# Refinement of Chart Patterns in CPL

**Raluca Musăloiu-Elefteri**

ralucam@mail.com

*A report presented in partial fulfilment for the degree of Bachelor of Science*

**Supervisors:**

**Khoo Siau Cheng**
School of Computing
National University of Singapore
khoosc@comp.nus.edu.sg

**Nicolae Ţăpuş**
Dept. of Computer Science
University "Politehnica" of Bucharest
ntapus@cs.pub.ro

**2003**

Dedicated to everyone

# Acknowledgments

Special thanks go to my project supervisor, A/P Khoo Siau Cheng for his beyond limits patience and support and for very interesting discussions during our meetings. Also, many thanks for my second supervisor, A/P Chin Wei Ngan. For these two guys EVERY day REALLY has 24 hours.

Saswat Anand created the CPL Language and also took part at almost all the meetings with Khoo Siau Cheng. I want to thank him for sharing ideas with me and in general, for making the project funnier.

I also want to thank to my supervisor from Romania, prof. Nicolae Tăpuş, for his optimistic way of view and for all his efforts in encouraging students to do their best.

Entire Programming Languages and Systems Lab team was very nice and all Romanians from there deserve many thanks for all good time we spend together.

Also, I want to thank prof. Cristian Giumale who teach an absolutely wonderful Functional Programming course at Computer Science Department at UPB, Romania.

# Contents

# Summary

This project analyzes several approaches that can be used for adjusting chart patterns definitions written in CPL (*Chart Pattern Language*) and then proposes and implements two distinct methods for doing this.

# List of Tables

# List of Figures

Chapter 1

# Introduction

In Chapter 2 we will talk a little about Haskell and Chart Pattern Language and about what is already done using this language. Chapter 3 will present the concrete definition of the problem, in the sense of what this pattern refinement is supposed to mean. The next chapter (Chapter 4) will contain the theoretical part of the project, how the refinement problem can be approached and what problems occur. Chapter 5 is about implementation of algorithms, modules, scripts, file formats and so on. The next chapter, Chapter 6, will contain the results of the tests and some ways to evaluate the goodness of the new definitions and then, the conclusion and possible future work (Chapter 7). Of course the references part cannot be skipped.

# Chapter 2

# About CPL

Chart Pattern Language (CPL) was proposed by Saswat Anand in his MSc thesis [22] [23] in 2001 and it is a high-level language to facilitate pattern discovery process. The language enables financial analysts to do the following tasks:

1. Define patterns with fuzzy constraints. Through incremental addition of fuzzy constraints to a pattern, the user is able to refine patterns iteratively.

2. Reuse patterns. Complex patterns are built by composing simpler patterns and adding further constraints on them.

By embedding *Chart Pattern Language* within Haskell, the user can reap benefits from various nice features of Haskell. Specifically,

- Haskell's strong type system infers the type of pattern definition automatically. This frees programmers, who are financial analysts by profession, from the mundane task of declaring variables and specifying types – a task which they are not at all comfortable with or enjoy doing.

- Higher-order functions are used extensively throughout the system to create a natural and concise syntax for the language. CPL has been designed with the aim of making it similar to spread-sheet formula language, which all financial analysts feel familiar with.

- Searching for patterns in price history involves multiple constraints satisfaction, which has a worst case exponential running time. Lazy Evaluation plays a crucial role by automatically avoiding unnecessary computation during searching for patterns.

Chapter 3

# The problem of adjusting definitions of chart patterns

Chart patterns are generally used for detecting systematic behavior of stock performance. That is, the occurrence of a pattern bodes the rising or falling of stock price in the near future. It is known that chart pattern specifications are subjective in nature. The usual yardstick to measure the effectiveness of pattern specification lies in its ability to predict the future correctly at most of the time. We investigate how a pattern specification can be refined to boost its predictive power.

As a yardstick for measuring the effectiveness of pattern specification is mainly its prediction ability, we rely on past stock data to determine this ability. Our experiment begins with a collection of past stock data. We intend to break the data set into two groups: the controlled and uncontrolled groups.

For fairness, both the controlled and uncontrolled groups should have similar kind of stock constituents. That is, the composition of stock in one group should be similar to that in the other group.

Given a chart pattern specification $c$, and two sets of stock data, labeled as $C$ (for controlled group) and $U$ (for uncontrolled group), let $s(c,\ D)$ be the average predictability (ranged between 0 and 1) of pattern specification $c$ over a set of data $D$. The pattern-refinement problem is as follows:

Suppose $s(c,\ C)$ and $s(c,\ U)$ are $p_1$ and $p_2$ respectively, where $0 < p_i < 0.5$, for $i = 1,\ 2$. Use the data set $C$ and specification $c$ to find a refined pattern specification $c'$ such that:

1. $[\![c']\!] \subset [\![c]\!]$, where $[\![c]\!]$ represents the set of pattern instances conforming to $c$ found in a set of stock data
2. The syntactic size of the specification $c'$ is as small as possible in comparison with the original specification $c$,
3. and predicting $s(c',\ U) = p'$ where $p' = p_2 + \delta$ for $\delta$ significantly large (wanted).

