Data Mining Using Pseudo-Cellular Automata with Update Rules based on Local Gradients

Branden Keck, Johns Hopkins University

1 Introduction

Classification is a type of supervised learning which is used to analyze data based on its features. Classification algorithms categorize datapoints and often make decisions based on those categorizations. Among the popular data classification techniques are Neural Networks, K Nearest Neighbors, Naïve Bayes Algorithm, and Support Vector Machines [1]. These methods are applicable to a wide array of problems in a variety of fields including analysis of text, multimedia, social networks, and biological data [2]. Therefore, these methods are valueable data mining tools.

One technique that is not commonly used for data classification is cellular automation. Cellular automation simulations could be used to "grow" classification regions, which could prove helpful when data can only be separated into categories using complicated, nonlinear boundaries. A simple form of using cellular automation as a data mining tool was introduced in a paper by Fawcett [3] in which local decisions are shown to viably define overall classification for specific two-dimensional datasets. However, the design of this cellular automation is kept simple, which can lead to inaccuracies when there is not a very clear separation between datapoints that belong to different classes. Additionally, Multiple Attractor Cellular Automata (MACA) have gained attention as potential classifiers for bioinformatics problems. This method relies on genetic algorthims to create a system of states that can be used to categorize data [4]. However, this process involves many components with an abundant amount of underlying theory. In this paper, a pseudo-cellular automation approach for classifying data is purposed which seeks to bridge the gap between classifier accuracy, practical implementation and complexity of theory.

1.1 Cellular Automata

There are many ways to construct a cellular automation. However, all cellular automata follow the same basic principals: they are discrete in both space and time, they are homogenous in space and time, and they are local in their interactions [5]. A simple form of cellular automation can be conducted on a two-dimensional grid, with each "block" in the grid representing a cell. Cells are often represented as being in one of two states: dead ("0") or alive ("1").

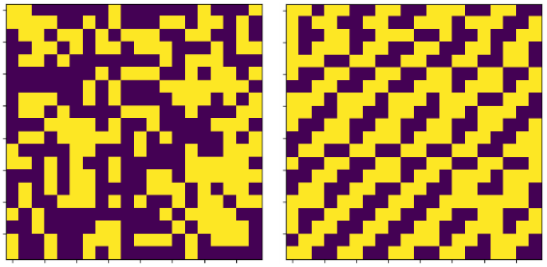

Fig. 1: Two examples of cell grids – purple cells are dead ("0") and yellow cells are alive ("1").

With each passing time step, the state of each cell (0 or 1) is updated. The "homogenous" property of cellular automata means that the same update rule is applied to each cell. Additionally, updates are applied to all cells simulataneously for a given time step [5].

Update rules for a given cell are usually based on the states of "neighboring" cells. A neighboring cell can be defined as any cell within a certain "radius" of the cell in question. Additionally, it is important to specify whether the radius applies only along the axes of the cell space or if it can be applied diagonally. These specifications are called the "von Neumann neighborhood" and "Moore neighborhood", respectively. In two dimentions, the most commonly used neighborhoods are the radius-1 von Neumann neighborhood and the radius-1 Moore neighborhood, which are shown in the below figure.

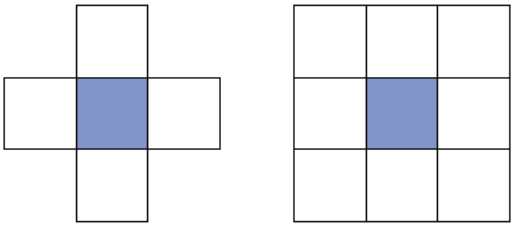

Fig. 2: Illustration of the von Neumann neighborhood (left) and the Moore neighborhood (right)

One popular example of cellular automation that can be used to better explain update rules and neighborhoods is Conway's Game of Life. This game uses a radius-1 Moore neighborhood with the following update rules [6]:

1. A cell is "born" (a dead cell becomes alive) when it has exactly three living neighbors
2. A living cell dies if it has fewer than two living neighbors or more than three living neighbors
3. A living cell remains alive if it has two or three alive neighbors

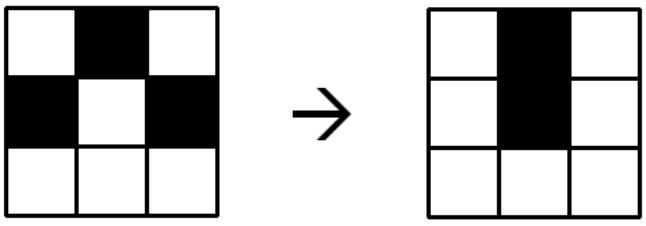

Fig. 3: Example of an update to a 3-by-3 grid of cells based on the rules of Conway's Game of Life – black cells are alive and white cells are dead.

