Faculty of **Applied Computer Sciences and Biosciences**

MASTER THESIS

Cognitive Bias-Powered GLVQ: Illogical Machines

Author: **Mert Saruhan**

1st Examiner: Prof. Dr. rer. nat. habil. Thomas Villmann
2nd Examiner: Dr. rer. nat. Marika Kaden

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Applied Mathematics for Network and Data Sciences

Mittweida University of Applied Sciences
Applied Mathematics for Network and Data Sciences
Mittweida, Germany

Submission: 08 October 2023

Declaration

This thesis is submitted in partial fulfillment of the requirements for the degree of Master of Science in Applied Mathematics for Network and Data Sciences at the Mittweida University of Applied Sciences (Hochschule Mittweida).

I declare that this work has been completed according to the guidelines established by the faculty and has not been submitted for any other purpose.

Mittweida, Germany

08 October 2023

Mert Saruhan 

Abstract

In this paper, we conduct experiments to optimize the learning rates for the Generalized Learning Vector Quantization (GLVQ) model. Our approach leverages insights from cognitive science rooted in the profound intricacies of human thinking. Recognizing that human-like thinking has propelled humankind to its current state, we explore the applicability of cognitive science principles in enhancing machine learning.

Prior research has demonstrated promising results when applying learning rate methods inspired by cognitive science to Learning Vector Quantization (LVQ) models. In this study, we extend this approach to GLVQ models. Specifically, we examine five distinct cognitive science-inspired GLVQ variants: Conditional Probability (CP), Dual Factor Heuristic (DFH), Middle Symmetry (MS), Loose Symmetry (LS), and Loose Symmetry with Rarity (LSR).

Our experiments involve a comprehensive analysis of the performance of these cognitive science-derived learning rate techniques across various datasets, aiming to identify optimal settings and variants of cognitive science GLVQ model training. Through this research, we seek to unlock new avenues for enhancing the learning process in machine learning models by drawing inspiration from the rich complexities of human cognition.

Keywords: machine learning, GLVQ, cognitive science, cognitive bias, learning rate optimization, optimizers, human-like learning, Conditional Probability (CP), Dual Factor Heuristic (DFH), Middle Symmetry (MS), Loose Symmetry (LS), Loose Symmetry with Rarity (LSR).

Acknowledgments

Thanks to

my family and my friends,
for supporting me in my life
and helping me to bring myself where I am right now.

Thanks to

Prof. Dr. Thomas Villmann,
for his help and guidance throughout my studies and on my thesis.

Contents

List of Figures

List of Tables

Abbreviations

LVQ: Learning Vector Quantization. 1

GLVQ: Generalized LVQ. 1

OLVQ: Optimized-Learning-Rate LVQ. 2

OGLVQ: Optimized-Learning-Rate GLVQ. 2

CGLVQ: Cognitive GLVQ. 2

CP: Conditional Probability. 1

DFH: Dual Factor Heuristic. 1

MS: Middle Symmetry. 1

LS: Loose Symmetry. 1

LSR: Loose Symmetry with Rarity. 1

Symbols

- $\mathbb{R}$: The set of real numbers. 3
- $\mathbb{C}$: The set of complex numbers. 3
- i: Imaginary unit ($i^2 = -1$). 3
- $\mathbb{N}$: The set of natural numbers. 4
- $\Delta(t)$: Small time step in a discrete case. 4
- x_i: ith element (feature) of sample $\mathbb{x}$. 5
- $\mathbb{x}$: Feature array of a sample. 5
- $*$: Cross-correlation operator. 6
- $\otimes$: Convolution operator. 6
- #: Means "number of". 15
- $\mathbb{x}^c$: Feature array of complex valued sample. 20
- ω: Feature array of a prototype. 20
- $\mathbb{x}_i$: Feature array of the ith sample. 21
- y_i: ith element (feature) of sample ω. 20
- $d(\mathbb{x}, \omega_j)$: Distance metric between feature arrays of $\mathbb{x}$ and ω_j. 20
- $\epsilon(t)$: Learning rate at time t. 20
- $\epsilon(0)$: Initial ($t = 0$) learning rate. 20
- ϵ_i: Local learning rate of ith prototype. 20

- ω^+: Feature array of the winner (closest) prototype with the same label as the sample. 21
- ω^-: Feature array of the winner (closest) prototype with a different label as the sample. 21
- $d^+(\mathbb{x})$: Distance metric between $\mathbb{x}$ and ω^+. 21
- $d^-(\mathbb{x})$: Distance metric between $\mathbb{x}$ and ω^-. 21
- ϵ^+: Local learning rate of winner prototype ω^+. 21
- ϵ^-: Local learning rate of winner prototype ω^-. 21
- ∂: Partial derivative. 21
- σ: Sigmoid function. 21
- ω_i: Feature array of ith prototype. 22
- $L(\mathbb{x})$: label of sample $\mathbb{x}$. 24
- $\omega^{\mathbb{x}}$: Feature array of the winner prototype of sample $\mathbb{x}$. 24
- $R^{\mathrm{XX}(q|p)}$: Causal relationship between p and q under given cognitive science model XX. 26
- R_i: Causal relationship of ω_i under chosen cognitive science model. 28

Chapter 1

Introduction

One of the main aspects of machine learning methods is the implementation of learning rates. Selecting the correct learning rate method is a big problem since different learning rate methods change learning rates during the training differently; hence, the same model learns differently. This paper uses different learning rate optimizer methods implemented from cognitive bias methods for GLVQ. The cognitive (science) learning rate optimizers have been researched with one of the sub-models of the LVQ model; however, not with the GLVQ model. To uncover the performance of the learning rate methods with GLVQ, in this paper, we investigate the learning rate changes during the training using some datasets that offered open-source and extra datasets we created.

LVQ is a prototype-based machine learning method that Kohonen (1995) introduced in his work "Self-Organizing Maps" [1]. LVQ is a computing-friendly machine learning model. According to Kohonen (1990), in another work, LVQ has similar accuracy rates to Neural Networks with smaller computing power [2]. Instead of weight vectors such as Neural Network algorithms, LVQ, and branches of LVQ, use part of the data to train the model for classification [1]. LVQ provides a more human-like learning system than Neural Network provides. We use the GLVQ model in this paper.

There are several learning rate methods introduced throughout the beginning of LVQ. One of them we use is optimized GLVQ (OGLVQ). Changing any LVQ model to an optimized version is mentioned in the "Self-Organizing Maps" book by Kohonen (1995) [1]. We show how to optimize GLVQ to OGLVQ in our paper while mentioning the GLVQ method.

Cognitive learning rate optimizers have been mentioned by Takahashi et al. (2010) in the paper "Cognitive Symmetry: Illogical but Rational Biases"; these cognitive learning rate optimizers are CP (Conditional Probability), DP (Contingency Model), DFH (Dual Factor Heuristic), RS (Rigidly Symmetric), MS (Middle Symmetry), LS (Loose Symmetry), and LSR (Loose Symmetry with Rarity) [3, 4]. However, the research lacks a learning rate analysis. The methods CP, DFH, MS, LS, and LSR show high performance (with $> .9$ determination coefficients) on *human* data from the paper "Contributions of specific cell information to judgments of interevent contingency" by Wasserman et al. (1990) [5] done by Takahashi et al. (2010) [3]. These cognitive learning rate methods are valuable for further research, which we do in this paper.

In addition to CP, LS from cognitive learning rate methods, eLS (enhanced loose symmetric) introduced and compared performance against famous machine learning classification methods such as neural networks (NN), support vector machine (SVM), random forest (RF), and logistic regression (LR) in the paper "A machine learning model with

human cognitive biases capable of learning from small and biased datasets" (Taniguchi et al., 2018) [6]. The cognitive learning rate methods are implemented under the Naïve Bias model. The results include accuracy and F1 scores analysis, showing that cognitive models compete well with other classification models.

Some of these learning methods we discussed, CP, RS, and LS, have been analyzed deeply under the Self-incremental LVQ (SILVQ) model in the paper "Self-incremental learning vector quantization with human cognitive biases" (Manome et al., 2021) [7]. The paper also includes analysis under the learning rate change with one dataset, *Glass* dataset [8, 7]. The paper compares the performances of OLVQ and LVQ with different initial learning rates and SILVQ with three cognitive learning methods. We extend this research with new datasets and a different base model, GLVQ.

Learning of the LVQ model is present if the learning rate is decreasing near 0 during the training period. Accuracy can still increase in some cases, even though learning rates do not change, corresponding to no learning. So, just examining accuracy scores does not give us much answer if the model is learning or not. However, examining the learning rate change during the training would give us some answers. The papers by Takahashi et al. (2010) [3] and Taniguchi et al. (2018) [6] lack a deep analysis of the learning rates. We included the best-performing cognitive learning rate methods from the paper "Cognitive Symmetry: Illogical but Rational Biases" [3], CP, DFH, MS, LS, and LSR, in our analysis, including OGLVQ as a comparison model. Our task is to discover the learning rate analysis on the GLVQ model, using the mentioned learning rate methods and supporting the results with accuracy and F-1 scores. For this task, we use various datasets, some of these datasets used in the paper "Self-incremental learning vector quantization with human cognitive biases" by Manome et al. (2021) [7]: *Ionosphere* dataset [9], *Iris* dataset [10], and *Sonar* dataset [11], additionally an open-source *Breast Cancer Wisconsin* dataset [12], and custom IFE Blood Samples datasets: *NSP* and *SP* datasets created by Saruhan (2023) in the report "Informational Image Data Pre-processing: IFE Blood Samples" [13]. In this paper, we name GLVQ models, which use learning rate optimizers implemented from cognitive learning rate methods as CGLVQ (cognitive GLVQ) to simplify the refer of the group.

Chapter 2

Methods

2.1 Data Preparation

Before introducing the datasets we use, we first need to learn some tools to use on those datasets. We use the Fourier transform and normalization for data preparation.

2.1.1 Fourier Transform

Fourier transform on $\mathbb{R}$ is a mathematical operation that transforms the input into the frequencies. The frequency function by Fourier transform gives a complex-valued function on $\mathbb{C}$. There are two Fourier transform types: Continuous Fourier Transform (CFT) and Discrete Fourier Transform (DFT or DtFT) [14].

CFT

As we can understand from the name, the Continuous Fourier Transform uses a continuous function, $x(t)$, as a time signal function with frequency function f to transform into frequency representation, $X(f)$. There are two versions of CFT, where one is Direct in Equation (2.1.1), and the other one is Inverse in Equation (2.1.2) [14]. The inverse Fourier transform allows us to return to the original signal from frequency transformation. If we take $i^2 = -1$ (imaginary unit), then:

Direct:

$$X_{\text{CFT}}(f) = \int_{-\infty}^{\infty} x(t) \cdot e^{-i2\pi ft} dt \tag{2.1.1}$$

Inverse:

$$x_{\text{CFT}}(t) = \int_{-\infty}^{\infty} X(f) \cdot e^{i2\pi ft} df \tag{2.1.2}$$

Since our transform function has e^i, is the complex value we can dissect the exponent to $\cos$ and $i \sin$ values for simplicity as Equation (2.2), and rewrite the Fourier equation like in Equation (2.3).

Since;

$$e^{i\theta} = cos(\theta) + isin(\theta), \quad \forall\theta \in [0, 2\pi) \tag{2.2}$$

Then;

$$\begin{aligned} X_{\text{CFT}}(f) &= \int_{-\infty}^{\infty} x(t) \cdot e^{-i2\pi ft} dt \\ &= \int_{-\infty}^{\infty} x(t) \cdot \cos(2\pi ft) dt - i \int_{-\infty}^{\infty} x(t) \cdot \sin(2\pi ft) dt \end{aligned} \tag{2.3}$$

DFT

DFT is the discrete counterpart of CFT and is used when we have discrete valued inputs (discrete-time signals). We are interested in DFT since the inputs we use for the LVQ model in this paper are discrete values. In DFT, we take $x(n)$ for time signals instead of $x(t)$ to indicate discreteness, where $n \in \mathbb{N}$. Then, the DFT formula will be:

$$\begin{aligned} X_{\text{CFT}}(f) &= \int_{-\infty}^{\infty} x(t) \cdot e^{-i2\pi ft} dt \\ \implies X_{\text{DFT}}(f) &= \sum_{n=-\infty}^{\infty} x(n \cdot \Delta t) \cdot e^{-i2\pi f(n \cdot \Delta t)} \end{aligned} \tag{2.4}$$

As we can see from the Equation (2.4), DFT has a similar calculation to CFT. The difference is that instead of using integral, we are using infinite sums because of the discreteness of DFT. Since the domain is discrete, we need to arrange the summation step accordingly. If f_s is the sampling frequency, we can denote the period as $\Delta t = \frac{1}{f_s}$. Then, the time t would be $t = n \cdot \Delta t$. We can further transform the Equation (2.4) into the Equation (2.5.1) and find the inverse of DFT equation as in the Equation (2.5.2) [14]:

Direct (Analysis):

$$X_{\text{DFT}}(\frac{f}{f_s}) = \sum_{n=-\infty}^{\infty} x(n) \cdot e^{-i2\pi \frac{f}{f_s} n} \tag{2.5.1}$$

Inverse (Synthesis):

$$x_{\text{DFT}}(n) = \frac{1}{f_s} \int_{-fs/2}^{fs/2} X(\frac{f}{f_s}) \cdot e^{i2\pi \frac{f}{f_s} n} df \tag{2.5.2}$$

Since $e^{i2\pi \frac{f}{f_s} n}$ is periodic, we do not need to calculate the equation for both $n > 0$ and $n < 0$ to save time and calculation power [14]. Then, we can get the final equation, Equation (2.6) for DFT with finite signals $|S| = N$ [14].

$$X_{\text{DFT}}(\frac{f}{f_s}) = \frac{1}{N} \sum_{n=0}^{N-1} x(n) \cdot e^{-i2\pi \frac{f}{f_s} n}, \quad -f_s/2 \leqslant f < f_s/2 \tag{2.6}$$

We do not directly transform the math into code but use a Python library, NumPy, to use the Fourier transform. The function is the Fast Fourier transform, and it uses DFT.

More on NumPy's Fourier transform at:
`https://numpy.org/doc/stable/reference/generated/numpy.fft.fft.html`

2.1.2 Normalization

Normalization is a statistical process to reduce the feature ranges to the same scale. It is essential to use normalization in some datasets since some features in datasets might have bigger range differences than others, and this difference shifts decision-making to the bigger range in machine learning methods. The weight vector can adjust the range difference in weight vector-based machine learning models. However, in prototype-based models, we do not have many options.

We use squared Euclidian distance in LVQ models, and the closeness to the given sample selects the winner. Suppose one of the features has a greater range difference than other features. In that case, that feature contributes the overall distance more than other features, resulting in the model's decision on the given sample being based mostly on that feature.

There are several normalization methods, and the common ones are min-max scaling (scaling to a range), clipping, log scaling, and z-score [15]. We use scaling to a range method in our paper, so we talk about it. If someone is curious about other methods, please visit the link by developers.google or source [15] to learn more.

Min-max scaling

The scaling method scales the features in the range $[0, 1]$. The method transforms the feature x_i of array $\mathbb{x}$ into $\hat{x}_i$ by:

$$\hat{x}_i = \frac{x_i - min(X_i)}{max(X_i) - min(X_i)} \tag{2.7}$$

Here in the Equation (2.7), $min(X_i)$ and $max(X_i)$ represent the minimum an maximum values of the ith element between all the feature arrays in a given dataset X, respectively.

2.2 Creating Datasets from IFE Data

2.2.1 What is un/structured data?

First, we need to understand what structured and unstructured data is. Structured data is the data that we can store in spreadsheets. These data have column names and rows for each data. Structured data columns can be strings, numerical, datetime, Boolean, or null values. Anything other than structured data, we call all data unstructured data. This data type contains image data, video data (which is also a type of image data with order), audio data, and text data. We would be using image data to train our algorithms, so we talk about how we use the image data to train machine learning algorithms and skip the other unstructured data types. We mentioned that machine learning algorithms require numerical values, but an image is hard to imagine as a numerical value. The idea of using

images as data for machine learning is to use pixel values of the data. Of course, we can use raw pixel values to train the algorithm, but to increase performance, we can convolute the pixel values for different algorithms. The machine learning methods in this paper use structured data, so we will not go deep into the concept and how to use unstructured data for machine learning.

2.2.2 Cross-correlation

We use (discrete) cross-correlation to detect the bars in our data. That is why, first, we need to understand what cross-correlation is. Since our data contain 2-dimensional images, we talk about 2-dimensional cross-correlation on images. Cross-correlation is similar to convolution, and the only difference is that in convolution, we rotate the second component (kernel) and then do the cross-correlation operation. If the kernel values are uniform, then the output of cross-correlation and convolution would give the same result. (Discrete) cross-correlation in Equation (2.8) and (Discrete) convolution in Equation (2.9).

$$x(n) * y(n) = \sum_{k=0}^{\infty} y(k) \cdot x(n+k) \tag{2.8}$$

$$x(n) \otimes y(n) = \sum_{k=0}^{\infty} y(k) \cdot x(n-k) \tag{2.9}$$

There are many ways to use cross-correlation in mathematics and computer science. Cross-correlation and convolution are used mostly with images to extract information from the image. Cross-correlation starts with a kernel (in a 2-dimensional case, we can see a kernel as a 2-dimensional array or a matrix). The kernel values run over all the image data. For every step, the kernel does an elementwise multiplication with the rows and columns of the matrix to create a combined value for the given position regarding kernel values. For example, if the kernel size is a 3×3 matrix with $\frac{1}{9}$ for each of the kernel values, then the kernel takes the mean of the image values with radius 1 pixel. So, using the cross-correlation (or convolution) operation makes our image more compact or blurry depending on the kernel's step size (stride). The kernel values can get any value, and different kernel values can find different aspects of the data that cross-correlation works on. We saw how to blur the image by taking the mean of surrounding pixel values. However, if we pick the top row of the 3×3 cross-correlation kernel as $+\frac{1}{3}$ and the bottom row as $-\frac{1}{3}$, the kernel finds the horizontal lines in the given image data.

After deciding the kernel size and values, we must also decide the kernel's step size. As we mentioned, step size also changes the interpretation of the convoluted image. If we take a 3×3 kernel and step size $= 3$, the image will shrink to one-third. However, if we take step size to 1 for the same example, we achieve a blurred image with the same size as the original image.

Lastly, we have a padding option for the cross-correlation and convolution. The padding adds empty spaces to our image's border. We add padding to adjust the image shape to our liking. Without any padding, the edge of the image will always be convoluted with the edge value of the kernel, but if we add padding, we give the edge values of the image a better chance to be in the center of the kernel.

We do not need a rectangular kernel or padding or stride to the width and height of the image. We can choose different sizes for stride and padding to width and height values. After passing through one convolution operation, the new image size changes for height and width given in Equation (2.10.1) and (2.10.2), respectively.

- H_{in} = input height
- W_{in} = input width
- H_{out} = output height
- W_{out} = output width

$$H_{out} = \lfloor \frac{H_{in} + 2 \times H_{padding} - H_{dilation} \times (H_{kernel\ size} - 1) - 1}{H_{stride}} + 1 \rfloor \tag{2.10.1}$$

$$W_{out} = \lfloor \frac{W_{in} + 2 \times W_{padding} - W_{dilation} \times (W_{kernel\ size} - 1) - 1}{W_{stride}} + 1 \rfloor \tag{2.10.2}$$

The equations in (2.10) are adapted from PyTorch documentation.

For more information:
`https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html#torch.nn.Conv2d`

2.2.3 Where is our data coming from?

We use IFE (immunofixation electrophoresis) test results to create datasets for our models to test. Our data is image data and contains 6 bars. We will not use the image as the input for our machine learning methods; instead, we will transform the image data into structured data to train the OGLVQ and CGLVQ methods.

IFE samples can be either blood or urine samples, but the IFE data uses blood samples [13]. The test can help to diagnose various diseases. The IFE test can detect problems such as [17]:

- Help in the diagnose and monitoring of lymphoma, chronic lymphocytic leukemia, or monoclonal gammopathies, such as multiple myeloma
- Investigate abnormal findings on other laboratory tests, such as total protein, albumin level, elevated calcium levels, or low white or red blood cell counts
- Evaluate someone for an inflammatory condition, an autoimmune disease, an infection, a kidney or liver disorder

Note: Adapted from Testing.com. (2021, March 24). Protein Electrophoresis, Immunofixation Electrophoresis, Testing.com `https://www.testing.com/tests/protein-electrophoresis-immunofixation-electrophoresis/` [17].

The IFE test is used to detect the abnormal globulin values in the sample. Immunoglobulins are proteins in the blood that contain a pair of heavy and light chains. α (Alpha), γ (Gamma), and μ (Mu) are the heavy chains, and these heavy chains combine with one of the light chains; κ (Kappa) or λ (Lambda), to form immunoglobulin as seen on the Figure 2.1.

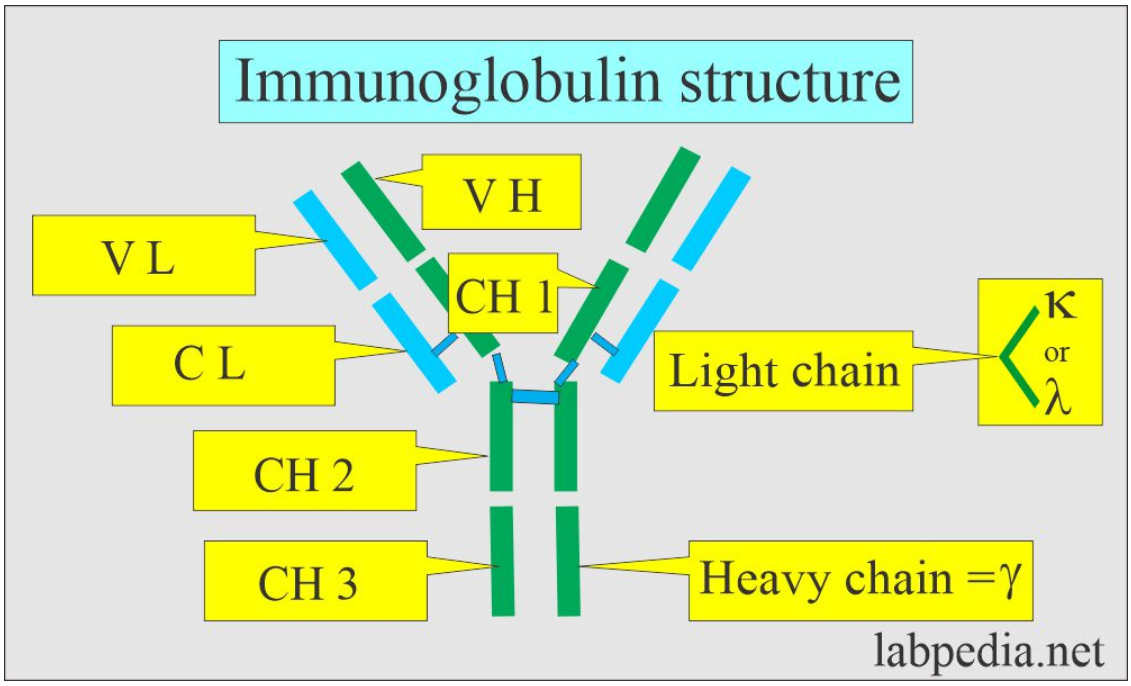
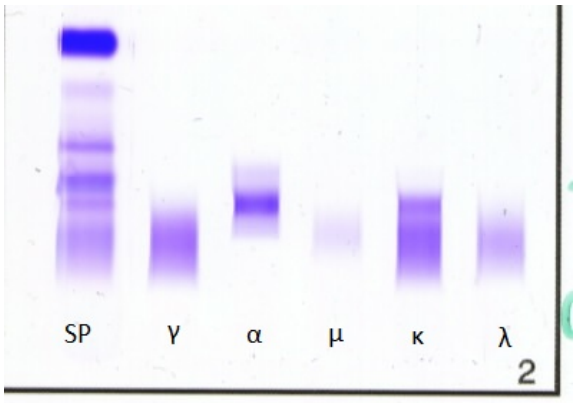


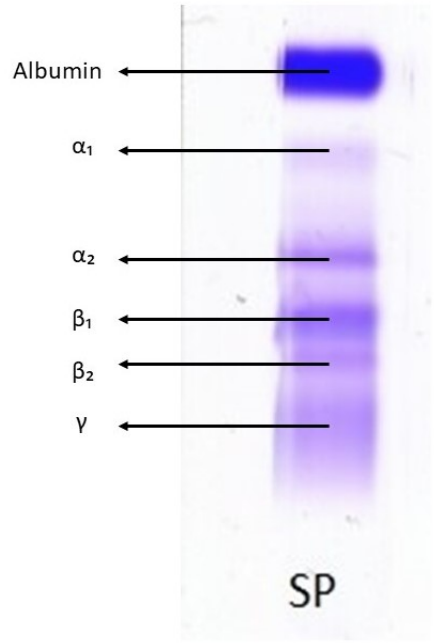
Figure 2.1: Immunoglobulin structure.

Note: From "Monoclonal Immunoglobulin (Ig), Monoclonal antibody, Immunofixation Electrophoresis (IFE)" by Labpedia.net. (2020, January 25) `https://labpedia.net/monoclonal-immunoglobulin-ig-monoclonal-antibody-immunofixation-electrophoresis-ife/` [16].

IFE test samples have 6 bars: SP (Serum Protein Electrophoresis) bar or Marker bar, γ bar, α bar, μ bar, κ bar, and λ bar from left to right can also be seen in Figure 2.2a. In Figure 2.2b, we see the SP bar contains 6 bands, namely; albumin, $\alpha - 1$ (Alpha-1), $\alpha - 2$ (Alpha-2), $\beta - 1$ (Beta-1), $\beta - 2$ (Beta-2), μ (Mu), and γ (Gamma) from top to bottom. The SP bar quantitively measures the albumin and globulins in the blood sample, so we cannot compare which globulin is abundant in the sample by just looking at the SP bar [18]. However, just checking the SP bar, we can detect if any of the globulin in the sample is abnormal, which would be useful to distinguish healthy and unhealthy patients. Except for the SP bar, the rest of the bars in IFE results measure the globulins qualitatively, so we can compare the bars and find which globulins are abundant in the blood sample.



(a) Sample IFE image.



(b) Sample image's SP part. These band name positions are just representative to show the order of the band names, and should not be taken as exact correct places for the bands.

Figure 2.2: IFE and SP images.

Note: Adapted from "Informational Image Data Preprocessing: IFE Blood Samples [Unpublished manuscript]" by M. Saruhan, Applied Mathematics for Network and Data Sciences, Mittweida University of Applied Sciences, p. 2 [13].

When an immunoglobulin is more abundant than it should be, we name the case the abundant immunoglobulin's name. For example, if the sample is marked as IgM-λ, the test found immunoglobulin μ-λ more than it should be in the sample. We have eight different diagnoses in our dataset, of which 6 of the types are labeled as unhealthy results with abundant immunoglobulin: IgA-κ, IgA-λ, IgM-κ, IgM-λ, IgG-κ, IgG-λ in their blood, one is healthy and the last one is unclear diagnosis.

We mentioned alpha and gamma for bands in the SP bar and bars in the IFE test results. While α and γ in IFE results represent the globulins, they do not represent the same in the SP bar. In the SP bar, α and γ are just band names. The similar name usage might be confusing, so read carefully that we talk about α bands in the SP bar or α globulin in IFE results.

The machine that gives the immunofixation results groups the albumin and globulin by their electrical charge [19]. The grouping can be mostly visible in the SP bar since the bar has albumin and different globulins altogether in the sample. Albumin is the most abundant in the blood sample and has the most negative charge than globulins [18]. Because of this reason, albumin has a distinct thickness and position in the SP bar.

2.2.4 Transforming to structured data

We need to pre-process the bar image data into structured data to prepare it for use in GLVQ models. We use the pre-processing of bar image data according to the report from Saruhan (2023), and the pre-processing we use in this paper is taken from that report [13] until we transform the data into a dataset. According to the report [13], we need one reference image for creating an albumin mask to find the SP bar position on each image file, another reference image for approximating the length of the bars, and one last reference image for finding the approximate distance between each bar.

After extracting the albumin mask from the respective reference image, we create a kernel (matrix) that is the same size as the albumin mask and has values one everywhere. We use this kernel to calculate the cross-correlation with the images to find the albumin's position in every image in our data. The kernel runs over any image and finds the position where it gets its maximum value. Since the albumin on each image file has a high pixel value, we expect to get the maximum value at the exact place of the albumin. However, to find the position of the albumin, we use a kernel with value one, and we do not need to convolute the kernel across the whole image. We know the albumin is located on the top-left part of each image, so checking the maximum value of the cross-correlation at the top-left part would be enough. An example of how the search with the albumin mask works can be seen in Figure 2.3.

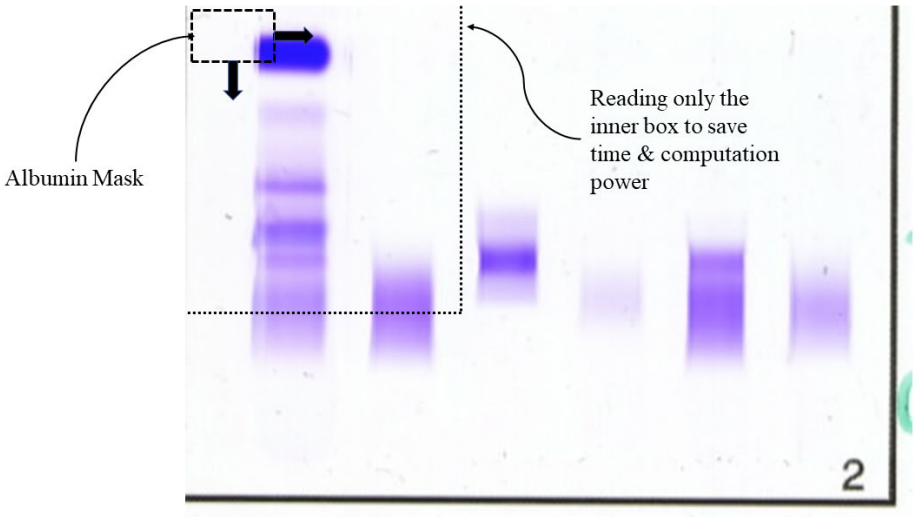


Figure 2.3: Finding mask's maximum value to detect albumin on test image.

Note: Adapted from "Informational Image Data Preprocessing: IFE Blood Samples [Unpublished manuscript]" by M. Saruhan, Applied Mathematics for Network and Data Sciences, Mittweida University of Applied Sciences, p. 2 [13]

Since we now can locate the SP bar, we can use it on our second reference image, approximating the length of the bars. After finding the SP bar in our second image, we find the length of the bar by checking the values of the SP bar from bottom to top. We find the position where the pixel values at the SP bar position pass a given threshold and mark the transaction position as the bottom of the SP bar. We can see the example of

how to find the bottom line of the bar in Figure 2.4. Since we already found the top of the albumin with the albumin mask, we take the top of the albumin, the same as the top of the SP bar, and calculate the difference from the bottom line we found. The difference would be the length of the SP bar.

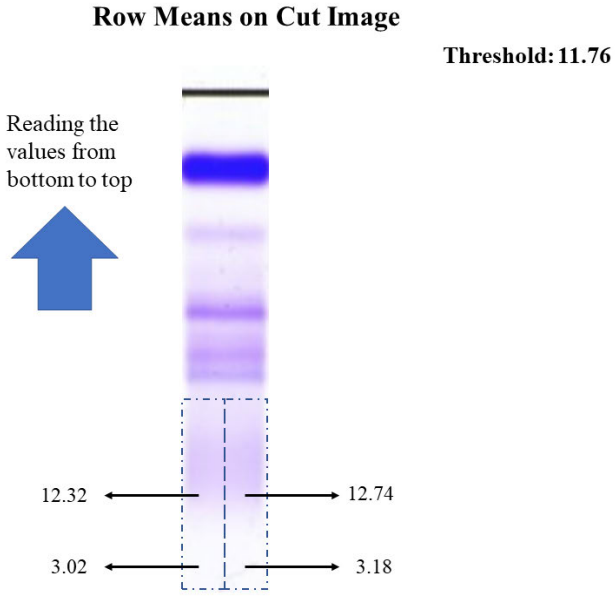


Figure 2.4: Finding bottom index of the bar.

Note: Adapted from "Informational Image Data Preprocessing: IFE Blood Samples [Unpublished manuscript]" by M. Saruhan, Applied Mathematics for Network and Data Sciences, Mittweida University of Applied Sciences, p. 2 [13]

After finding the SP bars' position and length in every image, we need to find the other bars. To find other bars, we use our third reference image. We use reference images because every bar pair distance is nearly the same in every image. So, we can find the distances from one image and implement them to others, which saves us lots of computational power and time. In this image, we first use an albumin mask to locate the SP bar to find our starting point. After finding the SP bar, we create a bar mask using the dimensions of the SP bar we found using albumin and length reference images. Same process as SP location using albumin mask, we use SP bar mask to locate other bars. The SP bar mask also has values one everywhere. Since every bar is ordered next to each other, we do not want to go all the way in the bar row and calculate the cross-correlation of the mask and the image. If we do that, we locate the bar with the maximum value with the kernel, which might not be the next bar after the SP bar. So, we need to take cross-correlation until the end of the next bar. Figure 2.5 shows how the method finds the second bar, γ bar. After finding the second bar, we can use the same process on the second bar to find the third bar, and so on. Lastly, to find the distances between each bar, we can use the distance information on other images to find the other bars after finding the SP bar via an albumin mask.

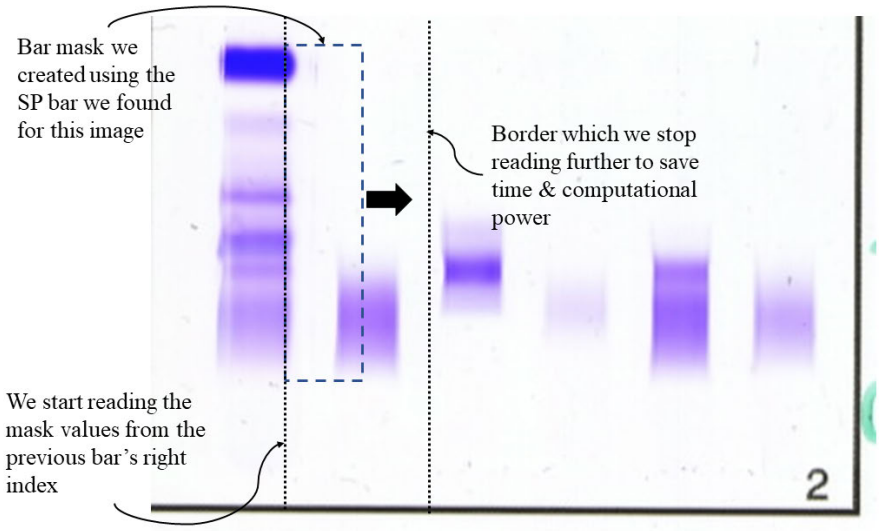
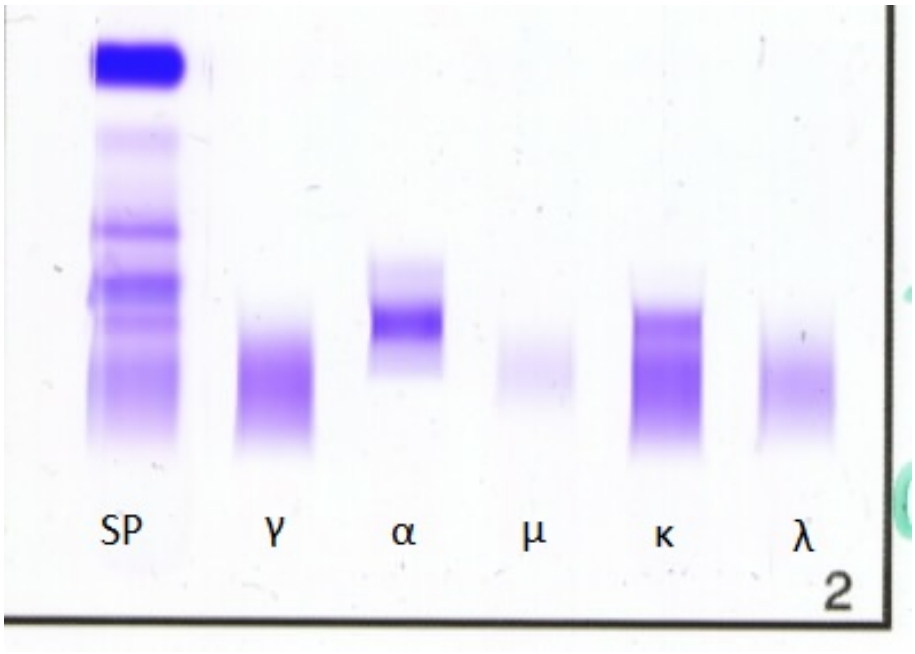


Figure 2.5: Example: Using detected SP bar as a mask to find other bars.

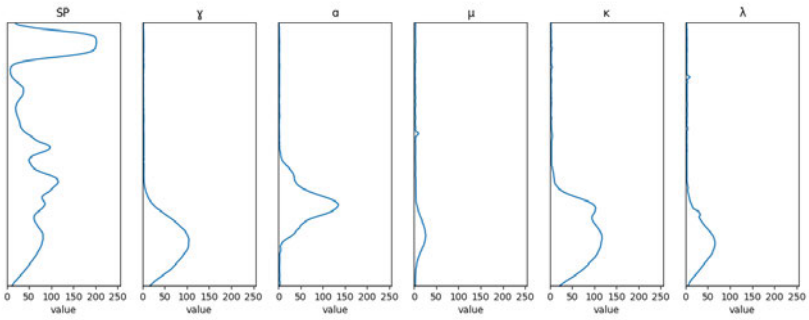
Note: Adapted from "Informational Image Data Preprocessing: IFE Blood Samples [Unpublished manuscript]" by M. Saruhan, Applied Mathematics for Network and Data Sciences, Mittweida University of Applied Sciences, p. 2 [13].

Now, we have all the bar locations in all the images, so it is time to extract the bar values to structure and organize data. We extract the bar values in the image's grayscale since we want to take the mean values for each row. As we mentioned, the positions of the bars are approximate. To get the correct value of the bars with confidence, we take the mean values of each row of the bar from the center of the bar with some small radius. This process helps us not to count the border of the bars since borders contain the background value of the images and could give us the wrong measure if we include the borders.

After reading each of the bar values for all our images, we get six arrays: SP, γ, α, μ, κ, and λ. The bar images and the corresponding value graphs are given in Figure 2.6. We have bar values to use as input for our models, but another way to use bars as input is to use the frequency of each bar. IFE bars (especially SP bars) have density changes in color. Since we have bars with color grading, to find the density of the bar values, we transform the bar values into the frequency of the values with the help of the Fourier transform. The Fourier transform is taken here because the bars' pixel values might differ for similar samples. Some samples might have tinted than others. However, if we look at the Fourier transform (DFT in this case) of each bar, we would be looking at the frequency of the bar, which differentiates the sudden changes on the bar better than without using the Fourier transform. We use DFT on all the bars separately on IFE blood test data because we do not want any relation between the end of each bar and the top of the next bar.



(a) Test IFE image.



(b) Graphs, created by reading the grayscale test image's bars.

Figure 2.6: Extracting the image values into structured data.

Note: Adapted from "Informational Image Data Preprocessing: IFE Blood Samples [Unpublished manuscript]" by M. Saruhan, Applied Mathematics for Network and Data Sciences, Mittweida University of Applied Sciences, p. 2 [13].

After we take each bar's Fourier transform separately, we combine the results in one 1-dimensional array since to use the bar values in a machine learning method as input, we need to flatten the bar values into a 1-dimensional array.

According to Leung (2016), the SP bar is used quantitively, showing if the patient is sick or not but not what kind of sickness [18]. The SP bar is useful for distinguishing sick patients, so we should include it in our dataset. However, the sickness is also shown in the other bars, and the SP bar and the SP bar have broader values than others, which results in models taking the SP bar into account more than the other bars. So, to test both approaches, we create two datasets: one with the SP bar included and one without the SP bar. We name the dataset that the SP bar included as *SP* and the dataset that does not have the SP bar as a feature named as *NSP* (noSP).

We also have labels for the images which are given to us. The labels contain "IgA-κ" (Ak), "IgA-λ" (Al), "IgM-κ" (Mk), "IgM-λ" (Ml), "IgG-κ" (Gk), "IgG-λ" (Gl), "without any sickness" (wo), and "unclear decision" (ud). "Unclear decision" cannot be used with

the other labels since it does not give us a clear answer. We store the remaining labels next to the bar values for each image data we transform.

At the end, we have two values in each row for both of the datasets; one is the input, which is the combination of the bar values we read, and the second one is the label of the data, which has seven different class discluding unclear decision as a label.

2.3 Datasets

Let us start by introducing the datasets we use in this paper. The datasets *Glass*, *Ionosphere*, *Iris*, and *Sonar* datasets we use, used in the results of the report "Self-incremental learning vector quantization with human cognitive biases" [7]. Additionally, we added the *Breast Cancer Wisconsin* Dataset [12] to our experiments to increase the variety. Since the features have distinct range differences, we use normalization on the *Breast Cancer Wisconsin* and *Iris* Dataset.

2.3.1 Breast Cancer Wisconsin dataset

The *Breast Cancer Wisconsin* dataset was collected and computed from a digitized image of a fine needle aspirate (FNA) of a breast mass [12]. Dataset has two classes: Diagnosis "Malignant" (M) and "Benign" (B), which are the diagnoses given to the patient. The dataset contains ten central values:

- radius (mean of distances from the center to points on the perimeter)
- texture (standard deviation of gray-scale values)
- perimeter
- area
- smoothness (local variation in radius lengths)
- compactness (perimeter^2 / area - 1.0)
- concavity (severity of concave portions of the contour)
- concave points (number of concave portions of the contour)
- symmetry
- fractal dimension ("coastline approximation" - 1)

These values branch into 30 features by taking each value's mean, standard error, and worst (largest) [12]. Since *Breast Cancer Wisconsin* dataset have various range between its feature values, we use normalization on the dataset. Number of samples for each class in the dataset is given in Figure 2.7.

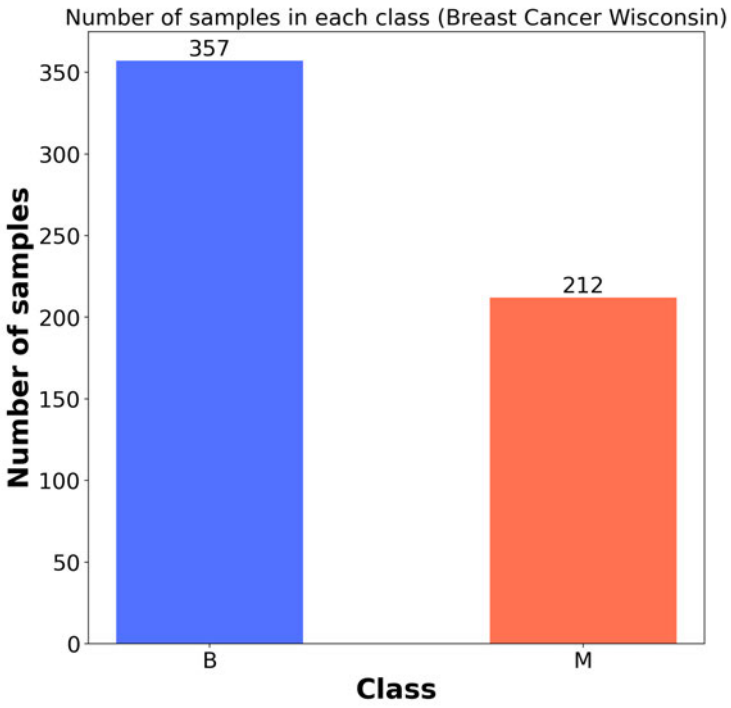


Figure 2.7: # Samples for each class of *Breast Cancer Wisconsin* dataset.

