theta), d\tilde{f} \qquad (27)$$

There in equation (27) the probability of the class label $\tilde{y}$ at a new data point $\tilde{x}$ given data D and hyperparameters, Θ has been represented.

The second part of equation (27) is obtained by another integration over f =[f_1, f_2, ..., f_n], the values of the latent function at the info points in equation (28),

$$p(\tilde{y}|\tilde{x}, D, \Theta) = \int p(f, \tilde{f}|D, \tilde{x}, \Theta), d\tilde{f} = \int p(\tilde{f}|\tilde{x}, f, \Theta)\, p(\tilde{f}|D, \Theta) df \qquad (28)$$

Where $p(\tilde{f}|\tilde{x}, f, \Theta) = \frac{p(\tilde{f}, f|\tilde{\mathrm{x}}, \mathrm{X}, \Theta)}{\mathrm{p(f|X, \Theta)}}$ and

$$\mathrm{p(f|D, \Theta) \propto p(Y|f, X, \Theta) p(f|X, \Theta)}$$

$$\therefore \mathrm{p(f|D, \Theta) = \{\textstyle\prod_{i=1}^{n} p(y_i|f_i, \Theta)\} p(f|X, \Theta)} \qquad (29)$$

Between two terms in equation (29), the first term is the likelihood: the probability for each observed class given the latent function value, while the second term is the GP prior over functions evaluated at the data. [38] Writing the dependence of f on x implicitly, the GP prior over functions can be written as following equation (30),

$$\mathrm{p(f|X, \Theta) = \frac{1}{(2\pi)^{N/2}\sqrt{|C_\Theta|}} \times e^{(-\frac{1}{2}(f-\mu)^T C_\Theta^{-1})(f-\mu)}} \qquad (30)$$

Where,

the mean, μ = zero vector = 0

each term of a covariance matrix C_{ij}= a function of x_i and x_j, i.e., $C(x_i, x_j)$.

One form for the likelihood term $p(y_i|f_i, \Theta)$, which related to $f(x_i)$ monotonically to the probability of y_i = +1, is presented in equation (31)

$$p(y_i|f_i, \Theta) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{y_i f(x_i)} e^{(-\frac{z^2}{2})dz} = \mathrm{erf}\,(y_i f(x_i)) \tag{31}$$

Other possible forms for the likelihood:

- a sigmoid function $\frac{1}{1+e^{-y_i f(x_i)}}$
- a step function $H(y_i f(x_i))$,
- a step function with labeling error $\epsilon + (1 - 2\epsilon)H(y_i f(x_i))$

3.3 Voting System

A voting system is a set of rules that determine how elections and selection processes are conducted and how their results are determined.

Here we have 3 sets of classes. They are-

1. AHP based ABC Classes
2. SAW based ABC Classes &
3. VIKOR based ABC Classes.

So, we have taken an approach to combine them into one set of class. And the approach is the voting system. If two of three class is same then that will be taken as a final class. In table 3.2 it is shown.

Table 3.2: Sample of VOTING System

Idem_Number	**AHP_Class**	**SAW_Class**	**VIKOR_Class**	**Combo_Class**
1	C	C	C	C
2	B	A	A	A
3	A	A	A	A
4	A	A	A	A
5	C	B	A	B

Here for Item number - 2

AHP Class: B

SAW Class: A

VIKOR Class: A

So, as per the definition, the combined class result will be A, cause class A got 2 votes of 3.

But in the Case of Item_number - 5, this voting system can't be applied. Cause none of them are similar to each other.

In this special case, we took another approach for assigning a class to this type of inventory item. And the second approach is, firstly we will determine the average of the test, train accuracy, and run time by using 4 ML algorithms on 3 MCDM based ABC Classes. Each MCDM model has 4 ML models' corresponding train, test accuracies, and runtimes. On these 3 (train, test accuracies, and runtimes) criteria, the corresponding MCDM model will be applied to identify the best ML model performance on the particular MCDM model (described in table 6.1 to 6.6). Then we will identify which ML model gave the highest performance on which MCDM method. Suppose, ANN gives the highest performance on the SAW method. So, here Combined class for Item_number - 5 will be the class of SAW for that particular item and the combined class for Item_number - 5 will be "B".

Example:

AHP class: C; highest performance: 97%; ML MODEL: KNN

SAW class: B; highest performance: 98%; ML MODEL: ANN

VIKOR class: A; highest performance: 96%; ML MODEL: SVM

So, the Combined Class will be “B". (As the class of Item number - 5)

Chapter 4

Methodology

4.1 Overview

This research study's motivation is to inspect the degree to which Multi-Criteria Decision-Making strategies in alliance with Machine Learning models can be used in the ABC Analysis. The performance of Artificial Neural Network (ANN), Gaussian Process Classification (GPC), and K-Nearest Neighbors (KNN) and Support Vector Machine (SVM) methods are assessed. The accuracy of the classifications or combined classifications they produce inside a real case investigation that looks at an inventory analysis issue is analyzed. The first step of the research includes taking the inventory items' dataset as input to be analyzed before selecting features based on a strong relationship with item properties. These selected features set the foundation for the MCDM methods (i.e., Simple Additive Weighting (SAW), Analytical Hierarchy Process (AHP), and VIKOR) to lead the ABC Analysis. When the data is sorted and the measures distinguished, SAW-, AHP- and VIKOR-based MCDM techniques are applied to the inventory items to recognize the A, B, and C classes. After that, the recognized A, B, and C classes of three MCDM methods are combined into One by VOTING System. This result is providing a particular class (A, B, or C) to each inventory item.

Following this, machine learning algorithms are implemented on both the inventory classes that are previously recognized by the Multi-Criteria Decision-Making techniques and combined classes that are generated by the VOTING system. The algorithms are trained and tested, utilizing the initially decided classes. Cross-validation, Train-Test accuracy, and run-time measure are used to recognize the perfectly fitted algorithm model to classify the inventory items.

4.2 Research Procedure Flowchart

A flowchart identifying every step of the proposed strategy is portrayed in Figure 4.1:

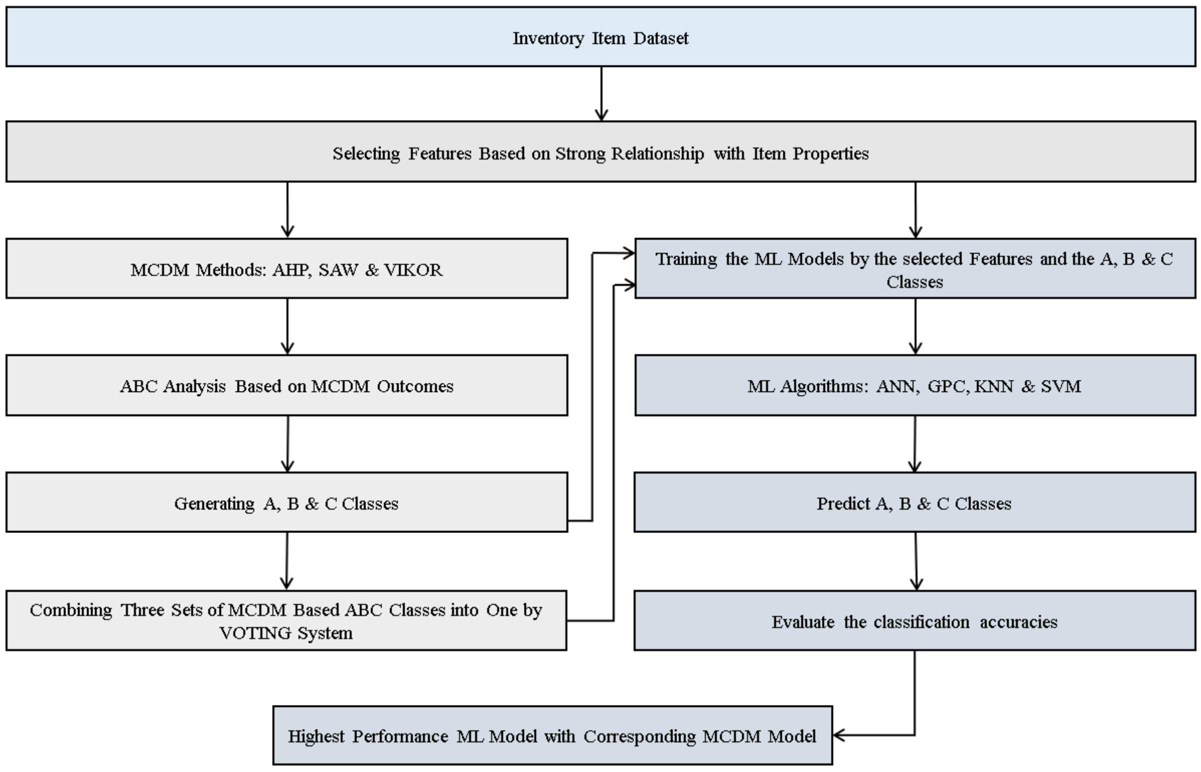

Inventory Item Dataset
Selecting Features Based on Strong Relationship with Item Properties
MCDM Methods: AHP, SAW & VIKOR
ABC Analysis Based on MCDM Outcomes
Generating A, B & C Classes
Combining Three Sets of MCDM Based ABC Classes into One by VOTING System
Training the ML Models by the selected Features and the A, B & C Classes
ML Algorithms: ANN, GPC, KNN & SVM
Predict A, B & C Classes
Evaluate the classification accuracies
Highest Performance ML Model with Corresponding MCDM Model

Figure 4.1: Flowchart of the proposed methodology

4.3 Research Procedure

4.3.1 Data Segmentation

Firstly, we select the most effective features among all the features available in the data set. Those features are selected which have a strong relationship with inventory item properties and the correlation coefficient of the features with each other. The features are Manufacturing Cost, Fixed Cost, Demand, Raw Material Availability, etc. are shown in table 4.1.

Secondly, the features with the categorical object data value are converted into numerical values by weightings like High = 3, Medium = 2, and Low = 1.

After that, the MCDM methods are applied.

Table 4.1: Data Set

Item_number	mfg	sell	demand	fcost	rma
0	144	318.24	1537	100.8	1
1	166	366.86	4290	116.2	3
2	287	634.27	4017	200.9	2
3	286	632.06	4925	200.2	2
4	205	453.05	2846	143.5	2

4.3.2 Implementation of Multi-Criteria Decision-Making Techniques

4.3.2.1 Application of Simple Additive Weighting (SAW)

The Simple Additive Weighting Method or SAW is a Multi-Criteria Decision-Making method. In the Simple Additive Weighting method, the selected features may be normalized with Max-Min or Linear normalization method. In normalization, the Beneficial and Non-beneficial features must be defined like, manufacturing cost must be low and selling price must be high, so here manufacturing cost (mfg) will be Non-beneficial and selling cost (sell) will be defined as a beneficial feature.

Here, for this study, Beneficial and Non-beneficial features are shown in table 4.2

Table 4.2: Feature Catagories

Features Category	Features
1. Beneficial	a. Demand(demand), b. Raw Material Availability(rma), c. Selling Price(sell)
2. Non-beneficial	a. Manufacturing Cost(mfg), b. Fixed Cost(fcost)

The result of normalization is shown in table 4.3.

Table 4.3: Normalized Value

Item_number	mfg	sell	demand	fcost	rma
0	0.790646	0.209354	0.006728	0.790646	0.0
1	0.741648	0.258352	0.507365	0.741648	1.0
2	0.472160	0.527840	0.457720	0.472160	0.5
3	0.474388	0.525612	0.622841	0.474388	0.5
4	0.654788	0.345212	0.244772	0.654788	0.5

After that, user-defined weights of each feature will be multiplied to the corresponding features like table 4.4 and the summation of the weights will be equal to 1.

Table 4.4: Weighted features

Item_number	mfg	sell	demand	fcost	rma
0	0.118597	0.041871	0.002019	0.118597	0.0
1	0.111247	0.051670	0.152209	0.111247	0.2
2	0.070824	0.105568	0.137316	0.070824	0.1
3	0.071158	0.105122	0.186852	0.071158	0.1
4	0.098218	0.069042	0.073432	0.098218	0.1

Finally, the performance of each item is measured by summing all of the features row-wise of the dataset as table 4.5. in table 4.5, the item with the highest performance value will be given top priority.

Table 4.5: Determination of item performance

Item_number	mfg	sell	demand	fcost	rma	performance
0	0.118597	0.041871	0.002019	0.118597	0.0	0.281083
1	0.111247	0.051670	0.152209	0.111247	0.2	0.626374
2	0.070824	0.105568	0.137316	0.070824	0.1	0.484532
3	0.071158	0.105122	0.186852	0.071158	0.1	0.534291
4	0.098218	0.069042	0.073432	0.098218	0.1	0.438910

4.3.2.2 Implementation of Analytic Hierarchy Process (AHP)

Analytic Hierarchy Process also is known as AHP is a very used and appreciated Multi-criteria decision-making method by many authors. In this method, a user-defined pare-wise matrix is formed (shown in table 4.6) by using the feature. Basically, this matrix is a weighting process of each two features, and the weighting value is from 1 to 9 (described in table 3.1).

Table 4.6: Pair-Wise Matrix

	mfg	sell	demand	fcost	rma
mfg	1	1/7	1/5	2	1/3
sell	7	1	2	9	3
demand	5	1/2	1	7	3
fcost	1/2	1/9	1/7	1	1/5
rma	3	1/3	1/3	5	1

Then all the features will be divided by their summation shown in table 4.7.

Table 4.7: Normalizing the values

	mfg	sell	demand	fcost	rma
mfg	0.060606	0.068519	0.054407	0.083333	0.044205
sell	0.424242	0.479157	0.544070	0.375000	0.398248
demand	0.303030	0.239578	0.272035	0.291667	0.398248
fcost	0.030303	0.053186	0.038901	0.041667	0.026550
rma	0.181818	0.159559	0.090588	0.208333	0.132749

Following that, criteria weight is measured averaging each row shown in table 4.8,

Table 4.8: Determining the feature weight

	mfg	sell	demand	fcost	rma	criteria_weight
mfg	0.060606	0.068519	0.054407	0.083333	0.044205	0.062214
sell	0.424242	0.479157	0.544070	0.375000	0.398248	0.444143
demand	0.303030	0.239578	0.272035	0.291667	0.398248	0.300912
fcost	0.030303	0.053186	0.038901	0.041667	0.026550	0.038121
rma	0.181818	0.159559	0.090588	0.208333	0.132749	0.154610

By multiplying the feature weight with each feature of the pair-wise matrix and some calculation the consistency of the pair-wise matrix is checked. If it is found that the pair-wise matrix is consistent, then the feature weights are considered valid weights that can be used for future calculations.

Then Normalization of the data set is done, which is already described above in SAW (paragraph 4.3.2.1 and Table 4.3).

After normalization, the previously measured “criteria_weights” from table 4.8 are multiplied with respective features of the normalized data set.
Finally, the performance of each item is measured by summing all the features row-wise of the data set (like previously discussed in SAW, paragraph 4.3.2.1) in table 4.9 and the item with a higher performance value will be given the top priority,

Table 4.9: Determination of each products’ performance

Item_number	mfg	sell	demand	fcost	rma	performance
0	0.049189	0.092983	0.002025	0.030141	0.000000	0.174338
1	0.046141	0.114745	0.152672	0.028273	0.154610	0.496440
2	0.029375	0.234436	0.137733	0.017999	0.077305	0.496849
3	0.029514	0.233447	0.187420	0.018084	0.077305	0.545770
4	0.040737	0.153323	0.073655	0.024961	0.077305	0.369981

4.3.2.3 Implementation of VlseKriterijumska Optimizacija I Kompromisno Resenje (VIKOR)

VlseKriterijumska Optimizacija I Kompromisno Resenje,' meaning multi-criteria optimization and compromise solution also known as VIKOR. Normalization and user-defined weight multiplication with the data set’s respective features, VIKOR remains the same as SAW (discussed in paragraph 4.3.2.1). Then summation (Si) and maximum value (Ri) of features are measured row-wise respectively shown in table 4.10,

Table 4.10: Determination of Si & Ri value

Item_number	mfg	sell	demand	fcost	rma	si	ri
0	0.118597	0.041871	0.002019	0.118597	0.0	0.281083	0.118597
1	0.111247	0.051670	0.152209	0.111247	0.2	0.626374	0.152209
2	0.070824	0.105568	0.137316	0.070824	0.1	0.484532	0.137316
3	0.071158	0.105122	0.186852	0.071158	0.1	0.534291	0.186852
4	0.098218	0.069042	0.073432	0.098218	0.1	0.438910	0.098218

By using these two sets of values S_i & R_i another set of values Q_i is measured by equation (09) (paragraph 3.1.3). And the item with a lower value of Q_i will place to a higher rank, means will get top priority. This procedure has lead us to the following table 4.11,

Table 4.11: Finding Q_i using S_i& R_i

Item_number	mfg	sell	demand	fcost	rma	S_i	R_i	Q_i
0	0.118597	0.041871	0.002019	0.118597	0.0	0.281083	0.118597	0.142616
1	0.111247	0.051670	0.152209	0.111247	0.2	0.626374	0.152209	0.511825
2	0.070824	0.105568	0.137316	0.070824	0.1	0.484532	0.137316	0.357622
3	0.071158	0.105122	0.186852	0.071158	0.1	0.534291	0.186852	0.515180
4	0.098218	0.069042	0.073432	0.098218	0.1	0.438910	0.098218	0.227921

4.3.3 ABC Analysis with MCDM Backend

4.3.3.1 SAW based ABC classification

Rearranging the performance by ascending order and then cumulative sum and cumulative percentage are measured respectively. Based on the cumulative percentage classification is done. Here Pareto distribution 80-20 rule is applied to the top 50% and the bottom 20% is taken as class C & class A respectively and the rest of the items are given class B shown in table 4.12,

Table 4.12: SAW based ABC classification

Item_number	mfg	sell	demand	fcost	rma	performance	RunCumCost	RunCostPerc	Class
4658	491	1085.11	1513	343.7	1	0.202491	0.202491	0.004051	C
3117	483	1067.43	1534	338.1	1	0.205418	0.407909	0.008161	C
4984	495	1093.95	1605	346.5	1	0.206619	0.614528	0.012295	C
767	471	1040.91	1557	329.7	1	0.209346	0.823874	0.016483	C
2396	469	1036.49	1564	328.3	1	0.210173	1.034047	0.020688	C
...	...	...	...	...	...	...	...	...	...
8113	438	967.98	6350	306.6	2	0.578179	3212.532568	64.272401	B
473	410	906.10	6236	287.0	2	0.578196	3213.110764	64.283969	B
3153	440	972.40	6359	308.0	2	0.578225	3213.688989	64.295537	B
...	...	...	...	...	...	...	...	...	...
9900	52	114.92	6913	36.4	3	0.794863	4997.511705	99.984068	A
265	53	117.13	6944	37.1	3	0.796331	4998.308036	100.000000	A

4.3.3.2 AHP based ABC classification

In AHP, the performance is also rearranged in ascending order then cumulative sum, and the cumulative percentage is measured based on the performance. After that ranking is done based on cumulative percentage. Then the cumulative item sum and cumulative item percentage are measured. Based on cumulative item percentage class is given and Pareto distribution 80-20 rule is followed where op 50% and bottom 20% is taken as class C & class A respectively and rest of the items are given class B [25] shown in table 4.13,

Table 4.13: AHP based ABC classification

Item number	**mfg**	**sell**	**demand**	**fcost**	**rma**	**Performance**	**RunCum Cost**	**RunCost Perc**	**Rank**	**RunItem Cum**	**RunItem Perc**	**Class**
3414	56	123.7	1544	39.2	1	0.1073	0.107338	0.002146	1.0	1.0	0.000002	C
2883	58	128.1	1517	40.6	1	0.1073	0.214729	0.004292	2.0	3.0	0.000006	C
492	62	137.0	1559	43.4	1	0.1127	0.327482	0.006546	3.0	6.0	0.000012	C
9861	56	123.7	1681	39.2	1	0.1148	0.442317	0.008841	4.0	10.0	0.000020	C
3137	52	114.9	1737	36.4	1	0.1148	0.557153	0.011137	5.0	15.0	0.000030	C
...	...	...	...	...	...	...	...	...	...	...	...	...
2798	431	952.5	2499	301	3	0.6013	3215.6343	64.27581	7396	27354106	54.70274	B
3387	493	1089	4458	345	1	0.6014	3216.2357	64.28783	7397	27361503	54.71753	B
8871	382	844.2	3187	267	3	0.6014	3216.8372	64.29985	7398	27368901	54.73232	B
...	...	...	...	...	...	...	...	...	...	...	...	...
4223	497	1098	6905	347	3	0.8929	5001.9716	99.98207	9999	49995000	99.98000	A
3625	497	1098	6978	347	3	0.8969	5002.8685	100.0000	10000	50005000	100.0000	A

4.3.3.3 VIKOR based ABC classification

VIKOR performance is arranged in descending order and the rest of the calculation is same as the AHP method discussed above in paragraph 4.3.3.2. And table 4.14 shows the item performance for the VIKOR method,

Table 4.14: VIKOR based ABC classification

Item number	mfg	sell	demand	fcost	rma	qi	RunCum Cost	RunCost Perc	Rank	RunItem Cum	RunItem Perc	Class
265	53	117.13	6944	37.1	3	0.992994	0.992994	0.021034	1.0	1.0	0.000002	C
7364	105	232.05	6980	73.5	3	0.989482	1.982476	0.041994	2.0	3.0	0.000006	C
9900	52	114.92	6913	36.4	3	0.987809	2.970285	0.062919	3.0	6.0	0.000012	C
979	55	121.55	6912	38.5	3	0.987073	3.957358	0.083827	4.0	10.0	0.000020	C
2829	123	271.83	6981	86.1	3	0.986280	4.943638	0.104720	5.0	15.0	0.000030	C
...	...	...	...	...	...	...	...	...	...	...	...	...
9528	422	932.62	3183	295.4	1	0.276077	4314.157406	91.385461	8008.0	32068036.0	64.129659	B
8369	422	932.62	3183	295.4	1	0.276077	4314.433483	91.391309	8009.0	32076045.0	64.145675	B
3752	78	172.38	3052	54.6	1	0.276065	4314.709548	91.397157	8010.0	32084055.0	64.161694	B
...	...	...	...	...	...	...	...	...	...	...	...	...
9200	246	543.66	1696	172.2	1	0.057729	4720.778847	99.998797	9999.0	49995000.0	99.980002	A
4660	236	521.56	1606	165.2	1	0.056770	4720.835617	100.000000	10000.0	50005000.0	100.000000	A

4.3.4 Implementation of Machine Learning

4.3.4.1 Implementation of SVM

Support Vector Machine (SVM) can be used for two types of problems, Regression and Classification. The algorithm used for classification is known as Support Vector Classifier (SVC). As our research is based on classification, we used SVC. SVM comes with many hyperparameters and various kernels. In our research, we kept all the hyperparameters as the default value and set the kernel as “Poly Kernel”.

4.3.4.2 Implementation of KNN

K-Nearest Neighbors (KNN) can perform both regression and classification. In this study classification algorithm is adopted. In KNN, a user defines the number of clusters to be used. As we are working with three classes so the cluster size has been set to three. And other hyperparameters are set to a default value.

4.3.4.3 Implementation of GPC

Gaussian Process Classifier (GPC) is a classification algorithm that is a co-domain of the Gaussian Process. It uses lazy learning and a measure of the similarity between points to predict the value for an unseen point from the training data set. One hyperparameter ‘multi_class' is set to 'one_vs_one' so that our multiclass problem can be solved. In one_vs_one, one binary Gaussian process classifier is fitted for each pair of classes, which is trained to separate these two classes. The prediction of these binary predictors is combined into multi-class prediction. And other hyperparameters are kept default.

4.3.4.4 Implementation of ANN

An Artificial Neural Network is the virtual representation of neurons. It works like the human brain neurons. We implemented this algorithm by using “Keras” an Artificial Neural Network Python library that acts as an interface for the Backend library TensorFlow. Here Sequential Model is used with three layers, an input layer, one hidden layer, and an output layer. There are 3 neurons in the hidden layer and 3 neurons in the output layer respectively. In the input and hidden layer “ReLU” (Rectified Linear Unit) Activation Function is used for fast calculation as it's not an exponential function. On the other hand, the ‘Softmax’ function, also known as ‘Softargmax’ or ‘Normalized Exponential’ Function, is used in the output layer as it deals with multiple dimensions or multi-class output. In this neural network for weight and bias, ‘Adam’ (Adaptive Moment Estimation) optimizer and loss function for multi-class classification 'categorical_crossentropy' is used. For training ‘mini-batch size’ = 128, and 10% test data, 10% ‘validation data’ is fitted with 20 epochs.

The datasets were randomly split as 90 to 80% training and 10 to 20% testing initially. After that, this training and testing process was run 5 times in each MCDM model and each time we measured the train, test accuracy, and total run time. And took the average of the train, test accuracy & total run time for each MCDM model for further study.

And manufacturing cost, selling price, fixed cost, demand & raw material availability are the inputs or ‘features’ of the algorithms, and ‘Class’ is the ‘target’ or ‘label’ of those features.

