Rule based Machine Learning (RBML) is a relatively broad term. It encompasses any machine learning (ML) method that identifies, learns, or evolves a set of a rules to store, manipulate, or apply. As such, there is a wide range of methods this includes. However, let's look at the bigger picture of it all first.

1. What is RBML?

First, it is important to know what machine learning is. According to expertsysem.com, "Machine learning is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed." The computer is made to complete a specific task without being explicitly programmed to do so, as programming the computer for the given task is usually impractical. It must rely on patterns and inferences learned through a given data set or repeated experience and apply them to complete the task. The ultimate goal is to use computers to find methods of completing tasks that either replaces or improves upon human intuition.

The concept of machine learning is not new. According to dataversity.net, the term Machine Learning was coined by Arthur Samuel in 1952, who stated "it gives computers the ability to learn without being explicitly programmed." However, at the time, computers were extremely limited in what they could do. As such, machine learning was given a back shelf to other projects.

It was not until more recently that machine learning has gained new traction, as computers are now capable of doing much more complex tasks that make applying ML worthwhile. For example, they are able to apply complex calculations to big data at a rate that allows them to derive patterns humans would normally not be able to find or even understand.

This is different from simply a machine calculating every possible outcome and deciding the best course of action to take. In chess, for example, there are more possible games of chess than there are atoms in the universe (about 10^120 versus roughly 10^80). It would be impractical for a computer to calculate every move possible. Instead, the computer uses ML to detect useful tactics and significantly narrow down the possible outcomes to make computing the best move more practical.

With rule-based machine learning, it applies the same concept, except with a set a rules, usually in the form of if-then statements (if yellow and round, then lemon). The if-then statement on its own is not a blueprint for an RBML system, since the rule is only pertinent when the if condition is satisfied. Instead, the system uses a knowledge base that comprises of a large set of rules to create the prediction model.

An important distinction to make here is between a rule-based system (RBS) and rule-based machine learning (RBML). A rule-based system requires a human to manually construct rules and create a rule set based on prior knowledge and intuition. Rule-based machine learning uses a learning algorithm to automatically identify useful rules.

Take for example, a program that finds the best possible route for a mail truck to take. In an RBS, a human would tell the computer rules such as lining up each destination and avoiding

roads with high amounts of traffic. In RBML, the computer would have to find patterns in the faster routes and derive these rules to find the best route. This could result in the computer finding rules humans would have normally been unaware of, such as if the RBML system determined taking only right hand turns in the city would lower the travel time.

There is also the distinction between RBML and other ML systems. One of the larger distinctions is RBML does not require training, while general ML systems do. ML systems generally use sample data to create a universal model and base future predictions based off of the model. In contrast, RBML systems focus on identifying and utilizing a set of rules to represent material within the database.

Take for example, a computer identifying an apple within a group of fruits. A general ML system will look at a set of pictures of apples to create a model of what an apple commonly looks like, and then uses that model to identify future apples. In RBML, the system will find rules to determine what is and isn't an apple (e.g. If red and round and has a stem, then it is an apple). The difference would be training the machine to find what an apple typically looks like and comparing the two pictures of an apple, versus discovering rules from pictures to represent apples. It is a small, but important distinction between the two types of systems.

2. Learning classifier systems (LCS)

A specific example of RBML is a learning classifier system, or LCS. Learning classifier systems is a fairly old concept but has a lot of disadvantages that prevent it from being more widely used. This includes being complicated, computationally expensive, and lack popularity, resulting in LCS to be rarely considered in place of other machine learning techniques.

LCSs combine a discovery element with a learning element and utilize distributed optimization techniques.

The discovery component will generally be a genetic algorithm. A genetic algorithm is a mimic of Darwin's theory of natural selection to evolve sets of solutions towards better solutions. To begin, you start out with a population, or a set of solutions, that can typically range to the thousands. The population with then be filtered and selected through a fitness function that measures a solution's ability to achieve a given goal. Once the population is filtered, a new generation of solutions is created, resulting from parent solutions undergoing breeding using crossover and mutation. Then, the process is repeated until a satisfactory solution is found.