Establishing a set of rules like those used in Conway's Game of Life creates an interesting set of properties. Additionally, these properties can lead to global phenomena, which is interesting because the rules for each update of a cell depend only on local conditions.

1.2 Mathematical Representation of Cellular Automata

To better explain the rules of Cellular Automata and their resulting properties, some mathematical definitions are useful. The same notation used by Kari [5] will be used here. A cellular automation is defined as follows:

$$A = (d, S, N, f)$$

d represents the dimension, S is a finite set of states, N is a vector of m neighbors to be used in the update rule and f is the local update function which produces a mapping of form:

$$f: S^m \to S$$

Therefore, in the case of Conways Game, the cellular automation is defined with the following properties:

$$d = 2$$
$$S = \{0, 1\}$$
$$N = (\overrightarrow{n_1}, \dots, \overrightarrow{n_8})$$

This is because the grid used is two-dimensional, the only possible states are dead ("0") or alive ("1") and eight neighboring cells are being consided in the update function.

It should also be noted that a global transition function can be defined as:

$$G: S^{\mathbb{Z}^d} \to S^{\mathbb{Z}^d}$$

This function is the "main object of study" [5] being that it is used to define the changing configuration of the entire cell space. Additionally, the configuration of the cell space at time t is often written as $G^t(c)$ where c is the starting configuration.

2 METHODS

A model was developed in Python version 3.5.2. The goal of this model was to first recreate the classification method introduced by Fawcett [3], then to extend this method to cases where cells do not necessarily exhibit a discrete state but rather a mixture of states. Thus, a "true" cellular automation is not being used, but rather a pseudo-cellular automation. Extention of the cellular automation-based classification method to the n-dimentional case was also a primary goal. A secondary goal was to fit a classifier function to the class boundaries that were produced by the simulation.

2.1 Data Sets

2-dimentional sets of data were created by drawing random samples from multivariate normal distributions. This was accomplished using builtin functionality of the numpy python library. Data for each class was drawn from a Gaussian distribution with arbitrary – but unique – mean vector and covariance matrix. Additionally, each class was specified to have the same number of samples.

Another dataset was created for better comparison between this method and the method introduced by Fawcett [3]. This data set contains two classes separated by the boundary $y = x^2$. To produce this data, random samples were drawn from the space of (x, y) such that $x \in (-2, 2)$ and $y \in (-1, 3)$. Then, classes were assigned based on wether each pair of random draws satisfied $y < x^2$ or $y > x^2$. An example of data generated in this way can be seen in the figure below.

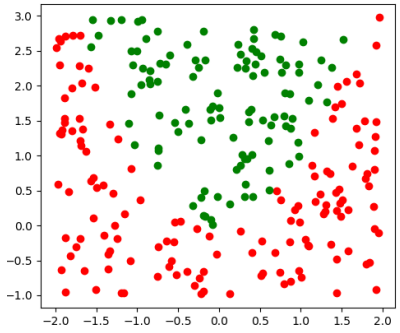

Fig. 4: Parabolic data set. Points above $y=x^2$ are assigned to the "green" class and points below $y=x^2$ are assigned to the "red" class.

The popular Iris flower dataset was the last set of data used to test the model. This set of data can be directly imported using the sklearn python library. It was selected because it can be used to test the performance of the model on 4-dimensional (non-visualizable) data and it is easy to import and manipulate.