Chapter 5

Data Description

Two dummy data sets of inventory items having five features named after Manufacturing Cost per Product, Fixed Cost per Product, Selling Price, Demand per week, and Raw Material Availability, are used for this analysis. Two datasets were generated with the dimensions of

- Dataset 01 : 100000 x 5
- Dataset 02 : 150000 x 5

Properties of the features are described below,

1.<u>Manufacturing Cost per Product:</u> It is the manufacturing cost of a product. As we generated the datasets, that's why we could manipulate the data. Here we kept the manufacturing cost between 50 to 500BDT.

Manufacturing Cost = np.random.randint(50,500, n)

Where, n = number of items

2.<u>Fixed Cost per Product:</u> We kept a relation between the Manufacturing Cost and Fixed Cost, and it is

Fixed Cost = (manufacturing cost / 2) + (20% of manufacturing cost)

3.<u>Selling Price:</u> The selling price is the combination of manufacturing cost and fixed cost, and the relation is

Selling Price = (manufacturing cost + fixed cost) + (manufacturing cost + fixed cost) / 2

4.<u>Demand per week:</u> The Demand per week is 1500 to 7000 units.

Demand per Week = np.random.randint(1500, 7000, n)

5.<u>Raw Material Availability:</u> Raw material availability of a product is a categorical data, like

a) High

b) Medium &

c) Low

and this categorical data is further converted to numerical weights as follows-

Table 5.1: Numerical Weights of Categorical Data

High	3
Medium	2
Low	1

<u>Data Types of The Data Sets:</u> The data types of the dataset is described in table 5.2

Table 5.2: Data Types of The Data Sets

Features	**Data Type**	**Data Set Type**
Manufacturing Cost	int64	Discrete
Fixed Cost	float64	Continuous
Selling Price	float64	Continuous
Demand per Week	int64	Discrete
Raw Material Availability	int64	Categorical (scale of 1 to 3)

Reason Behind Choosing These Features:

The above five features are the major attribute of a product as we will do ABC Classification of these products with the help of MCDM methods. That’s why the selected features should have a direct relation to the properties of the product.

These five features are directly related to the product, and any of these five can’t be ignored or overlooked.

Other properties of a product can be included with these five, and the thesis can be extended in the future.

Chapter 6

Numerical Illustrations

Two automatically generated datasets have been classified by three different MCDM methods and trained by four machine learning algorithms to get the best MCDM model and ML algorithm by some performance measures. The applied programing language is PYTHON.

The MCDM methods are:

I. Simple Additive Weighting (SAW)
II. Analytic Hierarchy Process (AHP)
III. VIšekriterijumsko KOmpromisno Rangiranje (VIKOR)

The machine learning algorithms are:

I. Support Vector Machine (SVM)
II. Artificial Neural Network (ANN)
III. K-nearest neighbors (KNN)
IV. Gaussian Process Classification (GPC)

All the machine learning algorithms are applied under each MCDM method for individual datasets. Thus the train accuracy, test accuracy, and runtime were measured. Then on these three measures, corresponding MCDM methods were applied again to measure identify the best ML model for that particular MCDM method. Peng et al.[39] took the same approach to measure the performance of ML models and to identify the best ML model. All the individual outcomes are described below:

6.1 MCDM methods and Machine Learning Algorithms Application

6.1.1 AHP with Machine Learning Algorithms

The AHP and machine learning algorithms for datasets of two different dimensions are classified by PYTHON. The data sets contain 5 attributes and one label column. Defining all the input as x and output as y, the training accuracy, testing accuracy, and runtime for each dataset is determined by running the models five times and took the average value. We took 80 to 90 percent as training data and 10 to 20 percent as testing data. We have found the average testing accuracy, average training accuracy, average run time for individual datasets. Besides, The MCDM method AHP was applied on those average results and got the performance for each ML model. Table 6.1 and Table 6.2 represents the Accuracy, Runtime, and Performance Measures of Machine Learning models on AHP for Dataset 01and 02. The results are given below:

Table 6.1 Accuracy, Runtime and Performance Measures of Machine Learning models on AHP for Dataset 01

AHP (100K)	**train_Acc**	**test_Acc**	**time**	**Performance By AHP**
ANN	99.73	99.73	16.34	0.982
SVM	77.85	77.83	116.98	0.045
KNN	87.07	75.03	0.11	0.381
GPC	100	73.5	103.53	0.654

Table 6.2 Accuracy, Runtime and Performance Measures of Machine Learning models on AHP for Dataset 02

AHP (150K)	train_Acc	test_Acc	time	Performance By AHP
ANN	99.8	99.8	24.99	0.991
SVM	77.94	77.83	278.32	0.043
KNN	87.38	75.67	0.18	0.392
GPC	100	74.49	301.35	0.633

6.1.2 SAW with Machine Learning Algorithms

The SAW and machine learning algorithms for datasets of two different dimensions are classified by PYTHON. The data sets contain 5 attributes and one label column. Defining all the input as x and output as y, the training accuracy, testing accuracy, and runtime for each dataset is determined by running the models five times and took the average value. We took 80 to 90 percent as training data and 10 to 20 percent as testing data. We have found the average testing accuracy, average training accuracy, average run time for individual datasets. Lastly, The MCDM method SAW was applied on those average results and got the performance for each ML model. The results are given below in table 6.3 & 6.4:

Table 6.3 Accuracy, Runtime and Performance Measures of Machine Learning models on SAW for Dataset 01

SAW (100K)	Train_Acc	Test_Acc	Time	Performance By SAW
ANN	99.77	99.77	16.47	0.984
SVM	64.6	65.1	219.86	0.027
KNN	80.4	63.21	0.11	0.436
GPC	100	62.32	100.7	0.615

Table 6.4 Accuracy, Runtime and Performance Measures of Machine Learning models on SAW for Dataset 02

SAW (150K)	Train_Acc	Test_Acc	Time	Performance By SAW
ANN	99.79	99.79	23.46	0.995
SVM	64.74	64.95	566.32	0.027
KNN	80.82	63.94	0.18	0.443
GPC	100	63.03	308.44	0.592

6.1.3 VIKOR with Machine Learning Algorithms

The VIKOR and machine learning algorithms for datasets of two different dimensions are classified by PYTHON. The data sets contain 5 attributes and one label column. Defining all the input as x and output as y, the training accuracy, testing accuracy, and runtime for each dataset is determined by running the models five times and took the average value. We took 80 to 90 percent as training data and 10 to 20 percent as testing data. We have found the average testing accuracy, average training accuracy, average run time for individual datasets. Finally, The MCDM method VIKOR was applied on those average results and got the performance for each ML model. The results are given below in table 6.5 & 6.6:

Table 6.5 Accuracy, Runtime and Performance Measures of Machine Learning models on VIKOR for Dataset 01

VIKOR (100K)	Train_Acc	Test_Acc	Time	Performance By VIKOR
ANN	98.89	98.89	36.14	0.980
GPC	100	76.05	314.4	0.821
KNN	88.37	77.73	0.54	0.513
SVM	75.6	75.57	784.86	0

Table 6.6 Accuracy, Runtime and Performance Measures of Machine Learning models on VIKOR for Dataset 02

VIKOR (150K)	Train_Acc	Test_Acc	Time	Performance By VIKOR
ANN	98.26	98.26	13.56	0.925
GPC	100	73.42	113.49	0.701
SVM	91.93	92.04	117.51	0.335
KNN	88.07	77.2	0.11	0

6.1.4 Combined Classification with Machine Learning Algorithms

The results from the previous classification of three MCDM models are combined into one by the voting system which is previously discussed in Chapter 3, Paragraph 3.3. There SAW with ANN had given the highest performance among other MCDM and ML algorithm combinations. To minimize the conflict of classes in the voting system, the class predicted by SAW was considered to be the final class for that particular conflict. Then four ML algorithms were applied to the combined classes for individual datasets. The average testing accuracy, average training accuracy, average run time have been found for individual datasets. Finally, The MCDM method VIKOR was applied on those average results and got the performance for each ML model. The outcomes are given below presented in table 6.7 & 6.8:

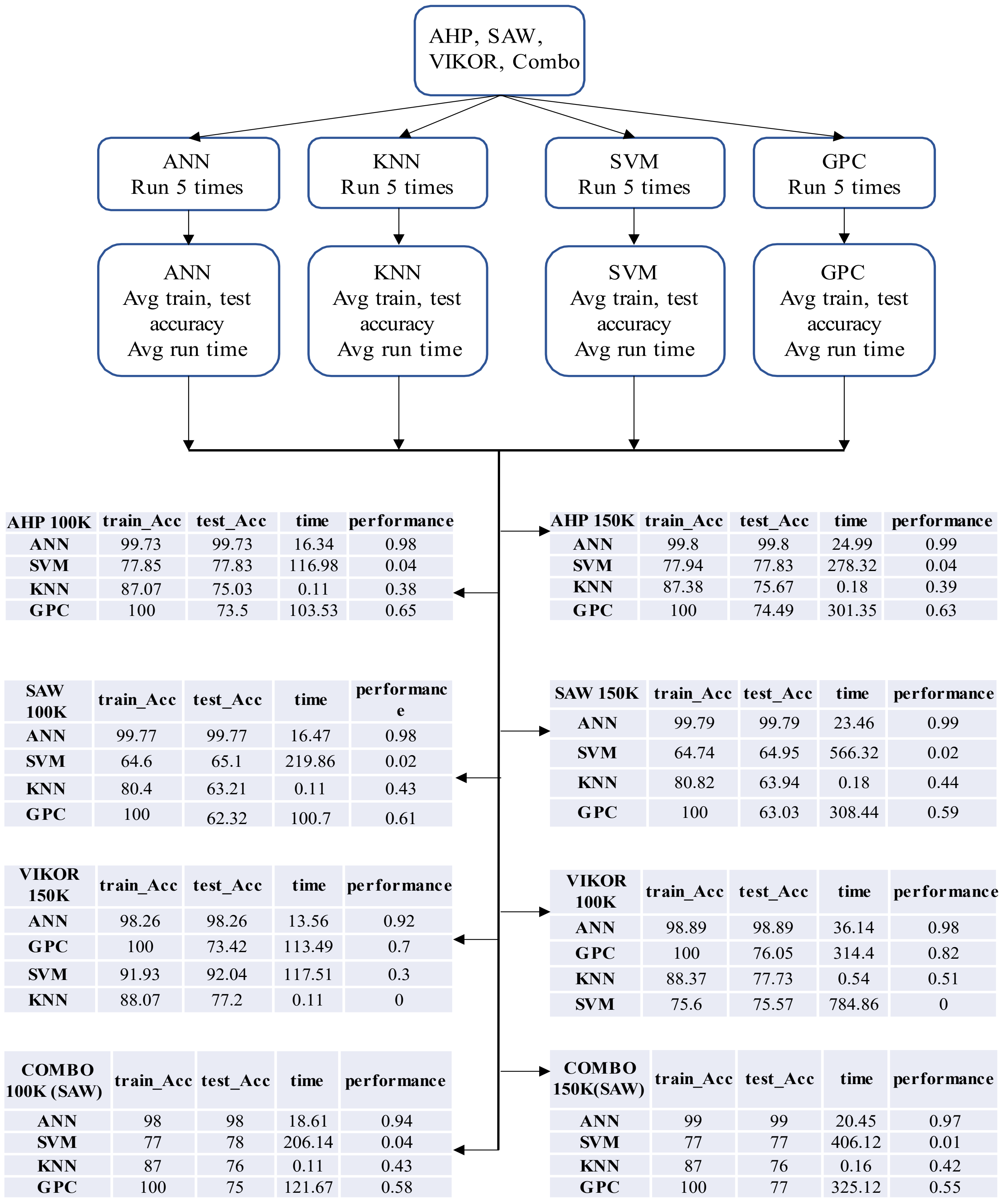
Table 6.7 Accuracy, Runtime and Performance Measures of Machine Learning models on Combined Class for Dataset 01

COMBO 100K (SAW)	Train_Acc	Test_Acc	Time	Performance By SAW
ANN	98	98	18.61	0.944
SVM	77	78	206.14	0.047
KNN	87	76	0.11	0.436
GPC	100	75	121.67	0.588

Table 6.8 Accuracy, Runtime and Performance Measures of Machine Learning models on Combined Class for Dataset 02

COMBO 150K (SAW)	**Train_Acc**	**Test_Acc**	**Time**	**Performance By SAW**
ANN	99	99	20.45	0.973
SVM	77	77	406.12	0.015
KNN	87	76	0.16	0.421
GPC	100	77	325.12	0.556

All tables from table 6.1 to 6.8 are presented together in figure 6.1.


AHP, SAW, VIKOR, Combo
ANN Run 5 times
KNN Run 5 times
SVM Run 5 times
GPC Run 5 times
ANN Avg train, test accuracy Avg run time
KNN Avg train, test accuracy Avg run time
SVM Avg train, test accuracy Avg run time
GPC Avg train, test accuracy Avg run time

AHP 100K	train_Acc	test_Acc	time	performance
ANN	99.73	99.73	16.34	0.98
SVM	77.85	77.83	116.98	0.04
KNN	87.07	75.03	0.11	0.38
GPC	100	73.5	103.53	0.65

AHP 150K	train_Acc	test_Acc	time	performance
ANN	99.8	99.8	24.99	0.99
SVM	77.94	77.83	278.32	0.04
KNN	87.38	75.67	0.18	0.39
GPC	100	74.49	301.35	0.63

SAW 100K	train_Acc	test_Acc	time	performanc e
ANN	99.77	99.77	16.47	0.98
SVM	64.6	65.1	219.86	0.02
KNN	80.4	63.21	0.11	0.43
GPC	100	62.32	100.7	0.61

SAW 150K	train_Acc	test_Acc	time	performance
ANN	99.79	99.79	23.46	0.99
SVM	64.74	64.95	566.32	0.02
KNN	80.82	63.94	0.18	0.44
GPC	100	63.03	308.44	0.59

VIKOR 150K	train_Acc	test_Acc	time	performance
ANN	98.26	98.26	13.56	0.92
GPC	100	73.42	113.49	0.7
SVM	91.93	92.04	117.51	0.3
KNN	88.07	77.2	0.11	0

VIKOR 100K	train_Acc	test_Acc	time	performance
ANN	98.89	98.89	36.14	0.98
GPC	100	76.05	314.4	0.82
KNN	88.37	77.73	0.54	0.51
SVM	75.6	75.57	784.86	0

COMBO 100K (SAW)	train_Acc	test_Acc	time	performance
ANN	98	98	18.61	0.94
SVM	77	78	206.14	0.04
KNN	87	76	0.11	0.43
GPC	100	75	121.67	0.58

COMBO 150K(SAW)	train_Acc	test_Acc	time	performance
ANN	99	99	20.45	0.97
SVM	77	77	406.12	0.01
KNN	87	76	0.16	0.42
GPC	100	77	325.12	0.55

Figure 6.1: Flowchart of the numerical illustrations

6.2 Results

From table 6.1 to table 6.8, all the highest performing MCDM-ML models are compiled to table 6.9. In table 6.9, the Train Accuracy and Runtime of each MCDM-ML method are utilized to get a fair performance score by Simple Additive Weighting (SAW), as Train Accuracy is a beneficial criterion and Runtime is a non-beneficial criterion. So, the Implementation of SAW was a better way out than the usual weighting average performance.

After compiling all the individual performances of MCDM-ML models for both datasets at table 6.9, it was found that Machine Learning Algorithm ANN had given higher performance value for each MCDM model among other Machine Learning Algorithms KNN, SVM, and GPC. So, ML model ANN has a great performing value regarding Accuracy and Runtime among other ML algorithms used for this study. ANN as an ML algorithm gives a satisfactory performance value whether the dataset dimension is small or large.

On the other hand, MCDM methods show an obscure output for different datasets. For a small dimensional dataset VIKOR with ANN gives the highest performing score of 0.989. where else for the large dimensional dataset, the combination of AHP, SAW, and VIKOR (by Voting System, Chapter 3, Paragraph 3) with ANN gives better feedback than individual MCDM methods. The combined MCDM model with ANN gives a performance score of 0.994.

Table 6.9 Overall Performance Measure by SAW for Both Datasets

Dataset 01 (100k)	MCDM	ML	TRAIN ACCURACY	RUN TIME	INDIVIDUAL PERFORMANCE	Performance By SAW
0	VIKOR (100K)	ANN	98.26	13.56	0.980	0.989
1	AHP (100K)	ANN	99.73	16.34	0.982	0.957
2	SAW (100K)	ANN	99.77	16.47	0.984	0.956
3	Combo (100k) (SAW)	ANN	98	18.61	0.944	0.919
Dataset 02 (150k)	MCDM	ML	TRAIN ACCURACY	RUN TIME	INDIVIDUAL PERFORMANCE	Performance By SAW
0	Combo (150k) (SAW)	ANN	99	20.45	0.973	0.994
1	SAW (150k)	ANN	99.79	23.46	0.995	0.968
2	AHP (150K)	ANN	99.8	24.99	0.991	0.955
3	VIKOR (150K)	ANN	98.89	36.14	0.925	0.885

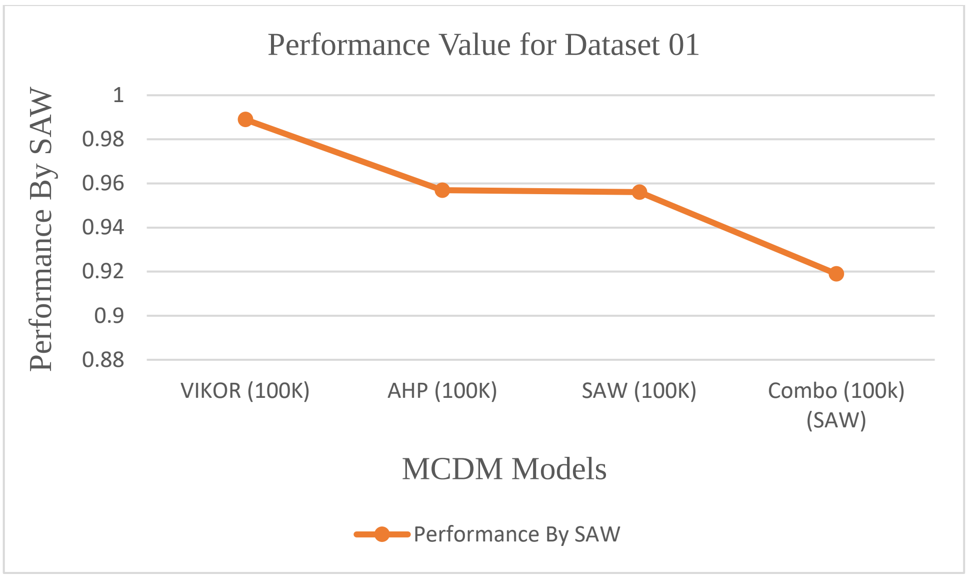


Figure 6.2: Performance Value Vs MCDM Models for Dataset 01

Figure 6.2 illustrates the relationship between different MCDM-ANN models and Performance by SAW for dataset 01 where AHP and SAW showed a moderate performance, but VIKOR performed eminently.

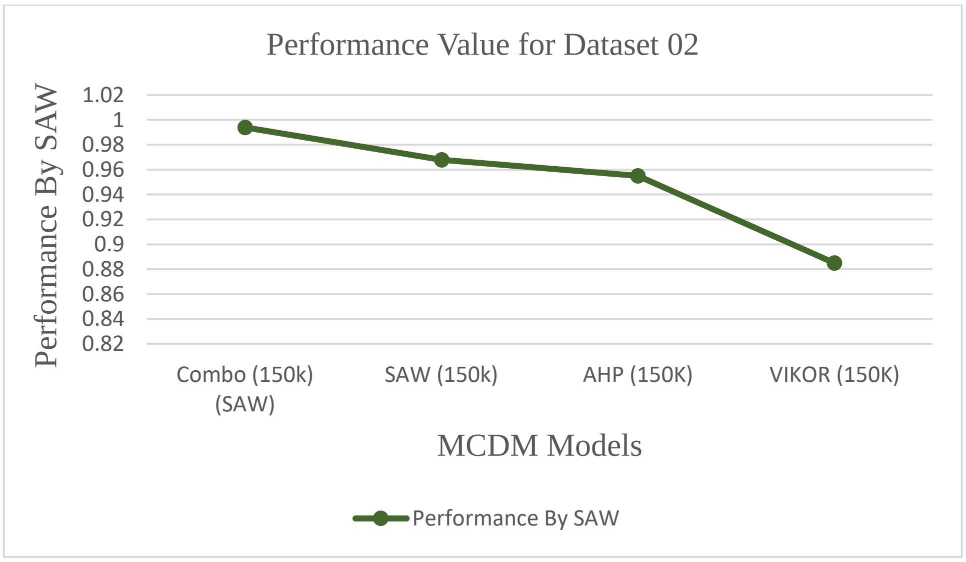


Figure 6.3: Performance Value Vs MCDM Models for Dataset 02

But Figure 6.3 indicates that as a matter of the relation between performances and different MCDM-ANN models of dataset 02, a Combination of all MCDM methods showed an eminent performance over individual MCDM models.

Chapter 7

Discussion

Inventory item classification is a mandatory job to do for any sort of manufacturing industry. ABC Analysis is a traditional tool to classify inventory items into three basic groups. But convectional ABC Analysis works based on annual cost consumption which gives a classification with a single feature.

Now ABC Analysis with MCDM methods gives more precise and absolute feedback as this time ABC Analysis takes multiple features on the count. Furthermore, Machine Learning Algorithms in a combination with MCDM models of ABC Analysis can easily predict classes for future items included in the inventory.

In this research, ML algorithms and MCDM methods are combinedly implemented to classify two different inventory datasets of multi-features. The results show that ANN as an ML algorithm has the most balanced performance among the other three ML algorithms. Whether dataset dimension is large or small, ANN has a moderate runtime and a better train & test accuracy. So, ANN may be reliable enough to predict the class of future included items to the inventory. Now for ABC classification of existing inventory items, MCDM methods were used to count on different features of inventory items. All three MCDM methods are implemented individually and combinedly for each dataset. For small dimensional datasets, VIKOR is more efficient to classify inventory items. Whereas, the Combined approach of the three MCDM methods gives a more precise classification if the dataset has a larger dimension.

This integrated approach of ABC Analysis with MCDM models and Machine Learning Algorithms will add a new dimension to inventory classification in the supply chain. This approach will pave the way to classify inventory items more specifically and will reduce the re-work time whenever new items are included in the inventory. If researchers do the same analysis with different MCDM models or different ML Algorithms, they may acquire different performance measures.

Reference

[1] N. F. Aziz, S. Sorooshian, and F. Mahmud, “MCDM-AHP method in decision makings,” *ARPN J. Eng. Appl. Sci.*, vol. 11, no. 11, pp. 7217–7220, 2016, doi: 10.1109/TIE.2013.2297315.

[2] G. Kou and D. Ergu, “AHP/ANP theory and its application in technological and economic development: the 90th anniversary of Thomas L. Saaty,” *Technol. Econ. Dev. Econ.*, vol. 22, no. 5, pp. 649–650, 2016, doi: 10.3846/20294913.2016.1202353.

[3] S. Radhika, D. S. Kumar, and D. Swapna, “VIKOR Method for Multi Criteria Decision Making in Academic Staff Selection,” vol. 3, no. 2, pp. 30–35, 2013.

[4] M. C. Yu, “Multi-criteria ABC analysis using artificial-intelligence-based classification techniques,” *Expert Syst. Appl.*, vol. 38, no. 4, pp. 3416–3421, 2011, doi: 10.1016/j.eswa.2010.08.127.

[5] P. Grant, “Assessment and Selection,” *Bus. Giv.*, 2014, doi: 10.1057/9780230355033.0018.

[6] M. Kuss and C. E. Rasmussen, “Assessing approximations for Gaussian process classification,” *Adv. Neural Inf. Process. Syst.*, pp. 699–706, 2005.

[7] “Discriminatory Analysis-Nonparametric Discrimination : Consistency Properties,” vol. 57, no. 3, pp. 238–247, 2014.

[8] N. S. Altman, “An introduction to kernel and nearest-neighbor nonparametric regression,” *Am. Stat.*, vol. 46, no. 3, pp. 175–185, 1992, doi: 10.1080/00031305.1992.10475879.

[9] F. Lolli, E. Balugani, A. Ishizaka, R. Gamberini, B. Rimini, and A. Regattieri, “Machine learning for multi-criteria inventory classification applied to intermittent demand,” *Prod. Plan. Control*, vol. 30, no. 1, pp. 76–89, 2019, doi: 10.1080/09537287.2018.1525506.

[10] B. E. Flores, D. L. Olson, and V. K. Dorai, “Management of multicriteria inventory classification,” *Math. Comput. Model.*, vol. 16, no. 12, pp. 71–82, 1992, doi: 10.1016/0895-7177(92)90021-C.

[11] R. Ramanathan, “ABC inventory classification with multiple-criteria using weighted linear optimization,” *Comput. Oper. Res.*, vol. 33, no. 3, pp. 695–700, 2006, doi: 10.1016/j.cor.2004.07.014.

