Complexity of Rule Sets Induced from Incomplete Data with Lost Values and Attribute-Concept Values

By

Priyanka Saha

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas in fulfillment of the requirements for the degree of Master of Science.

Abstract

Data is a very rich source of knowledge and information. However, special techniques need to be implemented in order to extract interesting facts and discover patterns in large data sets. This is achieved using the technique called Data Mining. Data mining is an interdisciplinary subfield of computer science and statistics with an overall goal to extract information from a data set and transform the information into a comprehensible structure for further use. Rule induction is a Data Mining technique in which formal rules are extracted from a set of observations. The rules induced may represent a full scientific model of the data, or merely represent local patterns in the data.

The data sets, however, is not always complete and might contain missing values. Data mining also provides techniques to handle the missing values in a data set. In this project, we've implemented lost value and attribute-concept value interpretations of incomplete data. Experiments were conducted on 176 datasets using three types of approximations (lower, middle and upper) of the concept and Modified Learning from Examples Module, version 2 (MLEM2) rule induction algorithm was used to induce rule sets.

The goal of the project was to prove that the complexity of rule sets derived from datasets having missing attributes is better for attribute-concept value interpretation compared to the lost value interpretation. The size of the rule set was always smaller for the attribute-concept value interpretation. Also, as a secondary objective, we tried to explore what type of approximation provides the smallest size of the rule sets.

Acknowledgement

I would like to express my special thanks of gratitude to my adviser Dr. Grzymala-Busse, Jerzy W for providing me useful resources, valuable remarks and engagement throughout the course of this project.

Furthermore, I would like to thank Dr. Kim and Dr. Zhong for their enthusiasm and encouragement on the project.

I am also thankful to EECS department for providing excellent infrastructure, resources and opportunity for completing this project successfully.

Finally, I would like to take this opportunity to thank all those who provided direct or indirect support in completion of this project.

Table of Contents

1. Introduction

Data mining is the process of finding anomalies, patterns and correlations within large data sets to predict outcomes. Using a broad range of techniques, this information can be used to increase revenues, cut cost, improve customer relationships, reduce risks and many more. Data mining can help sifting through the noisy data and help extracting the relevant useful information. Data mining is at the heart of analytics efforts across a variety of industries and disciplines e.g. multimedia and telecommunication companies, insurance companies, education, manufacturing, banking, retail industries etc. The application varies in different industries while the underlying concepts being the same.

Rule induction is a very important technique in the field of data mining or machine learning. Rules are usually extracted from a set of observations. In this project, we've followed supervised learning approach where all of the input cases are pre-classified by an expert. The example below explains the required concepts:

Case	Wind	Humidity	Temperature	Trip
1	low	low	medium	yes
2	medium	high	low	no

The table presents two cases which are the set of observations. Wind, Humidity and Temperature are the attributes whose values are to be used to determine whether the Trip is possible or not – Trip is called the decision. Therefore, a set of attributes can collectively determine a decision. The above observations can be utilized to form a rule e.g. (Wind, low) & (Humidity, low) -> (Trip, yes). As it's shown, the rules are usually of the form (attribute, value) $\rightarrow$ (decision, value). Now, LEM2 algorithm is a popular approach to induce rules from a dataset having symbolic values like the one in the above example. However, there is different approaches of rule induction for a dataset having numeric values. MLEM2 algorithm is one of the widely popular approaches and used in this project. MLEM2 essentially converts the numeric attribute values into symbolic by using cut point approach. Afterwards, the obtained intervals are used in rule induction. The details of MLEM2 can be found under references. We've explained the difference with the help of below example:

Case	Temperature	Humidity	Snow
1	15	low	High
2	27	low	High
3	40	high	Low
4	27	high	High

Case	Temperature	Humidity	Snow
1	15..21	low	High
2	21..33.5	low	High
3	33.5..40	high	Low
4	21..33.5	high	High

In the above example, the attribute Temperature is numeric and hence it was represented as intervals based on the cut points 21 and 33.5. This conversion process is known as discretization.

In this project, we've also dealt with incomplete data i.e. the datasets having missing attribute values. Practically, in real life, it's very hard to obtain a data set with all the values specified. Hence, we must focus on techniques to handle the missing attributes. Furthermore, sometimes various approximation methods if used, provide a better result in terms of rule complexity. Usually this is a good technique when there is inconsistency in the data set. We've used two missing value interpretation namely attribute-concept value and lost value for each of the lower, middle and upper approximations of the concept in the data set. We'll discuss these in the following sections. Our primary objective in this project is to show that the attribute-concept value interpretation always yields smaller rule sets. Additionally, we'll also check which of the three types of approximations is the best from the point of view of rule complexity.

2. Incomplete Data and Approximations

In this section, we're going to discuss different techniques of handling missing values in a data set and converting it into a complete data set in order to perform the main processes e.g. rule induction in our case. There are a number of ways to do it - deleting cases with missing attribute values, replace the missing data with most common value of an attribute, assigning all possible attribute values, replacing by attribute mean etc. The details of all such methods can be found under reference. However, in this project we've considered attribute-concept values and lost values as discussed below.

Case	Education	Skills	Experience	Productivity
1	higher	high	-	high
2	?	high	low	high
3	secondary	-	high	high
4	higher	?	high	high
5	elementary	high	low	low
6	secondary	-	high	low
7	-	low	high	low
8	elementary	?	-	low

Table 1: Decision table

A set of example observations have been presented in form of a decision table (Table 1). We'll discuss a few important terms in order to explain the missing value interpretation and approximations.

a) The set of all cases (*universe*): *U = {1, 2, 3, 4, 5, 6, 7, 8}*
b) Independent variables are called *attributes*, the set of all attributes: *A = {Education, Skills, Experience, Productivity}*
c) Dependent variable is called a *decision* denoted by *d*: *Productivity*
d) The value for a case x and an attribute a is denoted by *a(x)*: *Skills(high)*
e) Lost values are denoted by *"?"* and means that the original attribute value is no longer accessible and only existing values will be used during rule induction.
f) Attribute-concept values are denoted by *"-"* and means that the original attribute value is unknown, however since we know the concept to which a case belongs, we know all possible attribute values.
g) The set X of all cases defined by the same value of decision d is called a *concept*, e.g. a concept associated with the value *low* of our decision *Productivity* is set {1, 2, 3, 4}
h) For an attribute *a* and its value *v*, *(a, v)* is called an attribute-value pair and a block of (a, v) is denoted by *[(a, v)]*. For incomplete data, we form the attribute-value pair in the following way:
    - If a(x) = "?", i.e. corresponding value is lost, the case x shouldn't be included in any blocks [(a, v)] for all values *v* of attribute *a*.
    - If a(x) = "-", i.e. corresponding value is unknown, the case x should be included in blocks [(a, v)] for all specified values $v \in V(x, a)$ of attribute a.

Computation of attribute-value pairs: As we've presented the idea of all the required terms and defined them, we'll discuss with the examples of how the attribute-value pairs have been computed for this project. From Table 1, the attribute-concept values are defined as: V(1, Experience) = {low, high}, V (3, Skills) = {high}, V (6, Skills) = {low, high}, V (7, Education) = {elementary, secondary} and V(8, Experience) = {low, high}. Now, accordingly the blocks of attribute-value pairs will be:

- [(Education, elementary)] = {5, 7, 8}
- [(Education, secondary)] = {3, 6, 7}
- [(Education, higher)] = {1, 4}
- [(Skills, low)] = {6, 7}
- [(Skills, high)] = {1, 2, 3, 5, 6}
- [(Experience, low)] = {1, 2, 5, 8}
- [(Experience, high)] = {1, 3, 4, 6, 7, 8}

Hence, it clearly shows that the missing value interpretation needs to be finalized and built before we can proceed with the formation of attribute-value blocks. Our next step is to define and compute the different types of approximations.

Characteristic Sets: If A is the set of attributes, U is the set of all cases and any case $x \in U$, then the characteristic set K(x) is defined as the intersection of the sets K(x, a) where the set K(x, a) is defined in the following way:

- If a(x) is specified, then K(x, a) is the block [(a, a(x))] of attribute a and its value a(x)
- If a(x) = "?" then the set K(x, a) = U, where U is the set of all cases
- If a(x) = "-" then the corresponding set K(x, a) is the union of all blocks of attribute-value pairs (a, v), where $v \in V(x, a)$ if V(x, a) is nonempty. If V (x, a) is empty, K(x, a) = U.

For Table 1, K(1) = [(Education, higher)] $\cap$ [(Skills, high)] $\cap$ [(Experience, low) $\cup$ (Experience, high)] = {1, 4} $\cap$ {1, 2, 3, 5, 6} $\cap$ {1, 2, 3, 4, 5, 6, 7, 8} = {1}. Similarly, K(2) = {1, 2, 5}, K(3) = {3, 6}, K(4) = {1, 4}, K(5) = {5}, K(6) = {3, 6, 7}, K(7) = {6, 7}, and K(8) = {5, 7, 8}.

For incomplete data, there are a number of possible ways to define the approximations, however we've used *concept approximations* since it was proved to be more efficient in past research projects. More details can be found under reference.

Computing Approximations: A *concept-lower approximation* of the concept X is defined as: $\underline{A}X = \cup\{K(x) \mid x \in X, K(x) \subseteq X\}$ and *concept-upper approximation is defined as:* $\bar{A}X = \cup\{K(x) \mid x \in X\}$ where A is the set of all attributes, K(x) is the characteristic set. To illustrate this with an example, we'll consider Table 1 and the characteristic sets computed in the last section. A-concept lower and A-concept upper approximations of the concept {1, 2, 3, 4} are respectively:

- $\underline{A}\{1, 2, 3, 4\}=K(1) \cup K(4)=\{1\} \cup \{1, 4\}=\{1, 4\}$
- $\bar{A}\{1, 2, 3, 4\}=K(1) \cup K(2) \cup K(3) \cup K(4)=\{1\} \cup \{1, 2, 5\} \cup \{3, 6\} \cup \{1, 4\}=\{1, 2, 3, 4, 5, 6\}$

However, following this approach we wouldn't be able to define the *middle approximation* and hence, we've followed *Probabilistic Approxim*ation method to derive the same.

Probabilistic Approximations: For incomplete data sets, a *concept-probabilistic approximation* is defined mathematically as: $\cup\{K(x) \mid x \in X, Pr(X|K(x)) >= \alpha\}$ where α is a parameter, $0 <= \alpha <= 1$. Details on this definition and how it is related to the value precision asymmetric rough sets can be found under reference, however we're not using the information for this project. The three types of interpretation on α are as below:

- if α = 1, the probabilistic approximation is essentially the standard lower approximation
- if α is small and close to 0, this describes the standard upper approximation
- if α = 0.5, this is what is essentially the middle approximation of the concept

We'll illustrate this with the example below considering the details taken from Table 1:

K(x)	**Pr({1, 2, 3, 4} | K(x))**
{1}	|{1}∩ {1, 2, 3, 4}| / |{1}| = 1
{1, 2, 5}	|{1, 2, 5}∩ {1, 2, 3, 4}| / |{1, 2, 5}| = 0.67
{3, 6}	|{3, 6}∩ {1, 2, 3, 4}| / |{3, 6}| = 0.5
{1, 4}	|{1, 4}∩ {1, 2, 3, 4}| / |{1, 4}| = 1
{5}	|{5}∩ {1, 2, 3, 4}| / |{5}| = 0
{3, 6, 7}	|{3, 6, 7}∩ {1, 2, 3, 4}| / |{3, 6, 7}| = 0.33
{6, 7}	|{6, 7}∩ {1, 2, 3, 4}| / |{6, 7}| = 0
{5, 7, 8}	|{5, 7, 8}∩ {1, 2, 3, 4}| / |{5, 7, 8}| = 0

Table 2: Conditional Probabilities

We've computed and presented the conditional probabilities of the concept X={1, 2, 3, 4} given all of the characteristic sets in Table 2. Now as from the definition of *middle approximation*, we need to take the union of all characteristic sets where condition probability is >= 0.5. Therefore, *concept-middle probabilistic approximation* of X would be {1} ∪ {1, 2, 5} ∪ {3, 6} ∪ {1, 4} = {1, 2, 3, 4, 5, 6}.

Also, using the general definition we can compute *concept-lower approximation* (where α = 1) to be {1} ∪ {1, 4} = {1, 4} and *concept-upper approximation* (where α = 0) to be {1} ∪ {1, 2, 5} ∪ {3, 6} ∪ {1, 4} = {1, 2, 3, 4, 5, 6} which are exactly the same as computed in the previous section.

Therefore, in this project, we've followed the probabilistic approximation calculation algorithm to define all three types of approximations namely lower, middle and upper.

3. Modified Learning from Examples Module, version 2 (MLEM2)

The MLEM2 rule induction algorithm is a modified version of LEM2 that deals with the numeric data with core algorithm being the same. MLEM2 can be used for any approximation of the concept and for the data set having missing attributes. Therefore, we've used MLEM2 for rule induction in this project since we dealt with numerical data, missing attributes as well as three types of approximations discussed in the previous sections. Three of the data sets used in this project contained only numerical attributes including Iris, Bankruptcy, Echocardiogram.