The learning component will use either supervised learning, unsupervised learning, or reinforcement learning.

Supervised Learning is based around using a set of training examples and coming up with a function that generalizes beyond the examples. You give a computer a set of inputs and a set of outputs and the computer will create a function for mapping future inputs to outputs. For instance, take the set of input-output pairs (1,2), (2,4), (3,6), and (4,8). One could infer from this A implies B such that $B = 2A$.

Unsupervised Learning lacks labels for the data set. Rather than inputs and outputs, you simply have a set of numbers, such as 1, 2, 2, 4, 3, 6, 4, 8, taken from the previous example. The computer then has to find undetected structures or patterns in the set. It may find the function found in the previous example, or a completely new pattern that we may never have found.

Reinforced Learning is a bit different. It focuses more on a computer learning to find the best actions for the best possible reward. A basic reinforcement model called the Markov decision process goes as this:

You have a set of environment states, S, and a set of actions, A. The probability of transition at time t from state s to state s' under action a is $P_a(s, s') = Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$. The immediate reward after transition from s to s' due to action a is $R_a(s, s')$. Then you have a set of rules that result from what the computer finds.

At each time t, the computer gets an observation and possible reward. It then chooses from a set of actions which will bring the greatest reward and sends it to the environment. The environment moves to the new state, and the reward associated with the action is recorded.

Michigan Style vs. Pittsburgh Style

The two main types of LCS are Michigan-Style and Pittsburgh-style, with Michigan-Style being the more traditional form of the two. Michigan-Style LCSs functions incrementally, evolving rules in a single rule-set one step at a time to arrive at a solution, as opposed to Pittsburgh-style which branches out into multiple potential rule-set solutions. This allows Michigan-Style LCS to operate with both supervised learning and reinforced learning. Michigan-style uses all classifiers within the population to cooperatively create a solution. In Pittsburgh-style, the classifiers compete to create a single solution. The style is less important, as it usually comes later in the process of creating the algorithm that the style is determined.

Expected Risk vs. Empirical Risk

Let's assume that we have a finite set of observations created by a series of tasks running a stochastic process. Each task has some hypothesis or model for how the process works. The process maps an input from space X to an output in space Y, so each observation (x,y) contains an input $x \in X$ that occurred and an output $y \in Y$ that resulted. The set of inputs ($X = \{x_1, x_2, \ldots, x_n\}$) and set of outputs ($Y = \{y_1, y_2, \ldots, y_n\}$) form the training set ($D = \{X, Y\}$).

In this example, we will approach using batch learning, assuming the entire training set D with n observations is available.

In order to model the process, we need to express the process using a function f: x -> y to create an observation x,y. Specifically, it needs to be ensured that given two equivalent inputs, say x and x' where x=x', the output will be the same. A lack of similar outputs means a lack of smoothness in the function. Typically, you want as smooth of a function as possible (particularly

for solutions represented by exact answers, like a math equation), though often exceptions are made (e.g. low risk high reward actions in a game). Smooth functions are required to make predictions, otherwise we could not generalize the training data. The function will be in the form $y = f(x) + \varepsilon$, where ε represents the stochasticity

Let a model with structure M have the hypothesis $f_M : X \rightarrow Y$. To be a good model, f_M has to be relatively close to f. Let $L : Y \times Y \rightarrow R^+$ be a loss function that describes the distance measurement Y between f and f_M. This means $L(y,y') > 0 \; \forall \; y \neq y$. To get f_M close to f, one needs to minimize the expected risk $\int L\big(f(x), f_M(x)\big)dp(x)$ where p(x) is the probability density of having input x. In other words, the goal is to minimize the distance between the output of the process and our model of it, with each x weighted by the possibility of observing it.