[12] D. Sabaei, J. Erkoyuncu, and R. Roy, “A review of multi-criteria decision making methods for enhanced maintenance delivery,” *Procedia CIRP*, vol. 37, pp. 30–35, 2015, doi: 10.1016/j.procir.2015.08.086.

[13] S. Opricovic and G. H. Tzeng, “Compromise solution by MCDM methods: A comparative analysis of VIKOR and TOPSIS,” *Eur. J. Oper. Res.*, vol. 156, no. 2, pp. 445–455, 2004, doi: 10.1016/S0377-2217(03)00020-1.

[14] H. B. Kartal and F. Cebi, “Support Vector Machines for Multi-Attribute ABC Analysis,” *Int. J. Mach. Learn. Comput.*, no. January 2013, pp. 154–157, 2013, doi: 10.7763/ijmlc.2013.v3.292.

[15] W. L. Ng, “A simple classifier for multiple criteria ABC analysis,” *Eur. J. Oper. Res.*, vol. 177, no. 1, pp. 344–353, 2007, doi: 10.1016/j.ejor.2005.11.018.

[16] F. Y. Partovi and M. Anandarajan, “Classifying inventory using an artificial neural network approach,” *Comput. Ind. Eng.*, vol. 41, no. 4, pp. 389–404, 2002, doi: 10.1016/s0360-8352(01)00064-x.

[17] M. Braglia, A. Grassi, and R. Montanari, “Multi-attribute classification method for spare parts inventory management,” *J. Qual. Maint. Eng.*, vol. 10, no. 1, pp. 55–65, 2004, doi: 10.1108/13552510410526875.

[18] A. Kumar *et al.*, “A review of multi criteria decision making (MCDM) towards sustainable renewable energy development,” *Renew. Sustain. Energy Rev.*, vol. 69, no. November 2016, pp. 596–609, 2017, doi: 10.1016/j.rser.2016.11.191.

[19] M. R. Asadabadi, E. Chang, and M. Saberi, “Are MCDM methods useful? A critical review of Analytic Hierarchy Process (AHP) and Analytic Network Process (ANP),” *Cogent Eng.*, vol. 6, no. 1, 2019, doi: 10.1080/23311916.2019.1623153.

[20] A. Jahan, F. Mustapha, M. Y. Ismail, S. M. Sapuan, and M. Bahraminasab, “A comprehensive VIKOR method for material selection,” *Mater. Des.*, vol. 32, no. 3, pp. 1215–1221, 2011, doi: 10.1016/j.matdes.2010.10.015.

[21] I. Lajili, T. Ladhari, and Z. Babai, “Adaptive machine learning classifiers for the class imbalance problem in ABC inventory classification,” *ILS 2016 - 6th Int. Conf. Inf. Syst. Logist. Supply Chain*, pp. 1–8, 2016.

[22] S. H. Zanakis, A. Solomon, N. Wishart, and S. Dublish, “Multi-attribute decision making: A simulation comparison of select methods,” *Eur. J. Oper. Res.*, vol. 107, no. 3, pp. 507–529, 1998, doi: 10.1016/S0377-2217(97)00147-1.

[23] E. Ozcan and J. Atkin, “An Analysis of the Taguchi Method for Tuning a Memetic Algorithm with Reduced,” *31st Int. Symp. Isc. 2016*, vol. 1, pp. 12–20, 2016, doi:

10.1007/978-3-319-47217-1.

[24] T. M. Cover and P. E. Hart, “Nearest Neighbor Pattern Classification,” *IEEE Trans. Inf. Theory*, vol. 13, no. 1, pp. 21–27, 1967, doi: 10.1109/TIT.1967.1053964.

[25] M. P. Praveen, J. B. Simha, and R. Venkataram, “Techniques for Inventory Classification: A Review,” *Int. J. Res. Appl. Sci. Eng. Technol.*, vol. Vol. 4, no. X, pp. 508–518, 2016, [Online]. Available: www.ijraset.com.

[26] a a Balkema and L. De Haan, “Institute of Mathematical Statistics is collaborating with JSTOR to digitize, preserve, and extend access to The Annals of Probability. ® www.jstor.org,” *Statistics (Ber).*, vol. 2, no. 5, pp. 347–370, 1974, [Online]. Available: http://projecteuclid.org/euclid.aop/1176996548.

[27] A. F. Atiya and A. H. Abdel-gawad, “A New Monte Carlo Based Algorithm for the Gaussian Process Classification Problem,” no. June 2014, 2013.

[28] C. L. Hwang and K. Yoon, “Multiple Objective Decision Making - Methods and Applications,” *Lect. Notes Econ. Math. Syst.*, vol. 186, pp. 58–191, 1981, doi: 10.1007/978-3-642-45511-7.

[29] M. Marcus and H. Minc, *M. Marcus and H. Minc, Introduction to Linear Algebra.* 2020.

[30] S. I. Gass and T. Rapcsák, “Singular value decomposition in AHP,” *Eur. J. Oper. Res.*, vol. 154, no. 3, pp. 573–584, 2004, doi: 10.1016/S0377-2217(02)00755-5.

[31] J. R. S. C. Mateo, “Multi-Criteria Analysis in the Renewable Energy Industry,” *Green Energy Technol.*, vol. 83, pp. 7–10, 2012, doi: 10.1007/978-1-4471-2346-0.

[32] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. 2000.

[33] D. P. Mandic and V. S. L. Goh, “Adaptive and Learning Systems for Signal, Processing, Communications, and Control,” *Complex Valued Nonlinear Adaptive Filters*. pp. 325–325, 2009, doi: 10.1002/9780470742624.scard.

[34] E. Osuna, R. Freund, and F. Girosi, “Support Vector Machines : Training and Applications,” *Massachusetts Inst. Technol.*, vol. 9217041, no. 1602, 1997.

[35] Min-Ling Zhang and Zhi-Hua Zhou, “A k-nearest neighbor based algorithm for multi-label classification,” no. December, pp. 718-721 Vol. 2, 2005, doi: 10.1109/grc.2005.1547385.

[36] G. Bonaccorso, *Mastering Machine Learning Algorithms*, no. October 2013. 2018.

[37] C. C. Aggarwal, A. Hinneburg, and D. A. Keim, “On the surprising behavior of distance metrics in high dimensional space,” *Lect. Notes Comput. Sci. (including*

Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 1973, pp. 420–434, 2001, doi: 10.1007/3-540-44503-x_27.

[38] H. C. Kim, D. Kim, Z. Ghahramani, and S. Y. Bang, "Appearance-based gender classification with Gaussian processes," *Pattern Recognit. Lett.*, vol. 27, no. 6, pp. 618–626, 2006, doi: 10.1016/j.patrec.2005.09.027.

[39] Y. Peng, G. Kou, G. Wang, and Y. Shi, "FAMCDM: A fusion approach of MCDM methods to rank multiclass classification algorithms," *Omega*, vol. 39, no. 6, pp. 677–689, 2011, doi: 10.1016/j.omega.2011.01.009.

Appendix

Machine Specifications:

Machine Name: Virtual Machine powered by Kaggle

Processor: 4 Core

Clockspeed : 2.2 GHz

Ram: 16 GB

The PYTHON codes for inventory Dataset 01 are shown in the next page. The same codes are followed for inventory dataset 02 for the prediction.

1.AHPML_v4.0,100k

Importing Libraries

In [1]:

```
import pandas as pd
import numpy as np
import seaborn as sns

import tensorflow as tf
import keras
from keras.utils import to_categorical
from keras.models import Sequential
from keras.optimizers import SGD
from keras.layers import Dense

from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.neighbors import KNeighborsClassifier

from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import time
```

Importing Data

In [2]:

```
data = pd.read_csv('../input/thesis/dumy100000.csv')
```

MCDM

AHP

In [3]:

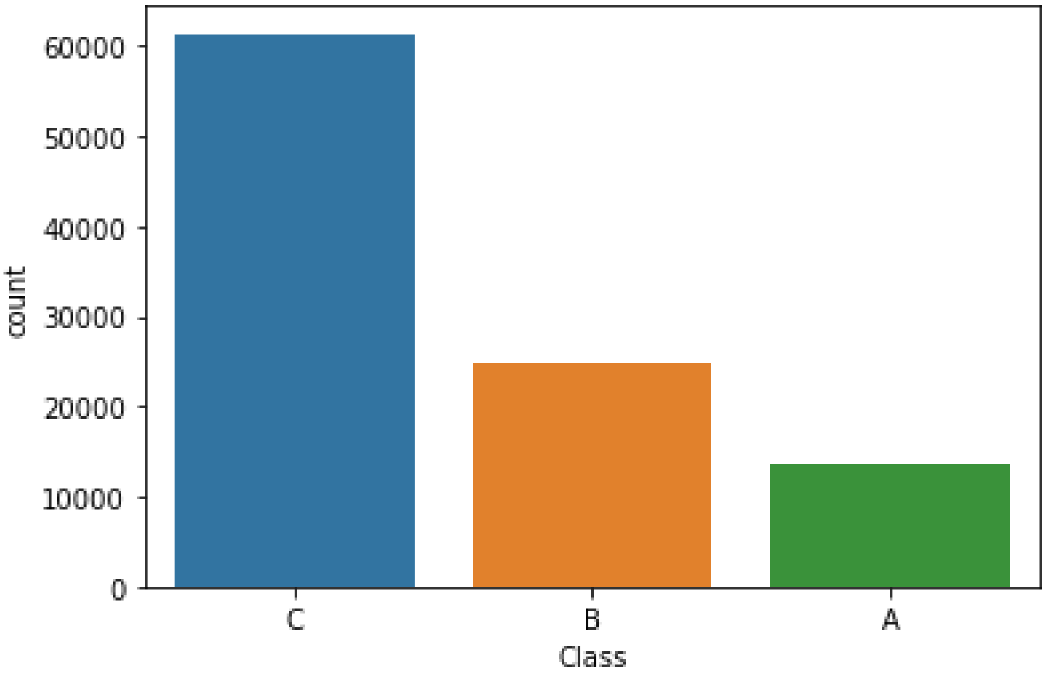
```
d_m = pd.read_csv('../input/pairwise/pairWiseMatrix.csv',index_col=0)

for i in range(5):
    d_m.iloc[:,i]=d_m.iloc[:,i]/d_m.iloc[:,i].sum()

d_m['criteria_weight'] = d_m.mean(axis=1)

dmat = pd.read_csv('../input/pairwise/pairWiseMatrix.csv',index_col=0)
```

```
for i in range(5):
    dmat.iloc[:,i] = dmat.iloc[:,i] * d_m.iloc[i,5]

dmat['weighted_sum_value'] = dmat.sum(axis=1)

dmat['weighted_sum_value/criteria_weight'] = dmat['weighted_sum_value']/d_
m['criteria_weight']

lamda = dmat['weighted_sum_value/criteria_weight'].mean()
r = len(d_m.index)
CI = (lamda-r)/(r-1)
k= pd.read_csv('../input/pairwise/RCI.csv')
for i in range(15):
    if i==r-1:
        RI=k.iloc[i,1]

CR=CI/RI

if CR < 0.1:
    print("The Matrix is Consistent.")
else:
     print("The Matrix is In-Consistent.")

dn = 0
dn = data.copy()
del dn['Unnamed: 0']
d1 = dn.copy()

d2 = pd.DataFrame(columns = d1.columns)
d2 = d2.T
d2["max"] = 0
d2["min"] = 0
d2 = d2.T

for i in range(len(d2.columns)):
    d2.iloc[0,i] = d1.iloc[:,i].max()
    d2.iloc[1,i] = d1.iloc[:,i].min()

# max-min normalizer = 0
# linear normalizer = 1
normalizer = 0

if normalizer == 0 :
    for i in range(len(d1.columns)):
        if i==0 or i==3: #Non Beneficial
            d1.iloc[:,i] = (d2.iloc[0,i]-d1.iloc[:,i])/(d2.iloc[0,i] - d2.
iloc[1,i])

        else: #Beneficial
            d1.iloc[:,i] = (d1.iloc[:,i]-d2.iloc[1,i])/(d2.iloc[0,i] - d2.
iloc[1,i])

else:
    for i in range(len(d1.columns)):
        if i==0 or i==3: #Non Beneficial
            d1.iloc[:,i] = d2.iloc[1,i]/d1.iloc[:,i]

        else: #Beneficial
            d1.iloc[:,i] = d1.iloc[:,i]/ d2.iloc[0,i]

w = np.array(d_m["criteria_weight"]).reshape(1,5)
d1 = d1*w

d1['performance'] = d1.iloc[:].sum(axis=1)

dn['performance'] = d1['performance']
dn = dn.sort_values(by=['performance'], ascending=True)

dn['RunCumCost']   = dn['performance'].cumsum()
TotSum             = dn['performance'].sum()
dn['RunCostPerc']  = (dn['RunCumCost']/TotSum)*100

dn['Rank']         = dn['RunCostPerc'].rank()
dn['RunItemCum']   = dn['Rank'].cumsum()
TotItemSum         = dn['Rank'].sum()
dn['RunItemPerc']  = (dn['RunItemCum']/TotItemSum)*100

def ABC_segmentation(perc):
    '''
   A - top 20%, C - last 50% and B - between A & C
    '''
    if perc > 80 :
        return 'A'
    elif perc >= 50 and perc <= 80:
        return 'B'
    elif perc < 50:
        return 'C'

dn['Class'] = dn['RunCostPerc'].apply(ABC_segmentation)

ax = sns.countplot(x = dn['Class'],data = dn,label= 'Count')
dn['Class'].value_counts()

dn['productid'] = dn.index
dn = dn.sort_values(by=['productid'], ascending=True)
del dn['productid']
AHP_Class = dn['Class']

C,B,A = dn['Class'].value_counts()
print(round(100*A/dn['Class'].value_counts().sum(),2),"% ->","A:",A)
print(round(100*B/dn['Class'].value_counts().sum(),2),"% ->","B:",B)
print(round(100*C/dn['Class'].value_counts().sum(),2),"% ->","C:",C)
print("Total: ",dn['Class'].value_counts().sum())
dn.head()
```

```
The Matrix is Consistent.
13.65 % -> A: 13648
24.95 % -> B: 24953
61.4 % -> C: 61399
Total:  100000
```

Out [3]:

	mfg	sell	demand	fcost	rma	performance	RunCumCost	RunCostPerc	Rank	RunItemCum	RunItemPerc	Class
0	71	156.91	1595	49.7	3	0.276224	1494.307675	2.985227	6658.0	2.216781e+07	0.443352	C
1	474	1047.54	6877	331.8	3	0.873845	49945.574810	99.777899	99874.0	4.987458e+09	99.748160	A
2	70	154.70	5182	49.0	3	0.471743	15481.763973	30.928423	42637.0	9.089782e+08	18.179382	C
3	417	921.57	3914	291.9	3	0.668061	40135.751226	80.180495	86488.0	3.740130e+09	74.801858	A
4	239	528.19	4496	167.3	2	0.486306	17225.943313	34.412827	46278.0	1.070850e+09	21.416781	C



Machine Learning

In [4]:

```
iteration = 5
row_col = np.zeros([iteration,])

acc = pd.DataFrame()
acc['train_Acc'] = row_col
```

```
acc['test_Acc']  = row_col
acc['time']      = row_col
acc['test_size'] = row_col
start_time  = 0
end_time    = 0
train_score = 0
test_score  = 0

#split_size
n = .1
```

ANN

In [5]:

```
for i in range(iteration):

    X = dn.iloc[:,0:5].astype('float32')
    y = dn['Class'] #.iloc[0:100,-1]

    encoder = LabelEncoder()
    encoder.fit(y)
    y = encoder.transform(y)

    y = to_categorical(y)

    X_train,X_test,y_train,y_test = train_test_split(X, y ,test_size = n,
random_state=0)
    X_train,X_val,y_train,y_val = train_test_split(X_train, y_train ,test_
size = n, random_state=0)

    sc = StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test  = sc.fit_transform(X_test)
    X_val  = sc.fit_transform(X_val)

    #time start
    start_time = time.time()

    model = Sequential()
    model.add(Dense(5, activation='relu', input_shape=(X_train.shape[1],))
)
    model.add(Dense(3, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metri
cs=['accuracy'])
    model.fit(X_train,y_train,batch_size = 128, epochs = 20, validation_da
ta = (X_val,y_val))

    #time end
    end_time = time.time()

    train_score = model.evaluate(X_train,y_train,verbose=0)
    test_score = model.evaluate(X_test,y_test,verbose=0)

    #time calculation
    acc.iloc[i,2] = end_time - start_time

    #Accuracy
    acc.iloc[i,0] = train_score[1]*100
    acc.iloc[i,1] = train_score[1]*100

    #test size
    acc.iloc[i,3] = n*100

    if n == .1:
        n = .2
    else :
        n = .1
```

```
Epoch 1/20
633/633 [==============================] - 2s 2ms/step - loss: 0.9913 - ac
curacy: 0.6493 - val_loss: 0.4076 - val_accuracy: 0.8039
Epoch 2/20
633/633 [==============================] - 1s 1ms/step - loss: 0.3372 - ac
curacy: 0.8769 - val_loss: 0.2347 - val_accuracy: 0.9396
Epoch 3/20
633/633 [==============================] - 1s 1ms/step - loss: 0.2175 - ac
curacy: 0.9532 - val_loss: 0.1721 - val_accuracy: 0.9718
Epoch 4/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1609 - ac
curacy: 0.9759 - val_loss: 0.1352 - val_accuracy: 0.9793
Epoch 5/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1273 - ac
curacy: 0.9832 - val_loss: 0.1114 - val_accuracy: 0.9853
Epoch 6/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1048 - ac
curacy: 0.9869 - val_loss: 0.0949 - val_accuracy: 0.9868
Epoch 7/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0897 - ac
curacy: 0.9893 - val_loss: 0.0832 - val_accuracy: 0.9893
Epoch 8/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0787 - ac
curacy: 0.9908 - val_loss: 0.0739 - val_accuracy: 0.9912
Epoch 9/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0702 - ac
curacy: 0.9927 - val_loss: 0.0661 - val_accuracy: 0.9923
Epoch 10/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0624 - ac
curacy: 0.9939 - val_loss: 0.0601 - val_accuracy: 0.9931
Epoch 11/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0565 - ac
curacy: 0.9946 - val_loss: 0.0560 - val_accuracy: 0.9932
Epoch 12/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0523 - ac
curacy: 0.9946 - val_loss: 0.0518 - val_accuracy: 0.9946
Epoch 13/20
```

```
633/633 [==============================] - 1s 1ms/step - loss: 0.0487 - ac
curacy: 0.9949 - val_loss: 0.0482 - val_accuracy: 0.9950
Epoch 14/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0452 - ac
curacy: 0.9950 - val_loss: 0.0452 - val_accuracy: 0.9950
Epoch 15/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0424 - ac
curacy: 0.9955 - val_loss: 0.0430 - val_accuracy: 0.9959
Epoch 16/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0405 - ac
curacy: 0.9962 - val_loss: 0.0439 - val_accuracy: 0.9937
Epoch 17/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0386 - ac
curacy: 0.9963 - val_loss: 0.0394 - val_accuracy: 0.9951
Epoch 18/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0361 - ac
curacy: 0.9965 - val_loss: 0.0375 - val_accuracy: 0.9961
Epoch 19/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0343 - ac
curacy: 0.9963 - val_loss: 0.0359 - val_accuracy: 0.9967
Epoch 20/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0334 - ac
curacy: 0.9970 - val_loss: 0.0351 - val_accuracy: 0.9958
Epoch 1/20
500/500 [==============================] - 1s 2ms/step - loss: 1.0377 - ac
curacy: 0.5978 - val_loss: 0.7219 - val_accuracy: 0.7456
Epoch 2/20
500/500 [==============================] - 1s 1ms/step - loss: 0.6490 - ac
curacy: 0.7882 - val_loss: 0.4521 - val_accuracy: 0.9374
Epoch 3/20
500/500 [==============================] - 1s 1ms/step - loss: 0.3987 - ac
curacy: 0.9487 - val_loss: 0.2658 - val_accuracy: 0.9659
Epoch 4/20
500/500 [==============================] - 1s 1ms/step - loss: 0.2369 - ac
curacy: 0.9743 - val_loss: 0.1712 - val_accuracy: 0.9758
Epoch 5/20
500/500 [==============================] - 1s 1ms/step - loss: 0.1553 - ac
curacy: 0.9796 - val_loss: 0.1226 - val_accuracy: 0.9804
Epoch 6/20
500/500 [==============================] - 1s 1ms/step - loss: 0.1125 - ac
curacy: 0.9833 - val_loss: 0.0930 - val_accuracy: 0.9861
Epoch 7/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0877 - ac
curacy: 0.9863 - val_loss: 0.0739 - val_accuracy: 0.9896
Epoch 8/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0692 - ac
curacy: 0.9901 - val_loss: 0.0608 - val_accuracy: 0.9909
Epoch 9/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0568 - ac
curacy: 0.9930 - val_loss: 0.0503 - val_accuracy: 0.9926
Epoch 10/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0468 - ac
curacy: 0.9941 - val_loss: 0.0460 - val_accuracy: 0.9891
Epoch 11/20
```

```
500/500 [==============================] - 1s 1ms/step - loss: 0.0414 - ac
curacy: 0.9942 - val_loss: 0.0378 - val_accuracy: 0.9933
Epoch 12/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0357 - ac
curacy: 0.9957 - val_loss: 0.0347 - val_accuracy: 0.9929
Epoch 13/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0313 - ac
curacy: 0.9961 - val_loss: 0.0311 - val_accuracy: 0.9947
Epoch 14/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0286 - ac
curacy: 0.9960 - val_loss: 0.0296 - val_accuracy: 0.9914
Epoch 15/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0260 - ac
curacy: 0.9967 - val_loss: 0.0276 - val_accuracy: 0.9912
Epoch 16/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0234 - ac
curacy: 0.9970 - val_loss: 0.0240 - val_accuracy: 0.9941
Epoch 17/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0215 - ac
curacy: 0.9971 - val_loss: 0.0223 - val_accuracy: 0.9951
Epoch 18/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0201 - ac
curacy: 0.9976 - val_loss: 0.0216 - val_accuracy: 0.9939
Epoch 19/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0188 - ac
curacy: 0.9976 - val_loss: 0.0192 - val_accuracy: 0.9959
Epoch 20/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0175 - ac
curacy: 0.9976 - val_loss: 0.0195 - val_accuracy: 0.9935
Epoch 1/20
633/633 [==============================] - 1s 1ms/step - loss: 0.9835 - ac
curacy: 0.5163 - val_loss: 0.4981 - val_accuracy: 0.8269
Epoch 2/20
633/633 [==============================] - 1s 1ms/step - loss: 0.4624 - ac
curacy: 0.8298 - val_loss: 0.3772 - val_accuracy: 0.8527
Epoch 3/20
633/633 [==============================] - 1s 1ms/step - loss: 0.3669 - ac
curacy: 0.8504 - val_loss: 0.3309 - val_accuracy: 0.8559
Epoch 4/20
633/633 [==============================] - 1s 1ms/step - loss: 0.3264 - ac
curacy: 0.8547 - val_loss: 0.3046 - val_accuracy: 0.8588
Epoch 5/20
633/633 [==============================] - 1s 1ms/step - loss: 0.3023 - ac
curacy: 0.8574 - val_loss: 0.2880 - val_accuracy: 0.8604
Epoch 6/20
633/633 [==============================] - 1s 1ms/step - loss: 0.2884 - ac
curacy: 0.8583 - val_loss: 0.2778 - val_accuracy: 0.8592
Epoch 7/20
633/633 [==============================] - 1s 1ms/step - loss: 0.2764 - ac
curacy: 0.8589 - val_loss: 0.2693 - val_accuracy: 0.8599
Epoch 8/20
633/633 [==============================] - 1s 1ms/step - loss: 0.2693 - ac
curacy: 0.8587 - val_loss: 0.2633 - val_accuracy: 0.8598
Epoch 9/20
```

```
633/633 [==============================] - 1s 1ms/step - loss: 0.2628 - ac
curacy: 0.8580 - val_loss: 0.2540 - val_accuracy: 0.8572
Epoch 10/20
633/633 [==============================] - 1s 1ms/step - loss: 0.2447 - ac
curacy: 0.8589 - val_loss: 0.1625 - val_accuracy: 0.9637
Epoch 11/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1281 - ac
curacy: 0.9765 - val_loss: 0.0728 - val_accuracy: 0.9903
Epoch 12/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0632 - ac
curacy: 0.9910 - val_loss: 0.0460 - val_accuracy: 0.9924
Epoch 13/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0401 - ac
curacy: 0.9964 - val_loss: 0.0329 - val_accuracy: 0.9962
Epoch 14/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0299 - ac
curacy: 0.9973 - val_loss: 0.0270 - val_accuracy: 0.9968
Epoch 15/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0248 - ac
curacy: 0.9974 - val_loss: 0.0230 - val_accuracy: 0.9964
Epoch 16/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0209 - ac
curacy: 0.9979 - val_loss: 0.0202 - val_accuracy: 0.9964
Epoch 17/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0189 - ac
curacy: 0.9979 - val_loss: 0.0193 - val_accuracy: 0.9956
Epoch 18/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0167 - ac
curacy: 0.9977 - val_loss: 0.0159 - val_accuracy: 0.9969
Epoch 19/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0148 - ac
curacy: 0.9979 - val_loss: 0.0151 - val_accuracy: 0.9963
Epoch 20/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0140 - ac
curacy: 0.9979 - val_loss: 0.0147 - val_accuracy: 0.9963
Epoch 1/20
500/500 [==============================] - 1s 2ms/step - loss: 0.9322 - ac
curacy: 0.5652 - val_loss: 0.4591 - val_accuracy: 0.7883
Epoch 2/20
500/500 [==============================] - 1s 1ms/step - loss: 0.3938 - ac
curacy: 0.8237 - val_loss: 0.2408 - val_accuracy: 0.9298
Epoch 3/20
500/500 [==============================] - 1s 1ms/step - loss: 0.2071 - ac
curacy: 0.9368 - val_loss: 0.1459 - val_accuracy: 0.9522
Epoch 4/20
500/500 [==============================] - 1s 1ms/step - loss: 0.1306 - ac
curacy: 0.9592 - val_loss: 0.0962 - val_accuracy: 0.9711
Epoch 5/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0861 - ac
curacy: 0.9741 - val_loss: 0.0632 - val_accuracy: 0.9898
Epoch 6/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0583 - ac
curacy: 0.9919 - val_loss: 0.0497 - val_accuracy: 0.9891
Epoch 7/20
```

```
500/500 [==============================] - 1s 1ms/step - loss: 0.0465 - ac
curacy: 0.9956 - val_loss: 0.0401 - val_accuracy: 0.9947
Epoch 8/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0375 - ac
curacy: 0.9969 - val_loss: 0.0346 - val_accuracy: 0.9930
Epoch 9/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0326 - ac
curacy: 0.9968 - val_loss: 0.0312 - val_accuracy: 0.9931
Epoch 10/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0277 - ac
curacy: 0.9974 - val_loss: 0.0271 - val_accuracy: 0.9954
Epoch 11/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0263 - ac
curacy: 0.9973 - val_loss: 0.0268 - val_accuracy: 0.9919
Epoch 12/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0234 - ac
curacy: 0.9974 - val_loss: 0.0259 - val_accuracy: 0.9904
Epoch 13/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0216 - ac
curacy: 0.9977 - val_loss: 0.0211 - val_accuracy: 0.9961
Epoch 14/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0208 - ac
curacy: 0.9976 - val_loss: 0.0217 - val_accuracy: 0.9933
Epoch 15/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0187 - ac
curacy: 0.9978 - val_loss: 0.0191 - val_accuracy: 0.9946
Epoch 16/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0182 - ac
curacy: 0.9975 - val_loss: 0.0199 - val_accuracy: 0.9928
Epoch 17/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0165 - ac
curacy: 0.9979 - val_loss: 0.0180 - val_accuracy: 0.9944
Epoch 18/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0157 - ac
curacy: 0.9979 - val_loss: 0.0188 - val_accuracy: 0.9927
Epoch 19/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0151 - ac
curacy: 0.9977 - val_loss: 0.0152 - val_accuracy: 0.9953
Epoch 20/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0147 - ac
curacy: 0.9981 - val_loss: 0.0164 - val_accuracy: 0.9940
Epoch 1/20
633/633 [==============================] - 1s 1ms/step - loss: 0.8979 - ac
curacy: 0.5528 - val_loss: 0.3774 - val_accuracy: 0.8407
Epoch 2/20
633/633 [==============================] - 1s 1ms/step - loss: 0.3084 - ac
curacy: 0.8682 - val_loss: 0.1528 - val_accuracy: 0.9690
Epoch 3/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1297 - ac
curacy: 0.9718 - val_loss: 0.0843 - val_accuracy: 0.9814
Epoch 4/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0761 - ac
curacy: 0.9839 - val_loss: 0.0585 - val_accuracy: 0.9913
Epoch 5/20
```

```
633/633 [==============================] - 1s 1ms/step - loss: 0.0538 - ac
curacy: 0.9911 - val_loss: 0.0454 - val_accuracy: 0.9938
Epoch 6/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0419 - ac
curacy: 0.9958 - val_loss: 0.0369 - val_accuracy: 0.9961
Epoch 7/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0350 - ac
curacy: 0.9966 - val_loss: 0.0316 - val_accuracy: 0.9963
Epoch 8/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0294 - ac
curacy: 0.9969 - val_loss: 0.0274 - val_accuracy: 0.9977
Epoch 9/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0266 - ac
curacy: 0.9974 - val_loss: 0.0255 - val_accuracy: 0.9932
Epoch 10/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0236 - ac
curacy: 0.9968 - val_loss: 0.0224 - val_accuracy: 0.9969
Epoch 11/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0216 - ac
curacy: 0.9973 - val_loss: 0.0211 - val_accuracy: 0.9972
Epoch 12/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0192 - ac
curacy: 0.9976 - val_loss: 0.0195 - val_accuracy: 0.9969
Epoch 13/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0182 - ac
curacy: 0.9977 - val_loss: 0.0182 - val_accuracy: 0.9967
Epoch 14/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0169 - ac
curacy: 0.9979 - val_loss: 0.0166 - val_accuracy: 0.9974
Epoch 15/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0158 - ac
curacy: 0.9980 - val_loss: 0.0170 - val_accuracy: 0.9951
Epoch 16/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0144 - ac
curacy: 0.9981 - val_loss: 0.0156 - val_accuracy: 0.9968
Epoch 17/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0136 - ac
curacy: 0.9979 - val_loss: 0.0153 - val_accuracy: 0.9959
Epoch 18/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0129 - ac
curacy: 0.9980 - val_loss: 0.0135 - val_accuracy: 0.9978
Epoch 19/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0126 - ac
curacy: 0.9980 - val_loss: 0.0131 - val_accuracy: 0.9973
Epoch 20/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0123 - ac
curacy: 0.9981 - val_loss: 0.0122 - val_accuracy: 0.9980
```