It is quite natural to classify each pattern instance found in the controlled group according to their predictability. We call those instances in the controlled group that matches the prediction the *positive* instances, and we call those instances that does not match the prediction the *negative*.

For example, let's consider a famous pattern in stock market called Head-And-Shoulder (Figure 3.1). It is known [10] that after an occurrence of this pattern on the stock market the price should drop for a certain period of time. In this case we will consider that an *instance* (an occurrence of the pattern in the data set) is positive if the price really drops (as can be seen in Figure 3.2) and negative otherwise (Figure 3.3).

A clear image of the entire process is illustrated in Figure 3.4.

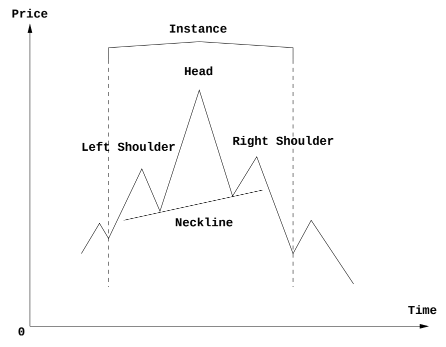

Figure 3.1: Head-And-Shoulder pattern

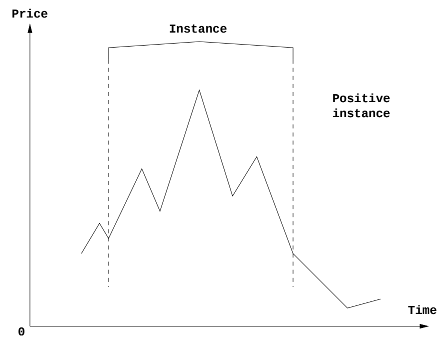

Figure 3.2: Example of a positive instance of Head-And-Shoulder pattern

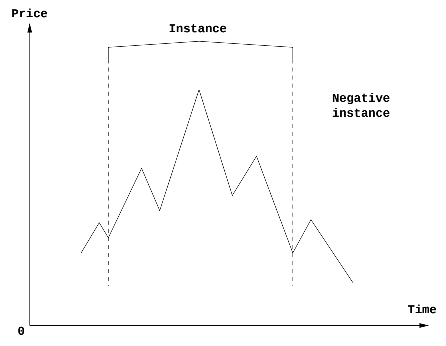

Figure 3.3: Example of a negative instance of Head-And-Shoulder pattern

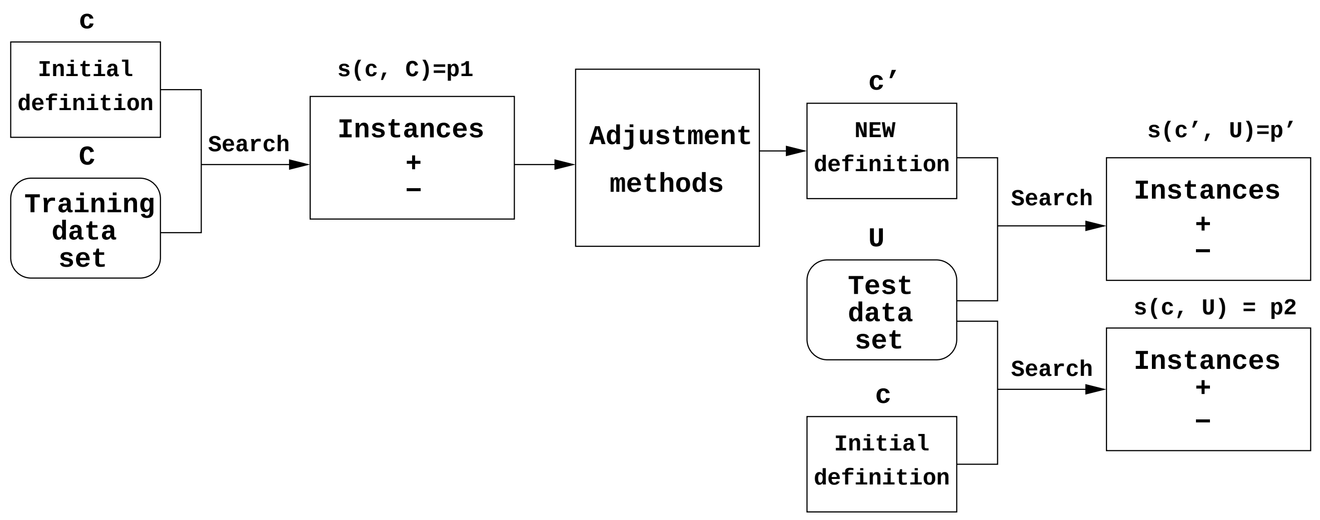

Figure 3.4: Entire process

Chapter 4

# Design

Let's see how can we automate this training process.