The main idea behind MLEM2 is that the rule sets derived are in the LERS format i.e. Learning from Examples based on Rough Sets where every rule is equipped with three numbers – the total number of attribute-value pairs on the left-hand side of the rule, the total number of examples correctly classified by the rule during training and the total number of training cases matching the left-hand side of the rule. The actual algorithm LEM2 can be found under reference section. However, we'll discuss in this section, how Modified LEM2 works and was used in this project with the help of a sample dataset given in Table 3.

Case	Wind	Humidity	Temperature	Trip
1	4	low	medium	yes
2	8	low	low	yes
3	4	medium	medium	yes
4	8	medium	high	maybe
5	12	low	medium	maybe
6	16	high	low	no
7	30	high	high	no
8	12	high	high	no

Table 3: A dataset with numerical attributes

In rule induction from numerical data, usually a preliminary step called discretization is conducted. In this approach, a domain on the numerical attribute is divided into intervals defined by cutpoints (left and right delimiters of intervals) e.g. an interval delimited by two cutpoints c, d will be denoted as c..d. It is therefore important to discuss the process of discretization used in this project.

We've different discretization approaches in practice, however MLEM2 doesn't need any preliminary discretization of numerical attributes. The domain of any numerical attribute is sorted first, then potential cutpoints are selected as averages of any two consecutive values of the sorted list. For each cutpoint c, the MLEM2 algorithm creates two blocks, the first block contains cases for which values of the numerical attributes are smaller than c and the second block consists of rest of the cases. Afterwards, the rule induction is done in the same way as that of LEM2. In this project, we've applied MLEM2 algorithm in three different types of approximation. However, while calculating an approximation following the procedure mentioned in the previous section, the numerical attributes are treated merely as symbolic attributes and corresponding attribute-value pair is formed and we don't use the cutpoint approach in this case e.g. while calculating approximations an example of a case would be (Wind, 4) and not an interval defined by cutpoint. Hence, we've two different stages of handling numerical attributes.

For Table 3, we'll discuss the MLEM2 discretization and rule induction process for concept (Trip, yes). Hence, our goal would be G = {1, 2, 3}. The cutpoints for the domain "Wind" are clearly 6, 10, 14, 23. We wouldn't need to discretize rest of the attribute values since they're not numerical and hence treated as symbolic values. The intervals and steps of the algorithm has been recorded in the table below:

(a, v) = t	[(a, v)]	{1, 2, 3}	{1, 2, 3}	{2}	{2}
(Wind, 4..6)	{1, 3}	{1, 3}	{1, 3} ●	-	-
(Wind, 6..30)	{2, 4, 5, 6, 7, 8}	{2}	{2}	{2}	{2}
(Wind, 4..10)	{1, 2, 3, 4}	{1, 2, 3} ●	-	{2}	{2}
(Wind, 10..30)	{5, 6, 7, 8}	-	-	-	-
(Wind, 4..14)	{1, 2, 3, 4, 5, 8}	{1, 2, 3}	-	{2}	{2}
(Wind, 14..30)	{6, 7}	-	-	-	-
(Wind, 4..23)	{1, 2, 3, 4, 5, 6, 8}	{1, 2, 3}	-	{2}	{2}
(Wind, 23..30)	{7}	-	-	-	-
(Humidity, low)	{1, 2, 5}	{1, 2}	{1, 2}	{2}	{2} ●
(Humidity, medium)	{3, 4}	{3}	{3}	-	-
(Humidity, high)	{6, 7, 8}	-	-	-	-
(Temperature, low)	{2, 6}	{2}	{1, 3}	{2} ●	-
(Temperature, medium)	{1, 3, 5}	{1, 3}	{1, 3}	-	-
(Temperature, high)	{4, 7, 8}	-	-	-	-
Comments		1	2	3	4

Table 4: MLEM2 Algorithm recorded steps for [(Trip, yes)]

Rule Induction: The MLEM2 algorithm is traced on Table 4 above. The corresponding comments will explain the approach:

1. In this step, we've the primary goal as G {1, 2, 3}, best attribute-value pair with the largest cardinality of the intersection of [t] and G and smallest cardinality of [t] is (Wind, 4..10) whose corresponding entry in third column has been marked. But, [(Wind, 4..10)] = {1,2,3,4} $\not\subset$ {1, 2, 3} and hence we move forward,

2. The goal G remains the same. However, we would discard and hence put dashes for rows (Wind, 4..14) and (Wind, 4..23) since the corresponding intervals include 4..10. In this step, the best attribute-value pair would be (Wind, 4..6) covering the max goal and having least number of elements in its block. Now, {1, 2, 3, 4} $\cap$ {1, 3} = {1, 3} $\subset$ {1, 2, 3}. Obviously, the common part of both these selected intervals are 4..6, hence {(Wind, 4..6)} is the first element of our condition set T,

3. Since, we covered a part of our original goal, the newly set goal would be {1, 2, 3} – {1, 3} = {2}. The pair [(Temperature, low)] has the smallest cardinality of [t] and covering max goal, hence it is chosen. But, [(Temperature, low)] = {2, 6} $\not\subset$ {1, 2, 3}, so we move forward,

4. In this step, the best pair is (Humidity, low) and {2, 6} $\cap$ {1, 2, 5} = {2} $\subset$ {1, 2, 3}. This is not numeric and hence , there is no need to combine interval. Also, we can't drop any condition here to make it a subset. Hence {[(Temperature, low), (Humidity, low)]} is the second element of T.

Now, T = { {(Wind, 4..6)}, {(Temperature, low), (Humidity, low))} } and the induced rules would be:

- (Wind, 4..6) $\rightarrow$ (Trip, yes),
- (Temperature, low) & (Humidity, low) $\rightarrow$ (Trip, yes)

Applying the same steps of this algorithm, we can derive rule sets for the rest of the concepts as well. Also, as an additional note, in case of using the algorithm on concept approximations, the rules will be computed in the same way except for treating the non-computing decision as SPECIAL e.g. if we would've used any approximation of the concept (Trip, yes) in the above example, then any decision value other than the approximated cases would be consider as (Trip, SPECIAL) and use that to derive a separate rule set. In this project, we've used this approach since we used all three types of approximation for all of the test data sets.

Hence, combining all the information and steps discussed in the previous section for handling missing attributes, computing concept probabilistic approximations and MLEM2 algorithm in this section would produce the final rule set as result which is the potential input for our rule complexity model implemented as part of this project. This is what we wanted to measure essentially. We'll discuss in the next section how the complexity of the rulesets has been computed to generate the required results in this project.

4. Rule Complexity

Rule complexity in this project has essentially been measured in terms of number of rules and number of conditions. For example, if a rule set contains the below rules (taken from the example in previous section and added the rule sets for other two concepts) :

- (Wind, 4..6) → (Trip, yes),
- (Temperature, low) & (Humidity, low) → (Trip, yes)
- (Wind, 6..30) & (Humidity, medium) → (Trip, maybe)
- (Wind, 10..30) & (Temperature, medium) → (Trip, maybe)
- (Humidity, high) → (Trip, no)

Here, number of rules = 5 and total number of conditions = 8. These are the two parameters used to measure rule complexity in this project across all eight datasets and templates having missing values. The implementation was done writing a python script to read the corresponding rule files and then extracting the rule and condition counts in order to be used for computing complexity for each approximation in each dataset templates. In the end, the comparison was displayed as line graph as results. More details on the results will be discussed in the *Experiments* section of this document.

5. Implementation

5.1 Computer Platform

Any standard machine configuration with 8GB of RAM, 64-bit processor and 2GHz CPU with Windows/Linux/MacOS platform would be sufficient to run the source code. The machine should also support Python 3 and be able to run code written in Python 3.

5.2 Programming Language

The algorithms were implemented using the high level interpreted general purpose programming language Python. The Python version 3 was used in this project. Python emphasizes code readability and notably using significant whitespaces. Python also supports automatic memory management, multiple programming paradigms and contains a comprehensive standard library. However, in this project we didn't use any specific data science or sci-kit-learn libraries. All the algorithms were developed using basic Pandas and NumPy framework of python and thorough implementation logic was used. In order to ensure the source code in this project runs smooth, it is advised to have the Python installed with Anaconda distribution which includes all the required framework. However, if using without Anaconda Python distribution, the following frameworks need to be installed for Python 3:

- Pandas
- NumPy
- Itertools
- Matplotlib
- Scipy

A few interesting features of Python that helped with the smooth implementation of this project are:

Interpreted language: Python is an interpreted language i.e. interpreter executes the code line by line at a time. This makes debugging very easy and can be tested on platform like *JuPyter Notebook*.

Cross-platform language: Python can run equally on different platforms such as Windows, Linux, Unix, Macintosh etc. which makes it portable and reduce platform compatibility concerns for the programmer.

Integrated: It can be easily integrated with languages like C, C++, JAVA etc.

Performance: Being an interpreted language, python is comparatively slower than different high-level language like JAVA, C++ etc. However, the speed optimization is noticeable when using in larger scale. Hence for this project, we didn't find much of a performance drawback in terms of running algorithms or generating results. Ultimately it depends on the type of project and what kind of tradeoff would be most profitable in certain scenarios.

Memory management: Python has automatic memory management scheme. Python memory manager uses garbage collection process to periodically check for any object no longer referenced by the program and free the corresponding memory.

5.3 Data Structures

In this project, we aimed to implement and compare the results of rule complexity of eight data sets for two different types of missing value interpretations and three types of concept probabilistic approximations. Hence, on a closer look, the project can be considered to be a complex problem that can be subdivided into different problems i.e. handling missing attributes, calculating approximations, developing MLEM2 algorithm, calculate the rule complexities, generate the graph/results. Often times, in practice, that's how we solve a complex problem by dividing into sub problems and eventually reaching to the final solution. In other words, we divide the tasks into separate modules for convenience of achieving goal with minimized effort and code readability. The different data mining algorithms used in this project are complex, exploration intensive and goal oriented where the time complexity would spike up with the size of input dataset. In real world, we deal with even larger amount of data. Hence, it was one of the goals of this project to implement the algorithms with more efficiency, reduced space and time complexity.

Python provides a good range of data structures for easy and convenient handling of data processing. Pandas, NumPy are some of the widely popular data structure frameworks used in this project. We can implement arrays, data-frames, linked list, dictionary etc. using the implicit python libraries. Some of the data structures used in this project are:

- *Dictionary:* Python dictionary is an unordered collection of data values used to store data values like a map and hence holds key::value pair. Key-value is provided in the dictionary to make it more optimized. Keys of a dictionary must be unique and of immutable data type such as strings, integers and tuples, but the key-value can be repeated and be of any type. In python, dictionary can be created in multiple ways, however we've used curly braces convention in this project. The dictionary was primarily used to hold the characteristic sets in case of our project.

- *Pandas Data-frame:* Python Pandas DataFrame is two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes. In real world, a Pandas dataframe is usually created by the datasets from existing storage e.g. SQL Database, CSV file, Excel file etc. We've heavily used Pandas dataframe in this project while the initial input was read from CSV files. This framework makes it very convenient to manipulate data on row, column and perform various operations on them.

- *NumPy Array:* NumPy is a general-purpose array processing package. It provides a high-performance multidimensional array object and tools for working with them. Array in NumPy is a table of elements indexed by a tuple of positive integers. In NumPy, number of dimensions of the array is called the rank and a tuple of integers giving the size of the array along each dimension is known as shape. We've used NumPy array for processing a couple of specific rows on a dataset.

- *List:* Lists are basically like the arrays declared in other languages. Lists need not be homogeneous always which makes it a most powerful tool in Python. A single List may contain data types like integers, strings as well as objects. Unlike Sets, the List in Python are ordered and have a definite count. In this project, Lists were used extensively for intermediate processing.

- *Set:* Python Set is an unordered collection data type that is iterable, mutable and has no duplicate elements. This is based on a data structure known as Hash Table. The main advantage of using Set over List is that it has a highly optimized method for checking whether a specific element is contained in the Set. Also, we can perform all kind of set operations which proved to be very useful for this project. We've used Set extensively in the implementation of MLEM2 algorithm in this project.

6. Datasets

Our experiments are based on eight datasets taken from University of California at Irvine Machine Learning Repository (UCI Machine Learning Repository) – *Bankruptcy*, *Breast Cancer*, *Echocardiogram*, *Hepatitis*, *Image Segmentation*, *Iris*, *Lymphography* and *Wine Recognition*.

For every dataset, a set of templates was created. Templates were formed by replacing incrementally (with 5% increment) existing specified attribute values by *lost values*. Thus, each series of experiments was started with no missing data, then gradually we added 5% of *lost values* each time until at least one entire row of the data sets was full of *lost values*. Additionally, the same processed templates were edited for further experiments by replacing lost values ("?") by attribute-concept values ("-").