This dataset is available through Kaggle:
`https://www.kaggle.com/datasets/uciml/breast-cancer-wisconsin-data`

This database is also available through the UW CS FTP server:
ftp: ftp.cs.wisc.edu
cd math-prog/cpo-dataset/machine-learn/WDBC/

Also can be found on the UCI Machine Learning Repository:
`https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29`

2.3.2 Iris dataset

The *Iris* dataset is a well-known dataset used by biologist Ronald Fisher (1936) in his paper The use of multiple measurements in taxonomic problems as an example of linear discriminant analysis [10]. The dataset contains three iris species as classes: "Iris-setosa," "Iris-virginica," and "Iris-versicolor," with 50 samples each. The dataset has four features for each sample:

- length of the sepals (range: 4.3cm – 7.9cm)
- width of the sepals (range: 2.0cm – 4.4cm)
- length of the petals (range: 1.0cm – 6.9cm)

- width of the petals (range: 0.1cm – 2.5cm)

Since the range difference of the *Iris* dataset's features differs from each other when we take the feature difference between prototypes and the samples, we will see the biggest contribution to the distance comes from the length of the petals. Hence the decision mostly relies on the length of the petals because the length of the petals has the highest range difference between the features. We use normalization on the dataset's features to eliminate the unequal decision power between the features.

Number of samples for each class in the dataset is given in Figure 2.8.

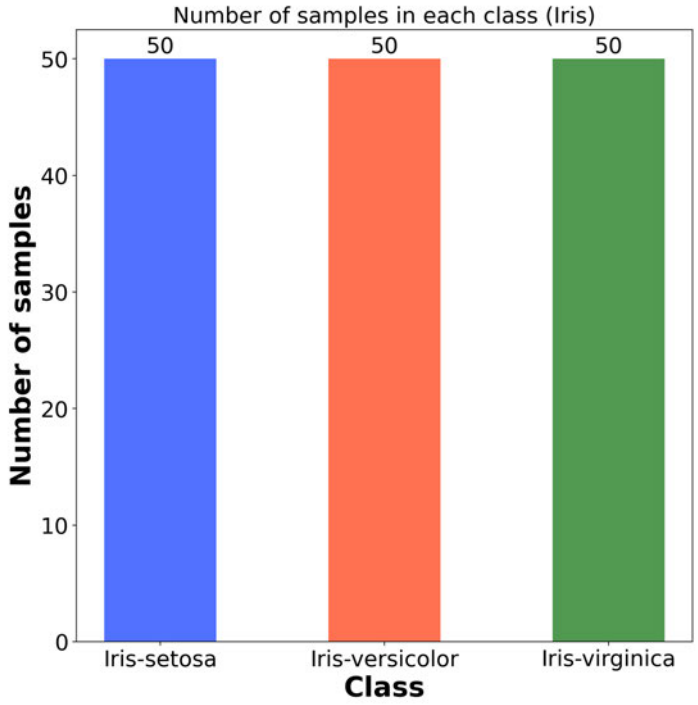


Figure 2.8: # Samples for each class of *Iris* dataset.

This dataset is available through Kaggle:
`https://www.kaggle.com/datasets/sims22/irisflowerdatasets`

2.3.3 Ionosphere dataset

The *Ionosphere* dataset is donated by Vince Sigillito and sourced by Space Physics Group, Applied Physics Laboratory at Johns Hopkins University. The data is created by measuring the ionosphere to see if there are any free electrons in the ionosphere or not. The data is binary classification with "Good" (G) and "Bad" (B) as class names. "Good" indicates there is evidence that there is some type of structure in the ionosphere, and "Bad" shows there is no structure. The feature size is 34, and there are 351 samples in the *Ionosphere* dataset. Features are signals generated by 16 high-frequency antennas with total transmitted power on the order of 6.4 [9]. In Figure 2.9 we see the sample distribution of the dataset.

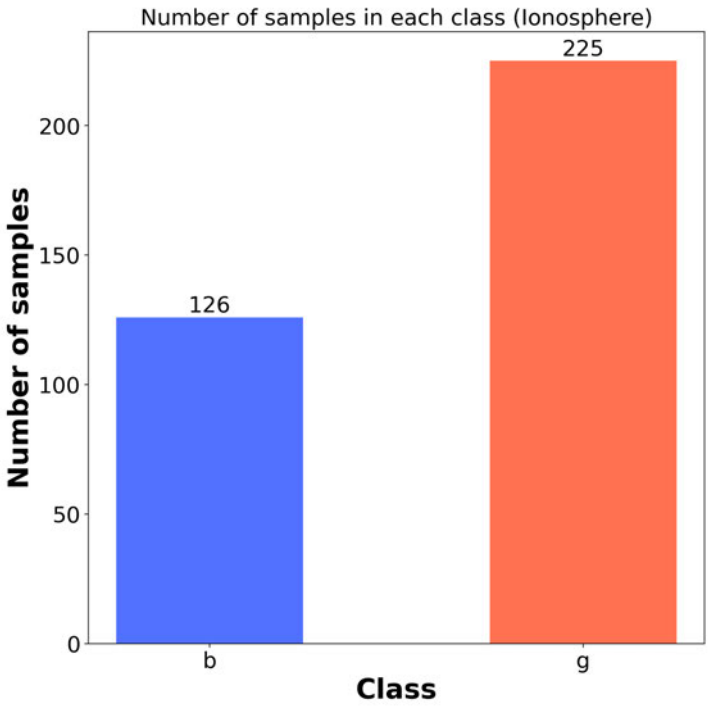


Figure 2.9: # Samples for each class of *Ionosphere* dataset.

This dataset is available through Kaggle:

`https://www.kaggle.com/datasets/prashant111/ionosphere`

Or through UC Irvine Machine Learning Repository:

`https://archive.ics.uci.edu/dataset/52/ionosphere`

2.3.4 Sonar dataset

The *Sonar* is created by Terry Sejnowski at the Salk Institute and the University of California at San Deigo with the collaboration of R. Paul Gorman of Allied-Signal Aerospace Technology Center [11], and the dataset is used in the study "Analysis of Hidden Units in a Layered Network Trained to Classify Sonar Targets." *Sonar* dataset contains sonar signals bouncing off metal cylinders of "Mines" (M) and "Rocks" (R) under various angles and conditions [11]. The dataset contains 111 "Mines" and 97 "Rocks" samples. Each sample contains 60 features in the range between 0.0 and 1.0, representing the energy in a particular frequency band. In their experiment, Gorman, R. P., and Sejnowski, T. J. achieved accuracy between 77.1% and 90.4% on a test set using neural networks varying between 0 to 24 hidden layers [11]. Figure 2.10 shows the sample distribution of the *Sonar* dataset.

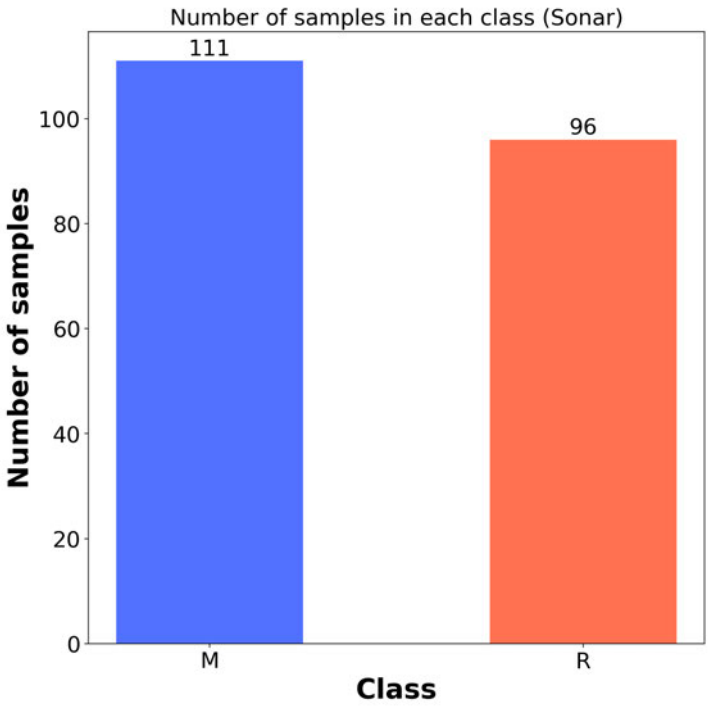


Figure 2.10: # Samples for each class of *Sonar* dataset.

This dataset is available through Kaggle:

`https://www.kaggle.com/datasets/rupakroy/sonarcsv`

2.3.5 SP and NSP datasets

Besides open-source datasets, our private datasets were created from the IFE blood test data we mentioned in the Section 2.2. The first dataset we created from IFE test data takes the Fourier transforms of all the bars, including SP bars, and groups the samples into "Normal" and "Abnormal" classes. Every group except "wo" and "ud" is labeled "Abnormal," and "wo" data turns into "Normal." We do not take into account undefined data. While selecting prototypes for this dataset, we first select the prototypes from each sick group and then join them. We select one prototype from the "Normal" class for each "Abnormal" class prototype. The prototypes selected from the "Abnormal" class are divided equally by every group contributing to the class. Figure 2.11 shows the sample number of each class before grouping the labels into two groups, while Figure 2.12 shows the sample number of each class after labeling the classes into two groups, "Normal" and "Abnormal."

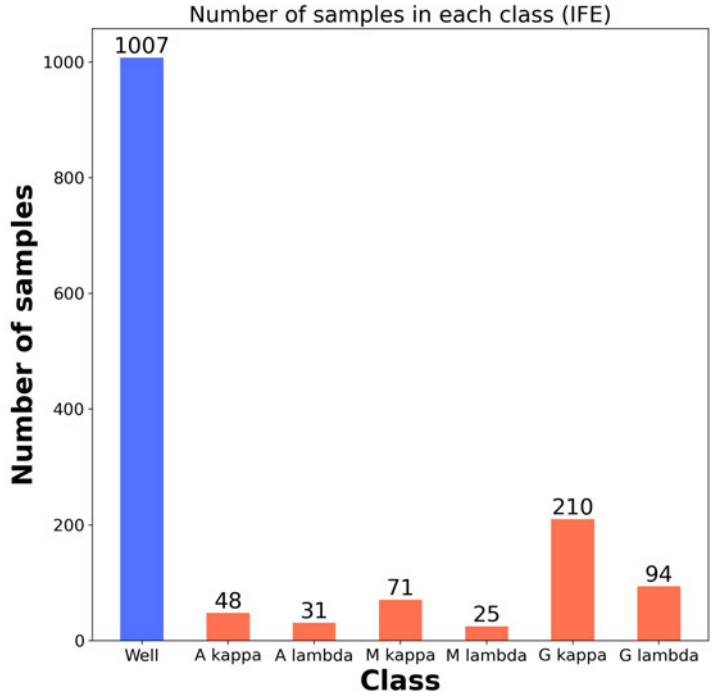


Figure 2.11: # Samples for each class of *SP and NSP* dataset.

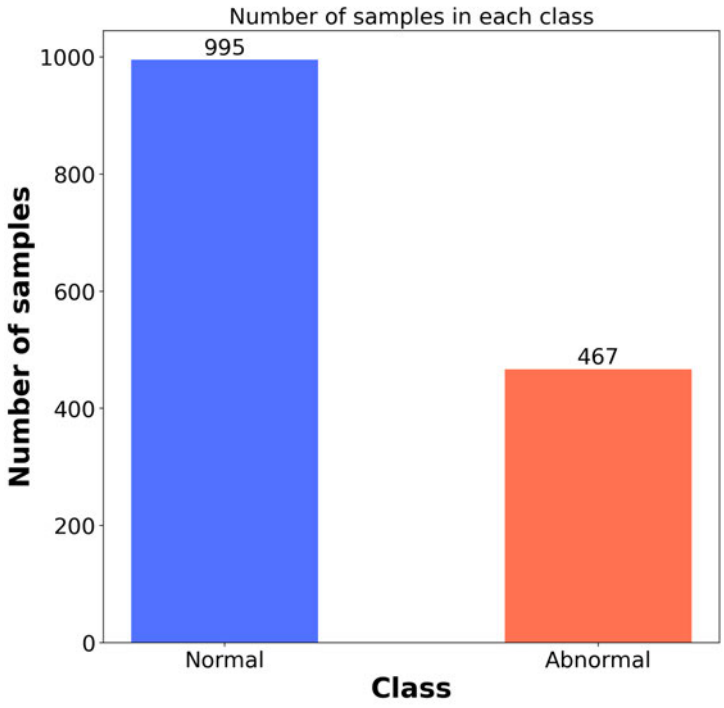


Figure 2.12: # Samples for each class of *SP and NSP* dataset after grouping classes into two groups.

The second dataset we created from IFE test data takes all the bars except the SP bar while applying the Fourier transform. We apply the same logic to the first dataset we created from IFE test data and divide the samples into "Normal" and "Abnormal" classes. Since each bar on IFE data has a length of 184 pixels (values), while the *SP* dataset has 1104 entries as a feature-length for each sample, *NSP* dataset has 920 entries. Both of the created IFE datasets have the same sample size of 1486 samples. We use the Fourier transform in both IFE datasets since the bars of IFE blood samples are examined by the coloration (frequency) change rather than the coloration value.

2.4 Models

2.4.1 GLVQ

There are many LVQ sub-models, but all the sub-models have the same principle: all prototype-based supervised learning algorithms. In a prototype-based algorithm, we use no weight vectors but prototypes (ω), a small group of our data. When we train the model, the prototypes move closer or farther away from the trained data depending on the situation and the sub-model. Ultimately, the distance of features of the sample from the selected prototypes' final values makes the decision. Since we use GLVQ (Generalized LVQ) [20] as the sub-model of LVQ in this paper, we first explain the construct of GLVQ.

GLVQ has been coined out by Sato and Yamada (1995) [20]. As with all the other LVQ models, GLVQ has prototypes with classes and local learning rates for each ω. First, we set the initial learning rate ($\epsilon(0)$), which (de)amplifies the prototypes' movement speed to the given direction. Learning rate at time t ($\epsilon(t)$) changes depending on the current state of the ω and prediction. We use different types of learning rate updates for different learning rate update approaches. Each ω_i can have different ϵ, namely local ϵ (ϵ_i), which makes each ω learn on a different scale.

After choosing our $\epsilon(0)$, we must choose our prototypes. Prototypes must include all the class types. If one of the class representatives would be missing, then we would not have a prediction for the given class. We measure the distance between each training sample and all the prototypes. Generally, squared Euclidian distances are used for the distance measurement for LVQ. The Equation (2.11.1) shows squared Euclidian distance with real-valued arrays. When we transform data with Fourier transform, data become complex-valued inputs. The calculation we use is still squared Euclidian distance for complex values, which is the Equation (2.11.2), where $\mathbb{x}^c \in \mathbb{C}^n$ is a complex-valued array with $\mathbb{x}^c = a + i \cdot b$, where $a, b \in \mathbb{R}^n$.

- n: number of features in $\mathbb{x}$
- x_i: i^{th} feature value of $\mathbb{x}$
- y_i: i^{th} feature value of ω_j
- $d_r(\mathbb{x}, \omega_j)$: distance for real values between $\mathbb{x}$ and ω_j features
- $d_c(\mathbb{x}^c, \omega_j^c)$ distance for complex values between $\mathbb{x}$ and ω_j features

Then:

$$d_r(\mathbb{x}, \omega_j) = \sum_{i=1}^{n}(x_i - y_i)^2 \tag{2.11.1}$$

$$d_c(\mathbb{x}^c, \omega_j^c) = \sum_{i=1}^{n}(|x_i^c - y_i^c|^2) \tag{2.11.2}$$

GLVQ has two winner ω, and both are updated. The first winner is the closest ω to the sample $\mathbb{x}$ with the same class with distance noted as $d^+(\mathbb{x})$, and the second winner is the closest ω to the sample with a different class with distance noted as $d^-(\mathbb{x})$. We can define a $\mu(\mathbb{x})$ of sample array $\mathbb{x}$ as a relative distance in Equation (2.12). According to Sato and Yamada (1995), when the $\mu(\mathbb{x})$ value for $\mathbb{x}$ is negative, the prediction is correct and false otherwise [20].

$$\mu(\mathbb{x}) = \frac{d^+(\mathbb{x}) - d^-(\mathbb{x})}{d^+(\mathbb{x}) + d^-(\mathbb{x})} \tag{2.12}$$

$\mu(\mathbb{x})$ calculation is used in cost function ($l(\mathbb{x})$) like in the Equation (2.13) to calculate the error of the model's current state. Error is used to update the model to a local minimum error state by using the derivative of $l(\mathbb{x})$.

$$l(\mathbb{x}) = f(\mu(\mathbb{x})) \tag{2.13}$$

To use $\mu(\mathbb{x})$ in the cost function, first $\mu(\mathbb{x})$ must go under an activation function f (which is primarily the Sigmoid function (σ) in the Equation (2.14)). The winners updated in the direction of the sample with amplitude of the $\epsilon(t)$ and $\frac{\partial l(\mathbb{x})}{\partial \omega}$.

$$\sigma(\mu(\mathbb{x})) = \frac{1}{1 + e^{-\mu(\mathbb{x})}} \tag{2.14}$$

The first winner ω (ω^+) is updated by the attraction in the Equation (2.15.1), while we apply repulsion on the second winner (ω^-) in the Equation (2.15.2) [20]. These calculations come from $\frac{\partial l(\mathbb{x})}{\partial \omega}$. Attraction makes ω^+ move closer to the given sample by its local ϵ value, ϵ^+; while repulsion makes the ω^- move further away from the sample with ϵ^-. This way, next time, ω^+ would have a better chance to be the winner for the same sample while ω^- would have less chance to be the winner.

Attraction:

$$\omega^+(t+1) = \omega^+(t) - \epsilon^+ \frac{\partial l(\mathbb{x})}{\partial \omega^+} \tag{2.15.1}$$

Repulsion:

$$\omega^-(t+1) = \omega^-(t) - \epsilon^- \frac{\partial l(\mathbb{x})}{\partial \omega^-} \tag{2.15.2}$$

The OGLVQ model and CGLVQ models are structured above the GLVQ model.

2.4.2 Learning rate optimizers

We use GLVQ to create a more human-like learning system for the learning algorithm. A component in every machine learning algorithm is the ϵ, which controls the learning speed of each training sample. Taking the $\epsilon(0)$ high might seem reasonable to create a fast learning machine learning method, but the reality is far away from that. A high $\epsilon(0)$ causes the method to fixate on the last training sample, which would diminish the knowledge learned from the previous training samples. On the other hand, if the $\epsilon(0)$ is too low, there would be no learning from the training set. That is why we need to find an adequate $\epsilon(0)$ with optimal learning rate update for the method since an optimal ϵ gives a better prediction. For this paper, we use the optimized learning rate method for GLVQ from Kohonen (1995) [1] and learning rate methods from cognitive science [6, 3, 7].

Optimized GLVQ

We use an Optimized Learning Rate on GLVQ as a basis for our experiments. In the book "Self-Organizing Maps" by Kohonen (1995) [1], the schema of OLVQ1 is given, but we implement only the learning rate part of the GLVQ to construct OGLVQ. The paper explains that the optimal learning rate for LVQ1 comes from the prototype update [1]. Assumption here is that when updating $\omega_i(t+1)$, $\omega_i(t)$ contains trance of $\epsilon_i(t-1)$. So to reach the optimal $\epsilon_i(t)$, we take the $\omega_i(t+1)$ update equation on GLVQ for both of the ω updates, change the $\omega_i(t)$ values with $\epsilon_i(t-1)$ and solve the equation for $\epsilon_i(t)$. The winner $\omega(t)$ updates for squared distance can be seen in the Equation (2.16.1) and (2.16.2) respectively for GLVQ under squared Euclidian distance and σ as activation function.

$$\begin{aligned}
\omega^+(t+1) &= \omega^+(t) - \epsilon^+ \cdot \frac{\partial l(\mathbb{x})}{\partial \omega^+} \\
&= \frac{\partial l(\mathbb{x})}{\partial \mu} \cdot \frac{\partial \mu}{\partial d^+(\mathbb{x})} \cdot \frac{\partial d^+(\mathbb{x})}{\partial \omega^+} \\
&= \omega^+(t) + \epsilon^+(t) \cdot \sigma(\mu(\mathbb{x})) \cdot [1 - \sigma(\mu(\mathbb{x}))] \cdot \frac{4d^-(\mathbb{x})}{[d^+(\mathbb{x}) + d^-(\mathbb{x})]^2} \cdot [\mathbb{x} - \omega^+(t)]
\end{aligned} \tag{2.16.1}$$

$$\begin{aligned}
\omega^-(t+1) &= \omega^-(t) - \epsilon^- \cdot \frac{\partial l(\mathbb{x})}{\partial \omega^-} \\
&= \frac{\partial l(\mathbb{x})}{\partial \mu} \cdot \frac{\partial \mu}{\partial d^-(\mathbb{x})} \cdot \frac{\partial d^-(\mathbb{x})}{\partial \omega^-} \\
&= \omega^-(t) - \epsilon^-(t) \cdot \sigma(\mu(\mathbb{x})) \cdot [1 - \sigma(\mu(\mathbb{x}))] \cdot \frac{4d^+(\mathbb{x})}{[d^+(\mathbb{x}) + d^-(\mathbb{x})]^2} \cdot [\mathbb{x} - \omega^-(t)]
\end{aligned} \tag{2.16.2}$$

To find the optimized learning rate update equation of GLVQ, we use the Equations (2.16.1) and (2.16.2) to generate The Equations (2.17.1) and (2.17.2) respectively.

$$\epsilon^+(t) = [1 - \epsilon^+(t)] \cdot \sigma(\mu(\mathbb{x})) \cdot [1 - \sigma(\mu(\mathbb{x}))] \cdot \frac{4d^-(\mathbb{x})}{[d^+(\mathbb{x}) + d^-(\mathbb{x})]^2} \cdot \epsilon^+(t-1) \quad (2.17.1)$$

$$\epsilon^-(t) = [1 + \epsilon^-(t)] \cdot \sigma(\mu(\mathbb{x})) \cdot [1 - \sigma(\mu(\mathbb{x}))] \cdot \frac{4d^+(\mathbb{x})}{[d^+(\mathbb{x}) + d^-(\mathbb{x})]^2} \cdot \epsilon^-(t-1) \quad (2.17.2)$$

At the end, if we re-arrange the Equations (2.17.1) and (2.17.2) for $\epsilon(t)$, the OGLVQ $\epsilon(t)$ equations gives us the Equations (2.18.1) and (2.18.2) respectively.

$$\epsilon^+(t) = \frac{\epsilon^+(t-1)}{1 + [4\epsilon^+(t-1) \cdot \sigma(\mu(\mathbb{x})) \cdot [1 - \sigma(\mu(\mathbb{x}))] \cdot \frac{d^-(\mathbb{x})}{[d^+(\mathbb{x}) + d^-(\mathbb{x})]^2}]} \quad (2.18.1)$$

$$\epsilon^-(t) = \frac{\epsilon^-(t-1)}{1 - [4\epsilon^-(t-1) \cdot \sigma(\mu(\mathbb{x})) \cdot [1 - \sigma(\mu(\mathbb{x}))] \cdot \frac{d^+(\mathbb{x})}{[d^+(\mathbb{x}) + d^-(\mathbb{x})]^2}]} \quad (2.18.2)$$

The Equations (2.18.1) and (2.18.2) are our learning rate updates for winner prototypes for each time step t for OGLVQ.

The Python code of OGLVQ can be found in Appendix A or on the following GitHub page:
`https://github.com/mertsaru/Cognitive-GLVQ/blob/main/OGLVQ.py`

Cognitive bias optimizers

Since we use LVQ, which mimics human-like learning, we try to find the optimal learning rate with the help of cognitive science. There are many learning rate methods proposed which are connected to human reasoning. Cognitive science gives us two concepts in human reasoning: symmetry bias and mutual exclusivity bias. Even though these biases are not always logical, they are supported by cognitive science [21]. Several methods are derived from these cognitive science assumptions, and these methods have been observed to increase LVQ models' performance [3, 7].

Symmetry bias (S) in human reasoning is assuming ($q \implies p$) from ($p \implies q$) [7]. Even though ($p \implies q$) does not have a logical connection with ($q \implies p$), humans are prone to assume that these logical sentences are equal (cited from Manome et al. (2021) and Taniguchi et al. (2018) which of these sources cited the Japanese translation of Shinohara et al. (2007) [3, 6, 4]). An example of symmetry in human reasoning would hear *"if the weather was rainy, then the ground is wet. ($p \implies q$)"* and assuming *"only if the ground is wet, then the weather was rainy a while ago. ($q \implies p$)"* [22].

Another logical thinking humans have is mutual exclusivity bias (MX), after hearing ($p \implies q$), assuming ($\neg p \implies \neg q$) [6, 7]. Here $\neg p$ corresponds to negation of p,

not p. An example of mutual exclusivity is when a kid is hearing *"if you don't clean up your room, then you will not be allowed to play ($p \implies q$)"* from their mother and interpret the sentence as *"if I clean up your room, then my mom will allow me to play. ($\neg p \implies \neg q$)"* by mutual exclusivity [22]. These two logical biases, symmetry and mutual exclusivity bias ($q \implies p \wedge \neg p \implies \neg q$), can be combined into a biconditional relationship ($p \iff q$), which we can also see in Figure 2.13 [3].

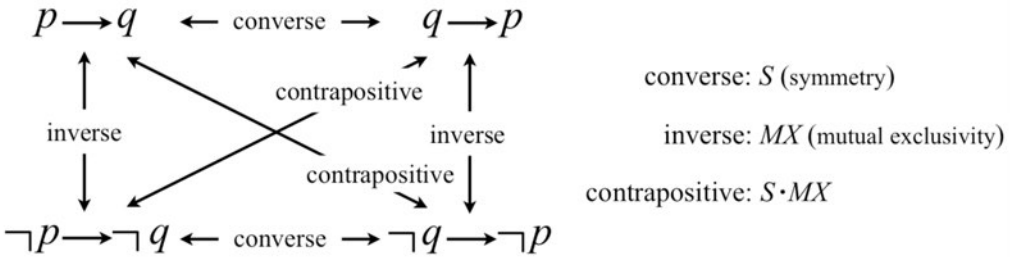


Figure 2: Logical relationship between conditionals.

Figure 2.13: Relation between cognitive biases.

Note: From "Cognitive Symmetry: Illogical but Rational Biases" by T. Takahashi, M. Nakano, and S. Shinohara, Symmetry Culture and Science. 21. 1-3, p. 7 `https://www.researchgate.net/publication/285850238_Cognitive_Symmetry_Illogical_but_Rational_Biases` [3].

It is not always easy to see that the connection is off in the example. However, if we take an example where p is "the shoe is white" and q "a star is printed on it," symmetric bias infers "if a star is printed on a shoe, then the shoe is white" ($q \implies p$) and mutual exclusivity bias infers "if the shoe is not white, then a star is not printed on it" ($\neg p \implies \neg q$) which are certainly not correct to assume from hearing "if the shoe is white, then a star is printed on it ($p \implies q$)" [6].

The Table 2.1 shows the co-occurrence table of p and q's relationships.

	q	$\neg q$
p	a	b
$\neg p$	c	d

Table 2.1: Co-occurrence frequency for event p and event q.

Note: Adapted from "Self-incremental learning vector quantization with human cognitive biases" by N. Manome, S. Shinohara, T. Takahashi, Y. Chen, and U. Chung, Scientific Reports 11(1), p. 3 (`https://doi.org/10.1038/s41598-021-83182-4`) [7].

We can map the logic relations to machine learning logic by assuming p as *the predicted label is prototype i's label* and q as *the predicted result is correct*, provided by Manome (2021) [7]. If we take $L(\mathbb{x})$ as the sample $\mathbb{x}$ label, $L(\omega^{\mathbb{x}})$ as the predicted prototype label (or winner class for short) of sample $\mathbb{x}$, and $L(\omega_i)$ as prototype i's label, then the co-occurrence frequency table be like in the Table 2.2.

	$L(\mathbb{x}) = L(\omega^{\mathbb{x}})(q)$	$L(\mathbb{x}) \neq L(\omega^{\mathbb{x}})(\neg q)$
$L(\omega_i) = L(\omega^{\mathbb{x}})(p)$	a_i	b_i
$L(\omega_i) \neq L(\omega^{\mathbb{x}})(\neg p)$	c_i	d_i

Table 2.2: Co-occurrence frequency for each prototype ω_i.

Note: Adapted from "Self-incremental learning vector quantization with human cognitive biases" by N. Manome, S. Shinohara, T. Takahashi, Y. Chen, and U. Chung, Scientific Reports 11(1), p. 5 (`https://doi.org/10.1038/s41598-021-83182-4`) [7].

We defined two biases (S and MX), and now we define other essential properties of the probabilistic model. One is excluded middle (XM), a natural condition of whether an event occurs, and another one is called estimation relativity (ER) [3]. Below in Table 2.3, we list the biases and properties respective to their logical dictation [3]. B denotes the probabilistic formula, and $B(q|p)$ represents how strong someone subjectively believes that q occurs after p happened [3]. If the relationship holds, we say B has the respective bias or property.

Symmetry bias (S):	$B(q\|p) \sim B(p\|q)$
Mutual exclusivity bias (MX):	$B(q\|p) \sim B(\neg q\|\neg p)$
The law of excluded middle (XM):	$B(q\|p) \sim 1 - B(\neg q\|p)$
Estimation relativity (ER):	$B(q\|p) \sim 1 - B(q\|\neg p)$

Table 2.3: Biases, bias properties.

Note: Adapted from "Cognitive Symmetry: Illogical but Rational Biases" by T. Takahashi, M. Nakano, and S. Shinohara, Symmetry Culture and Science. 21. 1-3, p. 7 `https://www.researchgate.net/publication/285850238_Cognitive_Symmetry_Illogical_but_Rational_Biases` [3].

There are many ways to implement one or a couple of these logical biases in learning rate optimizers. Since we do not want to clutter the paper with many cognitive learning rate optimizers, we filter and pick the ones that show higher performance according to Table 2.4.

	CP	**DP**	**DFH**	**RS**	**MS**$_{1,0}$	**LS**	**LSR**
H03	0.000	0.000	0.964	0.158	0.968	0.969	0.971
AS95	0.823	0.781	0.905	0.761	0.885	0.904	0.782
WDK90	0.944	0.844	0.961	0.888	0.962	0.969	0.922

Table 2.4: Determination coefficients of cognitive models.

Note: The text and human data collected from Hattori (2003) (H03) [23], Anderson & Sheu (1995) (AS95) [25], and Wasserman et al. (1990) (WDK90) [5].

Note: Adapted from "Cognitive Symmetry: Illogical but Rational Biases" by T. Takahashi, M. Nakano, and S. Shinohara, Symmetry Culture and Science. 21. 1-3, p. 15 `https://www.researchgate.net/publication/285850238_Cognitive_Symmetry_Illogical_but_Rational_Biases` [3].

CGLVQ optimizers

CP model This model is a conditional probability (CP) model. The model is the most basic among others. CP only satisfies XM [3]. If we use the probability notation p, q and the Table 2.2, t for time, the causal relationship between events ($R(t)$) of the CP would be the following [3, 7]:

$$R^{\text{CP}(q|p)}(t) = \frac{a(t)}{a(t) + b(t)} \tag{2.19}$$

We can see that CP satisfies XM by looking at the following equation [3]:

$$\begin{aligned} R^{\text{CP}(q|p)}(t) &= \frac{a(t)}{a(t) + b(t)} \\ &= 1 - \frac{b(t)}{a(t) + b(t)} \\ &= 1 - R^{\text{CP}(\neg p|q)}(t) \end{aligned} \tag{2.20}$$

DFH model The Dual factor heuristic (DFH) model (cited from Takahashi et al. (2010) [3] which cites from Hattori (2001) [24] Japanese translation and Hattori (2003) [23]) is one of the models that work best for a human-like model [22, 3]. DFH is derived from CP and defined by the product of CP and its inverse [3]:

$$\begin{aligned} R^{\text{DFH}(q|p)}(t) &= \sqrt{R^{\text{CP}(q|p)}(t) \cdot R^{\text{CP}(p|q)}(t)} \\ &= \frac{a(t)}{\sqrt{[a(t) + b(t)] \cdot [a(t) + c(t)]}} \end{aligned} \tag{2.21}$$

DFH satisfies the S bias, as we can see in the following equation [3]:

$$
\begin{aligned}
R^{\mathrm{DFH}(q|p)}(t) &= \sqrt{R^{\mathrm{CP}(q|p)}(t) \cdot R^{\mathrm{CP}(p|q)}(t)} \\
&= \sqrt{R^{\mathrm{CP}(p|q)}(t) \cdot R^{\mathrm{CP}(q|p)}(t)} \\
&= R^{\mathrm{DFH}(p|q)}(t)
\end{aligned}
\tag{2.22}
$$

MS model According to Takahashi et al. (2010), [3], representing human cognition, we should use neither too much symmetry nor no symmetry but somewhere in between. That is why we have a middle symmetry (MS) model. MS has parameters α and β to control the magnitude of the symmetry. $\mathrm{MS}_{\alpha,\beta}$ has the following equation [3]:

$$
R^{\mathrm{MS}_{\alpha,\beta}(q|p)}(t) = \frac{a(t) + \beta \cdot d(t)}{a(t) + \beta \cdot d(t) + b(t) + \alpha \cdot c(t)} \tag{2.23}
$$

$\mathrm{MS}_{0,0}$ corresponds to the CP model. According to Takahashi et al. (2010) [3], when $\alpha = 1$ and $\beta = 0$, Hattori (2003) achieved good performance with *Human* dataset with causal inductive experiments [3, 23]. With the given parameters, $\mathrm{MS}_{1,0}$ equation would look like [3]:

$$
R^{\mathrm{MS}_{1,0}(q|p)}(t) = \frac{a(t)}{a(t) + b(t) + c(t)} \tag{2.24}
$$

$\mathrm{MS}_{1,0}$ is symmetric, so it has S bias but lacks MX bias [3].

We use $\mathrm{MS}_{1,0}$ model in this paper, and from now on, we mention $\mathrm{MS}_{1,0}$ as MS for simplicity.

LS model Loose symmetry derived from the CP model is used as a parameter in the MS model to create a model with both S and MX biases. For LS, we take $\alpha = R^{\mathrm{CP}(p|q)}(t) = \frac{a(t)}{a(t)+c(t)}$ and $\beta = R^{\mathrm{CP}(p|\neg q)}(t) = \frac{b(t)}{b(t)+d(t)}$ in $R^{\mathrm{MS}_{\alpha,\beta}(q|p)}(t)$. According to Takahashi et al. (2010) [3], which sources the Japanese translation of Shinohara et al. (2007) [4], reported performing well in purely inductive and decision-theoretic (recursively inductive-deductive) tasks. The equation of LS is [3, 7]:

$$
R^{\mathrm{LS}(q|p)}(t) = \frac{a(t) + \frac{b(t)}{b(t)+d(t)} d(t)}{a(t) + \frac{b(t)}{b(t)+d(t)} d(t) + b(t) + \frac{a(t)}{a(t)+c(t)} c(t)} \tag{2.25}
$$

LS model satisfies XM and loosely satisfies S, MX biases, and ER [3].

LSR model The final model we use is loose symmetry under the rarity assumption (LSR) [3]. As we can understand from the name, LSR derives from the LS model. The rarity assumption [26] has been considered important in human causal inference. The assumption here is, that events p and q probabilities are small, which makes $d(t) = P(\neg p|\neg q)$ much higher than other components since the correlation of any two events is highly unlikely [22]. For example, any random event and you start your car in the morning are most likely not correlated with each other [22]. The equation of LSR can be achieved by diverging $d(t) \to \infty$ in the LS model [3].

$$\begin{aligned} R^{\mathrm{LSR}(q|p)}(t) &= lim_{d(t)\to\infty} R^{\mathrm{LS}(q|p)}(t) \\ &= lim_{d\to\infty} \frac{a(t) + \frac{b(t)}{b(t)+d(t)} d(t)}{a(t) + \frac{b(t)}{b(t)+d(t)} d(t) + b(t) + \frac{a(t)}{a(t)+c(t)} c(t)} \\ &= \frac{a(t) + b(t)}{a(t) + 2b(t) + \frac{a(t)}{a(t)+c(t)} c(t)} \end{aligned} \tag{2.26}$$

According to the Table 2.4 from Takahashi et al. (2010) [3], LSR fits the *Human* data from *H03* [23] slightly better than LS.

The Python code of the optimizers can be found in Appendix A or on the following GitHub page:

`https://github.com/mertsaru/Cognitive-GLVQ/blob/main/optimizer.py`

Updating learning rates with CGLVQ

The learning rate update of any CGLVQ optimizer is described in Figure 2.14. The methods first count the co-occurrence frequency for all the prototypes in each training sample to calculate R_i of each ω_i. After finding the co-occurrence frequency of the given ω_i, the R_i is calculated by the chosen cognitive science learning rate optimizer method according to Manome et al. (2021) [7].

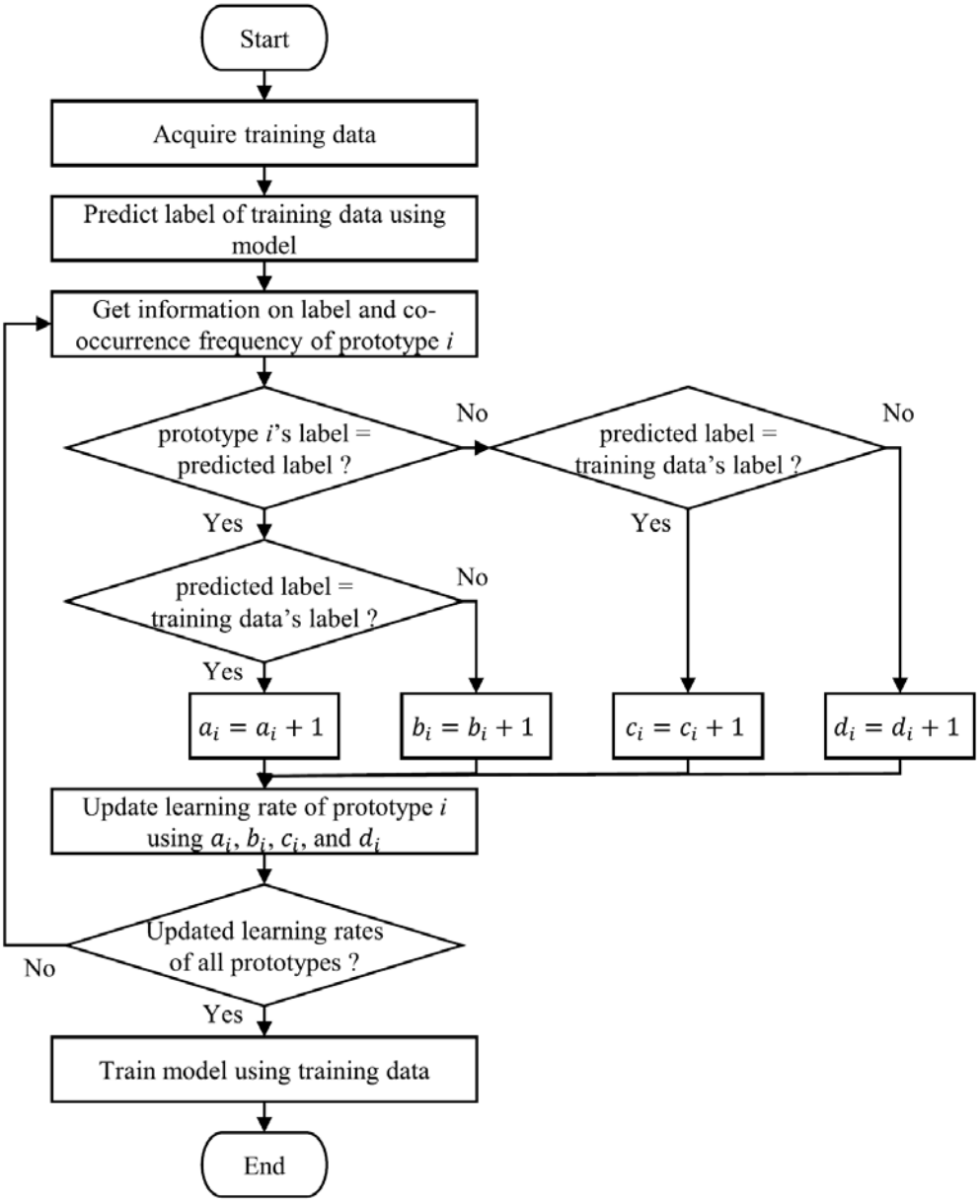


Figure 2.14: Training structure of CGLVQ models.

Note: From "Self-incremental learning vector quantization with human cognitive biases" by N. Manome, S. Shinohara, T. Takahashi, Y. Chen, and U. Chung, Scientific Reports 11(1), p. 3 (`https://doi.org/10.1038/s41598-021-83182-4`) [7].

After finding the $R_i(t)$ of the $\omega_i(t)$ we can calculate its $\epsilon_i(t)$ at time t by [7]:

$$\epsilon_i(t) = 1 - R_i(t) \tag{2.27}$$

However, the Equation (2.27) would give us $\epsilon_i(t) \in [0, 1]$, since $R_i(t) \in [0, 1]$ for any CGLVQ learning rate optimizer and $\forall t \in [0, \infty)$. To adjust the $\epsilon_i(t)$ of the model to the range of $\epsilon_i(0)$, we take $\epsilon_i(t)$ as:

$$\epsilon_i(t) = \epsilon(0)(1 - R_i(t)) \tag{2.28}$$

to make $\epsilon_i(t) \in [0, \epsilon(0)], \forall t \in [0, \infty)$.

The Python code of CGLVQ can be found in Appendix A or on the following GitHub page:
`https://github.com/mertsaru/Cognitive-GLVQ/blob/main/cognitive_GLVQ.py`

2.5 Test Measures

When training the data, we need a measure to understand how useful the machine learning classification model is. For that, we have different types of test measures. The most used and well-known method is the accuracy method, which is useful when the classes have an equal or close number of (balanced) samples in the dataset. If the sample numbers for classes are not equal (imbalanced), we have an F score to have a prediction power of the method. According to Kaden et al. (2014) [27], the F score performs better for the dataset with imbalanced class samples than for accuracy.

Another way to deal with datasets that have imbalanced class samples is to create artificial data for the missing classes. These methods are either using Generative AI to create artificial data from scratch or using data augmentation methods on the existing data to stretch, discolor, rotate, and more on the existing data to create new samples. However, these artificial data creation methods are not always helpful. There are some datasets that one cannot create artificially. Secondly, using a Generative adversarial network (GAN) to generate new data would also have problems since we need labels for generated data. However, finding the labels in the original data is hard for the human eye, and we do not have a label-generating model for the unlabeled data. So, we do not discuss how to create data artificially, but we talk about what we can do when we have a dataset where the classes are imbalanced.