We said before that the definition written in CPL helps us to find a set of instances classified into two groups: positive and negative. We can try to generate a new definition by including some constraints in the original definition based on information that describes the instances and their power of prediction. So we need to find some characteristics that can best describe the positive instances and push them in some way into the current definition.

For example, if we consider an instance like the one in the Figure 4.1 some characteristics might be:

$$\frac{h_1}{h}, \quad \frac{h_2}{h}, \quad \frac{h_1}{h_2}, \quad |h_1 - h_2| \quad etc.$$

But it's very hard to define characteristics that describe *all* patterns. Every kind of pattern (i.e. *head-and-shoulder*) has its own set of characteristics that best describes it. That's why when we want to refine a pattern definition we need to know from start what characteristics we want to adjust. Beside this, we need to have a function to evaluate power of prediction of each pattern.

From now we will call the values of the characteristics as *features*.

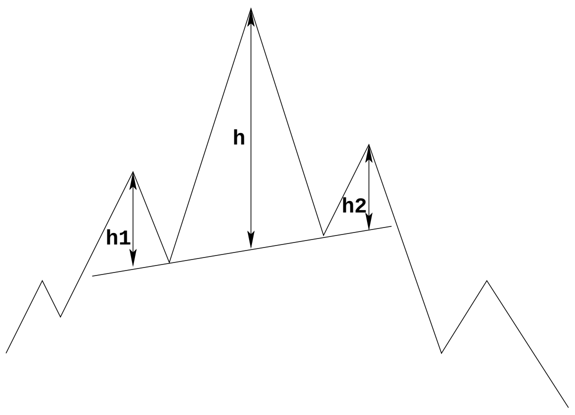

Figure 4.1: Possible characteristics of an instance

## 4.1 d-Dimensional space

An intuitive way to see an instance of a pattern is a point space. We can map the features of an instance to coordinates of a point in space. For example an instance described by 2 features can be represented by a point in plane. For 3 features we will have a point in 3D space (Figure 4.2).

This mapping of instances to points in $\Re^d$ has the advantage that it doesn't lose information, it takes in consideration all correlations between features. This representation also enables the evaluation of the difference between 2 instances by using the Euclidean distance formula between 2 points in $\Re^d$ space:

$$distance(X_i,\ X_j)\ =\ \sqrt{\sum_{k=1}^{d}(f_{ik}\ -\ f_{jk})^2}$$

where $X_i(f_{i1},\ f_{i2},\ \ldots,\ f_{i_d})$, $X_j(f_{j1},\ f_{j2},\ \ldots,\ f_{j_d})$.

Now we can see quite clearly the inputs to our automation process. We have:

- a pattern specification in CPL

- a set of data (represented by a evolution of price in time)

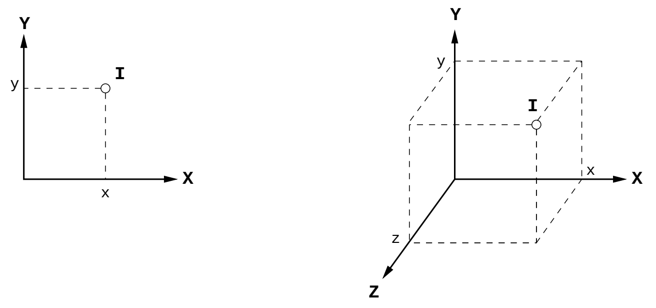

Figure 4.2: Instances in 2D and 3D space

- a function to evaluate the power of prediction of the instances found in the data set; This function does not only depend on information available in the instance, but also depend on the stock data immediately following the instance.

- a set of functions representing characteristics of the pattern; Using these functions we compute for every instance found a set of values correspond to the coordinates of a point in space.

With these inputs we can generate a new specification for the pattern, in CPL, that will detect instances with a better power of prediction. The bigger the training data set is, the better definition will be generated.

The entire problem seems now quite manageable. If we can find some methods to manipulate these data in a efficient way then the problem is solved. But still it doesn't seem clear how we can do this. In the following section we will present two totally different methods, both based on this representation of instances as points in space.

## 4.2 Voronoi Diagram

The evaluation function classifies some of the instances as *positive* (if they match the prediction) and some as *negative* (if they do not match the prediction). We want our new definition to be able to find out if an unknown instance is more likely to be a positive or a negative one. A very intuitive way is to assume that a newly discovered instance is positive if the closest instance in the training set is positive and negative if the closest one is negative.

Let's consider the simple case of $n$ points in 2D space. The number of points is assumed to be at least two and finite and all points must be distinctive (every two points differ by at least a coordinate). Given this set of points, we assign every location in the space to the nearest member of the point set. If the location happens to be equally close to two or more members of the point set, we assign the location to all those members. As a result every point in our set will have it's own cell (region) formed by all points in the space closest to if. All the points that are assigned to two or more members in the point set form the boundaries of the regions. This construction is called *planar ordinary Voronoi diagram* and the regions that constitute the Voronoi diagram are called *ordinary Voronoi polygons.*

Mathematically, the definition can be expressed like this [20, page. 67]:

Let $P = \{p_1, p_2, \ldots, p_n\}$, where $2 \leq n < \infty$ and $p_i \neq p_j$ for $i \neq j, x_i, x_j \in P$. The region given by

$$V(p_i) = \{x| \ \|x - p_i\| \leq \|x - p_j\| \ for \ j \neq i, \ j \in P\}$$

is called the *(ordinary) Voronoi polygon* associated with $p_i$ or the *Voronoi polygon of* $p_i$ and the set given by

$$\vartheta = \{V(p_1), \ldots, V(p_n)\}$$

is called *(planar ordinary) Voronoi diagram* or *Voronoi diagram of* $P$.