Unfortunately, the risk cannot be minimized directly, since f is only accessible through the finite set of observations. We will instead need to rely on an estimation of the expected risk when creating the model, or the empirical risk given by $\frac{1}{N}\sum_{n-1}^{N} L(y_n, f_m(x_n))$. This is the average loss over N observations.

Batch vs Incremental Learning

As previously stated, batch learning assumes the training set is available at once, meaning the order of observations is irrelevant. In Incremental learning, the set is updated with each observation, evolving the rules one step at a time. Incremental learning provides the unique advantage of being able to handle a non-stationary function, a process that changes with respect to when it is observed. However, incremental learning is less transparent in what it learns and is more complex.

Theoretically, incremental learning can be derived from batch learning created to solve the same problem. This allows it to have the flexibility of incremental learning while reserving the transparency of the batch learning method.

Take for example, an algorithm to estimate the probability of tossing a coin and landing on heads. The input $X = \emptyset$, and the output $Y = \{0,1\}$, with 0 for tail and 1 for head. The probability of heads will then be given by $P_N(H) = \frac{1}{N}\sum_{n=1}^{N} y_n$. Turning this into an incremental learning approach results in $P_N(H) = \frac{1}{N} y_N + \frac{1}{N}\sum_{n=1}^{N-1} y_n = P_{N-1}(H) + \frac{1}{N}(y_N - P_{N-1}(H))$. To update $P_{N-1}(H)$ with a new observation y_N, we only need to maintain the current number N of experiments. Now, it is less transparent with that it is calculating.

To implement a property changing over time, we can modify the equation to $P_N(H) = P_{N-1}(H) + \gamma(y_N - P_{N-1}(H))$, where γ is the factor that determines the influence between past observations and the current estimate, and $0 < \gamma \leq 1$.

3. Association Rule Learning

Association rule learning (ARL) is another form of rule-based machine learning. ARL focus around discovering similarities between variables in large datasets. The founders of the idea, Agrawal, Imielinski, and Swami introduced it for a market-basket analysis. For example, they found the rule {onions,potatoes}-> {burger}, meaning if customers bought onions and potatoes, they would also likely buy hamburger meat. This can be applied to a larger variety of problems as well.

To put things into a better perspective, take this example. Let $I = \{i_1,i_2,\ldots,i_n\}$, a set of n items. Let $D = \{t_1,t_2,\ldots,t_m\}$, a set of m transactions.

Transaction	Milk	Cereal	Eggs
1	1	1	1
2	1	0	1
3	0	0	0
4	1	1	0

An example rule from this data set is {cereal,eggs}->{milk}. When customers bought cereal or eggs, they also likely bought milk. Obviously, real life data sets contain many more transactions, and rules will need support from many of those transactions to be considered significant.

In order to find the best rules, constraints on measures of significance and interest are used. The most widely used constraints are minimum thresholds on support and confidence, though we will also cover lift and conviction. Let X and Y be item sets, X->Y the association rule, and T the set of transactions.

Support shows how often an itemset appears in the dataset. Let's say the support of X with respect to T is the proportion of transactions t which contain the itemset X. Then, $supp(X) = |\{t \in T; X \subseteq t\}|/|T|$. In the example above, the item set X = {cereal} has a support of 0.5 because it appears in 2/4 transactions.

Confidence shows how often a given rule is found to be true. The confidence value of a rule, X -> Y, with respect to T is the proportion of transactions that include X which also include Y. Then, conf(X->Y) = supp(X∪Y)/supp(X). So for the rule we found above, {cereal,eggs} -> {milk}, it has a confidence of 0.75/0.75 = 1, or the rule is correct 100% of the time.