In [6]:

```
acc = acc.round(2)
acc
```

Out [6]:

	train_Acc	test_Acc	time	test_size
0	99.75	99.75	17.00	10.0
1	99.73	99.73	13.61	20.0
2	99.81	99.81	16.04	10.0
3	99.91	99.91	13.77	20.0
4	99.87	99.87	16.25	10.0

In [7]:

```
ANNAcc = acc.mean()
ANNAcc
```

Out [7]:

```
train_Acc    99.814
test_Acc     99.814
time         15.334
test_size    14.000
dtype: float64
```

SVM

In [8]:

```
n = .1
for i in range(iteration):

    X = dn.iloc[:,0:5].astype('float32')
    y = dn['Class']

    encoder = LabelEncoder()
    encoder.fit(y)
    y = encoder.transform(y)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
n, random_state=0)

    #time start
    start_time = time.time()



    model = SVC(kernel='poly')
    model.fit(X_train,y_train)

    #time end
    end_time = time.time()

    train_score = model.score(X_train,y_train)
    test_score  = model.score(X_test,y_test)

    #time calculation
    acc.iloc[i,2] = end_time - start_time

    #Accuracy
    acc.iloc[i,0] = train_score*100
    acc.iloc[i,1] = test_score*100

    #test size
    acc.iloc[i,3] = n*100

    if n == .1:
        n = .2
    else :
        n = .1
    print(i+1," run")
```

```
1  run
2  run
3  run
4  run
5  run
```

In [9]:

```
acc = acc.round(2)
acc
```

```
Out [9]:
```

	train_Acc	test_Acc	time	test_size
0	77.87	77.82	122.13	10.0
1	77.83	77.84	97.32	20.0
2	77.87	77.82	123.19	10.0
3	77.83	77.84	97.60	20.0

4	77.87	77.82	122.43	10.0

In [10]:

```
SVMAcc = acc.mean()
SVMAcc
```

Out [10]:

```
train_Acc      77.854
test_Acc       77.828
time          112.534
test_size      14.000
dtype: float64
```

KNN

In [11]:

```
n = .1
for i in range(iteration):

    X = dn.iloc[:,0:5].astype('float32')
    y = dn['Class']

    encoder = LabelEncoder()
    encoder.fit(y)
    y = encoder.transform(y)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
n, random_state=0)


    #time start
    start_time = time.time()

    model = KNeighborsClassifier(n_neighbors=3)
    model.fit(X_train,y_train)

    #time end
    end_time = time.time()

    train_score = model.score(X_train,y_train)
    test_score  = model.score(X_test,y_test)

    #time calculation
    acc.iloc[i,2] = end_time - start_time

    #Accuracy
    acc.iloc[i,0] = train_score*100
    acc.iloc[i,1] = test_score*100

    #test size
    acc.iloc[i,3] = n*100

    if n == .1:
        n = .2
    else :
        n = .1
    print(i+1," run")
```

```
1  run
2  run
3  run
4  run
5  run
```

In [12]:

```
acc = acc.round(2)
acc
```

Out [12]:

	train_Acc	test_Acc	time	test_size
0	87.11	75.05	0.11	10.0
1	87.00	74.99	0.09	20.0
2	87.11	75.05	0.11	10.0
3	87.00	74.99	0.09	20.0
4	87.11	75.05	0.11	10.0

In [13]:

```
KNNAcc = acc.mean()
KNNAcc
```

Out [13]:

```
train_Acc    87.066
test_Acc     75.026
time          0.102
test_size    14.000
dtype: float64
```

GPC

In [14]:

```
n = .1
for i in range(iteration):
    X = dn.iloc[:,0:5].astype('float32')
    y = dn['Class']

    encoder = LabelEncoder()
    encoder.fit(y)
    y = encoder.transform(y)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
n, random_state=0)


    #time start
    start_time = time.time()

    model = GaussianProcessClassifier(multi_class='one_vs_one')
    model.fit(X_train,y_train)

    #time end
    end_time = time.time()

    train_score = model.score(X_train,y_train)
    test_score  = model.score(X_test,y_test)

    #time calculation
    acc.iloc[i,2] = end_time - start_time

    #Accuracy
    acc.iloc[i,0] = train_score*100
    acc.iloc[i,1] = test_score*100

    #test size
    acc.iloc[i,3] = n*100

    if n == .1:
        n = .2
    else :
        n = .1
    print(i+1," run")
```

```
1  run
2  run
3  run
4  run
5  run
```

In [15]:

```
acc = acc.round(2)
acc
```

Out [15]:

	train_Acc	test_Acc	time	test_size

0	100.0	73.60	105.62	10.0
1	100.0	73.35	79.16	20.0
2	100.0	73.60	106.46	10.0
3	100.0	73.35	79.50	20.0
4	100.0	73.60	105.55	10.0

In [16]:

```
GPCAcc = acc.mean()
GPCAcc
```

Out [16]:

```
train_Acc    100.000
test_Acc      73.500
time          95.258
test_size     14.000
dtype: float64
```

In [17]:

```
performance = {
    "Performance" : ['Avg Train Accuracy','Avg Test Accuracy','Avg Run Tim
e','Avg Test size'],
    "ANN"          : ANNAcc,
    "SVM"          : SVMAcc,
    "KNN"          : KNNAcc,
    "GPC"          : GPCAcc
}
Performance = pd.DataFrame(performance)
Performance
```

Out [17]:

	Performance	ANN	SVM	KNN	GPC
train_Acc	Avg Train Accuracy	99.814	77.854	87.066	100.000
test_Acc	Avg Test Accuracy	99.814	77.828	75.026	73.500
time	Avg Run Time	15.334	112.534	0.102	95.258
test_size	Avg Test size	14.000	14.000	14.000	14.000

In [18]:

```
pm = np.array([1,3,5,1/3,1,3,1/5,1/3,1]).reshape(3,3) #consistancy matrix
pm = pd.DataFrame(pm)


for i in range(3):
    pm.iloc[:,i]=pm.iloc[:,i]/pm.iloc[:,i].sum()

pm['criteria_weight'] = pm.mean(axis=1)

pmat = np.array([1,3,5,1/3,1,3,1/5,1/3,1]).reshape(3,3)#pd.read_csv('../in
put/pairwise/pairWiseMatrix.csv',index_col=0)
pmat = pd.DataFrame(pmat)

for i in range(3):
    pmat.iloc[:,i] = pmat.iloc[:,i] * pm.iloc[i,3]

pmat['weighted_sum_value'] = pmat.sum(axis=1)

pmat['weighted_sum_value/criteria_weight'] = pmat['weighted_sum_value']/pm
['criteria_weight']

lamda = pmat['weighted_sum_value/criteria_weight'].mean()
r = len(pm.index)
CI = (lamda-r)/(r-1)
k= pd.read_csv('../input/pairwise/RCI.csv')
for i in range(15):
    if i==r-1:
        RI=k.iloc[i,1]

CR=CI/RI

if CR < 0.1:
    print("The Matrix is Consistent.")
else:
     print("The Matrix is In-Consistent.")
```

```
The Matrix is Consistent.
```

In [19]:

```
p1 = Performance.iloc[0:3,1:5].T
p2 = pd.DataFrame(columns = p1.columns)
p2 = p2.T
p2["max"] = 0
p2["min"] = 0
p2 = p2.T
for i in range(len(p2.columns)):
    p2.iloc[0,i] = p1.iloc[:,i].max()
    p2.iloc[1,i] = p1.iloc[:,i].min()

# max-min normalizer = 0
# linear normalizer = 1
normalizer = 0
```

```
if normalizer == 0 :
    for i in range(len(p1.columns)):
        if i==2: #Non Beneficial
            p1.iloc[:,i] = (p2.iloc[0,i]-p1.iloc[:,i])/(p2.iloc[0,i] - p2.
iloc[1,i])

        else: #Beneficial
            p1.iloc[:,i] = (p1.iloc[:,i]-p2.iloc[1,i])/(p2.iloc[0,i] - p2.
iloc[1,i])

else:
    for i in range(len(p1.columns)):
        if i==2: #Non Beneficial
            p1.iloc[:,i] = p2.iloc[1,i]/p1.iloc[:,i]

        else: #Beneficial
            p1.iloc[:,i] = p1.iloc[:,i]/ p2.iloc[0,i]

#weight = np.array(pm["criteria_weight"]).reshape(1,3)
weight = np.array([0.5,0.3,0.2]).reshape(3,1)
p1["p-w"] = np.dot(p1,weight)
p1
```

Out [19]:

	train_Acc	test_Acc	time	p-w
ANN	0.991601	1.000000	0.864523	0.968705
SVM	0.000000	0.164475	0.000000	0.049343
KNN	0.415967	0.057992	1.000000	0.425381
GPC	1.000000	0.000000	0.153657	0.530731

In [20]:

```
p1.round(2)
```

Out [20]:

	train_Acc	test_Acc	time	p-w
ANN	0.99	1.00	0.86	0.97
SVM	0.00	0.16	0.00	0.05
KNN	0.42	0.06	1.00	0.43
GPC	1.00	0.00	0.15	0.53

2.SAWML_v4.0,100k

Importing Libraries

In [1]:

```
import pandas as pd
import numpy as np
import seaborn as sns

import tensorflow as tf
import keras
from keras.utils import to_categorical
from keras.models import Sequential
from keras.optimizers import SGD
from keras.layers import Dense

from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.neighbors import KNeighborsClassifier

from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import time
```

Importing Data

In [2]:

```
data = pd.read_csv('../input/thesis/dumy100000.csv')
```

MCDM

SAW

In [3]:

```
dn = 0
dn = data.copy()
del dn['Unnamed: 0']
d1 = dn.copy()

d2 = pd.DataFrame(columns = d1.columns)
d2 = d2.T
d2["max"] = 0
```

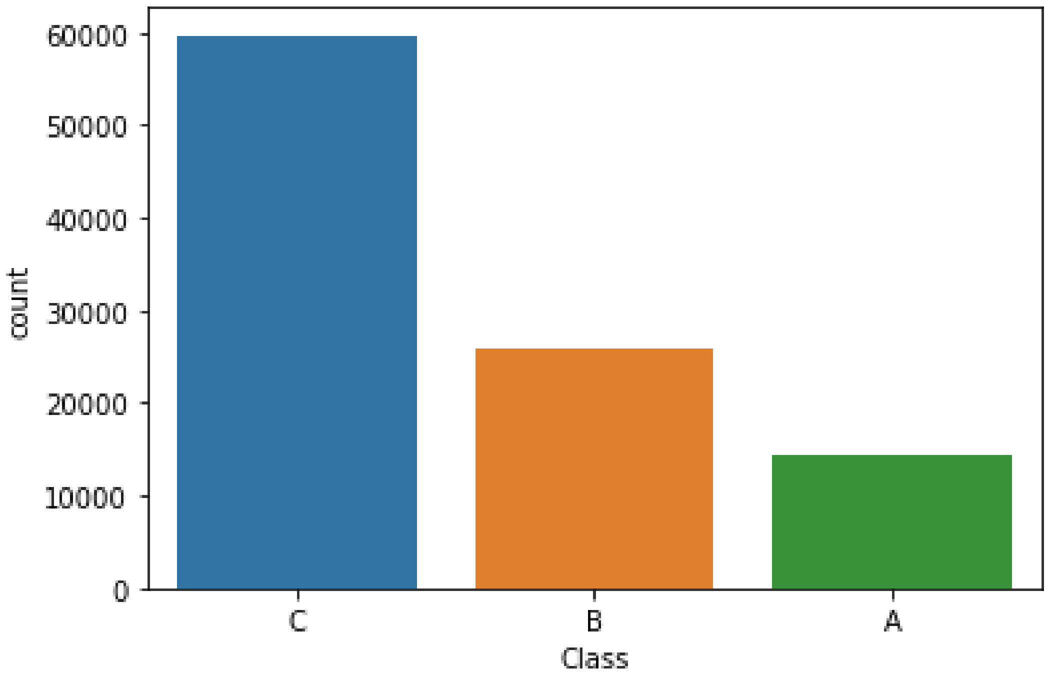
```
d2["min"] = 0
d2 = d2.T

for i in range(len(d2.columns)):
    d2.iloc[0,i] = d1.iloc[:,i].max()
    d2.iloc[1,i] = d1.iloc[:,i].min()

# max-min normalizer = 0
# linear normalizer = 1
normalizer = 0

if normalizer == 0 :
    for i in range(len(d1.columns)):
        if i==0 or i==3: #Non Beneficial
            d1.iloc[:,i] = (d2.iloc[0,i]-d1.iloc[:,i])/(d2.iloc[0,i] - d2.
iloc[1,i])

        else: #Beneficial
            d1.iloc[:,i] = (d1.iloc[:,i]-d2.iloc[1,i])/(d2.iloc[0,i] - d2.
iloc[1,i])

else:
    for i in range(len(d1.columns)):
        if i==0 or i==3: #Non Beneficial
            d1.iloc[:,i] = d2.iloc[1,i]/d1.iloc[:,i]

        else: #Beneficial
            d1.iloc[:,i] = d1.iloc[:,i]/ d2.iloc[0,i]

w  = [0.15, 0.20, 0.30, 0.15, 0.20]
d1 = d1*w

d1['performance'] = d1.iloc[:].sum(axis=1)

dn['performance'] = d1['performance']
dn = dn.sort_values(by=['performance'], ascending=True)

dn['RunCumCost']   = dn['performance'].cumsum()
TotSum             = dn['performance'].sum()
dn['RunCostPerc']  = (dn['RunCumCost']/TotSum)*100

def ABC_segmentation(perc):
    '''
   top A - top 20%, C - last 50% and B - between A & C
    '''
    if perc > 80 :
        return 'A'
    elif perc >= 50 and perc <= 80:
        return 'B'
    elif perc < 50:
        return 'C'

dn['Class'] = dn['RunCostPerc'].apply(ABC_segmentation)
```

```
ax = sns.countplot(x = dn['Class'],data = dn,label= 'Count')
dn['Class'].value_counts()

dn['productid'] = dn.index
dn = dn.sort_values(by=['productid'], ascending=True)
del dn['productid']

SAW_Class = dn['Class']

C,B,A = dn['Class'].value_counts()
print(round(100*A/dn['Class'].value_counts().sum(),2),"% ->","A:",A)
print(round(100*B/dn['Class'].value_counts().sum(),2),"% ->","B:",B)
print(round(100*C/dn['Class'].value_counts().sum(),2),"% ->","C:",C)
print("Total: ",dn['Class'].value_counts().sum())
dn.head()
```

```
14.43 % -> A: 14431
25.85 % -> B: 25850
59.72 % -> C: 59719
Total:  100000
```

Out [3]:

	mfg	sell	demand	fcost	rma	performance	RunCumCost	RunCostPerc	Class
0	71	156.91	1595	49.7	3	0.500506	20035.145963	40.033583	C
1	474	1047.54	6877	331.8	3	0.698912	45792.067588	91.500235	A
2	70	154.70	5182	49.0	3	0.696419	45583.448850	91.083379	A
3	417	921.57	3914	291.9	3	0.549959	27976.109958	55.900962	B
4	239	528.19	4496	167.3	2	0.521354	23532.483611	47.021851	C



Learning

In [4]:

```
iteration = 5
row_col = np.zeros([iteration,])

acc = pd.DataFrame()
acc['train_Acc'] = row_col
acc['test_Acc']  = row_col
acc['time']      = row_col
acc['test_size'] = row_col
start_time  = 0
end_time    = 0
train_score = 0
test_score  = 0

#split_size
n = .1
```

