DATA STREAM ANALYSIS

by

Mine Melodi Çalışkan

B.S., Mathematics, Marmara University, 2013

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Department of Computational Science and Engineering

Boğaziçi University

2018

DATA STREAM ANALYSIS

APPROVED BY:

Asst. Prof. Fatih Ecevit .....
(Thesis Supervisor)

Assoc. Prof. Atabey Kaygun .....
(Thesis Co-supervisor)

Prof. Ethem Alpaydın .....

Asst. Prof. Ümit Işlak .....

Assoc. Prof. Kürşat Aker .....

DATE OF APPROVAL: 18.05.2018

ACKNOWLEDGEMENTS

I would like to thank to my thesis supervisor Asst. Prof. Fatih Ecevit for his guidance, great support throughout my study.

I also would like to thank to my thesis co-supervisor Assoc. Prof. Atabey Kaygun for his constant support, availability and constructive suggestions helped me to finish the work presented in this thesis.

I wish to thank the members of my thesis committee: Prof. Ethem Alpaydın, Asst. Prof. Ümit Işlak, and Assoc. Prof. Kürşat Aker for generously offering their time, support, guidance throughout the review of this document.

I would like to express my gratitude to my friends who cherished me with their friendship and support during my dissertation.

Lastly, a special thanks to my parents Şefika Çalışkan, and Orhan Çalışkan for their continuing support, encouragement throughout these years.

ABSTRACT

DATA STREAM ANALYSIS

In this thesis we give a survey of online machine learning algorithms for data stream analysis. After giving an overview of standard batch algorithms, we explain batch-to-online conversion, and we give a in-depth description and analysis of data stream mining techniques. We particularly focus on online k-means algorithms and multilayer perceptron models as representative examples of online clustering and classification algorithms. We also present theoretical and empirical analyses of different approaches for online versions of these algorithms through numerical experiments.

ÖZET

VERİ IRMAĞI ANALİZİ

Bu tezde veri ırmağı analizi için çevrimiçi makine öğrenmesi algoritmalarını araştırdık. Standart çevrimdışı algoritmaları gözden geçirdikten sonra çevrimdışından çevrimiçine dönüştürmeleri açıkladık ve sonra veri ırmağı madenciliği tekniklerinin geniş kapsamlı tanımlamasını ve analizini yaptık. Çevrimiçi gruplama ve sınıflandırma algoritmalarından örnek olarak özellikle çevrimiçi k-ortalama ve çok katmanlı algılayıcı modellerine odaklandık. Sayısal deneylerimizi yaptıktan sonra teorik ve deneysel analizlerimizi farklı bakış açıları kullanarak bu tezde algoritmaların çevrimiçi biçimleri için bu deneyler üzerinden anlattık.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS/ABBREVIATIONS

2D	Two Dimensional
ART	Adaptive Resonance Theory
BP	Back Propagation
DBSCAN	Density Based Spatial Clustering of Applications with Noise
FIFO	First In First Out
LMS	Least Mean Square
LVQ	Learning Vector Quantization
MLP	Multilayer Perceptron
MSE	Mean Squared Error
NP	Non-deterministic Polynomial-time
RBF	Radial Basis Function
ReLU	Rectified Linear Unit
RLS	Recursive Least Squares
SGD	Stochastic Gradient Descent
SOM	Self Organizing Map
SVM	Support Vector Machine
UCI	University of California Irvine

1. INTRODUCTION

1.1. Motivation

Recent advances in technology created an explosion of data volume because relatively simple, or even mundane everyday interactions, such as the use of a credit card or a phone, creates some amount of data per individual. Telecommunications and social networks often are awash in user-generated or system-generated data too. Considering the scale, in aggregates, we have massive collections of data that need to be processed effectively, and in some cases, in real-time if data arrives continually. Such large data collections and continuous data streams present unprecedented challenges, especially in resource-constrained scenarios. Unfortunately, most standard machine learning techniques work within batch learning scenarios in which the algorithm has unfettered access to the whole dataset. Traditional statistical or machine learning techniques do not directly apply to such large collections, or data streams that arrive continually, that need to be processed within tight time and resource constraints.

In situations where data is arriving continuously, and we cannot wait until algorithm collects a static sample of data that faithfully represents the whole data, any static model obtained at a certain time will be outdated sooner or later. So, under such constraints, standard machine learning methods would have to revise their models by re-executing the underlying learning algorithm in batches. But, the cost, both in terms of computation power and available storage, for these algorithms to re-generate their models from scratch with updated data is rather high. Incremental learning algorithms are better suited for such scenarios in order to efficiently assimilate the novel data to the existing model. These algorithms must be able to incorporate new data, adopt their models to the most recent state of the data, and if necessary, forget outdated data.

In this thesis we give a survey on data analysis techniques, problems and algorithms that work particularly well for large data collections, or continuous data streams. We are going to investigate a handful of machine learning algorithms that are suitable for analyzing data under tight time and memory constraints.

1.2. Overview

Most machine learning algorithms are posed as optimization problems, finding optima for specific *cost functions*, and the algorithm needs to optimize this cost function within certain constraints determined by the problem, the data, the processing type and even platform on which the algorithms is run. As such optimization theory forms the bedrock for the whole theory of data analysis and machine learning algorithms. So, we start with Chapter 2 where introduce the general theory of optimization and its relation with machine learning problems. We then discuss different optimization techniques in the same chapter.

In Chapter 3, we introduce commonly used machine learning algorithms, but without any resource constraints. We investigate these methods from the perspective of optimization delving into the process of designing the cost function associated with these machine learning problems in detail.

We discuss the main challenges of streaming data analysis in terms of solving basic machine learning problems such as clustering, classification, regression and pattern mining under different scenarios with tight memory and time constraints in Chapter 4. We investigate using different optimization methods such as the stochastic gradient descent in order to re-implement standard machine learning algorithms to process large volumes of data that can not fit within the memory, or data that arrives continuously. We also investigate online and iterative versions of perceptron models and competitive neural networks.

In Chapter 5, we focus on facility location problem and general clustering problems based on the k-means algorithm for streaming data. We begin with Lloyd's batch k-means algorithm, and derive an online version of the k-means algorithm using the stochastic gradient descent method to solve the resulting optimization problem. We also introduce a version of the competitive neural network algorithm called ART which can also be used as an online clustering method. Then, we discuss semi-online and fully-online k-means algorithms which are recently introduced by [6]. All of these architectures had a lot in common, but some crucial distinctions.

We develop the standard backpropagation algorithm and then, introduce two different versions of online backpropagation using stochastic gradient descent and recursive least squares in Chapter 6. These algorithm use backpropagation to train a neural network, and are similar algorithmically. The main difference between these methods arise from their respective cost functions.

We analyze our numerical experiments in Section 7.1 and Section 7.2 for online k-means and online multilayer perceptron algorithms respectively in Chapter 7. We discuss the results of our experiments in Chapter 8. The code we used in our experiments are listed in the Appendix.

2. OPTIMIZATION

In this chapter, we are going to investigate optimization and its connections with various machine learning algorithms. Optimization refers to the task of either minimizing or maximizing an objective function that expresses the model at hand. We can formulate any optimization problem as follows:

Given an objective function $f : \mathbb{X} \to \mathbb{R}$, where $\mathbb{X}$ is our search space, find a subset $\mathbb{X}^*$ of $\mathbb{X}$ such that:

$$\mathbb{X}^* = \{x^* \mid f(x^*) \leq f(x), \text{ for all } x \in \mathbb{X}\} \tag{2.1}$$

In constraint optimization, we want our solutions to satisfy some properties expressed by equations or inequalities, which restricts our search space $\mathbb{X}$ to $\mathbb{S} \subset \mathbb{X}$. The set $\mathbb{S}$ can be defined in terms of functions that correspond to *equality constraints* and *inequality constraints*. In this case, points x of $\mathbb{S}$ are called *feasible* points. In general, one has the following type restrictions:

$$\mathbb{S} = \{x \mid g^{(i)}(x) = 0 \text{ and } h^{(j)}(x) \leq 0, \text{ for } i \in I, \text{ and for } j \in J\} \tag{2.2}$$

where $\{g^{(i)}\}_{i \in I}$ and $\{h^{(j)}\}_{j \in J}$ are two finite sets of constraint functions. Then the optimization problem reduces to find a subset $\mathbb{X}^*$ of $\mathbb{X}$ such that:

$$\mathbb{X}^* = \{x^* \in \mathbb{S} \mid f(x^*) \leq f(x), x \in \mathbb{X}\} \tag{2.3}$$

There are two main categories of algorithms for solving optimization problem:

(i) *First order optimization algorithms* use the *gradient* of the cost function with respect to its parameters. The gradient gives the information whether the function is increasing, or decreasing at a particular point.

(ii) *Second order optimization algorithms* use the *Hessian* of the cost function to estimate the curvature of the objective function together with the information on whether the first derivative is increasing or decreasing.

If an optimization algorithm uses the entire data set, then it is called *a batch algorithm*, or *a deterministic algorithm* because the algorithms processes all of the data reserved for building the model in a large batch. In contrast, an optimization algorithm that processes a sample of data points at a time is usually referred as *a stochastic algorithm*, or as *an online algorithm*. However, the term *online* is usually reserved for algorithms that use a continuously arriving stream of data points instead of a fixed set of data points on which the algorithm has random access. These concepts are discussed in detail in Chapter 4.

2.1. Optimization for Machine Learning Models

The traditional optimization algorithms differ from the optimization algorithms used for training machine learning models in several ways. In pure optimization problems, the data-generating probability distribution p_{data} is usually known. In contrast, in most machine learning tasks we only have data samples and our objective is to minimize the expected generalization error which is known as *the risk*. One can reformulate a machine learning task as an optimization problem in which we aim to minimize the expected loss function on the training set, which is called the *empirical risk*, expecting to minimize the risk as well.

If we assume that we sampled the stream *uniformly and independently* then we can expect that the training set and the test set share the same underlying data-generating distribution. With this assumption the expected training error of a randomly selected model is going to be equal to the expected test error of that model.

However, we can only take samples from the training set in order to choose the parameters that reduce the error on the training set. We then sample the test data set and measure the error without updating any parameters. Thus the expected error on the test set is always going to be greater than or equal to the expected error on the training set.

2.2. Expected Risk Function

A learning system is designed to minimize a function $J(w)$ called *the expected risk function* which is expressed as follows:

$$J(w) = \mathbb{E}[d(f(x; w), y)] = \mathbb{E}[d(\hat{y}, y)] \tag{2.4}$$

where d is a non negative function, called the *loss function*. The loss function measures the discrepancy between the real value y and its estimate $f(x; w) = \hat{y}$ when x is the input, and w is the parameter that must be adapted in the learning process by observing events y occurring in the real world.

2.3. Empirical Risk Minimization

We emphasize in Equation (2.4) that the expected is calculated over the true underlying distribution p_{data} such that:

$$J(w) = \mathbb{E}[d(f(x; w), y)] \tag{2.5}$$

In machine learning, we cannot minimize the expected risk function directly since the data generating distribution is unknown. Instead, we minimize the empirical risk in the training process and expect the true risk is going to decrease as well. So, the training process corresponds to empirical risk minimization:

$$\hat{J}_m(w) = \frac{1}{m} \sum_{i=1}^{m} d(f(x_i, w), y_i) \tag{2.6}$$

The expected value of the empirical risk $\hat{J}_m(w)$ is calculated via the empirical distribution $\hat{p}(x, y)$ given by the training set, and m is the number of training examples.

Optimizing the empirical risk function $\hat{J}_m(w)$ also yields a good estimate for the minimum of the expected risk function $J(w)$ if the training set is adequately large [7]. However, empirical risk minimization may cause *overfitting*. That is, models may simply memorize the training set. This area of research is related with *generalization phenomenon*. *Generalization* can be described as the capacity of a system to learn from a set of examples from a finite training set, and yet provide results that are valid in general.

Below, we present gradient based methods for batch and online processing.

2.4. Gradient Based Learning

Gradient based learning algorithms are *first order* optimization techniques which only require first-order derivatives of the terms of the parameters w in the objective function J. A commonly used variation is the *gradient descent* method. The gradient descent is an iterative optimization algorithm thats looks for the minimum of an objective function by following the opposite direction of the gradient.

Here, we consider two different versions of the gradient descent algorithms. These algorithms differ in how much data they consume in estimating the gradient of the objective function. There creates a trade-off between the time it takes to perform an update depending on the amount of data available to the algorithm in constructing a model, and the accuracy of the parameter update.

2.4.1. Batch Gradient Descent

In the batch gradient descent method, the algorithm computes the gradient of the objective function with respect to the parameters on the entire training dataset:

$$w = w - \eta \nabla_w J(w) \tag{2.7}$$

For a convex cost function, the batch gradient descent algorithm converges to the global minimum. Thus convexity of the cost function is desired since the algorithm may converge to one of the local extrema points otherwise.

One can minimize the empirical risk function $\hat{J}_m(w)$ using a batch gradient descent algorithm. Successive estimates w_t are computed as follows:

$$w(t+1) = w(t) - \eta_t \nabla_w \hat{J}_m(w(t)) = w(t) - \eta_t \frac{1}{m} \sum_{i=1}^{m} \nabla_w d(f(x_i; w), y_i)$$

Here η_t determines how big the updates are, and it is called the *learning rate.* When the learning rate η_t is small enough, the algorithm converges to a local minimum of the empirical risk function $\hat{J}_m(w)$. We give the pseudocode of the batch gradient descent algorithm below in Figure 2.1.

Initialize W
repeat
  for all (x_i, y_i) **do**
    $\nabla_w \hat{J}(w) \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_w d(f(x_i; w), y_i)$
    $w(t+1) \leftarrow w(t) - \eta \nabla_w \hat{J}(w)$
  end for
until w_i's converge

Figure 2.1. Batch gradient descent algorithm.

In the batch gradient descent, one needs to calculate the gradient for the whole dataset. This process can be slow if the training set is large, or even intractable for datasets that do not fit in memory. The batch gradient descent also does not allow us to update our models online.

2.4.2. Stochastic Gradient Descent

The second variation of gradient descend algorithms we are going to consider in depth in this thesis is called the *stochastic gradient descent* (SGD) method. The SGD algorithm does an update on parameters for each data point (x_i, y_i) in the training data set:

$$w = w - \eta \nabla_w J(w; x_i, y_i) \tag{2.8}$$

Since the stochastic gradient descent method works with one data point, or sometimes a mini-batch of data points, at a time, it is much less memory intensive. We give the pseudocode of stochastic gradient descent algorithm in Figure 2.3. In each iteration, the online gradient descent chooses an example x uniformly randomly, and then updates the parameter w using the following formula:

$$w(t+1) = w(t) - \eta_t \nabla_w d(f(x_i; w), y_i)$$

If we average the updates over all possible choices taken from the set of training examples, we would obtain the batch gradient descent algorithm.

The term *stochastic* comes from the fact that calculating the gradient on a single training sample is a *stochastic approximation* of the gradient of the *true* cost function.

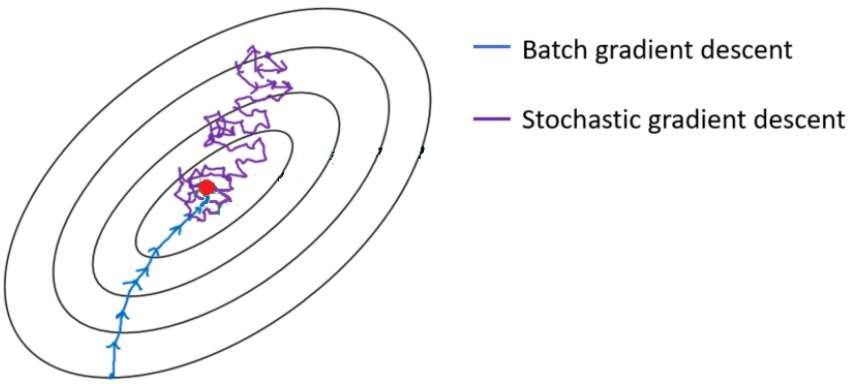

Figure 2.2. Due to its stochastic nature, the path towards the global cost minimum might be not *direct* as in batch gradient descent, but may make a *zig-zag* if we visualize the cost surface in a 2D space.

Initialize $W \leftarrow$ to small numbers

Learning rate schedule $\eta_1, \eta_2, \ldots$

repeat

  for (x_i, y_i) in random order **do**

    $\nabla_w \hat{J}(w) \leftarrow d(f(x_i; w), y_i)$

    $w \leftarrow w - \eta \nabla_w J(w)$

  end for

until w_i's converge

Figure 2.3. Stochastic gradient descent algorithm.

2.4.2.1. Convergence. Batch gradient descent needs to make one full pass over the data until an update is performed. This means each update has the time complexity of $O(nd)$, and also the computations are prone to be repeated on redundant data points. Since SGD performs one update at a time, it can be used to for online learning algorithms, and also it is usually much faster since the complexity is $O(d)$.

Batch gradient descent is shown [8] to converge to a local or the global minimum for non-convex and convex optimization functions, respectively. In stochastic gradient descent, random sampling of m training examples introduces a source of noise which causes fluctuations. This behavior of SGD enables to jump to a better local minimum, however, a non-vanishing gradient may cause to skip the global minimum. Therefore, the batch gradient descent algorithms can use a fixed learning rate effectively. However, Robbins and Monro [9] showed that when we gradually decrease the learning rate, the SGD algorithm converges as fast as the batch gradient descent for stationary data streams.

3. MACHINE LEARNING AS OPTIMIZATION

We continue with basic algorithms for unsupervised and supervised learning including linear and logistic regression, regression trees, and support vector machines.

3.1. Linear Regression

In linear regression problems, given the input data X we want to find the parameters that predict the output Y in a linear fashion. In this model, the output is a linear combination of the features. In other words,

$$y_i = f(x_i, w) = w^T \cdot x_i \tag{3.1}$$

Then we need to find the model parameters w that provides the best fit for the training data set. The best fitting model is measured using the mean squared error (MSE) over the training dataset:

$$J(w) = \frac{1}{|\mathbb{X}|} \sum_{i=1}^{|\mathbb{X}|} (f(x_i, w) - y_i)^2 \tag{3.2}$$

In order to find the best fitting line, one needs to minimize the MSE function $J(w)$. One can derive a closed-form solution, but one can also use the gradient descent to minimize $J(w)$.

3.1.1. Least-Squares Estimation

We can estimate the solution of least squares by minimizing the sum of squared errors over the whole data set. This leads to a closed-form expression for the estimated value of the unknown parameter w.

$$\begin{aligned} J(w) &= \frac{1}{|\mathbb{X}|}\sum_{i=1}\epsilon_i^2 \\ &= \frac{1}{|\mathbb{X}|}\sum_{i=1}(y_i - w_i^T x_i)^2 \\ &= \frac{1}{|\mathbb{X}|}\|y - XW\|_2^2 \\ &= \frac{1}{|\mathbb{X}|}(y - XW)^T(y - XW) \end{aligned}$$

The minimum of $J(w)$ is obtained by setting the derivatives of $J(w)$ equal to zero.

$$\begin{aligned} \nabla_w J(w) = 0 \Rightarrow \nabla_w \frac{1}{|\mathbb{X}|}\|y - XW\|_2^2 = 0 \\ \Rightarrow \nabla_w (y - XW)^T(y - XW) = 0 \end{aligned}$$

$$\begin{aligned} \nabla_w (y - XW)^T(y - XW) &= (y^T y - y^T XW - W^T X^T y + W^T X^T XW) \\ &= -2X^T y + 2X^T Xw \\ &= 0 \end{aligned}$$

This setting gives the normal equations as:

$$X^T Xw = X^T y$$

Solving for w, we obtain parameters estimation formula:

$$w = (X^T X)^{-1} X^T y$$

provided that the inverse of the matrix $X^T X$ exists. This happens when the matrix X has rank k. Since X is an $n \times k$ matrix, we must have $n \geq k$ which means the number of observations must be greater than or equal to the number of parameters.

3.2. Logistic Regression

Linear regression is used to predict continuous valued outputs as a linear function of continuous input while the logistic regression is used for predicting binary-valued output values again on continuous input.

In logistic regression, for a given collection of data points $\{x_i, y_i\}_{i=1}^{N}$ where each $x_i \in \mathbb{R}^d$ and $y_i \in \{0, 1\}$ we need to define a *linear classifier*. The *logistic regression* algorithm defines an objective function using a *likelihood* function on this data set. Then it optimizes this objective function using the gradient descent in order to to find the optimum parameters that yields highest probability for this data set.