Another equivalent definition was given by Berg ([4, page 149]).

An example of Voronoi diagram can be seen in Figure 4.3.

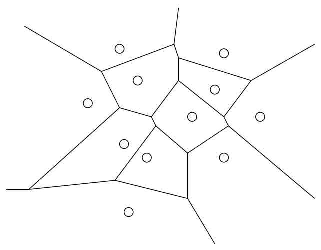

Figure 4.3: Voronoi Diagram

### 4.2.1 Voronoi Diagram in 2D space

There are some algorithms in computation geometry that are available for computing Voronoi diagram in 2-dimensional space. Here are 4 algorithms, briefly presented. More details can be found in *Computational Geometry in C* by Joseph O'Rourke [21].

#### Intersection of Halfplanes

Each Voronoi region can be constructed separately, by intersecting $n - 1$ halfplanes, using this formula:

$$V(p_i) \ = \ \bigcap_{i \neq j} H(p_i,\ p_j)$$

It is known that the intersection of $n$ halfplanes can be done in $O(n\ log\ n)$ time using a divide-and-conquer algorithm. Doing this for each point the cost of computing diagram will be $O(n^2\ log\ n)$.

### Incremental Construction

Suppose the Voronoi diagram for $k$ points is already constructed and we would like to construct the diagram after adding one more point $p$. Green and Sibson, in 1977, were the first to provide such an incremental algorithm. This is one of the cleanest algorithms for computing Voronoi diagram [21].

It spends $O(n)$ time for a point insertion, so the total complexity will be $O(n^2)$. In spite of squared complexity, this was the most popular method for constructing Voronoi diagrams.

Field gave an implementation for this in 1986. Then the incremental algorithm was revised using randomized algorithms.

### Divide and Conquer

A complex divide-and-conquer algorithm for constructing Voronoi diagram was proposed by Shamos and Hoey in 1975 with a complexity of $O(n \; log \; n)$. Although the complexity is nearly optimal, the algorithm is very difficult to implement.

Guibas and Stolfi made available an implementation of this algorithm in 1985, using some smart data structures.

### Fortune's Algorithm

In '80, most implementations for computing Voronoi diagrams used the $O(n^2)$ incremental algorithm, accepting a slower performance to avoid the complexity of the divide-and-conquer coding.

But in 1985, Fortune invented a clever plane-sweep algorithm that is as simple as the incremental algorithms, but has worst-case complexity of $O(n \; log \; n)$.

### 4.2.2 Voronoi in d-dimensional space

In our specific case (adjusting pattern definitions), every instance is described by a set of usually more than 2 of features which can be seen as coordinates of the point. The particular case of 2 dimensions is not enough, since our points are in a multi-dimensional space.

That's why we need to consider the problem of computing Voronoi diagram in higher dimensions.

A definition of Voronoi diagram can be like this:

Given a set $P$ of $n$ distinct points in $\Re^d$, Voronoi diagram is the partition of $\Re^d$ space into $n$ polyhedral regions $V(p_i),\ p_i \in P$. Similar with 2D space, each region $V(p_i)$ is defined as a set of points in $\Re^d$ which are closer to $p_i$ than to any other points in $P$:

$$V(p_i) = \{x \in \Re^d \mid distance(x,\ p_i) \leq distance(x,\ p_j)\ \forall\ p_j \in P - p_i\}$$

where *distance* is the Euclidean distance function:

$$distance(p_i,\ p_j) = \sqrt{\sum_{k=1}^{d} (f_{ik} - f_{jk})^2},$$

$$p_i(f_{i1},\ f_{i2},\ \ldots\ , f_{id})\ and\ p_j(f_{j1},\ f_{j2},\ \ldots\ , f_{jd})$$

In order to compute Voronoi diagram, the following construction is very important. For each point $p_i(f_{i1},\ f_{i2},\ \ldots\ , f_{id})$ in $P$, consider the hyperplane tangent to the paraboloid in $\Re^{d+1}$ at $p_i :\ x_{d+1} = x_1^2 + \ldots + x_d^2$. This hyperplane is represented by $h(p_i)$:

$$\sum_{j=1}^{d} f_{ij}^2 - \sum_{j=1}^{d} 2\ f_{ij}^2\ x_j + x_{d+1} = 0$$

By replacing the equality with inequality $\geq$ above for each point $p_i$, we obtain a system of $n$ inequalities. The polyhedron $T$ in $\Re^{d+1}$ of all solutions to the system of inequalities is a lifting of the Voronoi diagram to one higher dimensional space.