ANN

In [5]:

```
for i in range(iteration):

    X = dn.iloc[:,0:5].astype('float32')
    y = dn['Class'] #.iloc[0:100,-1]

    encoder = LabelEncoder()
    encoder.fit(y)
    y = encoder.transform(y)

    y = to_categorical(y)

    X_train,X_test,y_train,y_test = train_test_split(X, y ,test_size = n,
random_state=0)
    X_train,X_val,y_train,y_val = train_test_split(X_train, y_train ,test_
size = n, random_state=0)

    sc = StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test  = sc.fit_transform(X_test)
    X_val  = sc.fit_transform(X_val)

    #time start
    start_time = time.time()

    model = Sequential()
    model.add(Dense(5, activation='relu', input_shape=(X_train.shape[1],))
)
```

```
    model.add(Dense(3, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metri
cs=['accuracy'])
    model.fit(X_train,y_train,batch_size = 128, epochs = 20, validation_da
ta = (X_val,y_val))

    #time end
    end_time = time.time()

    train_score = model.evaluate(X_train,y_train,verbose=0)
    test_score = model.evaluate(X_test,y_test,verbose=0)

    #time calculation
    acc.iloc[i,2] = end_time - start_time

    #Accuracy
    acc.iloc[i,0] = train_score[1]*100
    acc.iloc[i,1] = train_score[1]*100

    #test size
    acc.iloc[i,3] = n*100

    if n == .1:
        n = .2
    else :
        n = .1
```

```
Epoch 1/20
633/633 [==============================] - 2s 2ms/step - loss: 0.7534 - ac
curacy: 0.7458 - val_loss: 0.2294 - val_accuracy: 0.9703
Epoch 2/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1681 - ac
curacy: 0.9828 - val_loss: 0.0893 - val_accuracy: 0.9870
Epoch 3/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0762 - ac
curacy: 0.9922 - val_loss: 0.0601 - val_accuracy: 0.9887
Epoch 4/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0525 - ac
curacy: 0.9948 - val_loss: 0.0462 - val_accuracy: 0.9922
Epoch 5/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0408 - ac
curacy: 0.9960 - val_loss: 0.0393 - val_accuracy: 0.9893
Epoch 6/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0343 - ac
curacy: 0.9958 - val_loss: 0.0340 - val_accuracy: 0.9914
Epoch 7/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0287 - ac
curacy: 0.9969 - val_loss: 0.0291 - val_accuracy: 0.9940
Epoch 8/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0255 - ac
curacy: 0.9972 - val_loss: 0.0268 - val_accuracy: 0.9932
Epoch 9/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0229 - ac
```

```
curacy: 0.9974 - val_loss: 0.0262 - val_accuracy: 0.9919
Epoch 10/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0210 - ac
curacy: 0.9973 - val_loss: 0.0235 - val_accuracy: 0.9932
Epoch 11/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0193 - ac
curacy: 0.9976 - val_loss: 0.0210 - val_accuracy: 0.9943
Epoch 12/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0176 - ac
curacy: 0.9978 - val_loss: 0.0215 - val_accuracy: 0.9931
Epoch 13/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0158 - ac
curacy: 0.9979 - val_loss: 0.0208 - val_accuracy: 0.9921
Epoch 14/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0158 - ac
curacy: 0.9973 - val_loss: 0.0181 - val_accuracy: 0.9942
Epoch 15/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0149 - ac
curacy: 0.9979 - val_loss: 0.0188 - val_accuracy: 0.9923
Epoch 16/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0139 - ac
curacy: 0.9975 - val_loss: 0.0188 - val_accuracy: 0.9923
Epoch 17/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0133 - ac
curacy: 0.9980 - val_loss: 0.0180 - val_accuracy: 0.9930
Epoch 18/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0125 - ac
curacy: 0.9979 - val_loss: 0.0158 - val_accuracy: 0.9942
Epoch 19/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0121 - ac
curacy: 0.9981 - val_loss: 0.0185 - val_accuracy: 0.9931
Epoch 20/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0120 - ac
curacy: 0.9977 - val_loss: 0.0158 - val_accuracy: 0.9938
Epoch 1/20
500/500 [==============================] - 1s 2ms/step - loss: 0.9427 - ac
curacy: 0.6090 - val_loss: 0.4895 - val_accuracy: 0.8090
Epoch 2/20
500/500 [==============================] - 1s 1ms/step - loss: 0.4188 - ac
curacy: 0.8294 - val_loss: 0.2606 - val_accuracy: 0.8430
Epoch 3/20
500/500 [==============================] - 1s 1ms/step - loss: 0.2257 - ac
curacy: 0.9346 - val_loss: 0.1551 - val_accuracy: 0.9776
Epoch 4/20
500/500 [==============================] - 1s 1ms/step - loss: 0.1387 - ac
curacy: 0.9836 - val_loss: 0.1045 - val_accuracy: 0.9858
Epoch 5/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0955 - ac
curacy: 0.9917 - val_loss: 0.0763 - val_accuracy: 0.9912
Epoch 6/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0720 - ac
curacy: 0.9948 - val_loss: 0.0594 - val_accuracy: 0.9932
Epoch 7/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0559 - ac
```

```
curacy: 0.9958 - val_loss: 0.0484 - val_accuracy: 0.9933
Epoch 8/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0452 - ac
curacy: 0.9960 - val_loss: 0.0406 - val_accuracy: 0.9954
Epoch 9/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0392 - ac
curacy: 0.9957 - val_loss: 0.0351 - val_accuracy: 0.9944
Epoch 10/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0341 - ac
curacy: 0.9966 - val_loss: 0.0311 - val_accuracy: 0.9946
Epoch 11/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0295 - ac
curacy: 0.9970 - val_loss: 0.0275 - val_accuracy: 0.9962
Epoch 12/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0265 - ac
curacy: 0.9968 - val_loss: 0.0251 - val_accuracy: 0.9954
Epoch 13/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0239 - ac
curacy: 0.9975 - val_loss: 0.0230 - val_accuracy: 0.9958
Epoch 14/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0224 - ac
curacy: 0.9971 - val_loss: 0.0226 - val_accuracy: 0.9929
Epoch 15/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0203 - ac
curacy: 0.9976 - val_loss: 0.0225 - val_accuracy: 0.9930
Epoch 16/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0188 - ac
curacy: 0.9970 - val_loss: 0.0209 - val_accuracy: 0.9938
Epoch 17/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0180 - ac
curacy: 0.9972 - val_loss: 0.0189 - val_accuracy: 0.9946
Epoch 18/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0171 - ac
curacy: 0.9974 - val_loss: 0.0169 - val_accuracy: 0.9960
Epoch 19/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0162 - ac
curacy: 0.9974 - val_loss: 0.0179 - val_accuracy: 0.9940
Epoch 20/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0156 - ac
curacy: 0.9972 - val_loss: 0.0174 - val_accuracy: 0.9932
Epoch 1/20
633/633 [==============================] - 1s 1ms/step - loss: 0.7680 - ac
curacy: 0.6386 - val_loss: 0.4006 - val_accuracy: 0.8244
Epoch 2/20
633/633 [==============================] - 1s 1ms/step - loss: 0.3377 - ac
curacy: 0.8852 - val_loss: 0.1903 - val_accuracy: 0.9661
Epoch 3/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1582 - ac
curacy: 0.9798 - val_loss: 0.1072 - val_accuracy: 0.9837
Epoch 4/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0929 - ac
curacy: 0.9929 - val_loss: 0.0720 - val_accuracy: 0.9931
Epoch 5/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0645 - ac
```

```
curacy: 0.9956 - val_loss: 0.0544 - val_accuracy: 0.9933
Epoch 6/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0488 - ac
curacy: 0.9966 - val_loss: 0.0434 - val_accuracy: 0.9930
Epoch 7/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0385 - ac
curacy: 0.9969 - val_loss: 0.0378 - val_accuracy: 0.9919
Epoch 8/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0320 - ac
curacy: 0.9968 - val_loss: 0.0340 - val_accuracy: 0.9918
Epoch 9/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0276 - ac
curacy: 0.9971 - val_loss: 0.0267 - val_accuracy: 0.9949
Epoch 10/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0235 - ac
curacy: 0.9973 - val_loss: 0.0258 - val_accuracy: 0.9940
Epoch 11/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0213 - ac
curacy: 0.9978 - val_loss: 0.0228 - val_accuracy: 0.9938
Epoch 12/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0194 - ac
curacy: 0.9975 - val_loss: 0.0226 - val_accuracy: 0.9924
Epoch 13/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0176 - ac
curacy: 0.9976 - val_loss: 0.0194 - val_accuracy: 0.9943
Epoch 14/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0160 - ac
curacy: 0.9978 - val_loss: 0.0200 - val_accuracy: 0.9921
Epoch 15/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0157 - ac
curacy: 0.9979 - val_loss: 0.0190 - val_accuracy: 0.9940
Epoch 16/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0140 - ac
curacy: 0.9980 - val_loss: 0.0211 - val_accuracy: 0.9914
Epoch 17/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0137 - ac
curacy: 0.9974 - val_loss: 0.0182 - val_accuracy: 0.9930
Epoch 18/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0128 - ac
curacy: 0.9980 - val_loss: 0.0177 - val_accuracy: 0.9932
Epoch 19/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0121 - ac
curacy: 0.9980 - val_loss: 0.0174 - val_accuracy: 0.9931
Epoch 20/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0119 - ac
curacy: 0.9978 - val_loss: 0.0141 - val_accuracy: 0.9943
Epoch 1/20
500/500 [==============================] - 1s 2ms/step - loss: 0.9685 - ac
curacy: 0.5783 - val_loss: 0.5623 - val_accuracy: 0.7966
Epoch 2/20
500/500 [==============================] - 1s 1ms/step - loss: 0.5131 - ac
curacy: 0.8115 - val_loss: 0.4289 - val_accuracy: 0.8266
Epoch 3/20
500/500 [==============================] - 1s 1ms/step - loss: 0.4060 - ac
```

```
curacy: 0.8380 - val_loss: 0.3751 - val_accuracy: 0.8359
Epoch 4/20
500/500 [==============================] - 1s 1ms/step - loss: 0.3617 - ac
curacy: 0.8440 - val_loss: 0.3439 - val_accuracy: 0.8392
Epoch 5/20
500/500 [==============================] - 1s 1ms/step - loss: 0.3318 - ac
curacy: 0.8499 - val_loss: 0.3241 - val_accuracy: 0.8409
Epoch 6/20
500/500 [==============================] - 1s 1ms/step - loss: 0.3143 - ac
curacy: 0.8510 - val_loss: 0.3106 - val_accuracy: 0.8408
Epoch 7/20
500/500 [==============================] - 1s 1ms/step - loss: 0.3020 - ac
curacy: 0.8505 - val_loss: 0.3000 - val_accuracy: 0.8433
Epoch 8/20
500/500 [==============================] - 1s 1ms/step - loss: 0.2945 - ac
curacy: 0.8500 - val_loss: 0.2920 - val_accuracy: 0.8435
Epoch 9/20
500/500 [==============================] - 1s 1ms/step - loss: 0.2836 - ac
curacy: 0.8534 - val_loss: 0.2859 - val_accuracy: 0.8410
Epoch 10/20
500/500 [==============================] - 1s 1ms/step - loss: 0.2808 - ac
curacy: 0.8486 - val_loss: 0.2800 - val_accuracy: 0.8432
Epoch 11/20
500/500 [==============================] - 1s 1ms/step - loss: 0.2743 - ac
curacy: 0.8490 - val_loss: 0.2755 - val_accuracy: 0.8410
Epoch 12/20
500/500 [==============================] - 1s 1ms/step - loss: 0.2695 - ac
curacy: 0.8497 - val_loss: 0.2715 - val_accuracy: 0.8382
Epoch 13/20
500/500 [==============================] - 1s 1ms/step - loss: 0.2652 - ac
curacy: 0.8489 - val_loss: 0.2639 - val_accuracy: 0.8421
Epoch 14/20
500/500 [==============================] - 1s 1ms/step - loss: 0.2561 - ac
curacy: 0.8474 - val_loss: 0.2540 - val_accuracy: 0.8372
Epoch 15/20
500/500 [==============================] - 1s 1ms/step - loss: 0.2458 - ac
curacy: 0.8422 - val_loss: 0.2335 - val_accuracy: 0.8833
Epoch 16/20
500/500 [==============================] - 1s 1ms/step - loss: 0.2165 - ac
curacy: 0.9107 - val_loss: 0.1559 - val_accuracy: 0.9770
Epoch 17/20
500/500 [==============================] - 1s 1ms/step - loss: 0.1491 - ac
curacy: 0.9742 - val_loss: 0.1289 - val_accuracy: 0.9760
Epoch 18/20
500/500 [==============================] - 1s 1ms/step - loss: 0.1261 - ac
curacy: 0.9772 - val_loss: 0.1138 - val_accuracy: 0.9778
Epoch 19/20
500/500 [==============================] - 1s 1ms/step - loss: 0.1145 - ac
curacy: 0.9796 - val_loss: 0.1053 - val_accuracy: 0.9797
Epoch 20/20
500/500 [==============================] - 1s 1ms/step - loss: 0.1052 - ac
curacy: 0.9831 - val_loss: 0.0998 - val_accuracy: 0.9756
Epoch 1/20
633/633 [==============================] - 1s 1ms/step - loss: 0.8508 - ac
```

```
curacy: 0.5980 - val_loss: 0.2921 - val_accuracy: 0.9246
Epoch 2/20
633/633 [==============================] - 1s 1ms/step - loss: 0.2249 - ac
curacy: 0.9502 - val_loss: 0.1180 - val_accuracy: 0.9813
Epoch 3/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0981 - ac
curacy: 0.9884 - val_loss: 0.0714 - val_accuracy: 0.9903
Epoch 4/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0628 - ac
curacy: 0.9925 - val_loss: 0.0526 - val_accuracy: 0.9933
Epoch 5/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0472 - ac
curacy: 0.9950 - val_loss: 0.0439 - val_accuracy: 0.9899
Epoch 6/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0386 - ac
curacy: 0.9962 - val_loss: 0.0370 - val_accuracy: 0.9920
Epoch 7/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0330 - ac
curacy: 0.9960 - val_loss: 0.0319 - val_accuracy: 0.9939
Epoch 8/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0277 - ac
curacy: 0.9972 - val_loss: 0.0300 - val_accuracy: 0.9918
Epoch 9/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0242 - ac
curacy: 0.9976 - val_loss: 0.0266 - val_accuracy: 0.9928
Epoch 10/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0222 - ac
curacy: 0.9970 - val_loss: 0.0241 - val_accuracy: 0.9937
Epoch 11/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0204 - ac
curacy: 0.9978 - val_loss: 0.0242 - val_accuracy: 0.9930
Epoch 12/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0193 - ac
curacy: 0.9972 - val_loss: 0.0229 - val_accuracy: 0.9934
Epoch 13/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0184 - ac
curacy: 0.9975 - val_loss: 0.0237 - val_accuracy: 0.9917
Epoch 14/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0165 - ac
curacy: 0.9977 - val_loss: 0.0194 - val_accuracy: 0.9939
Epoch 15/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0157 - ac
curacy: 0.9979 - val_loss: 0.0197 - val_accuracy: 0.9936
Epoch 16/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0151 - ac
curacy: 0.9974 - val_loss: 0.0193 - val_accuracy: 0.9927
Epoch 17/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0144 - ac
curacy: 0.9977 - val_loss: 0.0188 - val_accuracy: 0.9932
Epoch 18/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0135 - ac
curacy: 0.9975 - val_loss: 0.0188 - val_accuracy: 0.9930
Epoch 19/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0130 - ac
```

```
curacy: 0.9977 - val_loss: 0.0160 - val_accuracy: 0.9943
Epoch 20/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0127 - ac
curacy: 0.9979 - val_loss: 0.0180 - val_accuracy: 0.9933
```

In [6]:

```
acc = acc.round(2)
acc
```

Out [6]:

	train_Acc	test_Acc	time	test_size
0	99.81	99.81	16.34	10.0
1	99.84	99.84	13.22	20.0
2	99.80	99.80	15.70	10.0
3	97.95	97.95	13.45	20.0
4	99.81	99.81	15.58	10.0

In [7]:

```
ANNAcc = acc.mean()
ANNAcc
```

Out [7]:

```
train_Acc    99.442
test_Acc     99.442
time         14.858
test_size    14.000
dtype: float64
```

SVM

In [8]:

```
n = .1
for i in range(iteration):

    X = dn.iloc[:,0:5].astype('float32')
    y = dn['Class']

    encoder = LabelEncoder()
    encoder.fit(y)
    y = encoder.transform(y)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
n, random_state=0)

    #time start
    start_time = time.time()


    model = SVC(kernel='poly')
    model.fit(X_train,y_train)

    #time end
    end_time = time.time()

    train_score = model.score(X_train,y_train)
    test_score  = model.score(X_test,y_test)

    #time calculation
    acc.iloc[i,2] = end_time - start_time

    #Accuracy
    acc.iloc[i,0] = train_score*100
    acc.iloc[i,1] = test_score*100

    #test size
    acc.iloc[i,3] = n*100

    if n == .1:
        n = .2
    else :
        n = .1
    print(i+1," run")
```

```
1  run
2  run
3  run
4  run
5  run
```

In [9]:

```
acc = acc.round(2)
acc
```

Out [9]:

	train_Acc	test_Acc	time	test_size
0	64.60	65.28	528.50	10.0
1	64.59	64.84	297.06	20.0
2	64.60	65.28	281.57	10.0
3	64.59	64.84	354.49	20.0
4	64.60	65.28	376.64	10.0

In [10]:

```
SVMAcc = acc.mean()
SVMAcc
```

Out [10]:

```
train_Acc     64.596
test_Acc      65.104
time         367.652
test_size     14.000
dtype: float64
```

KNN

In [11]:

```
n = .1
for i in range(iteration):

    X = dn.iloc[:,0:5].astype('float32')
    y = dn['Class']

    encoder = LabelEncoder()
    encoder.fit(y)
    y = encoder.transform(y)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
n, random_state=0)


    #time start
    start_time = time.time()

    model = KNeighborsClassifier(n_neighbors=3)
    model.fit(X_train,y_train)

    #time end
    end_time = time.time()

    train_score = model.score(X_train,y_train)*100
    test_score  = model.score(X_test,y_test)*100

    #time calculation
    acc.iloc[i,2] = end_time - start_time

    #Accuracy
    acc.iloc[i,0] = train_score
    acc.iloc[i,1] = test_score

    #test size
    acc.iloc[i,3] = n*100

    if n == .1:
        n = .2
    else :
        n = .1
    print(i+1," run")
```

```
1   run
2   run
3   run
4   run
5   run
```

In [12]:

```
acc = acc.round(2)
acc
```

Out [12]:

	train_Acc	test_Acc	time	test_size
0	80.44	63.32	0.13	10.0
1	80.33	63.04	0.11	20.0
2	80.44	63.32	0.13	10.0
3	80.33	63.04	0.11	20.0
4	80.44	63.32	0.13	10.0

In [13]:

```
KNNAcc = acc.mean()
KNNAcc
```

Out [13]:

```
train_Acc    80.396
test_Acc     63.208
time          0.122
test_size    14.000
dtype: float64
```

GPC

In [14]:

```
n = .1
for i in range(iteration):

    X = dn.iloc[:,0:5].astype('float32')
    y = dn['Class'] #.iloc[0:100,-1]

    encoder = LabelEncoder()
    encoder.fit(y)
    y = encoder.transform(y)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
n, random_state=0)
```

```
    #time start
    start_time = time.time()

    model = GaussianProcessClassifier(multi_class='one_vs_one')
    model.fit(X_train,y_train)

    #time end
    end_time = time.time()

    train_score = model.score(X_train,y_train)*100
    test_score  = model.score(X_test,y_test)*100

    #time calculation
    acc.iloc[i,2] = end_time - start_time

    #Accuracy
    acc.iloc[i,0] = train_score
    acc.iloc[i,1] = test_score

    #test size
    acc.iloc[i,3] = n*100

    if n == .1:
        n = .2
    else :
        n = .1
    print(i+1," run")
```

```
1  run
2  run
3  run
4  run
5  run
```

In [15]:

```
acc = acc.round(2)
acc
```

Out [15]:

	train_Acc	test_Acc	time	test_size
0	100.0	61.8	86.91	10.0
1	100.0	63.1	65.36	20.0
2	100.0	61.8	86.40	10.0
3	100.0	63.1	65.02	20.0
4	100.0	61.8	85.79	10.0

In [16]:

```
GPCAcc = acc.mean()
GPCAcc
```

Out[16]:

```
train_Acc    100.000
test_Acc      62.320
time          77.896
test_size     14.000
dtype: float64
```

In [17]:

```
performance = {
    "Performance" : ['Avg Train Accuracy','Avg Test Accuracy','Avg Run Tim
e','Avg Test size'],
    "ANN"         : ANNAcc,
    "SVM"         : SVMAcc,
    "KNN"         : KNNAcc,
    "GPC"         : GPCAcc
}
Performance = pd.DataFrame(performance)
Performance.round(2)
```

Out[17]:

	Performance	ANN	SVM	KNN	GPC
train_Acc	Avg Train Accuracy	99.44	64.60	80.40	100.00
test_Acc	Avg Test Accuracy	99.44	65.10	63.21	62.32
time	Avg Run Time	14.86	367.65	0.12	77.90
test_size	Avg Test size	14.00	14.00	14.00	14.00

In [18]:

```
p1 = Performance.iloc[0:3,1:5].T
p2 = pd.DataFrame(columns = p1.columns)
p2 = p2.T
p2["max"] = 0
p2["min"] = 0
p2 = p2.T
for i in range(len(p2.columns)):
    p2.iloc[0,i] = p1.iloc[:,i].max()
    p2.iloc[1,i] = p1.iloc[:,i].min()

# max-min normalizer = 0
# linear normalizer = 1
normalizer = 0
```

```
if normalizer == 0 :
    for i in range(len(p1.columns)):
        if i==2: #Non Beneficial
            p1.iloc[:,i] = (p2.iloc[0,i]-p1.iloc[:,i])/(p2.iloc[0,i] - p2.
iloc[1,i])

        else: #Beneficial
            p1.iloc[:,i] = (p1.iloc[:,i]-p2.iloc[1,i])/(p2.iloc[0,i] - p2.
iloc[1,i])

else:
    for i in range(len(p1.columns)):
        if i==2: #Non Beneficial
            p1.iloc[:,i] = p2.iloc[1,i]/p1.iloc[:,i]

        else: #Beneficial
            p1.iloc[:,i] = p1.iloc[:,i]/ p2.iloc[0,i]

weight = np.array([0.5,0.3,0.2]).reshape(3,1)
p1["p-w"] = np.dot(p1,weight)
p1
```

Out[18]:

	train_Acc	test_Acc	time	p-w
ANN	0.984239	1.000000	0.959905	0.984101
SVM	0.000000	0.074996	0.000000	0.022499
KNN	0.446277	0.023921	1.000000	0.430315
GPC	1.000000	0.000000	0.788387	0.657677

In [19]:

```
p1.round(2)
```

Out[19]:

	train_Acc	test_Acc	time	p-w
ANN	0.98	1.00	0.96	0.98
SVM	0.00	0.07	0.00	0.02
KNN	0.45	0.02	1.00	0.43

GPC	1.00	0.00	0.79	0.66

3.VIKORML_v4.0,100k

Importing Libraries

In [1]:

```
import pandas as pd
import numpy as np
import seaborn as sns

import tensorflow as tf
import keras
from keras.utils import to_categorical
from keras.models import Sequential
from keras.optimizers import SGD
from keras.layers import Dense

from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.neighbors import KNeighborsClassifier

from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import time
```

Importing Data

In [2]:

```
data = pd.read_csv('../input/thesis/dumy100000.csv')
```

MCDM

VIKOR

In [3]:

```
dn = 0
dn = data.copy()
del dn['Unnamed: 0']
```

```
d1 = dn.copy()

d2 = pd.DataFrame(columns = d1.columns)
d2 = d2.T
d2["max"] = 0
d2["min"] = 0
d2 = d2.T

for i in range(len(d2.columns)):
    d2.iloc[0,i] = d1.iloc[:,i].max()
    d2.iloc[1,i] = d1.iloc[:,i].min()

# max-min normalizer = 0
# linear normalizer = 1
normalizer = 0

if normalizer == 0 :
    for i in range(len(d1.columns)):
        if i==0 or i==3: #Non Beneficial
            d1.iloc[:,i] = (d2.iloc[0,i]-d1.iloc[:,i])/(d2.iloc[0,i] - d2.
iloc[1,i])

        else: #Beneficial
            d1.iloc[:,i] = (d1.iloc[:,i]-d2.iloc[1,i])/(d2.iloc[0,i] - d2.
iloc[1,i])

else:
    for i in range(len(d1.columns)):
        if i==0 or i==3: #Non Beneficial
            d1.iloc[:,i] = d2.iloc[1,i]/d1.iloc[:,i]

        else: #Beneficial
            d1.iloc[:,i] = d1.iloc[:,i]/ d2.iloc[0,i]

w  = [0.15, 0.20, 0.30, 0.15, 0.20]
d1 = d1*w

d1['si'] = d1.iloc[:].sum(axis=1)

d1['ri'] = d1.iloc[:,0:-2].max(axis=1)

s_best  = d1['si'].min()
s_worst = d1['si'].max()
r_best  = d1['ri'].min()
r_worst = d1['ri'].max()
neu = 0.5

d1['qi'] = d1.apply(lambda row: neu*((row.si-s_best)/(s_worst-s_best))+(1-
neu)*((row.ri-r_best)/(r_worst-r_best)) , axis =1 )

dn['qi'] = d1['qi']

dn = dn.sort_values(by=['qi'], ascending=False)
```

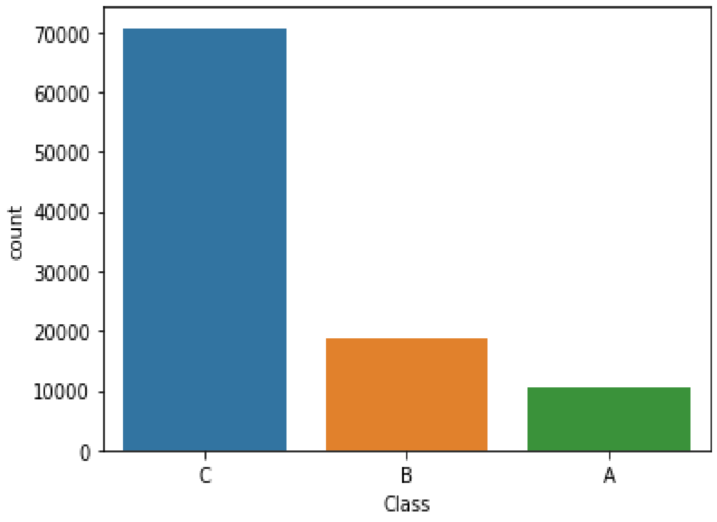
```
dn['RunCumCost']   = dn['qi'].cumsum()
TotSum             = dn['qi'].sum()
dn['RunCostPerc']  = (dn['RunCumCost']/TotSum)*100

dn['Rank']         = dn['RunCostPerc'].rank()
dn['RunItemCum']   = dn['Rank'].cumsum()
TotItemSum         = dn['Rank'].sum()
dn['RunItemPerc']  = (dn['RunItemCum']/TotItemSum)*100

def ABC_segmentation(perc):
    '''
    top A - top 20%, C - last 50% and B - between A & C

    '''
    if perc > 80 :
        return 'A'
    elif perc >= 50 and perc <= 80:
        return 'B'
    elif perc < 50:
        return 'C'

dn['Class'] = dn['RunItemPerc'].apply(ABC_segmentation)

ax = sns.countplot(x = dn['Class'],data = dn,label= 'Count')
dn['Class'].value_counts()

dn['productid'] = dn.index
dn = dn.sort_values(by=['productid'], ascending=True)
del dn['productid']
VIKOR_Class = dn['Class']

C,B,A = dn['Class'].value_counts()
print(round(100*A/dn['Class'].value_counts().sum(),2),"% ->","A:",A)
print(round(100*B/dn['Class'].value_counts().sum(),2),"% ->","B:",B)
print(round(100*C/dn['Class'].value_counts().sum(),2),"% ->","C:",C)
print("Total: ",dn['Class'].value_counts().sum())
dn.head()
```

```
10.56 % -> A: 10558
18.73 % -> B: 18732
70.71 % -> C: 70710
Total:  100000
```

Out [3]:

	mfg	sell	demand	fcost	rma	qi	RunCumCost	RunCostPerc	Rank	RunItemCum	RunItemPerc	Class
0	71	156.91	1595	49.7	3	0.383965	36257.514659	76.779078	59291.0	1.757741e+09	35.154468	C
1	474	1047.54	6877	331.8	3	0.901189	1962.408386	4.155605	2101.0	2.208151e+06	0.044163	C
2	70	154.70	5182	49.0	3	0.683190	16176.765170	34.255992	20391.0	2.079066e+08	4.158091	C