Lift is the ratio of the observed support to the support if X and Y were independent. The equation can be represented by lift(X->Y) = supp(X∪Y)/(supp(x)×supp(Y)). For the rule above, {cereal,eggs} -> {milk} has a lift of 0.75/(0.75x0.75) = 1.33. If the lift equals one, it means the probability of the two are independent and no rule can be drawn. If the lift is more than one, it shows to the degree the two occurrences are dependent on each other, and means the rules are useful for predicting future outcomes. If the lift is less than 1, then the items are substitute to each other, meaning the occurrence of one item has a negative effect on the occurrence of the other, and vice versa.

Conviction is the ratio of the frequency X occurs without Y. The equation can be represented as conv(X->Y) = (1-supp(Y))/(1-conf(X->Y)). Let's say, for the sake of the example, our rule is {milk} -> {eggs}, it has a conviction of (1-0.5)/(1-0.67) = 1.5. This means our rule would be incorrect 50% more often if the association between milk and eggs was purely random (which, given by our tiny data set, isn't too unlikely).

In order to find significant rules, a minimum support and minimum confidence are placed (other measures such as list or conviction can also be used). First, we will apply a minimum support threshold to find all frequent item-sets in the database. This can be a bit tricky, as the set of all possible item-sets is the powerset of I, the set of n items. This means it grows exponentially, with a size of 2^n-1.

An efficient search for the exponentially large dataset can be created using the downward-closure property of support, which checks that for a frequent itemset, all of its subsets are also frequent. Thus, no infrequent itemset can be a subset of a frequent itemset.

4. Artificial Immune System

Artificial Immune Systems (AIS) focuses on copying computation found in the immune system and implementing that into binary language for computers. It is similar to LCS in that it is biologically inspired, but relates closely to the field of immunology and evolves alongside the field as knowledge technology advances. AIS creates and uses a variety of algorithms solely inspired by processes our immune system takes. Given that AIS is related closely to one of the most complex security systems on the planet, it is extremely popular in the field of cybersecurity.

One such algorithm is the negative selection algorithm, inspired by the positive and negative selection processes that occur during the maturation of T cells in the thymus. It works in the same way that an immune system can react quicker to a disease it has previously encountered. The algorithm is developed to detect anomalies in a set of strings which could be changed by mal programs like viruses. The first step is generating detectors through a procedure called censoring. This splits the protected string into substrings, creating the set P of self-substrings, also known as the training set. The next step is to randomly generate a collection of random strings C. Those who match self-strings in P are classified as normal and deleted, while those that do not match become a member of set R repertoire. Then, the monitoring phase can begin, which continually matches strings from R to P, and non-self-patterns are detected. This can detect abnormalities in things such as network traffic.

In order to match between detectors and the data set, the Euclidean distance formula is used to measure their affinity. If the affinity between the arrays of C and at least one array of P is greater than the affinity threshold, it is rejected, or negatively selected. Otherwise, it is stored in R.

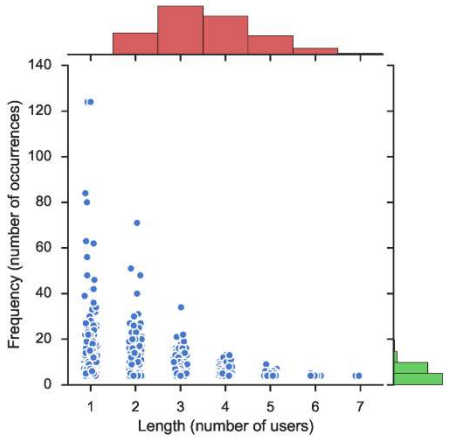
Let C = $\{c_1^k, c_2^k, \dots, c_Q^k\}$ be the set of detectors and P = $\{p_1^i, p_2^i, \dots, p_Q^i\}$ be the set of self data. Then dis(C^k,P^i) = $\sqrt{\sum_{j=1}^{Q} |c_j^k - p_j^i|^2}$.