Accuracy score

Accuracy is easy to understand and apply. We predict everything in the test set after training the model with the training set. To get the accuracy score, we divide the correctly predicted samples by the number of all samples in the test set, like in Equation 2.29.

$$\text{accuracy score} = \frac{\text{\# correct classification}}{\text{\# total samples}} \tag{2.29}$$

As we mentioned earlier, the accuracy is reasonable when the samples for each class are balanced. Let us examine what would happen if we use accuracy on an imbalanced sample of classes. For this example, we assume that the Test dataset 1 has 100 samples where 99 have class label 0 and 1 has class label 1. Let the model for classification label every input to class label 0. Then, according to the accuracy equation, the model's accuracy would be 99% for Test dataset 1 (2.30.1). If we calculate the accuracy for the same model but with an evenly distributed test dataset, where Test dataset 2 has 50 samples for class 0 and 50 samples for class 1 out of 100, the accuracy would be 50% with Test dataset 2 (2.30.2). So, with an unequal number of samples for each class, accuracy would be a misinterpretation of the model's prediction reliability.

Test set 1 accuracy score:

$$\frac{\text{\# correct classification}}{\text{\# total samples}} = \frac{99}{100} = 99\% \tag{2.30.1}$$

Test set 2 accuracy score:

$$\frac{\text{\# correct classification}}{\text{\# total samples}} = \frac{50}{100} = 50\% \tag{2.30.2}$$

F score

F1 score, or more generally F_β score, comes in handy when we cannot rely on accuracy measurement. The F1 score is a subclass of F_β score, which is calculated by recall (ρ) and precision (π) of the model. We need to dive into prediction cases to understand the recall and precision. We start with the binary classification model.

Let us start naming the classes as class 0 and class 1. For every prediction in a binary classification model, we have four options:

- The model predicts the sample $\mathbb{x}$ belongs to class 0, and it is correct (True positive (TP))
- The model predicts the sample $\mathbb{x}$ belongs to class 0, and it is wrong (False positive (FP))
- The model predicts the sample $\mathbb{x}$ belongs to class 1, and it is correct (True negative (TN))
- The model predicts the sample $\mathbb{x}$ belongs to class 1 and it is wrong (False negative (FN))

If $L(\mathbb{x})$ represents the label of $\mathbb{x}$ and $L(\omega^{\mathbb{x}})$ represents the prediction label of $\mathbb{x}$, then we can see the confusion matrix of class 0 in Table 2.5.

	$L(\mathbb{x}) = 0$	$L(\mathbb{x}) \neq 0$
$L(\omega^{\mathbb{x}}) = 0$	TP	FP
$L(\omega^{\mathbb{x}}) \neq 0$	FN	TN

Table 2.5: Co-occurrence frequency for each prototype ω_i.

Then recall (ρ) and precision (π) follows as:

$$\rho = \frac{TP}{TP + FP} \tag{2.31}$$

$$\pi = \frac{TP}{TP + FN} \tag{2.32}$$

and the F_β-score for the model would be:

$$F_{beta} = \frac{(1+\beta^2)\cdot\pi\cdot\rho}{(\beta^2\cdot\pi)+\rho} \tag{2.33}$$

Since we use the F1 score, the measurement we use in our results would be:

$$F_1 = \frac{2\pi\cdot\rho}{\pi+\rho} \tag{2.34}$$

The problem with F scores is that the score's output does not tell us how reliable the model is. We know with what percentage the model predicts the samples for the accuracy score, but the F scores do not give us such an answer. Most of the time, the F score of one class is not equal to the second class's F1 score in classification problems. If we check the example Table 2.6, for class 0, the F1 score is equal to 0.8. The Table 2.6 can be rewritten for class 1 as Table 2.7. However, when we check the F1 score of class 1 for the same table, the score would be $0.\bar{6}$. So, it is hard to understand what the F1 score means. However, it gives us a nice comparison between models and reflects the accuracy change of the model while using imbalanced samples for classes in the dataset.

Class 0

	$L(\mathbb{x}) = 0$	$L(\mathbb{x}) \neq 0$
$L(\omega^{\mathbb{x}}) = 0$	50	5
$L(\omega^{\mathbb{x}}) \neq 0$	20	25

Table 2.6: Example: confusion matrix for class 0.

Class 1

	$L(\mathbb{x}) = 1$	$L(\mathbb{x}) \neq 1$
$L(\omega^{\mathbb{x}}) = 1$	25	20
$L(\omega^{\mathbb{x}}) \neq 1$	5	50

Table 2.7: Example: confusion matrix of the Table 2.6, rewritten for class 1.

Regarding the Table 2.6 and Table 2.7 and using the F1 score on same $\omega^{\mathbb{x}}$, we achieve the following equations for each class:

$$F_1(\text{class = 0}) = \frac{2\cdot\frac{50}{55}\cdot\frac{50}{70}}{\frac{50}{55}+\frac{50}{70}} = 0.8 \tag{2.35}$$

$$F_1(\text{class = 1}) = \frac{2\cdot\frac{25}{45}\cdot\frac{25}{30}}{\frac{25}{45}+\frac{25}{30}} = 0.\bar{6} \tag{2.36}$$

To get a single score for the dataset, we combine the F1 scores. F1 scores can be combined by taking the average of the scores or the average by multiplying each class's

weight with their corresponding F1 score, or we can examine the F1 scores for each class in a dataset separately. The latter would be hard to comprehend, and the average of all F1 scores has its own problems. In our analysis, we take the F1 score for each model epoch as the weighted average of the class F1 score. This method can be used to find common F1 scores for multiclassification problems.

So, to use accuracy as an effective measure, we conduct separate experiments. We select an equal number of samples for classes in the dataset to get accurate accuracy scores. Besides that, we also use the dataset with imbalanced class samples and examine the F1 score. By conducting two experiments with different sample sizes, we would investigate the models we use in different situations and better understand the models' powers.

2.6 The Experiments

We used two experiments on the datasets in this paper. For both of the experiments, each dataset uses the same prototype set. First, we took an equal number of samples for each class in every dataset. In this experiment, the accuracy score is essential to look at. For our second test, we distributed the samples unevenly between classes. The second experiment compares F1 scores. We used two experiments with different sample sizes because, in real-world data, we do not have equally divided sample sizes most of the time. Sometimes, with unbalanced sample sizes, we can get different results than balanced sample sizes, and we want to see the models' performance in both cases.

We ran the same experiment three times with different $\epsilon(0)$: 0.1, 0.03, and 0.01. We randomly selected the prototypes and sample sets for each dataset in every experiment. After selecting the samples for the test set, training set, and prototypes, we used the same samples for each test and dataset. The prototypes are selected by comparing accuracy scores on experiment 1 using OGLVQ with $\epsilon(0) = 0.1$ and taking the prototype sample with the highest accuracy score among 20 different random ω set and dataset sampling. Prototype and dataset selection is still (semi-)random since we use seeds in random selection.

We have two plot graphs in both experiments for each test: the corresponding measure score of the experiment and learning rate values. We look at the learning rate values to see if the model is learning appropriately or not. Since ϵ_i differs for each prototype in GLVQ models, we have many learning rate plots in our graphs, differentiated labels by color. We associate learning rate in CGLVQ models with learning since learning rate in CGLVQ is positively related to a ($L(\omega_i) = L(\omega^{\text{✗}}) \implies L(\text{✗}) = L(\omega^{\text{✗}})$) in the Table 2.2 in all the models, and a is correctly predicted. So, an increase of a would decrease the $\epsilon_i(t)$ since $\epsilon_i(t) = \epsilon(0) \cdot (1 - R)$, and R in positive relation with a for all cognitive science learning rate optimizers.

Chapter 3

Results

The first thing to realize when looking at the learning rate graphs of the models is that every line on the learning rate graph of OGLVQ models represents each prototype's ϵ_i while in the CGLVQ model, it turns into class-based ϵ. We did nothing different for learning rate graphs for the CGLVQ; the graph still shows the ϵ_i of each ω_i, but every ω that shares the same class has the exact ϵ_i. This observation is because, in standard GLVQ or OGLVQ, ϵ_i are updated for the ω^+ and ω^- for that sample. However, in the CGLVQ models, ϵ update is based on classes. This ϵ update of CGLVQ happens because all of the ω_i in the same class share exact co-occurrence tables.

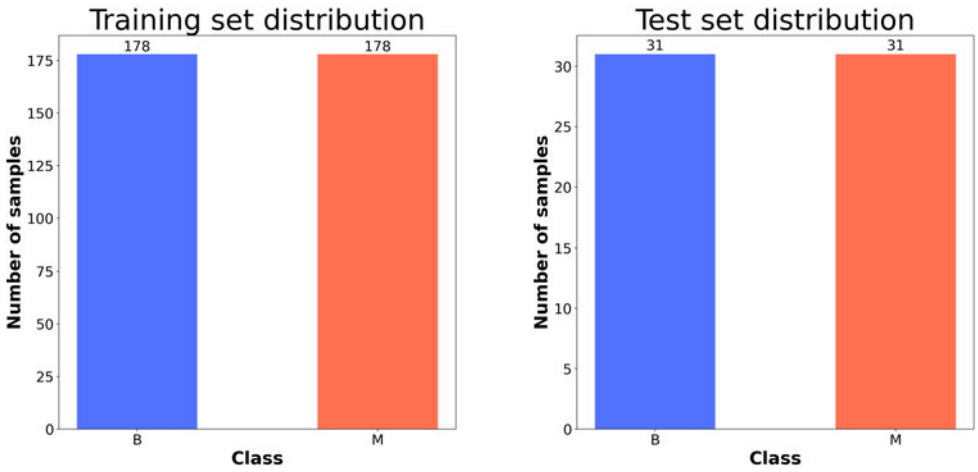
3.1 Experiment 1 (Balanced Dataset)

3.1.1 Breast Cancer Wisconsin dataset

Prototypes for each class: 3

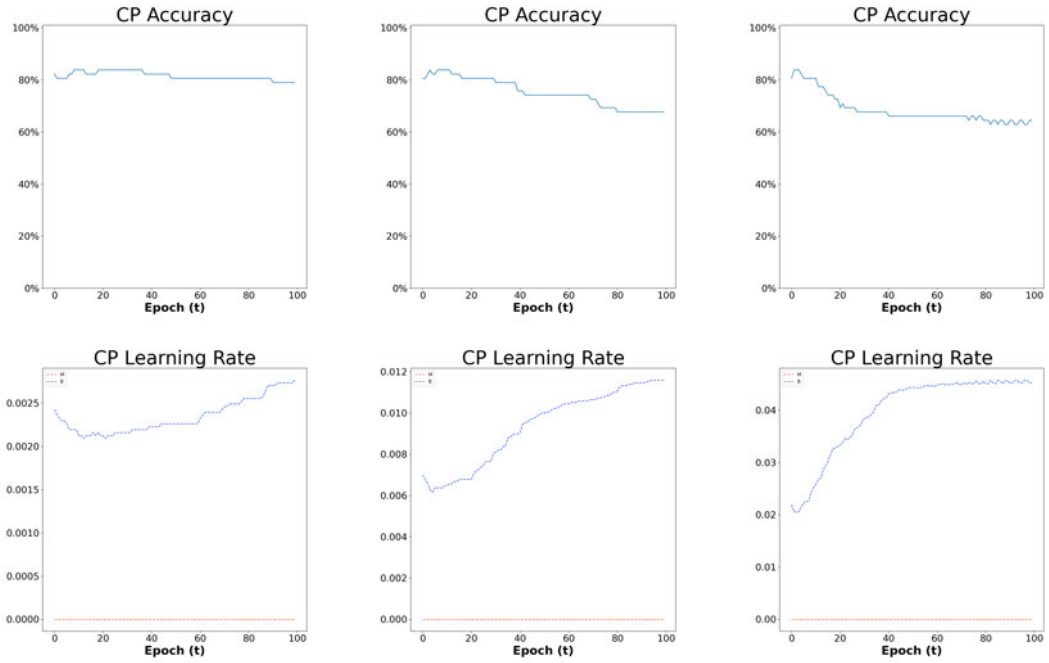
The learning process reflects the models' learning rate graph. The change in the learning rate graph indicates that the model adapts to a given dataset. We would like to see decreasing ϵ_i for all prototypes, which means the given model is learning positively. For *Breast Cancer Wisconsin* dataset on experiment 1 with OGLVQ, every CGLVQ model except the CP model increases accuracy and decreases ϵ_i values, which brings positive learning as we can see the results of the models below. CGLVQ models shows good ϵ_i curves with low $\epsilon(0)$, 0.01. The CP model also shows positive learning at the beginning of the process with $\epsilon(0) = 0.01$ but decreases accuracy afterward and increases ϵ_B of class "Benign." Still, CP is adapting to the model but not doing a good job regarding other models. Performance-wise, CGLVQ models except CP shows similar performance to OGLVQ.

Here is a small note to be careful about the axis of the given plots. With smaller $\epsilon(0)$, it is harder to visualize the motion of learning rates in the same scale as higher $\epsilon(0)$. So, the changes on the learning rate graphs do not reflect equal change with different $\epsilon(0)$. We investigate if there is a change in learning rates and, if so, in which direction.



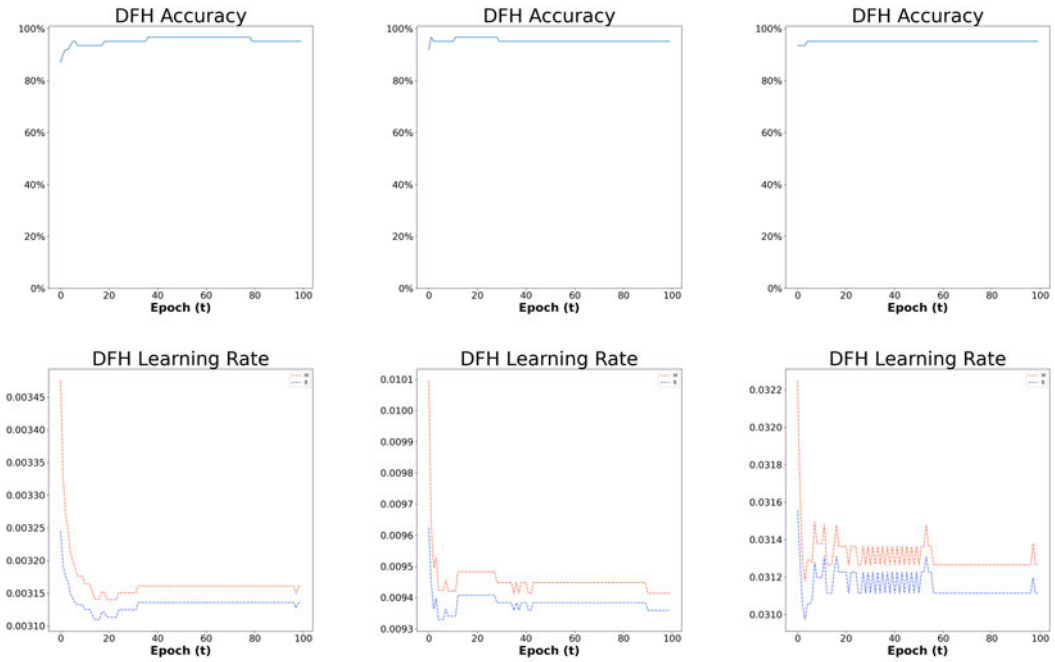
(a) Training set (b) Test set

Figure 3.1: *Breast Cancer Wisconsin* balanced dataset sample distribution.



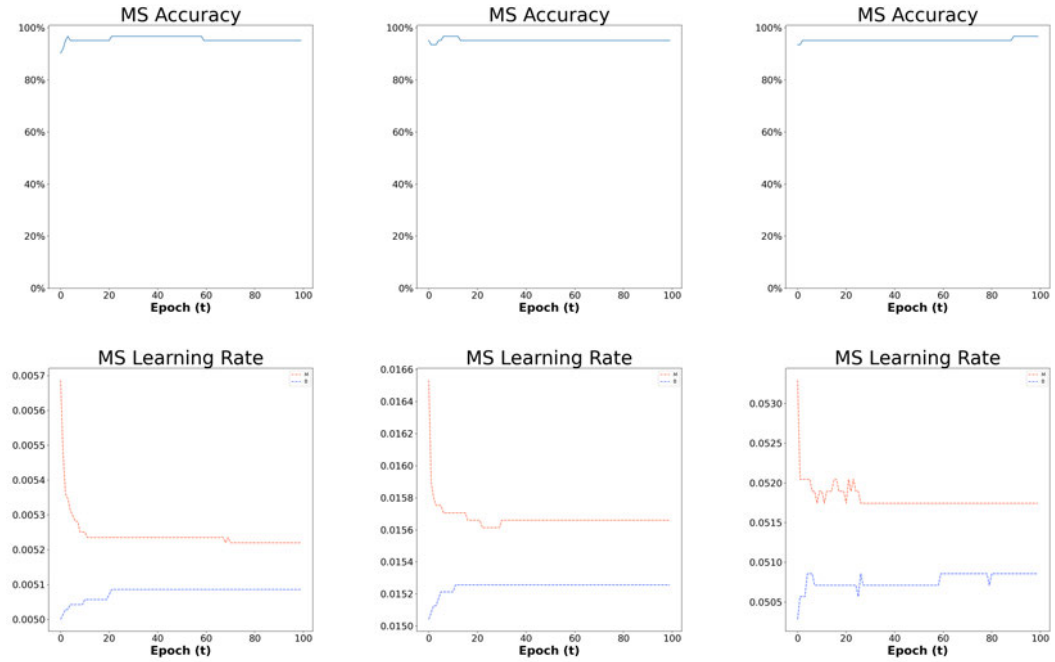
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.2: *Breast Cancer Wisconsin* dataset accuracy score and learning rate results under CP model using balanced dataset.



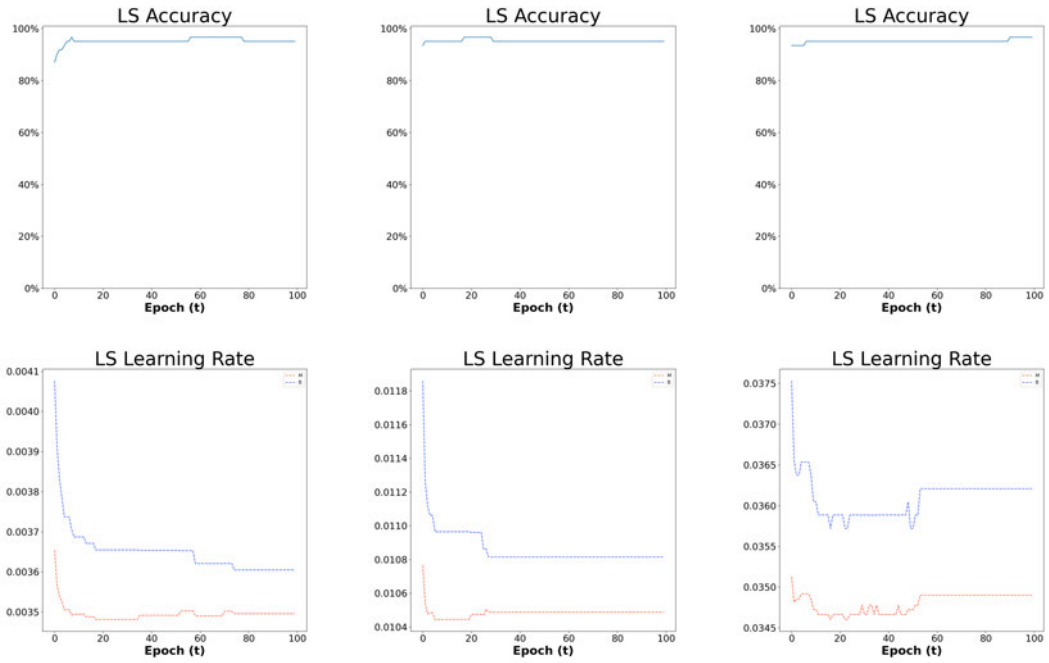
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.3: *Breast Cancer Wisconsin* dataset accuracy score and learning rate results under DFH model using balanced dataset.



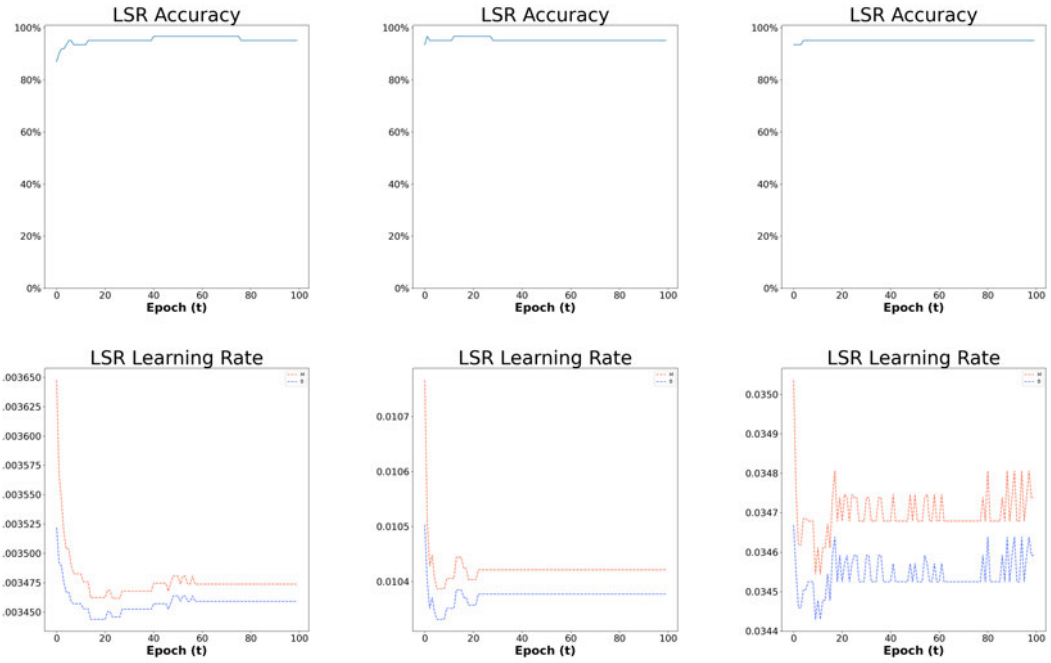
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.4: *Breast Cancer Wisconsin* dataset accuracy score and learning rate results under MS model using balanced dataset.



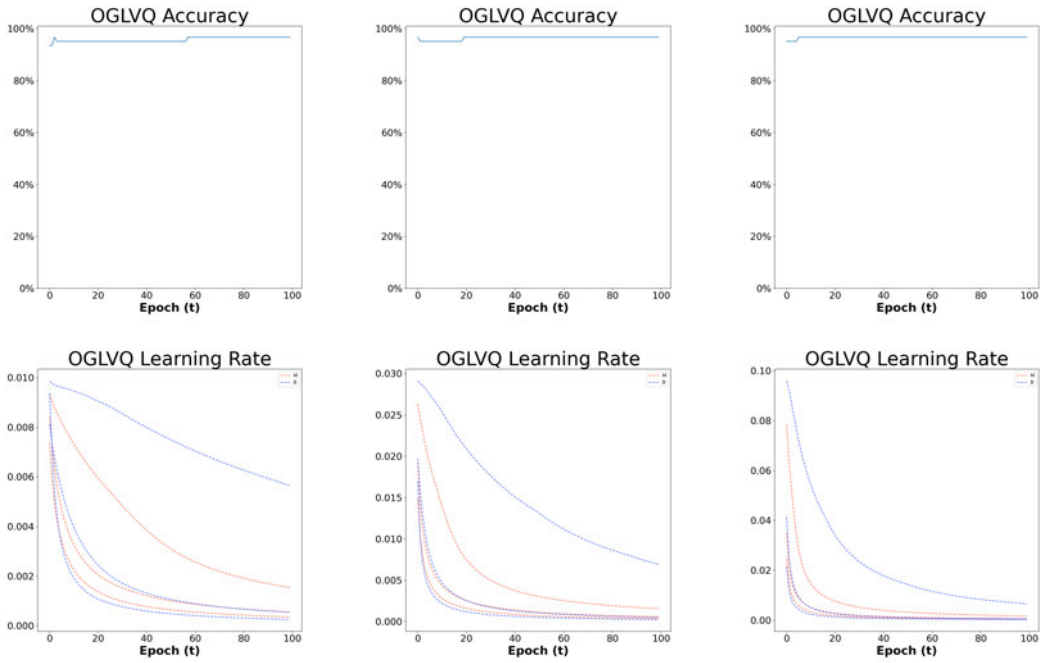
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.5: *Breast Cancer Wisconsin* dataset accuracy score and learning rate results under LS model using balanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.6: *Breast Cancer Wisconsin* dataset accuracy score and learning rate results under LSR model using balanced dataset.



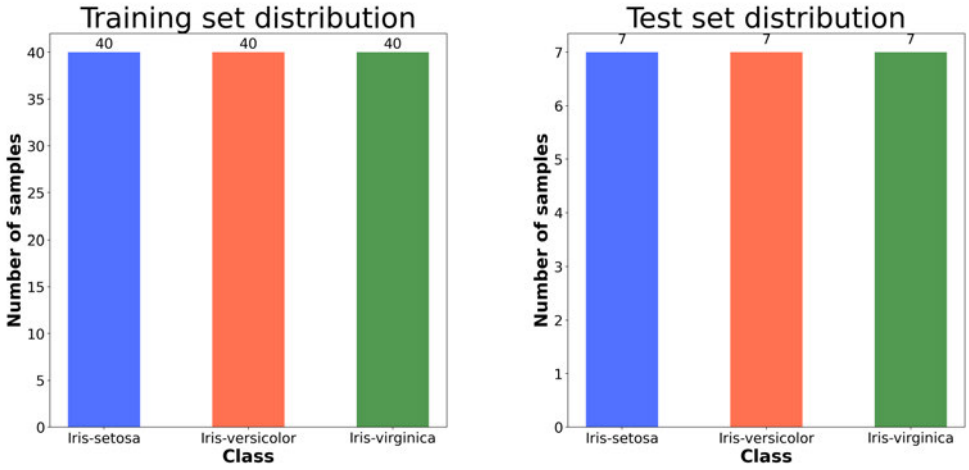
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.7: *Breast Cancer Wisconsin* dataset accuracy score and learning rate results under OGLVQ model using balanced dataset.

3.1.2 Iris dataset

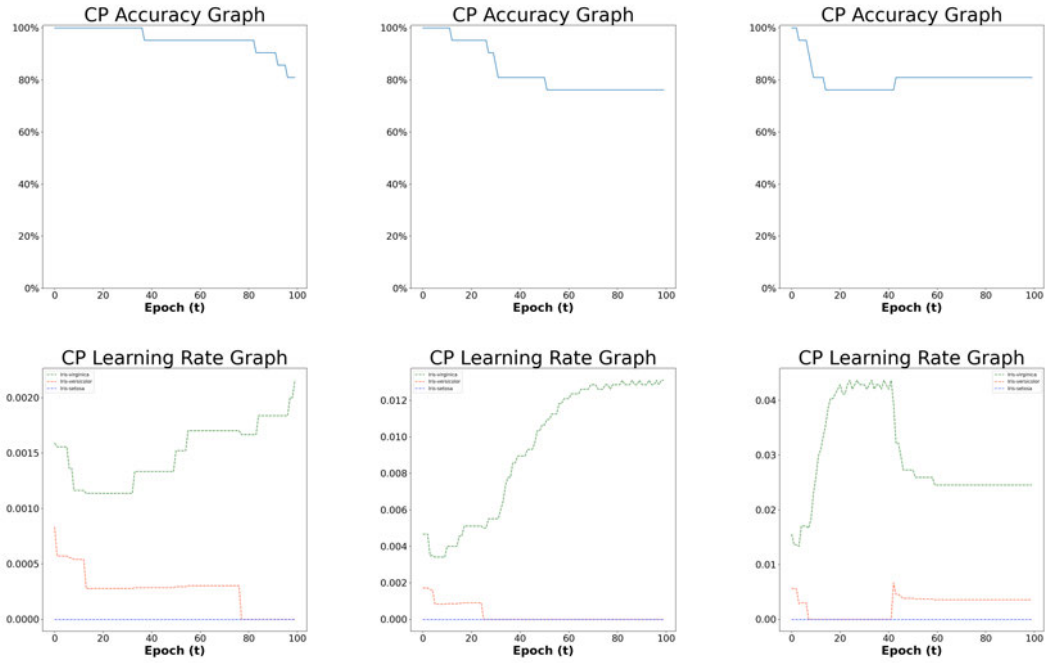
Prototypes for each class: 3

If we check the results of the *Iris* dataset, we see all models' accuracy scores start at 1 for at least one initial rate test. That indicates the dataset is already divided nicely for classification. Even though accuracy is high initially, that does not mean the model cannot learn. The model tries to find a nice line between data samples to divide samples as perfectly as possible. We see some decrease in accuracy scores of the CP model for any $\epsilon(0)$, which also affects the ϵ_i values during the training of the experiment. With accuracy getting lower, ϵ_i for all prototypes get higher, which indicates the model is learning badly. It classifies worse with every epoch that passes. For other CGLVQ models, we see that accuracies are stable at high accuracy scores, between 90% and 100%. Besides, the learning rates stuck in a constant value (for DFH($\epsilon(0) = 0.01$), DFH($\epsilon(0) = 0.1$), MS($\epsilon(0) = 0.1$), LS($\epsilon(0) = 0.1$), LSR($\epsilon(0) = 0.01$), LSR($\epsilon(0) = 0.1$)) or zigzagging between a value range (for DFH($\epsilon(0) = 0.03$), MS($\epsilon(0) = 0.01$), MS($\epsilon(0) = 0.03$), LS($\epsilon(0) = 0.01$), LS($\epsilon(0) = 0.03$), LSR($\epsilon(0) = 0.03$)). We can interpret this zigzagging effect as the model is indecisive about where to draw the line between the classes since at least one test sample is close to two different classes, and prototypes cannot decide. So, there is not much learning going on in CGLVQ models. However, in the OGLVQ model with low $\epsilon(0)$ such as 0.01 and 0.03, we see better ϵ_i curves diminishing to 0, indicating that the model adapts and learns greatly.



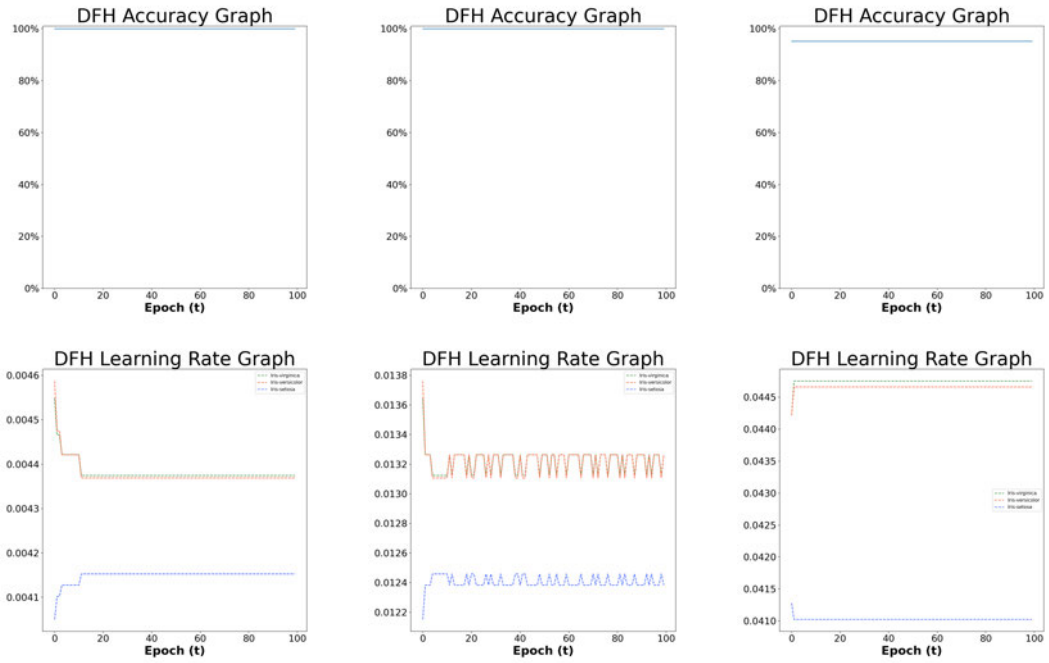
(a) Training set (b) Test set

Figure 3.8: *Iris* balanced dataset sample distribution.



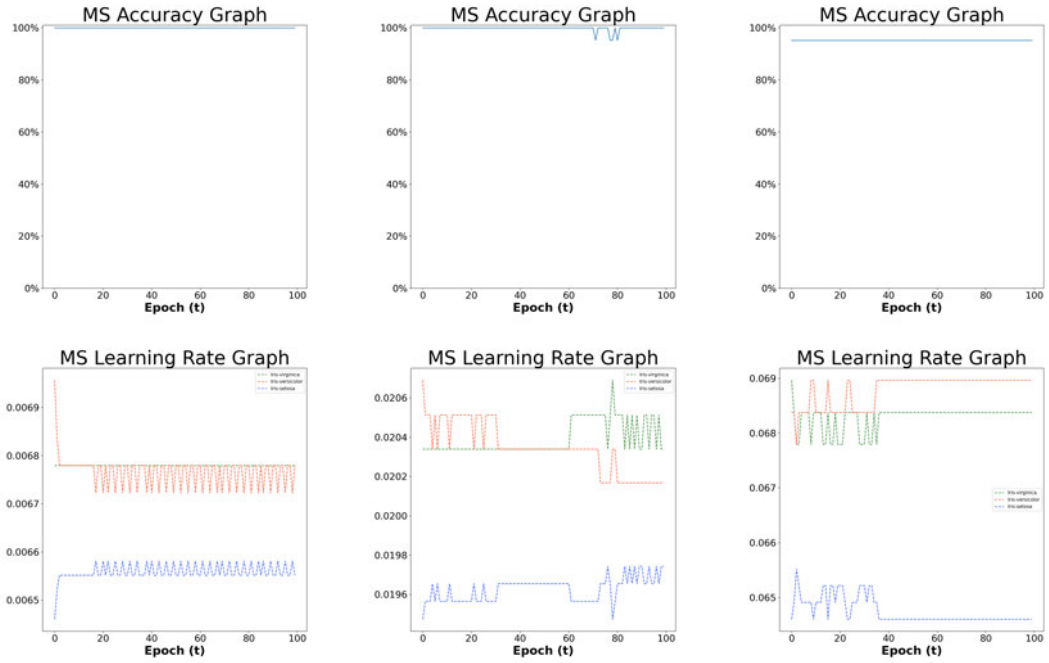
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.9: *Iris* dataset accuracy score and learning rate results under CP model using balanced dataset.



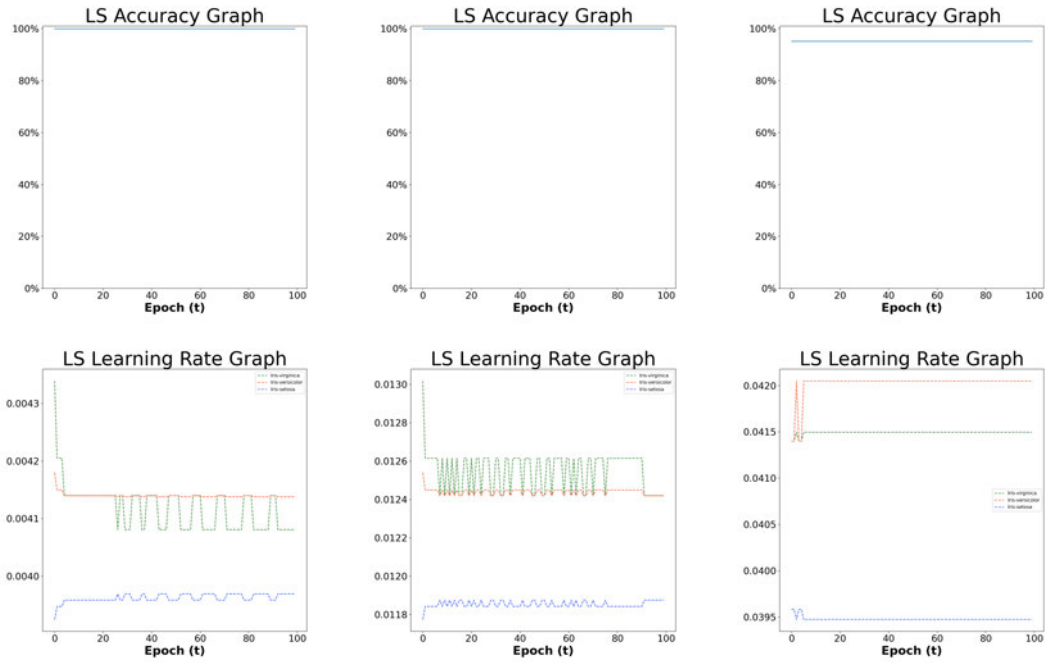
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.10: *Iris* dataset accuracy score and learning rate results under DFH model using balanced dataset using balanced dataset.



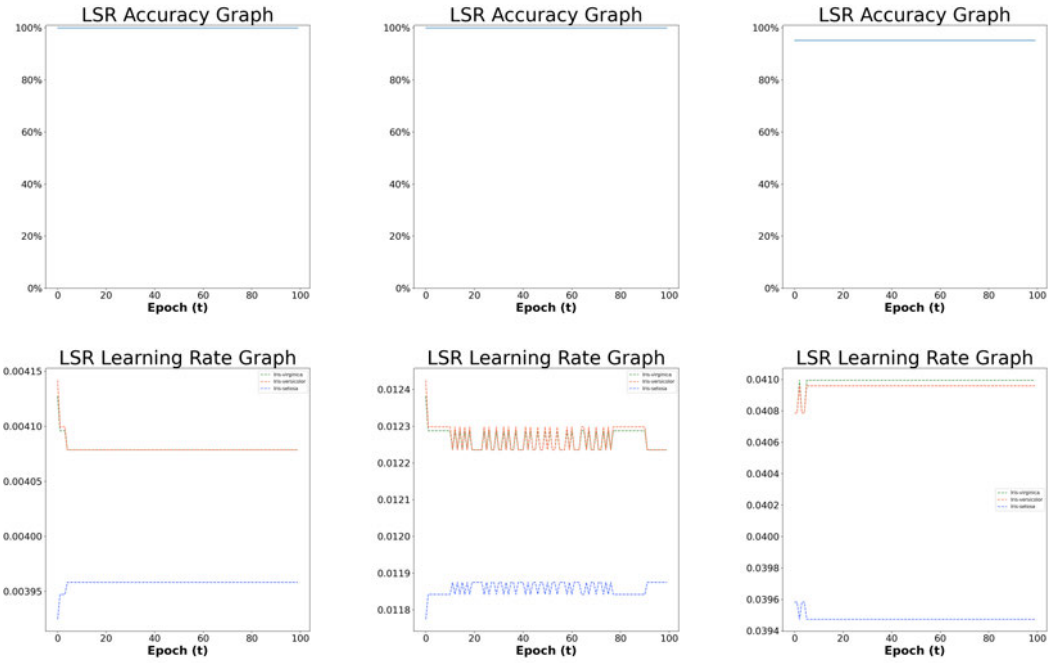
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.11: *Iris* dataset accuracy score and learning rate results under MS model using balanced dataset.



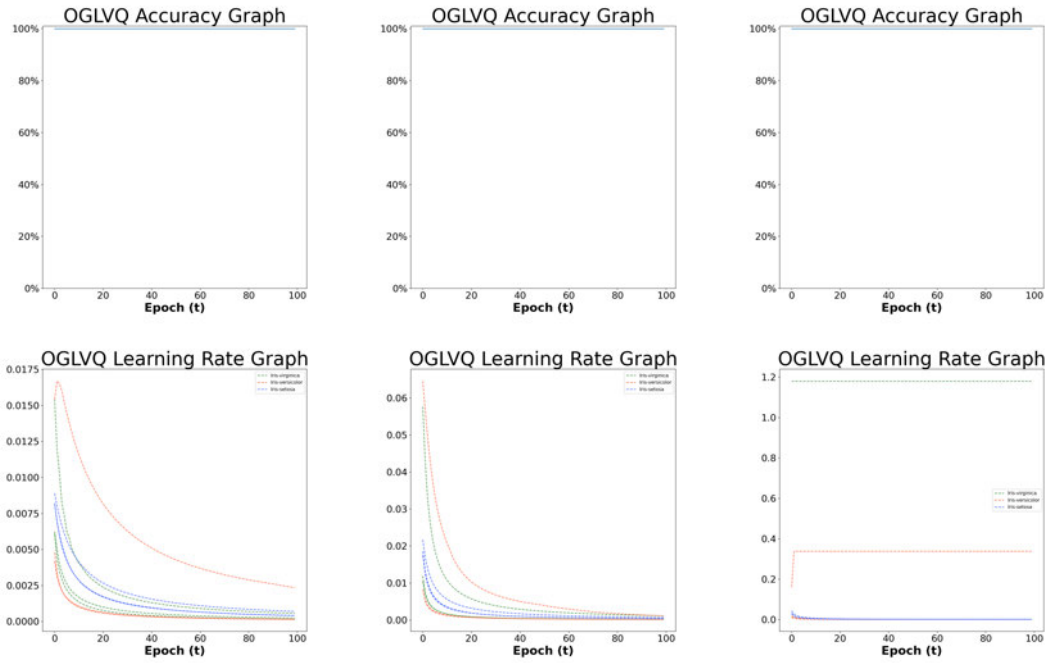
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.12: *Iris* dataset accuracy score and learning rate results under LS model using balanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.13: *Iris* dataset accuracy score and learning rate results under LSR model using balanced dataset.



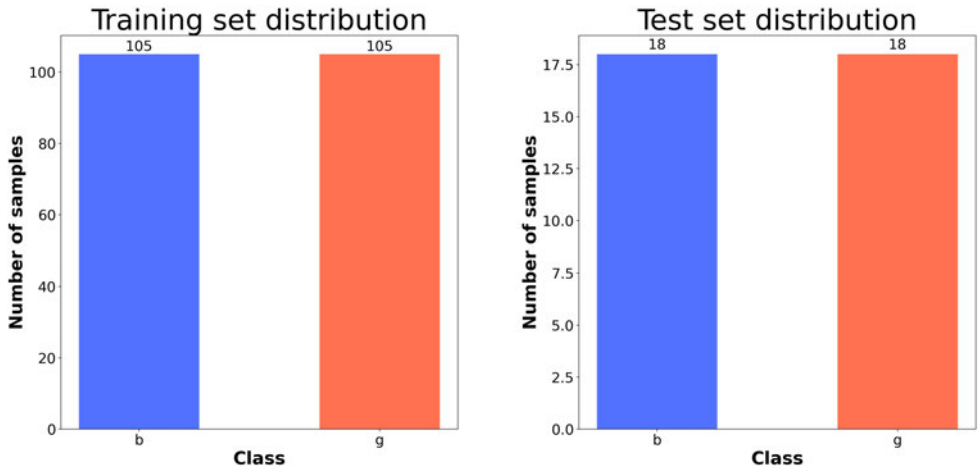
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.14: *Iris* dataset accuracy score and learning rate results under OGLVQ model using balanced dataset.