In other words, by projecting the polyhedron $T$ onto the original $\Re^{d+1}$ space, we obtain the Voronoi diagram in the sense that the projecting of each facet of $T$ associated with $p_i \in P$ is exactly the Voronoi cell for point $p_i$:

$$T = \{x \in \Re^{d+1} \mid \sum_{j=1}^{d} f_{ij}^2 - \sum_{j=1}^{d} 2\, f_{ij}^2\, x_j + x_{d+1} \geq 0 \quad \forall p_i(f_{i1},\, f_{i2},\, \ldots\,,\, f_{id}) \in P\}$$

or

$$T = \{x \in \Re^{d+1} \mid b - A\,x \geq 0\}$$

where $A$ is a given $n \times (d + 1)$ matrix and $b$ is a vector of size $n$.

This process can also be seen as the projection of lower part of convex hull of the points lifted in that special way in $\Re^{d+1}$ (Figure 4.4).

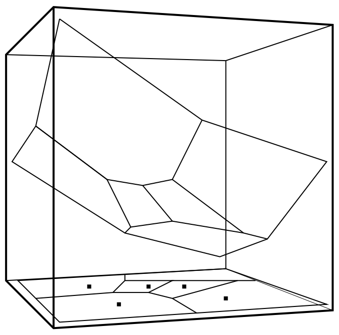

Figure 4.4: Projecting the polyhedron from 3D space onto original 2D space

The connection between the Voronoi diagrams in dimension $d$ and convex hulls in dimension $d + 1$ was first established in 1979 by Kevin Q. Brown [9] and then Edelsbrunner and Seidel [11] found that method described above.

### 4.2.3 Size of the diagram

It can be proven that, for Voronoi diagram in two dimensions, if the number of points is $n \geq 3$, there will be $n$ regions, with at most $3n - 6$ edges and at most $2n - 5$ vertices. So, considering the algorithms above, the diagram can be computed in $O(n\ log\ n)$ time using $O(n)$ storage.

For higher dimensions the situation is not as good as in 2D. Victor Klee [13] showed in 1980 that for $n$ points in $d$-dimensional space, the size of the diagram (the number of edges and vertices) is $O(n^{\lceil d/2 \rceil})$ and the complexity of the algorithm is $O(n\ log\ n + n^{\lceil d/2 \rceil})$. Another proof was given by Seidel in 1991 [25]:

$$O(n^{\lfloor (d+1)/2 \rfloor})$$

The exponential complexity of the size of this diagram make it hard to use if the number of dimensions (features) is higher than 2. In our case the exponential complexity make is unusable at all because the entire diagram should be included in the new definition of the pattern.

We can think that our problem is not exactly the classical Voronoi problem. We don't want to know for sure what is the exact region in space in which a given point is located, instead we only want to know if it belongs to a positive or a negative region. This looks like a big simplification, because we can think of combining all positive regions and eliminating the hyper-planes between two positive regions, but in these way we obtain some non-convex structures and the complexity remains exponential...

In our particular case, $n$ and $d$ don't take some extremely large values, but even considering some reasonable values, like $n = 100$ and $d = 10$, the size ($100^5$) is far from acceptable.

## 4.3 Nearest Neighbor problem

Until now we were focused on computing Voronoi diagram. If the complexity of this weren't so high we could form a new definition of the pattern by including this diagram inside it together with a low complexity searching method for locating an unknown point (instance) in this diagram. But as we saw above the size is exponential and we cannot accept it in the definition.

So we have to think in a slightly different way: instead of computing the whole Voronoi diagram we could consider this alternative problem:

> GIVEN $n$ POINTS IN $\Re^d$ SPACE AND A QUERY POINT $q$ FIND WHICH POINT IN THE SET IS CLOSEST TO $q$.

In computational geometry this problem is called *nearest neighbor problem* and it has been intensively studied in the last three decades. The goal of this problem is the same as in Voronoi case, but now, trying to avoid the computation of a huge data structure we accept the possibility of doing some more computation at query time if this could reduce the complexity of storage space.

There are several issues we need to consider when implementing an algorithm to find the nearest neighbor:

1. the *size of the data structure* $D_{NN}$, which is determined in the offline preprocessing stage
2. *time to reply* to a query, which is influenced by the design of the $D_{NN}$ as well as the algorithm used to locate nearest neighbor
3. *time to create* the data structure $D_{NN}$; this is less important because it occurs in preprocessing time

### 4.3.1 Exact problem

In high-dimensional space, the nearest-neighbor problem was first considered by Dobkin and Lipton [8]. In 1976 they were the first to provide an exponential in $d$ search algorithm ($O(2^d\ log\ n)$) using a double exponential in $d$ data structure ($O(n^{2^{d+1}})$).

Later Clarkson [6] showed in 1988 that queries could be answered in $O(d^d\ log\ n)$ time with $O(n^{\lceil d/2 \rceil (1+\delta)})$ space, for any $\delta > 0$. He reduced the storage complexity by increasing the query time cost.