For any dataset there was some maximum for the percentage of missing attribute values e.g. for the Bankruptcy dataset, it was 35%. Therefore, we had 7 lost value and 7 attribute-concept value templates of

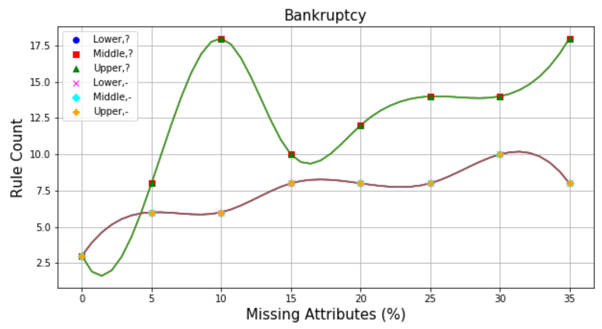
Bankruptcy data set which combined to total 15 datasets including the main dataset without any missing attribute. Following the same approach, the Breast Cancer, Echocardiogram, Hepatitis, Image Segmentation, Iris, Lymphography and Wine Recognition datasets had 19, 17, 29, 15, 29 and 27 data sets respectively. The total number of data sets used in this project was 176.

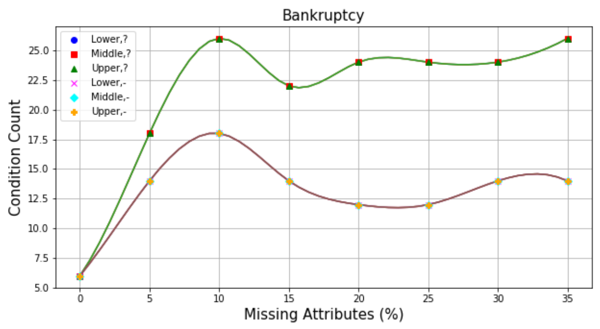
7. Experiment Results

For each series of experiments, we compared two interpretations of missing attribute values – lost and attribute-concept values, assuming the same type of approximations. We compared the complexity of the rule sets – size of the rule sets and total number of conditions separately for lower, middle and upper approximations. For all eight types of datasets and all three types of approximations, the rule set size was always smaller for attribute-concept values than for lost values. However, for total number of conditions in the rules sets, results were more complicated. The total number of conditions was smaller for attribute-concept values than for lost values for 17 combinations of the type of dataset and approximation, out of 24 possible combinations. The detailed results of each dataset have been discussed below:

7.1 Bankruptcy

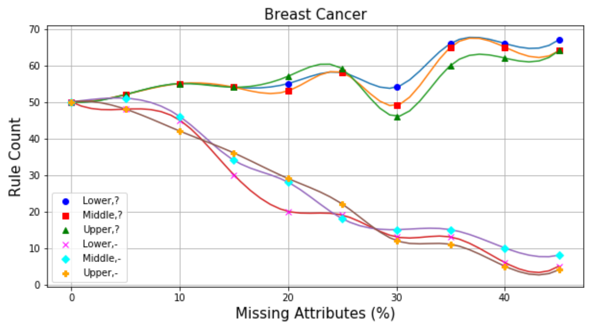
The trend of the rule set is pretty much straight forward in this case. The size of the rule set, and total number of conditions were always smaller for attribute-concept values than lost values. There is essentially no variation on these counts for different types of approximations.

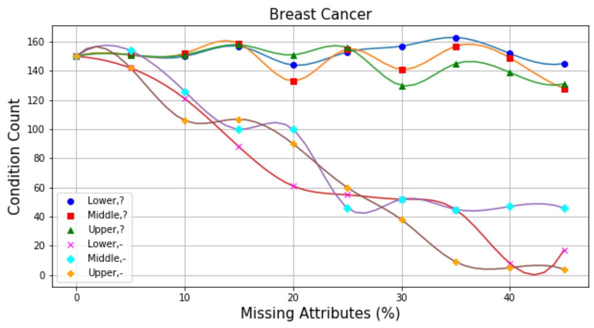




7.2 Breast Cancer

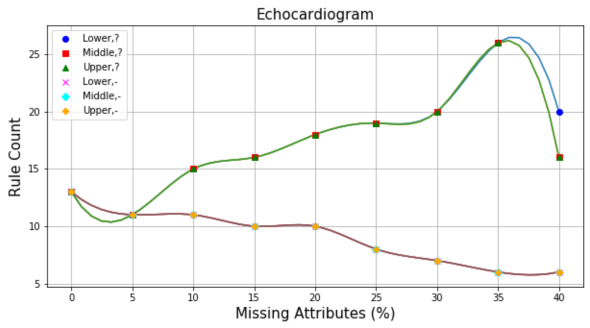
For attribute-concept values, the size of the rule set was smaller for upper approximations than for middle approximations in most of the cases. Thus, upper approximations were better than other approximations here.

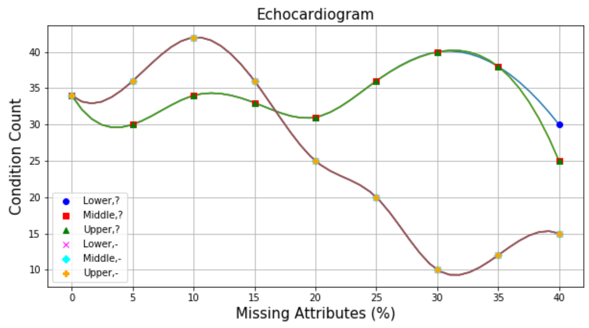




7.3 Echocardiogram

For all three types of approximations, the total number of rules and conditions in rule sets for both interpretations of missing attribute values, did not differ significantly. The general trend shows that the results were better in case of attribute concept values than lost values.

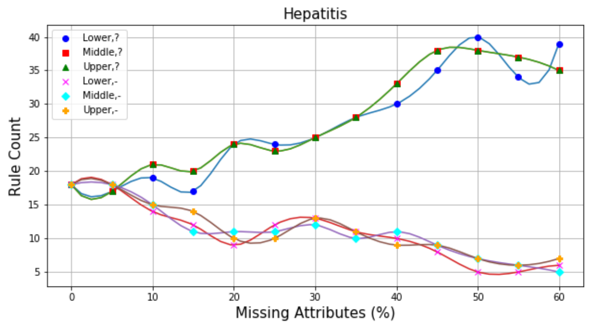


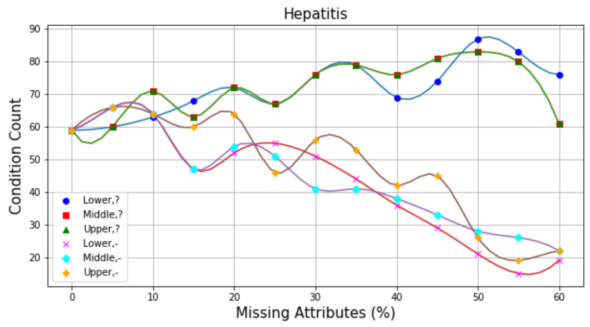


7.4 Hepatitis

For attribute-concept values, the size of the rule set was smaller for lower approximation than for upper approximation in most of the cases if not equal.

For attribute-concept values, total number of conditions in rule sets was smaller for lower approximations than for upper approximations in most of the cases.

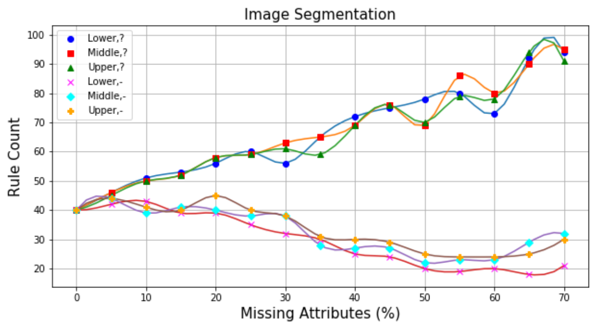


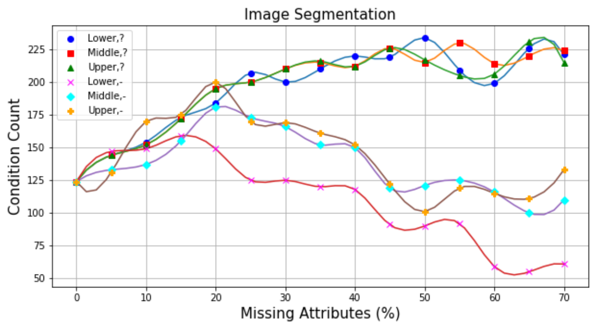


7.5 Image Segmentation

For both lost and attribute-concept values, the size of the rule set was smaller for lower approximation than for middle and upper approximation in most of the cases if not equal. Thus, we can say, lower approximations were better than other approximations.

For both lost and attribute-concept values, total number of conditions in rule sets was smaller for lower approximations than for middle or upper approximations in most of the cases.



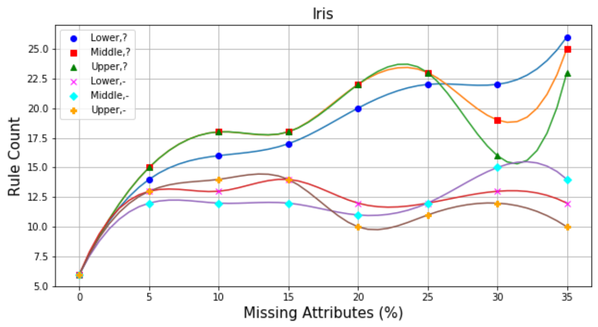


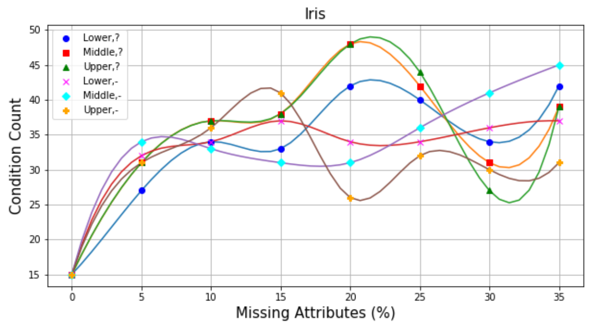
7.6 Iris

The size of rule set was always smaller for attribute-concept values.

For all three types of approximations, the total number of conditions in rule sets for both interpretations of missing attribute values, did not differ significantly.

However, for lost values, total number of conditions in rule sets was smaller for lower approximations than for upper approximations in most of the cases.



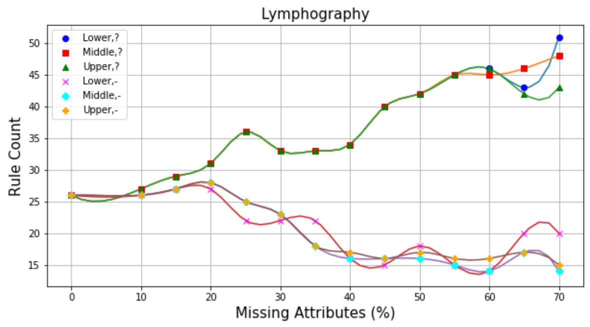


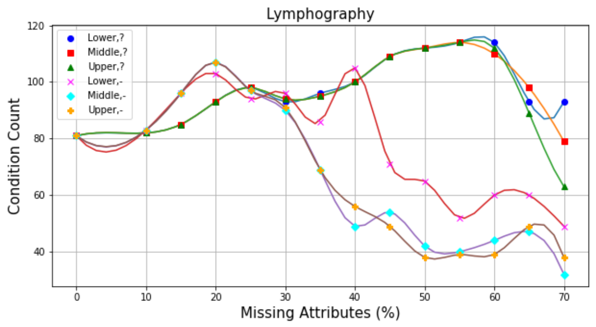
7.7 Lymphography

The size of the rule set was smaller for upper approximations than for lower approximations in most of the cases of attribute-concept values. Thus, upper approximations were better than other approximations here.

For lower approximation, the total number of conditions in rule sets for both interpretations of missing attribute values, did not differ significantly.

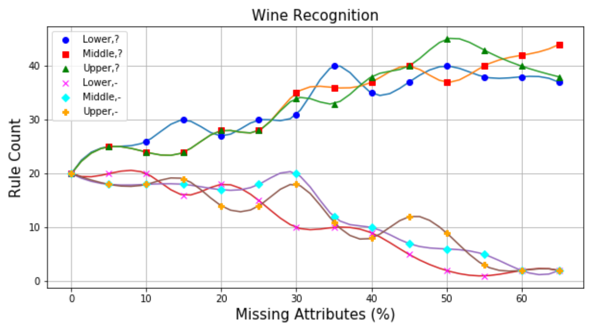
For attribute-concept values, total number of conditions in rule sets was smaller for middle and upper approximations than for lower approximations.