3.1.3 Ionosphere dataset

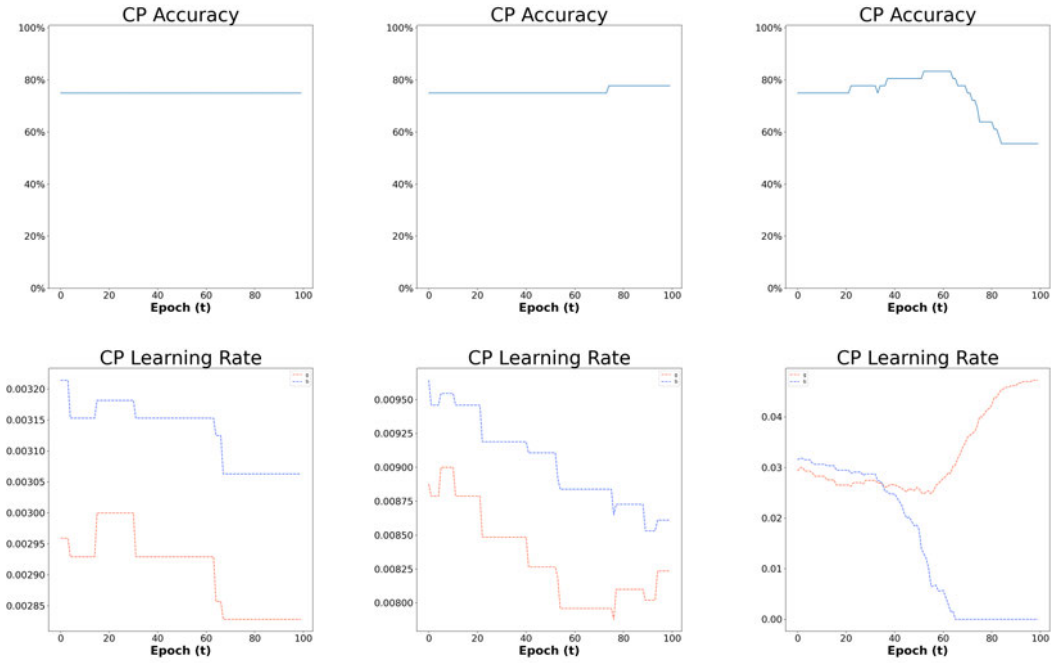
Prototypes for each class: 3

Ionosphere dataset results of experiment 1 for all models look promising, even though we do not have high or constantly increasing accuracy scores. All the models learn with proper $\epsilon(0)$ (CP with $\epsilon(0)$ 0.01 and 0.03, MS with $\epsilon(0)$ 0.01 and all other CGLVQ models) during their training period, as we can see by the decrease of ϵ_i values. The CP model with $\epsilon(0) = 0.1$ graph looks bad for learning as we can see on Figure 3.16c because of the increasing ϵ_g, but ϵ_g curve might be the result of the high $\epsilon(0)$ since other CP models with lower $\epsilon(0)$ have a nice ϵ_g curves. So $\epsilon(0) = 0.1$ might be overshooting for the CP model in this case. Additionally, the OGLVQ models' learning rate graphs do not look promising (Figure 3.21). Even if we see an increase in accuracy, learning rates of class "Bad" during the learning tends to increase or stay stable, which ends up in bad decision-making for class "Bad" with the model.



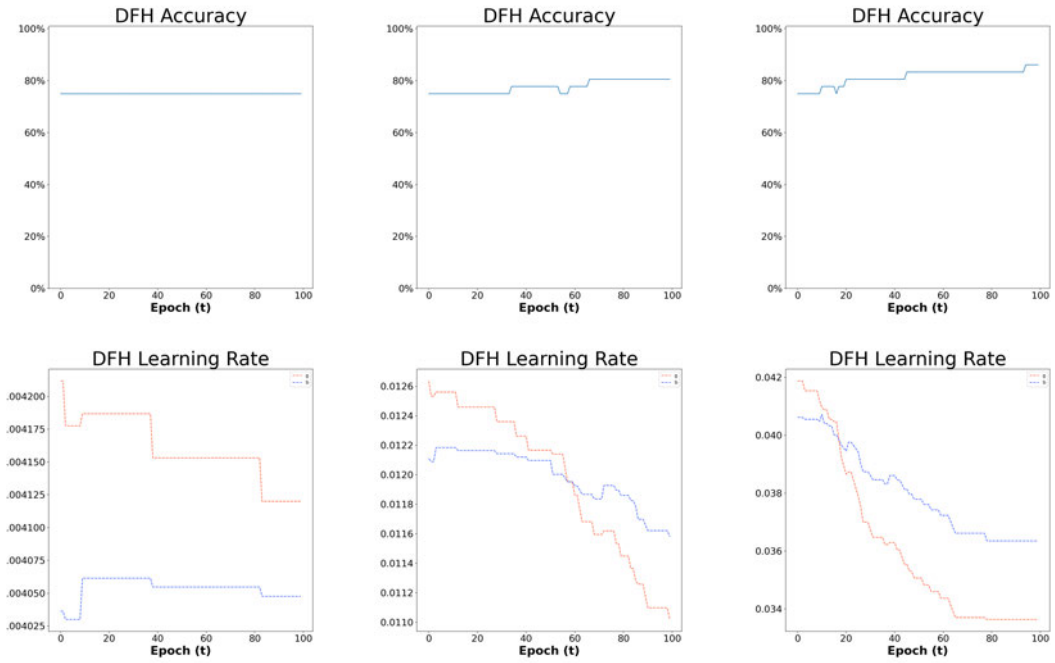
(a) Training set (b) Test set

Figure 3.15: *Ionosphere* balanced dataset sample distribution.



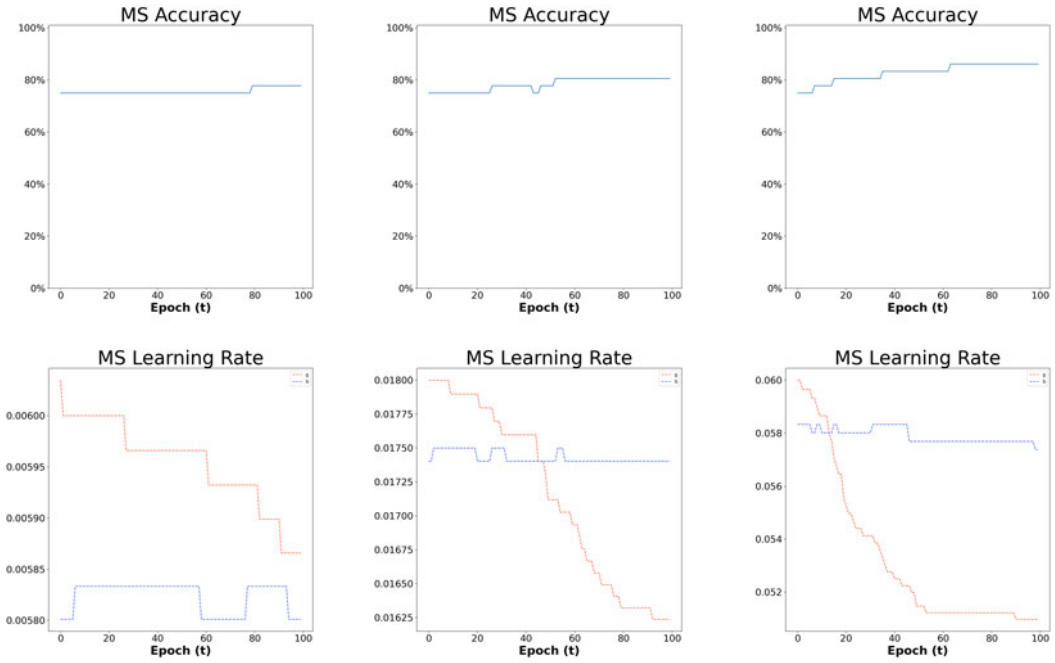
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.16: *Ionosphere* dataset accuracy score and learning rate results under CP model using balanced dataset.



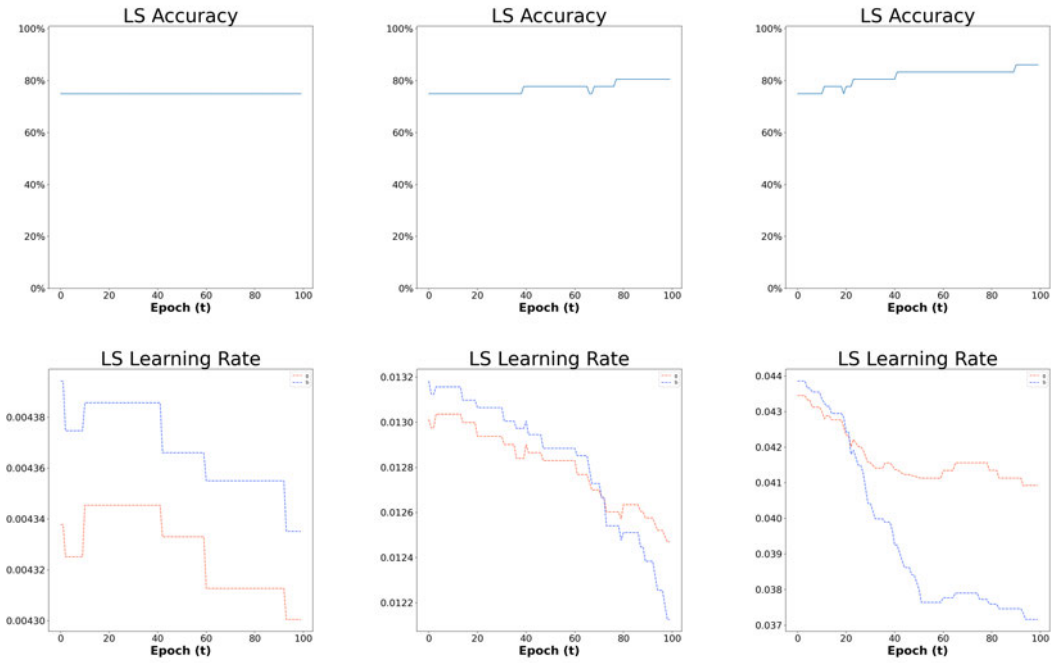
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.17: *Ionosphere* dataset accuracy score and learning rate results under DFH model using balanced dataset using balanced dataset.



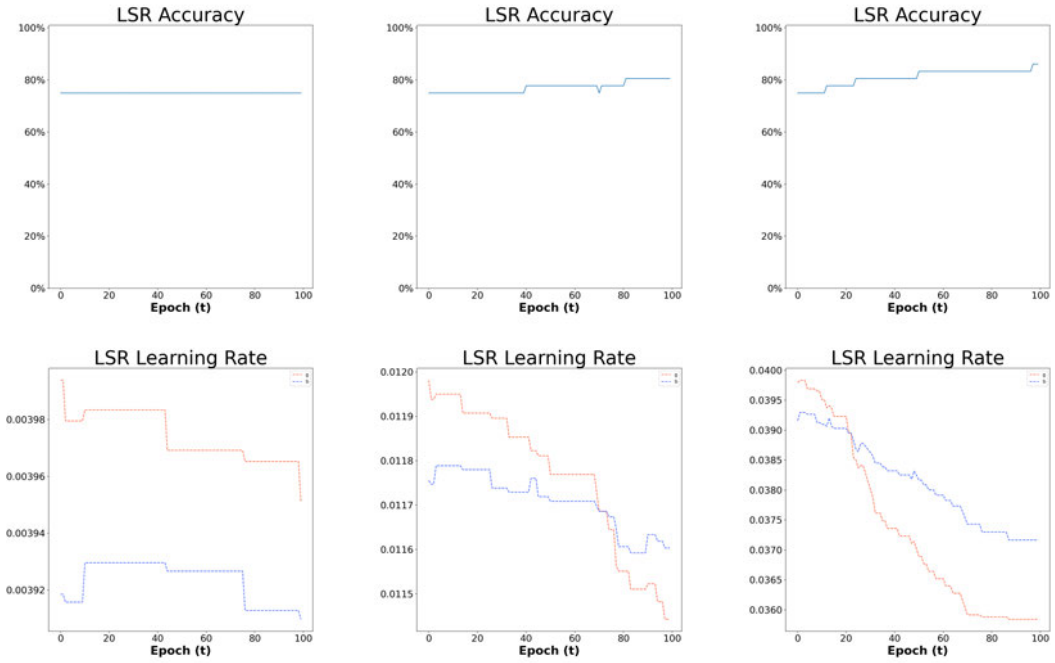
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.18: *Ionosphere* dataset accuracy score and learning rate results under MS model using balanced dataset.



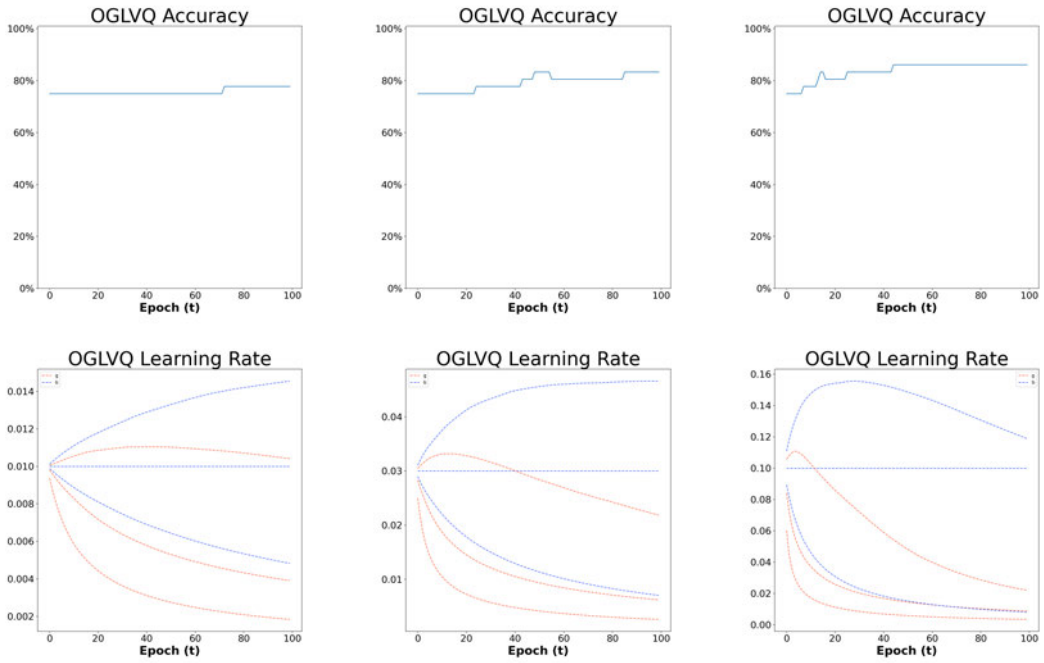
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.19: *Ionosphere* dataset accuracy score and learning rate results under LS model using balanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.20: *Ionosphere* dataset accuracy score and learning rate results under LSR model using balanced dataset.



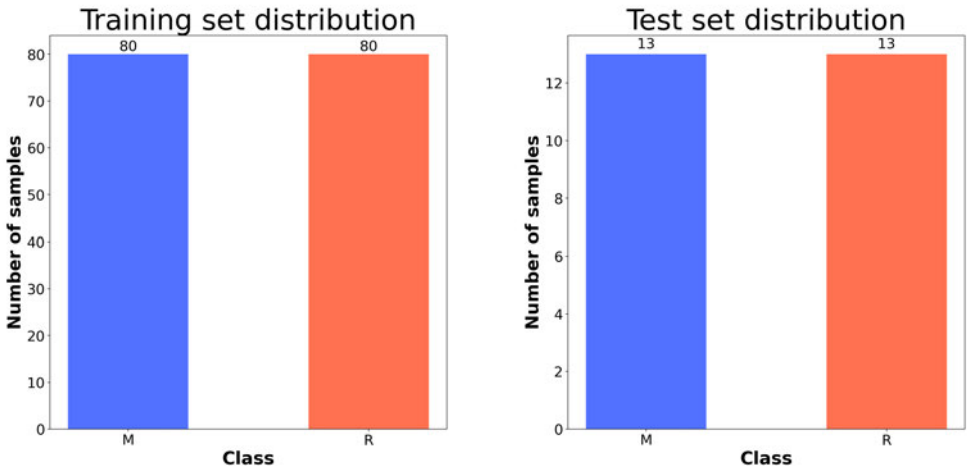
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.21: *Ionosphere* dataset accuracy score and learning rate results under OGLVQ model using balanced dataset.

3.1.4 Sonar dataset

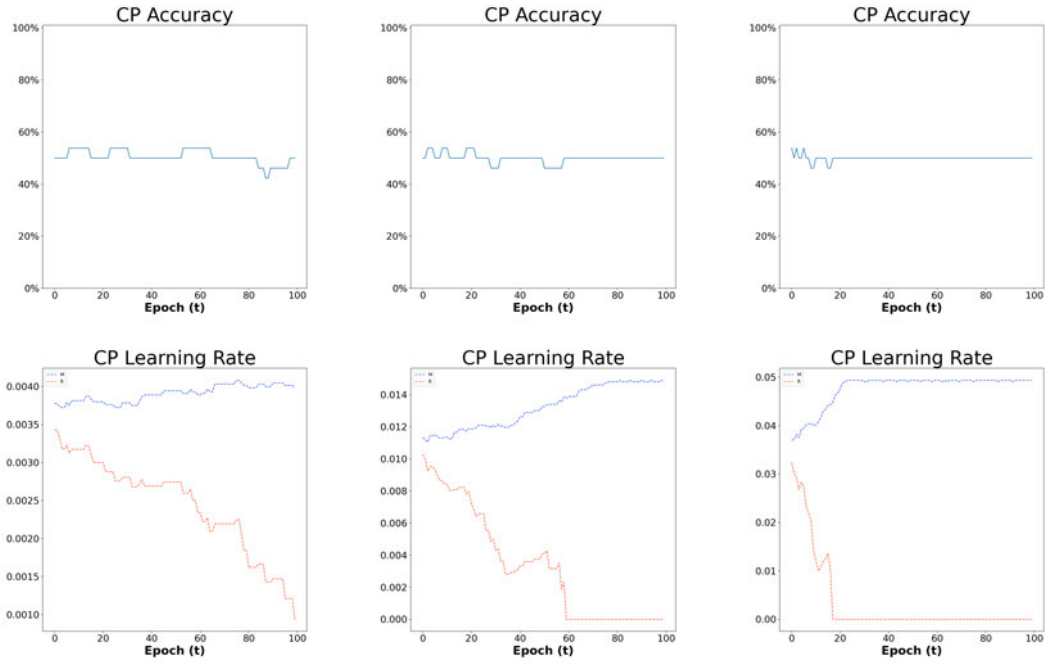
Prototypes for each class: 3

Like earlier datasets, all models have nice ϵ_i curves and an increase in accuracy, except CP models in this experiment. For all $\epsilon(0)$ of the CP model, ϵ_M for class "Mines" increases. That increase happens because ω_M of "Mines" could not predict its class well anymore, which results in lower accuracy for the model. Still, this might be because of the high $\epsilon(0)$, but other models have better accuracies with their best $\epsilon(0)$. In this dataset, we get a nice ϵ_i decrease and accuracy score increase in the models DFH, MS, LS, LSR, and OGLVQ. However, in *Sonar* dataset, we see again that the CP model is the least favorable among other CGLVQ models.



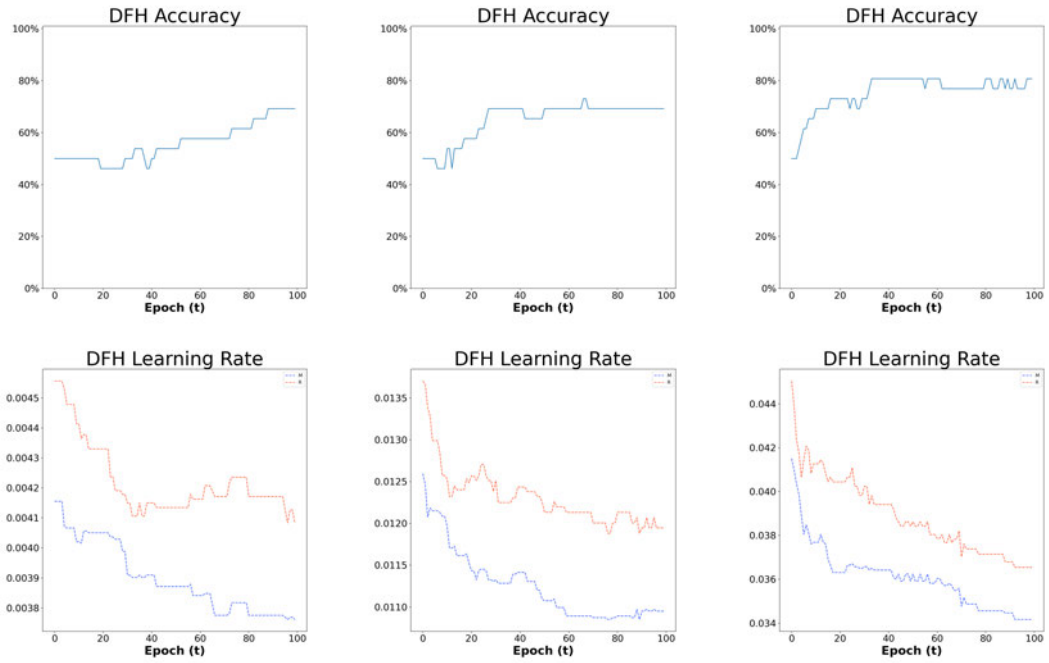
(a) Training set (b) Test set

Figure 3.22: *Sonar* balanced dataset sample distribution.



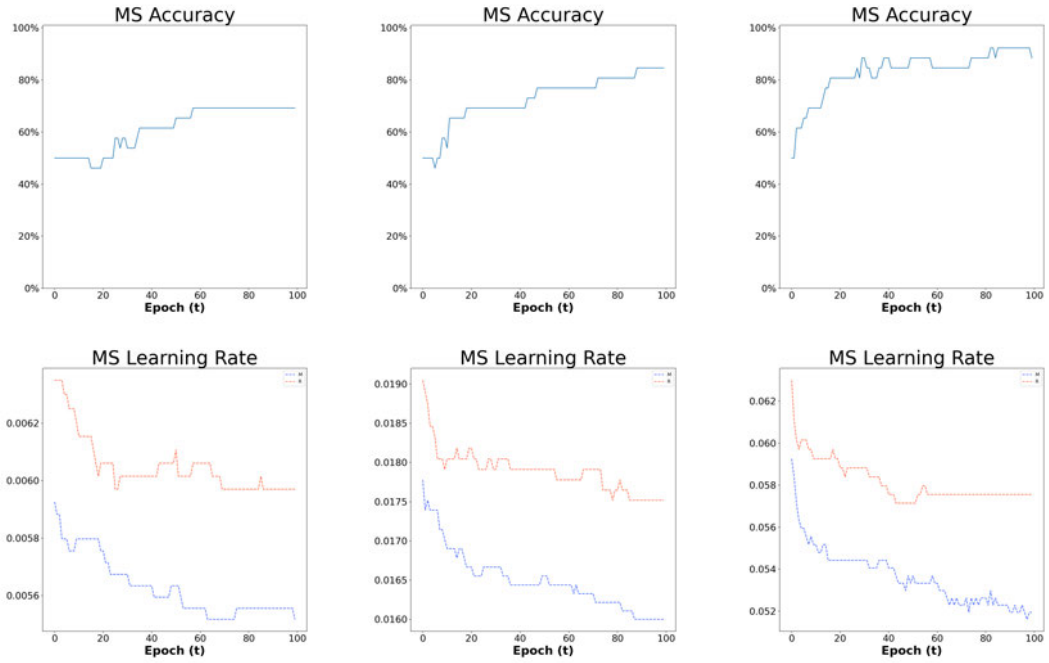
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.23: *Sonar* dataset accuracy score and learning rate results under CP model using balanced dataset.



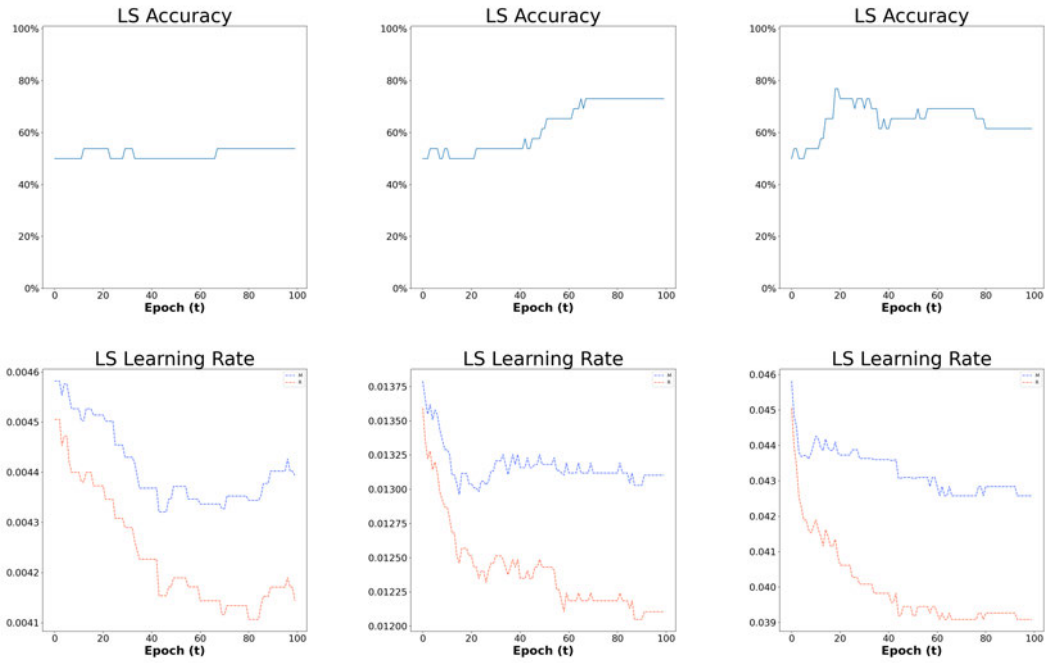
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.24: *Sonar* dataset accuracy score and learning rate results under DFH model using balanced dataset using balanced dataset.



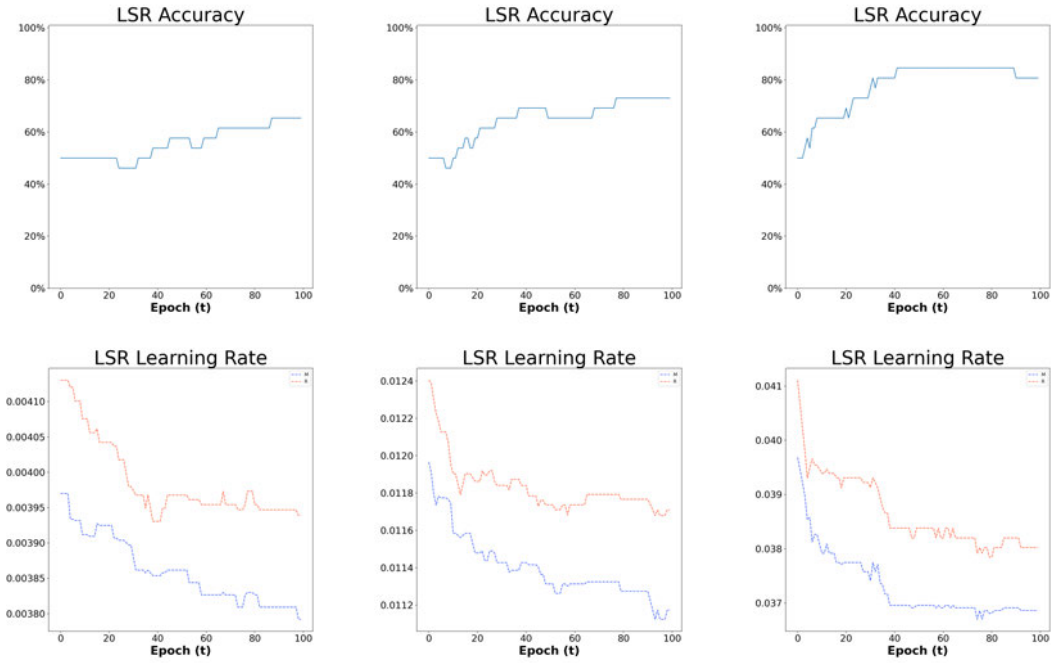
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.25: *Sonar* dataset accuracy score and learning rate results under MS model using balanced dataset.



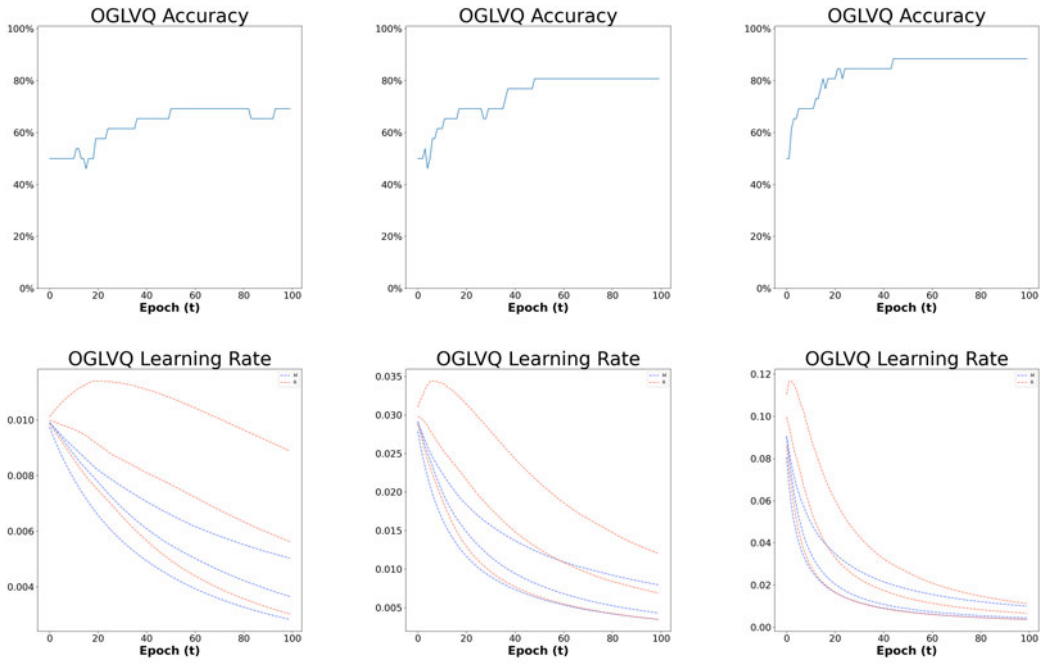
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.26: *Sonar* dataset accuracy score and learning rate results under LS model using balanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.27: *Sonar* dataset accuracy score and learning rate results under LSR model using balanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

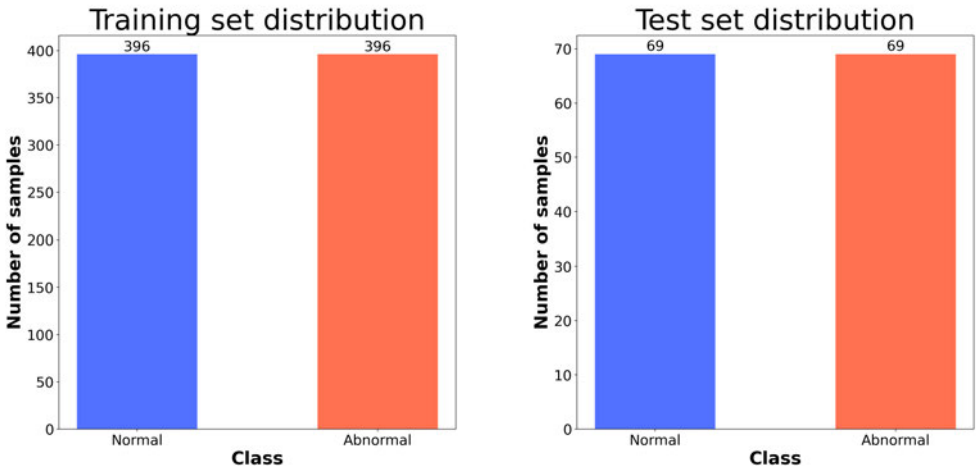
Figure 3.28: *Sonar* dataset accuracy score and learning rate results under OGLVQ model using balanced dataset.

3.1.5 SP and NSP datasets

Prototypes for each class: 12

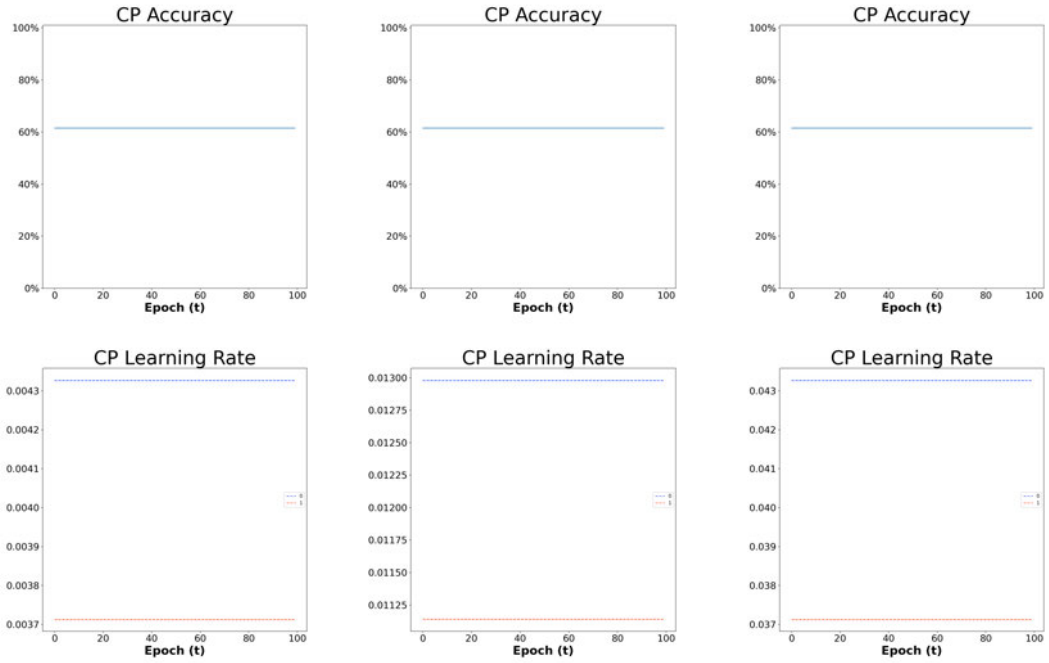
Both datasets we created from IFE data (*NSP* and *SP*) have similar results. We see flat accuracy score near 60% for any model with any $\epsilon(0)$ used to generate our results. We have also flat ϵ_i curves for any CGLVQ models. Our $\epsilon(0)$ might be small for the dataset, and using higher $\epsilon(0)$ than the ones we used might give us more answers if the dataset can fit with CGLVQ models. On the other hand, the OGLVQ model has a stable increase or decrease ϵ_i curves, but in the end, it does not show a change in the accuracy of the models. The change in learning rates does not always indicate that the model is learning. In this case, the change in learning rates is linear, and that might be happening because of the structure of the OGLVQ learning rate optimizer. Since we see flat graphs for both accuracy and learning rates, we can also say the datasets are too noisy to be trainable by our models. For now, we do not experiment with higher $\epsilon(0)$ and conclude that the dataset is not trainable with our models.

To not create a couple of pages long of the same accuracy results and similar, flat learning rate results, with the only difference being the starting point of the learning rates, we add results of one CGLVQ model, the CP model, to represent all other CGLVQ models. Even though OGLVQ accuracy scores are equal to other CGLVQ models, the shape of the learning rate graph is different, which is why we also include OGLVQ results on the view. We add one CGLVQ example, CP model, for *SP* dataset and one for *NSP* dataset. The actual results of other models can be found in Appendix B.



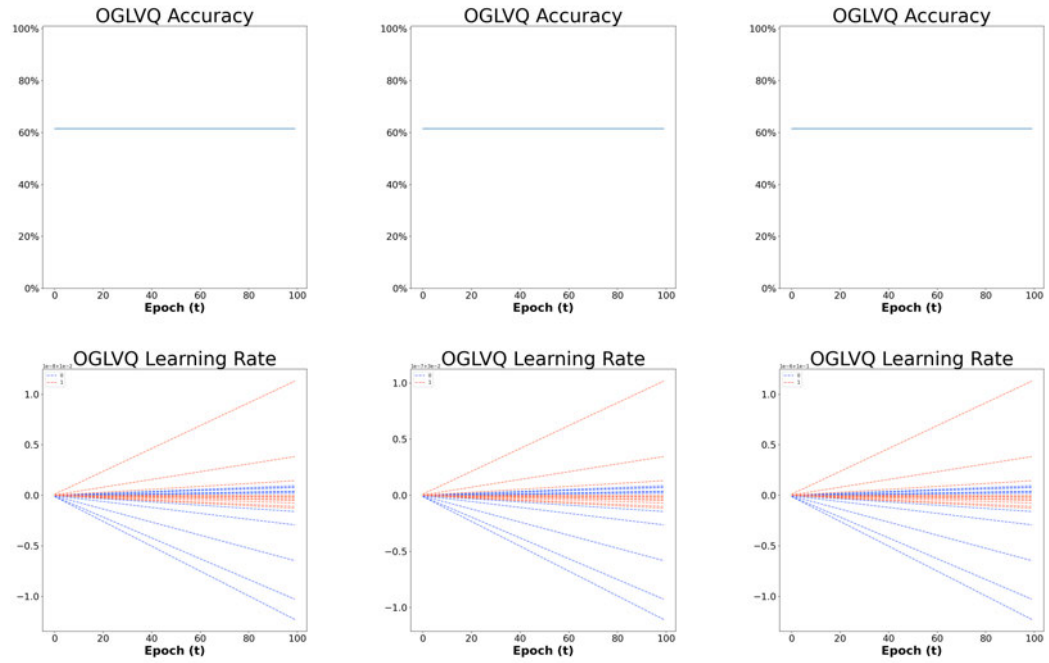
(a) Training set (b) Test set

Figure 3.29: *SP* and *NSP* balanced datasets sample distribution.



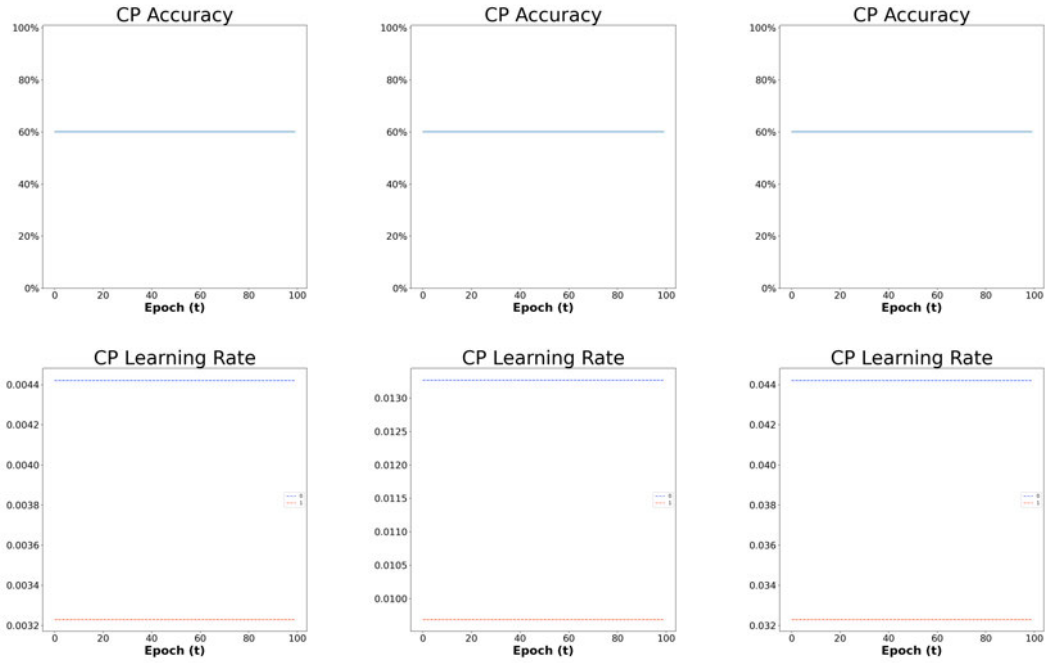
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.30: *SP* dataset accuracy score and learning rate results under CP model using balanced dataset.



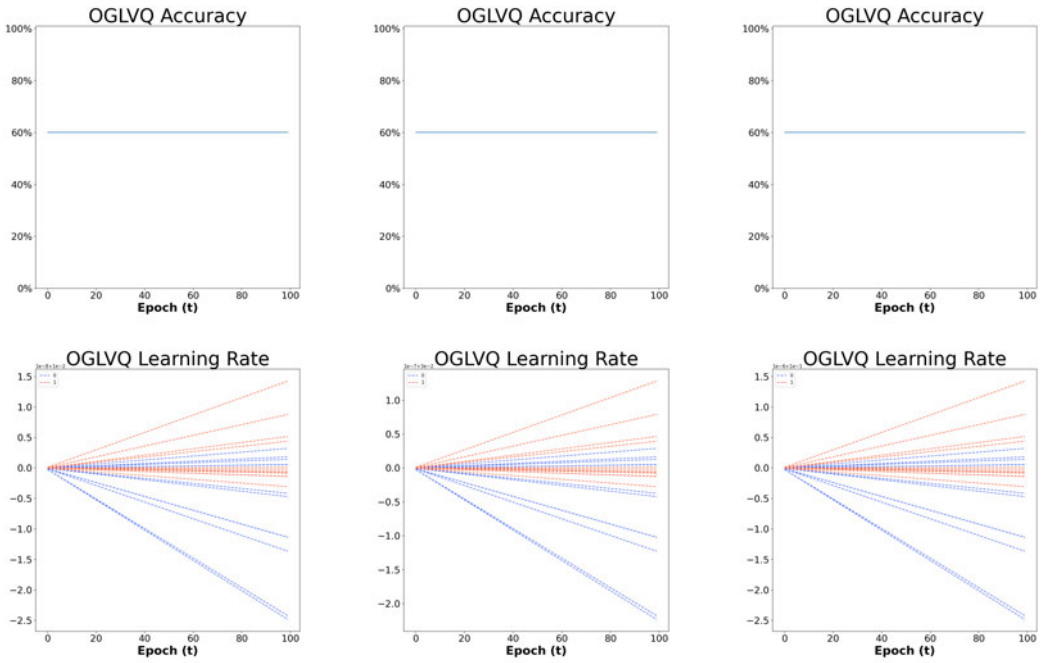
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.31: *SP* dataset accuracy score and learning rate results under OGLVQ model using balanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.32: *NSP* dataset accuracy score and learning rate results under CP model using balanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

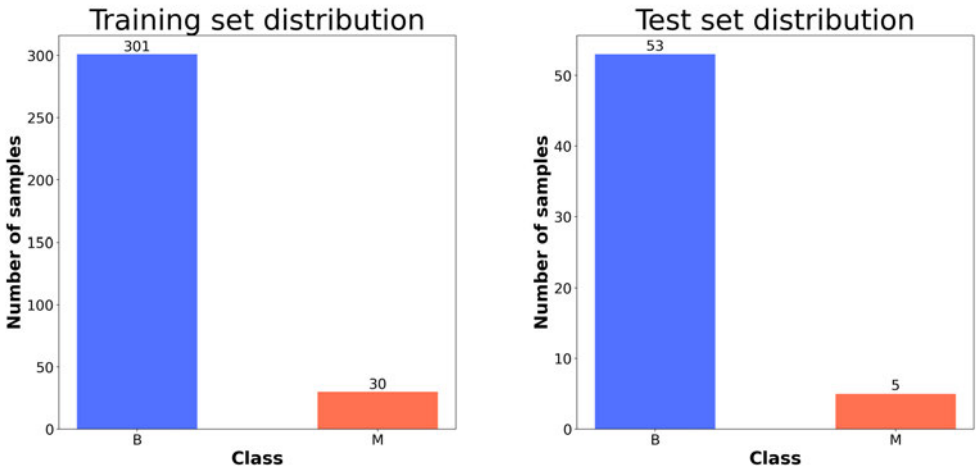
Figure 3.33: *NSP* dataset accuracy score and learning rate results under OGLVQ model using balanced dataset.