2.2 Defining a Cell Space

The first step in constructing this model was to determine how to map a standard set of data to a "cell space". To do this, an initialization function was developed that creates "bins" for each dimention of the data. The number of bins for each dimension is specified beforehand and evenly spaced values based on the maximum and minimum data values (with some additional margin) are assigned to each bin. For example, imagine that all points in a 2-dimensional (x, y) dataset fall between x-values $x = 2$ and $x = 4$ and fall between y-values $y = -1$ and $y = 3$. These data points could be divided into two bins in the x-direction and two bins in the y-direction as follows:

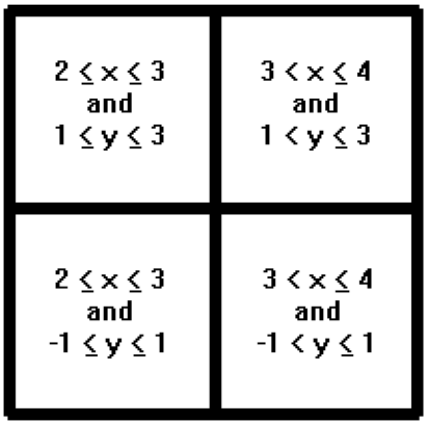

Fig. 5: Example bin values for division of two-dimesional space into four bins – two along the x-axis and two along the y-axis

2.3 Initialization and Update Rules

After establishing cell assignments, the properties of the cellular automation were defined. To do so, the traditional properites of cellular automata were neglected. There are a variety of reasons for this.

Using discrete states for the cells implies that each cell belongs exclusively to one class. For the purpose of classification, it makes sense to define two properties for each cell – the first property being "species" which defines the class to which the cell belongs and the second being "fitness" which defines the degree to which the cell belongs to that class. The following notation will be used in indicate these properties:

$$S_s \in \{1, \dots, k\}$$
$$S_f \in [0,1]$$

Here, S_s represents the "species" whereas S_f denotes the "fitness". Given this second property, it is possible for cells to form a "gradient" of membership to each class over the cell space. This is akin to "fuzziness" in the Fuzzy K Means algorithm wherein introducing "degrees" of membership to each class improves overall clustering performance.

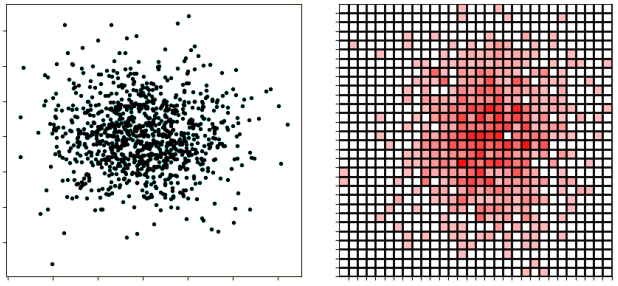

Fig. 6: Example of Gaussian data (left) organized into bins (right), where darker bin color implies more datapoints within that bin, which translates to a higher fitness in the cell space. Note that the data transforms in shape based on the size of each bin along the x-axis and y-axis.

These properties are used in the update rules that were defined for the cellular automation. A unique update rule is used to initialize the system. To start the simulation, cells are grown in a "Moore-like" fashion using the principal of global repulsion between cells of the same species. To achieve this "repulsion" effect, Coulomb's law is used. Coulombs law for the total force on a charge is represented by the equation below.

$$\vec{F}_i = \sum_{i \neq j} \frac{k q_i q_j}{\left\| \vec{r}_i - \vec{r}_j \right\|} \vec{r}_i$$

Here, k is a constant, q_i is the charge of the particle for which net force is being calculated, $\vec{r}_i$ is the position of this particle, and q_j and $\vec{r}_j$ are the charges and positions of the other particles, $i \neq j$. This idea can be applied to an initialization step for a cellular automation by defining the respective fitnesses of cell i and cell j as the two charges. In this scenario, the constant k is dropped from the equation and the positions of the charges, $\vec{r}_i$ and $\vec{r}_j$ are replaced with the n-dimensional cell indices. The final "repulsion vector" is also normalized because it is being used for the purposes of directionality, not magnitude. An example of this initialization step for a system involving two cells can be seen in the following figure. This figure also helps visualize the "Moore-like" neighborhood to which the update rule is applied.