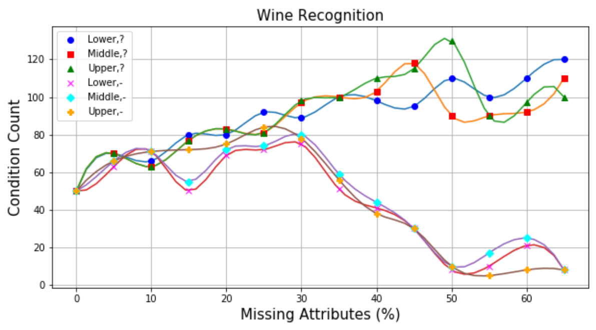




7.8 Wine Recognition

The usual trend both for total number of rules and total number of conditions show that the results are better for attribute-concept values than lost values. There is not much significant difference in results between the approximations.





8. Conclusion

On summarizing all of the above results, it is clear that the size of the rule set was always smaller for attribute-concept values than for lost values. The total number of conditions in the rule sets was smaller for attribute-concept values for 17 combinations out of 24 and didn’t vary significantly for rest of the 7 combinations. Thus, we can conclude that attribute-concept values are better than lost values in terms of rule complexity.

On analyzing the quality of approximation, the results were more complicated. The individual results outlined in the previous section shows that out of total 24 combinations, lower approximations were better than other approximations in 5 combinations, whereas for 2 combinations, upper approximations were better. The difference between all three approximations was insignificant for rest of the 17 combinations.

9. References

[1] Jerzy W. Grzymala-Busse & G. Clark, Patrick, Complexity of Rule Sets Induced from Incomplete Data with Attribute-concept Values and "Do Not Care" Conditions. DATA 2014 - Proceedings of 3rd International Conference on Data Management Technologies and Applications. 56-63. 10.5220/0005003400560063.

[2] Grzymala-Busse J.W., Grzymala-Busse W.J. (2005) Handling Missing Attribute Values. In: Maimon O., Rokach L. (eds) Data Mining and Knowledge Discovery Handbook. Springer, Boston, MA

[3] Grzymala-Busse, Jerzy W. "Rule Induction from Rough Approximations." *Handbook of Computational Intelligence* (2015).

[4] J. W. Grzymala-Busse, "Rough set strategies to data with missing attribute values," in Workshop Notes, Foundations and New Directions of Data Mining, in conjunction with the 3-rd International Conference on Data Mining, 2003, pp. 56–63.

[5] ——, "Generalized parameterized approximations," in Proceedings of the RSKT 2011, the 6-th International Conference on Rough Sets and Knowledge Technology, 2011, pp. 136–145.

[6] Grzymala-Busse, Jerzy W. "Rule Induction." *The Data Mining and Knowledge Discovery Handbook* (2005).

[7] https://docs.python.org/3/using/index.html

[8] Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

APPENDIX A

A.1 Lost-Value.py

```python
#-------Importing Libraries-------#
import time
import pandas as pd
import numpy as np
import pylab as pl
from pandas.api.types import is_numeric_dtype
from functools import reduce
from itertools import groupby
from collections import OrderedDict

#-------Definition of all the Functions-------#

#-------Discretization-------#
def discretize(numeric_col):
    #Sorting the values of numeric column

    list1 = df[numeric_col].tolist()
    list1 = [ elem for elem in list1 if elem != '?']
    list1 = [float(x) for x in list1]
    sort_col = sorted(list1)

    point_list = set(sort_col)
    point_list = sorted(list(point_list))
    print(point_list)

    #Finding average between each two points
    avg_list = []
    for i in range(len(point_list)-1):
        avg = (point_list[i] + point_list[i+1])/2
        avg_list.append(round(float(avg),3))
    print(avg_list)

    #Performing the discretization and adding the cases
    for i in avg_list:
        case = str(numeric_col) + "," + str(point_list[0]) + ".." + str(i)
        case2 = str(numeric_col) + "," + str(i) + ".." + str(point_list[-1])

        case_list.append(case)
        case_list.append(case2)

#-------Computing Lower Approximation-------#
def lowerApproximation(charac_set,concept):
    #set to contain lower approximations
    lower = set()

    #Check for each element of the concept
    for item in concept:
        key = 'K_%d' % (item)
        set_value = charac_set[key]

        if set_value.issubset(set(concept)):
            lower = lower.union(set_value)

    return lower

#-------Computing Upper Approximation-------#
def upperApproximation(charac_set,concept):
    #set to contain lower approximations
    upper = set()

    #Check for each element of the concept
    for item in concept:
        key = 'K_%d' % (item)
        set_value = charac_set[key]
        upper = upper.union(set_value)
    return upper

#-------Computing Concept Probabilistic Approximation-------#
def probabilisticApproximation(concept):
    prob = []
    for index, row in prob_approx.iterrows():
        if len(row['charset_value']) != 0:
            probability_conditional = len(row['charset_value'].intersection(set(concept))) /
len(row['charset_value'])
        else:
            probability_conditional = 0.0

        prob.append(round(probability_conditional,2))
    return prob

#-------Computing Middle Approximation-------#
def findMiddleApprox(concept):
    probapprox = set()
    cond_prob = probabilisticApproximation(concept) #Need to put the goal
    prob_approx['cond_probability'] = cond_prob
    for index, row in prob_approx.iterrows():
        part1,part2 = row['charset_name'].split("_")

        if row['cond_probability'] >= 0.50:
            if int(part2) in concept:
                probapprox = probapprox.union(row['charset_value'])
    return probapprox

#-------Finding goal interescts for MLEM2-------#
def findGoalIntersect(goal):
    goalIntersect = []

    for index, row in df3.iterrows():
        #List containing intersection of (a,v) pairs and goal
        goalIntersect.append(set(row['att_val']).intersection(set(goal)))

    #Check if goal_intersect column exists
    if 'goal_intersect' in df3:
        df3['goal_intersect'] = goalIntersect
    else:
        #Insert new column with the recent iteration
        df3.insert(2, 'goal_intersect', goalIntersect)

#-------Updating goal interescts for MLEM2-------#
def updateGoalIntersect(goal):
    for index, row in df3.iterrows():
        if row['goal_intersect'] != set():
            row['goal_intersetct'] = set(row['att_val']).intersection(set(goal))

#-------Finding a case for MLEM2-------#
def findCases(df3):
    case_to_be = []
    #Find the cases with maximum goal coverage
    m = max(df3['goal_intersect'], key=len)
    possible_cases = [i for i, j in enumerate(df3['goal_intersect'].tolist()) if j == m]

    #Index of the case covering max goal and having min no. of elements
    new_df = df3.iloc[possible_cases,:]

    m1 = min(new_df['att_val'], key=len)

    for index,row in new_df.iterrows():
        if row['att_val'] == m1:
            case_to_be.append(index)

    return case_to_be[0]

#-------Combining intervals for MLEM2-------#
def combineInterval(test_condition):
    test_num = [] #This will contain the conditions having interval
    test_str = [] #This will contain the conditions having no interval

    #Loop through to seprate contions having intervals and no intervals
    for item in test_condition:
        if ".." in item:
            test_num.append(item)
        else:
            test_str.append(item)

    #Group the conditions having interval based on same attributes
    grouped = [list(g) for k, g in groupby(test_num, lambda s: s.partition(',')[0])]

    final_list = []

    #Actually combining the intervals
    for list1 in grouped:
        greatest = 0
        smallest = 0
        for item in list1:
            part1,part2 = item.split(",")
            start,stop = part2.split("..")
            start = float(start)
            stop = float(stop)

            if greatest == 0 and smallest == 0:
                greatest = start
                smallest = stop

            if start > greatest:
                greatest = float(start)

            if stop < smallest:
                smallest = float(stop)

        con_tmp = part1 + "," + str(greatest) + ".." + str(smallest)
        final_list.append(con_tmp)

    actual_condition = final_list + test_str

    return actual_condition

#-------Logic to drop condition for MLEM2-------#
def dropCondition(condition,current_goal):

    for item in range(0,len(condition)):
        temp_att_val = []
        temp_cond = condition.copy() #use list.copy() as equal operator simply copies over the
reference
        temp_cond.remove(condition[item])

        #temp_cond contains the elements after removing the current element
        for i in temp_cond:
            if i is not None:
                location = df3.index[df3['Cases'] == i].tolist()
                element = df3['att_val'].loc[location[0]]
                temp_att_val.append(set(element))

        #temp_att_val contains the actual value set of the corresponding cases
        #Find the intersection if the set has more than one element, otherwise no need
        if len(temp_att_val) > 1:
            intersection = set.intersection(*temp_att_val)

            #if the set still remains a subset of the original goal after removing current
element
            #then set the current element to None as we want to drop this later
            if intersection.issubset(current_goal):
                condition[item] = None

    condition = [x for x in condition if x is not None]
    return condition

#-------Actual MLEM2 Algorithm-------#
def stepAlgo(df3,selected_case,current_goal,B,condition,concept_curr):

    rule_set = []
    original_goal = current_goal.copy()


    while current_goal != set():
        #Check if the selected case is a subset of the current goal
        #List of current case
        A = df3['att_val'].loc[selected_case]

        if B == set():
            #Copy over the current set elements to B
            for i in range(len(A)):
                B.add(A[i])

        #Elements of intersection of current and previous set
        A = set(A).intersection(B)
        B = A.copy()

        print("Intersection set: ", A)
        #Check if intersecting elements are subset of Goal
        if A.issubset(original_goal):
            print("SUBSET")
            #Current goal is updated after discarding the already covered goal by new rule
            if len(A) != 0:
                current_goal = set(current_goal) - df3['goal_intersect'].loc[selected_case]
            else:
                current_goal = set()

            print("Current goal: ", current_goal)
            #Extract the current case
            curr_case = df3['Cases'].loc[selected_case]
            #Add the conditions of a Rule
            condition.append(curr_case)

            print("Condition before drop or combine: ", condition)
            #Check for possibility of dropping conditions
            if len(condition) > 1:
                condition = dropCondition(condition,original_goal)

            #Combine the interval
            if len(condition) > 1:
                condition = combineInterval(condition)


            #Join conditions
            cond = ""
            for item in condition:
                cond = cond + "(" + str(item) + ")" + " & "

            cond = cond[:-2] + "->"
            rule = cond + "  (" + concept + "," + concept_curr + ")"
            rule_set.append(rule)

            #Reset everythng and continue for covering rest of the goal
            condition = []
            B = set()
            findGoalIntersect(current_goal)
            selected_case = findCases(df3)

        #If not a subset of current goal
        else:
            print("NOT")
            #Assign empty set for the selected case for next iteration
            df3['goal_intersect'].loc[selected_case] = set()
            print("need to be set to NULL", df3['Cases'].loc[selected_case])
            #Extract the current case