Let us assume that the conditional probabilities are given by

$$P(y = 1 \mid x; w) = h_w(x) = \frac{1}{1 + e^{-w^T \cdot x}} = \sigma(w^T \cdot x) \tag{3.3}$$

$$P(y = 0 \mid x; w) = 1 - P(y = 1 \mid x; w) = 1 - h_w(x) \tag{3.4}$$

where

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

is called the *logistic function* or the *sigmoid function*. The logistic function maps the value of the dot product $w^T \cdot x$ into the interval $[0, 1]$ so that we may interpret $h_w(x)$ as a probability. Combining these probabilities we get:

$$P(y \mid x; w) = h_w(x)^y (1 - h_w(x))^{1-y}$$

Assuming the training set was sampled independently, we write the likelihood function as:

$$\begin{aligned} L(w) &= p(y \mid X; w) \\ &= \prod_{i=1}^{m} p(y_i \mid x_i; w) \\ &= \prod_{i=1}^{m} (h_w(x_i))^{y_i} (1 - h_w(x_i))^{1-y_i} \end{aligned}$$

In order to simplify the process, we can maximize the logarithm of the likelihood function instead of the likelihood function:

$$\begin{aligned} \ell(w) &= \log(L(w)) \\ &= \sum_{i=1}^{m} y_i \log(h(x_i) + (1 - y_i) \log(1 - h(x_i) \end{aligned}$$

Similar to linear regression, we can use gradient descent to maximize the likelihood. Note that $\sigma'(z) = \sigma(z)(1 - \sigma(z))$. In order to optimize this function we need the derivatives

$$\begin{aligned} \frac{\partial \ell(w)}{\partial w_j} &= \sum_{i=1}^{m} \left(\frac{y_i}{h_w(w^T x_i)} - \frac{1 - y_i}{1 - h_w(w^T x_i)} \right) \frac{\partial h_w(w^T x_i)}{\partial w_j} \\ &= \sum_{i=1}^{m} \left(\frac{y_i}{h_w(w^T x_i)} - \frac{1 - y_i}{1 - h_w(w^T x_i)} \right) h_w(w^T x_i)(1 - h_w(w^T x_i)) \frac{\partial w^T x_i}{\partial w_j} \\ &= \sum_{i=1}^{m} \big(y_i (1 - h_w(w^T x_i)) - (1 - y_i) h_w(w^T x_i) \big) x_{ij} \\ &= \sum_{i=1}^{m} (y_i - h_w(x)) x_{ij} \end{aligned}$$

Then the update rule for *maximizing* the log likelihood becomes

$$w_j = w_j + \eta \sum_{i=1}^{m} (y_i - h_w(x_i)) x_{ij} \tag{3.5}$$

3.3. Regression Trees

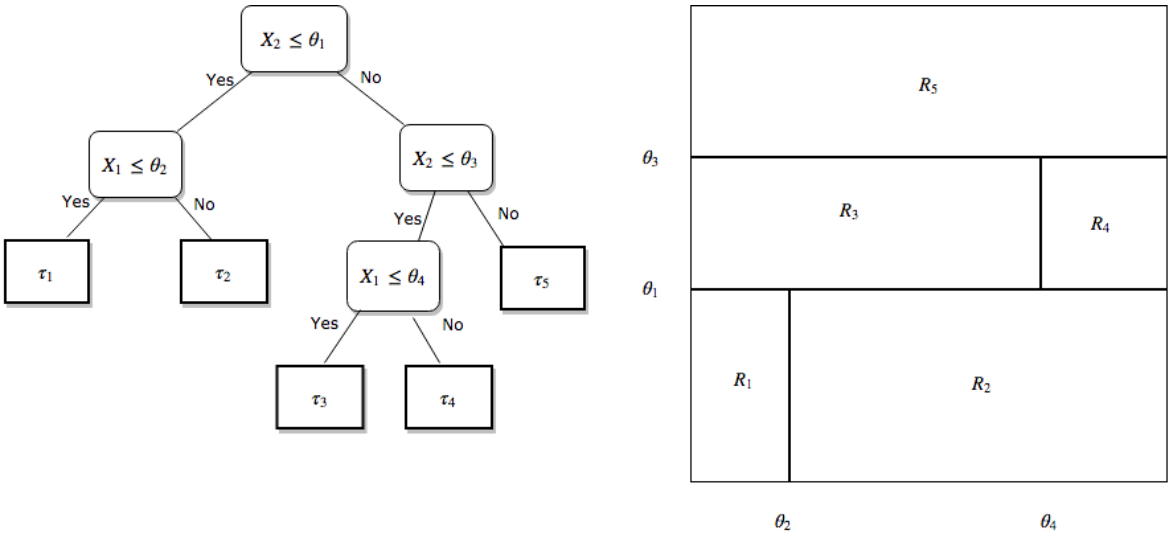

Figure 3.1. Tree model: Partition the feature space into a set of rectangles.

In regression trees, we create partitions of the feature space into a set of rectangles. Then we fit a simple model, e.g. a constant, in each rectangle. Figure 3.1 shows interpretation of these partitions.

Given input $X \in \mathbb{R}^{n\times p}$ and corresponding outputs $y \in \mathbb{R}^n$. Each observation corresponds to $(y_i, x_i) \in \mathbb{R}^{p+1}$, $i = 1, \ldots, n$. Suppose we have a partition of $\mathbb{R}^p$ into M regions $R_1, \ldots, R_m$. We predict the response using a constant on each R_i:

$$f(x) = \sum_{i=1}^{m} c_i . \chi_{R_i}(x) \tag{3.6}$$

where χ_U is the *characteristic function*, which is also known as the *indicator function*, of a subset $U \subseteq \mathbb{R}^p$. In order to minimize $\sum_{i=1}^{n}(y_i - f(x_i))^2$, one needs to choose:

$$\hat{c}_i = avg(y_j : x_j \in R_i) \tag{3.7}$$

3.3.1. Splitting and Pruning Criteria

For continuous variables, we picked a constant in each box R_i to minimize the sum of squares in that region:

$$\min_{c\in\mathbb{R}} \sum_i \sum_{x_k\in R_i} (y_k - c_i)^2 \tag{3.8}$$

As a result, we choose:

$$\hat{c}_i = \frac{1}{N_i} \sum_{x_k\in R_i} y_k \tag{3.9}$$

where N_i denotes the number of observations in R_i.

In order to determine the regions R_i, we need to decide on variable we are going to split, and where to split that variable. Unfortunately, finding a globally optimal tree is computationally infeasible. So, we use a greedy algorithm as follows:

(i) Take a splitting variable $j \in \{1, \ldots, p\}$, and consider a splitting point $s \in \mathbb{R}$.

(ii) Define the two half-planes:

$$R_1(j,s) := \{x \in \mathbb{R}^p : x_j \leq s\}, \quad R_2(j,s) := \{x \in \mathbb{R}^p : x_j > s\} \tag{3.10}$$

(iii) We choose j, s to minimize:

$$\min \left[\min_{c_1\in\mathbb{R}} \sum_{x_i\in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2\in\mathbb{R}} \sum_{x_i\in R_2(j,s)} (y_i - c_2)^2 \right] \tag{3.11}$$

One can find the splitting point s very quickly, and therefore, determining the best pair (j, s) is feasible. Then we repeat the same process for each block recursively.

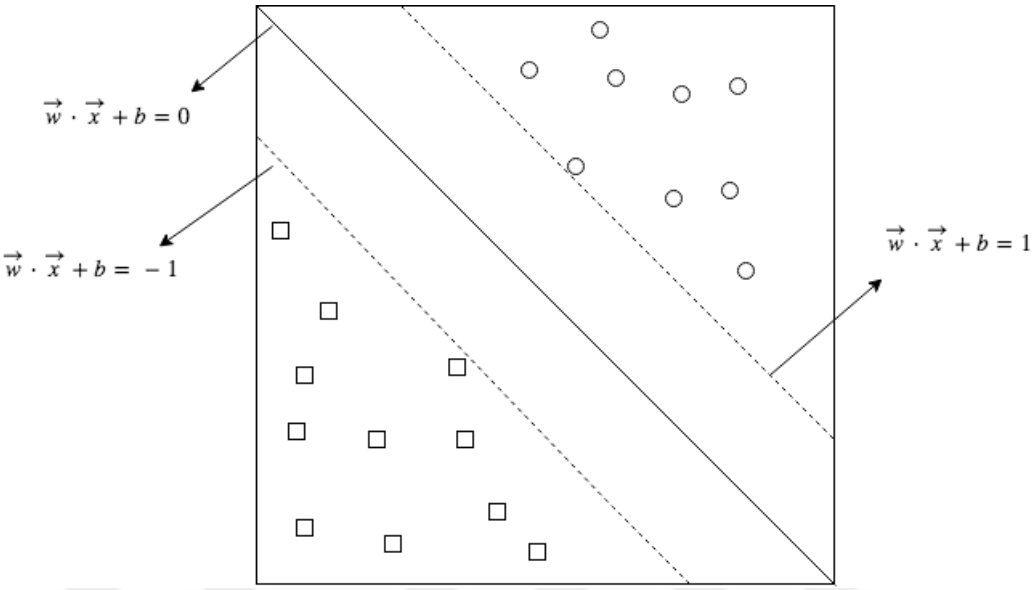

Figure 3.2. Support vector machine.

3.4. Support Vector Machines

Support vector machine is a supervised classification algorithm that aims to find a suitable collection of hyperplanes that separate the data into a finite number of classes. Given a labeled training dataset $\{(x_i, y_i)\}$, the algorithm outputs an optimal arrangement of hyperplanes to categorize new examples.

In our model, separating hyper-planes are given by $y = w^T x + b$ where y corresponds to classification label, w is a normal to the plane, and b which moves the hyperplane from origin. On the decision boundary we have $w^T \cdot x + b = 0$. Let us define

$$y_i = \begin{cases} 1 & \text{if } w \cdot x_i + b \geq 0 \\ -1 & \text{if } w \cdot x_i + b < 0 \end{cases}$$

The distance of the i^{th} sample (x_i, y_i) to the hyper-plane is given as:

$$d_i = \frac{\|w \cdot x_i + b\|}{\|w\|} \tag{3.12}$$

We are looking for data points, called *support vectors*, for which this distance is minimal, for these data points, we can add the constraint $w \cdot x_i + b = 1$. Then the total distance, i.e. the *margin* M, is equal to $\frac{2}{\|w\|}$.

The problem of finding a classifier with the maximum *margin*, i.e. maximal total distance of the *support vectors*, can be formulated as an optimization problem:

$$\underset{i}{\operatorname{argmax}}\, 2d_i = \underset{w,b}{\operatorname{argmax}}\, \frac{2}{\|w\|} = \underset{w,b}{\operatorname{argmin}}\, \|w\| \tag{3.13}$$

Therefore, maximizing $\frac{2}{\|w\|}$, is equivalent to minimizing $\|w\|$. To turn the problem into *quadratic programming* we can also minimize $\|w\|^2$. Hence, the original problem 3.13 is equivalent to finding

$$\operatorname{argmin} \|w\|^2 \quad \text{subject to} \quad y_i(w \cdot x_i + b) \geq 1.$$

Now, we have a constrained convex optimization problem in which we have a convex quadratic objective function with linear constraints. The solution gives us the *optimal margin classifier.*

4. DATA STREAM MINING

In this chapter, we are going to investigate data analysis techniques over data streams.

4.1. Introduction

A data stream $\mathbb{S}$ is an ordered sequence of data objects $\mathbb{S} = \{x_i\}_{i=1}^{N}$. Each data object is usually described by a finite dimensional vector $x_i = [x_{ij}]_{j=1}^{d}$. The components of these vectors called *features* that belongs to an attribute space Ω. Attributes can either be continuous, categorical, or mixed.

Most machine learning algorithms assume that there is a finite sequence of data points generated by a fixed unknown probability distribution. In *batch learning algorithms*, the algorithm passes over a static dataset, called the *training dataset*, at least once. Then the model generated by the algorithm from this training set is used for making predictions on a different dataset, called *the test dataset*. In this approach, we assume the data available for the algorithm has stationary statistical properties that do not change over time. Since the data is assumed to be stationary, one can store it and analyze it in multiple steps.

For streaming data analysis data objects arrive continuously, and the unknown distribution that generates the data might also be non-stationary. The changes in probability distribution over time is also referred as the *concept drift* [10]. There is also the possibility that we have no control on the order in which the data arrives and is processed, or even the possibility that data objects need to discarded after they are processed due to tight memory constraints.

The most common data streaming data analysis tasks are regression, clustering, classification and frequent pattern mining. We particularly focus on online methods. Roughly speaking, online methods are methods that "process one datum at a time."

We give a brief introduction to these data stream analysis tasks below. But before we embark on this task, we are going to investigate the differences between *batch learning, streaming data analysis,* and their variations.

4.2. Batch, Online and Incremental Learning

4.2.1. Batch vs. Stream Processing

Table 4.1. Differences between batch and stream data processing [5].

	Batch	Stream
Number of passes	Multiple	Single
Processing time	Unlimited	Restricted
Memory usage	Unlimited	Restricted
Type of result	Accurate	Approximate
Distributed	No	Yes

Machine learning algorithms that use the whole training dataset are usually called *deterministic* or *batch* methods, because they process every point in the training example at least once over a single pass. On the other hand, machine learning algorithms that use only a single data point at a time are usually called *stochastic*, *online* or *incremental*. The term "online" refers to the cases where examples are taken from a data stream of continually arriving data points rather than from a static training set over which the algorithm may perform several passes.

An online learning system does not rely on stationarity of the data, and makes no distinction between the training and test datasets. An online learning model continuously learn by using the most recent information available since every data point updates the model at hand with new information. As a result, if the statistical properties of the data change with new examples, the model easily adapts itself.

4.2.2. Online vs. Incremental Processing

In an online learning scenario, we only know the features of the new data point but without labels. Then the system reacts and suffer a loss or a penalty, and use the loss to update our model. Our model here may be a set of different models so that one can update them or remove the bad ones. In an incremental learning scenario, we receive new data points with labels, and the system updates our already-trained model without using the entire dataset.

Online learning may also refer to the case where each example is used only once while incremental methods usually sample one example at a time from a static dataset and may process the same examples multiple times. That said, the distinction between incremental and online is not always clear. In fact, the same method can often be applied to both situations. An example is stochastic gradient descent: in the online case, each new example is a sample from the data distribution, while in the incremental case typically one picks a uniformly random example from the dataset. Online methods typically require recursive techniques that update faster than the sampling rate. Also, while the updates are similar the objective functions we need to optimize are different: we calculate an expectation over the data distribution for the former, and an expectation over the empirical distribution of the dataset for the latter.

Some of the oldest algorithms in machine learning are online. This comes from the fact that many machine learning algorithms use stochastic optimization techniques such as online backpropagation. There are also such online extensions of support vector machines (SVM) [11]. All prototype-based models such as radial basis function networks (RBF), self-organizing maps (SOM) and supervised learning vector quantization (LVQ) are online learning algorithms, since they use stochastic optimization techniques [12–15].

4.2.3. Stochastic Descent

Many machine learning algorithms are either online by design, or can be made online even though the standard implementations may not support it originally.
For example, decision trees has online variations [16]. The support vector machines (SVM) method was first described [17] with traditional batch processing optimization techniques. The stochastic gradient descent for perceptrons [18], and for the Adaline method [19] are also examples. Table 4.2 illustrates tha way the stochastic gradient descent is used in some classic machine learning algorithms [20]. There are also algorithms which the standard implementation already supports online learning. For example in *nearest neighbor clustering* [21] one simply adds a new instance to the set, and from then on it is available to match as a nearest neighbor.

Table 4.2. Stochastic gradient algorithms for various learning systems.

Model	Loss	Stochastic gradient algorithm
Perceptron	$J_{perceptron} = max\{0, -yw^T x\}$ $y = \mp 1$	$w \leftarrow w + \eta y_t x_t$ if $y_t w^T x_t \leq 0$
k-means	$J_{k-means} = min_k \frac{1}{2}(x - w_k)^2$	$k^* = \text{argmin}_k (x_t - w_k)^2$ $w_{k^*} \leftarrow w_{k^*} + \eta(x_t - w_{k^*})$
SVM	$J_{svm} = \lambda w^2 + \max\{0, 1 - yw^T x\}$ $y = \mp 1$	$w \leftarrow w - \eta_t \lambda w$ if $y_t w^T x_t > 1$ $w \leftarrow w - \eta_t(\lambda w - y_t x_t)$ otherwise

For many machine learning schemes, *stochastic gradient descent* algorithm is useful for processing data because it allows online processing. Traditional gradient descent methods, such as the conjugate gradient descent, compute the gradient of the entire dataset first, and then search the optimal learning rate along that gradient. Such searches are expensive since they need to calculate likelihoods over the entire dataset in order to update the parameters along the gradient. Each global gradient is unlikely to point at the solution and the search curve resembles a *zig-zag* because each next update direction must be orthogonal the previous update. On the other hand, stochastic search methods, such as the stochastic gradient descent, move in the direction of the contribution of a single training pattern and a scalar multiple of the prior gradient.

With such incremental updates the algorithm searches the optimum more directly in the direction of the global minimum by making small adjustments with each training instance.

Stochastic approaches are commonly used with online machine learning algorithms because they store only the parameters and a single training example, but they need to make multiple passes either over the same data more than once, or over fresh data. With a large volume of data, it is typically better to use an algorithm that visits each data element once or a few times rather than sampling a small batch and feed it to a batch algorithm. Hybrid algorithms that sample batches of larger than one but smaller than the entire dataset are also frequently used, and called *mini-batch methods*.

4.3. Regression on Data Streams

In many machine learning applications, one often formulates the parameter estimation problem as a linear regression problem. Online or sequential, recursive estimation of such parameters can be done via the stochastic gradient descent, or the recursive least squares (RLS) algorithm.

4.3.1. Recursive Least Squares Estimation

Suppose that at a given time we have a collection of input vectors $\mathbb{X}_{(n)}$, and corresponding output vectors $\mathbb{Y}_{(n)}$. Recall that in linear regression problem we want to find parameters W such that:

$$y_i = W^T \cdot x_i \tag{4.1}$$

Suppose that we have n observations on the m input variables. Let us write our data set as a matrix.

$$\mathbb{X}_{(n)} = \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_n^T \end{bmatrix} = \begin{bmatrix} x_{11} & \cdots & x_{1m} \\ x_{21} & \cdots & x_{2m} \\ \vdots & & \\ x_{n1} & \cdots & x_{nm} \end{bmatrix} \qquad \mathbb{Y}_{(n)} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

Then the least squares solution for the optimization problem is

$$\hat{W}(n) = (\mathbb{X}_{(n)}^T \mathbb{X}_{(n)})^{-1} \mathbb{X}_{(n)}^T \mathbb{Y}_{(n)}$$

Here $\hat{W}(n)$ corresponds to estimation when we have n number of data pairs. When a new data point (x_{n+1}, y_{n+1}) arrives, we must increase n by 1. This requires that we need to calculate $\hat{W}(n+1)$ from scratch by inverting large matrices, but recalculating an inverse matrix is expensive both in terms of time and storage. So, we are going to develop an iterative method below utilizing the expansion calculation we performed for already arrived data points $\mathbb{X}_{(n)}$ and $\mathbb{Y}_{(n)}$.

Let us define $P_{n+1}^{-1} = \mathbb{X}_{(n+1)}^T \mathbb{X}_{(n+1)}$.

$$P_{n+1}^{-1} = \sum_{i=1}^{n+1} x_i x_i^T = \sum_{i=1}^{n} x_i x_i^T + x_{n+1} x_{n+1}^T \tag{4.2}$$

$$= P_n^{-1} + x_{n+1} x_{n+1}^T \tag{4.3}$$

As for the newly arriving data, we get

$$\mathbb{X}_{(n+1)}^T \mathbb{Y}_{(n+1)} = \sum_{i=1}^{n+1} x_i y_i = \sum_{i=1}^{n} x_i y_i + x_{n+1} y_{n+1} \tag{4.4}$$

$$= \mathbb{X}_{(n)}^T \mathbb{Y}_{(n)} + x_{n+1} y_{n+1}. \tag{4.5}$$

Then our calculation of $\hat{W}(n+1)$ yields

$$\hat{W}(n+1) = P_{n+1}\mathbb{X}_{(n+1)}^T\mathbb{Y}_{(n+1)} \tag{4.6}$$

We can derive recursive formula for $\hat{W}(n+1)$ using $\hat{W}(n)$

$$\begin{aligned}
\hat{W}(n+1) &= P_{n+1}(\mathbb{X}_{(n+1)}\mathbb{Y}_{(n+1)}^T) \\
&= P_{n+1}(\mathbb{X}_{(n)}\mathbb{Y}_{(n)}^T + x_{n+1}y_{n+1}) \quad \text{(Equation (4.5))} \\
&= P_{n+1}(P_n^{-1}\hat{W}(n) + x_{n+1}y_{n+1}) \quad \text{(Equation (4.6))} \\
&= P_{n+1}((P_{n+1}^{-1} - x_{n+1}x_{n+1}^T)\hat{W}(n) + x_{n+1}y_{n+1}) \quad \text{(Equation (4.3))} \\
&= P_{n+1}(P_{n+1}^{-1}\hat{W}(n) - x_{n+1}x_{n+1}^T\hat{W}(n) + x_{n+1}y_{n+1}) \\
&= \hat{W}(n) + P_{n+1}x_{n+1}y_{n+1} - P_{n+1}x_{n+1}x_{n+1}^T\hat{W}(n) \\
&= \hat{W}(n) + P_{n+1}x_{n+1}(y_{n+1} - x_{n+1}^T\hat{W}(n)) \\
&= \hat{W}(n) + K_{n+1}\epsilon_{n+1}
\end{aligned}$$

where

$$K_{n+1} = P_{n+1}x_{n+1}, \text{ and } \epsilon_{n+1} = y_{n+1} - x_{n+1}^T\hat{W}(n) \tag{4.7}$$

Here K corresponds to *gain vector* at each weight update, similar to *learning rate*. Now we take attention to recursive formula for P where $P_{n+1} = \left(P_n^{-1} + x_{n+1}x_{n+1}^T\right)^{-1}$. We will use Woodburry's [22] matrix inversion formula:

If A, C, BCD are non-singular square matrix (the inverse exists) then

$$[A + BCD]^{-1} = A^{-1} - A^{-1}B[C^{-1} + DA^{-1}B]^{-1}DA^{-1} \tag{4.8}$$

Now, in Equation (4.8) we identify A, B, C, D as follows:

$$A = P_{n+1}^{-1} \quad B = x_{n+1} \quad C = 1 \quad D = x_{n+1}^T \tag{4.9}$$

So that, we have:

$$P_{n+1} = P_n + P_{n+1}x_{n+1}[I + x_{n+1}^T P_{n+1} x_{n+1}]^{-1} x_{n+1}^T P_n \tag{4.10}$$

Here the term which its inverse is calculated is scalar since P_n is $m \times m$ and X^T is $1 \times m$ and X is $m \times 1$ matrices.

Error term decides the magnitude of change, and it is scalar. K decides the direction of the change should be made, because W is a vector.

The pseudocode of this procedure is given in Figure 4.1 where λ is a parameter called forgetting factor.

Initialize $w_0 = 0$, $P_0 = \sigma I$
repeat
  for $t = 1 \rightarrow |X|$ **do**
    $w_t \leftarrow w_{t-1} + P_t x_t (y_t - w_{t-1}^T x_t)$
    $P_t \leftarrow \frac{1}{\lambda + x_t^T P_{t-1} x_t}(P_{t-1} - P_{t-1} x_t x_t^T P_{t-1})$
  end for
until w_i's converge

Figure 4.1. RLS algorithm.

4.3.2. Least Mean Squares Estimation

One can solve the least squares optimization problem by using the *least-mean squares estimation* (LMS) algorithm, which is also known as *the delta rule.* The algorithm starts by randomly assigning small positive values to the weight parameter w.

Then we update the weights in each step by finding the gradient of the mean square error (MSE). The LMS algorithm uses the stochastic gradient descent to minimize the MSE by recursively estimating of the optimal weight vector $\text{argmin}_w J(w)$ for

$$J(w) = \frac{1}{m}\sum_{i=1}^{m}(y_i - w^T x_i)^2. \tag{4.11}$$

Using *batch gradient descent* we derive our update rule as:

$$w(t+1) = w(t) - \eta\nabla J(w) = w(t) + \frac{2\eta}{m}\sum_{i=1}^{m}(y_i - w^T x_i)x_{ij} \tag{4.12}$$

We can use the stochastic gradient descent for online processing as follows:

$$w(t+1) = w(t) + \eta(y_i - w^T x_i)x_{ij} \tag{4.13}$$

Figure 4.2 shows the pseudo-code for a solution with stochastic gradient descent.