Another model is the CLONALG algorithm, or Clonal selection, inspired by the clonal selection theory of acquired immunity. The theory is about how B and T lymphocytes improve their response to antigens over time. First, it generates an initial population of antibodies randomly, including the memory population AB_m and the reservoir population AB. Then, you introduce an antigen to the system and find the affinities between it and the antibodies. This will use the Euclidean distance formula above, with AG (antigen) and AB (antibody) instead. Using the formula, find the highest affinity AB* and create a temporary population of its clones AB_c. Then, mutate the clones in the temporary population, recalculate the affinity between the antigen and the mutated clone population, find the antibodies with a higher affinity than the current cell, and create a new antibody as a candidate memory pattern. If the affinity is higher than the current memory pattern, it becomes the new memory pattern. Then, remove the lower affinity antibodies and replace them with randomly generated members. Repeat the entire process until all antigens have been created.

5. Finding Influential Users In Social Media Using Association Rule Learning

This study done by Fredrik Erlandsson, Piotr Brodka, Anton Borg, and Henric Johnson looks at using association rule learning to argue that groups of Facebook users that follow each other can influence the participation of each other, and it is possible to detect influential users that impact the participation of many users. We will only be looking at the first part, that a group can impact a particular user's activity on a social media website.

In this study, they used four measurements we talked about earlier: support, confidence, lift, and conviction.



The first task was an experiment to create frequent item-sets and develop association rules for those sets. The graph shows the results from the experiment. As you can see, as the number of users in the sets increased, the frequency decreased. This is expected to happen. The red represents the distribution of the sets, showing there are more size 2 and 3 user groups than the rest, since more combinations (or groups) of 2 and 3 users exist than single users. The green shows a large density of user collaboration at lower frequencies.

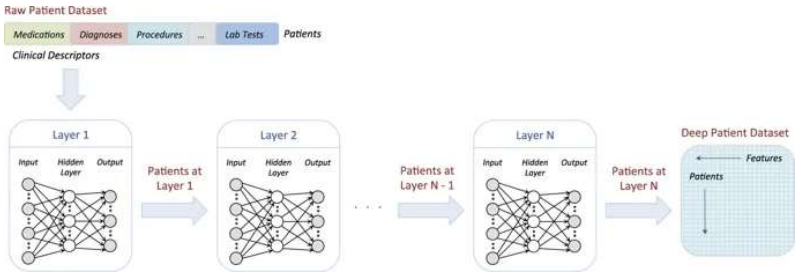
From this, association rules that predicted future participation of users' activities based on others was calculated. Then, for each calculated rule, their support, confidence, lift, and conviction were also calculated. Rules with a high confidence and list, for example $\{u_{580}, u_{861}, u_{1352}, u_{1466}\} \rightarrow \{u_{896}, u_{1291}\}$, means the user set on the left strongly influences the user set on the right. In other words, if the four users on the left all commented or liked something, the users on the right were likely to do the same. Out of 55,166 rules, 4,959 rules had a confidence ≥95%, meaning about 5,000 rules strongly indicated users were affected by each other participating on online social media sites.

As we talked about earlier in association rule learning, this can then be used to create marketing techniques. Had these been grocery items like earlier, a store may strategically place specific items near each other based off of some of these rules with strong confidence and high support. Or, in this case, Facebook could theoretically direct special, high-priced marketing at influential users found by these rules for an increased collateral affect (not that I am claiming they do).

6. Deep Patient (A More Complicated Approach)

Deep Patient is a program designed to utilize electronic health records (EHR) and rule-based machine learning to allow a computer to predict future health problems based off of a patients current record. The experiment used 700,000 patients from the Mount Sinai data warehouse. Previous attempts had been made with supervised learning, but this was aimed at using unsupervised learning broadly with EHR data.

The process used a series of layers to start from raw patient datasets in HER to a Deep Patient dataset. Each layer was designed to produce a higher-level representation of observed patterns in the dataset, based on input received from the previous layer, as shown below.