            curr_case = df3['Cases'].loc[selected_case]
            #Add the case to the condition list
            condition.append(curr_case)

            #Check for Range overlapping of the remaining cases
            if ".." in curr_case:
                second_part = curr_case.split(',')[1]
                start = float(second_part.split('..')[0])
                end = float(second_part.split('..')[1])
                for index, row in df3.iterrows():
                    if ".." in row['Cases'] and row['Cases'] == curr_case:
                        part2 = row['Cases'].split(',')[1]
                        start1 = float(part2.split('..')[0])
                        end1 = float(part2.split('..')[1])

                        #Assign blank set for cases with overlapping ranges
                        if set((pl.frange(start,end))).issubset(pl.frange(start1,end1)) == True:
                            if start == start1 and end <= end1:
                                row['goal_intersect'] = set()
                                print("also need to be set to NULL", row['Cases'])

            updateGoalIntersect(A.intersection(current_goal))
            selected_case = findCases(df3)

    return rule_set

#-------Reading Input Dataset-------#
df = pd.read_csv('../data/Iris/Iris-35-lost.csv')

#--------Find the Decision and unique Concepts-------#
df_headers = list(df)
concept = df_headers[-1]
concept_list = df[concept].unique()

#-------Calculating cases by concepts and making list of Goals-------#
#universal list containing all cases
U = []
temp_list = []
goal_list = []
for item in concept_list:
    for index, row in df.iterrows():
        U.append(index+1)
        if row[concept] == item:
            temp_list.append(index)
    goal_list.append(temp_list)
    temp_list = []

#-------Find out all the attributes in the dataset-------#
attributes = list(df)

#-------Find the numeric column-------#
numeric_col = df.select_dtypes(include=[np.number]).columns.tolist()

#-------Build all the cases-------#
case_list = []

#Discretization considering upto 2 decimal point
for item in attributes[:-1]:
        discretize(item)

case_list = list(OrderedDict.fromkeys(case_list))
#-------Calculating cases by concepts and making sets-------#
temp_list = []
goal_list = []
for item in concept_list:
    for index, row in df.iterrows():
        if row[concept] == item:
            temp_list.append(index+1)
    goal_list.append(temp_list)
    temp_list = []
```

```
#-------Calculating blocks for attribute-value pairs-------#
temp_list = []
att_val_list = []
for item in case_list:
    a,b = item.split(",") #a = attribute and b = value

    if ".." in b:
        start,end = b.split("..")
        for index, row in df.iterrows():
            if row[a] is not '?':
                if ".." not in row[a]:
                    if float(row[a]) >= float(start) and float(row[a]) <= float(end):
                        temp_list.append(index+1)
                else:
                    if row[a] == b:
                         temp_list.append(index+1)


        att_val_list.append(temp_list)
        temp_list = []

    else:
        for index, row in df.iterrows():
            if type(row[a]) == list:
                tmp_list = row[a]
                for case in tmp_list:
                    if float(case) == float(b):
                        temp_list.append(index+1)

            if row[a] == b:
                temp_list.append(index+1)

        att_val_list.append(temp_list)
        temp_list = []

#-------Creating data for case and att-value list-------#
data = {'Cases': case_list, 'att_val': att_val_list}
df2 = pd.DataFrame(data)

#-------Here starts the Approximation calculations-------#
attributes = list(df)
U = set(U)

case_list = []
#Loop through all the attributes except last - Concept
for item in attributes[:-1]:
    print(item)

    #check for non numeric columns
    if not is_numeric_dtype(df[item]):
        temp = df[item].unique()
        for i in temp:
            if i == '?' or i == '-':
                continue
            else:
                case = item + "," + i
                case_list.append(case)

temp_list = []
att_val_list = []
for item in case_list:
    a,b = item.split(",") #a = attribute and b = value
    for index, row in df.iterrows():
        if type(row[a]) == list:
            tmp_list = row[a]
            for case in tmp_list:
                if case == b:
                    temp_list.append(index+1)

        if row[a] == b:
            temp_list.append(index+1)

    att_val_list.append(temp_list)
    temp_list = []

#-------reating dictionary combining case_list and att_val list--------#
block = dict(zip(case_list, att_val_list))

#-------Building Characteristic Sets-------#
dic = {}
for index, row in df.iterrows():
    tmp_set = set()
    final_union = []
    char_list = []
    char_list_2 = []
    final_union_set = []

    for cols in attributes[:-1]:
        #If the value for corresponding attribute is a list then create all of the att-value
pairs
        if type(df.loc[index,cols]) == list:
            print("When values are list")
            for item in df.loc[index,cols]:
                block_key = cols + "," + item
                char_list.append(block_key) #char_list has all att-val cases
                print(char_list)

            union_set = set()
            #Compute union of att-concept value case
            for item in char_list:
                union_set = union_set.union(set(block[item]))

            print("Union Set: ", union_set)
            final_union.append(union_set)

        else:
            print("When value is single")
            block_key = cols + "," + str(df.loc[index,cols])
            char_list_2.append(block_key) #char_list_2 has all single cases
            print(char_list_2)

    #Compute instersection for this current row for Characteristics set

    print("final_union: ", final_union)
    if len(final_union):
        final_union_set = list(reduce(set.intersection, [set(item) for item in final_union]))


    print("Final Union Set: ", final_union_set)

    for item in char_list_2:
        if item in block:
            print("When item is found as key in the block")
            if tmp_set == set():
                #Copy over the current set elements to B
                for i in range(len(block[item])):
                    tmp_set.add(block[item][i])

            tmp_set = tmp_set.intersection(set(block[item]))
            print(tmp_set)

        #If item not in block
        else:
            print("When item is not found as key in the block")
            print(tmp_set)
            print(U)
            if tmp_set == set():
                tmp_set = U

            tmp_set = tmp_set.intersection(U)
            print(tmp_set)
    print("Final tmp_set: ", tmp_set)
    final_union_set = set(final_union_set)

    if final_union_set == set():
        tmp_set = tmp_set
    else:
        tmp_set = tmp_set.intersection(final_union_set)

    print("This is the final value: ", tmp_set)

    key = ('K_%d' % (index+1))
    print(key)
    dic[key] = tmp_set
    print("\n")

#-------Form the Approximations-------#
lower_approximations = {}
for item in goal_list:
    #Key is the string converted list so as to add as dictionary key
    lower_approximations[str(item)] = lowerApproximation(dic,item)

lower_goal_list=list(lower_approximations.values())

upper_approximations = {}
for item in goal_list:
    #Key is the string converted list so as to add as dictionary key
    upper_approximations[str(item)] = upperApproximation(dic,item)

upper_goal_list=list(upper_approximations.values())

first_column = list(dic.keys())
second_column = list(dic.values())
prob_approx = pd.DataFrame(
    {'charset_name': first_column,
     'charset_value': second_column
    })

middle_approximations = {}
for item in goal_list:
    #Key is the string converted list so as to add as dictionary key
    middle_approximations[str(item)] = findMiddleApprox(item)

middle_goal_list=list(middle_approximations.values())

df3=df2

#concept_list and goal_list has 1:1 mapping
final_rules = []
start_time = time.time()

#-------Running algorithm for all the goals - Middle Approximation/concepts-------#
for i in range(0,len(middle_goal_list)):
    SPECIAL = U - middle_goal_list[i]
    findGoalIntersect(list(middle_goal_list[i]))

    condition = []
    B = set()
    selected_case = findCases(df3)

    rule_set = stepAlgo(df3,selected_case,middle_goal_list[i],B,condition,concept_list[i])

    #Now do for the SPECIAL cases
    findGoalIntersect(list(SPECIAL))
    condition_SPECIAL = []
    B_SPECIAL = set()
    selected_case_SPECIAL = findCases(df3)
    rule_set2 = stepAlgo(df3,selected_case_SPECIAL,SPECIAL,B_SPECIAL,condition_SPECIAL,"SPECIAL")

    final_rules.append(rule_set)
    final_rules.append(rule_set2)
    print("End of goal")

elapsed_time = time.time() - start_time

print("Time to run the algorithm (for Middle Approximation): ", round(elapsed_time, 3), "Sec")
print("\n")
print(*final_rules, sep='\n')

with open('../data/Iris/Rules/Iris_test_lost_middle.txt', 'w') as f:
    for item in final_rules:
        f.write("%s\n" % item)

#-------Running algorithm for all the goals - Lower Approximation/concepts-------#

#concept_list and goal_list has 1:1 mapping
final_rules = []
start_time = time.time()

#Running algorithm for all the goals - Lower Approximation/concepts
for i in range(0,len(lower_goal_list)):
    SPECIAL = U - lower_goal_list[i]
    findGoalIntersect(list(lower_goal_list[i]))
    condition = []
    B = set()
    selected_case = findCases(df3)

    rule_set = stepAlgo(df3,selected_case,lower_goal_list[i],B,condition,concept_list[i])

    #Now do for the SPECIAL cases
    findGoalIntersect(list(SPECIAL))
    condition_SPECIAL = []
    B_SPECIAL = set()
    selected_case_SPECIAL = findCases(df3)
    rule_set2 = stepAlgo(df3,selected_case_SPECIAL,SPECIAL,B_SPECIAL,condition_SPECIAL,"SPECIAL")

    final_rules.append(rule_set)
    final_rules.append(rule_set2)
    print("End of goal")

elapsed_time = time.time() - start_time


print("Time to run the algorithm (for Lower Approximation): ", round(elapsed_time, 3), "Sec")
print("\n")
print(*final_rules, sep='\n')

with open('../data/Iris/Rules/Iris_test_lost_lower.txt', 'w') as f:
    for item in final_rules:
        f.write("%s\n" % item)

#-------Running algorithm for all the goals - Upper Approximation/concepts-------#

#concept_list and goal_list has 1:1 mapping
final_rules = []
start_time = time.time()

#Running algorithm for all the goals - Lower Approximation/concepts
for i in range(0,len(upper_goal_list)):
    SPECIAL = U - upper_goal_list[i]
    findGoalIntersect(list(upper_goal_list[i]))
    condition = []
    B = set()
    selected_case = findCases(df3)

    rule_set = stepAlgo(df3,selected_case,upper_goal_list[i],B,condition,concept_list[i])

    #Now do for the SPECIAL cases
    findGoalIntersect(list(SPECIAL))
    condition_SPECIAL = []
    B_SPECIAL = set()
    selected_case_SPECIAL = findCases(df3)
    rule_set2 = stepAlgo(df3,selected_case_SPECIAL,SPECIAL,B_SPECIAL,condition_SPECIAL,"SPECIAL")

    final_rules.append(rule_set)
    final_rules.append(rule_set2)
    print("End of goal")

elapsed_time = time.time() - start_time

print("Time to run the algorithm (for Upper Approximation): ", round(elapsed_time, 3), "Sec")
print("\n")
print(*final_rules, sep='\n')

with open('../data/Iris/Rules/Iris_test_lost_upper.txt', 'w') as f:
    for item in final_rules:
        f.write("%s\n" % item)
#-----------------------END-------------------------#
```