Initialize $w(0)$ to random small values
repeat
  for $j = 1, \ldots, |\mathbb{X}|$ **do**
    $w(t+1) \leftarrow w(t) + \eta(y_i - w^T x_i)x_{ij}$
  end for
until w's converge

Figure 4.2. LMS algorithm.

4.3.3. RLS vs. LMS

The LMS algorithm is the standard first order SGD and takes a scalar as learning rate. The recursive least squares algorithm (RLS), on the other hand, is based on weighted least squares in which past values are taken in account to determine future values. It considers an online approach to the least squares problem, and optimizes the mean square error by processing the input deterministically rather than stochastically.

Compared to the LMS, we need higher precision implementations for the RLS. In other words, in comparing the computational costs of the LMS and the RLS in terms of the number of operations, these numbers should be interpreted along with the fact that the calculations used by RLS have to have higher precisions. Otherwise, the result may diverge which adds another computational disadvantage of the LMS.
Additionally, if the input data is non-stationary, we must set a suitable forgetting factor λ in order to determine the covariance (gain) matrix P_t that controls the convergence speed.

Because of the matrix-product operations, the complexity of the RLS algorithm is $O(m^2)$ per iteration. On the other hand, the complexity of LMS is $O(m)$. This means the RLS algorithm does not scale well with dimensionality. The main advantage of the RLS algorithm is that it converges much faster than the least mean squares (LMS) method [23].

4.4. Classification Algorithms for Data Streams

Recent research on streaming classifiers similar to data stream clustering are mainly based on modifying batch methods to obtain incremental versions, or batch methods applied to sketch of the data provided from summarization techniques such as *frequent pattern mining* which we introduced in Section 4.5. For example, support vector machine can be made incremental using stochastic gradient descent to by updating the training data with the continuous data stream. Hoeffding trees [24] is presented for tree based data streaming classification. It relies on the idea that in the split stage of a decision tree a small subset of the training examples may be sufficient. The size of this subset relates with Hoeffding bound [24]. This algorithm has advantage with limited memory necessity. The drawback is, when a concept drift occurs its architecture does not allow to handle this, since once a node is created, it can never change.

Most neural network algorithms need to iterate through the training data many times to determine their model parameters. Even though in the streaming scenario this is not possible, it may not be necessary if the number of samples are large enough. Incremental approach to the model update process of neural network algorithms makes them particularly suitable for data stream scenario. The extensions of multilayer perceptron to the stream scenario is discussed in the Chapter 6. Below we first introduce basic units of a neural network, *perceptron model*, to compare its online structure with different algorithms that can be converted to online using stochastic gradient descent. There is a number of different neural network architectures [25] specifically designed for clustering which are mostly based on the concept of *competitive learning* [26]. In Section 4.6.1 we introduce competitive neural networks, and in Chapter 5 Section 5.5 we discuss its variation adaptive resonance theory.

4.4.1. Perceptron

In machine learning, a perceptron is a supervised learning algorithm used for binary classification problems. Original model is proposed by Rosenblatt [18]. It is a linear method because the classification model is represented by a linear function. Its task is to learn a classification function which corresponds to a hyper-plane separating two classes of data points that lie in a vector or an affine space. The perceptron takes a linear combination of the weights (parameters) in combination with the features of a data point, and feeds it through a step function. The neuron fires if the linear combination of the inputs exceeds the threshold, and it lays dormant otherwise. Figure 4.3 shows the architecture of the perceptron model. We are going to investigate multi-layer perceptron methods in Chapter 6 in detail.

Assume we have n points in the training dataset $(x_i, y_i)_{i=1}^{n}$ where each x_i has d features. We need to obtain a classification function $f : \mathbb{R}^d \to \mathbb{Y}$. Let

$$f(x) = sign(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ -1 & \text{otherwise.} \end{cases}$$

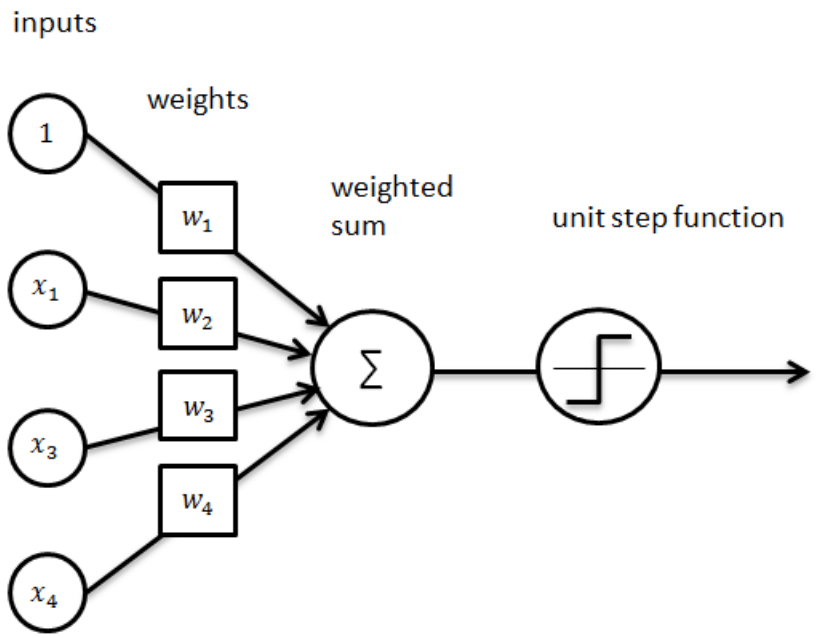

Figure 4.3. Perceptron model.

We minimize the following objective function:

$$J(w) = \frac{1}{2}\sum_{i=1}^{n}(f(x_i; w_i) - y_i)^2 \tag{4.14}$$

Here, w represents the real-valued weights, $w \cdot x$ represents the ordinary inner product $\sum_{i=1}^{m} w_i x_i$ where m is the number of inputs, and b represents the bias. The bias does not depend on input values, and pushes the decision boundary away from the origin. The algorithm looks for an optimal hyperplane $w \cdot x = b$ that separates two classes of input by updating the parameters w using the *gradient descent*

$$w(k+1) = w(k) + \eta.(y - \hat{y})x \tag{4.15}$$

where η is *learning rate*. The convergence is guaranteed only when the two classes are *linearly separable* [27].

Initialize W to random small values
repeat
  for $(x_i, y_i) \in \mathbb{X}$ in random order **do**
    if $y_i w \cdot x_i \leq 0$ **then**
      $w(t) \leftarrow w(t-1) + y_i x_i$
    else
      $w(t) \leftarrow w(t-1)$
    end if
  end for
until T time step
return $w \leftarrow \sum_{t=1}^{T} w(t)$

Figure 4.4. Perceptron algorithm.

4.4.1.1. LMS vs. The Perceptron. Difference between the LMS and the perceptron is that for the latter we can construct a differentiable cost function J while for the former the hard non-linearity f is replaced with a linear function. Also, LMS has a continuous error while for the perceptron error is discrete: it is either 0 or ± 2.

$$\nabla w = \eta(y - \hat{y}) f^{'}(.) \cdot x \tag{4.16}$$

If we set a linear activation function, e.g $f(x) = x \Rightarrow f^{'}(x) = 1$

$$\Rightarrow \nabla w = \eta(y - \hat{y}) \cdot x \tag{4.17}$$

If we choose activation function as the logistic (sigmoid) function, the single-layer network is identical to the logistic regression model.

4.4.2. Logistic Regression

Recall that in Section 3.2 we defined a *linear classifier* optimizing likelihood objective function for a given collection of data points $\{x_i, y_i\}_{i=1}^N$ with $x_i \in \mathbb{R}^d$ and $y_i \in \{0, 1\}$ as follows:

$$\ell(w) = \sum_{i=1}^{m} y_i \log(h(x_i) + (1 - y_i) \log(1 - h(x_i) \tag{4.18}$$

We have the following update rule:

$$w_j = w_j + \eta \sum_{i=1}^{m} (y_i - h_w(x_i)) x_{ij} \tag{4.19}$$

Using stochastic gradient descent for a sample (x_i, y_i) the update rule becomes:

$$w_j = w_j + \eta (y_i - h_w(x_i)) x_{ij} \tag{4.20}$$

We note that the stochastic gradient descent update is identical with the update rule of the linear regression with a squared error objective function as given in Equation (4.13), with the logistic regression update rule with a log likelihood objective function as given in Equation (4.20), and finally with the perceptron update rule with a sign objective function as given in Equation (4.15). However, in each case the model is different where we have $\vec{w} \cdot \vec{x}$, $\frac{1}{1+e^{-\vec{w} \cdot \vec{x}}}$, and $sign(\vec{w} \cdot \vec{x})$, respectively.

4.4.3. Online Support Vector Machines

Our task is to find a suitable w that minimizes an objective function of the form

$$\min_w \sum_{i=1}^{m} f_i(w) \tag{4.21}$$

The stochastic gradient descent iterates over such functions f_i, and updates the vector w as in

$$w_{t+1} \leftarrow w_t + \eta_t \nabla_{w_t} f_i(w_t) \tag{4.22}$$

after each iteration where $\nabla_w f_i(w)$ denotes the gradient of the function $f_i(w)$.

Recall that the SVM optimization problem is defined as:

$$w^* = \underset{w}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} \max\{0, 1 - y_i w \cdot x_i\} + \frac{\lambda}{2} \|w\|^2 \tag{4.23}$$

If we apply SGD to SVM optimization problem we see that here the functions f_i are of the form $f_i(w) = \max\{0, 1 - y_i w \cdot x_i\}$. The gradient of these functions are not easily defined. But we can use their *sub-gradients* which are defined as follows:

$$\nabla_w \max\{0, 1 - yw \cdot x\} = \begin{cases} -yx & \text{if } 1 - yw \cdot x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Hence, the sub-gradient of Equation (4.23) for a sample i is:

$$\begin{cases} -yx + \lambda w & \text{if } 1 - yw \cdot x \geq 0 \\ \lambda w & \text{otherwise} \end{cases}$$

We can now use this in our iterative updates.

4.5. Frequent Pattern Mining

Frequent pattern mining refers to extracting frequency information from a given dataset of the form of item sets, sequences, or subtrees depending on the type of the dataset and the mining tasks. One can also consider any such information on frequent patterns within a dataset as a particular form of a summary of that datasets.

Initialize λ, η_0
$W \leftarrow 0$
repeat
  for $(x_i, y_i) \in \mathbb{X}$ in random order **do**
    $\eta_t \leftarrow \eta_0/\sqrt{t}$
    if $1 - y_i w \cdot x_i \geq 0$ **then**
      $w(t) \leftarrow (1 - \eta_t \lambda) w(t-1) + \eta_t y_i x_i$
    else
      $w(t) \leftarrow (1 - \eta_t \lambda) w(t-1)$
    end if
  end for
until T time step
return $w \leftarrow \sum_{t=1}^{T} w(t)$

Figure 4.5. SVM by stochastic gradient descent.

The method can also be used for other machine learning tasks, such as classification, clustering, and change detection [28–31].

Many frequent pattern algorithms require multi-passes over the dataset, therefore they need persistent storage for the datasets they process [32–34]. With recent interest in mining data streams, where only one-pass is allowed, new frequent mining algorithms are proposed over data streams. However, such streaming algorithms can only yield approximate results.

Lossy counting is introduced by Manku and Motwani as a one-pass algorithm to find all frequent item sets over over the entire history of a stream [35]. Their algorithm does not allow false negatives, and has a provable bound on false positives. The algorithm uses a user-provided error parameter E that controls the quality of the model.

A popular approach in data stream clustering is using a *time window*, a contiguous subsequence of the data of a fixed length, that covers the most recent data to summarize the continuously arriving stream of data by giving greater importance to most recent objects appearing in the stream. Among the distinct window models used in the literature, we highlight the landmark model, sliding-window model, and damped model [36, 37].

The *landmark models* consider the data points in the stream starting from the beginning. Algorithms processing a stream using landmark windows perform their task on disjoint chunks of the stream separated by relevant objects called *landmarks*. Landmarks can be determined in terms of time, or in terms of the number of elements observed since the previous landmark [36].
Objects arriving after a landmark are kept and the algorithm creates suitable summary statistics within that window of recent data. All objects kept previously are removed once a new landmark arrives, and the algorithm repeats the summarization process in the new window.

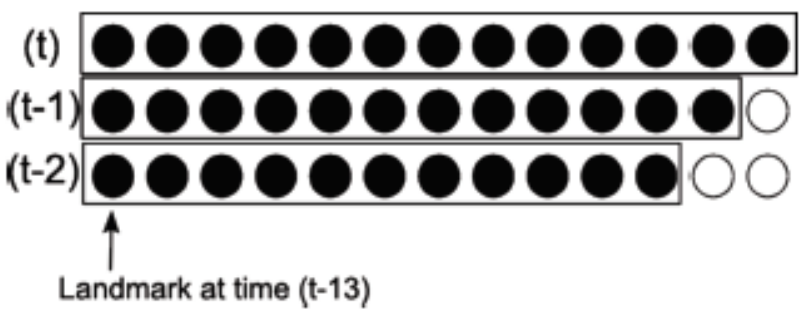

Figure 4.6. Landmark model [1].

In the *sliding-window* model, we consider a specific interval of data between now extending to a certain range into the past. Algorithms using the sliding window model store only the most recent information from the stream. Algorithms using the sliding window model maintain their frequent item sets in sliding windows usually by employing a *first-in-first-out* (FIFO) data structure. Such algorithms store and process only the part of the data stream within the sliding window.

The particular size of the sliding window depend on the application and system constraints. All the transactions within the window are kept until they move out of the range of the sliding window. The challenge of using sliding window method is in avoiding *load shedding effect*. This happens when the data arrival rate in the stream is higher than the processing rate of the algorithm, and packets of data may be dropped which may never be processed.

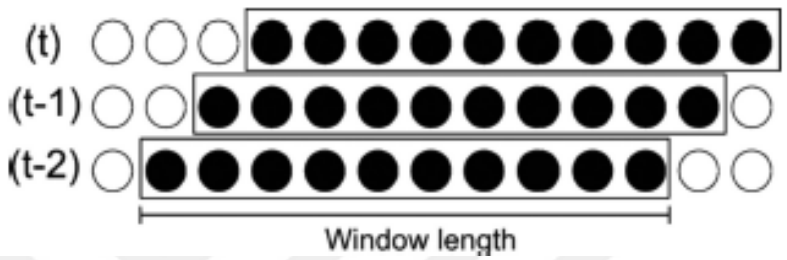

Figure 4.7. Sliding window model [1].

Damped model, also referred to as *time-fading model*, associates weights to data points in the stream in such a way that recently arriving data has higher weights than the data that arrived earlier [37]. We do this by assigning each data point, or a transaction, a weight that decreases with age. In calculating the result, older transactions contribute less than the newer transactions. This is suitable for applications in which newly arrived data has more effect on the results, or the effect reduces with time.

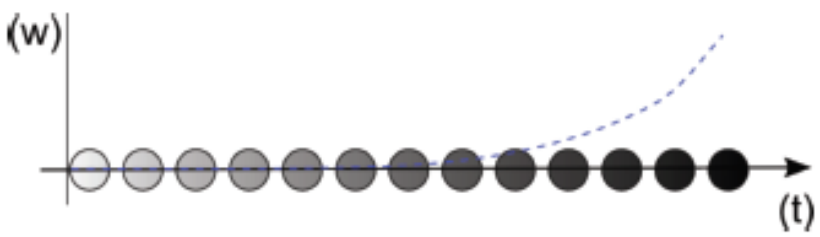

Figure 4.8. Damped model [1].

4.6. Clustering Algorithms for Data Streams

In this section, we are going to give an overview of clustering algorithms for data streams, and briefly mention commonly used methods. We will investigate the k-means algorithm in depth later in Chapter 5.

Clustering algorithms belong ta a basic class of unsupervised machine learning algorithms that assigns data points labels from a fixed finite set in order to split them into partitions. We would like to form such partitions in such a way that points within each group are more similar than the points in different groups.

For static data sets one can use a number of algorithms such as k-means, k-medoids, hierarchical clustering, and density-based methods [38]. The standard clustering algorithms need to access all of the data points and typically iterate over the available data multiple times. This requirement renders these algorithms unsuitable for large data streams especially for datasets that are too large to fit within the memory available, or for datasets where data points arrive continuously.

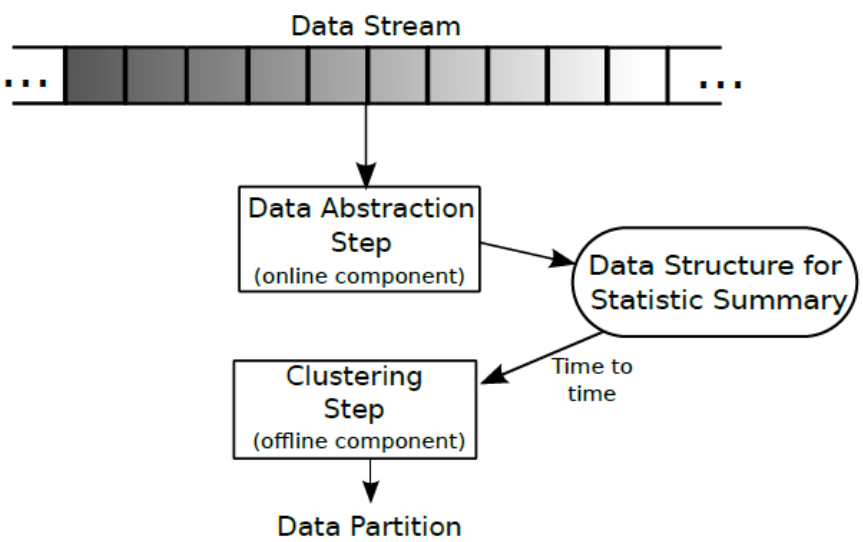

Figure 4.9. Data stream clustering framework with abstraction [1].

Most stream clustering algorithms can be thought as object clustering algorithms. One strategy is to apply a data abstraction step, such as using a summary of frequent items (see Section 4.5), and then obtain a data partition via an offline clustering step. This strategy is usually very efficient in practice [39]. In this setup, traditional clustering algorithms such as DBSCAN [40] or k-means [41, 42] can be used very effectively to partition data into relatively small clusters over the summaries.

4.6.1. Competitive Neural Networks

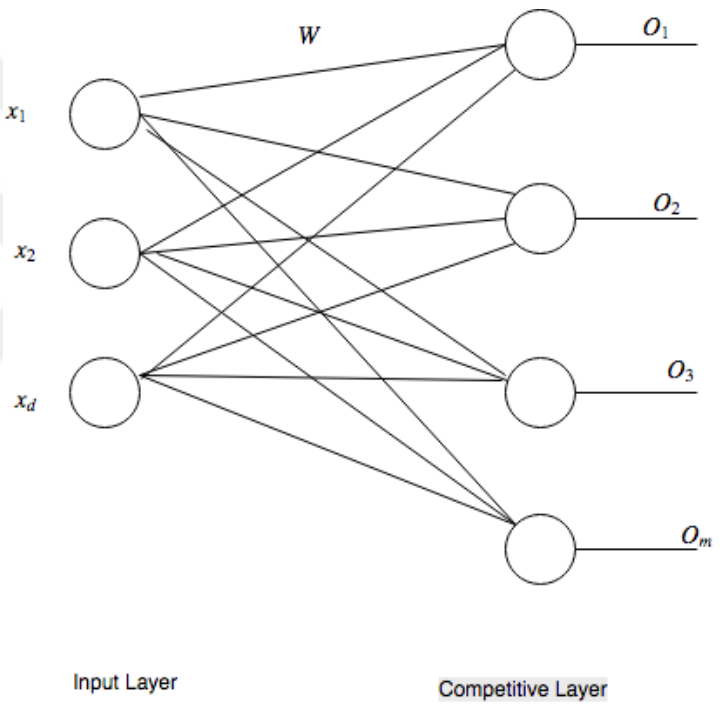

Figure 4.10. The winner-take-all competitive neural network. This is a network of k perceptrons with recurrent connections at the output. Each unit at the output reinforces its value and tries to suppress the other outputs. Under a suitable arrangement of recurrent weights, the maximum suppresses all the others.

Competitive neural networks are used mainly for online clustering problems. The input layer contains the input vector x, outputs are fully connected and there is no bias unit. Only binary inputs and outputs are considered. Figure 4.10 shows the basic structure of a competitive neural network. The values of the output units h_i are perceptrons:

$$h_i = w_i^T \cdot x$$

The activation, of the i^{th} output node is just the inner product of the weight vector of the node and the current input vector:

$$h_i = \sum_{j=1}^{d} w_{ij} \cdot x_j^t = w_i^T \cdot x^t = \|w_i\| \cdot \|x^t\| \cdot cos(\theta) \tag{4.24}$$

which is closely related to the difference between the two vectors:

$$\|w_i - x^t\|^2 = \|w_i\|^2 + \|x^t\|^2 - 2\|w_i\| \cdot \|x^t\| \cdot cos(\theta)$$

where θ is the angle between the two vectors.

In a competitive network, only one output unit, called *the winner unit*, can be become active at a time. The winner unit i^* is the only node that fires and it has the largest net input. Hence, we must have

$$w_{i^*} \cdot x \geq w_i \cdot x, \text{ for all } i \tag{4.25}$$

where the winning unit i^* has $h_{i^*} = 1$. If we normalize all the weights to $\|w_i\| = 1$ for every i, since we have

$$\|x - y\|^2 = \|x\|^2 - 2\langle x, y \rangle + \|y\|^2 \tag{4.26}$$

we see that the winner unit is the one that minimizes the squared distance

$$i^* = \underset{i=1,\ldots,k}{\operatorname{argmin}} \|w_i - x\| \tag{4.27}$$

where the term k represents the number of nodes in that particular layer of the network.

This means the unit with the minimum Euclidean distance is the same as the one with the maximum dot product. In other words, the winner is the unit with normalized w closest to the input vector x.

Initialize $W \leftarrow$ the first k distinct vectors in $\mathbb{X}$
repeat
  for $x^t \in$ remainder of $\mathbb{X}$ **do**
    $i \leftarrow \text{argmin}_j \|x^t - w_j\|$
    $w_i \leftarrow w_i + \eta(x^t - w_i)$
    $w_i \leftarrow \frac{w_i}{\|w_i\|}$
  end for
until w_i's converge

Figure 4.11. Competitive neural network.

5. ONLINE K-MEANS

In this chapter we introduce the k-means algorithm and its variations using batch and online processing. The k-means clustering algorithm, which is an iterative method, is one of the most popular unsupervised clustering algorithms in machine learning for partitioning data into representative clusters [43–47]. The standard k-means algorithm requires renders data multiple times. This requires all of the data to be available. In the online version, the algorithm takes action as each point arrives. In the streaming case, the algorithm can defer action until a group of data points arrive, *de facto* forming batches. The main difference between these different processing techniques is the amount of memory available. Even though the nature of the data itself plays an important role in determining which type of processing is suitable, it is crucial to analyze trade-offs between these algorithms appropriately for our objectives and cost constraints.