3	417	921.57	3914	291.9	3	0.473220	29837.401955	63.183818	44227.0	9.780359e+08	19.560522	C
4	239	528.19	4496	167.3	2	0.449204	31487.105629	66.677238	47806.0	1.142731e+09	22.854386	C



Machine Learning

In [4]:

```
iteration = 5
row_col = np.zeros([iteration,])

acc = pd.DataFrame()
acc['train_Acc'] = row_col
acc['test_Acc']  = row_col
acc['time']      = row_col
acc['test_size'] = row_col
start_time  = 0
end_time    = 0
train_score = 0
test_score  = 0

#split_size
n = .1
```

ANN

In [5]:

```
for i in range(iteration):

    X = dn.iloc[:,0:5].astype('float32')
    y = dn['Class'] #.iloc[0:100,-1]

    encoder = LabelEncoder()
    encoder.fit(y)
    y = encoder.transform(y)

    y = to_categorical(y)

    X_train,X_test,y_train,y_test = train_test_split(X, y ,test_size = n,
random_state=0)
    X_train,X_val,y_train,y_val = train_test_split(X_train, y_train ,test_
size = n, random_state=0)

    sc = StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test  = sc.fit_transform(X_test)
    X_val  = sc.fit_transform(X_val)

    #time start
    start_time = time.time()

    model = Sequential()
    model.add(Dense(5, activation='relu', input_shape=(X_train.shape[1],))
)
    model.add(Dense(3, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metri
cs=['accuracy'])
    model.fit(X_train,y_train,batch_size = 128, epochs = 20, validation_da
ta = (X_val,y_val))

    #time end
    end_time = time.time()

    train_score = model.evaluate(X_train,y_train,verbose=0)
    test_score = model.evaluate(X_test,y_test,verbose=0)

    #time calculation
    acc.iloc[i,2] = end_time - start_time

    #Accuracy
    acc.iloc[i,0] = train_score[1]*100
    acc.iloc[i,1] = train_score[1]*100

    #test size
    acc.iloc[i,3] = n*100

    if n == .1:
        n = .2
    else :
        n = .1

Epoch 1/20
633/633 [==============================] - 2s 2ms/step - loss: 1.0209 - ac
curacy: 0.6256 - val_loss: 0.7108 - val_accuracy: 0.8104
Epoch 2/20
633/633 [==============================] - 1s 1ms/step - loss: 0.6138 - ac
```

```
curacy: 0.8121 - val_loss: 0.4350 - val_accuracy: 0.8699
Epoch 3/20
633/633 [==============================] - 1s 1ms/step - loss: 0.4056 - ac
curacy: 0.8741 - val_loss: 0.3378 - val_accuracy: 0.9120
Epoch 4/20
633/633 [==============================] - 1s 1ms/step - loss: 0.3225 - ac
curacy: 0.9158 - val_loss: 0.2827 - val_accuracy: 0.9413
Epoch 5/20
633/633 [==============================] - 1s 1ms/step - loss: 0.2719 - ac
curacy: 0.9421 - val_loss: 0.2448 - val_accuracy: 0.9486
Epoch 6/20
633/633 [==============================] - 1s 1ms/step - loss: 0.2371 - ac
curacy: 0.9522 - val_loss: 0.2171 - val_accuracy: 0.9506
Epoch 7/20
633/633 [==============================] - 1s 1ms/step - loss: 0.2119 - ac
curacy: 0.9535 - val_loss: 0.1937 - val_accuracy: 0.9550
Epoch 8/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1900 - ac
curacy: 0.9585 - val_loss: 0.1756 - val_accuracy: 0.9593
Epoch 9/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1723 - ac
curacy: 0.9609 - val_loss: 0.1611 - val_accuracy: 0.9614
Epoch 10/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1580 - ac
curacy: 0.9638 - val_loss: 0.1467 - val_accuracy: 0.9646
Epoch 11/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1448 - ac
curacy: 0.9652 - val_loss: 0.1356 - val_accuracy: 0.9669
Epoch 12/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1334 - ac
curacy: 0.9689 - val_loss: 0.1255 - val_accuracy: 0.9678
Epoch 13/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1232 - ac
curacy: 0.9709 - val_loss: 0.1171 - val_accuracy: 0.9704
Epoch 14/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1146 - ac
curacy: 0.9724 - val_loss: 0.1097 - val_accuracy: 0.9728
Epoch 15/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1061 - ac
curacy: 0.9747 - val_loss: 0.1027 - val_accuracy: 0.9758
Epoch 16/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0990 - ac
curacy: 0.9782 - val_loss: 0.0969 - val_accuracy: 0.9780
Epoch 17/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0931 - ac
curacy: 0.9796 - val_loss: 0.0905 - val_accuracy: 0.9812
Epoch 18/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0876 - ac
curacy: 0.9833 - val_loss: 0.0865 - val_accuracy: 0.9807
Epoch 19/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0828 - ac
curacy: 0.9846 - val_loss: 0.0803 - val_accuracy: 0.9854
Epoch 20/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0792 - ac
```

```
curacy: 0.9862 - val_loss: 0.0770 - val_accuracy: 0.9852
Epoch 1/20
500/500 [==============================] - 1s 2ms/step - loss: 0.8669 - ac
curacy: 0.7046 - val_loss: 0.4532 - val_accuracy: 0.7095
Epoch 2/20
500/500 [==============================] - 1s 1ms/step - loss: 0.4191 - ac
curacy: 0.7153 - val_loss: 0.3306 - val_accuracy: 0.8292
Epoch 3/20
500/500 [==============================] - 1s 1ms/step - loss: 0.3122 - ac
curacy: 0.8450 - val_loss: 0.2526 - val_accuracy: 0.9375
Epoch 4/20
500/500 [==============================] - 1s 1ms/step - loss: 0.2432 - ac
curacy: 0.9365 - val_loss: 0.2012 - val_accuracy: 0.9465
Epoch 5/20
500/500 [==============================] - 1s 1ms/step - loss: 0.1964 - ac
curacy: 0.9455 - val_loss: 0.1635 - val_accuracy: 0.9549
Epoch 6/20
500/500 [==============================] - 1s 1ms/step - loss: 0.1596 - ac
curacy: 0.9546 - val_loss: 0.1372 - val_accuracy: 0.9614
Epoch 7/20
500/500 [==============================] - 1s 1ms/step - loss: 0.1345 - ac
curacy: 0.9600 - val_loss: 0.1175 - val_accuracy: 0.9652
Epoch 8/20
500/500 [==============================] - 1s 1ms/step - loss: 0.1162 - ac
curacy: 0.9639 - val_loss: 0.1038 - val_accuracy: 0.9698
Epoch 9/20
500/500 [==============================] - 1s 1ms/step - loss: 0.1061 - ac
curacy: 0.9668 - val_loss: 0.0932 - val_accuracy: 0.9716
Epoch 10/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0936 - ac
curacy: 0.9691 - val_loss: 0.0847 - val_accuracy: 0.9722
Epoch 11/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0861 - ac
curacy: 0.9712 - val_loss: 0.0791 - val_accuracy: 0.9749
Epoch 12/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0797 - ac
curacy: 0.9739 - val_loss: 0.0735 - val_accuracy: 0.9762
Epoch 13/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0749 - ac
curacy: 0.9744 - val_loss: 0.0687 - val_accuracy: 0.9778
Epoch 14/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0701 - ac
curacy: 0.9762 - val_loss: 0.0667 - val_accuracy: 0.9769
Epoch 15/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0682 - ac
curacy: 0.9752 - val_loss: 0.0636 - val_accuracy: 0.9768
Epoch 16/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0660 - ac
curacy: 0.9775 - val_loss: 0.0606 - val_accuracy: 0.9786
Epoch 17/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0611 - ac
curacy: 0.9779 - val_loss: 0.0589 - val_accuracy: 0.9786
Epoch 18/20
500/500 [==============================] - 1s 2ms/step - loss: 0.0603 - ac
```

```
curacy: 0.9781 - val_loss: 0.0570 - val_accuracy: 0.9786
Epoch 19/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0583 - ac
curacy: 0.9788 - val_loss: 0.0550 - val_accuracy: 0.9798
Epoch 20/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0576 - ac
curacy: 0.9782 - val_loss: 0.0542 - val_accuracy: 0.9796
Epoch 1/20
633/633 [==============================] - 1s 1ms/step - loss: 0.7402 - ac
curacy: 0.7131 - val_loss: 0.4188 - val_accuracy: 0.8168
Epoch 2/20
633/633 [==============================] - 1s 1ms/step - loss: 0.3886 - ac
curacy: 0.8318 - val_loss: 0.3316 - val_accuracy: 0.8464
Epoch 3/20
633/633 [==============================] - 1s 1ms/step - loss: 0.3143 - ac
curacy: 0.8536 - val_loss: 0.2703 - val_accuracy: 0.8578
Epoch 4/20
633/633 [==============================] - 1s 1ms/step - loss: 0.2516 - ac
curacy: 0.8627 - val_loss: 0.1985 - val_accuracy: 0.9459
Epoch 5/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1814 - ac
curacy: 0.9529 - val_loss: 0.1469 - val_accuracy: 0.9579
Epoch 6/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1400 - ac
curacy: 0.9586 - val_loss: 0.1231 - val_accuracy: 0.9597
Epoch 7/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1201 - ac
curacy: 0.9613 - val_loss: 0.1098 - val_accuracy: 0.9628
Epoch 8/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1083 - ac
curacy: 0.9627 - val_loss: 0.1003 - val_accuracy: 0.9640
Epoch 9/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0990 - ac
curacy: 0.9654 - val_loss: 0.0940 - val_accuracy: 0.9660
Epoch 10/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0953 - ac
curacy: 0.9666 - val_loss: 0.0903 - val_accuracy: 0.9689
Epoch 11/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0897 - ac
curacy: 0.9677 - val_loss: 0.0870 - val_accuracy: 0.9696
Epoch 12/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0869 - ac
curacy: 0.9685 - val_loss: 0.0850 - val_accuracy: 0.9696
Epoch 13/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0850 - ac
curacy: 0.9688 - val_loss: 0.0820 - val_accuracy: 0.9699
Epoch 14/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0819 - ac
curacy: 0.9708 - val_loss: 0.0795 - val_accuracy: 0.9713
Epoch 15/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0809 - ac
curacy: 0.9717 - val_loss: 0.0761 - val_accuracy: 0.9751
Epoch 16/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0796 - ac
```

```
curacy: 0.9718 - val_loss: 0.0744 - val_accuracy: 0.9732
Epoch 17/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0791 - ac
curacy: 0.9722 - val_loss: 0.0715 - val_accuracy: 0.9750
Epoch 18/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0737 - ac
curacy: 0.9738 - val_loss: 0.0690 - val_accuracy: 0.9757
Epoch 19/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0718 - ac
curacy: 0.9737 - val_loss: 0.0664 - val_accuracy: 0.9768
Epoch 20/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0695 - ac
curacy: 0.9752 - val_loss: 0.0640 - val_accuracy: 0.9778
Epoch 1/20
500/500 [==============================] - 1s 2ms/step - loss: 0.9272 - ac
curacy: 0.4306 - val_loss: 0.3135 - val_accuracy: 0.8944
Epoch 2/20
500/500 [==============================] - 1s 1ms/step - loss: 0.2534 - ac
curacy: 0.9097 - val_loss: 0.1559 - val_accuracy: 0.9413
Epoch 3/20
500/500 [==============================] - 1s 2ms/step - loss: 0.1480 - ac
curacy: 0.9421 - val_loss: 0.1221 - val_accuracy: 0.9492
Epoch 4/20
500/500 [==============================] - 1s 1ms/step - loss: 0.1232 - ac
curacy: 0.9488 - val_loss: 0.1076 - val_accuracy: 0.9537
Epoch 5/20
500/500 [==============================] - 1s 1ms/step - loss: 0.1085 - ac
curacy: 0.9538 - val_loss: 0.0960 - val_accuracy: 0.9624
Epoch 6/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0971 - ac
curacy: 0.9619 - val_loss: 0.0875 - val_accuracy: 0.9671
Epoch 7/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0903 - ac
curacy: 0.9656 - val_loss: 0.0815 - val_accuracy: 0.9702
Epoch 8/20
500/500 [==============================] - 1s 2ms/step - loss: 0.0831 - ac
curacy: 0.9694 - val_loss: 0.0762 - val_accuracy: 0.9733
Epoch 9/20
500/500 [==============================] - 1s 2ms/step - loss: 0.0773 - ac
curacy: 0.9720 - val_loss: 0.0727 - val_accuracy: 0.9733
Epoch 10/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0742 - ac
curacy: 0.9743 - val_loss: 0.0684 - val_accuracy: 0.9762
Epoch 11/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0708 - ac
curacy: 0.9743 - val_loss: 0.0651 - val_accuracy: 0.9774
Epoch 12/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0700 - ac
curacy: 0.9749 - val_loss: 0.0631 - val_accuracy: 0.9778
Epoch 13/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0642 - ac
curacy: 0.9769 - val_loss: 0.0594 - val_accuracy: 0.9780
Epoch 14/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0607 - ac
```

```
curacy: 0.9780 - val_loss: 0.0588 - val_accuracy: 0.9781
Epoch 15/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0604 - ac
curacy: 0.9786 - val_loss: 0.0551 - val_accuracy: 0.9806
Epoch 16/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0569 - ac
curacy: 0.9803 - val_loss: 0.0522 - val_accuracy: 0.9821
Epoch 17/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0564 - ac
curacy: 0.9800 - val_loss: 0.0507 - val_accuracy: 0.9821
Epoch 18/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0533 - ac
curacy: 0.9815 - val_loss: 0.0498 - val_accuracy: 0.9817
Epoch 19/20
500/500 [==============================] - 1s 2ms/step - loss: 0.0506 - ac
curacy: 0.9828 - val_loss: 0.0477 - val_accuracy: 0.9839
Epoch 20/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0506 - ac
curacy: 0.9824 - val_loss: 0.0474 - val_accuracy: 0.9834
Epoch 1/20
633/633 [==============================] - 1s 2ms/step - loss: 0.7782 - ac
curacy: 0.7044 - val_loss: 0.3759 - val_accuracy: 0.7664
Epoch 2/20
633/633 [==============================] - 1s 1ms/step - loss: 0.3260 - ac
curacy: 0.8550 - val_loss: 0.2296 - val_accuracy: 0.9407
Epoch 3/20
633/633 [==============================] - 1s 1ms/step - loss: 0.2068 - ac
curacy: 0.9501 - val_loss: 0.1607 - val_accuracy: 0.9570
Epoch 4/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1489 - ac
curacy: 0.9611 - val_loss: 0.1219 - val_accuracy: 0.9681
Epoch 5/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1158 - ac
curacy: 0.9673 - val_loss: 0.0978 - val_accuracy: 0.9724
Epoch 6/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0943 - ac
curacy: 0.9727 - val_loss: 0.0821 - val_accuracy: 0.9764
Epoch 7/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0816 - ac
curacy: 0.9740 - val_loss: 0.0714 - val_accuracy: 0.9777
Epoch 8/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0713 - ac
curacy: 0.9777 - val_loss: 0.0634 - val_accuracy: 0.9806
Epoch 9/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0656 - ac
curacy: 0.9782 - val_loss: 0.0574 - val_accuracy: 0.9826
Epoch 10/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0594 - ac
curacy: 0.9802 - val_loss: 0.0534 - val_accuracy: 0.9838
Epoch 11/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0551 - ac
curacy: 0.9811 - val_loss: 0.0495 - val_accuracy: 0.9848
Epoch 12/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0514 - ac
```

```
curacy: 0.9828 - val_loss: 0.0471 - val_accuracy: 0.9867
Epoch 13/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0489 - ac
curacy: 0.9835 - val_loss: 0.0449 - val_accuracy: 0.9860
Epoch 14/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0472 - ac
curacy: 0.9837 - val_loss: 0.0417 - val_accuracy: 0.9876
Epoch 15/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0440 - ac
curacy: 0.9848 - val_loss: 0.0403 - val_accuracy: 0.9866
Epoch 16/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0426 - ac
curacy: 0.9850 - val_loss: 0.0392 - val_accuracy: 0.9880
Epoch 17/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0423 - ac
curacy: 0.9849 - val_loss: 0.0378 - val_accuracy: 0.9878
Epoch 18/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0411 - ac
curacy: 0.9858 - val_loss: 0.0367 - val_accuracy: 0.9880
Epoch 19/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0390 - ac
curacy: 0.9865 - val_loss: 0.0361 - val_accuracy: 0.9877
Epoch 20/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0374 - ac
curacy: 0.9866 - val_loss: 0.0348 - val_accuracy: 0.9884
```

In [6]:

```
acc = acc.round(2)
acc
```

Out [6]:

	train_Acc	test_Acc	time	test_size
0	98.67	98.67	18.15	10.0
1	97.79	97.79	14.77	20.0
2	97.56	97.56	17.19	10.0
3	98.35	98.35	15.23	20.0
4	98.72	98.72	17.28	10.0

In [7]:

```
ANNAcc = acc.mean()
ANNAcc
```

Out [7]:

```
train_Acc    98.218
test_Acc     98.218
time         16.524
test_size    14.000
dtype: float64
```

SVM

In [8]:

```
n = .1
for i in range(iteration):

    X = dn.iloc[:,0:5].astype('float32')
    y = dn['Class']

    encoder = LabelEncoder()
    encoder.fit(y)
    y = encoder.transform(y)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
n, random_state=0)

    sc = StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test  = sc.fit_transform(X_test)

    #time start
    start_time = time.time()


    model = SVC(kernel='poly')
    model.fit(X_train,y_train)

    #time end
    end_time = time.time()

    train_score = model.score(X_train,y_train)
    test_score  = model.score(X_test,y_test)

    #time calculation
    acc.iloc[i,2] = end_time - start_time

    #Accuracy
    acc.iloc[i,0] = train_score*100
    acc.iloc[i,1] = test_score*100

    #test size
    acc.iloc[i,3] = n*100

    if n == .1:
        n = .2
    else :
        n = .1
    print(i+1," run")
```

```
1  run
2  run
3  run
```

```
4   run
5   run
```

In [9]:

```
acc = acc.round(2)
acc
```

Out [9]:

	train_Acc	test_Acc	time	test_size
0	91.96	91.99	179.31	10.0
1	91.88	92.12	121.68	20.0
2	91.96	91.99	121.11	10.0
3	91.88	92.12	203.21	20.0
4	91.96	91.99	149.56	10.0

In [10]:

```
SVMAcc = acc.mean()
SVMAcc
```

Out [10]:

```
train_Acc      91.928
test_Acc       92.042
time          154.974
test_size      14.000
dtype: float64
```

KNN

In [11]:

```
n = .1
for i in range(iteration):

    X = dn.iloc[:,0:5].astype('float32')
    y = dn['Class']

    encoder = LabelEncoder()
    encoder.fit(y)
    y = encoder.transform(y)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 
n, random_state=0)


    #time start
    start_time = time.time()

    model = KNeighborsClassifier(n_neighbors=3)
    model.fit(X_train,y_train)

    #time end
    end_time = time.time()

    train_score = model.score(X_train,y_train)*100
    test_score  = model.score(X_test,y_test)*100

    #time calculation
    acc.iloc[i,2] = end_time - start_time

    #Accuracy
    acc.iloc[i,0] = train_score
    acc.iloc[i,1] = test_score

    #test size
    acc.iloc[i,3] = n*100

    if n == .1:
        n = .2
    else :
        n = .1
    print(i+1," run")
```

```
1  run
2  run
3  run
4  run
5  run
```

In [12]:

```
acc = acc.round(2)
acc
```

Out [12]:

	train_Acc	test_Acc	time	test_size
0	88.07	77.34	0.13	10.0
1	88.08	76.98	0.11	20.0
2	88.07	77.34	0.12	10.0
3	88.08	76.98	0.11	20.0
4	88.07	77.34	0.34	10.0

In [13]:

```
KNNAcc = acc.mean()
KNNAcc
```

Out [13]:

```
train_Acc    88.074
test_Acc     77.196
time          0.162
test_size    14.000
dtype: float64
```

GPC

In [14]:

```
n = .1
for i in range(iteration):

    X = dn.iloc[:,0:5].astype('float32')
    y = dn['Class'] #.iloc[0:100,-1]

    encoder = LabelEncoder()
    encoder.fit(y)
    y = encoder.transform(y)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
n, random_state=0)

    #time start
    start_time = time.time()

    model = GaussianProcessClassifier(multi_class='one_vs_one')
    model.fit(X_train,y_train)

    #time end
    end_time = time.time()

    train_score = model.score(X_train,y_train)
    test_score  = model.score(X_test,y_test)

    #time calculation
    acc.iloc[i,2] = end_time - start_time

    #Accuracy
    acc.iloc[i,0] = train_score*100
    acc.iloc[i,1] = test_score*100

    #test size
    acc.iloc[i,3] = n*100

    if n == .1:
        n = .2
    else :
        n = .1
    print(i+1," run")
```

```
1  run
2  run
```

```
3   run
4   run
5   run
```

In [15]:

```
acc = acc.round(2)
acc
```

Out [15]:

	train_Acc	test_Acc	time	test_size
0	100.0	73.20	98.25	10.0
1	100.0	73.75	73.01	20.0
2	100.0	73.20	96.54	10.0
3	100.0	73.75	74.31	20.0
4	100.0	73.20	99.64	10.0

In [16]:

```
GPCAcc = acc.mean()
GPCAcc
```

Out[16]:

```
train_Acc    100.00
test_Acc      73.42
time          88.35
test_size     14.00
dtype: float64
```

In [17]:

```
performance = {
    "Performance" : ['Avg Train Accuracy','Avg Test Accuracy','Avg Run Tim
e','Avg Test size'],
    "ANN"         : ANNAcc,
    "SVM"         : SVMAcc,
    "KNN"         : KNNAcc,
    "GPC"         : GPCAcc
}
Performance = pd.DataFrame(performance)
Performance
```

Out[17]:

	Performance	ANN	SVM	KNN	GPC
train_Acc	Avg Train Accuracy	98.218	91.928	88.074	100.00
test_Acc	Avg Test Accuracy	98.218	92.042	77.196	73.42
time	Avg Run Time	16.524	154.974	0.162	88.35
test_size	Avg Test size	14.000	14.000	14.000	14.00

In [18]:

```
p1 = Performance.iloc[0:3,1:5].T
p2 = pd.DataFrame(columns = p1.columns)
p2 = p2.T
p2["max"] = 0
p2["min"] = 0
p2 = p2.T
for i in range(len(p2.columns)):
    p2.iloc[0,i] = p1.iloc[:,i].max()
    p2.iloc[1,i] = p1.iloc[:,i].min()

# max-min normalizer = 0
# linear normalizer = 1
normalizer = 0

if normalizer == 0 :
    for i in range(len(p1.columns)):
        if i==2: #Non Beneficial
            p1.iloc[:,i] = (p2.iloc[0,i]-p1.iloc[:,i])/(p2.iloc[0,i] - p2.
iloc[1,i])

        else: #Beneficial
            p1.iloc[:,i] = (p1.iloc[:,i]-p2.iloc[1,i])/(p2.iloc[0,i] - p2.
iloc[1,i])

else:
    for i in range(len(p1.columns)):
        if i==2: #Non Beneficial
            p1.iloc[:,i] = p2.iloc[1,i]/p1.iloc[:,i]

        else: #Beneficial
            p1.iloc[:,i] = p1.iloc[:,i]/ p2.iloc[0,i]

weight = [0.5,0.3,0.2]
p1 = p1*weight

p1['si'] = p1.iloc[:].sum(axis=1)

p1['ri'] = p1.iloc[:,0:-2].max(axis=1)

s_best  = p1['si'].min()
s_worst = p1['si'].max()
r_best  = p1['ri'].min()
r_worst = p1['ri'].max()
neu = 0.5

p1['p-w'] = p1.apply(lambda row: neu*((row.si-s_best)/(s_worst-s_best))+(1
-neu)*((row.ri-r_best)/(r_worst-r_best)) , axis =1 )


p1 = p1.sort_values(by=['p-w'], ascending=False)
p1
```

Out[18]:

	train_Acc	test_Acc	time	si	ri	p-w
ANN	0.425289	0.300000	0.178862	0.904151	0.425289	0.917777
GPC	0.500000	0.000000	0.086071	0.586071	0.500000	0.758470
SVM	0.161580	0.225284	0.000000	0.386864	0.225284	0.304867
KNN	0.000000	0.045681	0.200000	0.245681	0.045681	0.000000

In [19]:

```
p1.round(2)
```

Out[19]:

	train_Acc	test_Acc	time	si	ri	p-w
ANN	0.43	0.30	0.18	0.90	0.43	0.92
GPC	0.50	0.00	0.09	0.59	0.50	0.76
SVM	0.16	0.23	0.00	0.39	0.23	0.30
KNN	0.00	0.05	0.20	0.25	0.05	0.00