Most of the subsequent approaches require a query time of at least $O(exp(d) \cdot log\ n)$: Yao and Yao [27], Matousěk [17] and Agarwal and Matousěk [1] all suffer from a query time that is exponential in $d$.

One exception is an algorithm given by Meiser [18]. He showed in 1993 that exponential factors in query time could be eliminated by giving a algorithm with $O(d^5\ log\ n)$ query time and $O(n^{d\ +\ \delta})$ space.

In any fixed dimension greater than 2, no known method achieves the simultaneous goals of linear space and logarithmic query time.

A summary of these algorithms can be seen in Table 4.1.

| Query | Storage | Paper |
|---|---|---|
| $2^d\ log\ n$ | $n^{2^{d+1}}$ | Dobkin, Lipton '76 |
| $d^d\ log\ n$ | $n^{\lceil d/2 \rceil (1+\delta)}$ | Clarkson '88 |
| | | Yao '85 |
| | | Matousěk '92 |
| | | Agarwal and Matousěk '92 |
| $d^5\ log\ n$ | $n^{d\ +\ \delta}$ | Meiser '93 |

Table 4.1: Nearest-neighbor - exact problem

### 4.3.2 Approximative methods

The apparent difficulty of obtaining algorithms that are efficient in the worst case with respect to both space and query time for dimensions higher than/then 2 suggests that the alternative approach of finding *approximate nearest neighbor* is worth considering:

> CONSIDER A SET $P$ OF POINTS IN $\Re^d$ AND A QUERY POINT $q \in \Re^d$. GIVEN $\epsilon > 0$ WE SAY THAT A POINT $p \in P$ IS $(1 + \epsilon) -$ *approximate nearest neighbor* OF $q$ IF
>
> $$distance(p,\ q) \ \leq\ (1 + \epsilon)\ distance(p*,\ q)$$
>
> WHERE $p*$ IS THE TRUE NEAREST NEIGHBOR TO $q$.

In other words, $p$ is within a relative error $\epsilon$ to the true nearest neighbor.

This problem was also extensively studied in computational geometry field (Table 4.2). Arya and Mount [3] obtained in 1993 an algorithm with query time exponential in $d$ ($O(exp(d) \cdot \frac{1}{\epsilon^d} \cdot log\ n)$) and nearly linear space: $O(n\ log\ n)$.

Clarkson [7] gave in 1994 a different algorithm which improves the dependence on $\epsilon$ in query time to $O(exp(d) \cdot \frac{1}{\epsilon^{\frac{d-1}{2}}} \cdot log\ n)$. This complexity was also obtained by Chan [5] in another algorithm in 1997.

Then in 1999, Arya, Mount, Netanyahu, Silverman and Wu [2] found an algorithm with $O(d\ n)$ complexity for storage, but the query time grows to $O(d^d \cdot \frac{1}{\epsilon^d})$.

Kleinberg [14] in 1997 showed it is possible to eliminate exponential dependencies on dimension in query time (O($d^2\ log^2\ n$)), but with $O((n\ log\ d)^{2d})$ space.

A year later, in 1998, Indyk and Montwani [12] and independently Kushilevitz [16] announced algorithms that eliminate all exponential dependencies in dimension: $O(d\ log^{\emptyset(1)}(d\,n))$ for query time and $O((d\ n)^{\emptyset(1)})$ space complexity. $\emptyset$-notation hides constant factors that depend exponentially on $\epsilon$, but not on dimension.

Unfortunately, it is quite obvious that if we consider the *Voronoi* problem, the size of the diagram is exponential (and the complexity of the search function low), and if we consider the *Nearest-Neighbor* problem (either exact or approximate),

| Query | Storage | Paper |
|---|---|---|
| $exp(d) \cdot \frac{1}{\epsilon^d} \cdot log\ n$ | $n\ log\ n$ | Arya, Mount '93 |
| $exp(d) \cdot \frac{1}{\epsilon^{\frac{d-1}{2}}} \cdot log\ n$ | $n\ log\ n$ | Clarkson '94, Chan '97 |
| $d^d \cdot \frac{1}{\epsilon^d}$ | $d\ n$ | Arya, Mount, Netanyahu, Silverman, Wu '94 |
| $d^2\ log^2\ n$ | $(n\ log\ d)^{2d}$ | Kleinberg '97 |
| $d\ log^{O(1)}(d\ n)$ | $(d\ n)^{O(1)}$ | Indyk and Montwani 1998<br>Kushilevtz 1998 |

Table 4.2: Nearest-neighbor - approximative problem

even the situation is slightly better, we still have to manage with an exponential complexity either in query time or in storage space.