5.1. The K-means Problem

The k-means algorithms aims to find optimal centroids for different clusters of data that lie in an Euclidean space for a desired number of partitions. One can reformulate the k-means problem as an optimization problem as follows:

Given a finite set $\mathbb{X} \subset \mathbb{R}^d$ and a positive integer $k \in \mathbb{Z}^+$, we need to find a function $w \colon \mathbb{X} \to \mathbb{R}^d$ such that $|im(w)| = k$ and the following cost function is minimized:

$$\sum_{x \in \mathbb{X}} \|x - w(x)\|^2. \tag{5.1}$$

Our search space for an optimal solution is the set of all functions $w \colon \mathbb{X} \to \mathbb{R}^d$ with $|im(w)| = k$, i.e. we need

$$\operatorname*{argmin}_{w} \sum_{x \in \mathbb{X}} \|x - w(x)\|^2.$$

Since the sum of the squared Euclidean distances is minimized when w assigns the mean of the cluster that the data point belongs to, this problem now is equivalent to finding cluster centroids and writing the function $w(x)$ that assigns the centroid of the cluster to each point $x \in \mathbb{X}$.

Finding globally optimal solution for this problem is known to be NP-hard [48]. So, rather than using an exhaustive search whose complexity is exponential, iterative and refinement based algorithms are commonly used to approximate the solution of the optimization problem we described above.

5.2. Facility Location Problem

In this problem we have a metric space with a multiset of demand points chosen from the space. It is a version of the k-means problem in which the cost of adding a centroid is added to the objective function to be minimized. For the facility location problem, the number of clusters is not a part of the input, as it was for k-means. Instead, we have a facility cost function. An algorithm designed to solve this problem may have as many clusters as it desires in its output simply by denoting some point as a facility. Then the optimal solution is going to be the sum of the resulting k-means cost, which can be considered as the *service cost*, plus the cost for opening the facilities. Our task is to find a set of locations in the metric space that minimizes the total service cost and the assignment cost where the service cost for a demand point is its distance to the nearest open facility. We determine the coordinates of the facilities we are going to open so that the sum of the total facility cost and the service cost is minimized.

In the online version, the set of demand points is not known in advance. Demand points arrive one at a time. After the arrival of a demand point, the algorithm has to decide whether to open a facility without any information about the further demands. In the plain k-means problem, $|im(w)|$ which is the total number of facilities, must be equal to k. In both cases, we are looking for solutions by optimizing a cost function. In this case, the optimization problem we need to solve can be summarized as follows:

Given a finite set $\mathbb{X} = \{x^t\}_{t=1}^N$ of points $x^t \in \mathbb{R}^d$, find a function $w\colon \mathbb{X} \to \mathbb{R}^d$ such that

$$\sum_{x \in \mathbb{X}} \|x - w(x)\|^2 + Cost(w) \tag{5.2}$$

is minimized where $Cost(w)$ is the total cost to open the collection of facilities determined by w.

5.3. K-means Clustering Algorithm

The k-means clustering algorithm we are going to define in this section is a multi-pass clustering algorithm, and it requires access to all of the data points multiple times. The algorithm groups the data based on their mutual euclidean distances, and therefore closer data points are more likely to be in the same group. The algorithm takes the mean value of a group as the similarity parameter and forms clusters by assigning data points based on the closest mean. This algorithm firstly reported by [49].

Given a data set $\mathbb{X} = \{x^t\}_{t=1}^N$, and a positive integer k, the algorithm starts by randomly selecting k points as the initial cluster centroids. Then it assigns each data point to its closest cluster centroid. Let $W = \{w_1, \ldots, w_k\}$ be the initial cluster centroids then each data point x^t is assigned to a cluster based on:

$$w(x) = \underset{w_i \in W}{\operatorname{argmin}} \|w_i - x^t\|^2.$$

Then the algorithm recalculates the cluster centroids $\{w_j\}_{j=1}^k$ as the mean of the data points in the current clusters.

Let S_i be the set of data point assignments for the i^{th} cluster. Let us define

$$h_i^t = \begin{cases} 1 & \text{if } i = \operatorname{argmin}_\ell \|x^t - w_\ell\| \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

The batch algorithm, k-means, updates the centroids as

$$w_i = \frac{\sum_t h_i^t x^t}{\sum_t h_i^t} = \frac{1}{|S_i|} \sum_{x^t \in S_i} x^t \tag{5.4}$$

Data points are reassigned to their new clusters with respect to the new cluster centroids and centroids are updated until the cluster centroids stabilize. The first four iterations of an example run of the k-means clustering algorithm is illustrated in Figure 5.1. The pseudocode of the k-means algorithm is given in Figure 5.2.

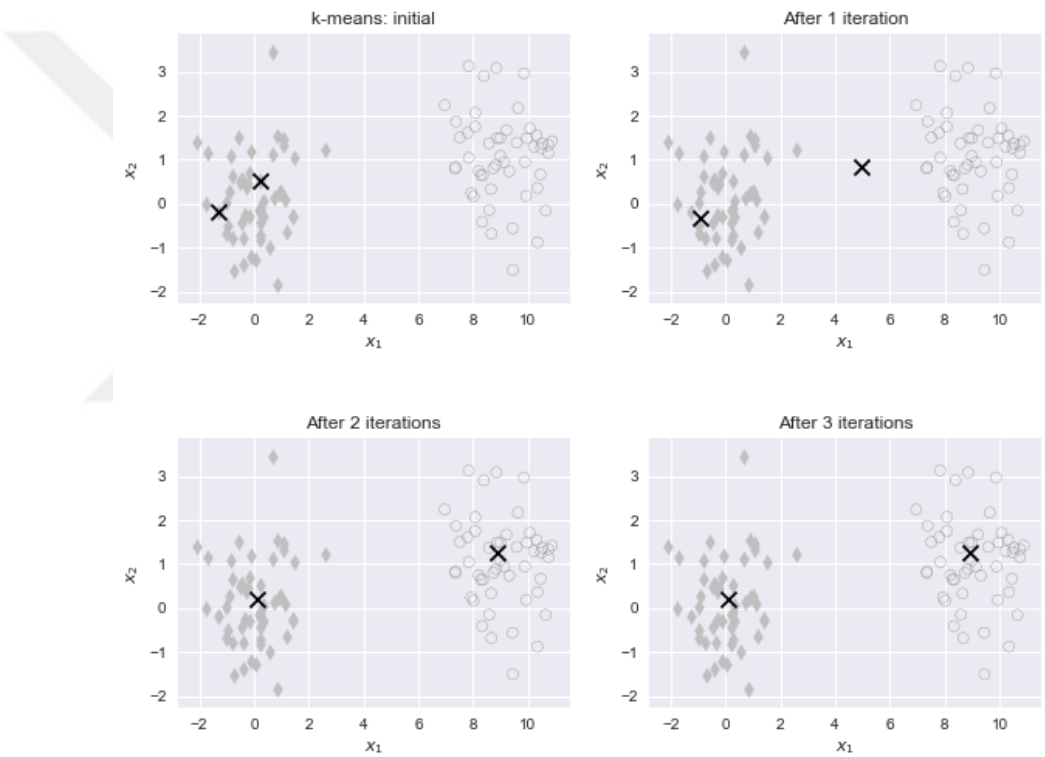

Figure 5.1. Illustration of the four steps of the k-means clustering algorithm.

5.3.1. Cost, Convergence and Complexity

The cost function of k-means algorithm is the squared objective function:

$$J(k, W) = \frac{1}{2} \sum_{t=1}^{N} \sum_{i=1}^{k} h_i^t \|w_i - x^t\|^2 \tag{5.5}$$

for $i \leftarrow 1$ to k **do**
  Randomly initialize w_i
end for
repeat
  for $t = 1 \rightarrow |X|$ **do**
    $\ell = \text{argmin}_j \|x^t - w_j\|$
    for $i \leftarrow 1$ to k **do**
      if $i = \ell$ **then**
        $h_i^t \leftarrow 1$
      else
        $h_i^t \leftarrow 0$
      end if
    end for
  end for
  for $i \leftarrow 1$ to k **do**
    $w_i \leftarrow \frac{\sum_t h_i^t x^t}{\sum_t h_i^t}$
  end for
until w_i's converge

Figure 5.2. k-means clustering algorithm.

The k-means algorithm is guaranteed to converge because J is monotonically decreasing, and J is bounded below. However, since J is a non-convex function, it is not guaranteed to converge to its global minimum. In other words, k-means can stuck on a local minimum. In other words, random initialization of the cluster centroids might give different results. In batch processing, a simple strategy to avoid this situation is to run k-means many times using different random initial values for the cluster centroids w_j, and then pick the one that gives the lowest value for J.

The complexity of this algorithm is $O(kNdT)$, where k is the number of clusters, N is the input length, d is the dimension of the data points, and T is the iterative of time for the algorithm. Normally, $k \ll N$ and $d \ll N$, and we can see that k-means is linear in the number of data points.

5.4. Online K-means with Stochastic Gradient Descent

In this section, we make the same assumption that there are k clusters. The crucial difference is that we do not have the whole sample for the training, and we receive instances one by one.

One approach [2] to obtain an online version of the k-means algorithm is to use stochastic gradient descent. Let $\mathbb{X} = \{x^t\}_{t=1}^N$ and $x^t \in \mathbb{R}^d$. Then the objective function for a single instance is:

$$J^t(k, W) = \frac{1}{2}\sum_{i=1}^{k} h_i^t \|x^t - w_i\|^2 = \frac{1}{2}\sum_{i=1}^{k}\sum_{j=1}^{d} h_i^t (x_j^t - w_{ij})^2 \tag{5.6}$$

where h_i^t is defined as Equation (5.4). Using the gradient descent method on this function, we get the following update rule for each instance x^t:

$$\Delta w_{ij} = -\eta \frac{\partial J^t}{\partial w_{ij}} = \eta h_i^t (x_j^t - w_{ij}) \tag{5.7}$$

Given a data point x^t, we update only a single centroid at a time, which we call *the winner unit* as in Section 4.6.1, by moving the centroid in the direction of x^t by η. We give the pseudocode of the online version of the k-means algorithm in Figure 5.3.

Initialize $W \leftarrow$ the first k distinct vectors in $\mathbb{X}$
repeat
  for $t = 1 \rightarrow |X|$ **do**
    $i = \text{argmin}_j \|x^t - w_j\|$
    $w_i \leftarrow w_i + \eta(x^t - w_i)$
  end for
until w_i's converge

Figure 5.3. Online k-means algorithm with SGD.

5.4.1. Cost, Convergence and Complexity

In order to gain convergence, we must let η go to 0 [9]. However, this may lead to what is called *the stability-plasticity dilemma*: If η is going towards 0 we may achieve stability but we may lose adaptivity because updates are now too small. On the other hand, if we keep η large then the values w_i may oscillate.

In a non-stationary environment, the initialization of cluster centroids with the first k instances from the stream may cause the algorithm to stuck in a local optimum, and give rise to centroids that are never used by the algorithm. For competitive networks, these are the centroids that would always lose because the initialization step put them too far away from most of the data points. There are algorithms for adding new centroids if an input is far from the existing centroids defined by a threshold value. One example is the ART model, which we discuss below in Section 5.5. To avoid this problem, one can also include the information of the global cost function to obtain an adaptive system as we do in Section 5.6.

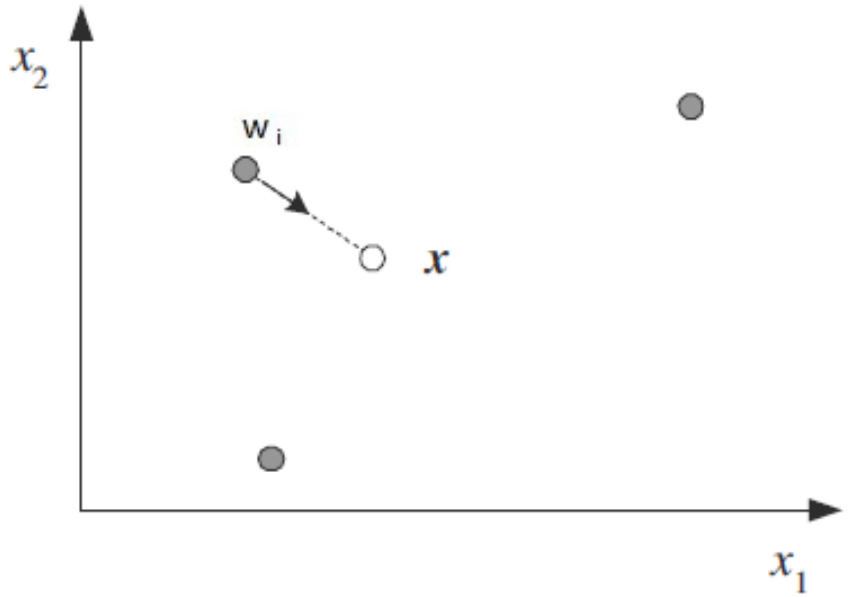

Figure 5.4. Centroid update in stochastic k-means: the closest centroid w_i to the input x moves along the direction of $(x - w_i)$ by a factor η [2].

Algorithm in Figure 5.3 keeps updating a set of k centroids using one sample at a time, so the complexity is reduced to $O(1)$ from $O(N)$ in the batch case for each iteration.

5.5. Adaptive Resonance Theory Networks

In the algorithms we discussed above, we needed to know the number of clusters before the algorithm starts processing the stream. If the number of clusters is not known, then we follow an incremental approach where we start with a single cluster, and then we gradually open up new clusters as data flows in and we need new centroids. The adaptive resonance theory (ART) algorithm [50] is an example of competitive neural networks, and it can be effectively used for this purpose.

In the ART method, all of the output units of the network calculate their values of similarity for any given input. Only one centroid, the one with the minimum Euclidean distance of its weight vector and the current input, can become active at a time.

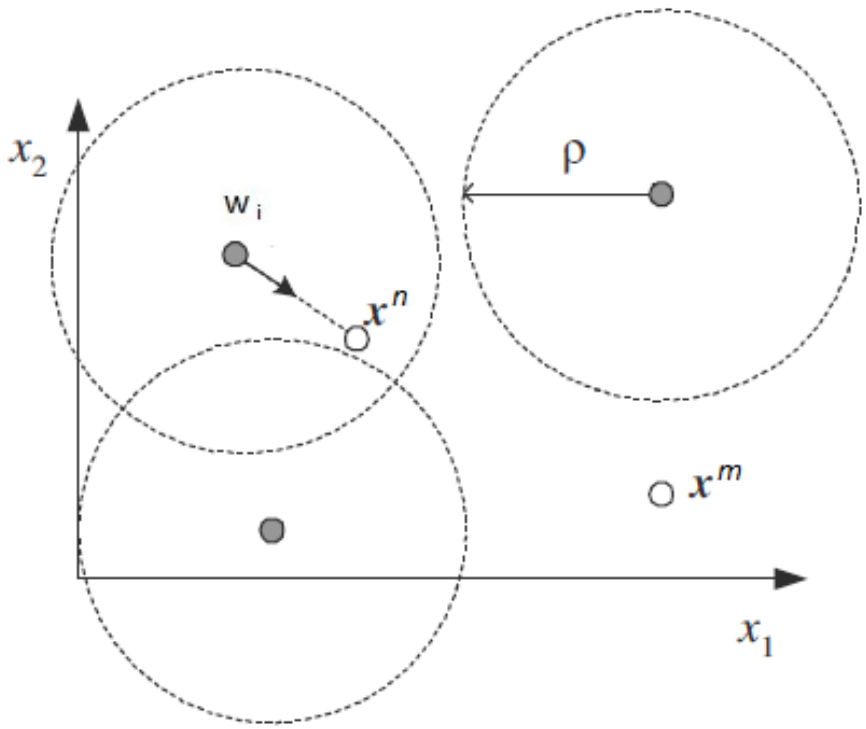

Figure 5.5. Distance between x^n and the closest centroid w_i is less than the vigilance value ρ and the centroid is updated as in online k-means. However, x^m is not close enough to any of the centroids and a new cluster should be created at that position [2].

The update is performed as in Algorithm 5.3 only when the minimum distance is smaller than a chosen *vigilance parameter*. If this distance is larger than the vigilance parameter, a new output unit is added and its center is initialized with this instance. See Figure 5.5.

Let $W = \{w_1, \ldots, w_k\}$
for $t = 1 \rightarrow |X|$ **do**
  $r_i \leftarrow \|w_i - x^t\| = min_{l=1}^{k} \|w_l - x^t\|$
  if $r_i > \rho$ **then**
    $w_{k+1} \leftarrow x^t$
    $W \leftarrow W \cup w_{k+1}$
  else
    $w_i \leftarrow w_i + \eta(x^t - w_i)$
  end if
end for

Figure 5.6. The ART algorithm.

We can see in the algorithm in Figure 5.6 that each output unit w_i corresponds to a cluster formed within the disk centered at the centroid with radius given by the vigilance. If an upcoming point does not fall into any cluster disk, a new cluster is opened.

5.5.1. Cost, Convergence and Complexity

In the ART method, we do not make an assumption on the number of clusters. However, we assume a prior knowledge about the density of the data in order to specify the vigilance parameter. This parameter is used as similarity criteria based on the Euclidean distance in our case.

Given a data set $\mathbb{X} = \{x^t\}_{t=1}^N$, one can write a function that relates the number of clusters $|W|$ to the vigilance parameter ρ. For a normalized data set, this function is expected to have a minimum value of $|W| = 1$ for $\rho = 1$ and a maximum value of $|W| = N$ for $\rho \to 0$. Forming clusters as disks of radius ρ limits the reconstruction error per instance by the square of the vigilance ρ.
Using Equation (5.6) and summing all over t, the total error is calculated as:

$$J(k, W) = \frac{1}{2} \sum_{t=1}^{N} \sum_{i=1}^{k} h_i^t \|w_i - x^t\|^2. \tag{5.8}$$

Notice that since each cluster lies within a ball of radius ρ, and only one of the terms h_i^t is non-zero for each t, we can get a hard bound to the objective function as

$$J(k, W) \leq \sum_{t=1}^{N} \sum_{i=1}^{k} h_i^t \rho^2 \leq N \cdot \rho^2$$

independent of the number of clusters k.

5.6. Semi-Online K-Means Algorithm

In this section, we are going to use a combination of streaming and online processing for the k-means clustering algorithm. The algorithm we are going to define assigns clusters using an online method while keeping the space and the time complexity at most poly-logarithmic in the length of the stream. To do that we follow the steps as in [6]. First, we need to introduce the semi-online k-means algorithm and then we are going to derive the *fully online k-means* algorithm from the semi-online algorithm.

In the semi-online model, we assume we have a lower bound $\mathfrak{j}^*$ on the total cost of the cost function J^* of the k-means algorithm , and we have an estimate for the length of the stream n. In the fully online model, we do not assume any prior knowledge nor on the lower bound $\mathfrak{j}^*$, neither on the length of the stream n. Algorithm in Figure 5.9 operates in this setting and opens a comparable number of clusters with the algorithm in Figure 5.7. But, its approximation factor guarantee degrades by a $\log(n)$-factor.

For the semi-online algorithm we are going to use ideas from the online facility location algorithm [51]. In this algorithm whose pseudocode is given in Figure 5.7, we begin with an empty set of clusters. In this setting, data points arrive in an arbitrary order. When a new data point arrives, the algorithm either assigns it to one of the existing clusters, or opens a new cluster which consists of this newly arrived point. It also adds *the cost of creating a cluster* to the total cost. This is analogous to the vigilance parameter in the ART algorithm. The *cost of opening a facility* is given in Equation (5.2). Our task is to find an approximation for the optimal cluster centroids.

In this algorithm, we start with a facility cost f_1 which is far less then the lower bound of the total cost $\mathfrak{j}^*$. So, in the beginning the algorithm tends to open many facilities (centroids). On the other hand, the algorithm also checks whether there are too many facilities because the current facility cost might be lower than optimum. If that is the case, the algorithm doubles the facility cost for future facilities.

In algorithm in Figure 5.7, the distances between a data point x and the cluster centroids W is denoted as

$$D(x, W) = \min_{w \in W} \|x - w\|.$$

Note that, if $W = \emptyset$ then $D(x, W) = \infty$ for any x.

$W \leftarrow \emptyset$

$r \leftarrow 1$

$q_1 \leftarrow 0$

$f_1 \leftarrow \frac{j^*}{k \log(n)}$

for $x \in \mathbb{X}$ **do**

  $z \leftarrow random(0, 1)$

  if $z < \min(D^2(x, W)/f_r, 1)$ **then**

    $W \leftarrow W \cup \{x\}$

    $q_r \leftarrow q_r + 1$

  end if

  if $q_r \geq 3k(1 + \log(n))$ **then**

    $r \leftarrow r + 1$

    $q_r \leftarrow 0$

    $f_r \leftarrow 2.f_{r-1}$

  end if

end for

return W

Figure 5.7. Semi-online k-means algorithm.

5.6.1. Cost, Convergence and Complexity

Let $S_1^*, \ldots, S_k^*$ be an optimal solution with cluster centroids $w_1^*, \ldots, w_k^*$. Let

$$J_i^* = \sum_{x \in S_i^*} \|x - w_i^*\|_2^2$$

be the cost of the i-th cluster in the optimal solution, and

$$J^* = \sum J_i^* \tag{5.9}$$

be the total value of the cost function for the optimal solution. In [6, Theorem 2] Liberty *et. al.* (2015) prove that the estimated expected cost of the clusters opened by algorithm in Figure 5.7 is

$$\mathbb{E}[J] = O(J^*)$$

where J is the cost function for algorithm in Figure 5.7. Moreover, if we write W for the set of clusters defined by algorithm in Figure 5.7, then by [6, Theorem 1] we get that the expected number of clusters opened by Algorithm 5.7 is

$$\mathbb{E}[|W|] = O(k \log n \log \frac{J^*}{\mathfrak{j}^*}).$$

5.7. Fully Online K-Means Algorithm

Algorithm in Figure 5.9 processes the data points in a fully online manner. The number of the data points n is unavailable to the algorithm. We initialize the cluster centroids as the first $k+1$ distinct vectors as the data points arrive. The algorithm determines a lower bound $\mathfrak{j}^*$ based on the initial cluster centroids. Our task is again to approximate the optimal centroids as data points arrive one by one. We again use a threshold value similar to the vigilance parameter that we used in the ART method.