3.2 Experiment 2 (Imbalanced Dataset)

3.2.1 Breast Cancer Wisconsin dataset

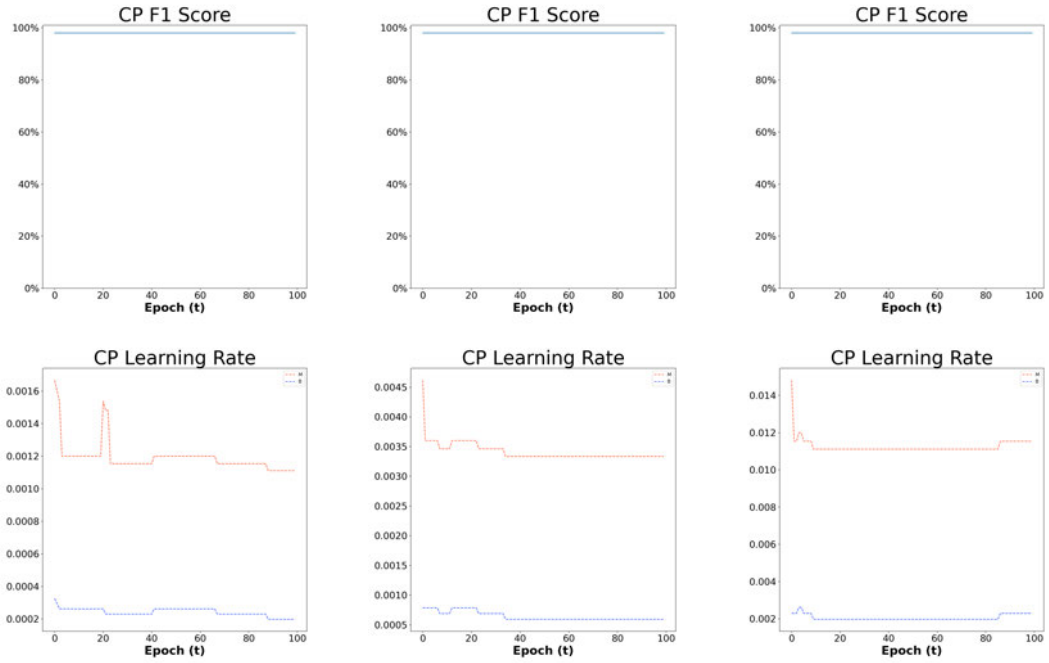
Prototypes for each class: 3

In experiment 2 of *Breast Cancer Wisconsin* dataset, we have a different story than the first experiment. Even if F1 scores change over time and have high F1 scores, learning rates does not move much for any CGLVQ model. The high F1 score for the models starts from the beginning of the models' training, which indicates that sample space does not have much noise and/or prototypes divides the sample space nicely. CGLVQ models have no learning regarding experiment 1, as we can understand from the flatness of the learning rate graphs. Additionally, the OGLVQ model's learning rate graphs look somewhat okay. Even if we see some of the learning rates decrease over time, one of the ϵ_i for class "M" does not decrease to 0 like other prototypes for the dataset. This ω_i, which shows bad learning, might be an outlier for the given dataset sample, and having this prototype might be coming because of using a smaller sample-sized class, "M."



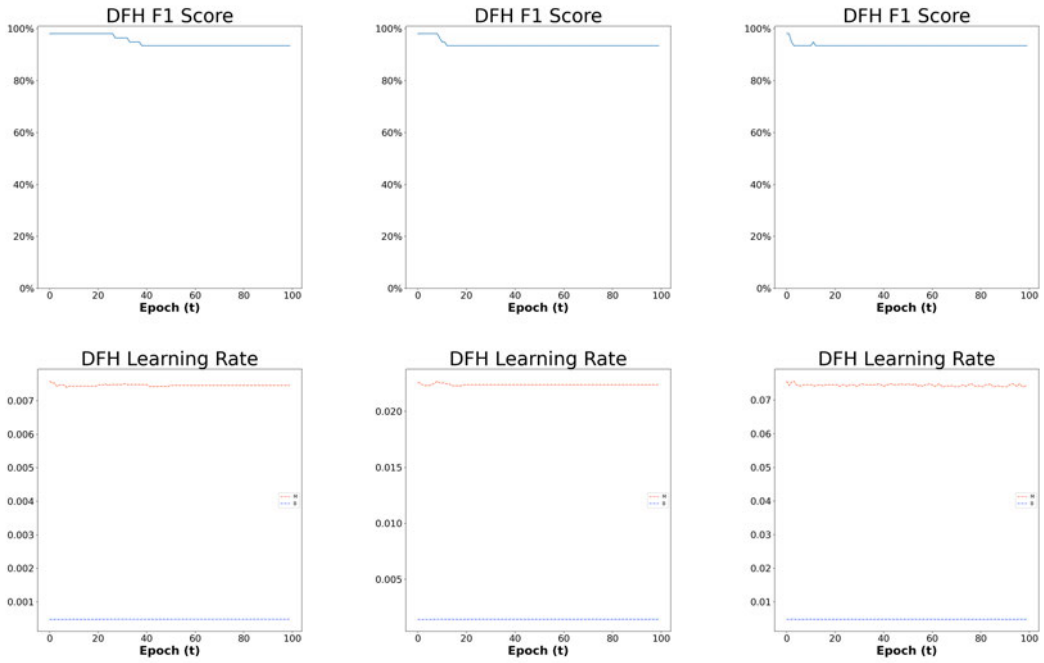
(a) Training set (b) Test set

Figure 3.34: *Breast Cancer Wisconsin* imbalanced dataset sample distribution.



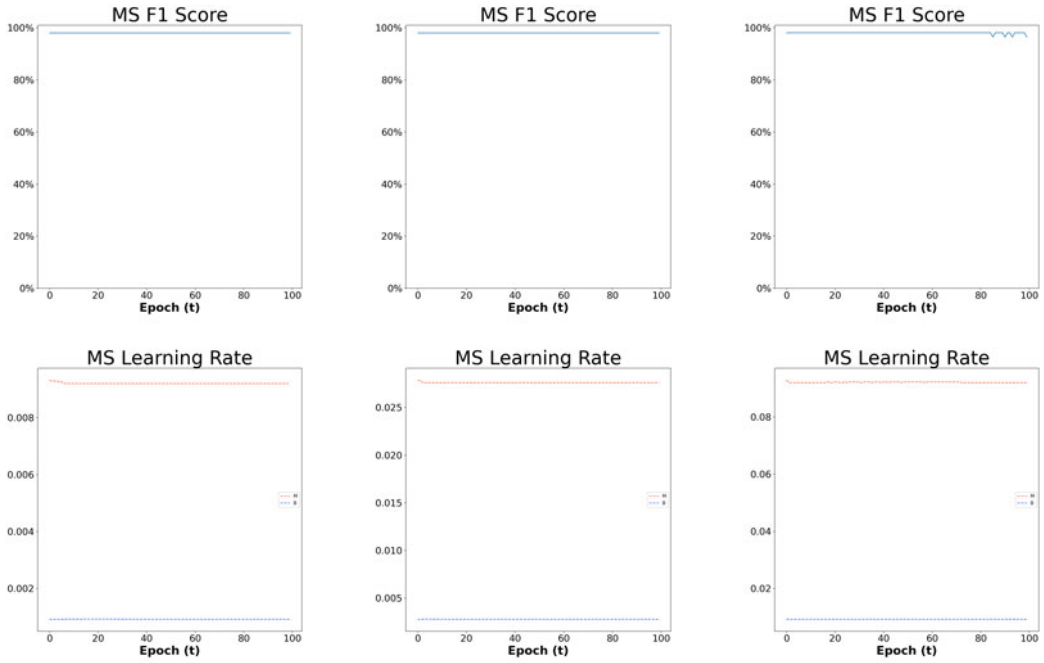
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.35: *Breast Cancer Wisconsin* dataset F1 score and learning rate results under CP model using imbalanced dataset.



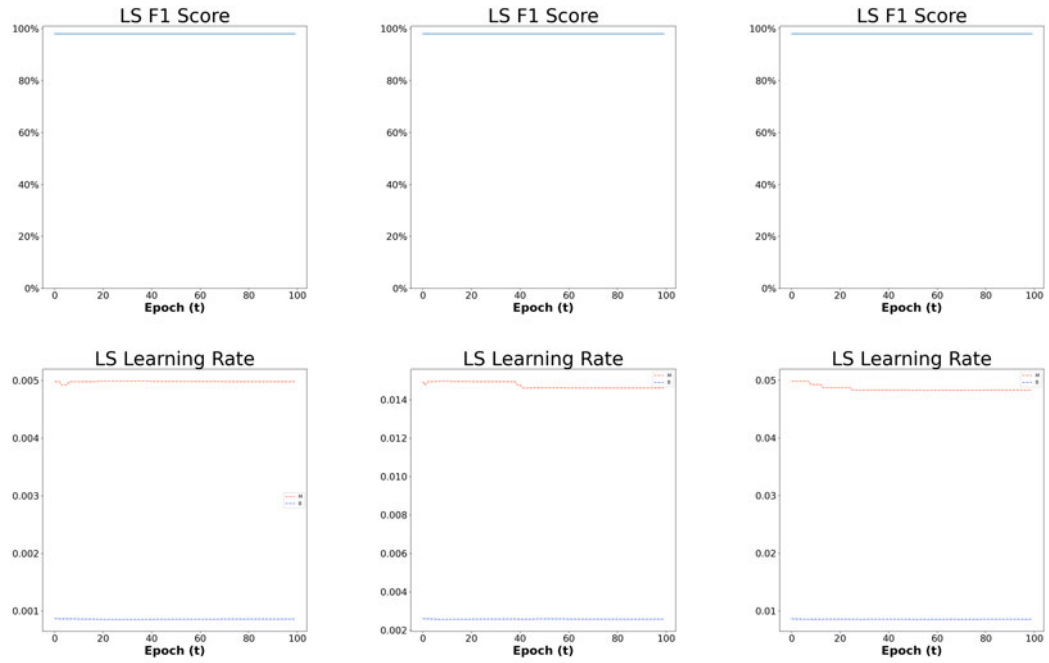
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.36: *Breast Cancer Wisconsin* dataset F1 score and learning rate results under DFH model using imbalanced dataset.



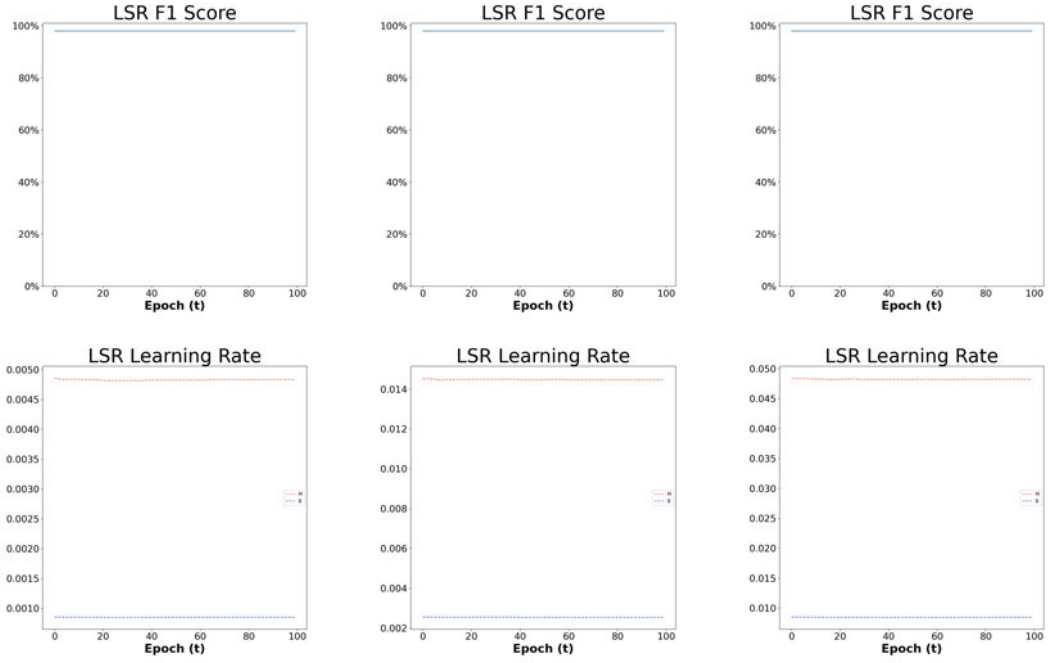
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.37: *Breast Cancer Wisconsin* dataset F1 score and learning rate results under MS model using imbalanced dataset.



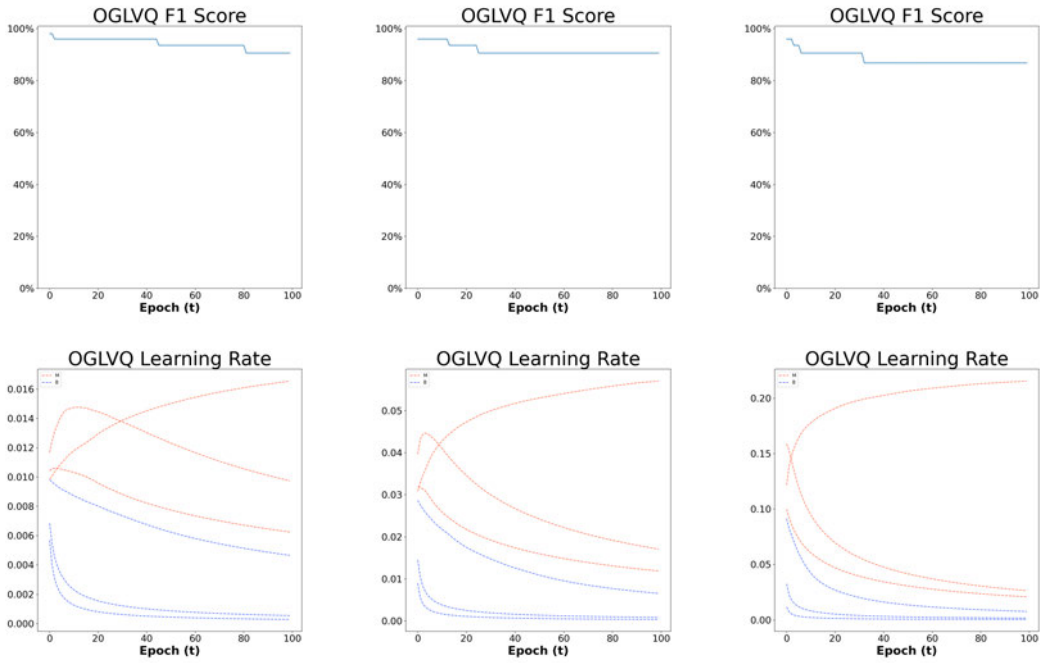
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.38: *Breast Cancer Wisconsin* dataset F1 score and learning rate results under LS model using imbalanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.39: *Breast Cancer Wisconsin* dataset F1 score and learning rate results under LSR model using imbalanced dataset.



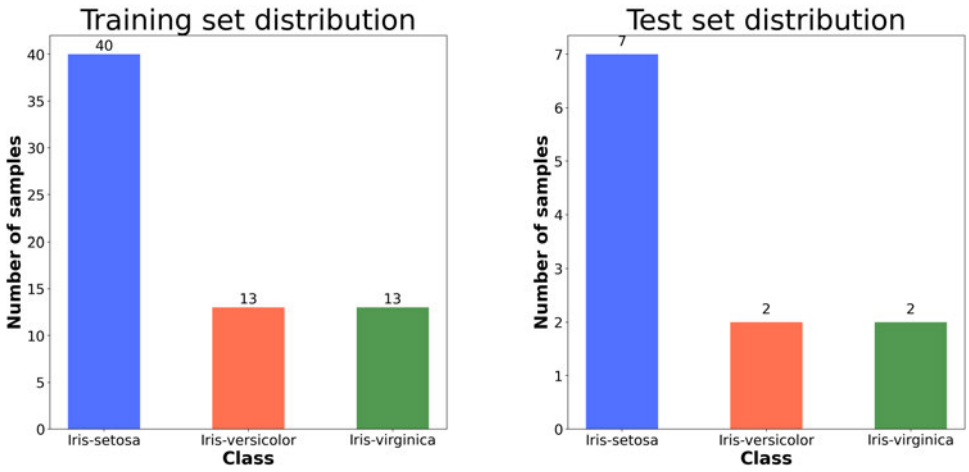
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.40: *Breast Cancer Wisconsin* dataset F1 score and learning rate results under OGLVQ model using imbalanced dataset.

3.2.2 Iris dataset

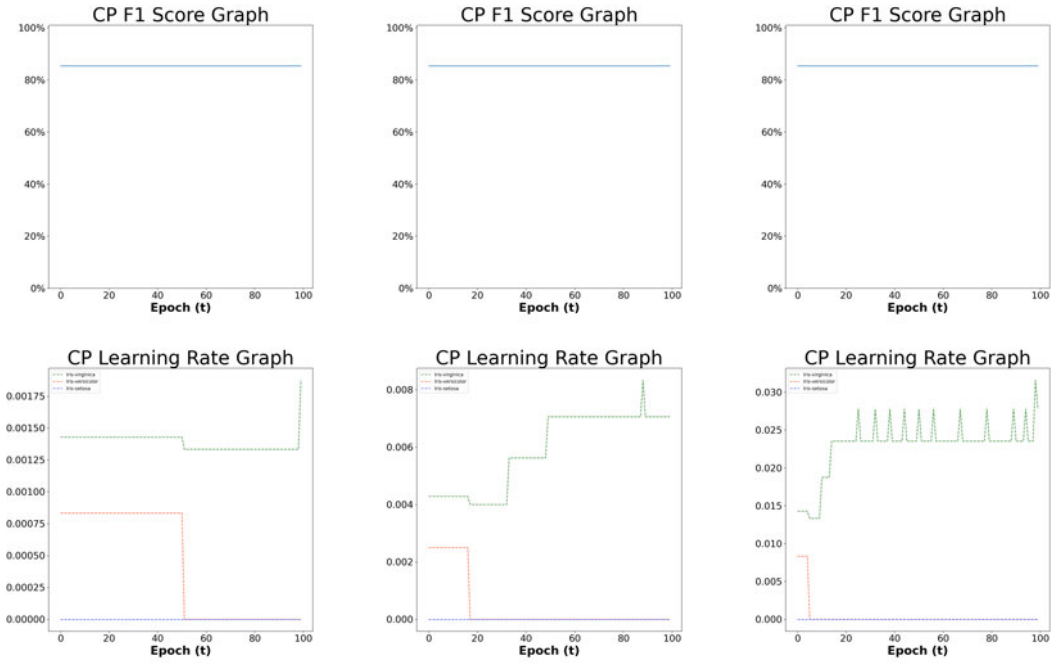
Prototypes for each class: 3

The same results of *Breast Cancer Wisconsin*'s experiment 2 can be said for most of the *Iris* datasets' experiment 2. The F1 score graphs are flat for any model, and CGLVQ models except the CP model all have near flat learning rates or learning rates are stuck in two values and changing periodically. Since the change in ϵ_i does not move much during the training, we say the model does not learn for these models. Even though the F1 scores for the CP model results are stable and high, learning rates are not good. For any result of the CP model on *Iris* dataset, we see an increase in "Iris-virginica's" ϵ, while a decrease in "Iris-versicolor's" ϵ during training. Both classes have a smaller sample size than the "Iris-setosa" class; however, one is learning while the other is not. We could say "Iris-virginica" might have the outlier prototypes for the given dataset sample if we did not see the results of OGLVQ. Like CGLVQ models, the OGVLQ model also has a flat F1 score. However, ϵ_i trend for OGLVQ is on decrease for $\epsilon(0) = 0.01$. Here, the "Iris-virginica" learning rates decreases with time. So, we cannot say the prototypes are outliers. On the other hand, one ω of the "Iris-versicolor" class's ϵ is higher than all other prototypes, which is the opposite of what we observed in the CP model with different learning rates. These results are not unexpected since we take small sample numbers for some classes in the dataset to create an imbalanced class samples environment. Because of the small dataset size, the models might be having a hard time adjusting the model to the sample space and hence learning rates.



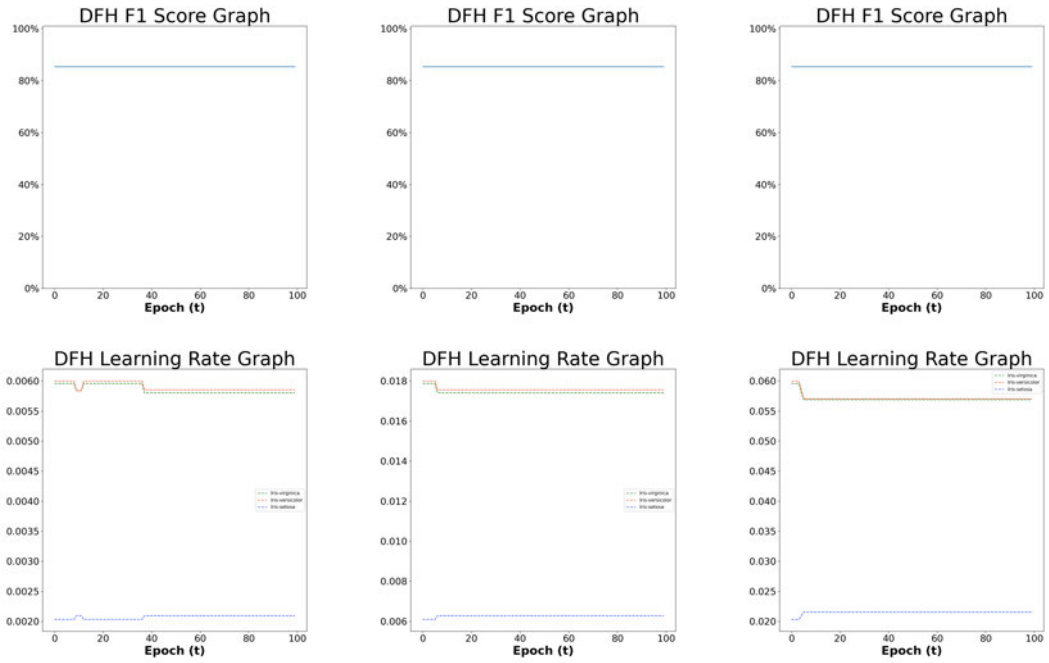
(a) Training set (b) Test set

Figure 3.41: *Iris* imbalanced dataset sample distribution.



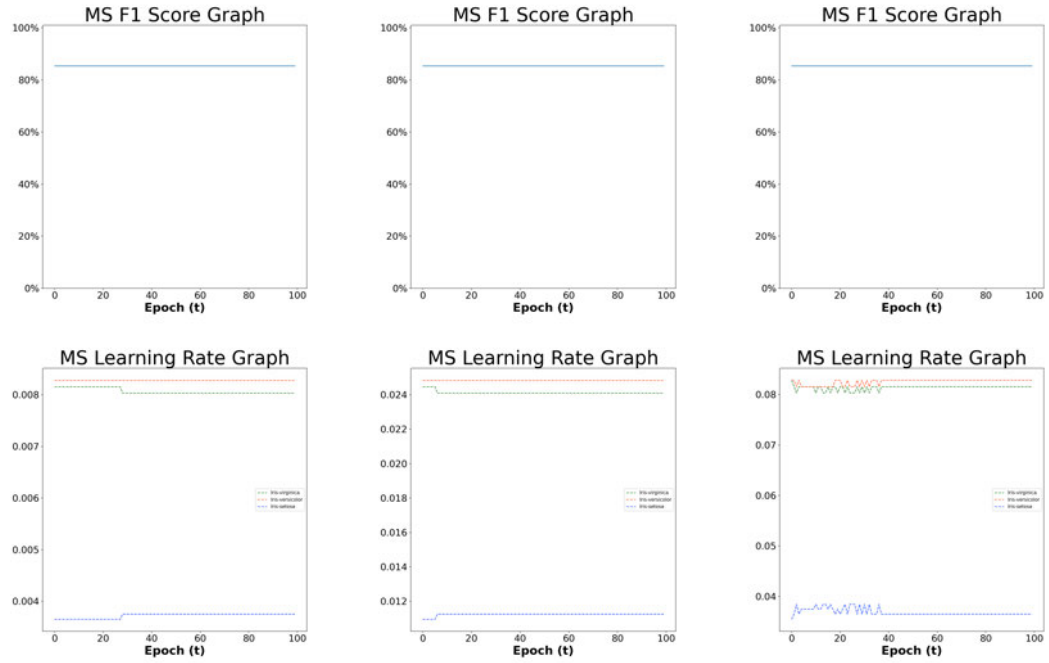
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.42: *Iris* dataset F1 score and learning rate results under CP model using imbalanced dataset.



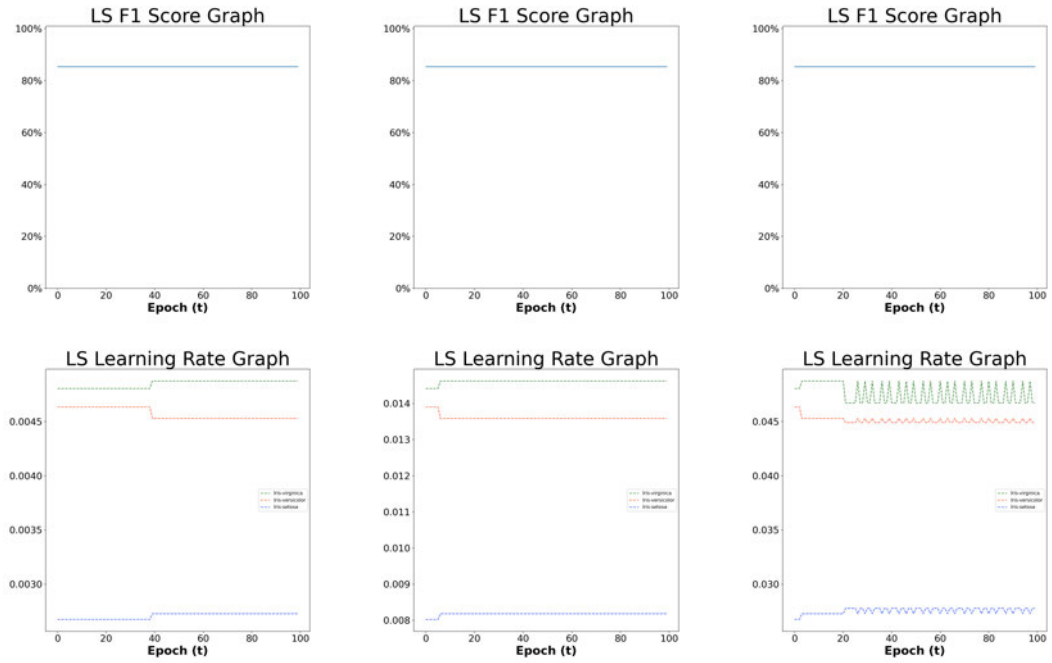
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.43: *Iris* dataset F1 score and learning rate results under DFH model using imbalanced dataset.



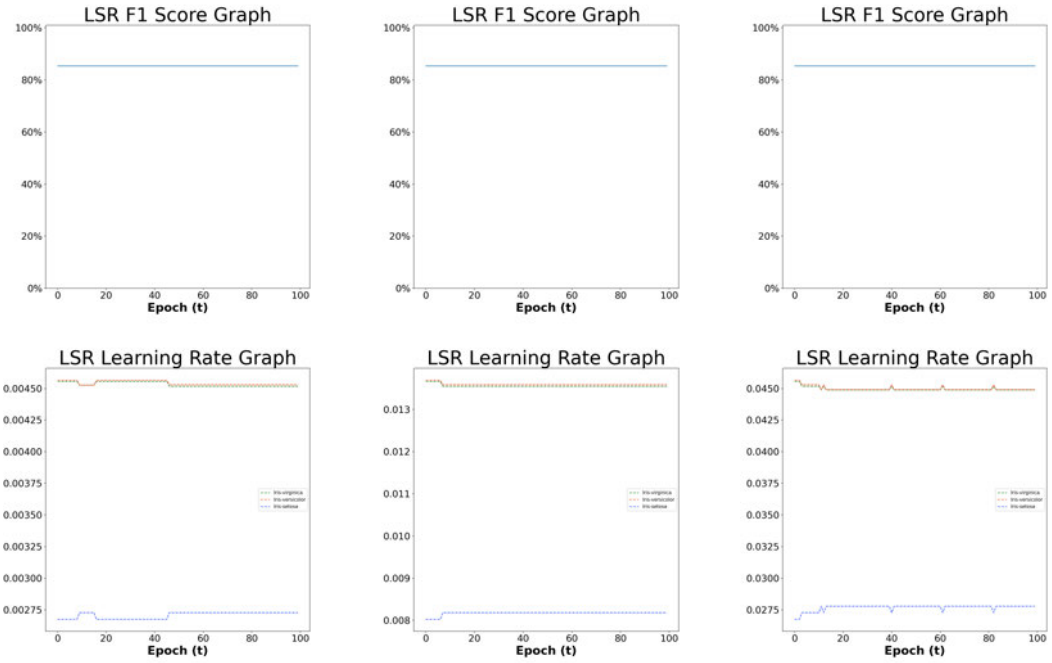
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.44: *Iris* dataset F1 score and learning rate results under MS model using imbalanced dataset.



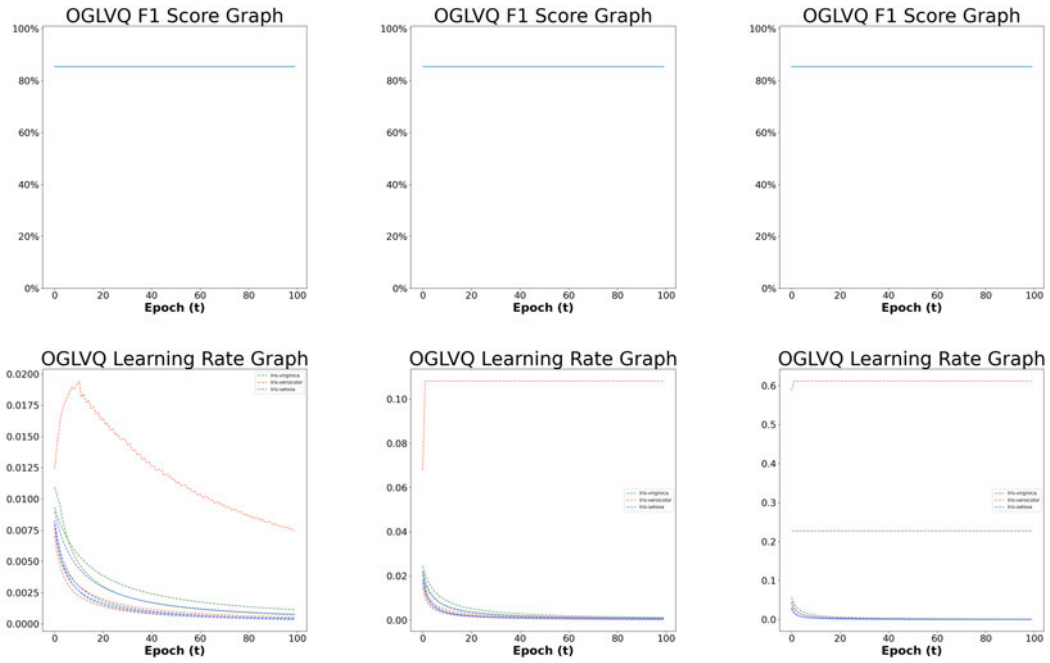
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.45: *Iris* dataset F1 score and learning rate results under LS model using imbalanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.46: *Iris* dataset F1 score and learning rate results under LSR model using imbalanced dataset.



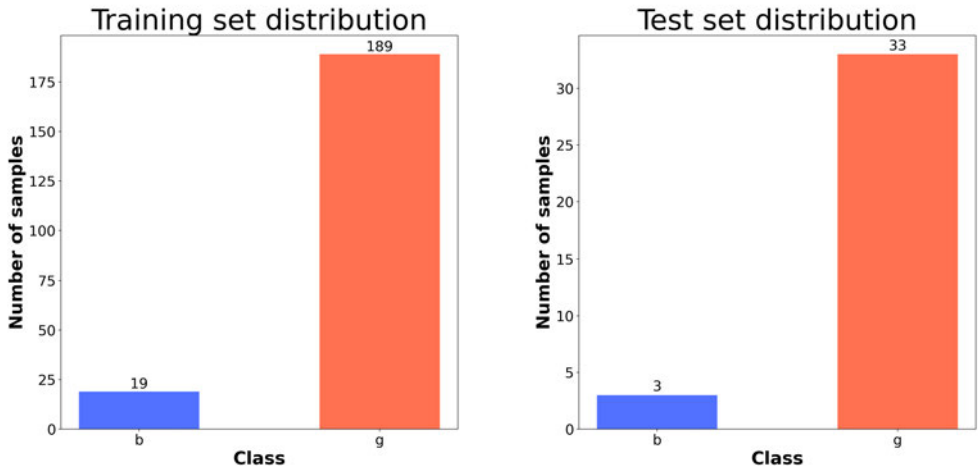
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.47: *Iris* dataset F1 score and learning rate results under OGLVQ model using imbalanced dataset.

3.2.3 Ionosphere dataset

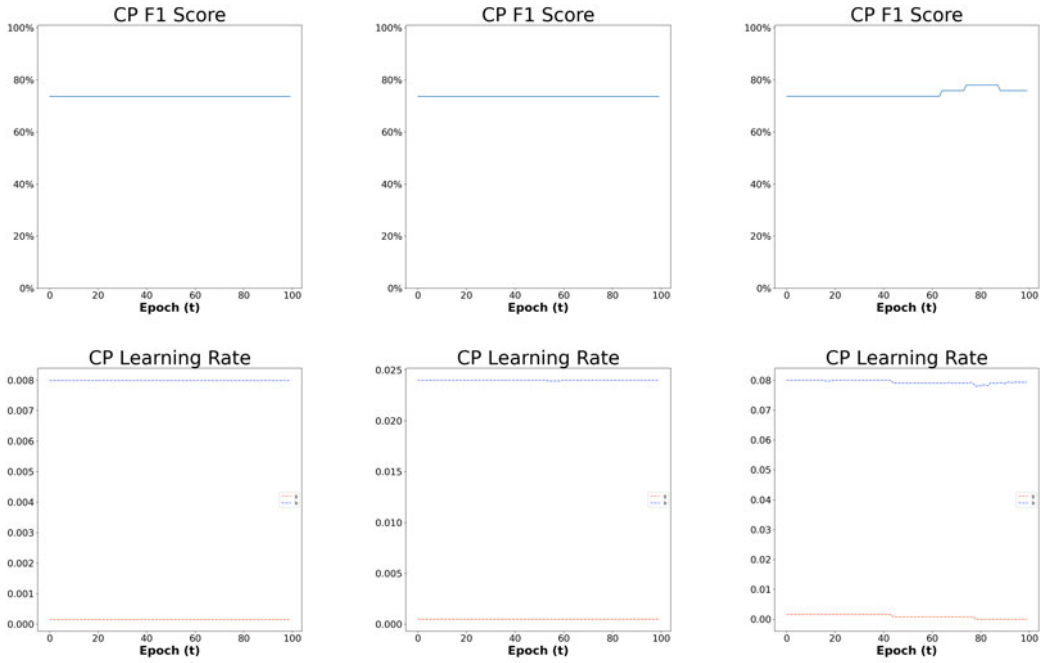
Prototypes for each class: 3

Ionosphere dataset shows some exciting results on experiment 2. In the experiment, we see great results of F1 score from MS, LS, and LSR with $\epsilon(0) = 0.1$, even better than OGLVQ results. These F1 scores also reflect these models' ϵ_i curves. LS model with $\epsilon(0) = 0.1$ has a nicely decreasing ϵ_b for class "Bad" (b) while the other class, "Good" (g), does not change much (Figure 3.52c). We can still see a slight change in the learning rates with smaller $\epsilon(0)$ of the LS model. These models still learn and increase the F1 score during training, but $\epsilon(0)$ is probably small to show significant steps, and to gain better results, we can increase the training time t. For MS and LSR models with $\epsilon(0) = 0.1$ (Figures 3.51c and 3.53c), we see not much of a change in learning rates, but we still see a change of learning rates with increasing F1 score. Higher $\epsilon(0)$ can show the performance of the models faster. When we look at OGLVQ, the learning rate graphs for all the runs with different $\epsilon(0)$ look bad. We see a similar situation in the experiment 1 results of *Ionosphere* dataset, but this time, all the learning rates of the prototype class "Bad" are on rise for OGLVQ model. It looks like fitting "Bad" in the sample space is hard for the OGLVQ model. It is good to mention that, in experiment 2, we used 19 "Bad" samples in the training set. So, with this small sample space, it is impressive to see CGLVQ models, mostly LS, having great results.



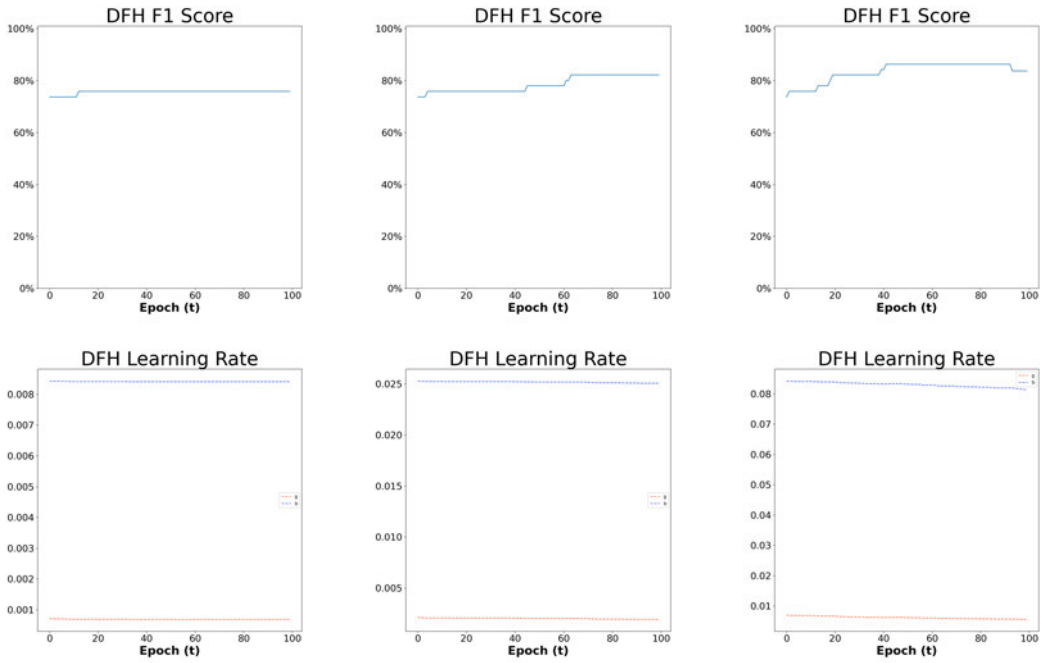
(a) Training set (b) Test set

Figure 3.48: *Ionosphere* imbalanced dataset sample distribution.



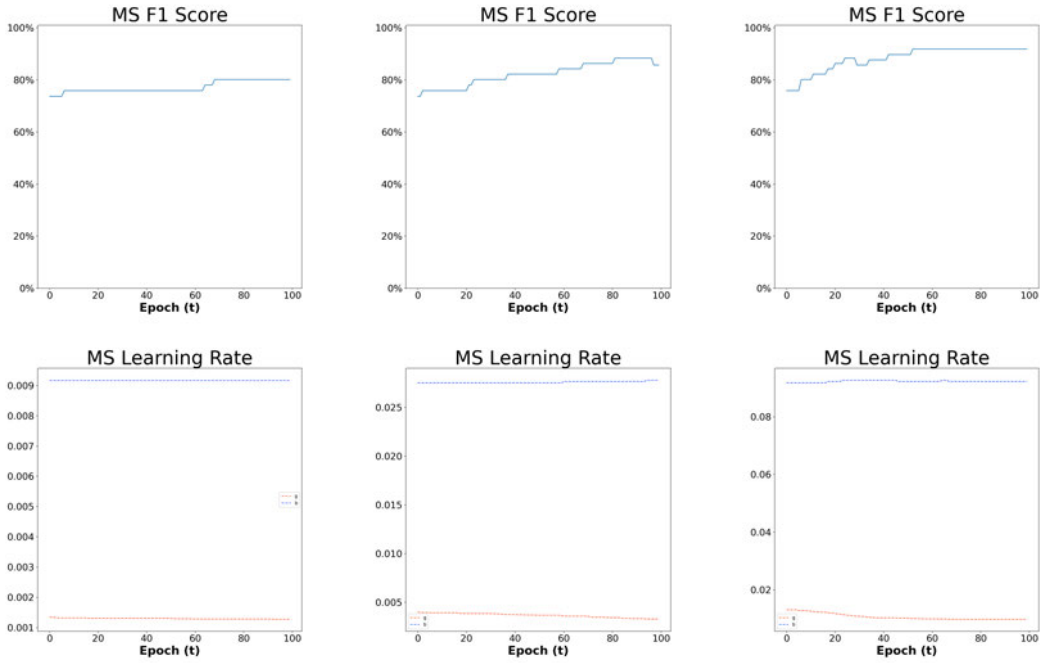
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.49: *Ionosphere* dataset F1 score and learning rate results under CP model using imbalanced dataset.



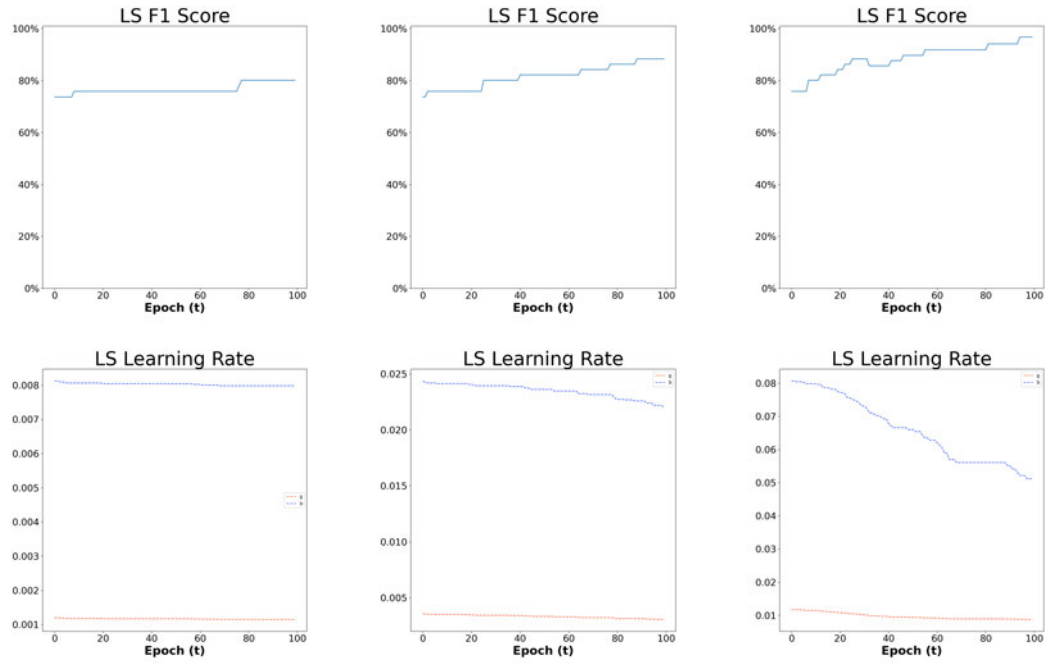
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.50: *Ionosphere* dataset F1 score and learning rate results under DFH model using imbalanced dataset.