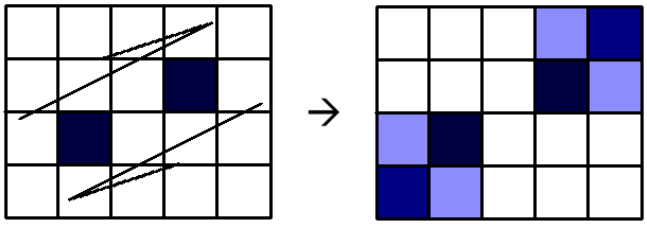

Fig. 7: Example of a global repulsion-based function applied in two dimensions. Darker color in a cell implies a higher fitness. "Force" vectors calculated using a forumal analogous to Coulomb's law. Then, the fitness of the cells is propaged to cells within their Moore neighborhood in the direction of these vectors.

It is this important to note that this initial update rule (along with all following update rules) is applied such that each cell is used to define the state of its neighbors instead of a cell's neighbors defining the state of the cell. The reason for this global initialization rule is to ensure that all cells have von Neumann neighbors which are configured in a way that is ideal for the standard update steps. The standard update steps use "local" repulsion rules where the "repulsion vector" is calculated in the same way as it is for the "global" initialization. However, there are several differences between the standard update steps and the initialization step. These differences are as follows:

1. Only neighboring cells are used to define the "repulsion" vector that is applied
2. A von Neumann neighborhood is used
3. A gradient function is applied to determine the amount of fitness that is propagated to neighboring cells

A von Neumann neighborhood is used due to the much larger number of computations required for Moore neighborhood calculations in higher dimentions. For the von Neumann case, the number of neighbors to consider is $2d$ where "d" is the number of dimentions. The Moore analog requires the consideration of $3^d - 1$ neighbors, which is considerably more computationally taxing for large values of "d".

The gradient that is applied ensures that a "center of mass" develops within each group of cells where the fitness of cells is highest near this center and lowest near the class boundaries. This is an improvement upon simply growing cells based on a von Neumann or Moore neighborhood because it produces smoother boundaries that "preserve" the shape of the training data and the resulting map of "fitnesses" is analogous to an estimated probability distribution. The overall update function – involving this gradient, local repulsion rules, and the von Neumann neighborhood – can be written qualitatively in the following way for a single neighbor of the cell in question:

$$fit_{n+1} = F_k\left(\frac{fit_n}{fit_{n-1}}\right)$$

In this (pseudo)-equation, fit_{n+1} represents the fitness to be "added" to a neighboring cell (cell $n+1$) given the fitness, fit_n, of the current cell (cell n) and the fitness fit_{n-1} of the cell on the opposite side of cell n – cell $n-1$ – along the dimension k where F_k represents the proportion of the repulsion vector that is along the k-axis. This can be visualized more easily than it can be explained. Therefore, the following figure is used to display the effects of this update rule in one dimension:

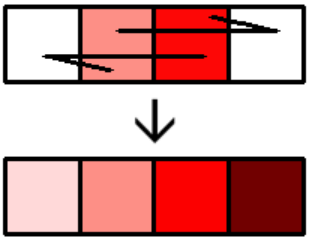

Fig. 8: Example of a local gradient function applied in one dimension. Darker color in a cell implies a higher fitness. The difference in fitness between two neighboring cells is used to define the fitness on either side of the two cells.

The overall effects of applying a gradient in this way can be observed from a global perspective in the following figure.

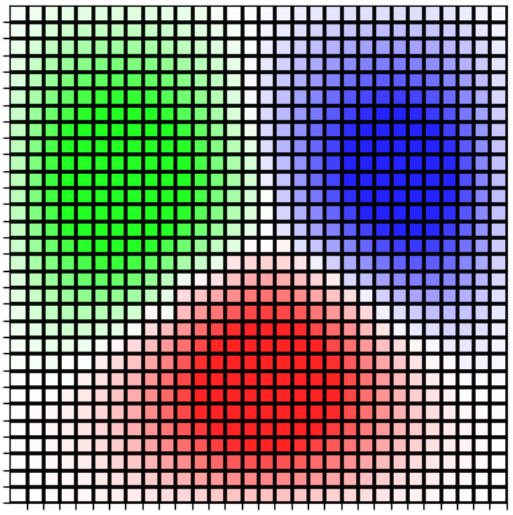

Fig. 9: Example of gradient boundaries between three classes drawn from separate Gaussian distributions. Darker color in a cell implies a higher fitness.