Note that $\mathfrak{j}^*$ is trivially smaller that J^*. Any clustering of $k+1$ points with at least two points in one cluster incurs a cost of

$$\frac{\|x - x'\|^2}{2} \geq \min_{x,x'} \frac{\|x - x'\|^2}{2}.$$

Figure 5.8 shows an illustration of the steps of the fully online k-means clustering algorithm.

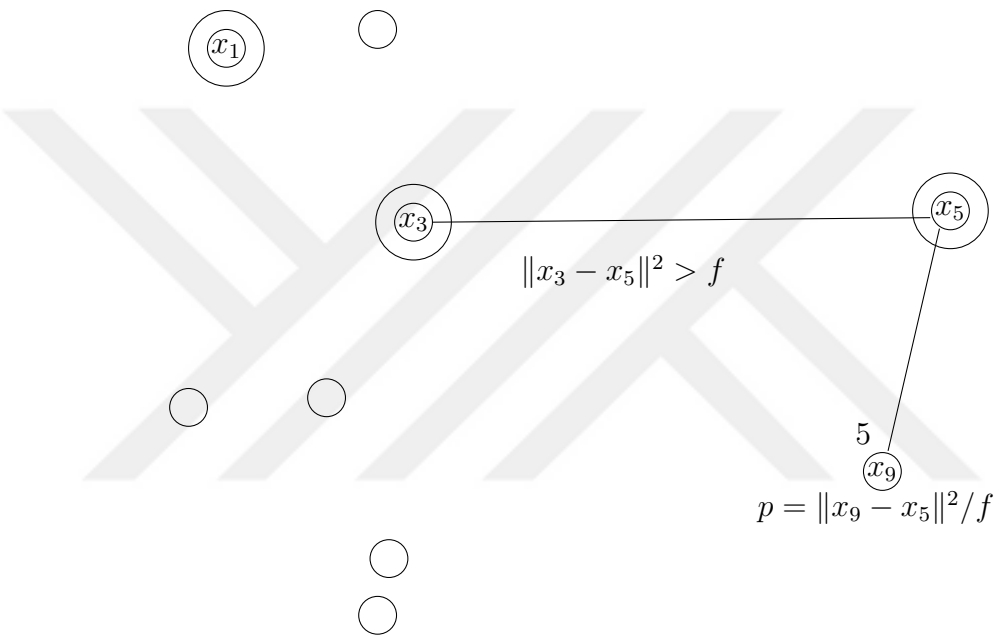

Figure 5.8. If the minimum squared distance between the entering point and the existing cluster centroids is more than the cost of creating a new cluster f, a new cluster is created. Otherwise, a new cluster is created with probability p. When the number of created clusters with the current cost f is more than $3k(1 + \log(n))$ the value of f is doubled.

The online k-means algorithm aims to bifurcate an online arriving stream of data $\mathbb{X}$ into relevant clusters. The algorithm is conducted in r phases. It takes as input a parameter k that denotes the minimum number of clusters that will be formed as output of the algorithm. Hence, the algorithm produces at least $k+1$ clusters. The initial $k+1$ points arriving from the online stream of data are assigned as the initial cluster centers. The distance between each of the points in W calculated and the square of the minimum distance divided by $2k$ is the initial facility opening cost.

Let f_r represent the facility opening cost for the r^{th} phase of the algorithm. Let us also use q_r to denote the number of clusters in r^{th} phase. Since no proper clusters are formed, the value of q_r remains zero in the beginning. New cluster is opened with the probability p which is the minimum squared distance between the incoming point x and the existing cluster centers divided by the cost f_r of that phase. With each successive phase, cost f_r is double of the previous cost so as to ensure that lesser clusters are opened in later phases. The algorithm moves to the next phase when $q_r \geq 3k(1 + \log(n))$ [52].

$W \leftarrow$ the first $k+1$ distinct vectors in $\mathbb{X}$; and $n = k+1$
$\mathfrak{j}^* \leftarrow \min_{x,x' \in W} \frac{\|x-x'\|^2}{2}$
$r \leftarrow 1; q_1 \leftarrow 0; f_1 = \mathfrak{j}^*/k$
for $t \in$ remainder of $\mathbb{X}$ **do**
  $n \leftarrow n+1$
  $z \leftarrow random(0,1)$
  if $z < \min(\frac{D^2(x,W)}{f_r}, 1)$ **then**
    $W \leftarrow W \cup \{x\}$, $q_r \leftarrow q_r + 1$
  end if
  if $q_r \geq 3k(1+\log(n))$ **then**
    $r \leftarrow r+1$; $q_r \leftarrow 0$; $f_r \leftarrow 2.f_{r-1}$
  end if
end for
return W

Figure 5.9. Fully online k-means algorithm.

5.7.1. Cost, Convergence and Complexity

Let J be the cost function for the algorithm in Figure 5.9, and let J^* be defined as in Section 5.6, Equation (5.9).

$$\mathbb{E}[J] = O(J^* \log n)$$

Let W be the set of clusters defined by Algorithm 5.9, and let

$$\gamma = \frac{\max_{x \neq x'} \|x - x'\|}{\min_{x \neq x'} \|x - x'\|}.$$

Observe that,

$$J^* \leq n \max_{x \neq x'} \|x - x'\|^2.$$

Then, by [6, Theorem 3]

$$\mathbb{E}[|W|] = O\left(k \log n \log \frac{J^*}{\mathfrak{j}^*}\right) = O(k \log n \log \gamma n).$$

The idea behind the proof is that if the cost for creating a cluster is too small, this fact results in creating many clusters because the total cost, the sum of squared distances, is going to be low. At the same time, every time $3k(1 + \log(n))$ clusters are added, the cost for creating a cluster is doubled which increases total cost in the long run. On the other hand, a large enough cost value for creating new clusters makes adding new clusters hard enough in such a way that at most $O(k \log(n) \log(n))$ clusters are created in expectation.

6. ONLINE MULTILAYER PERCEPTRON MODEL

6.1. Introduction

In this section we introduce multilayer neural networks with its standard backpropagation algorithm using batch gradient descent. Then we discuss stochastic gradient descent and recursive least squares estimation to implement backpropagation algorithm. The literature on the subject observes that neural networks need large volumes of data [53] to train a machine learning model. With current availability of large datasets in different applications, the main concern for implementing a neural network algorithm to solve a machine learning task is the computation time. We also discuss performance of these two models based on their training time, and accuracy.

Artificial Neural Networks are used for data approximation and data classifications problems. Originally, the inspiration for this architecture comes from biological neurons [18]. In this architecture, we have a collection of nodes and directed edges called *arrows*, and all computations flow through the network along the arrows. The value of a node P is determined by the nodes flowing into P along the arrows connected to P. Each arrow contains a weight which determines the strength of the connection.

Our discussion also contains the case of online network training. As in earlier sections, any online training method seeks to adjust the parameters as training inputs are presented rather than after a complete pass through the training set. In this case, we are going to use online techniques to determine the network parameters.

In the standard backpropagation algorithm, both for the stochastic and the batch cases, the weights are updated according to the generalized delta rule. In the recursive least squares backpropagation algorithm, the algorithm minimizes the mean-square error (MSE) between the target and the actual output with respect to the summation outputs. Delta values come from the standard backpropagation algorithm to correct the summation outputs.

In this algorithm weights are adjusted using these estimates together with the input vectors using a system of linear equations at each node. We solve these systems of linear equations using a recursive least squares estimation which we discussed in Section 4.3.1.

6.2. Multilayer Perceptron Model

The Multilayer Perceptron model is a modified single perceptron model which can separate data even when the data is not linearly separable. It has several layers between the input nodes and the output nodes. The data fed to a network flows from the input nodes to the output nodes. The training is performed using the *backpropagation algorithm*.

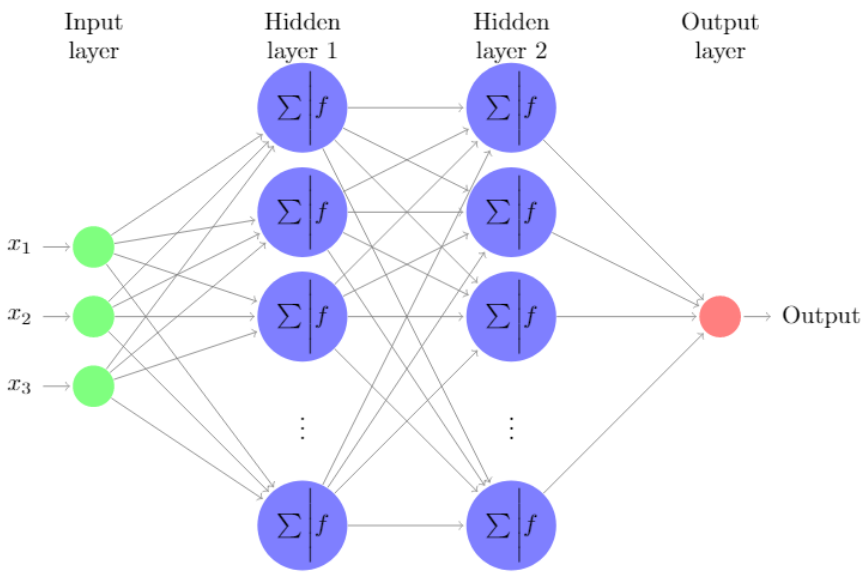

Figure 6.1. A multilayer perceptron model with 2 hidden layers, 3 input nodes, m hidden nodes and a single output node.

In the feed forward neural networks the information travels only in one direction as we calculate the sum of the the weights connecting one layer to the next. The value at a node then passes through a *transfer* (or an *activation*) function to determine if the neuron will produce a value propagating towards the output layer. See Figure 6.1.

Hidden layers with a non-linear activation function creates and maintains an internal representation of the data that can then be combined in different ways to model the functions we want to approximate. The nonlinearity of the activation functions causes most loss functions to become non-convex, hence, without a guarantee for convergence. For this reason, most neural networks use iterative gradient-based methods to find a minimal value for the cost function instead of linear solvers that we used in linear regression models, or the convex optimization algorithms which find global minimum that we used earlier to obtain logistic regression models or SVMs. Thus, initialization of a neural network becomes crucial since all feed-forward neural network algorithms initialize weights to small random values while the biases are initialized to zero, or to small positive values. In convex optimization, convergence of an iterative approximations is independent from any initial parameters one can choose because of the requirement of convexity on the objective function. However, the convergence of a stochastic gradient descent method applied to a non-convex objective function is sensitive to a chosen initialization.

6.3. Activation Function

A neuron in a layer of a neural network takes all its inputs from other neurons that are connected to it from its previous layer. After it multiplies these input values by their respective weights and calculates the sum, it passes this sum through an activation function. The output of this function is in the range $(0, 1)$ or $(-1, 1)$.

We prefer non-linear activation functions because linear activation functions reduce successive layers of a neural network to a single layer. This is because if all activation functions are linear, then the activation function on the last layer is just a linear function of the inputs on the first layer, and one can replace all layers with a single layer. In this way we loose the ability of stacking layers, and no matter how we stack, the whole network reduces to a single layer neural network with a linear activation function since a linear combination of linear functions is another linear function.

Additionally, in the backpropagation algorithm we use gradient information of the activations, and the derivative of a linear function with respect is constant. That means, the gradient of an activation function has no relationship with its input, and the descent is going to be on a constant gradient. If there is an error in the prediction, the changes made by back propagation are constant and do not depend on the change in input delta. So, the preferred activation functions are usually contractions which are non-linear and differentiable. Below we introduce commonly used activation functions.

(i) Logistic (Sigmoid) function: $\sigma(z) = \frac{1}{1+e^{-z}}$ where $\sigma : \mathbb{R} \to (0, 1)$

(ii) Hyperbolic Tangent: $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ where $\tanh : \mathbb{R} \to (-1, 1)$

(iii) Rectified Linear Unit: $R(z) = \max(0, z)$ where $R : \mathbb{R} \to (0, \infty)$

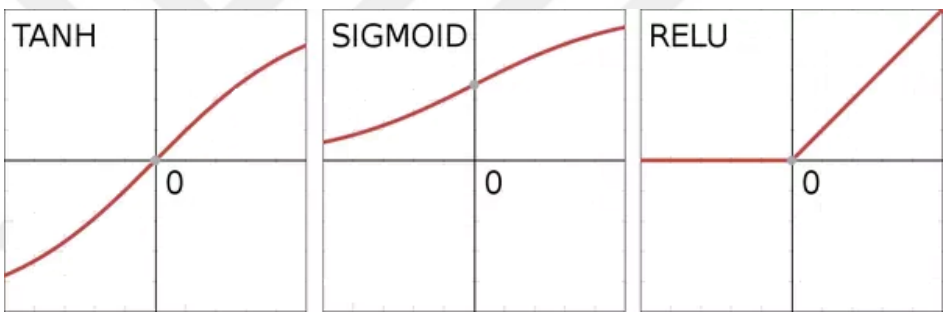

Figure 6.2. Graphs of sigmoid, tanh and ReLU.

6.4. Backpropagation Algorithm

The backpropagation algorithm is a recursive algorithm that searches for the minimum of the mean square error function in the weight space using the gradient descent. This means it adjusts each weight in the network in proportion to how much it contributes to overall error. We expect that the iterative reduction of each error associated with a weight eventually provides a series of weights that produce a good prediction. It propagates errors backward to obtain gradient information from hidden layers. This is crucial because for the hidden layers, the error E^{k+1} is not defined explicitly in terms of the weight parameters of the layer $(k + 1)$.

For input vectors x and target output vectors t in given a training data set, the algorithm propagates the error back by scaling according to the weights in the previous layer, and the gradients of the associated activation functions. After this step, the parameters are updated by using the calculated gradients.

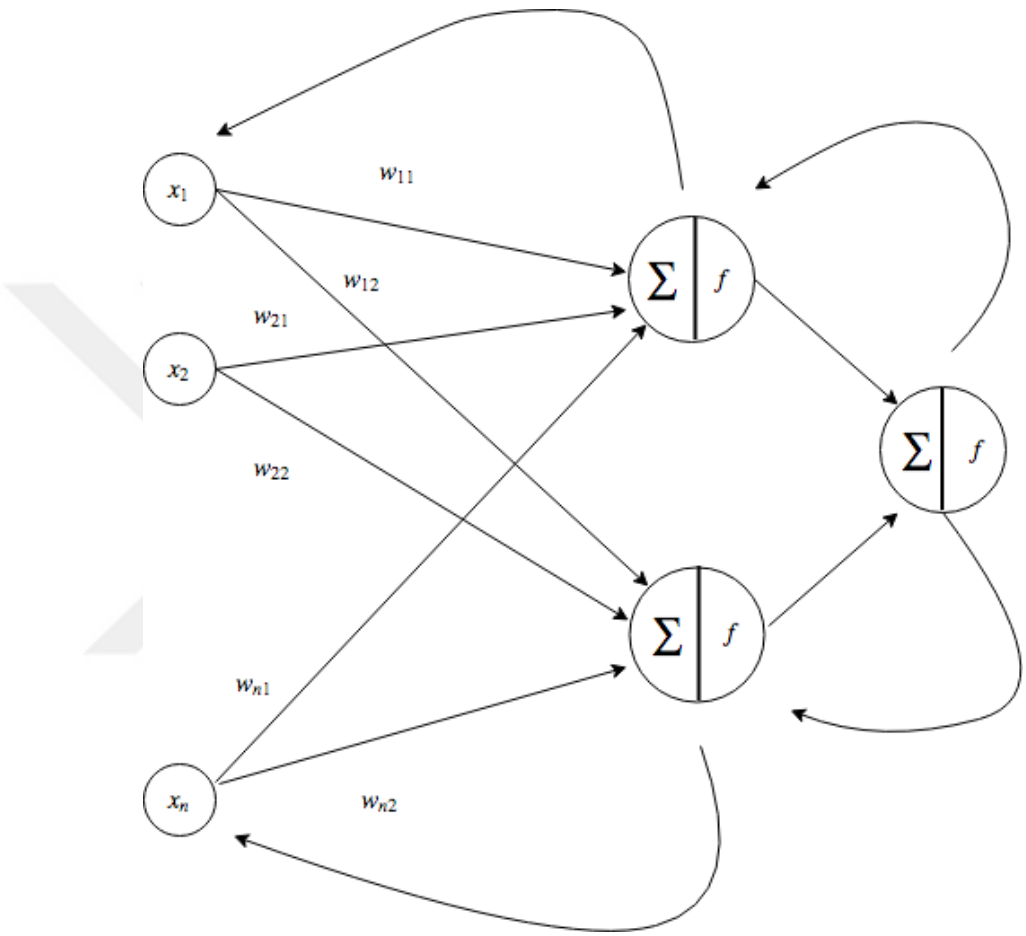

Figure 6.3. For given input vector x in a training data set of and target output vector t, the algorithm backpropagates the error by scaling it by the weights determined by the previous layer and the gradients of the associated activation functions [3].

The algorithm for a neural network with a single hidden layer (Figure 6.3) can be decomposed as in the following four steps [54].

(i) Feed-forward computation
(ii) Backpropagation to the output layer
(iii) Backpropagation to the hidden layer
(iv) Weight updates

The algorithm stops when the error is sufficiently small. The mean squared error function we use has the form:

$$E = \frac{1}{2}\sum_{j\in J}(O_j - t_j)^2$$

where t_j is the target value of node j, and O_j is the output value of node j which is obtained by the sum of the weighted input values passed through the activation function. The algorithm needs to find the best combination of weights to minimize the error. For this purpose we need to find the rate of change of the error function with respect to given connective weights.

6.5. Derivation

The cost function at the output layer (with index $k+1$):

$$E^{k+1} = \frac{1}{2}\sum_{j}(t_j - O_j^{k+1})^2 \tag{6.1}$$

where t_j is desired signals at the output layer, and O_j^{k+1} is the actual output at the output layer, and its output of a function γ whose variables include weights in *all* layers such that:

$$O_j^{k+1} = \gamma(w_1, w_2, \ldots, w_h, x, t) = f(S_j^{k+1}) \text{ where } S_j^{k+1} = \sum_i w_{ij}^k O_i^k.$$

Our task is to minimize the error E^{k+1} over the weight space of the network. To do this, we need the gradient information at each layer. The gradient computation depend on whether the nodes are on the *output layer*, or on the *hidden layer*. Nevertheless, we deploy a single gradient descent to minimize the cost function.

$$\Delta w = -\eta\frac{\partial E^{k+1}}{\partial w} \tag{6.2}$$

Since the equation above involves all the weights w, we can not compute the gradients directly because errors do not have readily available gradients at the hidden layers. But, in order to derive gradient equations we can use the *chain rule.*

We use the following notation:

k: Layer index
i: Neuron index of k^{th} layer
j: Neuron index of $(k+1)^{th}$ layer
w_{ij}^k : The weight between i^{th} neuron of k^{th} layer and j^{th} neuron of $(k+1)^{th}$ layer
S_i^k: Net input to the i^{th} neuron at the k^{th} layer
O_i^k: The output value of node i at the k^{th} layer
f: The activation function
t_j: The target value of node j

6.5.1. Output Layer

$$
\begin{aligned}
\frac{\partial E}{\partial \omega_{ij}^k} =& \frac{\partial E}{\partial S_j^{k+1}} \frac{\partial S_j^{k+1}}{\partial \omega_{ij}^k} \\
=& \frac{\partial E}{\partial O_j^{k+1}} \frac{\partial O_j^{k+1}}{\partial S_j^{k+1}} \frac{\partial S_j^{k+1}}{\partial \omega_{ij}^k} \\
=& (O_j^{k+1} - t_j) \frac{\partial f(S_j^{k+1})}{\partial S_j^{k+1}} \frac{\partial S_j^{k+1}}{\partial \omega_{ij}^k} \\
=& (O_j^{k+1} - t_j) f'(S_j^{k+1}) \frac{\partial \sum_i \omega_{ij}^k O_i^k}{\partial \omega_{ij}^k} \\
=& (O_j^{k+1} - t_j) f'(S_j^{k+1}) O_i^k \qquad (*) \\
\implies & \frac{\partial E}{\partial \omega_{ij}^k} = O_i^k \delta_j^{k+1} \quad where \quad \delta_j^{k+1} = (O_j^{k+1} - t_j) f'(S_j^{k+1})
\end{aligned}
$$

(*): In the second step the summation disappears because ω_{ij} is a specific weight connecting $i-th$ node to $j-th$ node from hidden and output layer respectively.

6.5.2. Hidden Layer

Assume that we are at an arbitrary layer k. We would like to generate the update rule for $(k+1)$, ∇w_{ij}^k. In this case layer $(k+1)$ is a hidden layer, and there is no explicit realization of the E^{k+1} in terms of w_{ij}^k. Let unit indices be i, j, l at layers k, $(k+1)$ and $(k+2)$ respectively.

$$
\begin{aligned}
\frac{\partial E^{k+1}}{\partial \omega_{ij}^k} &= \frac{\partial E^{k+1}}{\partial S_j^{k+1}} \frac{\partial S_j^{k+1}}{\partial \omega_{ij}^k} \\
&= \frac{\partial E^{k+1}}{\partial S_j^{k+1}} \frac{\partial}{\partial \omega_{ij}^k} \sum_i \omega_{ij}^k O_i^k \\
&= \frac{\partial E^{k+1}}{\partial O_j^{k+1}} \frac{\partial O_j^{k+1}}{\partial S_j^{k+1}} O_i^k \\
&= \frac{\partial E^{k+1}}{\partial O_j^{k+1}} f^{'}(S_j^{k+1}) O_i^k \\
&= \sum_l \frac{\partial E^{k+2}}{\partial S_l^{k+2}} \frac{\partial S_l^{k+2}}{\partial O_j^{k+1}} f^{'}(S_j^{k+1}) O_i^k \qquad (*) \\
&= \sum_l \frac{\partial E^{k+2}}{\partial S_l^{k+2}} \frac{\partial}{\partial O_j^{k+1}} \left(\sum_j w_{jl}^{k+1} O_j^{k+1} \right) f^{'}(S_j^{k+1}) O_i^k \\
&= \sum_l \frac{\partial E^{k+2}}{\partial S_l^{k+2}} w_{jl}^{k+1} f^{'}(S_j^{k+1}) O_i^k \\
&= f^{'}(S_j^{k+1}) O_i^k \sum_l \delta_l^{k+2} w_{jl}^{k+1} \\
&\Longrightarrow \frac{\partial E}{\partial \omega_{ij}} = O_i^k \delta_j^{k+1} \quad where \quad \delta_j^{k+1} = f^{'}(S_j^{k+1}) \sum_l \delta_l^{k+2} w_{jl}^{k+1}
\end{aligned}
$$

(*): In this step we use *total derivative* concept.