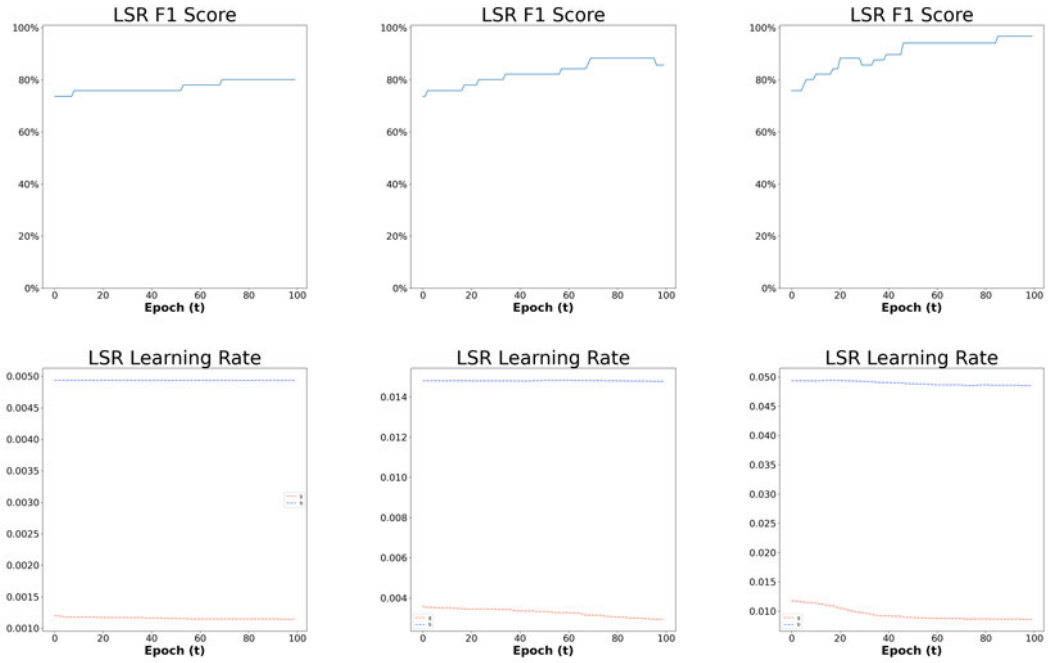
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.51: *Ionosphere* dataset F1 score and learning rate results under MS model using imbalanced dataset.



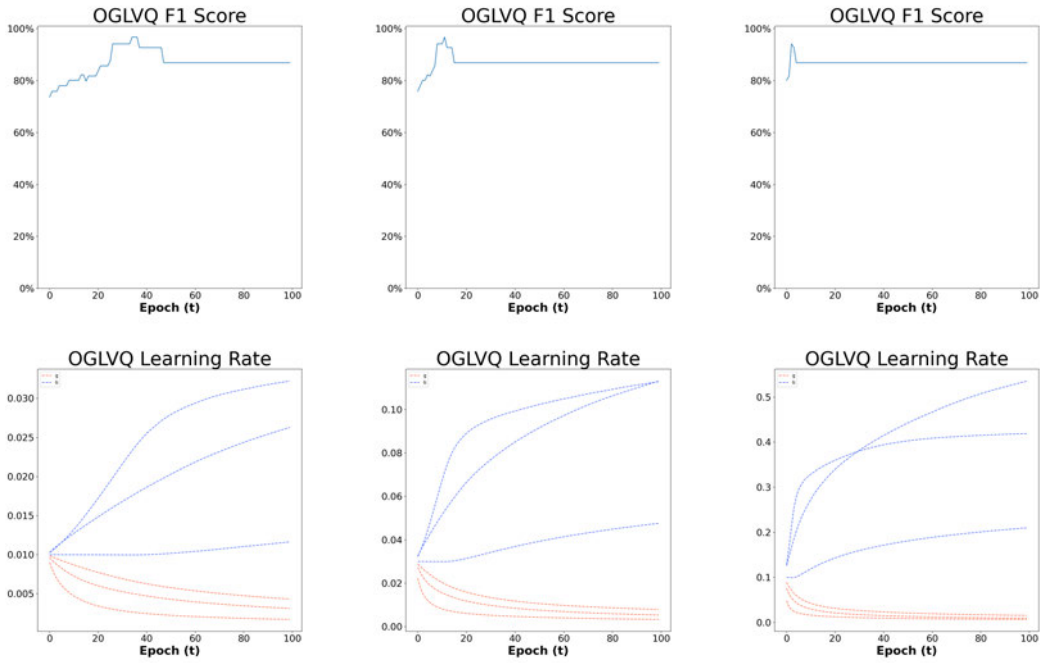
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.52: *Ionosphere* dataset F1 score and learning rate results under LS model using imbalanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.53: *Ionosphere* dataset F1 score and learning rate results under LSR model using imbalanced dataset.



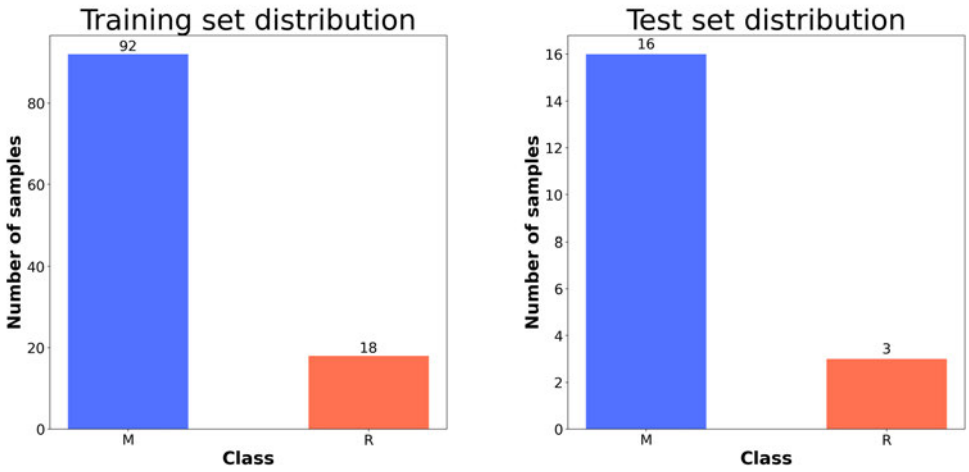
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.54: *Ionosphere* dataset F1 score and learning rate results under OGLVQ model using imbalanced dataset.

3.2.4 Sonar dataset

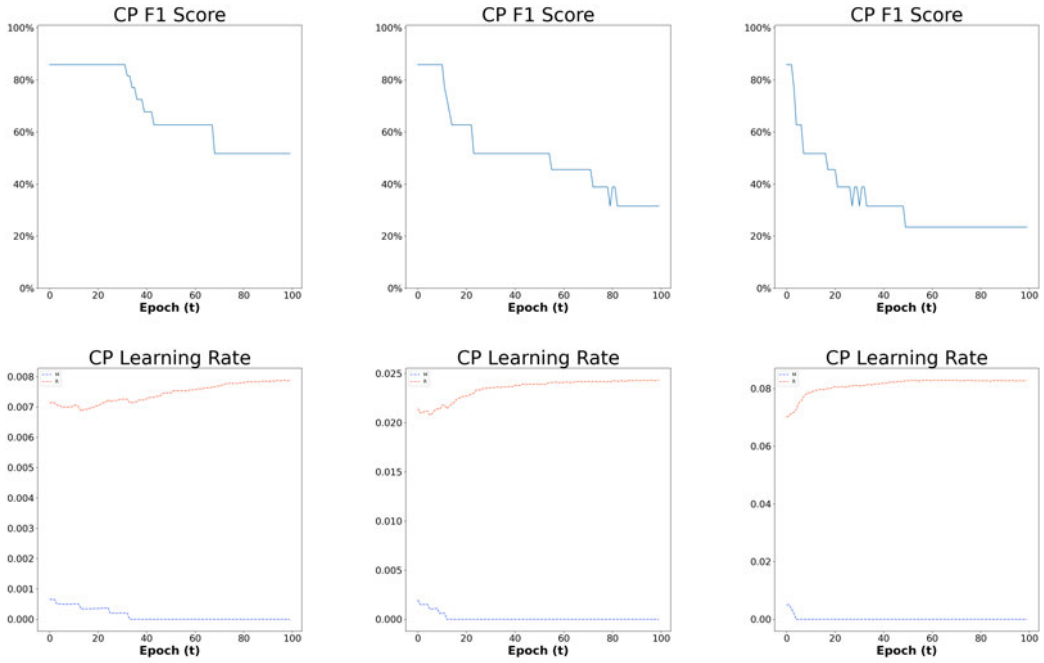
Prototypes for each class: 3

CP model on *Sonar* dataset has decreasing F1 scores with increasing ϵ_R on the "Rock" class. For DFH and MS tests with any $\epsilon(0)$ values, we see a slight decrease but not much change in ϵ_i values during the training; however, LS and LSR test with any $\epsilon(0)$ have increasing F1 scores and decreasing ϵ_i values. ϵ decrease on LS is mainly in "Rock" class; on LSR, the ϵ decrease is observed in the "Mine" class. The better ϵ_R for LS, which has the smaller training sample in our imbalanced sample space, results in a better F1 score for the model. If we check OGLVQ learning rate graphs in Figure 3.61, we also see that the "Rocks" class has difficulty decreasing its learning rates. Ultimately, we observe that the DFH, MS, LS, and LSR models have better results than the OGLVQ model with a smaller and imbalanced dataset.



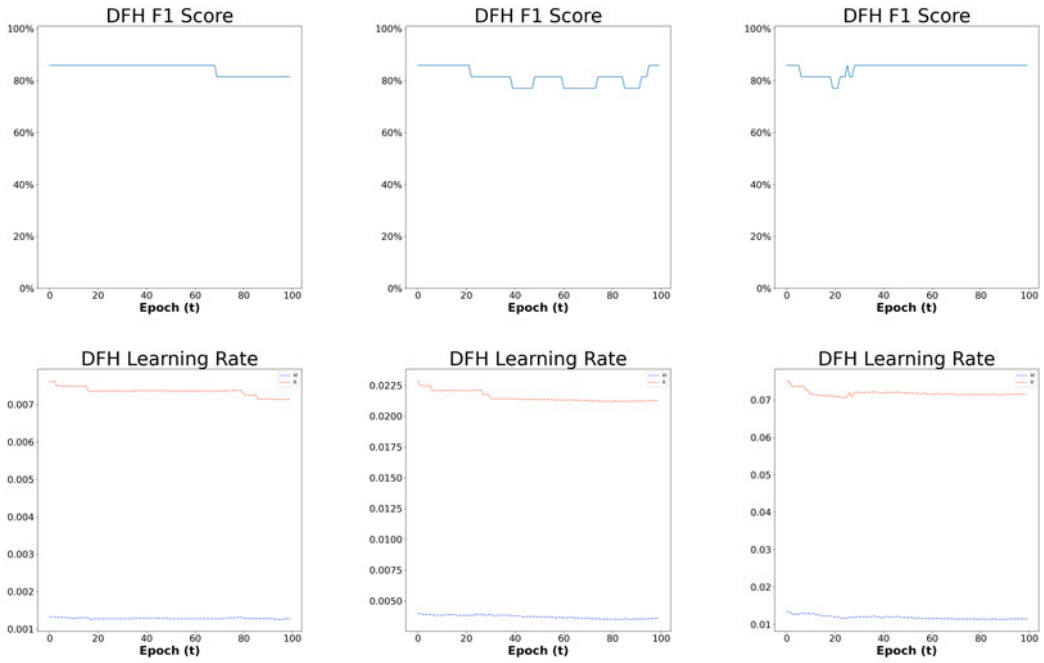
(a) Training set (b) Test set

Figure 3.55: *Sonar* imbalanced dataset sample distribution.



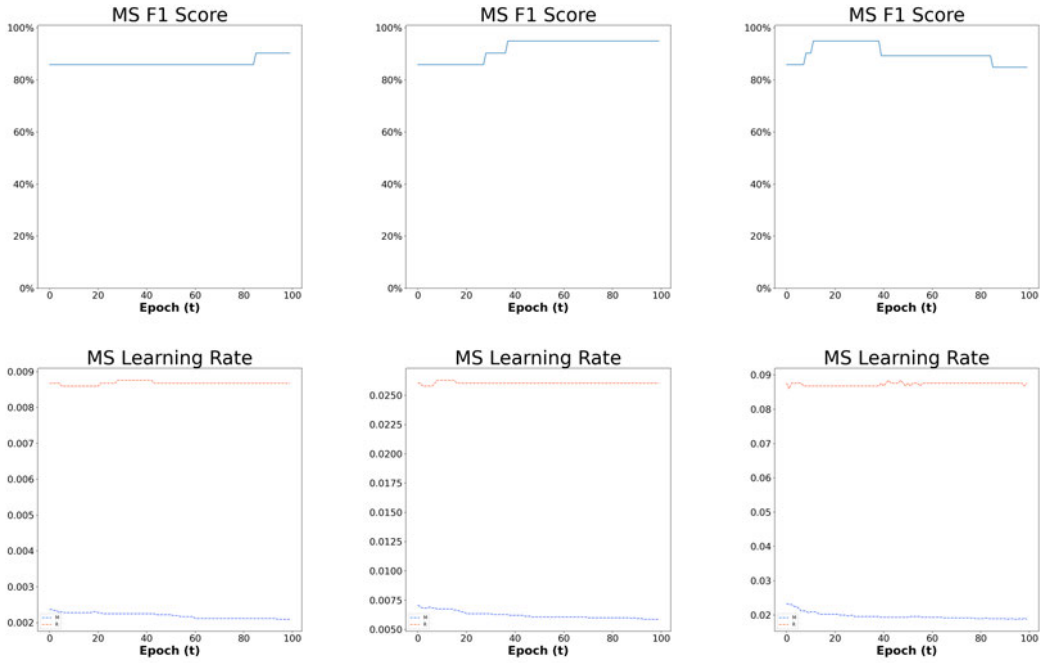
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.56: *Sonar* dataset F1 score and learning rate results under CP model using imbalanced dataset.



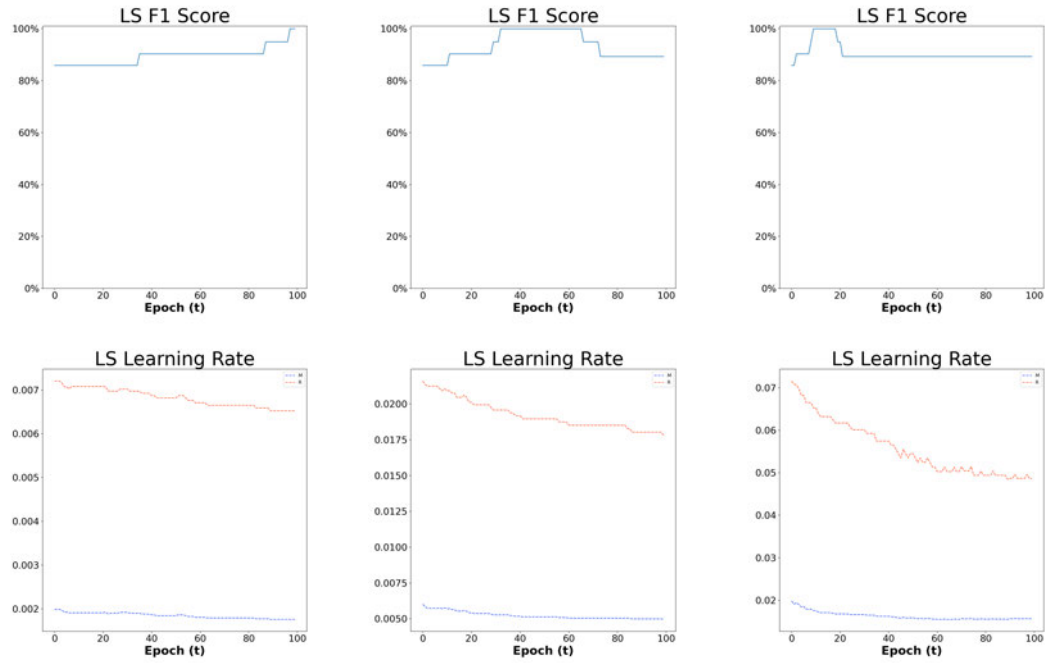
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.57: *Sonar* dataset F1 score and learning rate results under DFH model using imbalanced dataset.



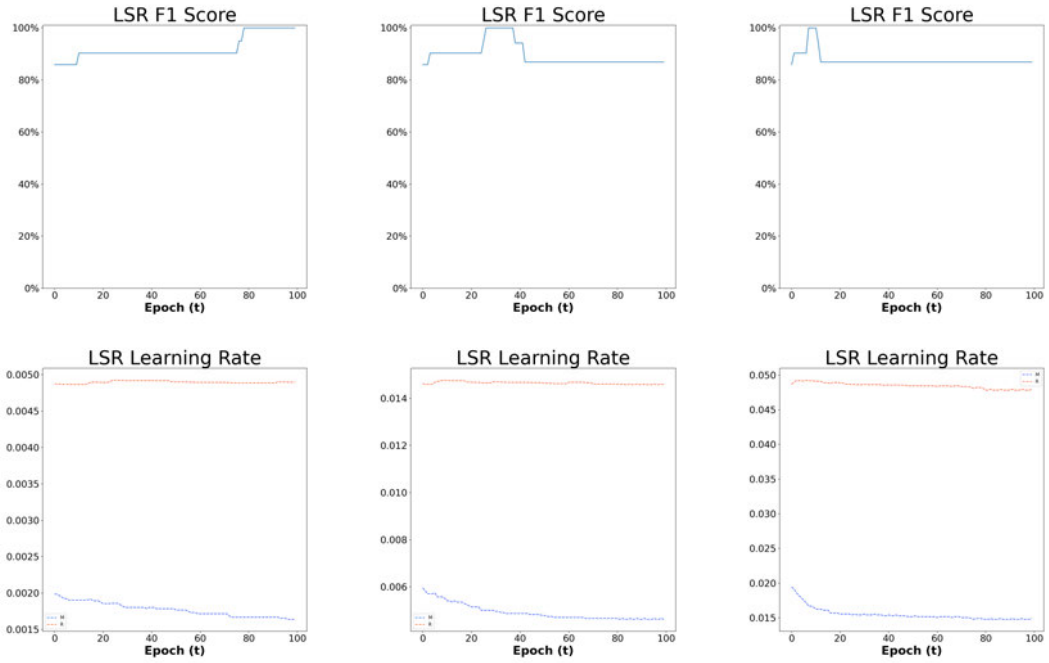
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.58: *Sonar* dataset F1 score and learning rate results under MS model using imbalanced dataset.



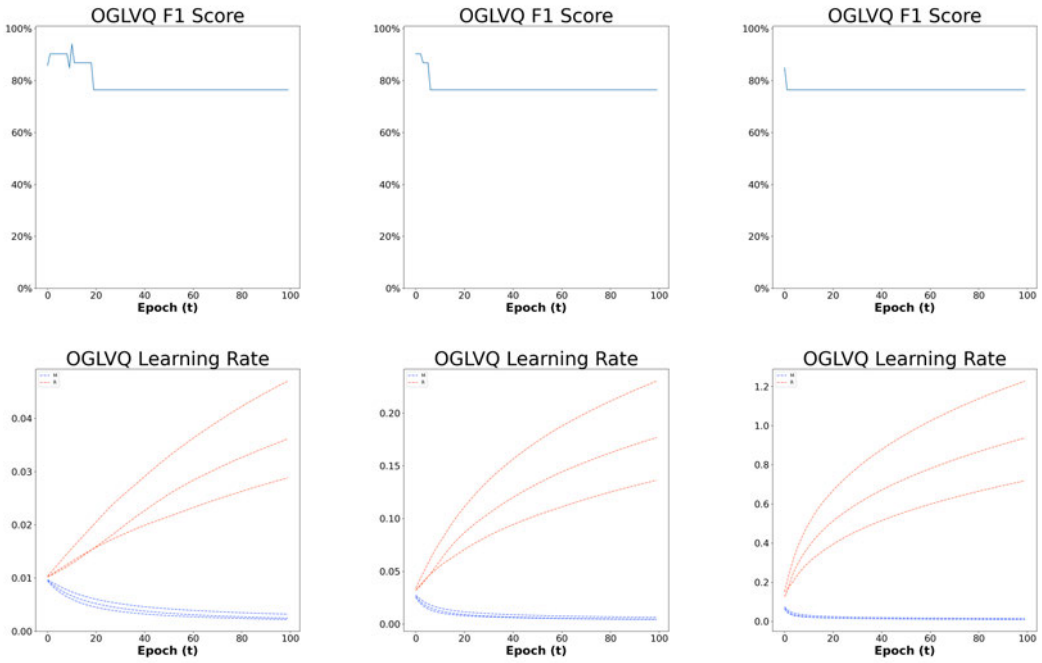
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.59: *Sonar* dataset F1 score and learning rate results under LS model using imbalanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.60: *Sonar* dataset F1 score and learning rate results under LSR model using imbalanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

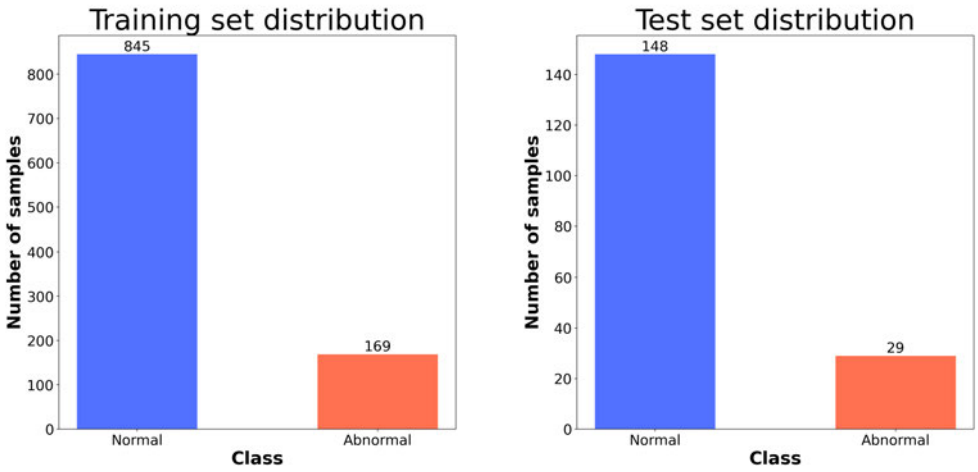
Figure 3.61: *Sonar* dataset F1 score and learning rate results under OGLVQ model using imbalanced dataset.

3.2.5 SP and NSP datasets

Prototypes for each class: 12

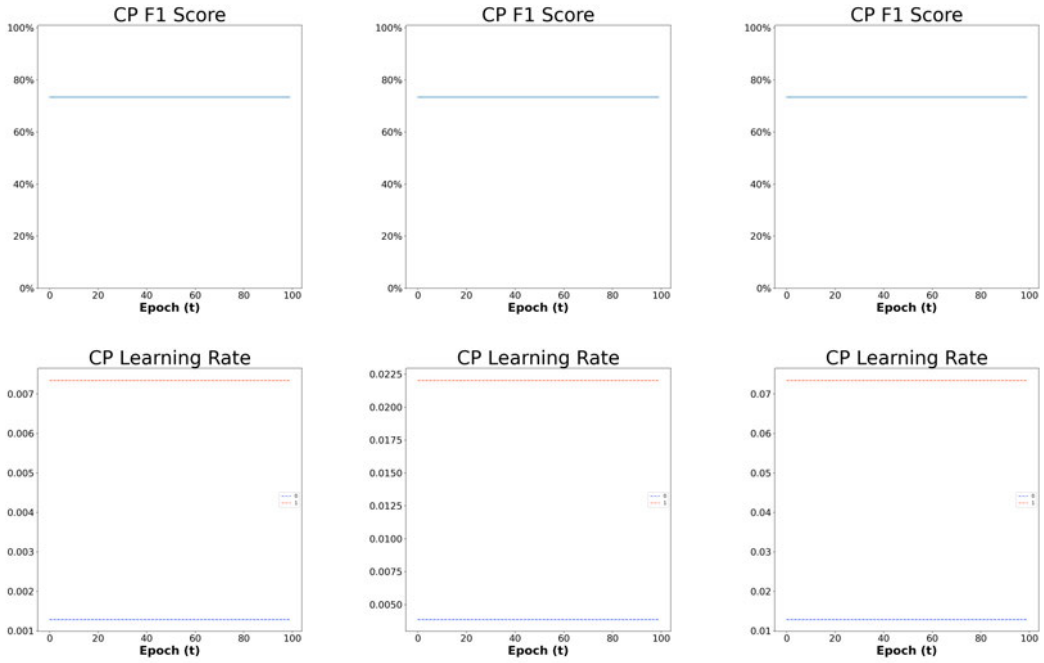
Similar to experiment 1 of *NSP* and *SP*, experiment 2 reinforces that these datasets are not possible to be trainable with the models we used, even with imbalanced datasets of *NSP* and *SP*. We can see that the models do not train by looking at the F1 scores and learning rates of the models for both datasets. Higher $\epsilon(0)$ might give us more answers if the dataset is trainable or not. OGLVQ results of experiment 2 are similar to the results of experiment 1.

Similar to experiment 1, we only include the CP and OGLVQ models for both the *SP* and *NSP* datasets, and the actual results can be found in Appendix B.



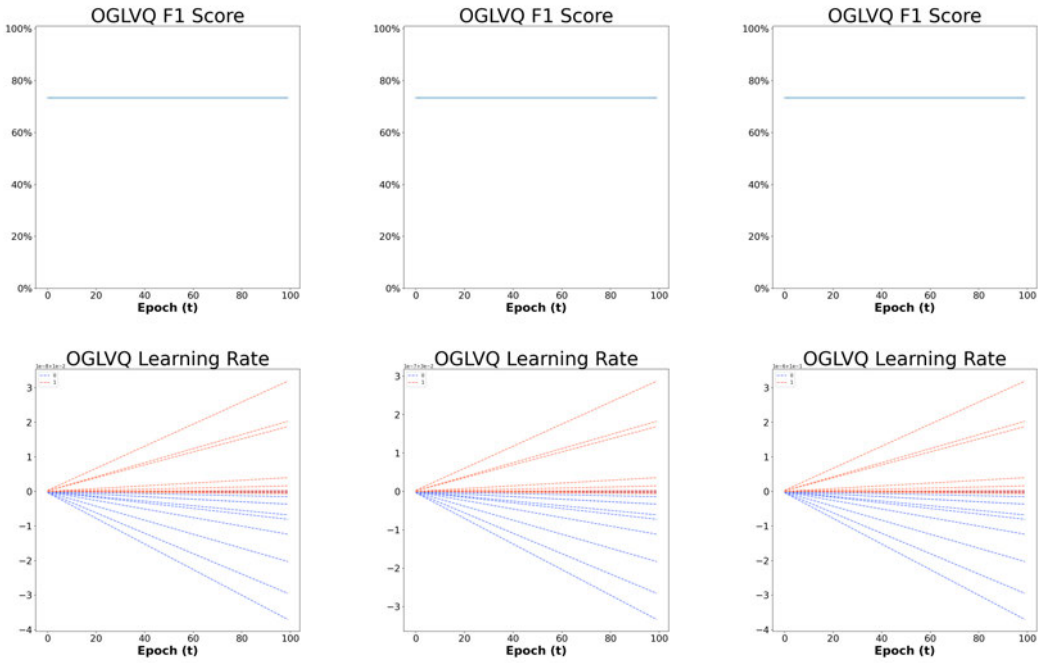
(a) Training set (b) Test set

Figure 3.62: *SP* and *NSP* imbalanced datasets sample distribution.



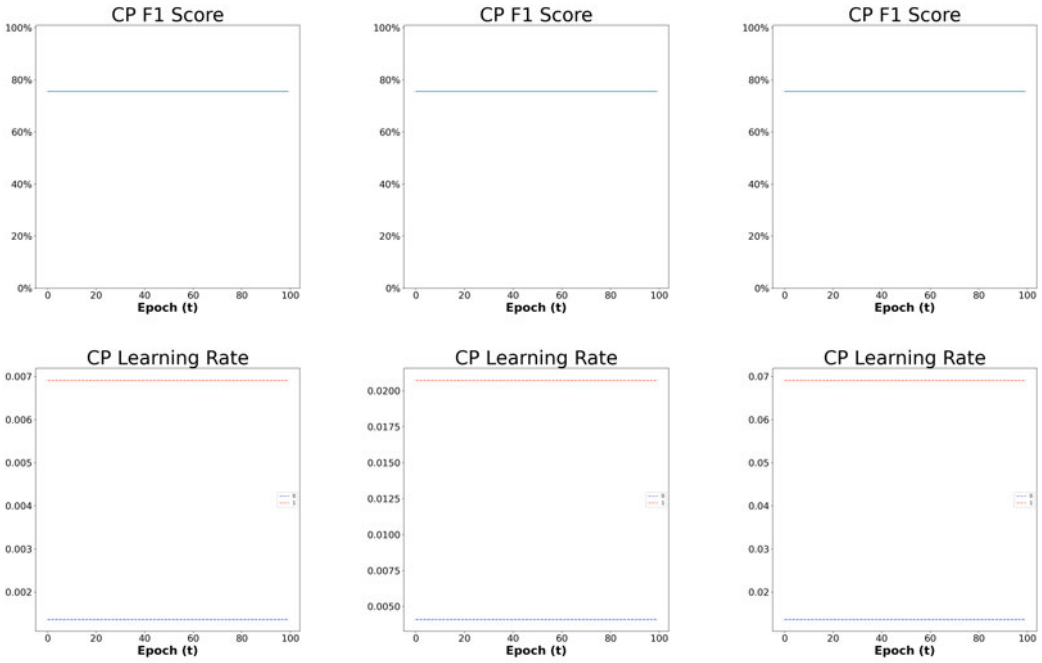
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.63: *SP* dataset F1 score and learning rate results under CP model using imbalanced dataset.



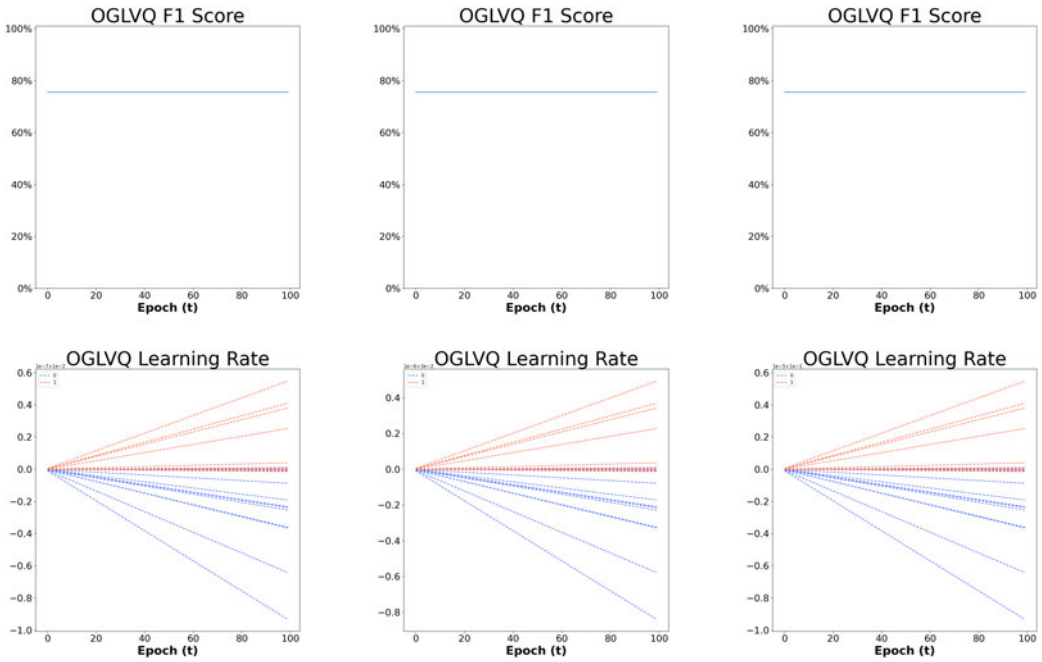
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.64: *SP* dataset F1 score and learning rate results under OGLVQ model using imbalanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.65: *NSP* dataset F1 score and learning rate results under CP model using imbalanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure 3.66: *NSP* dataset F1 score and learning rate results under OGLVQ model using imbalanced dataset.

Chapter 4

Discussion

With our experiments, we investigated the CGLVQ models concerning their learning rate changes and revealed the power of the models. The learning capability of humankind is higher than that of other species in the world, at least as we know it now. Also, learning might not be logical like machines. Tracing the human biases on machine learning models helps us to copy human-like learning, which allows us to create more human-like machines.

Unfortunately, our custom datasets, *NSP* and *SP*, were untrainable with our setup of the models; hence, we could not get much information from these datasets. On the other hand, with other datasets we used, which are open-source datasets, we found CGLVQ is adapting the sample spaces of these datasets and learning through the training. We had great results with our Experiment 1, which has datasets with balanced class samples for the training. In Experiment 1, CGLVQ models, except the CP model, are as competitive as the OGLVQ model. Even with the *Ionosphere* dataset, CGLVQ models outperform OGLVQ model. This experiment found that MS and LSR models perform better with the *Ionosphere* dataset. In contrast, LSR has the only good performance with the *Iris* dataset compared to other CGLVQ models.

Experiment 2 showed us another power of the CGLVQ models. We used imbalanced datasets on the models by reducing the sample size for one of the classes in the datasets. With an imbalanced dataset experiment, we found that there are 3 CGLVQ models that shine: MS, LS, and LSR. These models outperformed OGLVQ with imbalanced and small datasets, *Ionosphere* and *Sonar* datasets, and showed better learning rate graphs in their results. The LS model showed the biggest performances among other CGLVQ models concerning the F1 score.

One thing to note is that CGLVQ uses class-based learning rates rather than prototype-based ones. This behavior might seem like a bad idea in theory since the sample space might not be divided linearly, and updating the learning rates of the prototypes in the same class might cause problems. However, our experiments show great results, even better than OGLVQ. Still, because of the class-based learning rate adaptation, these CGLVQ models might perform worse than other LVQ models in a noisier sample space.

In summary, we found in this paper that the cognitive science learning rate optimizer approaches have great results, especially MS, LS, and LSR. These results align with the findings of Takahashi et al. (2010) [3]. However, the findings of Takahashi et al. (2010) [3] show good performance with CP and DFH learning rate methods according to Table 2.4. Still, our results found that these optimizers do not perform much compared to other learning rate methods. So, we can conclude that we can continue more research to

improve MS, LS, and LSR. There are many more cognitive science learning rate methods created from cognitive biases to optimize the learning rates that we can use on GLVQ models we did not include. One can expand the research, including other models we did not include and new models based on cognitive biases.

Since we noted that we used and experimented on datasets with simple (not noisy) sample space, it would be great to have further experiments with (reasonably) nosier datasets. In that way, we can see how the selected CGLVQ models generally perform with noisier datasets. Solving less noisy datasets is easy for most models. However, we see the models' usefulness when the dataset is harder to solve since it is closer to real-world problems.

Finally, we can say that CGLVQ models, especially MS, LS, and LSR, have better results than OGLVQ model. Since CGLVQ learning rate optimizers come from cognitive science, the models provide more human-like learning for machine learning, which helps us to create more human-like models while also allowing us to understand the model's reasoning.

Appendix A

GLVQ Codes

The project's Python codes can be found in the following link also and can be used freely: `https://github.com/mertsaru/Cognitive-GLVQ`

Or you can follow the codes for OGVLQ, CGLVQ and optimizers for CGLVQ here:

OGLVQ

```
"""
The model is Optimized GLVQ (OGLVQ) model.
Turning any LVQ model to optimized version of it introduced by Kohonen (1995, pp. 175-189) in "Self-Orgazing Maps" (DOI:https://doi.org/10.1007/978-3-642-97610-0), please refer to the paper when needed.
Optimization effects the model's learning rate update.

the model includes two performance measures:

- Accuracy
- F-Score (weighed average)

To use the model please import the file and use the class CGLVQ. Then use class method train with the following parameters:
    num_epochs: train time
    training_set: adjust the training set as: list[tuple[np.array, np.array]],
    test_set: adjust the test set as: list[tuple[np.array, np.array]],
    validation_set: if you want to use validation set adjust the validation set as: list[tuple[np.array, np.array]] = None,
    f_score_beta: beta value of the F score, default = 1 any float value can be used,
    sample_number: Number of training samples each class uses. It is needed to calculate the weighted F scores

One can use the following methods to see the results:
    lr_graph: shows the learning rate graph for each prototype
    acc_graph: shows the accuracy graph
    f1_graph: shows the f1 score graph

    methods use matplotlib.pyplot library. Title can be added to the graphs as string by adding the title in the method as parameter.
"""
```

```
26
27 import numpy as np
28 import copy
29 import matplotlib.pyplot as plt
30
31 __author__ = " Mert Saruhan "
32 __maintainer__ = " Mert Saruhan "
33 __email__ = " mertsaruhn@gmail.com "
34
35
36 class OGLVQ:
37     def __init__(self, prototypes: list, learning_rate: float):
38         self.feature_size = len(prototypes[0][0])
39         prototypes_copy = copy.deepcopy(prototypes)
40         self.prototypes = self.create_prototype_dict(prototypes_copy,
    learning_rate)
41         self.datatype = prototypes[0][0].dtype
42         self.labeltype = prototypes[0][1].dtype
43         self.epoch = 0
44         self.history = {
45             "lr": {i: [] for i in range(len(prototypes))},
46             "loss": [],
47             "accuracy": [],
48             "f_score": [],
49         }
50         self.classes = self.get_class(prototypes)
51         self.colors = self.get_colors(prototypes)
52
53     def get_colors(self, prototypes) -> dict:
54         """
55         Divides prototypes into color groups by classes in dictionary
    form
56         For now there are 3 colors: blue, red, green
57         The function used in __init__
58         """
59         color_list = ["#5171fF", "#fF7151", "#519951"]
60         unique_class = self.get_class(prototypes)
61         return {unique_class[i]: color_list[i % 3] for i in range(len(
    unique_class))}
62
63     def get_class(self, prototypes) -> np.ndarray:
64         """
65         Gets the distinct class groups.
66         The function used in __init__
67         """
68         list_labels = []
69         for p in prototypes:
70             list_labels.append(p[1][0])
71         unique_class = list(set(list_labels))  # get rid of duplicates
72         unique_class.sort()
73         unique_class = np.array(unique_class, dtype=self.labeltype)
74         return unique_class
75
76     def create_prototype_dict(self, prototypes, learning_rate) -> dict:
77         """
78         Creates each prototype's local values in __init__ part.
79         """
80         prototypes_dict = {}
```

```
81          for i, p in enumerate(prototypes):
82              prototypes_dict[i] = {"feature": p[0], "label": p[1], "lr":
     learning_rate}
83          return prototypes_dict
84
85      def sigmoid(self, x) -> float:
86          """
87          Activation function for loss
88          """
89          return 1 / (1 + np.exp(-x))
90
91      def prediction(self, x) -> tuple:
92          """
93          Function has one parameter, test features
94          Returns tuple of (winner prototype, winner class)
95
96          Test features should be same lenght as the prototypes
97
98          Winner prototype is the closest prototype to the parameter
    entered
99          Winner class is the class of the winner prototype
100
101         Function has different distance functions for real values and
    complex values
102
103         Real values: sum of square of feature diffrences
104         Complex values: sum of absolute value of feature diffrences
105         """
106         distance = None
107         for prototype, values in self.prototypes.items():
108             if self.datatype == np.csingle:
109                 dist_p_x = np.sum(np.abs(values["feature"] - x) ** 2)
110             else:
111                 dist_p_x = np.sum((values["feature"] - x) ** 2)
112
113             if distance is None:
114                 distance = dist_p_x
115                 winner_class = values["label"]
116                 winner_prototype = prototype
117             elif dist_p_x < distance:
118                 distance = dist_p_x
119                 winner_class = values["label"]
120                 winner_prototype = prototype
121         return winner_prototype, winner_class
122
123     def local_loss(self, x) -> tuple:
124         """
125         Local loss used in model training
126         The model is GLVQ model, so we calculate two winners:
    winner_true, winner_false
127
128         Winner_true: closest prototype to the sample with same class
    than the sample
129         Winner_false: closest prototype to the sample with different
    class than the sample
130
131         Function returns loss, winner_true to sample distance,
    winner_true, winner_false to sample distance, winner_false as tuple
```

```
132         All these values used in prototype update
133         """
134         x_feature, x_label = x
135         d_1 = None
136         d_2 = None
137         for prototype, values in self.prototypes.items():
138             if self.datatype == np.csingle:
139                 dist_p_x = np.sum(np.abs(values["feature"] - x_feature)
    ** 2)
140             else:
141                 dist_p_x = np.sum((values["feature"] - x_feature) ** 2)
142 
143             if values["label"] == x_label:
144                 if d_1 is None:
145                     d_1 = dist_p_x
146                     winner_true = prototype
147                 elif dist_p_x < d_1:
148                     d_1 = dist_p_x
149                     winner_true = prototype
150             else:
151                 if d_2 is None:
152                     d_2 = dist_p_x
153                     winner_false = prototype
154                 elif dist_p_x < d_2:
155                     d_2 = dist_p_x
156                     winner_false = prototype
157 
158         loss = self.sigmoid((d_1 - d_2) / (d_1 + d_2))
159         return loss, d_1, winner_true, d_2, winner_false
160 
161     def train(
162         self,
163         num_epochs: int,
164         training_set: list[tuple[np.array, np.array]],
165         test_set: list[tuple[np.array, np.array]],
166         f_score_beta: float = 1.0,
167         sample_number: dict = None,
168     ) -> dict:
169         """
170         Trains the model returns history of the model as dictionary
171         history = {
172             history of learning rate for each prototype,
173             history of loss,
174             history of accuracy,
175             history of f-score (weighted f-score)
176         }
177         To reach history of any prototype's learning rate use history["
    lr"][prototype_number]
178 
179         Parameters:
180         - num_epochs: number of epochs
181         - training_set: list of tuples (feature, label)
182         - test_set: list of tuples (feature, label)
183         - f_score_beta: beta value for f-score calculation default = 1
184         - sample_number: dictionary of sample numbers for each class (
    class_name: sample_number)
185 
186         sample number is used for weighted f-score calculation
```

```
        """
        if len(self.classes) == 1:
            print("Error: there is only one class in the prototypes")
            return

        if sample_number is None:
            print("Error: sample_number is None")
            return

        sum_samples = sum(sample_number.values())
        sample_weight = {
            class_num: sample / sum_samples
            for class_num, sample in sample_number.items()
        }

        if f_score_beta == int(f_score_beta):
            f_name = int(f_score_beta)
        else:
            f_name = f_score_beta

        for epoch in range(num_epochs):
            # Clear loss
            global_loss = 0
            # Tranining
            for x in training_set:
                x_feature, x_label = x
                loss, d_1, winner_true, d_2, winner_false = self.
local_loss(x)
                _, x_prediction = self.prediction(x_feature)

                # Update global_loss
                global_loss += loss

                common_multiplier = loss * (1 - loss) / ((d_1 + d_2) **
 2)

                # Update learning_rate
                self.prototypes[winner_true]["lr"] = self.prototypes[
winner_true][
                    "lr"
                ] / (
                    1
                    + (
                        1
                        * self.prototypes[winner_true]["lr"]
                        * 4
                        * common_multiplier
                        * d_2
                    )
                )

                self.prototypes[winner_false]["lr"] = self.prototypes[
winner_false][
                    "lr"
                ] / (
                    1
                    + (
                        -1
```

```
                        * self.prototypes[winner_false]["lr"]
                        * 4
                        * common_multiplier
                        * d_1
                    )
                )

                # Update prototypes
                ## update winner_true
                self.prototypes[winner_true]["feature"] += (
                    self.prototypes[winner_true]["lr"]
                    * 4
                    * common_multiplier
                    * d_2
                    * (x_feature - self.prototypes[winner_true]["
feature"])
                )

                ## update winner_false
                self.prototypes[winner_false]["feature"] -= (
                    self.prototypes[winner_false]["lr"]
                    * 4
                    * common_multiplier
                    * d_1
                    * (x_feature - self.prototypes[winner_false]["
feature"])
                )

            # Calculate f-score and accuracy
            correct = 0
            f_dict = {}
            for x in self.classes:
                f_dict[x] = {"TP": 0, "FP": 0, "FN": 0, "TN": 0}

            for x in test_set:
                x_feature, x_label = x
                _, x_prediction = self.prediction(x_feature)

                ## accuracy counter
                if x_prediction == x_label:
                    correct += 1

                ## f-score counter
                for class_name, value in f_dict.items():
                    if x_prediction == x_label:
                        if x_prediction == class_name:
                            value["TP"] += 1
                        else:
                            value["TN"] += 1
                    else:
                        if x_prediction == class_name:
                            value["FP"] += 1
                        else:
                            value["FN"] += 1

            ## calculate accuracy
            acc = correct / len(test_set)