2.3 Cell Conflicts

When multiple cells attempt to populate a space on the grid, a separate function is used to define the outcome. This function totalizes the fitness of all cells of a given species (including the cell that may have already been occupying the space). Then, the species with the maximum fitness value is given control of the space. The resulting cell has a fitness value that is the total of all cells of its species minus the total fitness of cells of other species that attempted to occupy the space. This can be decribed as follows:

$$fit_n = 2 * fit_{max} - fit_{total}$$

Here, fit_n represents the fitness of the resulting cell, fit_{max} represents the maximum total fitness for a given species attempting to take control of the cell, and fit_{total} represents the total fitness of all cells attempting to occupy the cell. It is important to note that if all competing cells are of the same species, the fitness of the resulting cell is simply the sum of their individual fitnesses.

2.4 A Note on Dimensionality

Although this procress is easiest to visualize when applied to 2-dimensional data, it can be extended to any n-dimensional feature space. The python model was specifically designed to iterate over a cell space with any number of dimensions and any size of data within those dimensions. In this way, the results cannot be visualized, but any data point can be used to locate the bin to which is belongs and extract the properties of that bin.

2.5 Use in Data Mining

The cellular automation that results from the rules applied here can be used for classification purposes. By conducting a simulation using k different "species"of cells for a training data set containing k classes of data, natural boundaries between these classes can be drawn. The results of the cellular automation along with the bins used to map the dataset to the cell space can be used to develop class predictions for data points outside of the training set. This is done simply by determining which "species" is assigned to the space corresponding to the bins within which the datapoints fall.

Additionally, the boundaries that develop between species could be extracted and used to fit polynomials for classification purposes. An example of this can be seen in the following figure in which a randomly generated cell boundary is fitted with polynomials of varying degree in the cell space. Given that boundary fitting is a numerical analysis problem that is beyond the scope of this paper, the built-in functionality of numpy's polyfit function is leveraged for this task. The python code used to create this example can be found in "boundary_fitting.py".

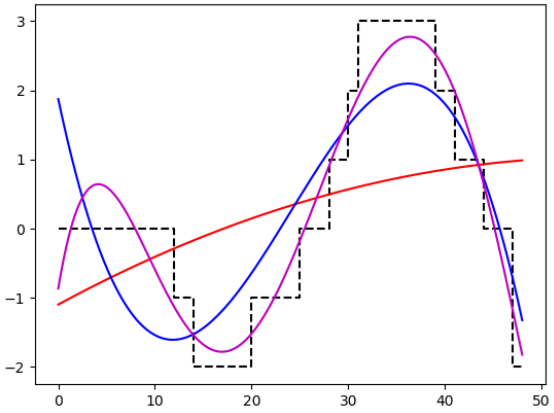

Fig. 10: Numpy polyfit function applied to random cell boundaries. The red line, blue line, and magenta line represent the 2nd degree, 3rd degree, and 5th degree fitted polynomials respectively

3 RESULTS

Several experiments were conducted to determine the viability of this classification method. The goal of these experiments was to assess the performance of this method as a classifier when compared to other cellular automata, compared to other classification methods, and given different data sizes.

3.1 Comparison to Simple Cell Growth via von Neumann and Moore Neighborhoods

Side-by-side simulations using the "gradient" cell growth method and simple von Neumann and Moore neighborhood growth methods were conducted to determine if added update rule complexity is beneficial in terms of classification results. In this context, "simple" growth means that the species of a cell is determined by the species found most frequently in the neighborhood of that cell. Relevant python files used for these tests are "biosystem.py", "vonneumann_biosystem.py", "moore_biosystem.py", and "run_2d_comparision.py".

Many trials were conducted, some of which can be seen in Appendix 1. All trials used 300 datapoints drawn in equal amounts from three Gaussian distributions. Additionally, a 20-by-20 grid of cells was used and the simulation was conducted for 100 growth cycles. The following figure shows a specific simulation for which the "gradient" method seemed to exhibit better performance than simple cell growth:

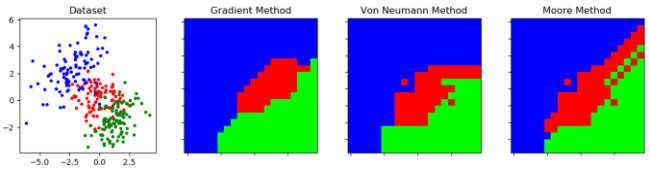

Fig. 11: Specific example of a simulation in which the "gradient" method proposed in this paper outperforms standard von Neumann and Moore cell growth from a classification perspective. Simulations were conducted with 100 datapoints per class, a 20-by-20 grid of cells, and 100 growth cycles.