Combining the results, back propagation algorithm updates the weights iteratively by:

$$\omega^{p+1} = \omega^p - b\nabla_\omega E(\omega^p)$$

The update rule for all the weights (output weights and the hidden layer weights) is

$$\Delta w_{ij}^k = \eta \delta_j^{k+1} O_i^k = -\eta \frac{\partial E^{k+1}}{\partial w_{ij}^k} \tag{6.3}$$

Let $S_j^{k+1} = \sum_i w_{ij}^k O_i^k$.

(i) If we are at *output layer:*

$$\delta_j^{k+1} = (t_j - O_j^{k+1}) f'(S_j^{k+1}) \tag{6.4}$$

(ii) If we are at *hidden layer*, for index $k+1$:

$$\delta_j^{k+1} = \left(\sum_j \delta_l^{k+2} w_{jl}^{k+1} \right) f'(S_j^{k+1}) \tag{6.5}$$

Implementation of the algorithm is done with a backward recursion, hence the name. We start the algorithm at the output layer, and go back through the hidden layers all the way to the input layer. We propagate errors backward to obtain gradient information from hidden layers. Algorithm in Figure 6.4 shows pseudocode backpropagation algorithm using the batch gradient descent method.

6.5.3. Convergence

The total error function E^{k+1} for the MLP model in the weight space can not be easily determined since we have intermediate layers with non-linear activation functions. In particular, we do not know if it is convex. This means it may have many local minimum apart from the absolute minimum.

Initialize each w_{ij} in the network to a small random number
repeat
  for each sample (x, y) in examples **do**
    for each node i in the input layer **do**
      $O_i \leftarrow x_i$
    end for
    for each layer k from 2 to L **do**
      for each node j in layer k **do**
        $S_j \leftarrow \sum_i w_{ij} O_i$ // forward pass
        $O_j \leftarrow f(S_j)$
      end for
    end for
    for each node j in the output layer **do**
      $\delta[j] \leftarrow f'(S_j)(y_j - O_j)$ // backward pass (6.4)
    end for
    for each layer k from $L-1$ to 1 **do**
      for each node i in layer k **do**
        $\delta[i] \leftarrow f'(S_i)(\sum_j w_{ij}\delta[j])$ // backward pass continued (6.5)
      end for
    end for
  end for
  for each weight w_{ij} in network **do**
    $w_{ij} \leftarrow w_{ij} + \eta \times O_i \times \delta[j]$ // update network weights
  end for
until w_{ij}'s converge

Figure 6.4. Backpropagation algorithm using batch gradient descent.

Initialize each w_{ij} in the network to a small random number
repeat
  for each sample (x, y) in examples **do**
    for each node i in the input layer **do**
      $O_i \leftarrow x_i$
    end for
    for each layer k from 2 to L **do**
      for each node j in layer k **do**
        $S_j \leftarrow \sum_i w_{ij} O_i$ // forward pass
        $O_j \leftarrow f(S_j)$
      end for
    end for
    for each node j in the output layer **do**
      $\delta[j] \leftarrow f'(S_j)(y_j - O_j)$ // backward pass (6.4)
    end for
    for each layer k from $L-1$ to 1 **do**
      for each node i in layer k **do**
        $\delta[i] \leftarrow f'(S_i)(\sum_j w_{ij}\delta[j])$ // backward pass continued (6.5)
      end for
    end for
    for each weight w_{ij} in network **do**
      $w_{ij} \leftarrow w_{ij} + \eta \times O_i \times \delta[j]$ // update network weights
    end for
  end for
until w_{ij}'s converge

Figure 6.5. Backpropagation algorithm using stochastic gradient descent.

The convergence of the update rule given with BP algorithm is not guaranteed to converge to the global minimum. Error might get stable at a local minimum point. Thus the choice of learning parameters gain importance in the update rule.

6.6. Backpropagation with Stochastic Gradient Descent

In the stochastic gradient descent (SGD) algorithm uses one sample at each iteration to update the weights of the model depending on the error determined by the example instead of using the average of the errors of all examples at each iteration. The only algorithmic difference between SGD and the ordinary gradient descent is that each algorithm optimizes a different cost-function. The cost-function for gradient descent iterates over all training samples while the cost function for stochastic gradient descent only accounts for one training sample chosen at random. Algorithm in Figure 6.5 shows the pseudocode of this algorithm. The trade-off for this pair of algorithms is between correctness (gradient descent) versus speed (stochastic gradient descent.) However, SGD converges to the global minimum when the cost function is convex, or pseudo-convex.

6.7. Backpropagation with Recursive Least Squares

Due to its fast convergence rate, Scalero et. al. [4] proposed using the recursive least squares algorithm to speed up training process in the multilayer perceptron model. The proposed algorithm uses the sum of squared error between the actual and the desired inputs on the output layer. This objective function differs from the objective function used in the standard backpropagation algorithm which minimizes the sum of squared error between the actual and the desired outputs on the output layer. The reason behind this modification is to reduce the non-linear problem to a linear optimization problem in order to apply recursive least squares estimation. Non-linearity of the backpropagation algorithm come from the use of the non-linear activation functions of the network.

Given an input and desired output pair, the outputs from the output layer are determined using forward propagation of the input, and the desired summation outputs S_j, which are now inputs for the output layer, are calculated using the inverses of the activation functions on the desired outputs t_j. For the hidden layers, the summation outputs S_j^{k+1} and the inputs O_i^k of a node are estimated. Since $O_j^{k+1} = f(d_j^{k+1})$, it is sufficient to estimate the desired outputs d_j^{k+1} only.

The estimates for the summation outputs S_j^{k+1} and the inputs O_i^k will be denoted by $\hat{d}_j^{k+1}$ and $\hat{O}_i^k$, respectively. The desired summation outputs can be estimated using Equation (6.4) and Equation (6.5)

$$\hat{d}_j^{k+1} = S_j^{k+1} + \eta \cdot \delta_j^{k+1} \tag{6.6}$$

Since these estimates do not provide an exact weight vector w_{ij}^k at each node we would need to update the desired summation output estimates $\hat{d}_j^{k+1}$ according to the error produced at the output layer of the network. Repetition of this procedure for each training data points improves these estimates, and therefore, the weight vectors.

Training process of the network is starts with the propagation of the randomly initialized weight vectors through the network. Then we update the desired summation output estimates $\hat{d}_j^{k+1}$ using calculated errors which we refer as *deltas*. This process continued until the convergence is achieved. As in the case of stochastic gradient descent, this method allows for stochastic updates to be performed for each incoming training pattern.

6.7.1. Linear Equations

In this section we consider the system of linear equations that relates the weight vectors to the input and the desired summation output on a node. Every time the desired summation outputs $\hat{d}_p$ are estimated, the weights of the networks are updated.

In the following subsection we derive the system of linear equations considering only a single neuron. For simplicity we drop the estimation identifier, since whether we use the actual or estimate values are not our concern in this section.

6.7.2. Error Function

Our objective is to minimize the total mean-squared error E with respect to the summation outputs S_p which is given as

$$E = \sum_{p=1}^{m} (d_p - S_p)^2 \tag{6.7}$$

where d_p is the desired summation output and S_p is the actual summation output for the p^{th} training pattern, and m is the number of points in the training dataset. This objective function can be minimized by setting partial derivatives of E with respect to each weight to zero. The result will be a set of linear equations of the number of weights connecting to the neuron. Minimizing the error E with respect to a weight w_n produces

$$\frac{\partial E}{\partial w_n} = -2 \sum_{p=1}^{m} (d_p - S_p) \frac{\partial S_p}{\partial w_n} = 0 \tag{6.8}$$

The summation output S_p is given as:

$$S_p = \sum_{i=0}^{N} w_i O_{pi} \tag{6.9}$$

where N is the number of weight of the neuron. So, the partial derivative with respect to the specific weight w_n is calculated as:

$$\frac{\partial S_p}{\partial w_n} = O_{pn} \tag{6.10}$$

Substituting Equation (6.9) and Equation (6.10) to Equation (6.8) we get

$$\frac{\partial E}{\partial w_n} = -2\sum_{p=1}^{m}(d_p - \sum_{i=0}^{N} w_i O_{pi})O_{pn} = 0 \tag{6.11}$$

$$\sum_{p=1}^{m} d_p O_{pn} = \sum_{p=1}^{m}\sum_{i=0}^{N} w_i O_{pi} O_{pn} \tag{6.12}$$

Rewriting the right-hand side of Equation (6.12) as

$$\sum_{p=1}^{m}\sum_{i=0}^{N} w_i O_{pi} O_{pn} = \sum_{p=1}^{m} O_{pn} \sum_{i=0}^{N} w_i O_{pi} \tag{6.13}$$

and changing the right summation to a vector multiplication, provide

$$\sum_{p=1}^{m} O_{pn} w^T O_p = \sum_{p=1}^{m} O_{pn} O_p^T w \tag{6.14}$$

By defining

$$P = \sum_{p=1}^{m} O_p O_p^T \tag{6.15}$$

and

$$r = \sum_{p=1}^{m} d_p O_p \tag{6.16}$$

we can express Equation (6.12) in matrix form as

$$r = P \cdot w \tag{6.17}$$

This equation can be interpreted as the matrix P that corresponds to the correlation matrix of the training patterns, and the vector r as the cross correlation between the training patterns and their desired outputs.

Now, we have a system of linear equations, and the weight vector w in (6.17) can be expressed by solving the normal equations in Equation (6.17)

$$w = P^{-1} \cdot r \tag{6.18}$$

Equation (6.17) is derived on a single neuron. This procedure must be repeated for each neuron every time the weights of the network are updated.

Since the procedure requires all the training patterns to be propagated through the network, in the case of large training sets solving such matrix equations at each node might pose problem with the amount of computations required.

6.7.3. Stochastic Iteration

We can correct this deficiency by modifying the correlations in Equation (6.15) and Equation (6.16), and also selecting the training patterns randomly. We rewrite Equation (6.15) as

$$P(t) = \sum_{k=1}^{t} O(k)O(k)^T \tag{6.19}$$

and Equation (6.16) as

$$r(t) = \sum_{k=1}^{t} d(k)O(k) \tag{6.20}$$

Here t corresponds to the index of the current iteration. Now (6.18) takes the form of

$$w(t) = P^{-1}(t)r(t) \tag{6.21}$$

Equation (6.19) and Equation (6.20) produce estimates for the correlation matrix P and correlation vector r, except for a factor of $\frac{1}{t}$, and they improve with increasing t. Now, the weight at each node can be updated at every iteration as training patterns propagated through the network one by one to calculate these estimates.

Stochastic iterations in this form may increase the training time of the network. Because we obtain the estimates for each layer from the data we receive from the previous layer, and each iteration in some form contains information about older correlation estimates even after the network updated its weights. To cope with this problem, we will use a factor called *forgetting factor*.

6.7.4. Forgetting Factor

The forgetting factor λ allows the new data points to dominate while exponentially reducing the effect of the earlier correlation estimates. It is inserted into the correlation equations (6.19) and (6.20) as follows:

$$P(t) = \sum_{k=1}^{t} \lambda^{t-k} O(k) O^T(k) \tag{6.22}$$

$$r(t) = \sum_{k=1}^{t} \lambda^{t-k} d(k) O^T(k) \tag{6.23}$$

Now, we can write both Equation (6.22) and Equation (6.23) recursively as:

$$P(t) = \lambda P(t-1) + O(t)O^T(t) \tag{6.24}$$

$$r(t) = \lambda r(t-1) + d(t)O(t) \tag{6.25}$$

Although Equation (6.24) and Equation (6.25) are in recursive form, we need a recursive equation for the inverse auto-correlation matrix $P^{-1}(t)$.

6.7.5. Recursive Least Squares

The recursive equation for the inverse auto-correlation matrix $P^{-1}(t)$ is calculated as in Section 4.3.1

$$P_j^{-1}(t) = \lambda^{-1}\left(P_j^{-1}(t-1) - k_j(t)O_j^T P_j^{-1}(t-1)\right) \tag{6.26}$$

where for the $(k+1)^{th}$ layer

$$k_j(t) = \frac{P_j^{-1}(t-1)O_i(t)}{\lambda_j + O_i^T(t)P_j^{-1}(t-1)O_i(t)} \tag{6.27}$$

And the weight vectors in (6.21) are updated as

$$w_{ij}(t) = \begin{cases} w_{ij}(t-1) + k_j(t)(d_j - S_j) & \text{for the output layer} \\ w_{ij}(t-1) + k_j(t)\eta\delta_j(t) & \text{for the hidden layers} \end{cases} \tag{6.28}$$

where t is the iteration number, and w_{ij} is the weight connecting the i^{th} node in the k^{th} layer to the j^{th} node in the $(k+1)^{th}$ layer. Constants λ and η are the forgetting factor and the learning rate, respectively. $\delta_j(t)$'s are obtained from Equation (6.4) and Equation (6.5).

6.7.6. The Algorithm

The proposed algorithm the authors used in [4] to train the neural network using the recursive least squares back propagation and the *sigmoid activation function* can be summarized as follows:

(i) Initialize:

(a) Randomize all weights in the network.

(b) Set the forgetting factor λ, and learning rate η.

(c) Initialize the inverse of the auto-correlation matrix P_j^{-1} to the σI where I is the identity matrix, and σ is a large constant.

(ii) Feed forward a randomly selected training pattern (x_0, t_0) to the network, and evaluate the summation outputs for each layer

$$S_j^{k+1} = \sum_i w_{ij}^k O_i^k$$

and the sigmoid activation function outputs

$$O_j^{k+1} = f(S_j^{k+1}) = \frac{1}{1 + \exp(-S_j^{k+1})}$$

(iii) Calculate the Kalman gain $K_j(t)$ for each node with index j in the $(k+1)^{th}$ layer to update P_j^{-1} where O_i is the output of the k^{th} layer with the node index i:

$$K_j(t) = \frac{P_j^{-1}(t-1)O_i(t)}{\lambda_j + O_i^T(t)P_j^{-1}(t-1)O_i(t)}$$

$$P_j^{-1}(t) = \lambda_j^{-1}\left(P_j^{-1}(t-1) - K_j(t)O_i^T(t)P_j^{-1}\right)$$

(iv) Backpropagate the errors through the network

(a) For output layer $k+1$ the delta terms are

$$\delta_j^{k+1} = f'(S_j^{k+1})(t_j - O_j^{k+1})$$

(b) And the delta terms for the interior hidden layer with index $k+1$ is

$$\delta_j^{k+1} = f'(S_j^{k+1}) \sum_l (\delta_l^{k+2} \cdot w_{jl}^{k+1})$$

Here the derivative of the sigmoid activation function $f'(S_j^{k+1})$ is given by

$$f'(S_j^{k+1}) = f(S_j^{k+1})(1 - f(S_j^{k+1}))$$

(v) Adjust the weights in each layer using

$$w_{ij}^k(t) = \begin{cases} w_{ij}^k(t-1) + K_j(t)(d_j^{k+1} - S_j^{k+1}) & \text{for output layers} \\ w_{ij}^k(t-1) + K_j(t)(\eta\delta_j^{k+1}(t)) & \text{for hidden layers} \end{cases}$$

where the desired summation output at the $(k+1)^{th}$ output layer calculated by using the inverse function

$$d_j^{k+1} = \ln\left(\frac{t_j}{1-t_j}\right) \tag{6.29}$$

for every j^{th} node in the output layer.

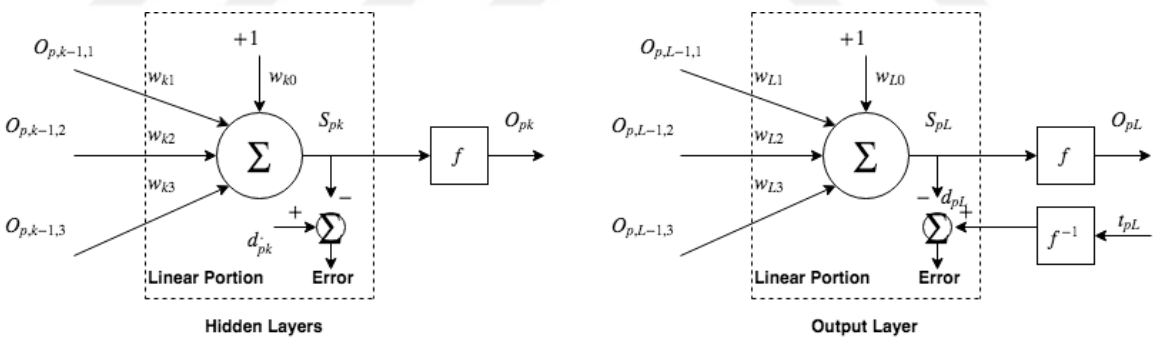

Figure 6.6. Linear and error portion of the neurons in the hidden and output layers [4].

Let the number of the network parameters be M. The computational complexity of computing the overall training set error and the gradients is $\mathcal{O}(M^2)$ for standard backpropagation algorithm. In the backpropagation algorithm using RLS, the matrix inversion has complexity $\mathcal{O}(D^3)$ where D is the dimension of the output layer. For a single output neural networks, this inversion operation reduces to scalar division.

The weight and auto-correlation matrix updates requires $\mathcal{O}(DM^2)$, so that total computational complexity is $\mathcal{O}(DM^2 + D^3)$. Even though training with RLS algorithm requires more computational power, the adaptive Kalman gain parameter increases the convergence rate of the algorithm [4].

7. EXPERIMENTS

7.1. Online K-means

7.1.1. Data

The proposed algorithms in Chapter 5 are tested on different data sets including a synthetically generated data set and data from UCI machine learning repository [55] and LibSvm [56]. We also used same data sets as proposed in [6] to compare our results.

For synthetically generated data we used `make_blobs` module from python's scikit-learn library [57]. The available parameters are as follows:

(i) `n_samples`: The total number of points equally divided among clusters.
(ii) `n_features` : The number of dimensions of each point.
(iii) `centers`: The number of centers to generate, or the fixed center locations.
(iv) `cluster_std` : The standard deviation of the clusters.

We refer our synthetically generated data with respect to their chosen parameters as shown in Table 7.1

7.1.2. Implementation

We listed the source code for our implementations of the algorithms we investigated in Chapter 5 in Appendix A.1.

Table 7.1. Description of the synthetically generated dataset.

Name	`n_samples`	`n_features`	`centers`	`cluster_std`
D1	$2 \cdot 10^3$	2	20	0.6
D2	$2 \cdot 10^3$	2	40	0.6
D3	$5 \cdot 10^4$	2	20	0.6
D4	$5 \cdot 10^4$	2	40	0.6
D5	$5 \cdot 10^4$	2	60	0.6
D6	$5 \cdot 10^4$	2	120	0.6

Table 7.2. Description of the datasets from UCI machine learning repository.

Name	`n_samples`	`n_features`	`centers`
Iris	150	4	3
Bridge-binarized	4096	16	256
House 5 bits	34112	3	256
magic04	19020	10	40

7.1.3. Evaluation Metrics

Evaluating the performance of a clustering algorithm requires more than measuring errors, calculating precision and recall for the proposed classification algorithm. One of the difficulties come from the fact that it is difficult to write a one-to-one matching between the original labels, and the new cluster sets. But, we can use the original cluster labels as a *discrete similarity measure*. For these reasons, we used *homogeneity*, and *completeness* metrics to measure similarity within each cluster. For datasets with unknown labels, batch k-means results is used as a baseline.

Given a data set with N points, let $S = \{s_i\}_{i=1}^{n}$ to represent a set of classes, and $W = \{w_j\}_{j=1}^{m}$ represent a set of clusters of these data points.

7.1.3.1. Homogeneity. Perfect homogeneity of a clustering is achieved when all points in that cluster are instances of a single class. Clusters that include samples from different classes are not considered as homogeneous. Homogeneity score is between 0.0 and 1.0. 1.0 score corresponds to perfect homogeneity [58].

$$h = 1 - \frac{H(S|W)}{H(S)}$$

Here $H(S|W)$ is the entropy of classes S conditioned on cluster assignments W, and is defined by:

$$H(S|W) = -\sum_{s=1}^{n}\sum_{w=1}^{m}\frac{a_{s,w}}{N}\dot{\log}\left(\frac{a_{s,w}}{a_w}\right) \tag{7.1}$$

$H(S)$ is the entropy of the classes and is given by:

$$H(S) = -\sum_{s=1}^{n}\frac{a_s}{N}\dot{\log}\left(\frac{a_s}{N}\right) \tag{7.2}$$

In the above equations, a_s and a_w represent the number of data points belonging to class s and assigned to cluster w respectively. The number of data points from class s assigned to cluster w is given by $a_{s,w}$.

7.1.3.2. Completeness. Completeness is satisfied if all points of a given class are grouped in the same cluster. Completeness score varies between 0.0 and 1.0, where 1.0 corresponds to perfect completeness [58].

$$c = 1 - \frac{H(W|S)}{H(W)}$$

where $H(W|S)$ is the entropy of clusters W conditioned on classes S and $H(W)$ is the entropy of clusters W, and symmetrically defined as in Equation (7.1) and Equation (7.2).

7.1.3.3. V-Measure. A clustering that splits classes into more than one clusters can also be homogeneous. On the other hand, a clustering that assign two different classes to the same cluster is still considered as complete. This means, any homogeneity or completeness measure by themselves cannot be enough to evaluate the success of a proposed clustering algorithm. We are going to combine these measure under a single measure called *the V-measure*. V-measure score is calculated by the harmonic mean between homogeneity and completeness [58].

$$v = 2 \cdot \frac{h \cdot c}{h + c}$$

7.1.3.4. Number of Clusters Ratio. This measure is the ratio of the number of clusters calculated by the algorithm and the desired number of clusters. Recall that, in online clustering algorithms the number of clusters is adjusted as new data points arrive.

Hence, this metric is used for online clustering algorithms where the algorithm takes a user-defined input for the target, i.e. the desired number of clusters, but may yield a larger number of clusters. A smaller ratio means the number of calculated clusters is closer to the target number of clusters.

7.1.4. Results

We used homogeneity and completeness as the main metrics to evaluate the success of a clustering algorithm. In addition, we also compared algorithms in terms of their time complexity and clustering cost. For the fully online k-means, we also used the ratio of k_{target} and k_{actual} as a metric [6].

In online k-means we use the stochastic gradient descent to update cluster centroids, but the number of clusters must be specified a priori. For the ART method one has to choose a *vigilance* threshold. This threshold allows for creation of new clusters as new instances of data points are fed to the algorithm. In the ART method, choosing a convenient value for the *vigilance* threshold is the major drawback. With a large vigilance value, even points that are too far from each other might fall in the same cluster. On the other hand, a small vigilance value may cause generation of too many clusters. In our experiments we observed that ART algorithm is too sensitive to vigilance parameter. However, this sensitivity does not impact the homogeneity or completeness metrics visibly.