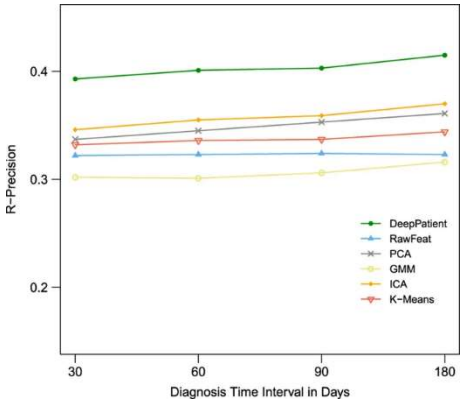
To implement this design, they used a stack of denoising autoencoders (SDA), each independently trained layer by layer. An autoencoder takes a input $x \in [0,1]^d$ and transforms it to a hidden representation $y \in [0,1]^{d'}$ through the mapping equation $y = f_\theta(x) = s(Wx + b)$, where $\theta = \{W,b\}$, s is a transformation called "activation function", W is a weight coefficient matrix d'xd, and b is a bias vector of dimensionality d'. The mapping $f_\theta(x)$ is called the encoder. Then, the representation y is mapped back to the reconstructed vector in input space $z \in [0,1]^d$, such that $y = g_{\theta'}(y) = s(W'y + b')$, where $\theta = \{W',b'\}$ and $W'=W^T$. The mapping $g_{\theta'}(y)$ is called the decoder. The code is intended such that y is a distributed representation that finds the coordinates along the main factors of variation in the dataset, and while training the model, the algorithm finds parameters that try to minimize the difference between x and z, or minimize the reconstruction error $L_H(x,z)$.

Autoencoders have to reconstruct the input from a noisy (random obscurities) set of initial data through a process known as denoising. First, they corrupt the initial input x and obtain a partially destroyed $\tilde{x}$ through stochastic mapping ($\tilde{x} \sim q_D(\tilde{x}|x)$). $\tilde{x}$ is then mapped to a hidden code $y = f_\theta(\tilde{x})$ and then to a decoded representation z. They then implemented the input corruption using the masking noise algorithm, which is turning off components considered

missing (or replacing their value by a default value). This takes place of missing components in EHRs, such as missing medications or diagnoses. The masking noise algorithm takes a fraction y of the elements of x chosen stochastically is set to 0. Then, through denoising the autoencoders, these created blanks are trained to be filled in.

In order to minimize the reconstruction error, θ and θ' are optimized through a training dataset through the equation $\theta,\theta'^{*}=argmin_{\theta,\theta'}\, L(x,z) = argmin_{\theta,\theta'} \frac{1}{N}\sum_{i=1}^{N} L(x^{(i)}, y^{(i)})$, where L(x,z) is the reconstruction error (or loss function) and N is the number of patients. Using the reconstruction cross-entropy function as the reconstruction error function, $L_H(x,z) = -\sum_{k=1}^{d}[x_k log z_k + (1-x_k)\log(1-z_k)]$. Optimization is carried out through stochastic gradient decent, which goes through smaller subsets of the patient set and changes parameters in the opposing direction of the gradient of the reconstruction error function to minimize the error. The learned encoding function $f_\theta(x)$ is applied, and the input x is cleaned up and the resulting output y is the distributed representation used by the next autoencoder in the SDA.

Once the framework was created, they evaluated by disease and by patient, or disease classification and patient disease tagging respectively. Evaluating by disease is measuring how well Deep Patient predicted a patient developing new diseases. Evaluation by patient looked at how well Deep Patient performed at a patient specific level. In both of these scenarios, Deep Patient drastically outperformed other methods across all diagnosis time intervals.



Deep Patient was the first step towards the next generation of predictive clinical systems. By implementing rule-based machine learning, it was able to find rules and predict diseases that outranked many doctors and methods. Unfortunately, it is still limited by size of data sets, and it performed poorly with some diseases. However, it goes to show the potential RBML has in future technology.