```

```
            ## calculate f-score
            for class_name, value in f_dict.items():
                if value["TP"] == 0:
                    score = 0
                else:
                    precision = value["TP"] / (value["TP"] + value["FP"
])
                    recall = value["TP"] / (value["TP"] + value["FN"])
                    score = (
                        (1 + (f_score_beta**2))
                        * precision
                        * recall
                        / (((f_score_beta**2) * precision) + recall)
                    )
                f_dict[class_name] = score
            weighted_f_score = 0
            for class_name, value in f_dict.items():
                weighted_f_score += value * sample_weight[class_name]

            self.epoch += 1

            # Update history
            ## Update learning rate history
            for i, values in enumerate(self.prototypes.values()):
                self.history["lr"][i].append(values["lr"])
            ## Update loss history
            self.history["loss"].append(global_loss)
            ## Update accuracy history
            self.history["accuracy"].append(acc)
            ## Update f-score history
            self.history["f_score"].append(weighted_f_score)

            if epoch % 10 == 0 or epoch == num_epochs:
                print(
                    f"Epoch: {self.epoch}, Loss: {global_loss:.4f},
Accuracy: {acc*100:.2f} %, F_{f_name}_score: {weighted_f_score
*100:.2f} %"
                )
        return self.history

    def lr_graph(self, title: str = None, marker: str = None) -> plt.
figure:
        """
        Shows learning rate graph for each prototype in combined graph
        Prototypes are grouped by their class with different colors (
for now max 3 colors)

        Function uses matplotlib.pyplot library so use markers
according to matplotlib.pyplot library
        Parameters:
        - title: title of the graph
        - marker: marker of the graph
        """
        used_labels = []
        fig, ax = plt.subplots(figsize=(10, 10))
        for prototype_name, lr in self.history["lr"].items():
            if self.prototypes[prototype_name]["label"][0] in
used_labels:
```

```
                label = None
            else:
                label = self.prototypes[prototype_name]["label"][0]
                used_labels.append(label)
            ax.plot(
                range(self.epoch),
                lr,
                label=label,
                color=self.colors[self.prototypes[prototype_name]["
label"][0]],
                linestyle="dashed",
                marker=marker,
            )
        plt.xlabel("Epoch (t)", fontsize=25, weight="bold")
        plt.legend()
        plt.yticks(fontsize=20)
        plt.xticks(fontsize=20)
        if title:
            plt.title(title, fontsize=40)
        plt.show()
        return fig

    def acc_graph(self, title: str = None):
        """
        Shows accuracy graph of the model

        Function uses matplotlib.pyplot library so use markers
according to matplotlib.pyplot library
        Parameters:
        - title: title of the graph
        """
        fig, ax = plt.subplots(figsize=(10, 10))
        ax.plot(
            range(self.epoch),
            self.history["accuracy"],
        )
        plt.xlabel("Epoch (t)", fontsize=25, weight="bold")
        plt.ylim(0, 1.01)
        plt.yticks(
            np.arange(0, 1.01, step=0.2),
            ["0%", "20%", "40%", "60%", "80%", "100%"],
            fontsize=20,
        )
        plt.xticks(fontsize=20)
        if title:
            plt.title(title, fontsize=40)
        plt.show()
        return fig

    def f1_graph(self, title: str = None):
        """
        Shows weighted f-score graph of the model

        Function uses matplotlib.pyplot library so use markers
according to matplotlib.pyplot library
        Parameters:
        - title: title of the graph
        """
```

```
        fig, ax = plt.subplots(figsize=(10, 10))
        ax.plot(
            range(self.epoch),
            self.history["f_score"],
        )
        plt.xlabel("Epoch (t)", fontsize=25, weight="bold")
        plt.ylim(0, 1.01)
        plt.yticks(
            np.arange(0, 1.01, step=0.2),
            ["0%", "20%", "40%", "60%", "80%", "100%"],
            fontsize=20,
        )
        plt.xticks(fontsize=20)
        if title:
            plt.title(title, fontsize=40)
        plt.show()
        return fig


```

CGLVQ

```
"""
The model uses GLVQ as base model and have learning rate methods from
    cognitive science
learning rate methods is in optimizers.py file
optimizers:
- Conditional Probalility
- Dual Factor Heuristic
- Middle Symmetry (alpha = 1, beta = 0)
- Loose Symmetry
- Loose Symmetry with Rarity

the model includes two performance measures:

- Accuracy
- F-Score (weighed average)

To use the model please import the file and use the class CGLVQ. Then
    use class method train with the following parameters:
    num_epochs: train time
    training_set: adjust the training set as: list[tuple[np.array, np.
    array]],
    test_set: adjust the test set as: list[tuple[np.array, np.array]],
    optimizer: import the optimizers from optimizers.py and use them as
     optimizer=optimizer_name,
    validation_set: if you want to use validation set adjust the
    validation set as: list[tuple[np.array, np.array]] = None,
    f_score_beta: beta value of the F score, default = 1 any float
    value can be used,
    sample_number: Number of training samples each class uses. It is
    needed to calculate the weighted F scores

One can use the following methods to see the results:
    lr_graph: shows the learning rate graph for each prototype
    acc_graph: shows the accuracy graph
```

```
    f1_graph: shows the f1 score graph

    methods use matplotlib.pyplot library. Title can be added to the
    graphs as string by adding the title in the method as parameter.
"""

import numpy as np
import copy
import matplotlib.pyplot as plt

__author__ = " Mert Saruhan "
__maintainer__ = " Mert Saruhan "
__email__ = " mertsaruhn@gmail.com "


class CGLVQ:
    def __init__(self, prototypes: list, lr: float):
        self.feature_size = len(prototypes[0][0])
        prototypes_copy = copy.deepcopy(prototypes)
        self.global_lr = lr
        self.prototypes = self.create_prototype_dict(prototypes_copy,
    lr)
        self.datatype = prototypes[0][0].dtype
        self.labeltype = prototypes[0][1].dtype
        self.epoch = 0
        self.history = {
            "lr": {i: [] for i in range(len(prototypes))},
            "loss": [],
            "accuracy": [],
            "f_score": [],
        }
        self.classes = self.get_class(prototypes)
        self.colors = self.get_colors(prototypes)

    def get_colors(self, prototypes) -> dict:
        """
        Divides prototypes into color groups by classes in dictionary
    form
        For now there are 3 colors: blue, red, green
        The function used in __init__
        """
        color_list = ["#5171fF", "#fF7151", "#519951"]
        unique_class = self.get_class(prototypes)
        return {unique_class[i]: color_list[i % 3] for i in range(len(
    unique_class))}

    def get_class(self, prototypes) -> np.ndarray:
        """
        Gets the distinct class groups.
        The function used in __init__
        """
        list_labels = []
        for p in prototypes:
            list_labels.append(p[1][0])
        unique_class = list(set(list_labels))  # get rid of duplicates
        unique_class.sort()
        unique_class = np.array(unique_class, dtype=self.labeltype)
        return unique_class
```

```
82
83      def create_prototype_dict(self, prototypes, lr) -> dict:
84          """
85          Creates each prototype's local values in __init__ part.
86          """
87          prototypes_dict = {}
88          for i, p in enumerate(prototypes):
89              prototypes_dict[i] = {"feature": p[0], "label": p[1], "lr":
        lr}
90          return prototypes_dict
91
92      def sigmoid(self, x):
93          """
94          Activation function for loss
95          """
96          return 1 / (1 + np.exp(-x))
97
98      def prediction(self, x) -> str:
99          """
100         Function has one parameter, test features
101         Returns winner prototype number
102
103         Test features should be same lenght as the prototypes
104
105         Winner prototype is the closest prototype to the parameter
    entered
106
107         Function has different distance functions for real values and
    complex values
108
109         Real values: sum of square of feature diffrences
110         Complex values: sum of absolute value of feature diffrences
111         """
112         distance = None
113         for values in self.prototypes.values():
114             if self.datatype == np.csingle:
115                 dist_p_x = np.sum(np.abs(values["feature"] - x) ** 2)
116             else:
117                 dist_p_x = np.sum((values["feature"] - x) ** 2)
118
119             if distance is None:
120                 distance = dist_p_x
121                 winner = values["label"]
122             elif dist_p_x < distance:
123                 distance = dist_p_x
124                 winner = values["label"]
125         return winner
126
127     def local_loss(self, x) -> tuple:
128         """
129         Local loss used in model training
130         The model is GLVQ model, so we calculate two winners:
    winner_true, winner_false
131
132         Winner_true: closest prototype to the sample with same class
    than the sample
133         Winner_false: closest prototype to the sample with different
    class than the sample
```

```
134
135         Function returns loss, winner_true to sample distance,
    winner_true, winner_false to sample distance, winner_false as tuple
136         All these values used in prototype update
137         """
138         x_feature, x_label = x
139         d_1 = None
140         d_2 = None
141         for prototype, values in self.prototypes.items():
142             if self.datatype == np.csingle:
143                 dist_p_x = np.sum(np.abs(values["feature"] - x_feature)
     ** 2)
144             else:
145                 dist_p_x = np.sum((values["feature"] - x_feature) ** 2)
146             if values["label"] == x_label:
147                 if d_1 is None:
148                     d_1 = dist_p_x
149                     winner_true = prototype
150                 elif dist_p_x < d_1:
151                     d_1 = dist_p_x
152                     winner_true = prototype
153             else:
154                 if d_2 is None:
155                     d_2 = dist_p_x
156                     winner_false = prototype
157                 elif dist_p_x < d_2:
158                     d_2 = dist_p_x
159                     winner_false = prototype
160
161         loss = self.sigmoid((d_1 - d_2) / (d_1 + d_2))
162         return loss, d_1, winner_true, d_2, winner_false
163
164     def train(
165         self,
166         num_epochs: int,
167         training_set: list[tuple[np.array, np.array]],
168         test_set: list[tuple[np.array, np.array]],
169         optimizer: callable,
170         validation_set: list[tuple[np.array, np.array]] = None,
171         f_score_beta: float = 1,
172         sample_number: dict = None,
173     ) -> dict:
174         """
175         Trains the model.
176         If validation_set is not None, the loss will be calculated with
     the validation set.
177         Else, the loss will be calculated with the training set.
178
179         Trains the model returns history of the model as dictionary
180         history = {
181             history of learning rate for each prototype,
182             history of loss,
183             history of accuracy,
184             history of f-score (weighted f-score)
185         }
186         To reach history of any prototype's learning rate use history["
    lr"][prototype_number]
187
```

```
        Parameters:
        - num_epochs: number of epochs
        - training_set: list of tuples (feature, label)
        - test_set: list of tuples (feature, label)
        - optimizer: function to update the learning rate
        - validation_set: validation set list of tuples (feature, label)
        - f_score_beta: beta value for f-score calculation default = 1
        - sample_number: dictionary of sample numbers for each class (class_name: sample_number)

        sample number is used for weighted f-score calculation


        """

        if len(self.classes) == 1:
            print("Error: there is only one class in the prototypes")
            return

        if sample_number is None:
            print("Error: sample_number is None")
            return

        sum_samples = sum(sample_number.values())
        sample_weight = {
            class_num: sample / sum_samples
            for class_num, sample in sample_number.items()
        }

        if f_score_beta == int(f_score_beta):
            f_name = int(f_score_beta)
        else:
            f_name = f_score_beta

        for epoch in range(num_epochs):
            # Clear accurence_frequncy
            for values in self.prototypes.values():
                values.update({"a": 0, "b": 0, "c": 0, "d": 0})

            # Clear loss
            global_loss = 0

            for x in training_set:
                x_feature, x_label = x
                loss, d_1, winner_true, d_2, winner_false = self.local_loss(x)
                x_prediction = self.prediction(x_feature)

                # Update global_loss
                if validation_set is None:
                    global_loss += loss

                # Update accurence_frequncy
                for values in self.prototypes.values():
                    if values["label"] == x_prediction and x_label == x_prediction:
                        values["a"] += 1
```

```
                    elif values["label"] == x_prediction and x_label !=
 x_prediction:
                        values["b"] += 1
                    elif values["label"] != x_prediction and x_label ==
 x_prediction:
                        values["c"] += 1
                    elif values["label"] != x_prediction and x_label !=
 x_prediction:
                        values["d"] += 1

            # Update learning rate
            for values in self.prototypes.values():
                optimizer(values=values, global_lr=self.global_lr)

            # Update prototypes
            common_multiplier = 4 * loss * (1 - loss) / ((d_1 + d_2
) ** 2)

            ## update winner_true
            self.prototypes[winner_true]["feature"] += (
                self.prototypes[winner_true]["lr"]
                * common_multiplier
                * d_2
                * (x_feature - self.prototypes[winner_true]["
feature"])
            )

            ## update winner_false
            self.prototypes[winner_false]["feature"] -= (
                self.prototypes[winner_true]["lr"]
                * common_multiplier
                * d_1
                * (x_feature - self.prototypes[winner_false]["
feature"])
            )

        if validation_set is not None:
            for x in validation_set:
                loss, _, _, _, _ = self.local_loss(x)
                global_loss += loss
            global_loss /= len(validation_set)
        else:
            global_loss /= len(training_set)

        # Calculate f-score and accuracy
        correct = 0
        f_dict = {}
        for x in self.classes:
            f_dict[x] = {"TP": 0, "FP": 0, "FN": 0, "TN": 0}

        for x in test_set:
            x_feature, x_label = x
            x_prediction = self.prediction(x_feature)

            ## accuracy counter
            if x_prediction == x_label:
                correct += 1

```

```
                ## f-score counter
                for class_name, value in f_dict.items():
                    if x_prediction == x_label:
                        if x_prediction == class_name:
                            value["TP"] += 1
                        else:
                            value["TN"] += 1
                    else:
                        if x_prediction == class_name:
                            value["FP"] += 1
                        else:
                            value["FN"] += 1

            ## calculate accuracy
            acc = correct / len(test_set)

            ## calculate f-score
            for class_name, value in f_dict.items():
                if value["TP"] == 0:
                    score = 0
                else:
                    precision = value["TP"] / (value["TP"] + value["FP"
    ])
                    recall = value["TP"] / (value["TP"] + value["FN"])
                    score = (
                        (1 + (f_score_beta**2))
                        * precision
                        * recall
                        / (((f_score_beta**2) * precision) + recall)
                    )
                f_dict[class_name] = score
            weighted_f_score = 0
            for class_name, value in f_dict.items():
                weighted_f_score += value * sample_weight[class_name]

            self.epoch += 1

            # Update history
            ## Update lr_history
            for i, values in enumerate(self.prototypes.values()):
                self.history["lr"][i].append(values["lr"])
            ## Update loss_history
            self.history["loss"].append(global_loss)
            ## Update accuracy_history
            self.history["accuracy"].append(acc)
            ## Update f_score_history
            self.history["f_score"].append(weighted_f_score)

            if epoch % 10 == 0 or epoch == num_epochs:
                print(
                    f"Epoch: {self.epoch}, Loss: {global_loss:.4f},
    Accuracy: {acc*100:.2f} %, F_{f_name}_score: {weighted_f_score
    *100:.2f} %"
                )
        return self.history

    def lr_graph(self, title: str = None, marker: str = None) -> plt.
    figure:
```

```
348         """
349         Shows learning rate graph for each prototype in combined graph
350         Prototypes are grouped by their class with different colors (
    for now max 3 colors)
351 
352         Function uses matplotlib.pyplot library so use markers
    according to matplotlib.pyplot library
353         Parameters:
354         - title: title of the graph
355         - marker: marker of the graph
356         """
357         used_labels = []
358         fig, ax = plt.subplots(figsize=(10, 10))
359         for prototype_name, lr in self.history["lr"].items():
360             if self.prototypes[prototype_name]["label"][0] in
    used_labels:
361                 label = None
362             else:
363                 label = self.prototypes[prototype_name]["label"][0]
364                 used_labels.append(label)
365             ax.plot(
366                 range(self.epoch),
367                 lr,
368                 label=label,
369                 color=self.colors[self.prototypes[prototype_name]["
    label"][0]],
370                 linestyle="dashed",
371                 marker=marker,
372             )
373         plt.xlabel("Epoch (t)", fontsize=25, weight="bold")
374         plt.legend()
375         plt.yticks(fontsize=20)
376         plt.xticks(fontsize=20)
377         if title:
378             plt.title(title, fontsize=40)
379         plt.show()
380         return fig
381 
382     def acc_graph(self, title: str = None) -> plt.figure:
383         """
384         Shows accuracy graph of the model
385 
386         Function uses matplotlib.pyplot library so use markers
    according to matplotlib.pyplot library
387         Parameters:
388         - title: title of the graph
389         """
390         fig, ax = plt.subplots(figsize=(10, 10))
391         ax.plot(
392             range(self.epoch),
393             self.history["accuracy"],
394         )
395         plt.xlabel("Epoch (t)", fontsize=25, weight="bold")
396         plt.ylim(0, 1.01)
397         plt.yticks(
398             np.arange(0, 1.01, step=0.2),
399             ["0%", "20%", "40%", "60%", "80%", "100%"],
400             fontsize=20,
```

```
        )
        plt.xticks(fontsize=20)
        if title:
            plt.title(title, fontsize=40)
        plt.show()
        return fig

    def f1_graph(self, title: str = None) -> plt.figure:
        """
        Shows weighted f-score graph of the model

        Function uses matplotlib.pyplot library so use markers according to matplotlib.pyplot library
        Parameters:
        - title: title of the graph
        """
        fig, ax = plt.subplots(figsize=(10, 10))
        ax.plot(
            range(self.epoch),
            self.history["f_score"],
        )
        plt.xlabel("Epoch (t)", fontsize=25, weight="bold")
        plt.ylim(0, 1.01)
        plt.yticks(
            np.arange(0, 1.01, step=0.2),
            ["0%", "20%", "40%", "60%", "80%", "100%"],
            fontsize=20,
        )
        plt.xticks(fontsize=20)
        if title:
            plt.title(title, fontsize=40)
        plt.show()
        return fig
```

Optimizers

```
"""
File contains optimizer functions for the learning rate of cognitive GLVQ (CGLVQ) model.
"""

import numpy as np

__author__ = " Mert Saruhan "
__maintainer__ = " Mert Saruhan "
__email__ = " mertsaruhn@gmail.com "


# Update the learning rate of the prototypes
def middle_symmetry(
    values: dict, global_lr: float, lr_alpha: float = 1, lr_beta: float = 0
) -> None:
    """
    updates the learning rate of the prototypes based on the middle symmetry with alpha = 1 and beta = 0
```

```
    """
    if values["a"] == 0:
        R = 0
    else:
        R = (values["a"] + (lr_beta * values["d"])) / (
            values["a"]
            + (lr_beta * values["d"])
            + values["b"]
            + (lr_alpha * values["c"])
        )
    updated_lr = global_lr * (1 - R)
    values.update({"lr": updated_lr})


def conditional_probability(values: dict, global_lr: float) -> None:
    """
    updates the learning rate of the prototypes based on the
    conditional probability
    """
    if values["a"] == 0:
        R = 0
    else:
        R = values["a"] / (values["a"] + values["b"])
    updated_lr = global_lr * (1 - R)
    values.update({"lr": updated_lr})


def dual_factor_heuristic(values: dict, global_lr: float) -> None:
    """
    updates the learning rate of the prototypes based on the dual
    factor heuristic
    """
    if values["a"] == 0:
        R = 0
    else:
        R = values["a"] / np.sqrt(
            (values["a"] + values["b"]) * (values["a"] + values["c"])
        )
    updated_lr = global_lr * (1 - R)
    values.update({"lr": updated_lr})


def loose_symmetry(values: dict, global_lr: float) -> None:
    """
    updates the learning rate of the prototypes based on the loose
    symmetry
    """
    if values["a"] == 0:
        if values["b"] == 0:
            R = 0
        else:
            R = (values["b"] * values["d"] / (values["b"] + values["d"
    ])) / (
                (values["b"] * values["d"] / (values["b"] + values["d"
    ])) + values["b"]
            )
    elif values["b"] == 0:
        if values["c"] == 0:
```

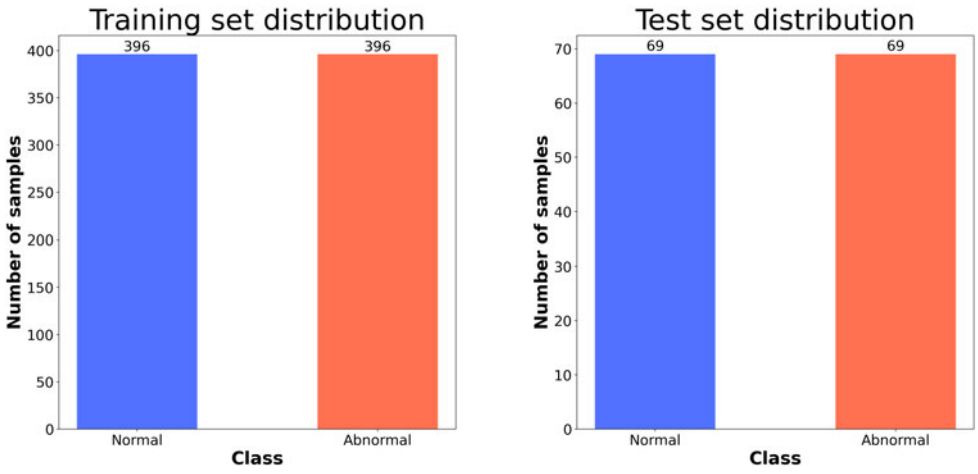
```
            R = 1
        else:
            R = values["a"] / (
                values["a"] + (values["c"] * values["a"] / (values["c"]
    + values["a"]))
            )

    else:
        R = (
            values["a"] + (values["b"] * values["d"] / (values["b"] +
    values["d"]))
        ) / (
            values["a"]
            + (values["b"] * values["d"] / (values["b"] + values["d"]))
            + values["b"]
            + (values["c"] * values["a"] / (values["c"] + values["a"]))
        )
    updated_lr = global_lr * (1 - R)
    values.update({"lr": updated_lr})


def loose_symmetry_rarity(values: dict, global_lr: float) -> None:
    """
    updates the learning rate of the prototypes based on the loose
    symmetry with rarity
    Loose symmetry with rarity is loose symmetry when d -> infinty
    """
    if values["a"] == 0:
        if values["b"] == 0:
            R = 0
        else:
            R = 0.5

    elif values["b"] == 0:
        if values["c"] == 0:
            R = 1
        else:
            R = values["a"] / (
                values["a"] + (values["c"] * values["a"] / (values["c"]
    + values["a"]))
            )
    elif values["c"] == 0:
        R = (values["a"] + values["b"]) / (values["a"] + (2 * values["b
    "]))

    else:
        R = (values["a"] + values["b"]) / (
            values["a"]
            + (2 * values["b"])
            + ((values["a"] * values["c"]) / (values["a"] + values["c"
    ]))
        )
    updated_lr = global_lr * (1 - R)
    values.update({"lr": updated_lr})
```

Appendix B

SP and NSP Results

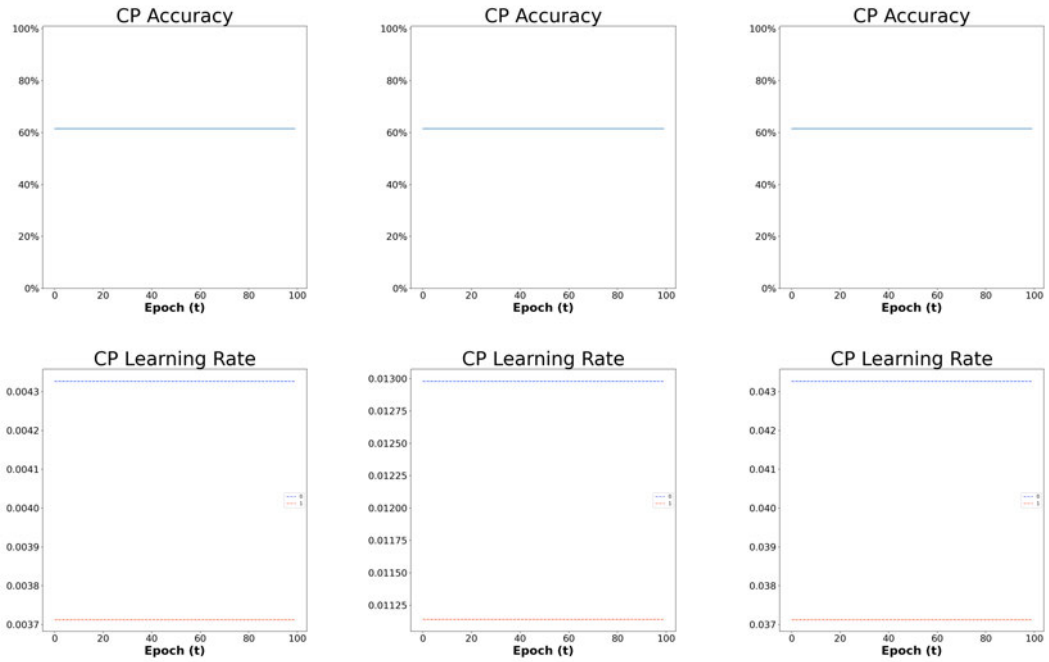
SP Results

Experiment 1



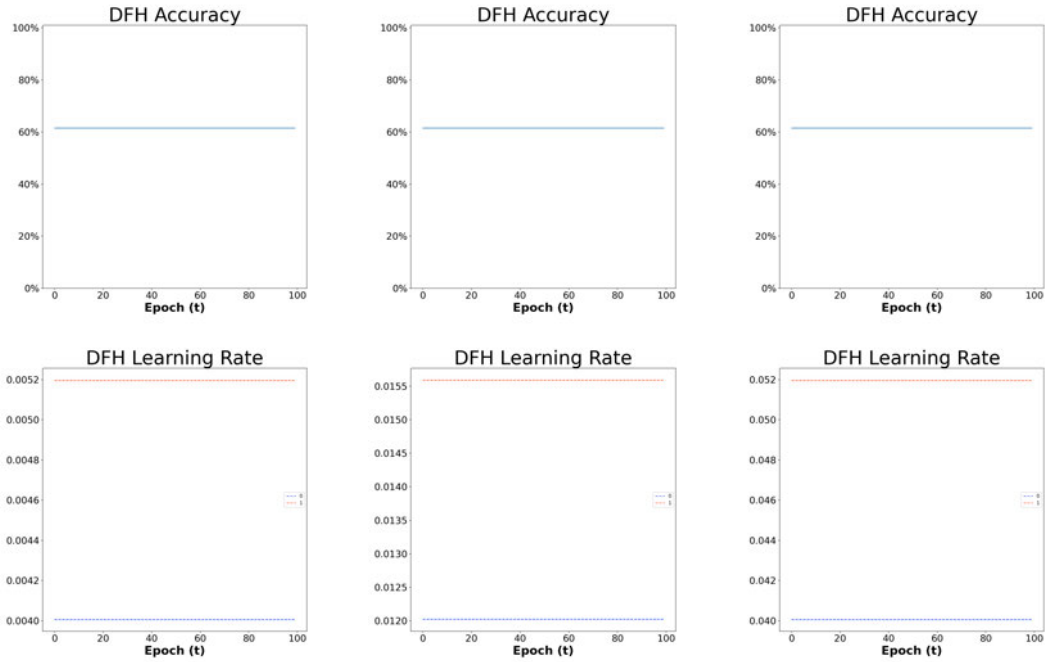
(a) Training set (b) Test set

Figure B.1: *SP* and *NSP* balanced datasets sample distribution.



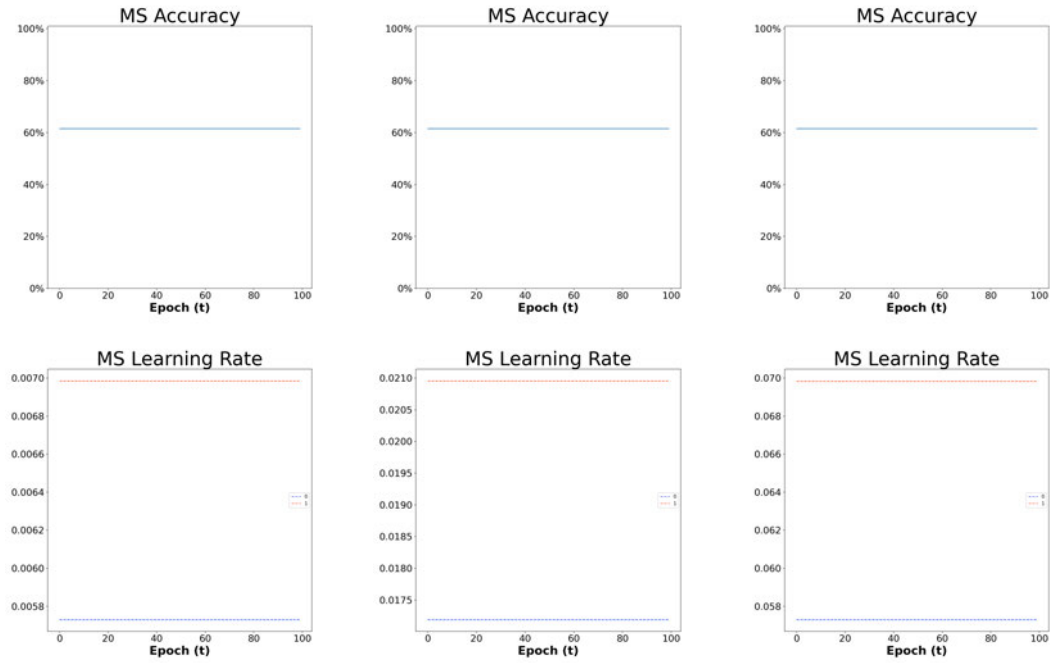
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.2: *SP* dataset accuracy score and learning rate results under CP model using balanced dataset.



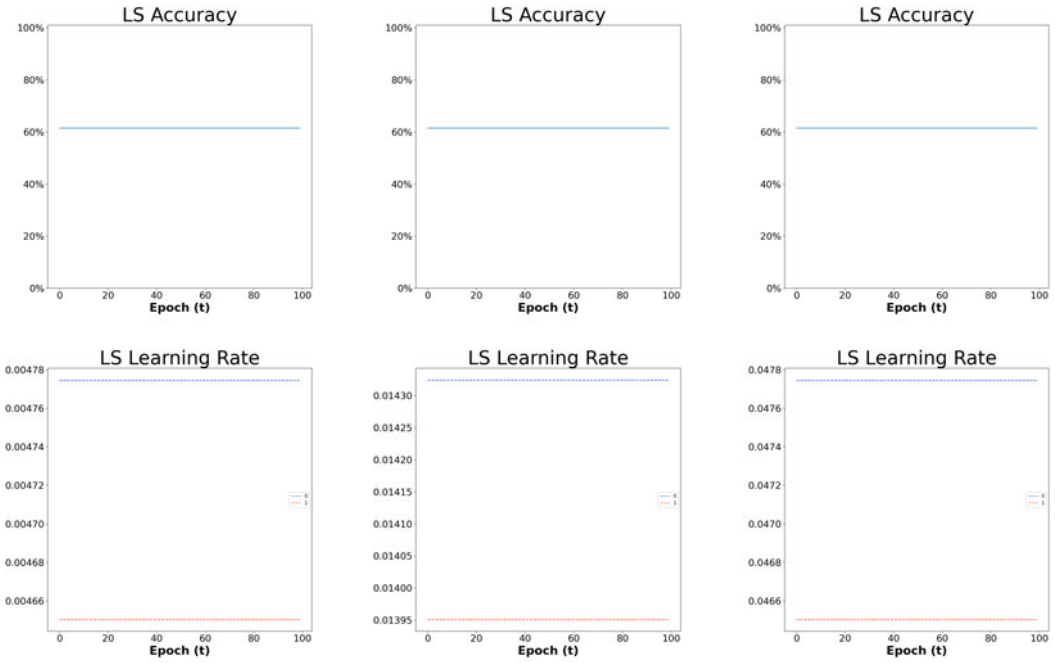
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.3: *SP* dataset accuracy score and learning rate results under DFH model using balanced dataset using balanced dataset.



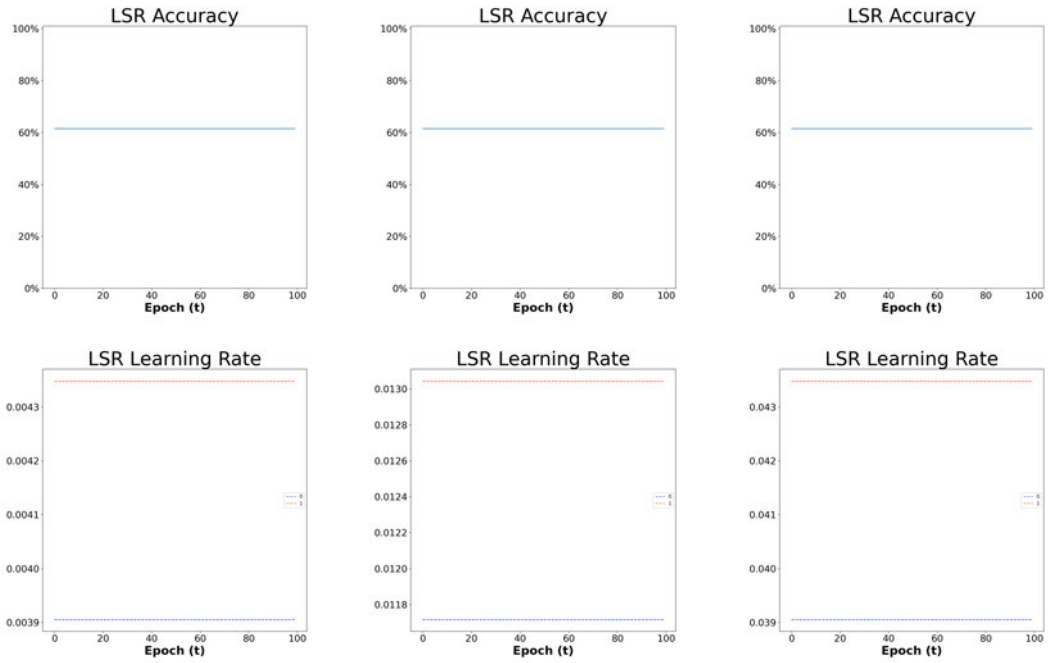
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.4: *SP* dataset accuracy score and learning rate results under MS model using balanced dataset.



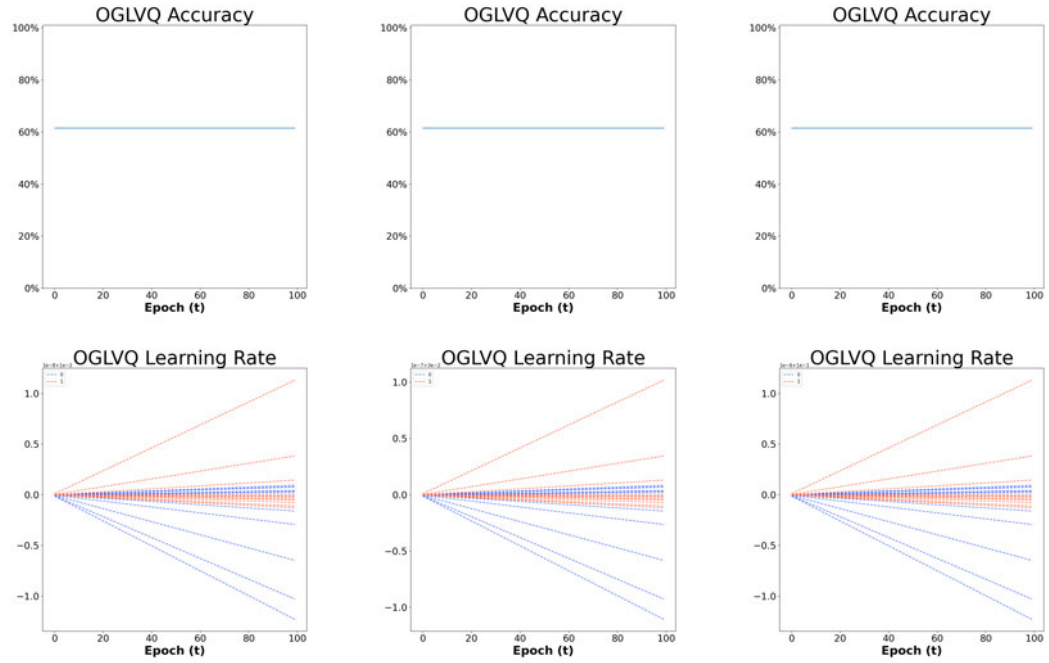
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.5: *SP* dataset accuracy score and learning rate results under LS model using balanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

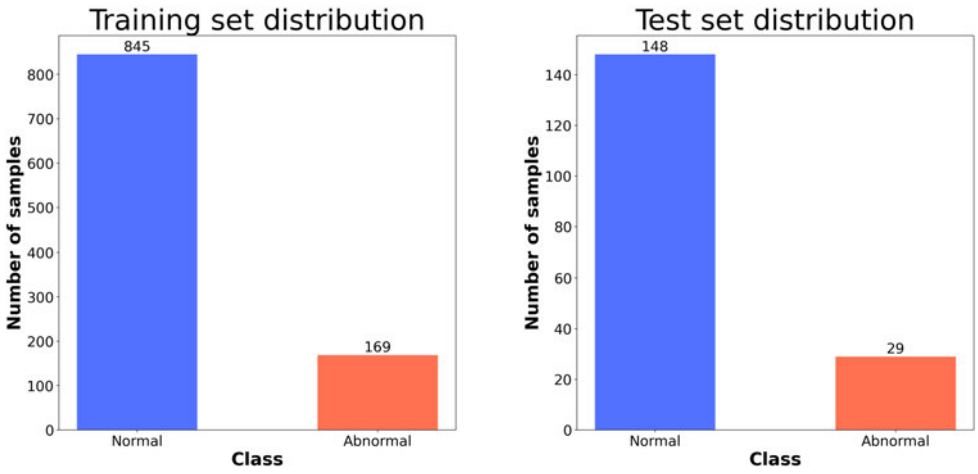
Figure B.6: *SP* dataset accuracy score and learning rate results under LSR model using balanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

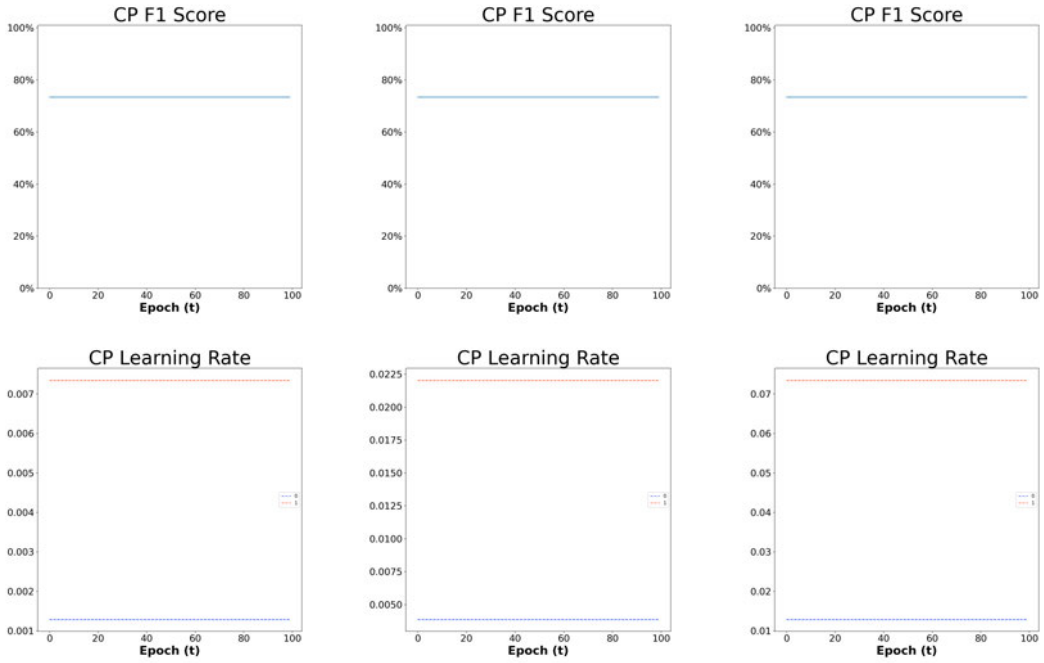
Figure B.7: *SP* dataset accuracy score and learning rate results under OGLVQ model using balanced dataset.

Experiment 2



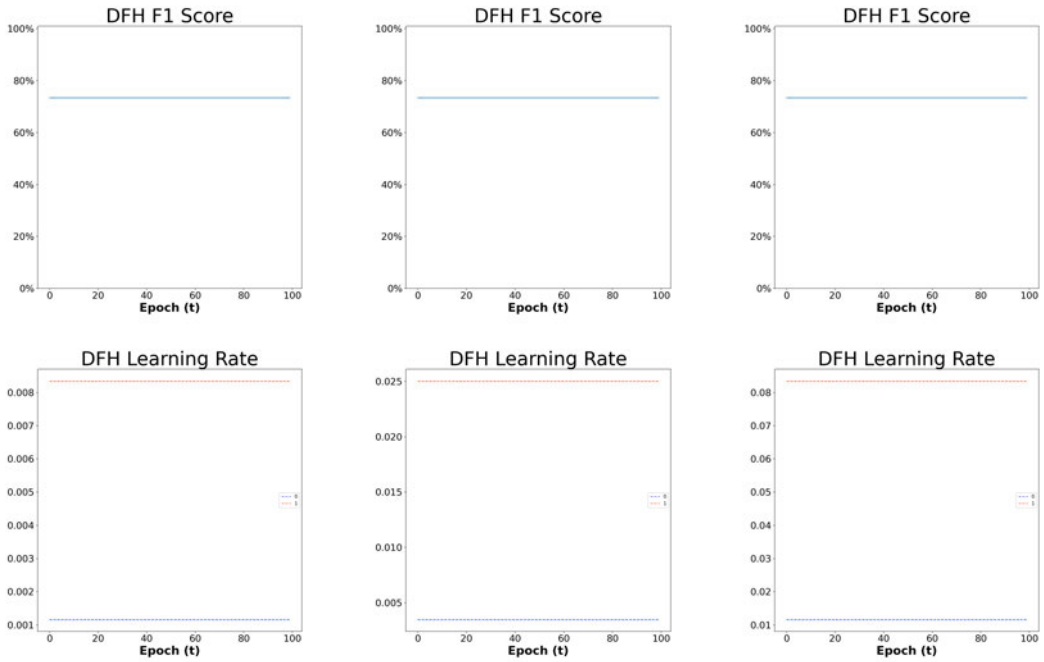
(a) Training set (b) Test set

Figure B.8: *SP* and *NSP* imbalanced datasets sample distribution.