A.2 Attribute-Concept.py

```
#-------Importing Libraries-------#
import time
import pandas as pd
import numpy as np
import pylab as pl
from pandas.api.types import is_numeric_dtype
from functools import reduce
from itertools import groupby
from collections import OrderedDict

#-------Definition of all the Functions-------#

#-------Discretization-------#
def discretize(numeric_col):
    #Sorting the values of numeric column

    list1 = df[numeric_col].tolist()
    list1 = [ elem for elem in list1 if elem != '-']
    list1 = [float(x) for x in list1]
    sort_col = sorted(list1)

    point_list = set(sort_col)
    point_list = sorted(list(point_list))
    print(point_list)

    #Finding average between each two points
    avg_list = []
    for i in range(len(point_list)-1):
        avg = (point_list[i] + point_list[i+1])/2
        avg_list.append(round(float(avg),3))
    print(avg_list)

    #Performing the discretization and adding the cases
    for i in avg_list:
        case = str(numeric_col) + "," + str(point_list[0]) + ".." + str(i)
        case2 = str(numeric_col) + "," + str(i) + ".." + str(point_list[-1])

        case_list.append(case)
        case_list.append(case2)

#-------Computing Lower Approximation-------#
def lowerApproximation(charac_set,concept):
    #set to contain lower approximations
    lower = set()

    #Check for each element of the concept
    for item in concept:
        key = 'K_%d' % (item)
        set_value = charac_set[key]
        if set_value.issubset(set(concept)):
            lower = lower.union(set_value)

    return lower

#-------Computing Upper Approximation-------#
def upperApproximation(charac_set,concept):
    #set to contain lower approximations
    upper = set()

    #Check for each element of the concept
    for item in concept:
        key = 'K_%d' % (item)
        set_value = charac_set[key]
        upper = upper.union(set_value)

    return upper

#-------Computing Concept Probabilistic Approximation-------#
def probabilisticApproximation(concept):
    prob = []
    for index, row in prob_approx.iterrows():
        if len(row['charset_value']) != 0:
            probability_conditional = len(row['charset_value'].intersection(set(concept))) /
len(row['charset_value'])
        else:
            probability_conditional = 0.0

        prob.append(round(probability_conditional,2))
    return prob

#-------Computing Middle Approximation-------#
def findMiddleApprox(concept):
    probapprox = set()
    cond_prob = probabilisticApproximation(concept) #Need to put the goal
    prob_approx['cond_probability'] = cond_prob
    for index, row in prob_approx.iterrows():
        part1,part2 = row['charset_name'].split("_")

        if row['cond_probability'] >= 0.50:
            if int(part2) in concept:
                probapprox = probapprox.union(row['charset_value'])
    return probapprox

#-------Finding goal interescts for MLEM2-------#
def findGoalIntersect(goal):
    goalIntersect = []

    for index, row in df3.iterrows():
        #List containing intersection of (a,v) pairs and goal
        goalIntersect.append(set(row['att_val']).intersection(set(goal)))

    #Check if goal_intersect column exists
    if 'goal_intersect' in df3:
        df3['goal_intersect'] = goalIntersect
    else:
        #Insert new column with the recent iteration
        df3.insert(2, 'goal_intersect', goalIntersect)

#-------Updating goal interescts for MLEM2-------#
def updateGoalIntersect(goal):
    for index, row in df3.iterrows():
        if row['goal_intersect'] != set():
            row['goal_intersetct'] = set(row['att_val']).intersection(set(goal))

#-------Finding a case for MLEM2-------#
def findCases(df3):
    case_to_be = []
    #Find the cases with maximum goal coverage
    m = max(df3['goal_intersect'], key=len)
    possible_cases = [i for i, j in enumerate(df3['goal_intersect'].tolist()) if j == m]

    #Index of the case covering max goal and having min no. of elements
    new_df = df3.iloc[possible_cases,:]

    m1 = min(new_df['att_val'], key=len)

    for index,row in new_df.iterrows():
        if row['att_val'] == m1:
            case_to_be.append(index)

    return case_to_be[0]

#-------Combining intervals for MLEM2-------#
def combineInterval(test_condition):

    test_num = [] #This will contain the conditions having interval
    test_str = [] #This will contain the conditions having no interval

    #Loop through to seprate contions having intervals and no intervals
    for item in test_condition:
        if ".." in item:
            test_num.append(item)
        else:
            test_str.append(item)

    #Group the conditions having interval based on same attributes
    grouped = [list(g) for k, g in groupby(test_num, lambda s: s.partition(',')[0])]

    final_list = []

    #Actually combining the intervals
    for list1 in grouped:
        greatest = 0
        smallest = 0
        for item in list1:
            part1,part2 = item.split(",")
            start,stop = part2.split("..")
            start = float(start)
            stop = float(stop)

            if greatest == 0 and smallest == 0:
                greatest = start
                smallest = stop

            if start > greatest:
                greatest = float(start)

            if stop < smallest:
                smallest = float(stop)

        con_tmp = part1 + "," + str(greatest) + ".." + str(smallest)
        final_list.append(con_tmp)

    actual_condition = final_list + test_str

    return actual_condition

#-------Logic to drop condition for MLEM2-------#
def dropCondition(condition,current_goal):

    for item in range(0,len(condition)):
        temp_att_val = []
        temp_cond = condition.copy() #use list.copy() as equal operator simply copies over the
reference
        temp_cond.remove(condition[item])

        #temp_cond contains the elements after removing the current element
        for i in temp_cond:
            if i is not None:
                location = df3.index[df3['Cases'] == i].tolist()
                element = df3['att_val'].loc[location[0]]
                temp_att_val.append(set(element))

        #temp_att_val contains the actual value set of the corresponding cases
        #Find the intersection if the set has more than one element, otherwise no need
        if len(temp_att_val) > 1:
            intersection = set.intersection(*temp_att_val)

            #if the set still remains a subset of the original goal after removing current
element
            #then set the current element to None as we want to drop this later
            if intersection.issubset(current_goal):
                condition[item] = None

    condition = [x for x in condition if x is not None]

    return condition

#-------Actual MLEM2 Algorithm-------#
def stepAlgo(df3,selected_case,current_goal,B,condition,concept_curr):

    rule_set = []
    original_goal = current_goal.copy()


    while current_goal != set():
        #Check if the selected case is a subset of the current goal
        #List of current case
        A = df3['att_val'].loc[selected_case]

        if B == set():
            #Copy over the current set elements to B
            for i in range(len(A)):
                B.add(A[i])

        #Elements of intersection of current and previous set
        A = set(A).intersection(B)
        B = A.copy()

        print("Intersection set: ", A)
        #Check if intersecting elements are subset of Goal
        if A.issubset(original_goal):
            print("SUBSET")
            #Current goal is updated after discarding the already covered goal by new rule
            if len(A) != 0:
                current_goal = set(current_goal) - df3['goal_intersect'].loc[selected_case]
            else:
                current_goal = set()

            print("Current goal: ", current_goal)
            #Extract the current case
            curr_case = df3['Cases'].loc[selected_case]
            #Add the conditions of a Rule
            condition.append(curr_case)

            print("Condition before drop or combine: ", condition)
            #Check for possibility of dropping conditions
            if len(condition) > 1:
                condition = dropCondition(condition,original_goal)

            #Combine the interval
            if len(condition) > 1:
                condition = combineInterval(condition)


            #Join conditions
            cond = ""
            for item in condition:
                cond = cond + "(" + str(item) + ")" + " & "

            cond = cond[:-2] + "->"
            rule = cond + " (" + concept + "," + concept_curr + ")"
            rule_set.append(rule)

            #Reset everythng and continue for covering rest of the goal
            condition = []
            B = set()
            findGoalIntersect(current_goal)
            selected_case = findCases(df3)

        #If not a subset of current goal
        else:
            print("NOT")
            #Assign empty set for the selected case for next iteration
            df3['goal_intersect'].loc[selected_case] = set()
            print("need to be set to NULL", df3['Cases'].loc[selected_case])

            #Extract the current case
            curr_case = df3['Cases'].loc[selected_case]
            #Add the case to the condition list
            condition.append(curr_case)

            #Check for Range overlapping of the remaining cases
            if ".." in curr_case:
                second_part = curr_case.split(',')[1]
                start = float(second_part.split('..')[0])
                end = float(second_part.split('..')[1])
                for index, row in df3.iterrows():
                    if ".." in row['Cases'] and row['Cases'] == curr_case:
                        part2 = row['Cases'].split(',')[1]
                        start1 = float(part2.split('..')[0])
                        end1 = float(part2.split('..')[1])

                        #Assign blank set for cases with overlapping ranges
                        if set((pl.frange(start,end))).issubset(pl.frange(start1,end1)) == True:
                            if start == start1 and end <= end1:
                                row['goal_intersect'] = set()
                                print("also need to be set to NULL", row['Cases'])

            updateGoalIntersect(A.intersection(current_goal))
            selected_case = findCases(df3)

    return rule_set

#-------Reading Input Dataset-------#
df = pd.read_csv('../data/Iris/Iris-35-attcon.csv')
appr_df = df.copy()

#--------Find the Decision and unique Concepts-------#
df_headers = list(df)
concept = df_headers[-1]
concept_list = df[concept].unique()

#-------Calculating cases by concepts and making list of Goals-------#
#universal list containing all cases
U = []
temp_list = []
goal_list = []
for item in concept_list:
    for index, row in df.iterrows():
        U.append(index+1)
        if row[concept] == item:
            temp_list.append(index)
    goal_list.append(temp_list)
    temp_list = []

#-------Find out all the attributes in the dataset-------#
attributes = list(df)

#-------Find the numeric column-------#
numeric_col = df.select_dtypes(include=[np.number]).columns.tolist()
```

```
#-------Build all the cases-------#
case_list = []

#Discretization considering upto 2 decimal point
for item in attributes[:-1]:
        discretize(item)
case_list = list(OrderedDict.fromkeys(case_list))

#-------Building pairs for attribute-concept values-------#
df2 = df.copy()

for col in df:
    for i, row_value in df[col].iteritems():

        #This is for attribute-concept value interpretation
        #For Lost value interpretation, we don't add that case to any block at all
        if row_value == '-':

            final_values = []
            print(col,i)
            concept_tmp = df[concept][i]
            print(concept_tmp)

            for item_list in goal_list:
                if i in item_list:
                    final_values = item_list

            print (final_values)

            #Taking out all unique cases corresponding to the matched concept
            col_values = []
            for item in final_values:
                col_values.append(df[col][item])


            uniqueList = []
            for letter in col_values:
                if letter not in uniqueList:
                    uniqueList.append(letter)

            if '-' in uniqueList:
                uniqueList.remove('-')

            if '?' in uniqueList:
                uniqueList.remove('?')
            print(uniqueList)

            df2.at[i, col] = uniqueList

    print("\n")

appr_df_attcon = df2.copy()
df = df2.copy()
df_headers = list(df)
concept = df_headers[-1]
concept_list = df[concept].unique()

#-------Calculating cases by concepts and making sets-------#
temp_list = []
goal_list = []
for item in concept_list:
    for index, row in df.iterrows():
        if row[concept] == item:
            temp_list.append(index+1)
    goal_list.append(temp_list)
    temp_list = []

#-------Calculating blocks for attribute-value pairs-------#
temp_list = []
att_val_list = []
for item in case_list:
    a,b = item.split(",") #a = attribute and b = value
    if ".." in b:
        start,end = b.split("..")
        for index, row in df.iterrows():
            if type(row[a]) == list:
                tmp_list = row[a]
                for case in tmp_list:
                    if float(case) >= float(start) and float(case) <= float(end):
                        temp_list.append(index+1)


            else:
                if float(row[a]) >= float(start) and float(row[a]) <= float(end):
                    temp_list.append(index+1)

        temp_list = list(set(temp_list))
        att_val_list.append(temp_list)
        temp_list = []

    else:
        for index, row in df.iterrows():
            if type(row[a]) == list:
                tmp_list = row[a]
                for case in tmp_list:
                    if float(case) == b:
                        temp_list.append(index+1)
            else:
                if float(row[a]) == b:
                    temp_list.append(index+1)

        temp_list = list(set(temp_list))
        att_val_list.append(temp_list)
        temp_list = []

#-------Creating data for case and att-value list-------#
data = {'Cases': case_list, 'att_val': att_val_list}
df2 = pd.DataFrame(data)

#-------Here starts the Approximation calculations-------#
attributes = list(appr_df)
del attributes[-1]
U = set(U)

case_list = []
#Loop through all the attributes except last 2 - Concept and sort_col
for item in attributes:
    print(item)

    #check for non numeric columns
    if not is_numeric_dtype(appr_df[item]):
        temp = appr_df[item].unique()
        print(temp)
        for i in temp:
            if i == '?' or i == '-':
                continue
            else:
                case = item + "," + i
                case_list.append(case)

temp_list = []
att_val_list = []
for item in case_list:
    a,b = item.split(",") #a = attribute and b = value
    if ".." in b:
        start,end = b.split("..")
        for index, row in appr_df_attcon.iterrows():
            if type(row[a]) == list:
                tmp_list = row[a]
                for case in tmp_list:
                    if float(case) >= float(start) and float(case) <= float(end):
                        temp_list.append(index+1)
            else:
                if float(row[a]) >= float(start) and float(row[a]) <= float(end):
                    temp_list.append(index+1)

        temp_list = list(set(temp_list))
        att_val_list.append(temp_list)
        temp_list = []

    else:
        for index, row in appr_df_attcon.iterrows():
            if type(row[a]) == list:
                tmp_list = row[a]
                for case in tmp_list:
                    if float(case) == float(b):
                        temp_list.append(index+1)
            else:
                if float(row[a]) == float(b):
                    temp_list.append(index+1)

        temp_list = list(set(temp_list))
        att_val_list.append(temp_list)
        temp_list = []

#-------reating dictionary combining case_list and att_val list--------#
block = dict(zip(case_list, att_val_list))
attributes = list(df)

#-------Building Characteristic Sets-------#
dic = {}
for index, row in df.iterrows():
    tmp_set = set()
    final_union = []
    char_list = []
    char_list_2 = []
    final_union_set = []

    for cols in attributes[:-1]:
        #If the value for corresponding attribute is a list then create all of the att-value
pairs
        if type(df.loc[index,cols]) == list:
            print("When values are list")
            for item in df.loc[index,cols]:
                block_key = cols + "," + item
                char_list.append(block_key) #char_list has all att-val cases
                print(char_list)

            union_set = set()
            #Compute union of att-concept value case
            for item in char_list:
                union_set = union_set.union(set(block[item]))

            print("Union Set: ", union_set)
            final_union.append(union_set)

        else:
            print("When value is single")
            block_key = cols + "," + str(df.loc[index,cols])
            char_list_2.append(block_key) #char_list_2 has all single cases
            print(char_list_2)

    #Compute instersection for this current row for Characteristics set

    print("final_union: ", final_union)
    if len(final_union):
        final_union_set = list(reduce(set.intersection, [set(item) for item in final_union]))


    print("Final Union Set: ", final_union_set)
    for item in char_list_2:
        if item in block:
            print("When item is found as key in the block")
            if tmp_set == set():
                #Copy over the current set elements to B
                for i in range(len(block[item])):
                    tmp_set.add(block[item][i])

            tmp_set = tmp_set.intersection(set(block[item]))
            print(tmp_set)

        #If item not in block
        else:
            print("When item is not found as key in the block")
            print(tmp_set)
            print(U)
            if tmp_set == set():
                tmp_set = U

            tmp_set = tmp_set.intersection(U)
            print(tmp_set)

    print("Final tmp_set: ", tmp_set)
    final_union_set = set(final_union_set)

    if final_union_set == set():
        tmp_set = tmp_set
    else:
        tmp_set = tmp_set.intersection(final_union_set)

    print("This is the final value: ", tmp_set)

    key = ('K_%d' % (index+1))
    print(key)
    dic[key] = tmp_set
    print("\n")

#-------Form the Approximations-------#
lower_approximations = {}
for item in goal_list:
    #Key is the string converted list so as to add as dictionary key
    lower_approximations[str(item)] = lowerApproximation(dic,item)

lower_goal_list=list(lower_approximations.values())

upper_approximations = {}
for item in goal_list:
    #Key is the string converted list so as to add as dictionary key
    upper_approximations[str(item)] = upperApproximation(dic,item)

upper_goal_list=list(upper_approximations.values())

first_column = list(dic.keys())
second_column = list(dic.values())
prob_approx = pd.DataFrame(
    {'charset_name': first_column,
     'charset_value': second_column
    })

middle_approximations = {}
for item in goal_list:
    #Key is the string converted list so as to add as dictionary key
    middle_approximations[str(item)] = findMiddleApprox(item)

middle_goal_list=list(middle_approximations.values())

df3=df2

#concept_list and goal_list has 1:1 mapping
final_rules = []
start_time = time.time()
```

```
#-------Running algorithm for all the goals - Middle Approximation/concepts-------#
for i in range(0,len(middle_goal_list)):
    SPECIAL = U - middle_goal_list[i]
    findGoalIntersect(list(middle_goal_list[i]))

    condition = []
    B = set()
    selected_case = findCases(df3)

    rule_set = stepAlgo(df3,selected_case,middle_goal_list[i],B,condition,concept_list[i])

    #Now do for the SPECIAL cases
    findGoalIntersect(list(SPECIAL))
    condition_SPECIAL = []
    B_SPECIAL = set()
    selected_case_SPECIAL = findCases(df3)
    rule_set2 = stepAlgo(df3,selected_case_SPECIAL,SPECIAL,B_SPECIAL,condition_SPECIAL,"SPECIAL")

    final_rules.append(rule_set)
    final_rules.append(rule_set2)
    print("End of goal")

elapsed_time = time.time() - start_time

print("Time to run the algorithm (for Middle Approximation): ", round(elapsed_time, 3), "Sec")
print("\n")
print(*final_rules, sep='\n')

with open('../data/Iris/Rules/Iris_test_attcon_middle.txt', 'w') as f:
    for item in final_rules:
        f.write("%s\n" % item)

#-------Running algorithm for all the goals - Lower Approximation/concepts-------#

#concept_list and goal_list has 1:1 mapping
final_rules = []
start_time = time.time()

#Running algorithm for all the goals - Lower Approximation/concepts
for i in range(0,len(lower_goal_list)):
    SPECIAL = U - lower_goal_list[i]
    findGoalIntersect(list(lower_goal_list[i]))
    condition = []
    B = set()
    selected_case = findCases(df3)

    rule_set = stepAlgo(df3,selected_case,lower_goal_list[i],B,condition,concept_list[i])

    #Now do for the SPECIAL cases
    findGoalIntersect(list(SPECIAL))
    condition_SPECIAL = []
    B_SPECIAL = set()
    selected_case_SPECIAL = findCases(df3)
    rule_set2 = stepAlgo(df3,selected_case_SPECIAL,SPECIAL,B_SPECIAL,condition_SPECIAL,"SPECIAL")

    final_rules.append(rule_set)
    final_rules.append(rule_set2)
    print("End of goal")

elapsed_time = time.time() - start_time


print("Time to run the algorithm (for Lower Approximation): ", round(elapsed_time, 3), "Sec")
print("\n")
print(*final_rules, sep='\n')

with open('../data/Iris/Rules/Iris_test_attcon_lower.txt', 'w') as f:
    for item in final_rules:
        f.write("%s\n" % item)
```

```
#-------Running algorithm for all the goals - Upper Approximation/concepts-------#

#concept_list and goal_list has 1:1 mapping
final_rules = []
start_time = time.time()

#Running algorithm for all the goals - Lower Approximation/concepts
for i in range(0,len(upper_goal_list)):
    SPECIAL = U - upper_goal_list[i]
    findGoalIntersect(list(upper_goal_list[i]))
    condition = []
    B = set()
    selected_case = findCases(df3)

    rule_set = stepAlgo(df3,selected_case,upper_goal_list[i],B,condition,concept_list[i])

    #Now do for the SPECIAL cases
    findGoalIntersect(list(SPECIAL))
    condition_SPECIAL = []
    B_SPECIAL = set()
    selected_case_SPECIAL = findCases(df3)
    rule_set2 = stepAlgo(df3,selected_case_SPECIAL,SPECIAL,B_SPECIAL,condition_SPECIAL,"SPECIAL")

    final_rules.append(rule_set)
    final_rules.append(rule_set2)
    print("End of goal")

elapsed_time = time.time() - start_time

print("Time to run the algorithm (for Upper Approximation): ", round(elapsed_time, 3), "Sec")
print("\n")
print(*final_rules, sep='\n')

with open('../data/Iris/Rules/Iris_test_attcon_upper.txt', 'w') as f:
    for item in final_rules:
        f.write("%s\n" % item)
#-----------------------END-------------------------#
```