Moreover, we can make some observations. For algorithms that require an exponential dependence on $d$ in query time, the bruteforce algorithm (which simply computes the distance from the query point to every point in $P$, in $O(dn)$ time), provides a faster query time even theoretically when $d \geq log\ n$ [14]. And, if the algorithm requires storage complexity to be polynomial in $n$ (for variable $d$), it appears that no algorithms are known with query time that improve bruteforce search once $d$ is comparable to $log\ n$ [14]. Arya [2] said in 1999 that:

> "... IF THE DIMENSION IS SIGNIFICANTLY LARGER THAN $log\ n$ (AS IS FOR A NUMBER OF PRACTICAL INSTANCES), THERE ARE NO APPROACHES WE KNOW THAT ARE SIGNIFICANTLY FASTER THAN BRUTE-FORCE SEARCH."

## 4.4 Bruteforce Method

Bruteforce method seems a good tradeoff between space and time complexity. Let's see what are that 3 factors we mentioned in Section 4.3:

1. The data structure $D_{NN}$ has $O(d\ n)$ complexity, that is the space required to store all points ($n$ points, $d$ dimensions); we do not manipulate a data structure to fit our needs, but simply store it as is.

2. The time required to create this data structure is $O(1)$ - points are only stored, there is no preprocessing at all.

3. Since we scan the entire space each time in order to find the nearest neighbor, query time is $O(d\ n)$; for a real-time application this complexity may not be sufficient but in our case it is quite acceptable, because in this way we can avoid an impossible-to-handle storage space.

### 4.4.1 Bruteforce Method 1

The first method is the classical bruteforce method. An unknown instance $X$ is evaluated by considering the power of prediction of the closest instance:

$$eval(X) \ = \ sgn(eval(X_k)), \quad where$$

$X_1,\ \ldots,\ X_n$ - instances
$X_i(f_{i1},\ \ldots,\ f_{id})$ - features for instance $X_i$
$distance(X,\ X_k)\ \leq\ distance(X,\ X_i)\ \forall\ i\ \neq\ k$

### 4.4.2 Bruteforce Method 2

A variation of the first method may be this: instead of taking the closest point we can evaluate an instance by considering the power of predicting all training instances. One way to do this is to ponderate the contribution of each point according to their distances to the point we want to evaluate. If a point is far away, if will not have much influence on the instance, but if it is just near to the point, then it might have many characteristics in common so the power of prediction of the point should weigh more.

For example we can choose to ponderate the values using the squared value of the distance:

$$eval(X) \ = \ sgn \sum_{i=1}^{n} \frac{eval(X_i)}{distance^2(X, \ X_i)}, \quad where$$

$X_1, \ \ldots, \ X_n$ - instances

$X_i(f_{i1}, \ \ldots, \ f_{id})$ - features for instance $X_i$

This method is more precise (at least theoretically) because it can accept some "noise" values, in the sense that even the closest neighbor is positive for example, an unknown instance can be negative if it is "surrounded" by many negative instances (Figure 4.5).

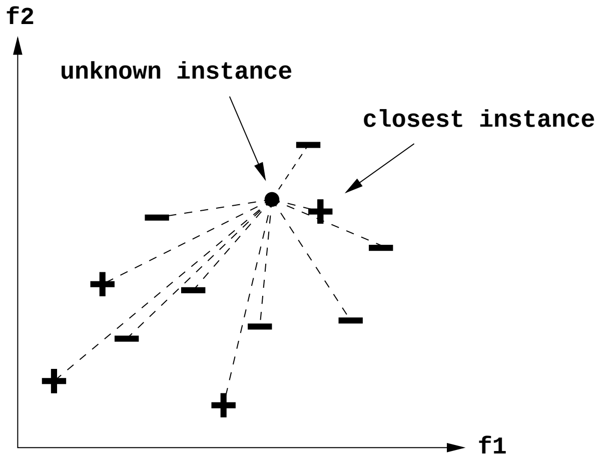

Figure 4.5: Bruteforce method 2

Also, although it seems a little more complicated than the first method, the complexity is exactly the same because in the bruteforce method, even we need only the nearest neighbor, we cannot skip the task of traversing all points for each query points.

### 4.4.3 Data normalization

As we said in Section 4.2.2 the distance between instances is the classical Euclidean distance, but computed in d-dimensional space:

$$distance(p_i,\ p_j) \ = \ \sqrt{\sum_{k=1}^{d} (f_{ik} \ - \ f_{jk})^2},$$
$$p_i(f_{i1},\ f_{i2},\ \ldots\ , f_{id})\ and\ p_j(f_{j1},\ f_{j2},\ \ldots\ , f_{jd})$$

The values of the features can be dispersed at different scales and because of this, the distance can be dominated by one dimension. For example, if one feature has a range of 0 to 1 while another has a range of 0 to 1000 then the contribution of the first feature to the distance will be swamped by that of the second feature. The contribution of one feature will depend heavily on its variability relative to other features. So it is essential to re-scale the features so that all values to be in the same range or having same standard deviation.

Suppose we want to normalize a feature $f$. Let the values for this feature, as obtained from $n$ data, be: $f_1,\ f_2, \ldots,\ f_n$.