From a qualitative perspective, there are several factors that indicate better performance for the "gradient" method. As shown in Figure 11 and Appendix 1, overlap between data of different classes causes the simple growth methods to produce cells that are outside of the "boundary" for their class. This is also true of the "gradient" method but to a much smaller degree. Additionally, it can be seen that the "gradient" method produces smoother boundaries which appear to be more optimal for classification and better suited for fitting a classification function to the resulting cell boundary.

3.2 Accuracy Improvements with Increase in Data Size

As is the case with most classification methods, the performance of cellular automation as a classifier improves with an increasing number of observations in the training set. To show the effect of training set size on performance of the cellular automation classifier, data was randomly drawn from a rectangular space, where $y > x^2$ implies that a datapoint belongs to the "green" class and $y < x^2$ implies that the datapoint belongs to the "red" class as previously shown in Figure 4. Simulations were conducted for 10, 50, 100, and 1000 datapoints in the training set. Relevant code for these simulations can be found in "biosystem.py" and "run_2d_animation.py", wherein the cell growth for this parabolic dataset is animated. Screenshots from each of these animations can be seen in the following figure:

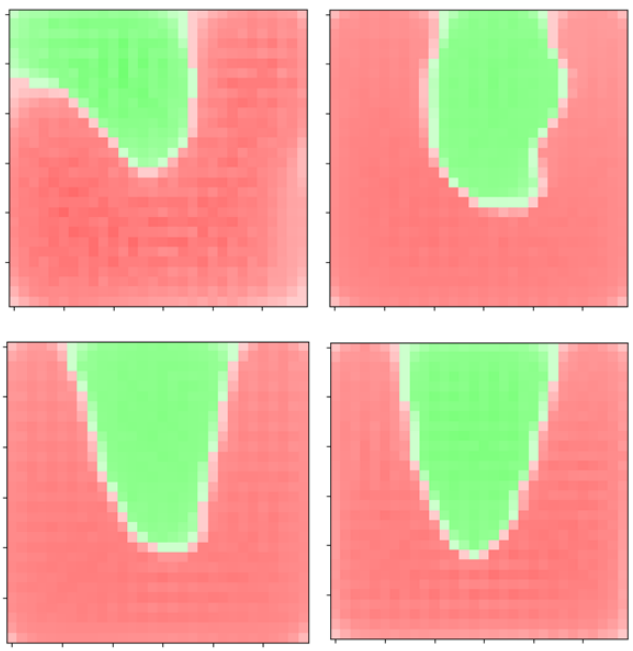

Fig. 12: Results of attempts to grow a parabolic classifier. For randomly selected data (number of data points in each class not necessarily equal) the plots correspond to the following numbers of starting datapoints (between both classes): 10 (upper-left), 50 (upper-right), 100 (lower-left), 1000 (lower-right)

This observed improvement in performance is notable because the boundary between the classes is almost a perfect recreation of the actual class boundary when a large dataset is used to train the model. Clearly 1000 datapoints is excessive for a training set of data. However, it can be seen from the above images that this method produces reasonably shaped cell boundaries even for very low amounts of available input data. As the amount of training data grows, the improvement in accuracy is substantial as can be seen in the third image, which shows a very accurate cell boundary for only having 100 total observations between the two classes.

3.3 Predictions with the Iris Flower Dataset

The well-known Iris Flower dataset was used to test the model's ability to classify multidimensional data. The Iris dataset contains observations with four feature vectors. A set of Iris Flower data was imported using the sklearn python library. This data contains 150 samples, which are evenly divided amoung 3 classes. To test the validity of this model, a cellular automation was trained using subsets of this data of varying size and then used to predict the classes of the remaining data. These two subsets of data will be refered to as the "Training Data" and the "Testing Data".