The fully online k-means algorithm proposes using the same objective function as the k-means clustering algorithm, but adds a new term for the cost of opening a new cluster for an incoming data which comes from the *online facility location problem*. The cost of opening a new cluster changes with the number of clusters created. Even though some of the initially created cluster centroids may fail to attract points arriving later in the stream Liberty *et. al.* (2015) in [6] showed that algorithm in Figure 5.9 creates only slightly more than the desired number of clusters. However, results of our experiments on this algorithm are not on par with the results obtained in [6].

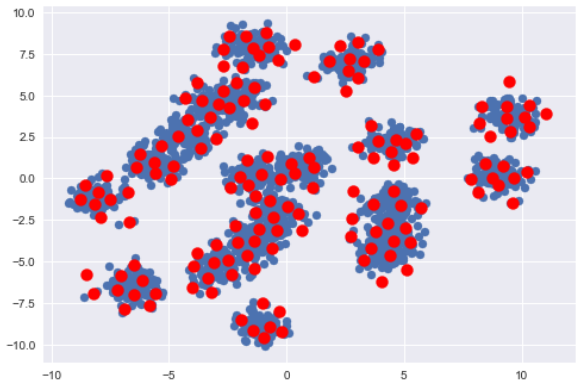

Figure 7.1. Clustering result of ART algorithm on D1.

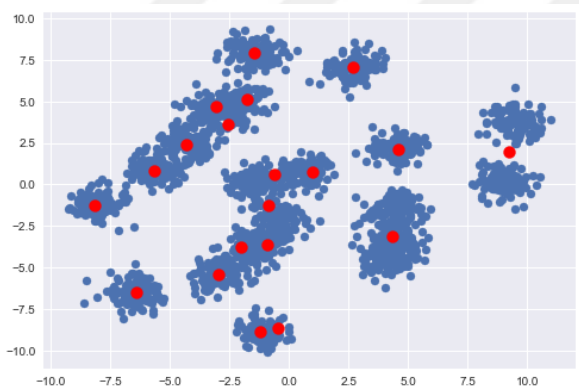

Figure 7.2. Clustering result of online k-means algorithm on D1.

In the same paper the authors also suggest some practical adjustments on the fully online k-means algorithm. With this modified online algorithm, initial k value is set to $(k_{target} - 15)/5$. The authors also modify the hyper-parameter f from 2 to 10. This modified version of the online algorithm in our implementation shows results similar with their experiments. However, on our datasets this algorithm creates less than k centers, and the ratio between k_{actual} and k_{target} ranges between 0.5 and 0.75. Figure 7.3 shows clustering on the data set $D1$. It can be seen that interior clusters are successfully detected by the algorithm, and outer clusters did not form correctly. Since these algorithms do not update the cluster centroids, the centroids are not located in the cluster centers.

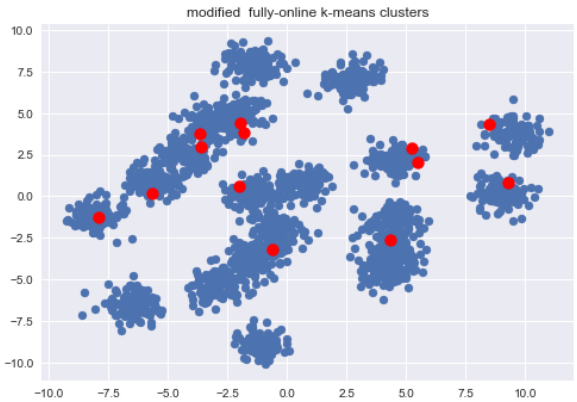

Figure 7.3. Clustering result of modified online algorithm on D1.

Table 7.3. The results of online k-means experiments.

Model	**Data**	k_0	η	ρ	**Homogeneity**	**Completeness**	**V-measure**	**CPU**	**Ratio**
Online k-means	Iris-normalized	3	0.3	-	0.913	0.915	0.914	0.005	1
Online k-means	Bridge-binarized	256	0.3	-	1	1	1	0.196	1
Online k-means	House-5 bits	256	0.3	-	0.814	0.849	0.831	0.0451	1
Online k-means	magic04	40	0.3	-	0.707	0.728	0.717	0.297	1
ART	Iris-normalized	-	0.3	0.02	1	1	1	0.009	1
ART	Iris-normalized	-	0.3	0.01	0.930	0.931	0.931	0.0117	1.6
ART	Iris-normalized	-	0.3	0.07	0.579	1.000	0.734	0.010	0.6
ART	Bridge-binarized	-	0.3	4	00.601	0.658	0.628	0.126	1.16
ART	Bridge-binarized	-	0.3	0.7	1	1	1	0.4933	10.98
ART	Bridge-binarized	-	0.3	1.2	0.866	0.876	0.871	0.310	6.48
ART	House-5 bits	-	0.3	100	0.843	0.864	0.854	0.073	2.42
ART	House-5 bits	-	0.3	1.6	0.940	0.996	0.967	0.119	7.17
ART	magic04	-	0.3	10000	0.394	0.433	0.413	0.541	6.8
Fully-online	Iris-normalized	3	-	-	-	-	-	0.010	48.6
Fully-online	Bridge-binarized	256	-	-	-	-	-	0.456	11.9
Fully-online	House-5 bits	256	-	-	-	-		0.262	8.17
Fully-online	magic04	40	-	-	-	-	-	12.62	465.725
Modified-online	Iris-normalized	3	-	-	-	-	-	0.008	3.3
Modified-online	Bridge-binarized	256	-	-	-	-	-	0.06918	0.25
Modified-online	House-5 bits	256	-	-	-	-	-	0.043	0.28
Modified-online	magic04	40	-	-	-	-	-	0.314	0.52

Table 7.4. The results of online k-means experiments on synthetically generated data.

Model	**Data**	k_0	η	ρ	**Homogeneity**	**Completeness**	**V-measure**	**CPU**	**Ratio**
Online k-means	D1	20	0.3	-	0.839	0.916	0.876	0.039	1
Online k-means	D2	40	0.3	-	0.855	0.905	0.879	0.074	1
Online k-means	D3	20	0.3	-	0.835	0.938	0.883	1.638	1
Online k-means	D4	40	0.3	-	0.822	0.879	0.849	1.791	1
Online k-means	D5	60	0.3	-	0.793	0.848	0.820	1.813	1
Online k-means	D6	120	0.3	-	0.751	0.798	0.774	2.035	1
ART	D1	-	0.3	0.6	0.918	0.921	0.920	0.117	7.75
ART	D2	-	0.3	0.6	0.891	0.893	0.892	0.133	5.7
ART	D3	-	0.3	0.6	0.930	0.930	0.930	3.874	20.35
ART	D4	-	0.3	0.6	0.833	0.834	0.833	4.234	11.8
ART	D5	-	0.3	0.6	0.801	0.804	0.803	4.65	9.2
ART	D6	-	0.3	0.6	0.755	0.761	0.758	4.92	5.14
Fully-online	D1	20	-	-	-	-	-	0.093	101.5
Fully-online	D2	40	-	-	-	-	-	0.218	51
Fully-online	D3	20	-	-	-	-	-	28.99	710.95
Fully-online	D4	40	-	-	-	-	-	48.67	740.75
Fully-online	D5	60	-	-	-	-	-	60.09	783.46
Fully-online	D6	120	-	-	-	-	-	60.09	417.675
Modified-online	D1	20	-	-	-	-	-	0.041	0.65
Modified-online	D2	40	-	-	-	-	-	0.074	0.5
Modified-online	D3	20	-	-	-	-	-	1.812	0.7
Modified-online	D4	40	-	-	-	-	-	1.829	0.72
Modified-online	D5	60	-	-	-	-	-	1.868	0.63
Modified-online	D6	120	-	-	-	-	-	1.944	0.625

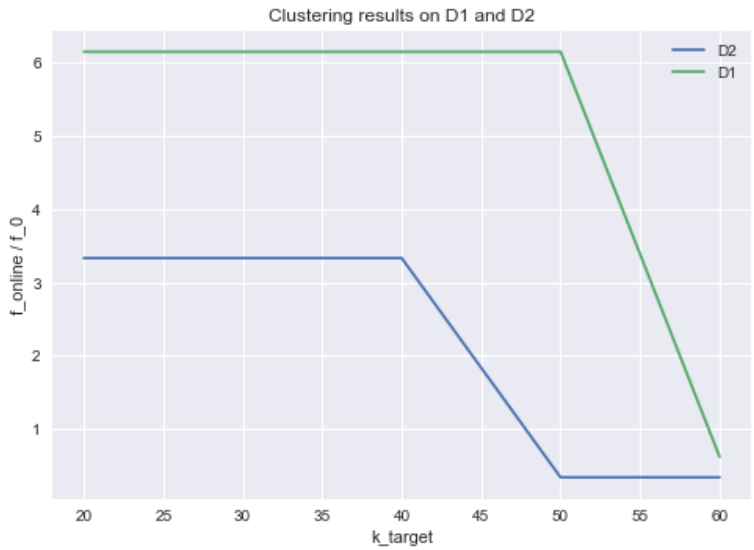

Figure 7.4. Normalized facility cost of the modified online k-means algorithm decreases as k_{target} increases.

7.2. Online Multilayer Perceptron

7.2.1. Data

We tested our implementations of the algorithms we investigated in Chapter 6 first on XOR and HTRU_2 data set [59]. HTRU_2 dataset consists of approximately 18 thousand datapoints each with 9 numerical features. Data comes from pulsar candidates collected during the High Time Resolution Universe Survey [60]. We also used a synthetically generated data set. For the data generation we used `make_classification` function in `datasets` module from python's scikit-learn library [57] which has parameters as follows:

(i) `n_samples` : The total number of points.
(ii) `n_features` : The number of dimensions of each point.
(iii) `n_classes` : Desired number of classes.
(iv) `random_state` : Random seed.

Table 7.5. Description of the datasets for classification.

Name	n_samples	n_features	n_classes
XOR	4	2	2
HTRU_2	17898	9	2
DD1	2000	2	2
DD2	5000	10	2
DD3	2000	10	2
DD4	5000	20	2
DD5	5000	4	2
DD6	5000	40	2

7.2.2. Implementation

We listed the source code written in Python for our implementations of the algorithms we investigated in Chapter 6 in Appendix A.2 In the algorithms we use the *sigmoid activation function* whose derivative is

$$f'(x) = \left(\frac{1}{1+e^{-x}}\right)\left(1 - \frac{1}{1+e^{-x}}\right) = f(x)(1-f(x)) \tag{7.3}$$

7.2.3. Results

We evaluated the multilayer perceptron model using stochastic gradient descent and recursive least squares backpropagation algorithms for two-class classification tasks. The results are based on accuracy, the number of iterations, training time, and mean squared error (MSE). Training and test sets are randomly sampled, and with a ratio 0.20. Then %75 of the training set is randomly chosen to be the actual training set and the remaining to be the validation set. We selected the parameter values of the models which provide the minimum cost on the validation set with minimum training iterations needed. During testing, we used 0.5 as classification probability threshold.

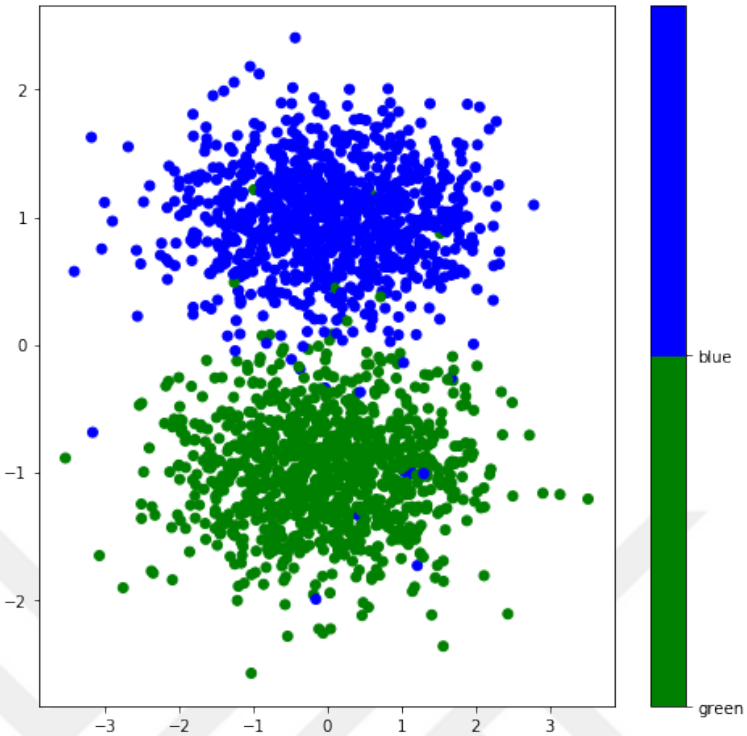

Figure 7.5. Visualization of the data points in DD1.

For HTRU_2 data set [59] we did our experiment fully online with only one epoch. For this data set we also added a momentum term [61] for the SGD algorithm with a factor 0.7. Table 7.6 shows our results.

Even though computational complexity of the RLS algorithm is higher than the SGD algorithm, in our experiments we observed that RLS has faster convergence speed than SGD. Figure 7.8 and Figure 7.7 shows cost values over epochs for XOR data set. These results suggest that the adaptive coefficient (Kalman gain) of the RLS algorithm makes the training algorithm less prone to get stuck in a local minimum. On the other hand, we observed that on the HTRU_2 data set which is trained in a fully online manner, SGD algorithm performed slightly better than RLS algorithm. This suggests that the momentum term in the SGD algorithm makes it more effective than the RLS algorithm is the online case.

Table 7.6. The results of classification with online MLP training on synthetically generated data.

Model	**Data**	η	λ	**Maximum Accuracy**	**Iterations**	**CPU (Training)**	**Training Cost**	**Test Cost**
SGD_backprop	DD1	0.1	-	0.9833	5	2.6529	59.02326	27.92
SGD_backprop	DD2	0.1	-	0.9884	8	7.20573	199.808	84.757
SGD_backprop	DD3	0.1	-	0.9075	80	30.6813	48.9132	23.22045
SGD_backprop	DD4	0.1	-	0.95	20	20.7066	118.187	59.598
SGD_backprop	DD5	0.1	-	0.938	10	8.6474	117.408	58.812
SGD_backprop	DD6	0.1	-	0.817	20	79.55414	286.7083824	137.8
RLS_backprop	DD1	0.1	0.95	0.9833	2	0.8417	15.18359	5.7204
RLS_backprop	DD2	0.01	0.95	0.9848	2	2.14675	33.9065	15.636
RLS_backprop	DD3	0.1	0.95	0.9045	4	2.33173	50.9307	22.5615
RLS_backprop	DD4	0.1	0.95	0.9606	3	4.1198	44.0093	24.6517
RLS_backprop	DD5	0.1	0.90	0.95151	2	1.87012	100.212	36.9373
RLS_backprop	DD6	0.1	0.85	0.8951	5	22.23	142.4299	63.4879

Table 7.7. The results of classification on XOR with online multilayer perceptron models experiments.

Model	**Data**	η	λ	**Network size**	**Iterations**	**CPU (Training)**	**Test Cost**	**MSE Threshold**
SGD_backprop	XOR	0.9	-	(2,8,1)	2527	1.7387	0.0006	1e-4
SGD_backprop	XOR	0.7	-	(2,8,8,1)	5935	3.5409	0.0005	1e-4
SGD_backprop	XOR	0.7	-	(2,8,8,1)	503	0.41977	0.0793	1e-2
RLS_backprop	XOR	0.7	0.3	(2,8,1)	9	0.13543	0.940	1e-4
RLS_backprop	XOR	0.9	0.1	(2,8,8,1)	3	0.10729	0.96316	1e-4
RLS_backprop	XOR	0.9	0.3	(2,8,8,1)	2	0.10779	0.95	1e-2
RLS_backprop	XOR	0.5	0.8	(2,8,8,1)	876	1.1520	1.4596279930479618e-07	1e-10

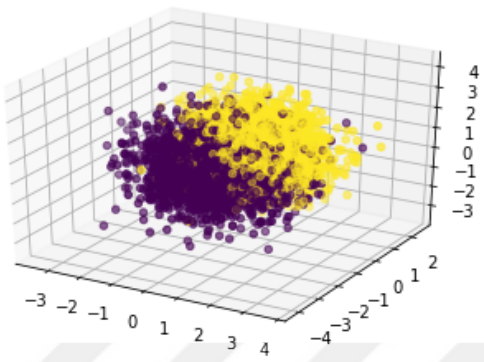

Figure 7.6. Data points in DD5.

Table 7.8. The results of fully online training with online MLP models on HTRU_2 data set.

Model	**Data**	η	λ	**Network size**	**Accuracy**	**CPU (Training)**	**Training Cost**	**Test Cost**
SGD_backprop	HTRU_2	0.6	-	(8,4,1)	0.96935	2.96438	326.245631588	133.34895
SGD_backprop	HTRU_2	0.5	-	(8,10,1)	0.975	2.90968	210.0264	80.6184
SGD_backprop	HTRU_2	0.8	-	(8,4,4,1)	0.978	2.50676	258.0811	55.6441
RLS_backprop	HTRU_2	0.1	0.85	(8,4,1)	0.91	2.60155	483.15869	226.14187
RLS_backprop	HTRU_2	0.5	0.9	(8,10,1)	0.9649	2.73465	557.4692	251.13751
RLS_backprop	HTRU_2	0.7	0.85	(8,4,4,1)	0.9673	3.45767	377.92	75.608

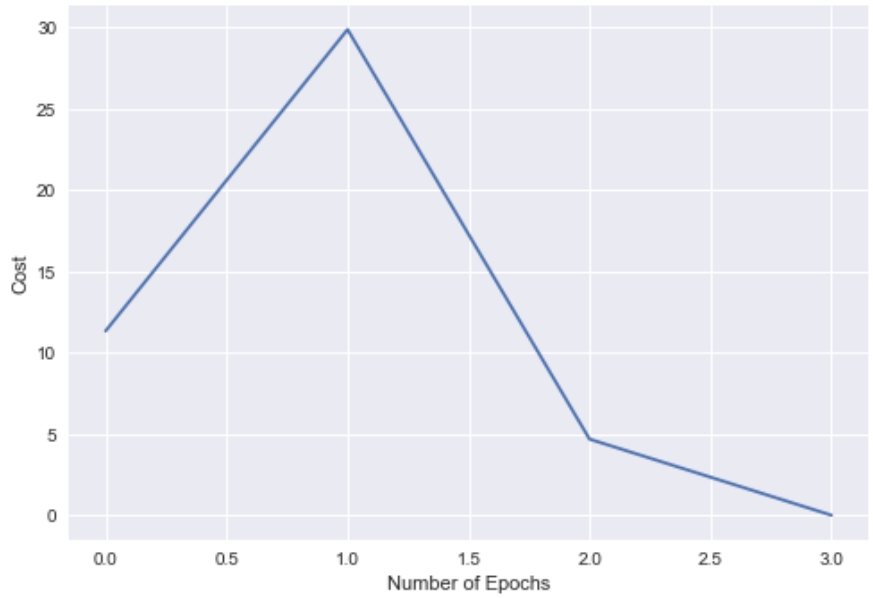

Figure 7.7. The cost changes over the number of epoch of the backpropagation algorithm using RLS on XOR.

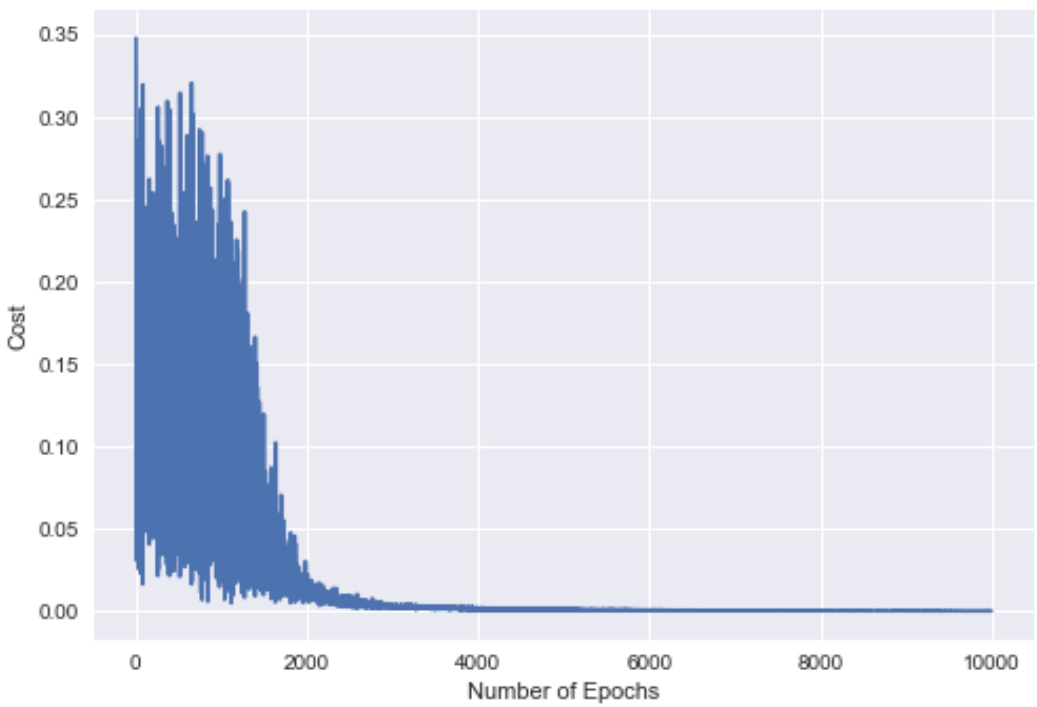

Figure 7.8. The cost changes over the number of epoch of the backpropagation algorithm using SGD on XOR.

8. CONCLUSION

This thesis is written as a survey of online and stochastic methods used in machine learning algorithms. Along with standard machine learning algorithms, we investigated the k-means algorithm and multilayer perceptron model in detail from this perspective. We wrote out own implementations (the python code for these experiments are listed in the Appendix) and did numerical experiments to compare our implementations of these different algorithms. The results of these experiments are analyzed in Chapter 7. We hope that this thesis gives a good starting point to any researcher who would like to learn more about the use of online and stochastic methods in machine learning algorithms and neural networks.

REFERENCES

1. Silva, J., E. Faria, R. Barros, E. Hruschka, A. Carvalho and J. Gama, "Data Stream Clustering: A Survey", *ACM Comput. Surv.*, Vol. 46, No. 1, pp. 13:1–13:31, Jul. 2013.

2. Alpaydin, E., *Introduction to machine learning*, The MIT Press, 2004.

3. Zhu, Y. and Y. Xiong, "Toward Data Science", *Data Science Journal*, 14, p. 8, 2015.

4. Scalero, R. and N. Tepedelenlioglu, "A fast new algorithm for training feedforward neural networks", *IEEE Transactions on signal processing*, Vol. 40, No. 1, pp. 202–210, 1992.