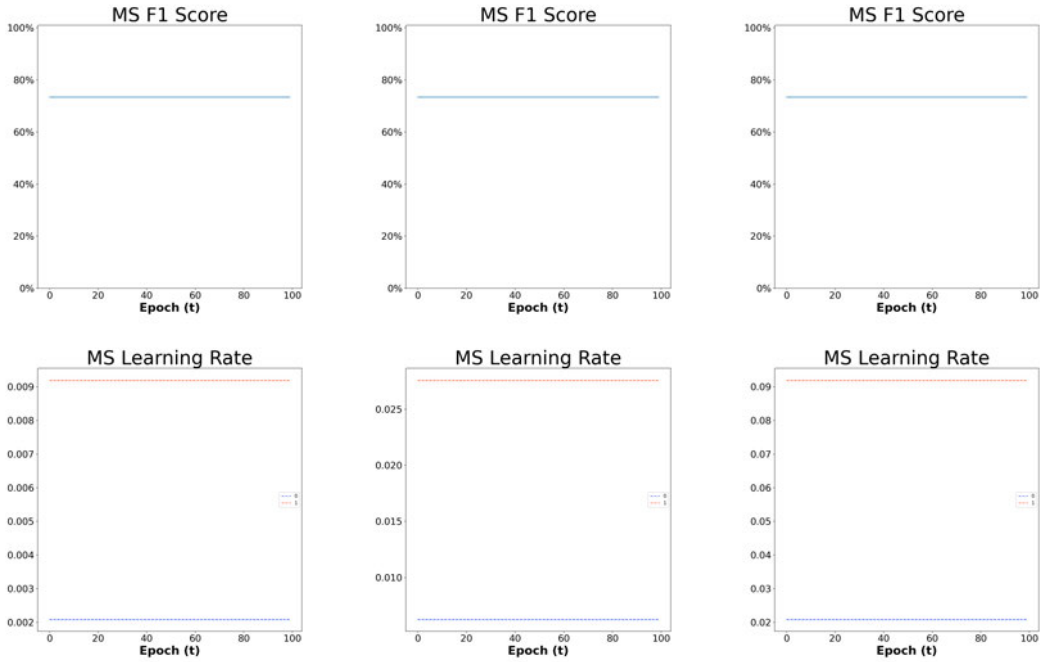
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.9: *SP* dataset F1 score and learning rate results under CP model using imbalanced dataset.



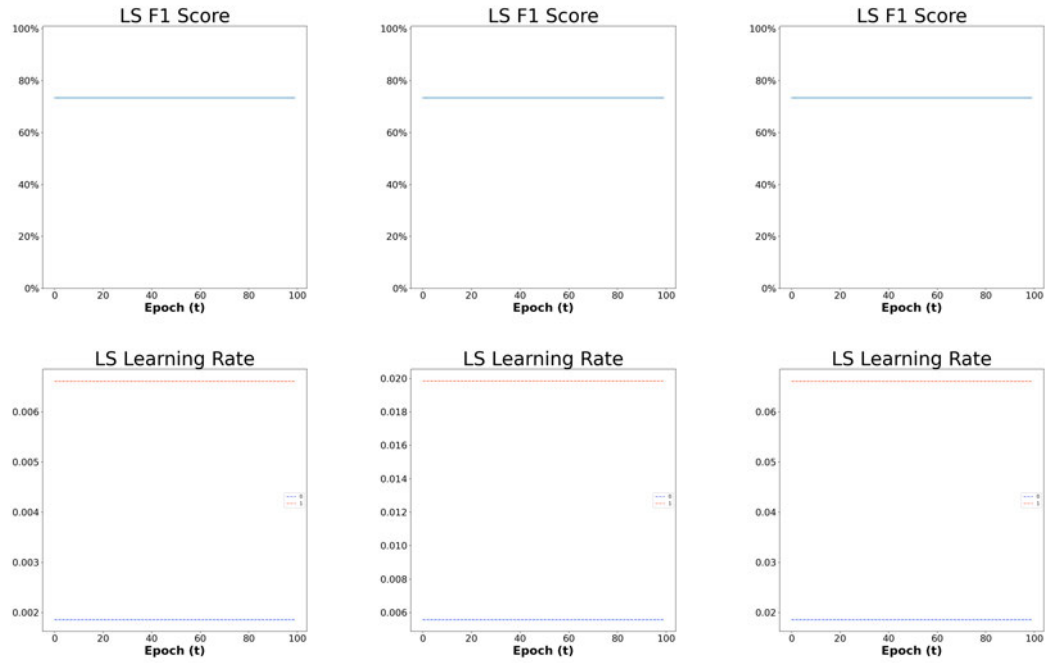
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.10: *SP* dataset F1 score and learning rate results under DFH model using imbalanced dataset.



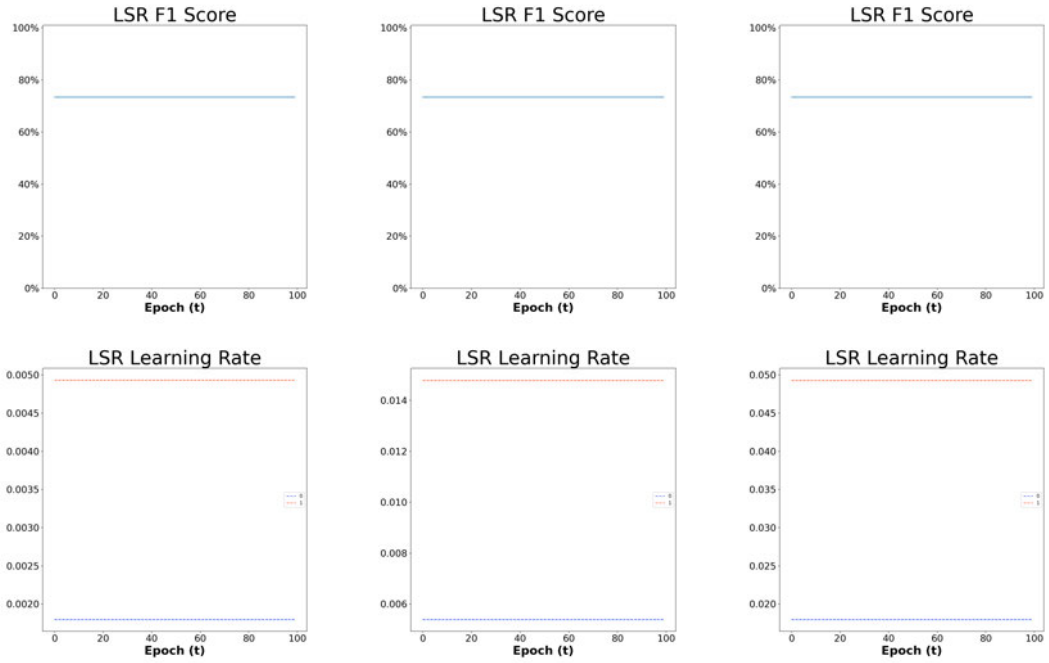
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.11: *SP* dataset F1 score and learning rate results under MS model using imbalanced dataset.



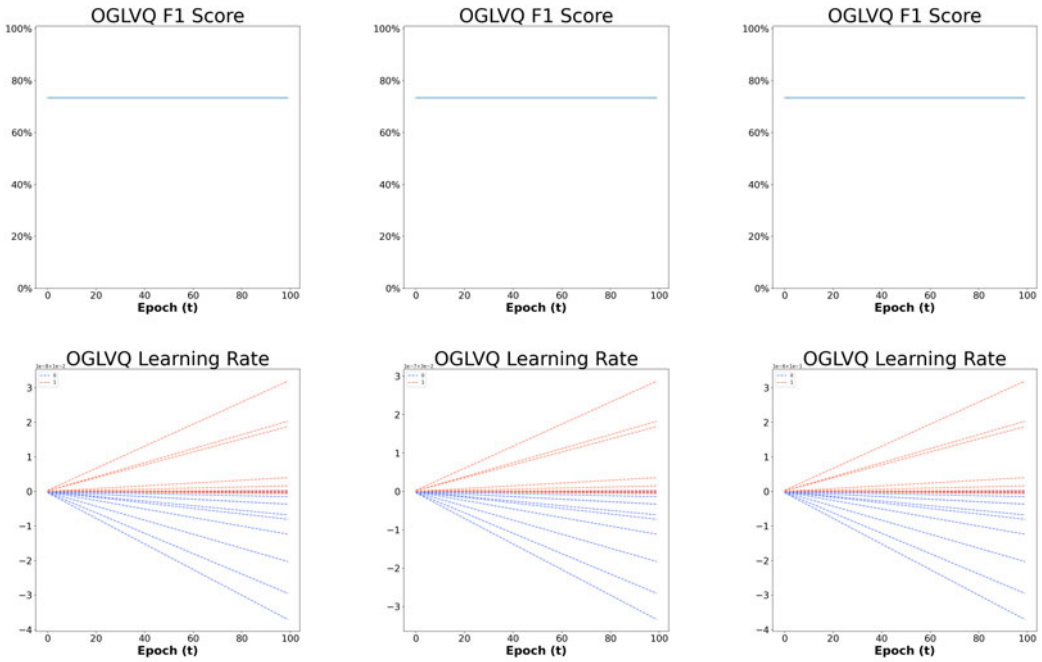
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.12: *SP* dataset F1 score and learning rate results under LS model using imbalanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.13: *SP* dataset F1 score and learning rate results under LSR model using imbalanced dataset.

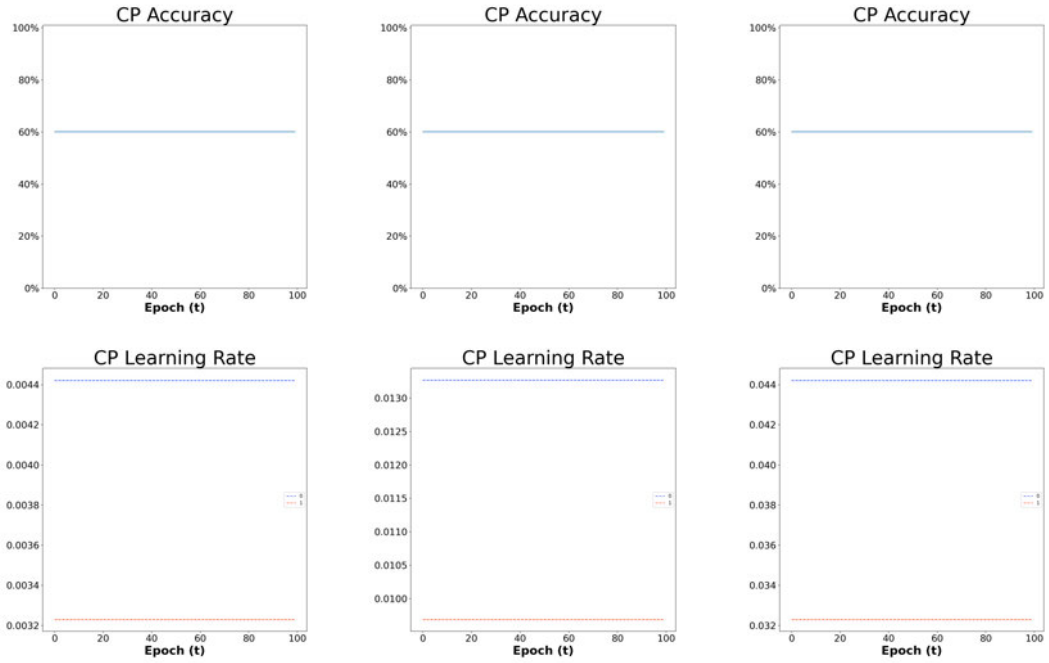


(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.14: *SP* dataset F1 score and learning rate results under OGLVQ model using imbalanced dataset.

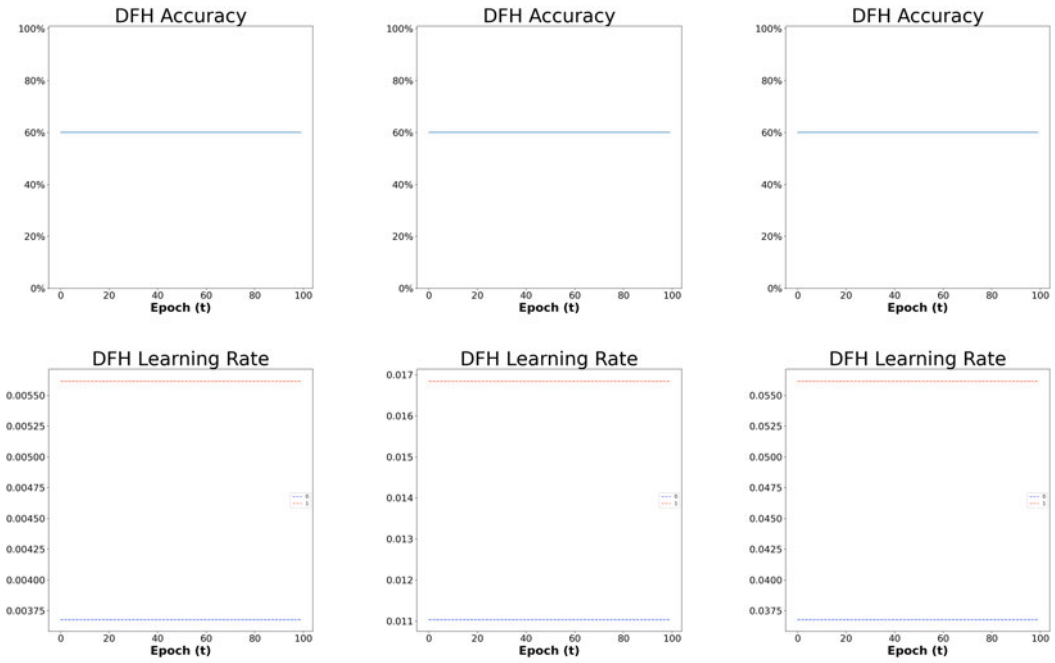
NSP Results

Experiment 1



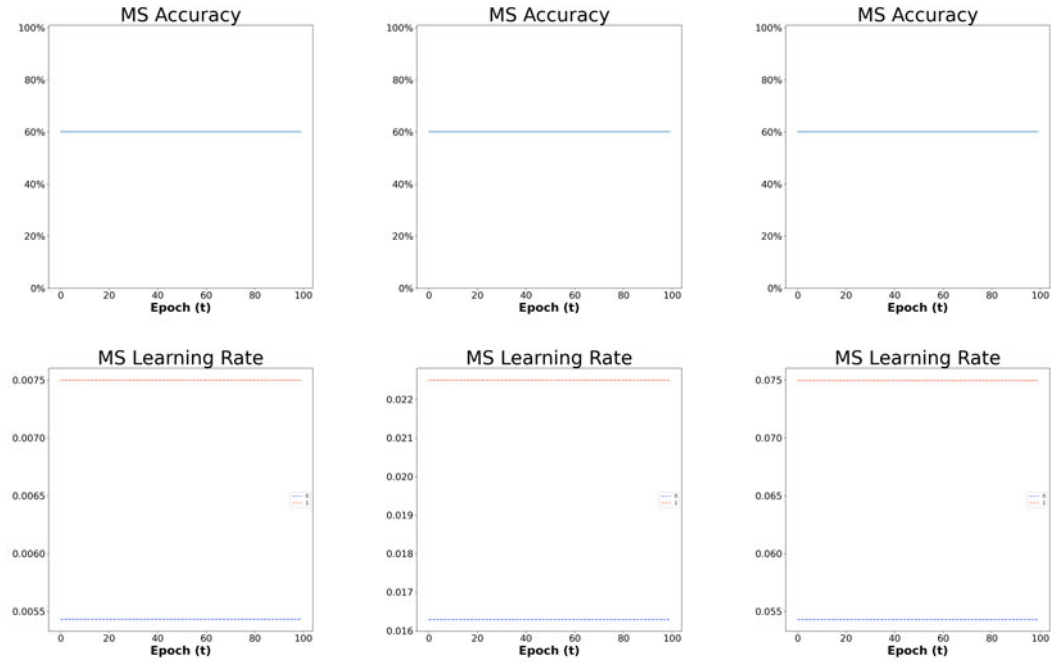
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.15: *NSP* dataset accuracy score and learning rate results under CP model using balanced dataset.



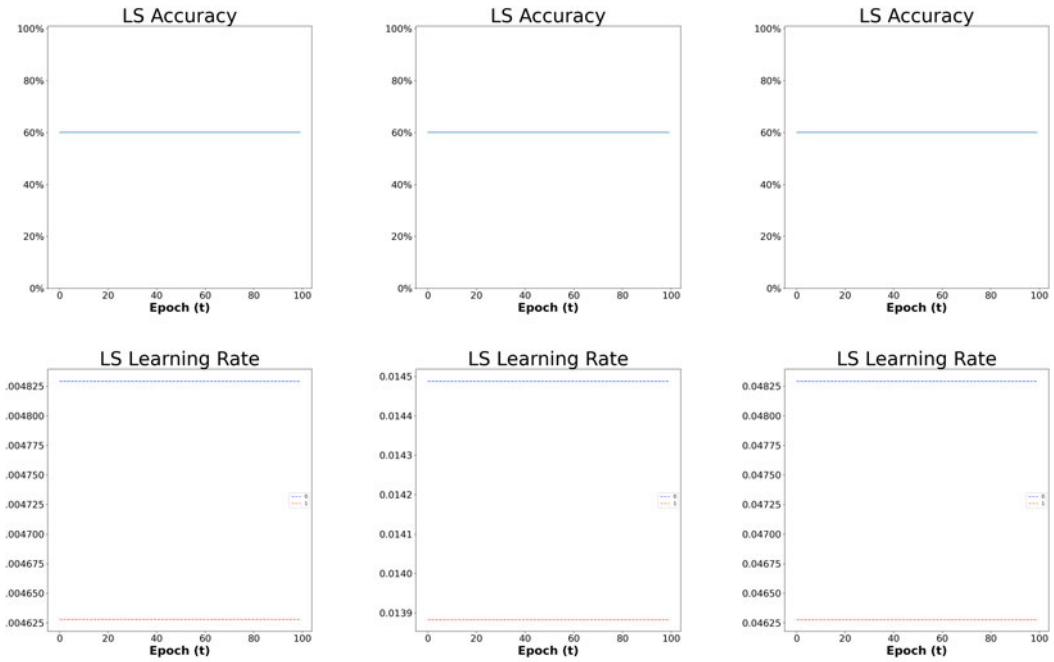
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.16: *NSP* dataset accuracy score and learning rate results under DFH model using balanced dataset using balanced dataset.



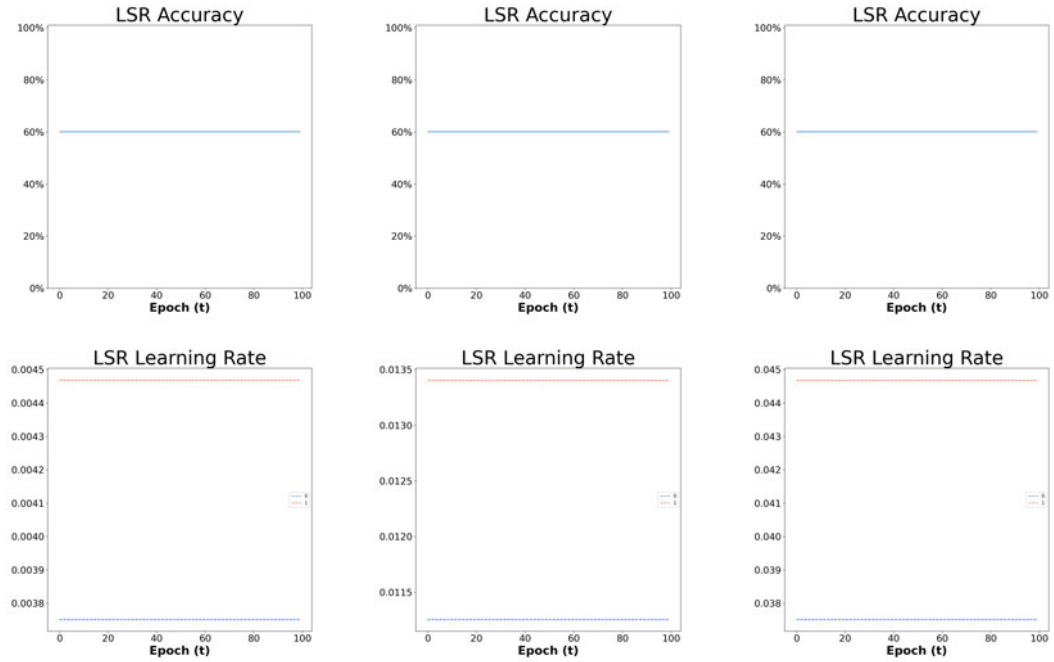
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.17: *NSP* dataset accuracy score and learning rate results under MS model using balanced dataset.



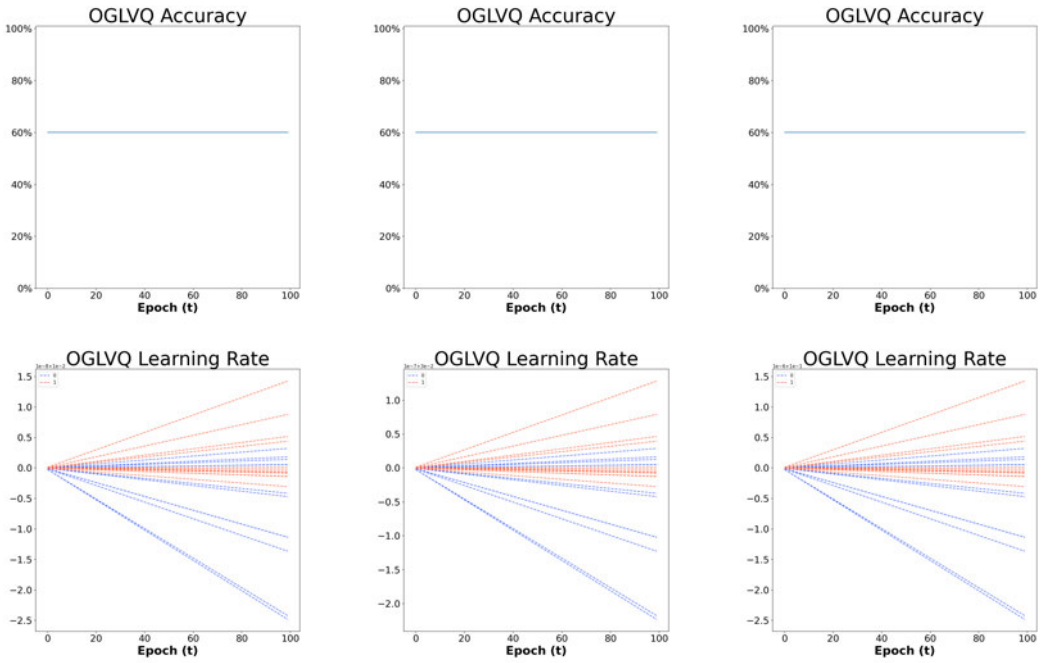
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.18: *NSP* dataset accuracy score and learning rate results under LS model using balanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

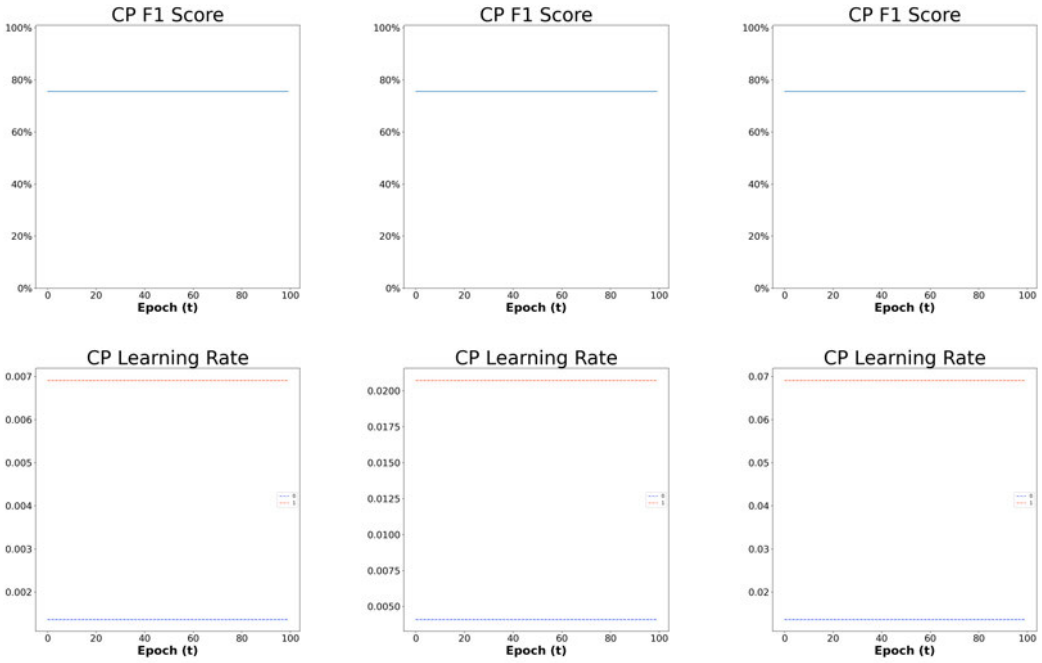
Figure B.19: *NSP* dataset accuracy score and learning rate results under LSR model using balanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

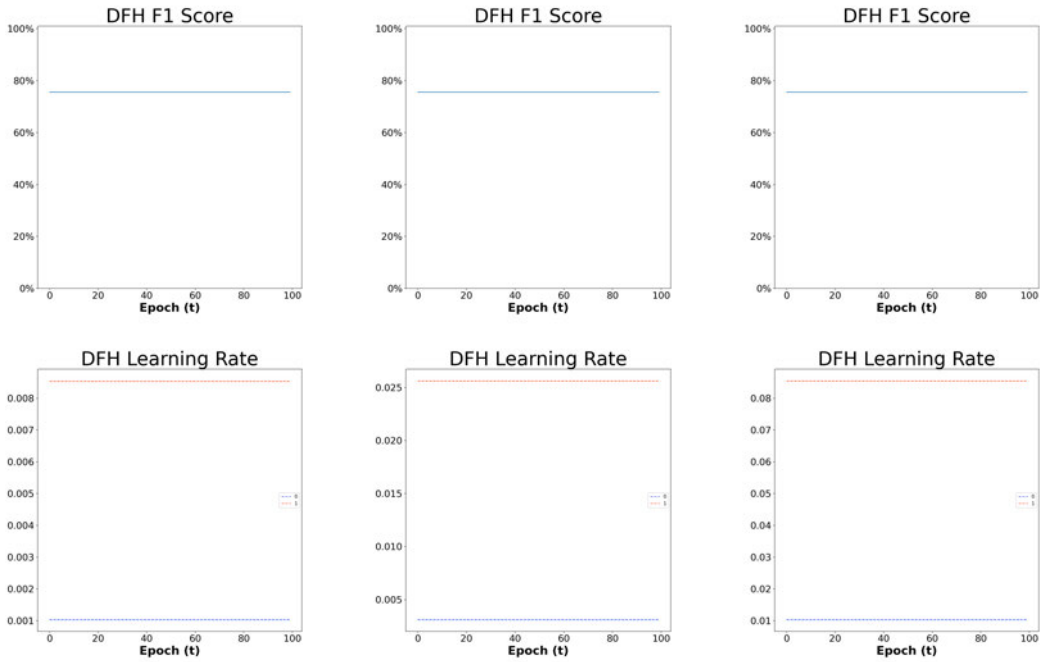
Figure B.20: *NSP* dataset accuracy score and learning rate results under OGLVQ model using balanced dataset.

Experiment 2



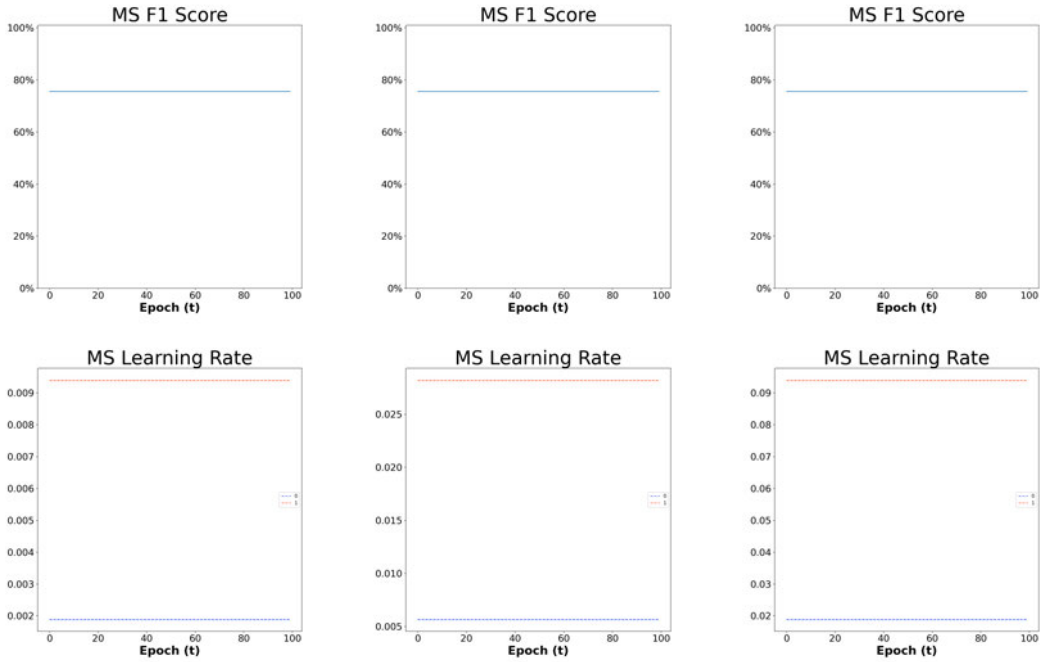
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.21: *NSP* dataset F1 score and learning rate results under CP model using imbalanced dataset.



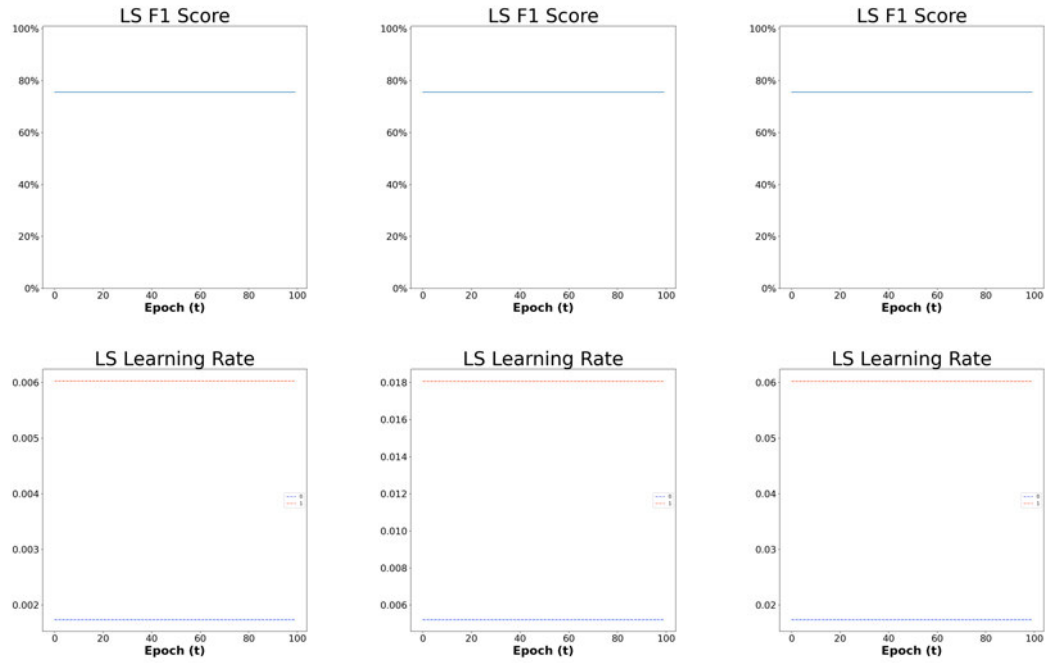
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.22: *NSP* dataset F1 score and learning rate results under DFH model using imbalanced dataset.



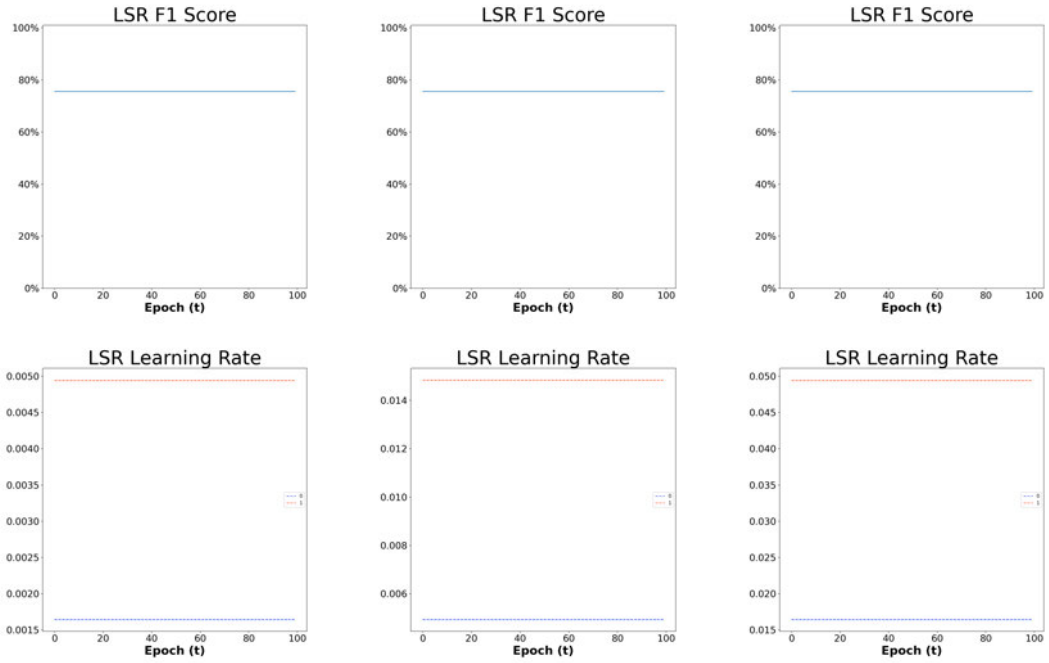
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.23: *NSP* dataset F1 score and learning rate results under MS model using imbalanced dataset.



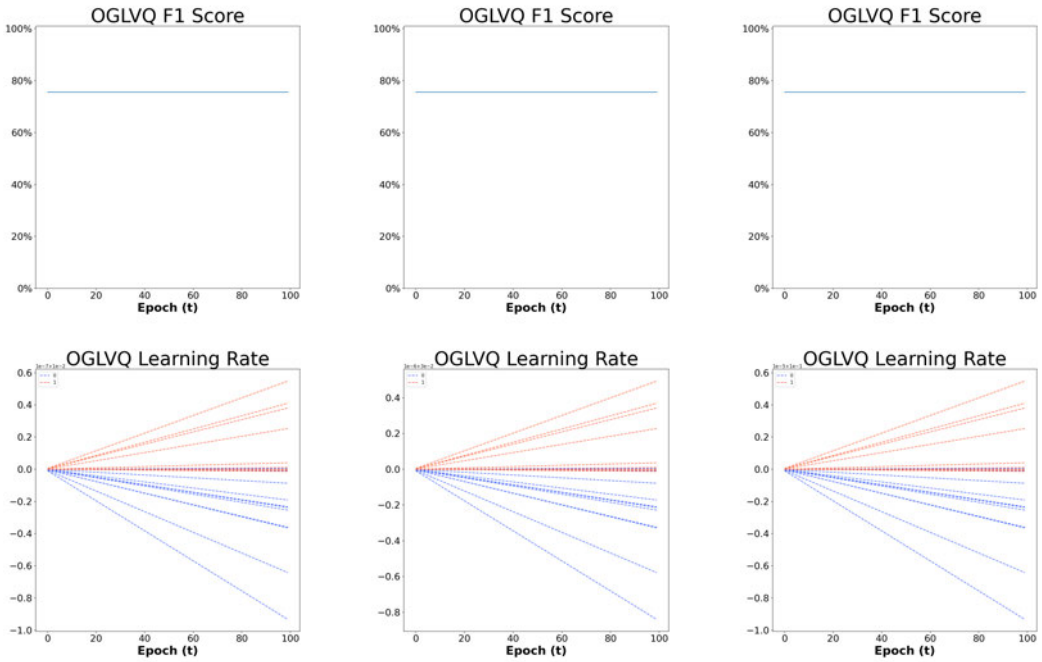
(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.24: *NSP* dataset F1 score and learning rate results under LS model using imbalanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.25: *NSP* dataset F1 score and learning rate results under LSR model using imbalanced dataset.



(a) $\epsilon(0) = 0.01$ (b) $\epsilon(0) = 0.03$ (c) $\epsilon(0) = 0.1$

Figure B.26: *NSP* dataset F1 score and learning rate results under OGLVQ model using imbalanced dataset.

Bibliography

[1] Kohonen, T. (1995). Self-Organizing Maps (pp. 175-189). Springer-Verlag Berlin Heidelberg. https://doi.org/10.1007/978-3-642-97610-0.

[2] Kohonen, T. (1990). Improved versions of learning vector quantization. International Joint Conference on Neural Network. https://doi.org/10.1109/ijcnn.1990.137622.

[3] Takahashi, T., Nakano, M., & Shinohara, S. (2010). Cognitive Symmetry: Illogical but Rational Biases. Symmetry Culture and Science. 21. 1-3. https://www.researchgate.net/publication/285850238_Cognitive_Symmetry_Illogical_but_Rational_Biases.

[4] Shinohara, S., Taguchi, R., Katsurada, K., & Nitta, T. (2007). A Model of Belief Formation Based on Causality and Application to N-armed Bandit Problem. Transactions of the Japanese Society for Artificial Intelligence, 22(1), 58–68. (in Japanese) https://doi.org/10.1527/tjsai.22.58.

[5] Wasserman, E. A., Dorner, W. W., & Kao, S. F. (1990). Contributions of specific cell information to judgments of interevent contingency. Journal of Experimental Psychology: Learning, Memory, and Cognition, 16(3), 509–521. https://doi.org/10.1037/0278-7393.16.3.509.

[6] Taniguchi, H., Sato, H., & Shirakawa, T. (2018). A machine learning model with human cognitive biases capable of learning from small and biased datasets. Scientific Reports, 8(1). https://doi.org/10.1038/s41598-018-25679-z.

[7] Manome, N., Shinohara, S., Takahashi, T., Chen, Y., & Chung, U. (2021). Self-incremental learning vector quantization with human cognitive biases. Scientific Reports, 11(1). https://doi.org/10.1038/s41598-021-83182-4.

[8] Evett, I. W., & Spiehler, E. J. (1989). Rule induction in forensic science. Knowl. Based Syst. 152–160.

[9] Sigillito, V. G., Wing, S. P., Hutton, L. V., & Baker, K. B. (1989). Classification of radar returns from the ionosphere using neural networks. Johns Hopkins APL Technical Digest 10, 262–266.

[10] Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. Ann. Eugenics 7, 179–188.

[11] Gorman, R. P., & Sejnowski, T. J. (1988). Analysis of hidden units in a layered network trained to classify sonar targets. Neural Netw. 1, 75–89.

[12] Bennett, K. P., & Mangasarian, O. L. (1992). Robust linear programming discrimination of two linearly inseparable sets. Optimization Methods and Software, 1(1), 23–34. https://doi.org/10.1080/10556789208805504.

[13] Saruhan, M. (2023). Informational Image Data Preprocessing: IFE Blood Samples [Unpublished manuscript]. Applied Mathematics for Network and Data Sciences, Mittweida University of Applied Sciences.

[14] Zieliński, T. P. (2021). Discrete Fourier Transforms: DtFT and DFT. Springer EBooks, 65–92. https://doi.org/10.1007/978-3-030-49256-4_4.

[15] Google. (2022, July 18). Normalization. Google Developers. https://developers.google.com/machine-learning/data-prep/transform/normalization.

[16] Monoclonal Immunoglobulin (Ig), Monoclonal antibody, Immunofixation Electrophoresis (IFE) - Labpedia.net. (2020, January 25). https://labpedia.net/monoclonal-immunoglobulin-ig-monoclonal-antibody-immunofixation-electrophoresis-ife/

[17] Testing.com. (2021, March 24). Protein Electrophoresis, Immunofixation Electrophoresis. Testing.com. https://www.testing.com/tests/protein-electrophoresis-immunofixation-electrophoresis/.

[18] Leung, N.R. (2016). Chapter 8 : Clinical Tests for Monoclonal Proteins.

[19] Protein electrophoresis. (n.d.). EClinpath. https://eclinpath.com/chemistry/proteins/electrophoretic-patterns/.

[20] Sato, A., & Yamada, K. (1995). Generalized Learning Vector Quantization. Neural Information Processing Systems, 8, 423–429.

[21] Tversky, A., & Kahneman, D. (1974). Judgment under uncertainty: Heuristics and Biases. Science, 185(4157), 1124–1131. https://doi.org/10.1126/science.185.4157.1124.

[22] Hattori, M., & Oaksford, M. (2007). Adaptive Non-Interventional Heuristics for Covariation Detection in Causal Induction: Model Comparison and Rational Analysis. Cognitive Science: A Multidisciplinary Journal, 31(5), 765–814. https://doi.org/10.1080/03640210701530755.

[23] Hattori, M. (2003). Adaptive heuristics of covariation detection: A model of causal induction. https://www.researchgate.net/publication/245197199_Adaptive_heuristics_of_covariation_detection_A_model_of_causal_induction.

[24] Hattori, M. (2001) Ingakinou-no niyouin heuristic model [A dual-factor heuristic model of causal induction], Cognitive Studies: Bulletin of the Japanese Cognitive Science Society, 8, 444-453. https://www.researchgate.net/publication/316814852_A_Dual-Factor_Heuristics_Model_of_Causal_Induction.

[25] Anderson, J. R., & Sheu, C. F. (1995). Causal inferences as perceptual judgements. Memory & cognition, 23(4), 510–524. https://doi.org/10.3758/bf03197251.

[26] Oaksford, M., & Chater, N. (1994). A rational analysis of the selection task as optimal data selection. Psychological Review, 101(4), 608–631. https://doi.org/10.1037/0033-295x.101.4.608.

[27] Kaden, M., Hermann, W., & Villmann, T. (2014). Optimization of General Statistical Accuracy Measures for Classification Based on Learning Vector Quantization. The European Symposium on Artificial Neural Networks. http://www.i6doc.com/fr/livre/?GCOI=28001100432440.