A.3 Rule-Complexity-Rule.py

```
#-------Importing Libraries-------#
import pandas
import itertools
import re
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy import interpolate
from scipy.interpolate import interp1d
from scipy.interpolate import splrep, splev
from collections import defaultdict

#-------Method to find the counts of Rule and Condiion-------#
def findRuleComplexity(lines):
    rule_count = 0
    condition_count = 0
    for item in lines:
        item = item[1:len(item)-2]
        item = item.replace(", ","\n")
        item = item.split("\n")

        rule_count += len(item)
        for rule in item:
            left,right = rule.split("->")
            conditions = left.split("&")
            length = len(conditions)
            condition_count += length
    return rule_count,condition_count
```

```
#-------When no missing attributes are present-------#
f1 = open('../data/Iris/Rules/Iris_main.txt', "r")

data = {}
data["Lower," + "?"] = list(f1)
data["Middle," + "?"] = data["Lower," + "?"]
data["Upper," + "?"] = data["Lower," + "?"]
data["Lower," + "-"] = data["Lower," + "?"]
data["Middle," + "-"] = data["Lower," + "?"]
data["Upper," + "-"] = data["Lower," + "?"]

Iris_0 = {}
for key, value in data.items():
    rule,condition = findRuleComplexity(value)
    Iris_0[key] = [rule]
    Iris_0[key].append(condition)

#-------5% missing values-------#
f1 = open('../data/Iris/Rules/Iris_5_lost_lower.txt', "r")
f2 = open('../data/Iris/Rules/Iris_5_lost_middle.txt', "r")
f3 = open('../data/Iris/Rules/Iris_5_lost_upper.txt', "r")
f4 = open('../data/Iris/Rules/Iris_5_attcon_lower.txt', "r")
f5 = open('../data/Iris/Rules/Iris_5_attcon_middle.txt', "r")
f6 = open('../data/Iris/Rules/Iris_5_attcon_upper.txt', "r")

data = {}
data["Lower," + "?"] = list(f1)
data["Middle," + "?"] = list(f2)
data["Upper," + "?"] = list(f3)
data["Lower," + "-"] = list(f4)
data["Middle," + "-"] = list(f5)
data["Upper," + "-"] = list(f6)

Iris_5 = {}
for key, value in data.items():
    rule,condition = findRuleComplexity(value)
    Iris_5[key] = [rule]
    Iris_5[key].append(condition)

#-------10% missing values-------#
f1 = open('../data/Iris/Rules/Iris_10_lost_lower.txt', "r")
f2 = open('../data/Iris/Rules/Iris_10_lost_middle.txt', "r")
f3 = open('../data/Iris/Rules/Iris_10_lost_upper.txt', "r")
f4 = open('../data/Iris/Rules/Iris_10_attcon_lower.txt', "r")
f5 = open('../data/Iris/Rules/Iris_10_attcon_middle.txt', "r")
f6 = open('../data/Iris/Rules/Iris_10_attcon_upper.txt', "r")

data = {}
data["Lower," + "?"] = list(f1)
data["Middle," + "?"] = list(f2)
data["Upper," + "?"] = list(f3)
data["Lower," + "-"] = list(f4)
data["Middle," + "-"] = list(f5)
data["Upper," + "-"] = list(f6)

Iris_10 = {}
for key, value in data.items():
    rule,condition = findRuleComplexity(value)
    Iris_10[key] = [rule]
    Iris_10[key].append(condition)

#-------15% missing values-------#
f1 = open('../data/Iris/Rules/Iris_15_lost_lower.txt', "r")
f2 = open('../data/Iris/Rules/Iris_15_lost_middle.txt', "r")
f3 = open('../data/Iris/Rules/Iris_15_lost_upper.txt', "r")
f4 = open('../data/Iris/Rules/Iris_15_attcon_lower.txt', "r")
f5 = open('../data/Iris/Rules/Iris_15_attcon_middle.txt', "r")
f6 = open('../data/Iris/Rules/Iris_15_attcon_upper.txt', "r")

data = {}
```

```
data["Lower," + "?"] = list(f1)
data["Middle," + "?"] = list(f2)
data["Upper," + "?"] = list(f3)
data["Lower," + "-"] = list(f4)
data["Middle," + "-"] = list(f5)
data["Upper," + "-"] = list(f6)

Iris_15 = {}
for key, value in data.items():
    rule,condition = findRuleComplexity(value)
    Iris_15[key] = [rule]
    Iris_15[key].append(condition)

#-------20% missing values-------#
f1 = open('../data/Iris/Rules/Iris_20_lost_lower.txt', "r")
f2 = open('../data/Iris/Rules/Iris_20_lost_middle.txt', "r")
f3 = open('../data/Iris/Rules/Iris_20_lost_upper.txt', "r")
f4 = open('../data/Iris/Rules/Iris_20_attcon_lower.txt', "r")
f5 = open('../data/Iris/Rules/Iris_20_attcon_middle.txt', "r")
f6 = open('../data/Iris/Rules/Iris_20_attcon_upper.txt', "r")

data = {}
data["Lower," + "?"] = list(f1)
data["Middle," + "?"] = list(f2)
data["Upper," + "?"] = list(f3)
data["Lower," + "-"] = list(f4)
data["Middle," + "-"] = list(f5)
data["Upper," + "-"] = list(f6)

Iris_20 = {}
for key, value in data.items():
    rule,condition = findRuleComplexity(value)
    Iris_20[key] = [rule]
    Iris_20[key].append(condition)

#-------25% missing values-------#
f1 = open('../data/Iris/Rules/Iris_25_lost_lower.txt', "r")
f2 = open('../data/Iris/Rules/Iris_25_lost_middle.txt', "r")
f3 = open('../data/Iris/Rules/Iris_25_lost_upper.txt', "r")
f4 = open('../data/Iris/Rules/Iris_25_attcon_lower.txt', "r")
f5 = open('../data/Iris/Rules/Iris_25_attcon_middle.txt', "r")
f6 = open('../data/Iris/Rules/Iris_25_attcon_upper.txt', "r")

data = {}
data["Lower," + "?"] = list(f1)
data["Middle," + "?"] = list(f2)
data["Upper," + "?"] = list(f3)
data["Lower," + "-"] = list(f4)
data["Middle," + "-"] = list(f5)
data["Upper," + "-"] = list(f6)

Iris_25 = {}
for key, value in data.items():
    rule,condition = findRuleComplexity(value)
    Iris_25[key] = [rule]
    Iris_25[key].append(condition)

#-------30% missing values-------#
f1 = open('../data/Iris/Rules/Iris_30_lost_lower.txt', "r")
f2 = open('../data/Iris/Rules/Iris_30_lost_middle.txt', "r")
f3 = open('../data/Iris/Rules/Iris_30_lost_upper.txt', "r")
f4 = open('../data/Iris/Rules/Iris_30_attcon_lower.txt', "r")
f5 = open('../data/Iris/Rules/Iris_30_attcon_middle.txt', "r")
f6 = open('../data/Iris/Rules/Iris_30_attcon_upper.txt', "r")

data = {}
data["Lower," + "?"] = list(f1)
data["Middle," + "?"] = list(f2)
data["Upper," + "?"] = list(f3)
data["Lower," + "-"] = list(f4)
data["Middle," + "-"] = list(f5)
```

```
data["Upper," + "-"] = list(f6)

Iris_30 = {}
for key, value in data.items():
    rule,condition = findRuleComplexity(value)
    Iris_30[key] = [rule]
    Iris_30[key].append(condition)

#-------35% missing values-------#
f1 = open('../data/Iris/Rules/Iris_35_lost_lower.txt', "r")
f2 = open('../data/Iris/Rules/Iris_35_lost_middle.txt', "r")
f3 = open('../data/Iris/Rules/Iris_35_lost_upper.txt', "r")
f4 = open('../data/Iris/Rules/Iris_35_attcon_lower.txt', "r")
f5 = open('../data/Iris/Rules/Iris_35_attcon_middle.txt', "r")
f6 = open('../data/Iris/Rules/Iris_35_attcon_upper.txt', "r")

data = {}
data["Lower," + "?"] = list(f1)
data["Middle," + "?"] = list(f2)
data["Upper," + "?"] = list(f3)
data["Lower," + "-"] = list(f4)
data["Middle," + "-"] = list(f5)
data["Upper," + "-"] = list(f6)

Iris_35 = {}
for key, value in data.items():
    rule,condition = findRuleComplexity(value)
    Iris_35[key] = [rule]
    Iris_35[key].append(condition)

#-------Rule Count-------#
dd = defaultdict(list)

for d in (Iris_0, Iris_5, Iris_10, Iris_15, Iris_20, Iris_25, Iris_30, Iris_35):
    for key, value in d.items():
        dd[key].append(value[0])

#------Condition Count-------#

cc = defaultdict(list)

for d in (Iris_0, Iris_5, Iris_10, Iris_15, Iris_20, Iris_25, Iris_30, Iris_35):
    for key, value in d.items():
        cc[key].append(value[1])

df = pd.DataFrame.from_dict(dd)
df2 = pd.DataFrame.from_dict(cc)

#-------Plotting-------#
x = np.arange(0, 40, 5)
df['x'] = x
df2['x'] = x

#-------Rule Count-------#
ax = plt.gca()
ax.grid(True)
plt.gcf().set_size_inches(10, 5)

tck1 = splrep(df['x'], df['Lower,?'])
xnew1 = np.linspace(0, 35)
ynew1 = splev(xnew1, tck1)
plt.plot(xnew1, ynew1)

tck2 = splrep(df['x'], df['Middle,?'])
xnew2 = np.linspace(0, 35)
ynew2 = splev(xnew2, tck2)
plt.plot(xnew2, ynew2)

tck3 = splrep(df['x'], df['Upper,?'])
xnew3 = np.linspace(0, 35)
ynew3 = splev(xnew3, tck3)
```

```
plt.plot(xnew3, ynew3)

tck4 = splrep(df['x'], df['Lower,-'])
xnew4 = np.linspace(0, 35)
ynew4 = splev(xnew4, tck4)
plt.plot(xnew4, ynew4)

tck5 = splrep(df['x'], df['Middle,-'])
xnew5 = np.linspace(0, 35)
ynew5 = splev(xnew5, tck5)
plt.plot(xnew5, ynew5)

tck6 = splrep(df['x'], df['Upper,-'])
xnew6 = np.linspace(0, 35)
ynew6 = splev(xnew6, tck6)
plt.plot(xnew6, ynew6)

plt.title("Iris", fontsize=15)
plt.xlabel('Missing Attributes (%)', fontsize=15)
plt.ylabel('Rule Count', fontsize=15)
plt.plot(df['x'], df['Lower,?'], 'o', color='blue', markerfacecolor='blue')
plt.plot(df['x'], df['Middle,?'], 's', color='red', markerfacecolor='red')
plt.plot(df['x'], df['Upper,?'], '^', color='green', markerfacecolor='green')
plt.plot(df['x'], df['Lower,-'], 'x', color='magenta', markerfacecolor='magenta')
plt.plot(df['x'], df['Middle,-'], 'D', color='aqua', markerfacecolor='aqua')
plt.plot(df['x'], df['Upper,-'], 'P', color='orange', markerfacecolor='orange')
plt.legend()
plt.savefig('../Results/Iris-RuleTest.png')
#------------END---------------#
```