5. Gama, J., *Knowledge discovery from data streams*, Chapman & Hall/CRC Data Mining and Knowledge Discovery Series, CRC Press, Boca Raton, FL, 2010, with a foreword by André Ponce de Leon Ferreira de Carvalho.

6. Liberty, E., R. Sriharsha and M. Sviridenko, "An algorithm for online k-means clustering", *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 81–89, SIAM, 2016.

7. Vapnik, V., *Estimation of dependences based on empirical data*, Springer Series in Statistics, Springer-Verlag, New York-Berlin, 1982, translated from the Russian by Samuel Kotz.

8. Boyd, S. and L. Vandenberghe, *Convex optimization*, Cambridge University Press, Cambridge, 2004.

9. Robbins, H. and S. Monro, "A stochastic approximation method", *Ann. Math. Statistics*, Vol. 22, pp. 400–407, 1951.

10. Alippi, C., "Learning in nonstationary and evolving environments", *Intelligence for Embedded Systems*, pp. 211–247, Springer, 2014.

11. Wu, C.-X. and Q. He, "Combining multiple support vector machine classifiers", *Dyn. Contin. Discrete Impuls. Syst. Ser. A Math. Anal.*, Vol. 14, No. Part 1, suppl., pp. 222–226, 2007.

12. Biehl, M., A. Ghosh and B. Hammer, "Dynamics and generalization ability of LVQ algorithms", *J. Mach. Learn. Res.*, Vol. 8, pp. 323–360, 2007.

13. Saarinen, J. and T. Kohonen, "Self-organized formation of colour maps in a model cortex.", *Perception*, Vol. 14, No. 6, pp. 711–9, 1985.

14. Moody, J. and C. Darken, "Fast learning in networks of locally-tuned processing units", *Neural computation*, Vol. 1, No. 2, pp. 281–294, 1989.

15. Runkler, T., *Data Analytics: Models and Algorithms for Intelligent Data Analysis*, Springer, 2012.

16. Li, P., X. Wu, X. Hu, Q. Liang and Y. Gao, "A random decision tree ensemble for mining concept drifts from noisy data streams", *Applied Artificial Intelligence*, Vol. 24, No. 7, pp. 680–710, 2010.

17. Cortes, C. and V. Vapnik, "Support-vector networks", *Machine learning*, Vol. 20, No. 3, pp. 273–297, 1995.

18. Rosenblatt, F., *The perceptron: A theory of statistical separability in cognitive systems*, Cornell Aeronautical Laboratory, Inc., Rep. No. VG-1196-G-1. U.S. Department of Commerce, Office of Technical Services, PB 151247, 1958.

19. Widrow, B., *An Adaptive "ADALINE" Neuron Using Chemical "Memistors"*, Tech. rep., Solid-State Electronics Laboratory, Stanford Electronics Laboratories, Stanford University, Stanford, California, 1960.

20. Bottou, L., "Stochastic gradient descent tricks", *Neural networks: Tricks of the trade*, pp. 421–436, Springer, 2012.

21. Altman, N. S., "An introduction to kernel and nearest-neighbor nonparametric regression", *Amer. Statist.*, Vol. 46, No. 3, pp. 175–185, 1992.

22. Woodbury, M. A., *Inverting modified matrices*, Statistical Research Group, Memo. Rep. no. 42, Princeton University, Princeton, N. J., 1950.

23. Haykin, S., *Adaptive filter theory (ise)*, Prentice Hall, New York, 2003.

24. Domingos, P. and G. Hulten, "Mining high-speed data streams", *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 71–80, ACM, 2000.

25. Vesanto, J. and E. Alhoniemi, "Clustering of the self-organizing map", *Neural Networks, IEEE Transactions on*, Vol. 11, No. 3, pp. 586–600, 2002.

26. Grossberg, S. (Editor), *Neural networks and natural intelligence*, A Bradford Book, MIT Press, Cambridge, MA, 1988.

27. Hagan, M., H. Demuth, M. Beale and et.al., *Neural network design*, Pws Pub. Boston, 1996.

28. Halkidi, M., D. Papadopoulos, V. Kalogeraki and D. Gunopulos, "Resilient and energy efficient tracking in sensor networks", *International Journal of Wireless and Mobile Computing*, Vol. 1, No. 2, pp. 87–100, 2006.

29. Abadi, D., S. Madden and W. Lindner, "Reed: Robust, efficient filtering and event detection in sensor networks", *Proceedings of the 31st international conference on Very large data bases*, pp. 769–780, 2005.

30. Guestrin, C., P. Bodik, R. Thibaux, M. Paskin and S. Madden, "Distributed regression: an efficient framework for modeling sensor network data", *Proceedings*

of the 3rd international symposium on Information processing in sensor networks, pp. 1–10, ACM, 2004.

31. Hammad, M. A., "Optimizing in-order execution of continuous queries over streamed sensor data", *In Proc. Int. Conf. on Scientific and Statistical Database Management (SSDBM)*, pp. 143–146, 2005.

32. Deshpande, A., C. Guestrin, W. Hong and S. Madden, "Exploiting correlated attributes in acquisitional query processing", *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pp. 143–154, IEEE, 2005.

33. Madden, S., M. Shah, J. Hellerstein and V. Raman, "Continuously adaptive continuous queries over streams", *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pp. 49–60, ACM, 2002.

34. Ali, M., M. Mokbel, W. Aref and I. Kamel, "Detection and Tracking of Discrete Phenomena in Sensor-Network Databases.", *SSDBM*, pp. 163–172, 2005.

35. Manku, G. and R. Motwani, "Approximate frequency counts over data streams", *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pp. 346–357, Elsevier, 2002.

36. Metwally, A., D. Agrawal and A. El Abbadi, "Duplicate detection in click streams", *Proceedings of the 14th international conference on World Wide Web*, pp. 12–21, ACM, 2005.

37. Jiang, N. and L. Gruenwald, "Research issues in data stream association rule mining", *ACM Sigmod Record*, Vol. 35, No. 1, pp. 14–19, 2006.

38. Aggarwal, C. and C. Reddy, *Data clustering: algorithms and applications*, CRC press, 2013.

39. Aggarwal, C., J. Han, J. Wang and P. Yu, "A framework for clustering evolving

data streams", *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pp. 81–92, 2003.

40. Ester, M., H.-P. Kriegel, J. Sander and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise.", *KDD*, Vol. 96, pp. 226–231, 1996.

41. MacQueen, J., "Some methods for classification and analysis of multivariate observations", *Proc. Fifth Berkeley Sympos. Math. Statist. and Probability (Berkeley, Calif., 1965/66)*, pp. Vol. I: Statistics, pp. 281–297, Univ. California Press, Berkeley, Calif., 1967.

42. Lloyd, S. P., "Least squares quantization in PCM", *IEEE Trans. Inform. Theory*, Vol. 28, No. 2, pp. 129–137, 1982.

43. Ng, H., S. Ong, K. Foong, P. Goh and W. Nowinski, "Medical image segmentation using k-means clustering and improved watershed algorithm", *2006 IEEE Southwest Symposium on Image Analysis and Interpretation*, pp. 61–65, 2006.

44. Zade, J., G. Bamnote and P. Agrawal, "Text Document Clustering Using K-means Algorithm with Its Analysis and Implementation", *Imperial Journal of Interdisciplinary Research*, Vol. 3, No. 2, 2017.

45. Coates, A., H. Lee and A. Ng., "An Analysis of Single-Layer Networks in Unsupervised Feature Learning", *In Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2011.

46. Frias-Martinez, V., V. Soto, A. Sánchez and E. Frias-Martinez, "Consensus clustering for urban land use analysis using cell phone network data", *International Journal of Ad Hoc and Ubiquitous Computing*, Vol. 17, No. 1, pp. 39–58, 2014.

47. Sasikumar, P. and S. Khara, "K-Means Clustering in Wireless Sensor Networks", *Proceedings of the 2012 Fourth International Conference on Computational Intel-*

ligence and Communication Networks, CICN '12, pp. 140–144, IEEE Computer Society, Washington, DC, USA, 2012.

48. Mahajan, M., P. Nimbhorkar and K. Varadarajan, "The planar k-means problem is NP-hard", *International Workshop on Algorithms and Computation*, pp. 274–285, Springer, 2009.

49. Hartigan, J. and M. Wong, "Algorithm AS 136: A k-means clustering algorithm", *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, Vol. 28, No. 1, pp. 100–108, 1979.

50. Carpenter, G. and S. Grossberg, *Adaptive resonance theory*, Springer, 2010.

51. Meyerson, A., "Online facility location", *42nd IEEE Symposium on Foundations of Computer Science (Las Vegas, NV, 2001)*, pp. 426–431, IEEE Computer Soc., Los Alamitos, CA, 2001.

52. Dadhich, D. and A. Sharma, "Improved Online k-Means Algorithm", *13th International Conference on Recent trends in Engineering Science and Management*, ICRTESM '18, pp. 353–362, 2018.

53. Barron, A., "Approximation and estimation bounds for artificial neural networks", *Machine learning*, Vol. 14, No. 1, pp. 115–133, 1994.

54. Rojas, R., *Neural networks: a systematic introduction*, Springer Science & Business Media, 2013.

55. Lichman, M., *UCI Repository of machine learning databases*, Tech. rep., University of California, Irvine, Dept. of Information and Computer Sciences, 2013, `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

56. Chang, C.-C. and C.-J. Lin, "LIBSVM: A Library for Support Vector Machines", *ACM Trans. Intell. Syst. Technol.*, Vol. 2, No. 3, pp. 27:1–27:27, May 2011.

57. Pedregosa, F., G. Varoquaux, A. Gramfort and et al., "Scikit-learn: machine learning in Python", *J. Mach. Learn. Res.*, Vol. 12, pp. 2825–2830, 2011.

58. Rosenberg, A. and J. Hirschberg, "V-measure: A conditional entropy-based external cluster evaluation measure", *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pp. 410–420, 2007.

59. Lyon, R., B. Stappers, S. Cooper, J. Brooke and J. Knowles, "Fifty years of pulsar candidate selection: from simple filters to a new principled real-time classification approach", *Monthly Notices of the Royal Astronomical Society*, Vol. 459, No. 1, pp. 1104–1123, 2016.

60. Keith, M., A. Jameson, W. Van Straten, M. Bailes, S. Johnston, M. Kramer, A. Possenti, S. Bates, N. Bhat, M. Burgay and et.al., "The high time resolution universe pulsar survey–i. system configuration and initial discoveries", *Monthly Notices of the Royal Astronomical Society*, Vol. 409, No. 2, pp. 619–627, 2010.

61. Rumelhart, D. E., G. E. Hinton and R. J. Williams, *Learning internal representations by error propagation*, Vol. 1 Label: 248, pp. 318–364, MIT Press, Cambridge,MA, 1986.

APPENDIX A: SOURCE CODE FOR THE EXPERIMENTS

A.1. ONLINE K-MEANS

```
#synthetical data generation
def synthetical_data_generator(N, k, d, std, seed):
    from sklearn.datasets.samples_generator import make_blobs
    X, y_true = make_blobs(n_samples=N, centers=k, n_features=d,
                           cluster_std=std, random_state=seed)
    return X, y_true


#initalization of the centroids
def initialize_centroids(points, k, shuffle=False):
    """returns k centroids from the initial points"""
    centroids = points.copy()
    if shuffle:
        np.random.shuffle(centroids)
    return centroids[:k]
```

Figure A.1. Data generation and initialization.

```
# to adjust obtained cluster labels to target labels
def adjust_labels(labels, labels_true, k):
    labels_adjusted = np.zeros_like(labels)
    for i in range(k):
        mask = (labels == i)
        labels_adjusted[mask] = mode(labels_true[mask])[0]
    return labels_adjusted
```

Figure A.2. Data postprocessing.

```
def kmeans(points,k,epochs):
    centroids = initialize_centroids(points, k)
    trajectories =[centroids]
    for epoch in range(epochs):
        distances = np.sqrt(((points-centroids[:,np.newaxis])**2).sum(
            axis=2))
        closest = np.argmin(distances, axis=0)
        centroids = np.array([points[closest==k].mean(axis=0)\
                              for k in range(centroids.shape[0])])
        if epoch%5 == 0:
            trajectories.append(centroids)
    return centroids,trajectories
```

Figure A.3. K-means (Batch).

```
def onlinekmeans(X,k):
    """ Online k-means using stochastic gradient descent.
    X : data matrix, C : initial centers
    eta: 1 / n_k gradually decreasing learning rate
    """
    C = initialize_centroids(X,k)
    etas = np.zeros(C.shape[0])
    for x in X:
        idx = np.argmin(((C - x)**2).sum(1))
        etas[j] += 1
        eta = 1.0 / etas[j]
        # update only the centroid with minimum distance
        C[j] = (1.0 - eta) * C[j] + eta * x
    return C
```

Figure A.4. Online k-means with stochastic gradient descent.

```
def art(X, rho, eta=0.3):
    """
    rho: vigilance parameter
    """
    C = initialize_centroids(X,1)
    for x in X:
        d = np.min(((C - x)**2).sum(1))
        if d < rho:
            j = np.argmin(((C - x)**2).sum(1))
            C[j] = (1.0 - eta) * C[j] + eta * x
        else:
            C = np.append(C, np.array([x]), axis=0)
    return C
```

Figure A.5. Adaptive resonance theory.

```
def fully_online(X, k, epsilon=1e-10):
    """
    X : data matrix
    C : initial centers
    f:  facility cost
    """

    C = initialize_centroids(X,k+1)
    n = k+1
    dists = []
    for x in C:
        d = ((C - x)**2).sum(1)
        d = [i for i in d if math.abs(i) > epsilon]
        dists.append(d)
    j = np.min(dists)/2
    f = j / k
    r=1
    q = 0
    for x in X:
        n += 1
        D2 = np.min((C - x).sum(1))**2
        p = np.min((D2/f, 1))
        random_number = np.random.random()
        if random_number < p:
            C = np.append(C, np.array([x]), axis=0)
            q += 1
        if q >= 3*k*(1+np.log(n)):
            r += 1
            q = 0
            f = 2*f

    return C
```

Figure A.6. Fully online k-means.

```
def modified_fully_online(X, k_target, epsilon=1e-10):
    """
    Modified algorithm in (Liberty et al.) for experimental design
    in which they did  pragmatic decisions about, e.g
    how to set the initial facility.
    k_target :  the number of clusters we would like
    the algorithm to output
    """

    k = (k_target - 15) / 5
    C = initialize_centroids(X,10)
    n = k+1
    dists=[]
    for x in C:
        d = ((C - x)**2).sum(1)
        d = [i for i in d if math.abs(i) > epsilon]
        dists.append(d)
    j = np.sum(sorted(dists)[:10])/2
    f = j
    r = 1
    q = 0
    for x in X:
        n += 1
        D2 = np.min((C - x).sum(1))**2
        p = np.min((D2/f, 1))
        if np.random.random() < p:
            C = np.append(C, np.array([x]), axis=0)
            q += 1
        if q >= k:
            r += 1
            q = 0
            f = 10*f

    return C
```

Figure A.7. Modified fully online k-means.

```
def estimated_cluster_num(centroids):
    k = len(centroids)
    return('Estimated number of clusters: %d' % k)

def homogenity_completeness_v_measure(labels, labels_true):
    homogeneity, completeness, v_measure \
        = metrics.homogeneity_completeness_v_measure(labels_true, labels)
    print("Homogeneity: %0.3f" % homogeneity)
    print("Completeness: %0.3f" % completeness)
    print("V-measure: %0.3f" % v_measure)


def adjusted_rand_index(labels, labels_true):
    return("Adjusted Mutual Information: %0.3f"% metrics.
        adjusted_mutual_info_score(labels_true, labels)
def adjusted_mutual_information(labels, labels_true):
            return("Adjusted Mutual Information: %0.3f"% metrics.
                adjusted_mutual_info_score(labels_true, labels))
def silhouette_coef(X, labels):
            return("Silhouette Coefficient: %0.3f"% metrics.
                silhouette_score(X, labels, metric='sqeuclidean'))
```

Figure A.8. Evaluation metrics.

A.2. MULTILAYER PERCEPTRON

```
class MultiLayerPerceptron:
    def __init__(self, network_size, model='SGD'):
        """Initialize the network

        network_size = (n_input,n_hidden1,...,n_hiddenk, n_output)
        n_input: number of neurons in input layer
        n_hiddenj: number of hidden neurons in hidden layer j
        where j = 1,2,..k
        n_output:number of output neurons
        """
        # set layer values
        self.indices = len(network_size) - 1
        self.shape = network_size
        self.inverse = inverse_exp
        self.model = model
        # to store inputs and outputs after forward propagation
        self._S = []
        self._O = []

        # Initialize weights
        layer_array = np.array([network_size[:-1], network_size[1:]]).T
        self.weights = []
        for (layerpair_1,layerpair_2) in layer_array:
            # create a random array
            temp = np.random.uniform(low=-1.0, high=1.0, size=layerpair_2
                *(layerpair_1 + 1))
            self.weights.append(temp.reshape(layerpair_2, layerpair_1 +
                1))
```

Figure A.9. Algorithms.

```
        if self.model =='RLS':
            self.P_inv = [100*np.eye(x+1) for x in network_size[:-1]] +
                100*[np.eye(network_size[-1])]
    #Forward Propagation
    def FeedForward(self, incoming):
        """Feed the network with inputs"""
        # Reset values
        self._S = []
        self._O = []

        # Feedforward
        for k in range(self.indices):
            # if we are at the input layer
            if k == 0:
                # we also add bias
                input_with_bias = np.array([np.append(i,1) for i in
                    incoming])
                S = self.weights[0].dot(input_with_bias.T)
            # else we are the hidden layer
            else:
                # we take the data from previous layer
                # hidden_input_with_bias
                b = np.ones([1, incoming.shape[0]])
                S = self.weights[k].dot(np.vstack([self._O[-1],b]))
            # layer inputs
            self._S.append(S)
            # layer outputs
            self._O.append(sigmoid(S))
        # return output from the last layer
        return self._O[-1].T
```

Figure A.9. Algorithms (cont.).

```
# Backpropagation
def RLS_BP(self,incoming,target,eta,forgetting_factor):
    """
    Backpropagate the network for one epoch
    eta:learning rate
    """
    # to store deltas
    delta = []

    i = np.random.randint(np.array(incoming).shape[0])
    x = np.array([incoming[i]])
    y = target[i]

    # FeedForward the network
    self.FeedForward(x)
    # Compute deltas
    # start from Output Layer and move backwards
    for k in range(self.indices)[::-1]:

    # if we are at Output Layer
        if k == self.indices - 1:
            e = self._O[k]-y.T
            output_delta = e*sigmoid_derivative(self._S[k])
            error = 0.5*np.sum(e**2)
            delta.append(output_delta)
        # else we are at hidden layer
        else:
            # delta_h --> following layer's delta
            delta_h = self.weights[k + 1].T.dot(delta[-1])
            f_deriv_S = sigmoid_derivative(self._S[k])

            # takes all the but last rows that correspond to biases
            hidden_delta = delta_h[:-1, :]*f_deriv_S
            delta.append(hidden_delta)
```

Figure A.9. Algorithms (cont.).

```
        # Compute weight changes
        for k in range(self.indices):
            if k == 0:
                # if we are in input layer
                # add biases also
                input_with_bias = np.array([np.append(i,1) for i in x])
                O = input_with_bias.T
            else:
                # output for previous layer
                # add biases also
                b = np.ones([1, self._O[k - 1].shape[1]])
                O = np.vstack([self._O[k - 1],b])
            # kalman gain
            divide_to = np.dot(np.dot(O.T, self.P_inv[k]), O) \
                 + forgetting_factor
            K = np.dot(self.P_inv[k], O)/divide_to
            self.P_inv[k] = np.divide((self.P_inv[k] \
                 - K * O.T * self.P_inv[k]), forgetting_factor )
            # adapt index of delta for reverse order
            k_delta = self.indices - 1 - k
            # take current deltas and multiply it with kalman gain
            Delta_w_current = delta[k_delta]

            # update the weights
            if k == self.indices - 1:
                 # output layer update
                desired_summation_output = self.inverse(y)
                self.weights[k] -= \
                    np.dot((self._S[-1]-np.array([\
                         desired_summation_output]).T),np.array(K).T)
            else:
                self.weights[k] -= eta*(Delta_w_current)*(np.array(K).T)
        # returns error
        return error
```

Figure A.9. Algorithms (cont.).

```
# Backpropagation algorithm for stochastic gradient descent
def SGD_BP(self, incoming, target, eta):
    """
    Backpropagate the network for one epoch
    eta:learning rate
    """
    # to store deltas
    delta = []

    i = np.random.randint(np.array(incoming).shape[0])
    x = np.array([incoming[i]])
    y = target[i]

    # FeedForward the network
    self.FeedForward(x)
    # Compute deltas
    # start from Output Layer and move backwards
    for k in range(self.indices)[::-1]:

        # if we are at Output Layer
        if k == self.indices - 1:
            e = self._O[k]-y.T
            output_delta = e*sigmoid_derivative(self._S[k])
            error = 0.5*np.sum(e**2)
            delta.append(output_delta)
        # else we are at hidden layer
        else:
            # delta_h --> following layer's delta
            delta_h = self.weights[k + 1].T.dot(delta[-1])
            f_deriv_S = sigmoid_derivative(self._S[k])

            # takes all the but last rows that correspond to biases
            hidden_delta = delta_h[:-1, :]*f_deriv_S
            delta.append(hidden_delta)
```

Figure A.9. Algorithms (cont.).

```
        #Compute weight changes
        for k in range(self.indices):
            '''
            * get outputs of the layers
            * multiply all the outputs from previous layer
              by all of the deltas from the current layer
            * update the weights that connect
              previous layer to the current layer
            * return error
            '''
            if k == 0:
                # if we are in input layer
                # add biases also
                input_with_bias = np.array([np.append(i,1) for i in x])
                O = input_with_bias.T
            else:
                # output for previous layer
                # add biases also
                b = np.ones([1, self._O[k - 1].shape[1]])
                O = np.vstack([self._O[k - 1],b])

         # adapt index of delta for reverse order
            k_delta = self.indices - 1 - k
            # take current deltas and multiply it
            # with previous layers' outputs
            delta_x_O = delta[k_delta][np.newaxis,:,:].transpose(2, 1, 0)
                * O[np.newaxis,:,:].transpose(2, 0 ,1)
            Delta_w_current = eta*np.sum(delta_x_O, axis = 0)
            # update the weights
            self.weights[k] -= eta*Delta_w_current

        # returns error
        return error
```

Figure A.9. Algorithms (cont.).