4.ComboML_v4.0,100k

Importing Libraries

In [1]:

```
import pandas as pd
import numpy as np
import seaborn as sns

import tensorflow as tf
import keras
from keras.utils import to_categorical
from keras.models import Sequential
from keras.optimizers import SGD
from keras.layers import Dense
from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.neighbors import KNeighborsClassifier

from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import time
```

Importing Data

In [2]:

```
data = pd.read_csv('../input/thesis/dumy100000.csv')
```

MCDM

AHP

In [3]:

```
d_m = pd.read_csv('../input/pairwise/pairWiseMatrix.csv',index_col=0)

for i in range(5):
    d_m.iloc[:,i]=d_m.iloc[:,i]/d_m.iloc[:,i].sum()

d_m['criteria_weight'] = d_m.mean(axis=1)

dmat = pd.read_csv('../input/pairwise/pairWiseMatrix.csv',index_col=0)

for i in range(5):
    dmat.iloc[:,i] = dmat.iloc[:,i] * d_m.iloc[i,5]

dmat['weighted_sum_value'] = dmat.sum(axis=1)

dmat['weighted_sum_value/criteria_weight'] = dmat['weighted_sum_value']/d_
m['criteria_weight']

lamda = dmat['weighted_sum_value/criteria_weight'].mean()
r = len(d_m.index)
CI = (lamda-r)/(r-1)
k= pd.read_csv('../input/pairwise/RCI.csv')
for i in range(15):
    if i==r-1:
        RI=k.iloc[i,1]
```

```
CR=CI/RI

if CR < 0.1:
    print("The Matrix is Consistent.")
else:
     print("The Matrix is In-Consistent.")

dn = 0
dn = data.copy()
del dn['Unnamed: 0']
d1 = dn.copy()

d2 = pd.DataFrame(columns = d1.columns)
d2 = d2.T
d2["max"] = 0
d2["min"] = 0
d2 = d2.T

for i in range(len(d2.columns)):
    d2.iloc[0,i] = d1.iloc[:,i].max()
    d2.iloc[1,i] = d1.iloc[:,i].min()

# max-min normalizer = 0
# linear normalizer = 1
normalizer = 0

if normalizer == 0 :
    for i in range(len(d1.columns)):
        if i==0 or i==3: #Non Beneficial
            d1.iloc[:,i] = (d2.iloc[0,i]-d1.iloc[:,i])/(d2.iloc[0,i] - d2.
iloc[1,i])

        else: #Beneficial
            d1.iloc[:,i] = (d1.iloc[:,i]-d2.iloc[1,i])/(d2.iloc[0,i] - d2.
iloc[1,i])

else:
    for i in range(len(d1.columns)):
        if i==0 or i==3: #Non Beneficial
            d1.iloc[:,i] = d2.iloc[1,i]/d1.iloc[:,i]

        else: #Beneficial
            d1.iloc[:,i] = d1.iloc[:,i]/ d2.iloc[0,i]

w = np.array(d_m["criteria_weight"]).reshape(1,5)
d1 = d1*w

d1['performance'] = d1.iloc[:].sum(axis=1)

dn['performance'] = d1['performance']
dn = dn.sort_values(by=['performance'], ascending=True)

dn['RunCumCost']   = dn['performance'].cumsum()
```

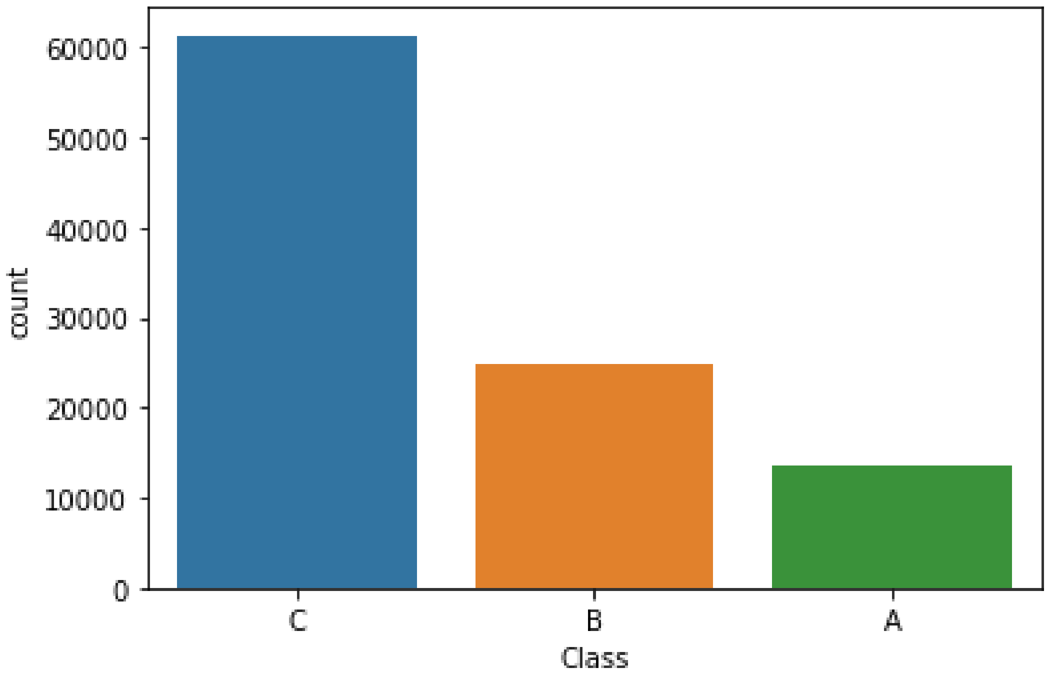
```
TotSum              = dn['performance'].sum()
dn['RunCostPerc']   = (dn['RunCumCost']/TotSum)*100

dn['Rank']          = dn['RunCostPerc'].rank()
dn['RunItemCum']    = dn['Rank'].cumsum()
TotItemSum          = dn['Rank'].sum()
dn['RunItemPerc']   = (dn['RunItemCum']/TotItemSum)*100

def ABC_segmentation(perc):
    '''
    top A - top 20%, C - last 50% and B - between A & C

    '''
    if perc > 80 :
        return 'A'
    elif perc >= 50 and perc <= 80:
        return 'B'
    elif perc < 50:
        return 'C'

dn['Class'] = dn['RunCostPerc'].apply(ABC_segmentation)

ax = sns.countplot(x = dn['Class'],data = dn,label= 'Count')
dn['Class'].value_counts()

dn['productid'] = dn.index
dn = dn.sort_values(by=['productid'], ascending=True)
del dn['productid']
AHP_Class = dn['Class']

C,B,A = dn['Class'].value_counts()
print(round(100*A/dn['Class'].value_counts().sum(),2),"% ->","A:",A)
print(round(100*B/dn['Class'].value_counts().sum(),2),"% ->","B:",B)
print(round(100*C/dn['Class'].value_counts().sum(),2),"% ->","C:",C)
print("Total: ",dn['Class'].value_counts().sum())
dn.head()
```

```
The Matrix is Consistent.
13.65 % -> A: 13648
24.95 % -> B: 24953
61.4 % -> C: 61399
Total:  100000
```

Out [3]:

	mfg	sell	demand	fcost	rma	performance	RunCumCost	RunCostPerc	Rank	RunItemCum	RunItemPerc	Class
0	71	156.91	1595	49.7	3	0.276224	1494.307675	2.985227	6658.0	2.216781e+07	0.443352	C
1	474	1047.54	6877	331.8	3	0.873845	49945.574810	99.777899	99874.0	4.987458e+09	99.748160	A

2	70	154.70	5182	49.0	3	0.471743	15481.763973	30.928423	42637.0	9.089782e+08	18.179382	C
3	417	921.57	3914	291.9	3	0.668061	40135.751226	80.180495	86488.0	3.740130e+09	74.801858	A
4	239	528.19	4496	167.3	2	0.486306	17225.943313	34.412827	46278.0	1.070850e+09	21.416781	C



SAW

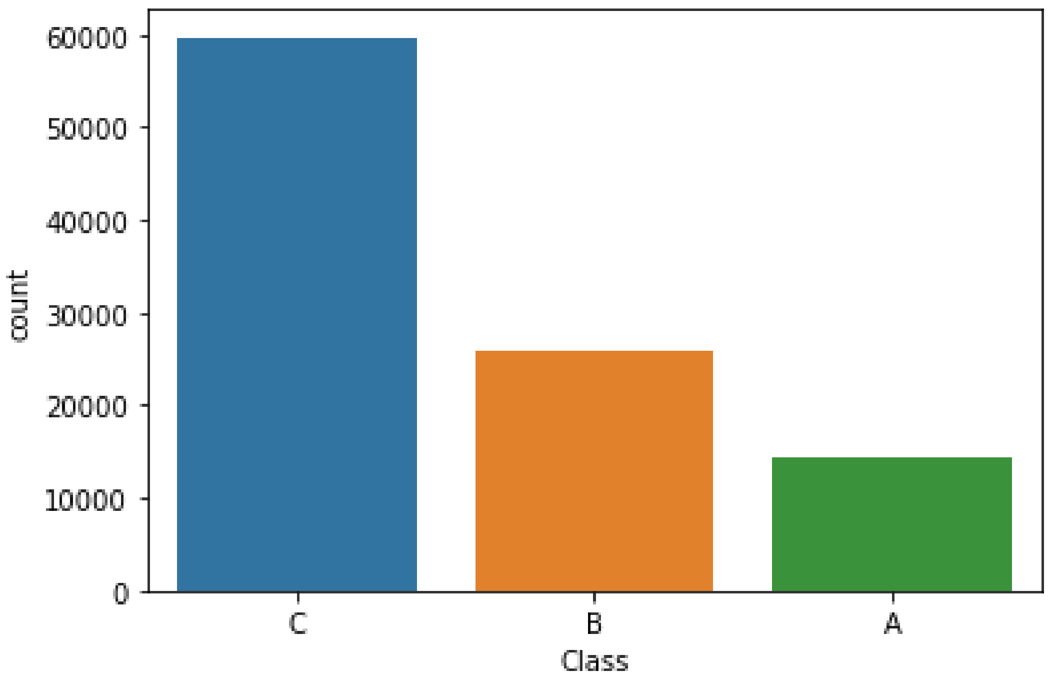
In [4]:

```
dn = 0
dn = data.copy()
del dn['Unnamed: 0']
d1 = dn.copy()

d2 = pd.DataFrame(columns = d1.columns)
d2 = d2.T
d2["max"] = 0
d2["min"] = 0
d2 = d2.T

for i in range(len(d2.columns)):
    d2.iloc[0,i] = d1.iloc[:,i].max()
    d2.iloc[1,i] = d1.iloc[:,i].min()

# max-min normalizer = 0
# linear normalizer = 1
normalizer = 0

if normalizer == 0 :
    for i in range(len(d1.columns)):
        if i==0 or i==3: #Non Beneficial
            d1.iloc[:,i] = (d2.iloc[0,i]-d1.iloc[:,i])/(d2.iloc[0,i] - d2.
iloc[1,i])

        else: #Beneficial
            d1.iloc[:,i] = (d1.iloc[:,i]-d2.iloc[1,i])/(d2.iloc[0,i] - d2.
iloc[1,i])

else:
    for i in range(len(d1.columns)):
        if i==0 or i==3: #Non Beneficial
            d1.iloc[:,i] = d2.iloc[1,i]/d1.iloc[:,i]

        else: #Beneficial
            d1.iloc[:,i] = d1.iloc[:,i]/ d2.iloc[0,i]

w  = [0.15, 0.20, 0.30, 0.15, 0.20]
d1 = d1*w

d1['performance'] = d1.iloc[:].sum(axis=1)

dn['performance'] = d1['performance']
dn = dn.sort_values(by=['performance'], ascending=True)

dn['RunCumCost']   = dn['performance'].cumsum()
TotSum             = dn['performance'].sum()
dn['RunCostPerc']  = (dn['RunCumCost']/TotSum)*100

def ABC_segmentation(perc):
    '''
    top A - top 20%, C - last 50% and B - between A & C

    '''
    if perc > 80 :
        return 'A'
    elif perc >= 50 and perc <= 80:
        return 'B'
    elif perc < 50:
        return 'C'

dn['Class'] = dn['RunCostPerc'].apply(ABC_segmentation)

ax = sns.countplot(x = dn['Class'],data = dn,label= 'Count')
dn['Class'].value_counts()

dn['productid'] = dn.index
dn = dn.sort_values(by=['productid'], ascending=True)
del dn['productid']

SAW_Class = dn['Class']

C,B,A = dn['Class'].value_counts()
print(round(100*A/dn['Class'].value_counts().sum(),2),"% ->","A:",A)
print(round(100*B/dn['Class'].value_counts().sum(),2),"% ->","B:",B)
print(round(100*C/dn['Class'].value_counts().sum(),2),"% ->","C:",C)
print("Total: ",dn['Class'].value_counts().sum())
dn.head()
```

```
14.43 % -> A: 14431
25.85 % -> B: 25850
59.72 % -> C: 59719
Total:  100000
```

Out [4]:

	mfg	sell	demand	fcost	rma	performance	RunCumCost	RunCostPerc	Class
0	71	156.91	1595	49.7	3	0.500506	20035.145963	40.033583	C
1	474	1047.54	6877	331.8	3	0.698912	45792.067588	91.500235	A
2	70	154.70	5182	49.0	3	0.696419	45583.448850	91.083379	A
3	417	921.57	3914	291.9	3	0.549959	27976.109958	55.900962	B
4	239	528.19	4496	167.3	2	0.521354	23532.483611	47.021851	C



VIKOR

In [5]:

```
dn = 0
dn = data.copy() #pd.read_csv('file:///home/shatiil/shatiil/DataScience/Th
esis/1.DATA/dumy10000.csv')
del dn['Unnamed: 0']
d1 = dn.copy()

d2 = pd.DataFrame(columns = d1.columns)
```

```
d2 = d2.T
d2["max"] = 0
d2["min"] = 0
d2 = d2.T

for i in range(len(d2.columns)):
    d2.iloc[0,i] = d1.iloc[:,i].max()
    d2.iloc[1,i] = d1.iloc[:,i].min()

# max-min normalizer = 0
# linear normalizer = 1
normalizer = 0

if normalizer == 0 :
    for i in range(len(d1.columns)):
        if i==0 or i==3: #Non Beneficial
            d1.iloc[:,i] = (d2.iloc[0,i]-d1.iloc[:,i])/(d2.iloc[0,i] - d2.
iloc[1,i])

        else: #Beneficial
            d1.iloc[:,i] = (d1.iloc[:,i]-d2.iloc[1,i])/(d2.iloc[0,i] - d2.
iloc[1,i])

else:
    for i in range(len(d1.columns)):
        if i==0 or i==3: #Non Beneficial
            d1.iloc[:,i] = d2.iloc[1,i]/d1.iloc[:,i]

        else: #Beneficial
            d1.iloc[:,i] = d1.iloc[:,i]/ d2.iloc[0,i]

w  = [0.15, 0.20, 0.30, 0.15, 0.20]
d1 = d1*w

d1['si'] = d1.iloc[:].sum(axis=1)

d1['ri'] = d1.iloc[:,0:-2].max(axis=1)

s_best  = d1['si'].min()
s_worst = d1['si'].max()
r_best  = d1['ri'].min()
r_worst = d1['ri'].max()
neu = 0.5

d1['qi'] = d1.apply(lambda row: neu*((row.si-s_best)/(s_worst-s_best))+(1-
neu)*((row.ri-r_best)/(r_worst-r_best)) , axis =1 )

dn['qi'] = d1['qi']

dn = dn.sort_values(by=['qi'], ascending=False)

dn['RunCumCost']   = dn['qi'].cumsum()
TotSum             = dn['qi'].sum()
dn['RunCostPerc']  = (dn['RunCumCost']/TotSum)*100
```

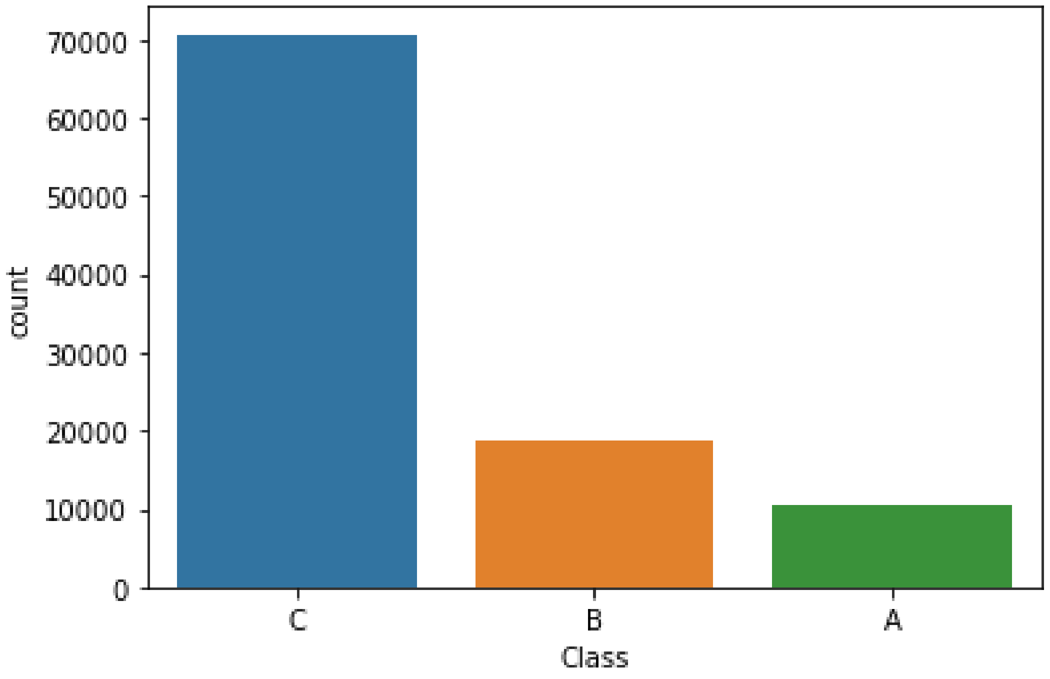
```
dn['Rank']          = dn['RunCostPerc'].rank()
dn['RunItemCum']    = dn['Rank'].cumsum()
TotItemSum          = dn['Rank'].sum()
dn['RunItemPerc']   = (dn['RunItemCum']/TotItemSum)*100

def ABC_segmentation(perc):
    '''
    top A - top 20%, C - last 50% and B - between A & C

    '''
    if perc > 80 :
        return 'A'
    elif perc >= 50 and perc <= 80:
        return 'B'
    elif perc < 50:
        return 'C'

dn['Class'] = dn['RunItemPerc'].apply(ABC_segmentation)

ax = sns.countplot(x = dn['Class'],data = dn,label= 'Count')
dn['Class'].value_counts()

dn['productid'] = dn.index
dn = dn.sort_values(by=['productid'], ascending=True)
del dn['productid']
VIKOR_Class = dn['Class']

C,B,A = dn['Class'].value_counts()
print(round(100*A/dn['Class'].value_counts().sum(),2),"% ->","A:",A)
print(round(100*B/dn['Class'].value_counts().sum(),2),"% ->","B:",B)
print(round(100*C/dn['Class'].value_counts().sum(),2),"% ->","C:",C)
print("Total: ",dn['Class'].value_counts().sum())
dn.head()
```

```
10.56 % -> A: 10558
18.73 % -> B: 18732
70.71 % -> C: 70710
Total:  100000
```

Out [5]:

	mfg	sell	demand	fcost	rma	qi	RunCumCost	RunCostPerc	Rank	RunItemCum	RunItemPerc	Class
0	71	156.91	1595	49.7	3	0.383965	36257.514659	76.779078	59291.0	1.757741e+09	35.154468	C
1	474	1047.54	6877	331.8	3	0.901189	1962.408386	4.155605	2101.0	2.208151e+06	0.044163	C
2	70	154.70	5182	49.0	3	0.683190	16176.765170	34.255992	20391.0	2.079066e+08	4.158091	C
3	417	921.57	3914	291.9	3	0.473220	29837.401955	63.183818	44227.0	9.780359e+08	19.560522	C

4	23 9	528.1 9	4496	167 .3	2	0.449 204	31487.10 5629	66.6772 38	4780 6.0	1.142731 e+09	22.8543 86	C



COMBO

In [6]:

```
Data = {
        "AHP_Class"   : AHP_Class,
        "SAW_Class"   : SAW_Class,
        "VIKOR_Class" : VIKOR_Class
       }
combo = pd.DataFrame(Data)
combo.head()
```

Out [6]:

	AHP_Class	SAW_Class	VIKOR_Class
0	C	C	C
1	A	A	C
2	C	A	C

3	A	B	C
4	C	C	C

In [7]:

```
combo['ComboClass'] = 0
m_c = 1
for i in range(len(combo.index)):

    if combo.iloc[i,0] == combo.iloc[i,1] or combo.iloc[i,0] == combo.iloc
[i,2]:
        combo.iloc[i,3] = combo.iloc[i,0]

    elif combo.iloc[i,1] == combo.iloc[i,0] or combo.iloc[i,1] == combo.il
oc[i,2]:
        combo.iloc[i,3] = combo.iloc[i,1]

    else :
        combo.iloc[i,3] = "ABC"#combo.iloc[i,m_c] #
combo.head()
```

Out[7]:

	AHP_Class	SAW_Class	VIKOR_Class	ComboClass
0	C	C	C	C
1	A	A	C	A
2	C	A	C	C
3	A	B	C	ABC
4	C	C	C	C

In [8]:

```
combo['ComboClass'].value_counts()
```

Out[8]:

```
C      76227
ABC    11252
B       7438
A       5083
Name: ComboClass, dtype: int64
```

In [9]:

```
m_c = 1 #ANN-SAW with best accuracy
for i in range(len(combo.index)):

    if combo.iloc[i,-1] == 'ABC':
        combo.iloc[i,-1] = combo.iloc[i,m_c]
combo.head()
```

Out[9]:

	AHP_Class	SAW_Class	VIKOR_Class	ComboClass
0	C	C	C	C
1	A	A	C	A
2	C	A	C	C
3	A	B	C	B
4	C	C	C	C

In [10]:

```
combo['ComboClass'].value_counts()
```

Out [10]:

```
C    76227
B    13543
A    10230
Name: ComboClass, dtype: int64
```

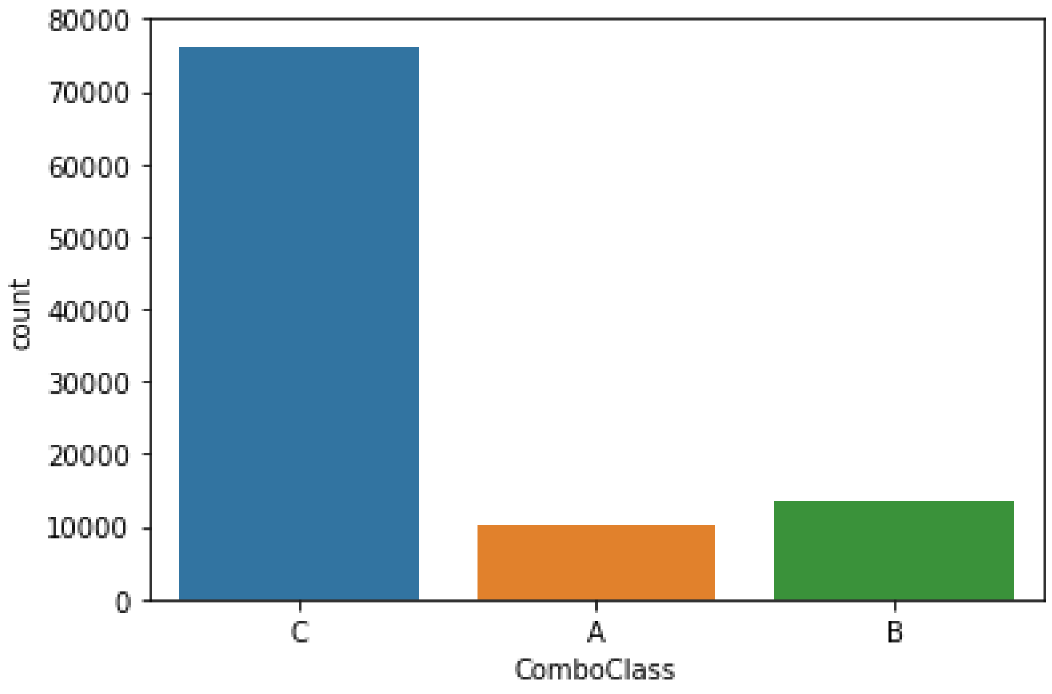
In [11]:

```
ax = sns.countplot(x = combo['ComboClass'],data = combo,label= 'Count')
C,B,A = combo['ComboClass'].value_counts()
print(round(100*A/combo['ComboClass'].value_counts().sum(),2),"% ->","A:",
A)
print(round(100*B/combo['ComboClass'].value_counts().sum(),2),"% ->","B:",
B)
print(round(100*C/combo['ComboClass'].value_counts().sum(),2),"% ->","C:",
C)
print("Total: "  ,combo['ComboClass'].value_counts().sum())
combo.head()
```

```
10.23 % -> A: 10230
13.54 % -> B: 13543
76.23 % -> C: 76227
Total:  100000
```