7. Managing Energy in a Virtual Power Plant Using Learning Classifier Systems

Many renewable power planets, such as wind or solar, are unstable and violatile throughout the year. This study focuses on using LCS to manage a renewable powerplant with real-world statistics to deal with a variety of factors like demand, availability, storage, etc.

The powerplant is modeled by the central smart control, three sources of energy being the wind farm, reserve plant, and energy storage, and finally consumers. The total power of the system at time t is $n(t) = \sum_i e_i(t)$, the sum of all producers with $e_p(t)\geq 0$ and the sume of all consumers with $e_c(t)\geq 0$. We assume every $c\in C$, the set of consumers, consumes a certain amount of energy $e_c(t)$ at time t. The basis of the energy consumption reflects real data by EirGrid, an Irish energy company, represented by $e_{EG}(t)$. To simulate fluctuations that occur and the number of consumer modules, they modeled noise that could be added by $e_c(t) = -e_{EG}(t)*(1+\gamma_1*N(0,1))$, controlling the noise strength through the variable γ_1. Wind energy was controlled by a similar equation, $e_w(t) = -e_{WIND}(t)*(1+\gamma_2*N(0,1))$, controlling the noise strength through the

variable γ_2. Wind data was also taken from EirGrid. The energy storage system $s \in S$ will have capacity $c_s(t)$ and a max of $(c_s)_{max}$. It can load energy if $n(t)>0$ and deliver if $n<0$, increasing or decreasing by Δ_s. Energy lost over time will use a similar equation to that above, with γ_3 as the noise strength variable. The reserve plant r can deliver energy $e_r(t)>0$ at time t, and increase or decrease production by Δ_r.

The study uses a Pittsburgh-style LCS. Rules will be interpreted as r_n: IF x_n, THEN α_n. s_t will be the state of the power plant, with T parts, corresponding to the total power of the system $s_1=n(t)$ and the predicted power for the next $i=1,\ldots,T-1$ steps $s_i = n(t+i)$. Similarly, there is a set of rules r_j with conditional part x_j for a minimal distance $d(x_j,s_t)$. For distance d, they use the Manhattan distance ($d = |x_1-x_2|+|y_1-y_2|$). For each future step t+i, each j performs α_{ij}, i.e. each α_n consists of T*A actions.

All rules are randomly initialized at the beginning, using a $(\mu + \lambda)$ population scheme. In each generation, λ solutions are created by selecting one of the μ parent solutions and using mutation. Each rule will be randomly selected with probability σ_1 and mutated with probability σ_2. For mutation, each component will be randomly chosen from the set of possible values, for example actions $A = \{-1,0,1\}$. In each run of the LCS, a new set of λ solutions are created with respect to the above process. Then, each rule set is tested with an entire run of the power plant. The goal is to get a rule set with a quality measure of f, which looks for the minimization of n. Once a specific f is reached, the process terminates.

In the end of the experiment, the LCS was able to evolve rules that successfully managed the virtual power plant, specifically rules that managed the storage and reserve plant in respect to the demand and available wind throughout the month of February. They were also able to test the rule sets in unknown situations, granted with some aid to deal with certain situations not found in the simulations resulting in no rules being evolved to handle such situations. The study aimed to show the potential of the model and demonstrate how LCS was able to solve it, and through the experiment, supported their claims.

8. Conclusion

In this paper, we went over what rule-based machine learning is, different types of RBML including learning classifier systems, artificial immune systems, and association rule learning systems, and examples of real life implementations of RBML.

Rule-based machine learning is a wide field in machine learning with a lot of potential as technology grows. Though the concepts are a bit old, new technology has allowed these concepts to be effectively applied in real life scenarios. RBML holds a lot of potential in marketplaces such as cybersecurity, marketing, and medicine, though I suspect as machine learning as a whole grows, we will see a lot more RBML be introduced into a variety of fields.