The sklearn library also contains built-in functionality for Support Vector Machine (SVM) and Decision Tree models. This built-in functionality is leveraged to compare the accuracy of these models with the Cellular Automata model purposed in this paper. The SVM and Decision Tree models are trained with the same set of "Training Data" and then tested with the same set of "Testing Data" as the Cellular Automata Model. The results of these simulations for random subsets of the Iris dataset can be seen in the following table:

Training Set Size	Test Set Size	CA Accuracy	SVM Accuracy	Tree Accuracy
15	135	0.689	0.963	0.881
30	120	0.950	0.967	0.950
45	105	0.924	0.943	0.943
60	90	0.944	0.978	0.978
75	75	0.933	0.920	0.960

Fig. 13: Results of classification on random subsets of the Iris Flower data set using 50 iterations and a 12-by-12-by-12-by-12 cell space. SVM Accuracy and Decision Tree Accuracy determined via built-in sklearn functionality. Accuracy of the methods is measured on a scale of 0 to 1.

The cellular automation method (abbreviated CA in the figure above) was conducted using a 12-by-12-by-12-by-12 size cell space and 50 growth cycles. Relevant code for these trials can be found in "run_main.py" and "biosystem.py". It can be seen that the accuracy of this method is very low in the first trial due to the small size of the Training set. However, in subsequent trials the accuracy of the cellular automation method is comparable to that of the SVM and Decision Tree methods. This is a notable achievement given that the sklearn library contains very optimized and accurate implementations of these methods.

Additionally, it should be noted that increase in training data size is not the only way to improve the accuracy of the cellular automation method. Increasing the size of the cell space and the number of growth cycles can also have a very positive impact. To prove this, a single simulation was run for a Training Data size of 15 and a Testing Data size of 135 with a 20-by-20-by-20-by-20 sized cell space and 100 growth cycles. The result was a drastically improved accuracy of 0.942. For the same sets of data, the SVM and Decision Tree methods achieved accuracies of 0.958 and 0.942, respectively. This shows that the difficult in creating a classifier with a small set of training data can be remedied by increasing the number of cells and cycles for a cellular automation. However, it is important to note that this increase in accuracy come with a substantial increase in computation time. This much larger simulation took slightly more than an hour and ten minutes to complete (on a Windows 8 machine with intel core i7 processor and 8 GB RAM).

3.4 Moving Forward

It has been show that cellular automata methods can be used to classify data with relative accuracy. However, there is much more that could be explored with regards to this topic. The "fitness" parameters used in this model could be used to create estimates of probability ditributions for each class by normalizing the fitness of each species so that the cell space contains values between 0 and 1 at each cell index.

In addition, the method of fitting functions to the resulting cell boundary, which was briefly mentioned in Section 2.5, could be used to gain insight to the nature of the data being classified. If a method for determining the terms of the function is implemented correctly, the cell boundaries produced by this model could be used to defined classifiers in mathematical terms and not only in terms of "bin" values in the cell space.

4 Conclusion

A classification method which uses cellular automation to "grow" class boundaries was implemented in the python programming language. The viability of this method was assessed by comparing simulation results to results from other cellular automata and results from other data mining methods. It is clear that this method has potential given the relative accuracy of this naïve implementation and given the potential for further analysis of the resulting cell boundary and cell fitnesses. Efficiency of the method needs improvement. However, the code could be made significantly faster by designing functions which limit the number of computations to only those that are necessary and structuring the model in a way that would make it possible to implement methods like Fast Matrix Multiplication to perform many calculations in fewer, more efficient steps. This model is currently a good starting point for a method that could be explored much further.

REFERENCES

[1] Sagar S. Nikam. "A Comparative Study of Classification Techniques in Data Mining Algorithms." *Oriental Journal of Computer Science and Technology*. 10 April 2015.

[2] Charu C. Aggarwal. *Data Classification: Algorithms and Applications*. CRC Press: Taylor and Francis Group, 2015.

[3] Tom Fawcett. "Data mining with cellular automata." *Association for Computing Machinery's Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD) Explorations*. 2008.

[4] N. Ganguly, et al. "Evolving cellular automata as pattern classifier." *Cellular Automata, Lecture Notes in Computer Science, Vol. 2493*. 2002.

[5] Jarkko Kari. "Cellular Automata." 2013.

[6] Hector Alfaro, et. al. "Generating Interesting Patterns in Conway's Game of Life Through a Genetic Algorithm." 2009.

Appendix 1 – Comparison of Cellular Automation Methods for Different Data Sets

Training Data	Gradient Model Performance	von Neumann Model Performance	Moore Model Performance