Out[11]:

	AHP_Class	SAW_Class	VIKOR_Class	ComboClass
0	C	C	C	C
1	A	A	C	A
2	C	A	C	C
3	A	B	C	B
4	C	C	C	C



In [12]:

```
dn = 0
dn = data.copy()
dn['ComboClass'] = combo['ComboClass']
del dn['Unnamed: 0']
dn.head()
```

Out [12]:

	mfg	sell	demand	fcost	rma	ComboClass
0	71	156.91	1595	49.7	3	C
1	474	1047.54	6877	331.8	3	A
2	70	154.70	5182	49.0	3	C
3	417	921.57	3914	291.9	3	B
4	239	528.19	4496	167.3	2	C

Machine Learning

In [13]:

```
iteration = 5
row_col = np.zeros([iteration,])

acc = pd.DataFrame()
acc['train_Acc'] = row_col
acc['test_Acc']  = row_col
acc['time']      = row_col
acc['test_size'] = row_col
start_time   = 0
```

```
end_time    = 0
train_score = 0
test_score  = 0

#split_size
n = .1
```

ANN

In [14]:

```
for i in range(iteration):

    X = dn.iloc[:,0:5].astype('float32')
    y = dn['ComboClass'] #.iloc[0:100,-1]

    encoder = LabelEncoder()
    encoder.fit(y)
    y = encoder.transform(y)

    y = to_categorical(y)

    X_train,X_test,y_train,y_test = train_test_split(X, y ,test_size = n,
random_state=0)
    X_train,X_val,y_train,y_val = train_test_split(X_train, y_train ,test_
size = n, random_state=0)

    sc = StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test  = sc.fit_transform(X_test)
    X_val  = sc.fit_transform(X_val)

    #time start
    start_time = time.time()

    model = Sequential()
    model.add(Dense(5, activation='relu', input_shape=(X_train.shape[1],))
)
    model.add(Dense(3, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metri
cs=['accuracy'])
    model.fit(X_train,y_train,batch_size = 128, epochs = 20, validation_da
ta = (X_val,y_val))

    #time end
    end_time = time.time()

    train_score = model.evaluate(X_train,y_train,verbose=0)
    test_score = model.evaluate(X_test,y_test,verbose=0)

    #time calculation
    acc.iloc[i,2] = end_time - start_time
    #Accuracy
    acc.iloc[i,0] = train_score[1]
    acc.iloc[i,1] = train_score[1]

    #test size
    acc.iloc[i,3] = n*100

    if n == .1:
        n = .2
    else :
        n = .1
```

```
Epoch 1/20
633/633 [==============================] - 2s 2ms/step - loss: 0.7664 - ac
curacy: 0.5902 - val_loss: 0.3453 - val_accuracy: 0.8377
Epoch 2/20
633/633 [==============================] - 1s 1ms/step - loss: 0.3121 - ac
curacy: 0.8528 - val_loss: 0.2387 - val_accuracy: 0.9223
Epoch 3/20
633/633 [==============================] - 1s 1ms/step - loss: 0.2101 - ac
curacy: 0.9391 - val_loss: 0.1385 - val_accuracy: 0.9640
Epoch 4/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1204 - ac
curacy: 0.9688 - val_loss: 0.0967 - val_accuracy: 0.9722
Epoch 5/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0864 - ac
curacy: 0.9749 - val_loss: 0.0777 - val_accuracy: 0.9736
Epoch 6/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0712 - ac
curacy: 0.9772 - val_loss: 0.0681 - val_accuracy: 0.9754
Epoch 7/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0629 - ac
curacy: 0.9792 - val_loss: 0.0624 - val_accuracy: 0.9754
Epoch 8/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0578 - ac
curacy: 0.9800 - val_loss: 0.0584 - val_accuracy: 0.9769
Epoch 9/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0532 - ac
curacy: 0.9810 - val_loss: 0.0556 - val_accuracy: 0.9768
Epoch 10/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0510 - ac
curacy: 0.9806 - val_loss: 0.0538 - val_accuracy: 0.9764
Epoch 11/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0465 - ac
curacy: 0.9827 - val_loss: 0.0521 - val_accuracy: 0.9754
Epoch 12/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0465 - ac
curacy: 0.9819 - val_loss: 0.0503 - val_accuracy: 0.9779
Epoch 13/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0437 - ac
curacy: 0.9831 - val_loss: 0.0511 - val_accuracy: 0.9763
Epoch 14/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0429 - ac
```

```
curacy: 0.9829 - val_loss: 0.0491 - val_accuracy: 0.9766
Epoch 15/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0425 - ac
curacy: 0.9830 - val_loss: 0.0473 - val_accuracy: 0.9780
Epoch 16/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0416 - ac
curacy: 0.9835 - val_loss: 0.0489 - val_accuracy: 0.9764
Epoch 17/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0412 - ac
curacy: 0.9834 - val_loss: 0.0462 - val_accuracy: 0.9786
Epoch 18/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0405 - ac
curacy: 0.9832 - val_loss: 0.0454 - val_accuracy: 0.9792
Epoch 19/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0401 - ac
curacy: 0.9833 - val_loss: 0.0455 - val_accuracy: 0.9783
Epoch 20/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0391 - ac
curacy: 0.9839 - val_loss: 0.0470 - val_accuracy: 0.9773
Epoch 1/20
500/500 [==============================] - 1s 2ms/step - loss: 0.9987 - ac
curacy: 0.4787 - val_loss: 0.4389 - val_accuracy: 0.7568
Epoch 2/20
500/500 [==============================] - 1s 2ms/step - loss: 0.3852 - ac
curacy: 0.7739 - val_loss: 0.3018 - val_accuracy: 0.8502
Epoch 3/20
500/500 [==============================] - 1s 2ms/step - loss: 0.2799 - ac
curacy: 0.8650 - val_loss: 0.2297 - val_accuracy: 0.8883
Epoch 4/20
500/500 [==============================] - 1s 2ms/step - loss: 0.2152 - ac
curacy: 0.9123 - val_loss: 0.1759 - val_accuracy: 0.9323
Epoch 5/20
500/500 [==============================] - 1s 2ms/step - loss: 0.1681 - ac
curacy: 0.9383 - val_loss: 0.1515 - val_accuracy: 0.9396
Epoch 6/20
500/500 [==============================] - 1s 1ms/step - loss: 0.1484 - ac
curacy: 0.9421 - val_loss: 0.1351 - val_accuracy: 0.9460
Epoch 7/20
500/500 [==============================] - 1s 2ms/step - loss: 0.1334 - ac
curacy: 0.9449 - val_loss: 0.1225 - val_accuracy: 0.9506
Epoch 8/20
500/500 [==============================] - 1s 2ms/step - loss: 0.1201 - ac
curacy: 0.9516 - val_loss: 0.1109 - val_accuracy: 0.9566
Epoch 9/20
500/500 [==============================] - 1s 2ms/step - loss: 0.1095 - ac
curacy: 0.9548 - val_loss: 0.0999 - val_accuracy: 0.9612
Epoch 10/20
500/500 [==============================] - 1s 2ms/step - loss: 0.0972 - ac
curacy: 0.9599 - val_loss: 0.0894 - val_accuracy: 0.9657
Epoch 11/20
500/500 [==============================] - 1s 2ms/step - loss: 0.0871 - ac
curacy: 0.9654 - val_loss: 0.0810 - val_accuracy: 0.9702
Epoch 12/20
500/500 [==============================] - 1s 2ms/step - loss: 0.0801 - ac
```

```
curacy: 0.9701 - val_loss: 0.0757 - val_accuracy: 0.9737
Epoch 13/20
500/500 [==============================] - 1s 2ms/step - loss: 0.0746 - ac
curacy: 0.9741 - val_loss: 0.0691 - val_accuracy: 0.9761
Epoch 14/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0702 - ac
curacy: 0.9757 - val_loss: 0.0648 - val_accuracy: 0.9786
Epoch 15/20
500/500 [==============================] - 1s 2ms/step - loss: 0.0660 - ac
curacy: 0.9777 - val_loss: 0.0587 - val_accuracy: 0.9825
Epoch 16/20
500/500 [==============================] - 1s 2ms/step - loss: 0.0586 - ac
curacy: 0.9839 - val_loss: 0.0547 - val_accuracy: 0.9849
Epoch 17/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0551 - ac
curacy: 0.9846 - val_loss: 0.0494 - val_accuracy: 0.9884
Epoch 18/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0527 - ac
curacy: 0.9856 - val_loss: 0.0469 - val_accuracy: 0.9891
Epoch 19/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0488 - ac
curacy: 0.9879 - val_loss: 0.0461 - val_accuracy: 0.9866
Epoch 20/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0471 - ac
curacy: 0.9877 - val_loss: 0.0427 - val_accuracy: 0.9889
Epoch 1/20
633/633 [==============================] - 1s 2ms/step - loss: 0.7476 - ac
curacy: 0.7289 - val_loss: 0.3408 - val_accuracy: 0.7602
Epoch 2/20
633/633 [==============================] - 1s 1ms/step - loss: 0.3026 - ac
curacy: 0.8423 - val_loss: 0.2284 - val_accuracy: 0.9554
Epoch 3/20
633/633 [==============================] - 1s 1ms/step - loss: 0.2048 - ac
curacy: 0.9633 - val_loss: 0.1590 - val_accuracy: 0.9672
Epoch 4/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1466 - ac
curacy: 0.9712 - val_loss: 0.1249 - val_accuracy: 0.9701
Epoch 5/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1152 - ac
curacy: 0.9722 - val_loss: 0.1035 - val_accuracy: 0.9698
Epoch 6/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0963 - ac
curacy: 0.9724 - val_loss: 0.0908 - val_accuracy: 0.9713
Epoch 7/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0825 - ac
curacy: 0.9743 - val_loss: 0.0823 - val_accuracy: 0.9716
Epoch 8/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0745 - ac
curacy: 0.9753 - val_loss: 0.0775 - val_accuracy: 0.9709
Epoch 9/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0681 - ac
curacy: 0.9758 - val_loss: 0.0724 - val_accuracy: 0.9718
Epoch 10/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0636 - ac
```

```
curacy: 0.9767 - val_loss: 0.0694 - val_accuracy: 0.9713
Epoch 11/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0606 - ac
curacy: 0.9773 - val_loss: 0.0678 - val_accuracy: 0.9730
Epoch 12/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0606 - ac
curacy: 0.9763 - val_loss: 0.0647 - val_accuracy: 0.9734
Epoch 13/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0577 - ac
curacy: 0.9774 - val_loss: 0.0638 - val_accuracy: 0.9729
Epoch 14/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0565 - ac
curacy: 0.9775 - val_loss: 0.0645 - val_accuracy: 0.9709
Epoch 15/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0546 - ac
curacy: 0.9783 - val_loss: 0.0614 - val_accuracy: 0.9742
Epoch 16/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0527 - ac
curacy: 0.9791 - val_loss: 0.0617 - val_accuracy: 0.9731
Epoch 17/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0537 - ac
curacy: 0.9783 - val_loss: 0.0602 - val_accuracy: 0.9749
Epoch 18/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0527 - ac
curacy: 0.9792 - val_loss: 0.0594 - val_accuracy: 0.9737
Epoch 19/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0509 - ac
curacy: 0.9796 - val_loss: 0.0578 - val_accuracy: 0.9741
Epoch 20/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0507 - ac
curacy: 0.9793 - val_loss: 0.0580 - val_accuracy: 0.9766
Epoch 1/20
500/500 [==============================] - 1s 2ms/step - loss: 0.7736 - ac
curacy: 0.6611 - val_loss: 0.3769 - val_accuracy: 0.7568
Epoch 2/20
500/500 [==============================] - 1s 1ms/step - loss: 0.3439 - ac
curacy: 0.7778 - val_loss: 0.3008 - val_accuracy: 0.8654
Epoch 3/20
500/500 [==============================] - 1s 1ms/step - loss: 0.2823 - ac
curacy: 0.8747 - val_loss: 0.2666 - val_accuracy: 0.8739
Epoch 4/20
500/500 [==============================] - 1s 2ms/step - loss: 0.2562 - ac
curacy: 0.8769 - val_loss: 0.2426 - val_accuracy: 0.8751
Epoch 5/20
500/500 [==============================] - 1s 1ms/step - loss: 0.2279 - ac
curacy: 0.8842 - val_loss: 0.2228 - val_accuracy: 0.8809
Epoch 6/20
500/500 [==============================] - 1s 1ms/step - loss: 0.2115 - ac
curacy: 0.8879 - val_loss: 0.1811 - val_accuracy: 0.9556
Epoch 7/20
500/500 [==============================] - 1s 1ms/step - loss: 0.1632 - ac
curacy: 0.9608 - val_loss: 0.1219 - val_accuracy: 0.9751
Epoch 8/20
500/500 [==============================] - 1s 2ms/step - loss: 0.1129 - ac
```

```
curacy: 0.9769 - val_loss: 0.0938 - val_accuracy: 0.9775
Epoch 9/20
500/500 [==============================] - 1s 2ms/step - loss: 0.0864 - ac
curacy: 0.9836 - val_loss: 0.0772 - val_accuracy: 0.9829
Epoch 10/20
500/500 [==============================] - 1s 2ms/step - loss: 0.0749 - ac
curacy: 0.9839 - val_loss: 0.0672 - val_accuracy: 0.9838
Epoch 11/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0661 - ac
curacy: 0.9845 - val_loss: 0.0600 - val_accuracy: 0.9846
Epoch 12/20
500/500 [==============================] - 1s 2ms/step - loss: 0.0590 - ac
curacy: 0.9849 - val_loss: 0.0547 - val_accuracy: 0.9845
Epoch 13/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0537 - ac
curacy: 0.9865 - val_loss: 0.0515 - val_accuracy: 0.9839
Epoch 14/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0504 - ac
curacy: 0.9855 - val_loss: 0.0470 - val_accuracy: 0.9851
Epoch 15/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0479 - ac
curacy: 0.9860 - val_loss: 0.0447 - val_accuracy: 0.9854
Epoch 16/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0453 - ac
curacy: 0.9872 - val_loss: 0.0417 - val_accuracy: 0.9868
Epoch 17/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0417 - ac
curacy: 0.9881 - val_loss: 0.0396 - val_accuracy: 0.9881
Epoch 18/20
500/500 [==============================] - 1s 2ms/step - loss: 0.0398 - ac
curacy: 0.9877 - val_loss: 0.0382 - val_accuracy: 0.9872
Epoch 19/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0386 - ac
curacy: 0.9886 - val_loss: 0.0387 - val_accuracy: 0.9856
Epoch 20/20
500/500 [==============================] - 1s 1ms/step - loss: 0.0378 - ac
curacy: 0.9878 - val_loss: 0.0350 - val_accuracy: 0.9893
Epoch 1/20
633/633 [==============================] - 1s 2ms/step - loss: 0.7733 - ac
curacy: 0.6439 - val_loss: 0.3701 - val_accuracy: 0.7602
Epoch 2/20
633/633 [==============================] - 1s 1ms/step - loss: 0.3244 - ac
curacy: 0.8036 - val_loss: 0.2370 - val_accuracy: 0.9373
Epoch 3/20
633/633 [==============================] - 1s 1ms/step - loss: 0.2160 - ac
curacy: 0.9452 - val_loss: 0.1764 - val_accuracy: 0.9496
Epoch 4/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1609 - ac
curacy: 0.9579 - val_loss: 0.1406 - val_accuracy: 0.9582
Epoch 5/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1283 - ac
curacy: 0.9639 - val_loss: 0.1177 - val_accuracy: 0.9621
Epoch 6/20
633/633 [==============================] - 1s 1ms/step - loss: 0.1077 - ac
```

```
curacy: 0.9687 - val_loss: 0.1022 - val_accuracy: 0.9641
Epoch 7/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0919 - ac
curacy: 0.9725 - val_loss: 0.0898 - val_accuracy: 0.9686
Epoch 8/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0829 - ac
curacy: 0.9738 - val_loss: 0.0812 - val_accuracy: 0.9698
Epoch 9/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0746 - ac
curacy: 0.9757 - val_loss: 0.0748 - val_accuracy: 0.9709
Epoch 10/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0655 - ac
curacy: 0.9788 - val_loss: 0.0690 - val_accuracy: 0.9730
Epoch 11/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0617 - ac
curacy: 0.9780 - val_loss: 0.0648 - val_accuracy: 0.9737
Epoch 12/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0580 - ac
curacy: 0.9789 - val_loss: 0.0625 - val_accuracy: 0.9732
Epoch 13/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0559 - ac
curacy: 0.9791 - val_loss: 0.0610 - val_accuracy: 0.9719
Epoch 14/20
633/633 [==============================] - 1s 2ms/step - loss: 0.0537 - ac
curacy: 0.9799 - val_loss: 0.0567 - val_accuracy: 0.9762
Epoch 15/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0507 - ac
curacy: 0.9810 - val_loss: 0.0565 - val_accuracy: 0.9740
Epoch 16/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0492 - ac
curacy: 0.9806 - val_loss: 0.0554 - val_accuracy: 0.9750
Epoch 17/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0477 - ac
curacy: 0.9810 - val_loss: 0.0534 - val_accuracy: 0.9756
Epoch 18/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0473 - ac
curacy: 0.9809 - val_loss: 0.0520 - val_accuracy: 0.9762
Epoch 19/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0460 - ac
curacy: 0.9811 - val_loss: 0.0539 - val_accuracy: 0.9733
Epoch 20/20
633/633 [==============================] - 1s 1ms/step - loss: 0.0453 - ac
curacy: 0.9817 - val_loss: 0.0519 - val_accuracy: 0.9737
```

In [15]:

```
acc = acc.round(2)
acc
```

Out [15]:

	train_Acc	test_Acc	time	test_size
0	0.98	0.98	18.95	10.0

1	0.99	0.99	16.02	20.0
2	0.98	0.98	18.02	10.0
3	0.99	0.99	15.62	20.0
4	0.98	0.98	18.16	10.0

In [16]:

```
ANNAcc = acc.mean()
ANNAcc
```

Out [16]:

```
train_Acc     0.984
test_Acc      0.984
time         17.354
test_size    14.000
dtype: float64
```

SVM

In [17]:

```
n = .1
for i in range(iteration):

    X = dn.iloc[:,0:5].astype('float32')
    y = dn['ComboClass']

    encoder = LabelEncoder()
    encoder.fit(y)
    y = encoder.transform(y)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
n, random_state=0)

    #time start
    start_time = time.time()


    model = SVC(kernel='poly')
    model.fit(X_train,y_train)

    #time end
    end_time = time.time()

    train_score = model.score(X_train,y_train)
    test_score  = model.score(X_test,y_test)

    #time calculation
    acc.iloc[i,2] = end_time - start_time

    #Accuracy
    acc.iloc[i,0] = train_score
    acc.iloc[i,1] = test_score

    #test size
    acc.iloc[i,3] = n*100

    if n == .1:
        n = .2
    else :
        n = .1
    print(i+1," run")
```

```
1  run
2  run
3  run
4  run
5  run
```

In [18]:

```
acc = acc.round(2)
acc
```

Out [18]:

	train_Acc	test_Acc	time	test_size
0	0.77	0.78	212.65	10.0
1	0.77	0.78	175.65	20.0
2	0.77	0.78	212.85	10.0
3	0.77	0.78	175.44	20.0
4	0.77	0.78	212.13	10.0

In [19]:

```
SVMAcc = acc.mean()
SVMAcc
```

Out[19]:

```
train_Acc      0.770
test_Acc       0.780
time         197.744
test_size     14.000
dtype: float64
```

KNN

In [20]:

```
n = .1
for i in range(iteration):

    X = dn.iloc[:,0:5].astype('float32')
    y = dn['ComboClass']

    encoder = LabelEncoder()
    encoder.fit(y)
    y = encoder.transform(y)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
n, random_state=0)


    #time start
    start_time = time.time()

    model = KNeighborsClassifier(n_neighbors=3)
    model.fit(X_train,y_train)

    #time end
    end_time = time.time()

    train_score = model.score(X_train,y_train)
    test_score  = model.score(X_test,y_test)

    #time calculation
    acc.iloc[i,2] = end_time - start_time

    #Accuracy
    acc.iloc[i,0] = train_score
    acc.iloc[i,1] = test_score

    #test size
    acc.iloc[i,3] = n*100

    if n == .1:
        n = .2
    else :
        n = .1
    print(i+1," run")
```

```
1  run
2  run
3  run
4  run
5  run
```

In [21]:

```
acc = acc.round(2)
acc
```

Out[21]:

	train_Acc	test_Acc	time	test_size
0	0.87	0.76	0.11	10.0
1	0.87	0.75	0.10	20.0
2	0.87	0.76	0.11	10.0
3	0.87	0.75	0.10	20.0
4	0.87	0.76	0.11	10.0

In [22]:

```
KNNAcc = acc.mean()
KNNAcc
```

Out [22]:

```
train_Acc     0.870
test_Acc      0.756
time          0.106
test_size    14.000
dtype: float64
```

GPC

In [23]:

```
n = .1
for i in range(iteration):

    X = dn.iloc[:,0:5].astype('float32')
    y = dn['Class'] #.iloc[0:100,-1]

    encoder = LabelEncoder()
    encoder.fit(y)
    y = encoder.transform(y)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
n, random_state=0)


    #time start
    start_time = time.time()

    model = GaussianProcessClassifier(multi_class='one_vs_one')
    model.fit(X_train,y_train)

    #time end
    end_time = time.time()

    train_score = model.score(X_train,y_train)
    test_score  = model.score(X_test,y_test)

    #time calculation
    acc.iloc[i,2] = end_time - start_time

    #Accuracy
    acc.iloc[i,0] = train_score
    acc.iloc[i,1] = test_score

    #test size
    acc.iloc[i,3] = n*100

    if n == .1:
        n = .2
    else :
        n = .1
    print(i+1," run")
```

```
1  run
2  run
3  run
4  run
5  run
```

In [24]:

```
acc = acc.round(2)
acc
```

Out[24]:

	train_Acc	test_Acc	time	test_size
0	1.0	0.75	144.81	10.0
1	1.0	0.76	102.65	20.0
2	1.0	0.75	138.78	10.0
3	1.0	0.76	102.15	20.0
4	1.0	0.75	152.64	10.0

In [25]:

```
GPCAcc = acc.mean()
GPCAcc
```

Out[25]:

```
train_Acc      1.000
test_Acc       0.754
time         128.206
test_size     14.000
dtype: float64
```

In [26]:

```
performance = {
    "Performance" : ['Avg Train Accuracy','Avg Test Accuracy','Avg Run Tim
e','Avg Test size'],
    "ANN"         : ANNAcc,
    "SVM"         : SVMAcc,
    "KNN"         : KNNAcc,
    "GPC"         : GPCAcc
}
Performance = pd.DataFrame(performance)
Performance = Performance.round(2)
Performance
```

Out [26]:

	Performance	ANN	SVM	KNN	GPC
train_Acc	Avg Train Accuracy	0.98	0.77	0.87	1.00
test_Acc	Avg Test Accuracy	0.98	0.78	0.76	0.75
time	Avg Run Time	17.35	197.74	0.11	128.21
test_size	Avg Test size	14.00	14.00	14.00	14.00

In [27]:

```
p1 = Performance.iloc[0:3,1:5].T
p2 = pd.DataFrame(columns = p1.columns)
p2 = p2.T
p2["max"] = 0
p2["min"] = 0
p2 = p2.T
for i in range(len(p2.columns)):
    p2.iloc[0,i] = p1.iloc[:,i].max()
    p2.iloc[1,i] = p1.iloc[:,i].min()

# max-min normalizer = 0
# linear normalizer = 1
normalizer = 0

if normalizer == 0 :
    for i in range(len(p1.columns)):
        if i==2: #Non Beneficial
            p1.iloc[:,i] = (p2.iloc[0,i]-p1.iloc[:,i])/(p2.iloc[0,i] - p2.
iloc[1,i])
        else: #Beneficial
            p1.iloc[:,i] = (p1.iloc[:,i]-p2.iloc[1,i])/(p2.iloc[0,i] - p2.
iloc[1,i])

else:
    for i in range(len(p1.columns)):
        if i==2: #Non Beneficial
            p1.iloc[:,i] = p2.iloc[1,i]/p1.iloc[:,i]

        else: #Beneficial
```

```
            p1.iloc[:,i] = p1.iloc[:,i]/ p2.iloc[0,i]

weight = np.array([0.5,0.3,0.2]).reshape(3,1)
p1["p-w"] = np.dot(p1,weight)
p1
```

Out[27]:

	train_Acc	test_Acc	time	p-w
ANN	0.913043	1.000000	0.912766	0.939075
SVM	0.000000	0.130435	0.000000	0.039130
KNN	0.434783	0.043478	1.000000	0.430435
GPC	1.000000	0.000000	0.351819	0.570364

In [28]:

```
p1.round(2)
```

Out [28]:

	train_Acc	test_Acc	time	p-w
ANN	0.91	1.00	0.91	0.94
SVM	0.00	0.13	0.00	0.04
KNN	0.43	0.04	1.00	0.43
GPC	1.00	0.00	0.35	0.57