A.4 Rule-Complexity-Condition.py

```
#-------Importing Libraries-------#
import pandas
import itertools
import re
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy import interpolate
from scipy.interpolate import interp1d
from scipy.interpolate import splrep, splev
from collections import defaultdict

#-------Method to find the counts of Rule and Condiion-------#
def findRuleComplexity(lines):
    rule_count = 0
    condition_count = 0
    for item in lines:
        item = item[1:len(item)-2]
        item = item.replace(", ","\n")
        item = item.split("\n")

        rule_count += len(item)
        for rule in item:
            left,right = rule.split("->")
            conditions = left.split("&")
            length = len(conditions)
            condition_count += length
    return rule_count,condition_count

#-------When no missing attributes are present-------#
f1 = open('../data/Iris/Rules/Iris_main.txt', "r")

data = {}
data["Lower," + "?"] = list(f1)
data["Middle," + "?"] = data["Lower," + "?"]
data["Upper," + "?"] = data["Lower," + "?"]
```

```
data["Lower," + "-"] = data["Lower," + "?"]
data["Middle," + "-"] = data["Lower," + "?"]
data["Upper," + "-"] = data["Lower," + "?"]

Iris_0 = {}
for key, value in data.items():
    rule,condition = findRuleComplexity(value)
    Iris_0[key] = [rule]
    Iris_0[key].append(condition)

#-------5% missing values-------#
f1 = open('../data/Iris/Rules/Iris_5_lost_lower.txt', "r")
f2 = open('../data/Iris/Rules/Iris_5_lost_middle.txt', "r")
f3 = open('../data/Iris/Rules/Iris_5_lost_upper.txt', "r")
f4 = open('../data/Iris/Rules/Iris_5_attcon_lower.txt', "r")
f5 = open('../data/Iris/Rules/Iris_5_attcon_middle.txt', "r")
f6 = open('../data/Iris/Rules/Iris_5_attcon_upper.txt', "r")

data = {}
data["Lower," + "?"] = list(f1)
data["Middle," + "?"] = list(f2)
data["Upper," + "?"] = list(f3)
data["Lower," + "-"] = list(f4)
data["Middle," + "-"] = list(f5)
data["Upper," + "-"] = list(f6)

Iris_5 = {}
for key, value in data.items():
    rule,condition = findRuleComplexity(value)
    Iris_5[key] = [rule]
    Iris_5[key].append(condition)

#-------10% missing values-------#
f1 = open('../data/Iris/Rules/Iris_10_lost_lower.txt', "r")
f2 = open('../data/Iris/Rules/Iris_10_lost_middle.txt', "r")
f3 = open('../data/Iris/Rules/Iris_10_lost_upper.txt', "r")
f4 = open('../data/Iris/Rules/Iris_10_attcon_lower.txt', "r")
f5 = open('../data/Iris/Rules/Iris_10_attcon_middle.txt', "r")
f6 = open('../data/Iris/Rules/Iris_10_attcon_upper.txt', "r")

data = {}
data["Lower," + "?"] = list(f1)
data["Middle," + "?"] = list(f2)
data["Upper," + "?"] = list(f3)
data["Lower," + "-"] = list(f4)
data["Middle," + "-"] = list(f5)
data["Upper," + "-"] = list(f6)

Iris_10 = {}
for key, value in data.items():
    rule,condition = findRuleComplexity(value)
    Iris_10[key] = [rule]
    Iris_10[key].append(condition)

#-------15% missing values-------#
f1 = open('../data/Iris/Rules/Iris_15_lost_lower.txt', "r")
f2 = open('../data/Iris/Rules/Iris_15_lost_middle.txt', "r")
f3 = open('../data/Iris/Rules/Iris_15_lost_upper.txt', "r")
f4 = open('../data/Iris/Rules/Iris_15_attcon_lower.txt', "r")
f5 = open('../data/Iris/Rules/Iris_15_attcon_middle.txt', "r")
f6 = open('../data/Iris/Rules/Iris_15_attcon_upper.txt', "r")

data = {}
data["Lower," + "?"] = list(f1)
data["Middle," + "?"] = list(f2)
data["Upper," + "?"] = list(f3)
data["Lower," + "-"] = list(f4)
data["Middle," + "-"] = list(f5)
data["Upper," + "-"] = list(f6)

Iris_15 = {}
```

```python
for key, value in data.items():
    rule,condition = findRuleComplexity(value)
    Iris_15[key] = [rule]
    Iris_15[key].append(condition)

#-------20% missing values-------#
f1 = open('../data/Iris/Rules/Iris_20_lost_lower.txt', "r")
f2 = open('../data/Iris/Rules/Iris_20_lost_middle.txt', "r")
f3 = open('../data/Iris/Rules/Iris_20_lost_upper.txt', "r")
f4 = open('../data/Iris/Rules/Iris_20_attcon_lower.txt', "r")
f5 = open('../data/Iris/Rules/Iris_20_attcon_middle.txt', "r")
f6 = open('../data/Iris/Rules/Iris_20_attcon_upper.txt', "r")

data = {}
data["Lower," + "?"] = list(f1)
data["Middle," + "?"] = list(f2)
data["Upper," + "?"] = list(f3)
data["Lower," + "-"] = list(f4)
data["Middle," + "-"] = list(f5)
data["Upper," + "-"] = list(f6)

Iris_20 = {}
for key, value in data.items():
    rule,condition = findRuleComplexity(value)
    Iris_20[key] = [rule]
    Iris_20[key].append(condition)

#-------25% missing values-------#
f1 = open('../data/Iris/Rules/Iris_25_lost_lower.txt', "r")
f2 = open('../data/Iris/Rules/Iris_25_lost_middle.txt', "r")
f3 = open('../data/Iris/Rules/Iris_25_lost_upper.txt', "r")
f4 = open('../data/Iris/Rules/Iris_25_attcon_lower.txt', "r")
f5 = open('../data/Iris/Rules/Iris_25_attcon_middle.txt', "r")
f6 = open('../data/Iris/Rules/Iris_25_attcon_upper.txt', "r")

data = {}
data["Lower," + "?"] = list(f1)
data["Middle," + "?"] = list(f2)
data["Upper," + "?"] = list(f3)
data["Lower," + "-"] = list(f4)
data["Middle," + "-"] = list(f5)
data["Upper," + "-"] = list(f6)

Iris_25 = {}
for key, value in data.items():
    rule,condition = findRuleComplexity(value)
    Iris_25[key] = [rule]
    Iris_25[key].append(condition)

#-------30% missing values-------#
f1 = open('../data/Iris/Rules/Iris_30_lost_lower.txt', "r")
f2 = open('../data/Iris/Rules/Iris_30_lost_middle.txt', "r")
f3 = open('../data/Iris/Rules/Iris_30_lost_upper.txt', "r")
f4 = open('../data/Iris/Rules/Iris_30_attcon_lower.txt', "r")
f5 = open('../data/Iris/Rules/Iris_30_attcon_middle.txt', "r")
f6 = open('../data/Iris/Rules/Iris_30_attcon_upper.txt', "r")

data = {}
data["Lower," + "?"] = list(f1)
data["Middle," + "?"] = list(f2)
data["Upper," + "?"] = list(f3)
data["Lower," + "-"] = list(f4)
data["Middle," + "-"] = list(f5)
data["Upper," + "-"] = list(f6)

Iris_30 = {}
for key, value in data.items():
    rule,condition = findRuleComplexity(value)
    Iris_30[key] = [rule]
    Iris_30[key].append(condition)
```

```
#-------35% missing values-------#
f1 = open('../data/Iris/Rules/Iris_35_lost_lower.txt', "r")
f2 = open('../data/Iris/Rules/Iris_35_lost_middle.txt', "r")
f3 = open('../data/Iris/Rules/Iris_35_lost_upper.txt', "r")
f4 = open('../data/Iris/Rules/Iris_35_attcon_lower.txt', "r")
f5 = open('../data/Iris/Rules/Iris_35_attcon_middle.txt', "r")
f6 = open('../data/Iris/Rules/Iris_35_attcon_upper.txt', "r")

data = {}
data["Lower," + "?"] = list(f1)
data["Middle," + "?"] = list(f2)
data["Upper," + "?"] = list(f3)
data["Lower," + "-"] = list(f4)
data["Middle," + "-"] = list(f5)
data["Upper," + "-"] = list(f6)

Iris_35 = {}
for key, value in data.items():
    rule,condition = findRuleComplexity(value)
    Iris_35[key] = [rule]
    Iris_35[key].append(condition)

#-------Rule Count-------#
dd = defaultdict(list)

for d in (Iris_0, Iris_5, Iris_10, Iris_15, Iris_20, Iris_25, Iris_30, Iris_35):
    for key, value in d.items():
        dd[key].append(value[0])

#------Condition Count-------#

cc = defaultdict(list)

for d in (Iris_0, Iris_5, Iris_10, Iris_15, Iris_20, Iris_25, Iris_30, Iris_35):
    for key, value in d.items():
        cc[key].append(value[1])

df = pd.DataFrame.from_dict(dd)
df2 = pd.DataFrame.from_dict(cc)

#-------Plotting-------#
x = np.arange(0, 40, 5)
df['x'] = x
df2['x'] = x

#-------Condition-------#
from scipy.interpolate import splrep, splev
ax = plt.gca()
ax.grid(True)
plt.gcf().set_size_inches(10, 5)

tck1 = splrep(df2['x'], df2['Lower,?'])
xnew1 = np.linspace(0, 35)
ynew1 = splev(xnew1, tck1)
plt.plot(xnew1, ynew1)

tck2 = splrep(df2['x'], df2['Middle,?'])
xnew2 = np.linspace(0, 35)
ynew2 = splev(xnew2, tck2)
plt.plot(xnew2, ynew2)

tck3 = splrep(df2['x'], df2['Upper,?'])
xnew3 = np.linspace(0, 35)
ynew3 = splev(xnew3, tck3)
plt.plot(xnew3, ynew3)

tck4 = splrep(df2['x'], df2['Lower,-'])
xnew4 = np.linspace(0, 35)
ynew4 = splev(xnew4, tck4)
plt.plot(xnew4, ynew4)
```

```
tck5 = splrep(df2['x'], df2['Middle,-'])
xnew5 = np.linspace(0, 35)
ynew5 = splev(xnew5, tck5)
plt.plot(xnew5, ynew5)

tck6 = splrep(df2['x'], df2['Upper,-'])
xnew6 = np.linspace(0, 35)
ynew6 = splev(xnew6, tck6)
plt.plot(xnew6, ynew6)

plt.title("Iris", fontsize=15)
plt.xlabel('Missing Attributes (%)', fontsize=15)
plt.ylabel('Condition Count', fontsize=15)
plt.plot(df2['x'], df2['Lower,?'], 'o', color='blue', markerfacecolor='blue')
plt.plot(df2['x'], df2['Middle,?'], 's', color='red', markerfacecolor='red')
plt.plot(df2['x'], df2['Upper,?'], '^', color='green', markerfacecolor='green')
plt.plot(df2['x'], df2['Lower,-'], 'x', color='magenta', markerfacecolor='magenta')
plt.plot(df2['x'], df2['Middle,-'], 'D', color='aqua', markerfacecolor='aqua')
plt.plot(df2['x'], df2['Upper,-'], 'P', color='orange', markerfacecolor='orange')
plt.legend()
plt.savefig('../Results/Iris-ConditionTest.png')
#----------END----------#
```