Two of the most useful ways to normalize the date are:

1. Mean 0 and standard deviation 1:

   Compute the mean of the values:

$$\overline{f} \ = \ \frac{\sum_{i=1}^{n} f_i}{n}$$

Compute the standard deviation of the values:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (f_i - \overline{f})^2}{n - 1}}$$

Then, the normalized value corresponding to $f_i$ will be:

$$new\ f_i = \frac{f_i - \overline{f}}{\sigma}$$

2. Midrange 0 and range 2 (i.e, minimum -1 and maximum 1):

   Compute the midrange of the values:

$$midrange = \frac{max\ f_i + min\ f_i}{2}$$

   Compute the range of the values:

$$range = max\ f_i - min\ f_i$$

   The normalized value corresponding to $f_i$ will be:

$$new\ f_i = \frac{f_i - midrange}{range}$$

Various other pairs of location and scale estimators can be used besides the mean and standard deviation, or midrange and range.

We have to note here that statistics such as the mean and standard deviation are computed from the training data, not from the validation or test data. But the validation and test data *must be normalized* using the statistics computed from the training set.

Even in our case normalization is a must, generally speaking, normalization of data should be done with caution because it discards information. If that information is irrelevant then normalization can be quite helpful, like in our case. But if that information is important, then normalization can be disastrous.

### 4.4.4 Problems

In the previous section we saw the importance of scaling data. There are situations when the scaling may not be enough and the distance may still be misleading.

Until now we scale the features so that all features have the same range. But we didn't take into consideration that some features might be more important than others. For example if each instance is described by 20 attributes, but only 2 of these are relevant to determining the classification, in this case, instances that have identical values for the 2 relevant attributes may be distant from each other in the 20-dimensional space because the distance will be dominated by the large number of irrelevant attributes.

So it is essential to re-scale the data so that their variability reflects their importance (the more important features should have larger variances and/or range than less important one).

One interesting approach to overcoming this problem is to weigh each attribute when calculating the distance between 2 instances. This corresponds to stretching the axes in the Euclidean space, shortening the axes that correspond to less relevant attributes, and lengthening the axes that correspond to more relevant attributes.

This project does not implement the feature-weighing solution. Some details and references about such implementation can be found in Chapter 7.

## 4.5 Bounding Box Method

Bruteforce methods should be the most accurate methods for classifying new instances, but, as we have seen, we have to adjust the relevance of the features in a right way in order to achieve best results. In this section we explore a totally different approach, a simpler one: we try to find a box in high dimensional space which includes all positive instances and assume that all new instances inside this hyperbox are positive and all outside the box are negative. By including all positive instances it is possible to include some negative instances inside the box. We accept this and hope the number of such instances will be as small as possible.

The main drawback of this method is that it is approximative, it is not complete (as bruteforce) according to the information that can be inferred from the training data. But it has some advantages that make it worth considering.

- The new definition of the pattern will be formed by easy to check constraints (coordinates to be in some intervals).
- The algorithm is easy to compute - we don't have to deal with complex algorithms as we saw in previous sections. We will see that the complexity used here is linear in $n$ and $d$.
- The size of the refined pattern is independent on the size of the training set. This is very important because we can improve a definition many times without worrying about growth in the of the data set. Patterns generated by this method are more reusable than those obtained with methods like bruteforce.

### 4.5.1 Method

The basic idea is that we want to compute a hyperbox with *all* positive instances in it and we want this box to be *as big as possible* but to include *as few* negative instances *as possible.*

A possible method to achieve this is the one we'll describe here. It requires 2 steps:

1. Compute the minimum interval in each dimension which contains all positive instances in it. Let these intervals be:

$$\{[min_1, max_1],\ \dots\ , [min_d, max_d]\}$$

These intervals form the limits of the minimal hyperbox. Any other hyperbox which enclose all positive instances has the volume bigger than this one.

2. Expand the box (to make if as big as possible) by expanding the interval $[min_i,\ max_i]$ for each dimension $i$ in a *safe way*. Let's say we want to expand interval for dimension $D$. For adjusting the limits we search (in both directions) for the nearest (negative) instance that has *all* its coordinates except $D$ inside current intervals and its $D$-coordinate outside the current interval for $D$. We expand the limits in this way, rather than by searching for the nearest negative instance, because we don't want to limit the intervals that will provide a box smaller than the optimal one. Basically when we adjust a dimension of the current hyperbox we keep the rest of the dimensions fixed and extend limits until we rich a negative instance.

There is one caveat in this method. At the end of this method (after adjusting all dimensions) we cannot guarantee the box is maximum, because it depends on the order of adjusting. If we want the maximum box for sure we have to try all possible ways of adjusting (backtracking or branch and bound).

### 4.5.2 Example

Let's take 6 instances in 2D space (Figure 4.6):

- 1, 2, 3 - positive instances
- 4, 5, 6 - negative instances

Instance $i$ is described by the pair of features $(X_i,\ Y_i)$.

After stage 1, the minimal box with all positive instances in it (Figure 4.6) is delimited by these intervals:

$$X:\ [X_1, X_3]$$

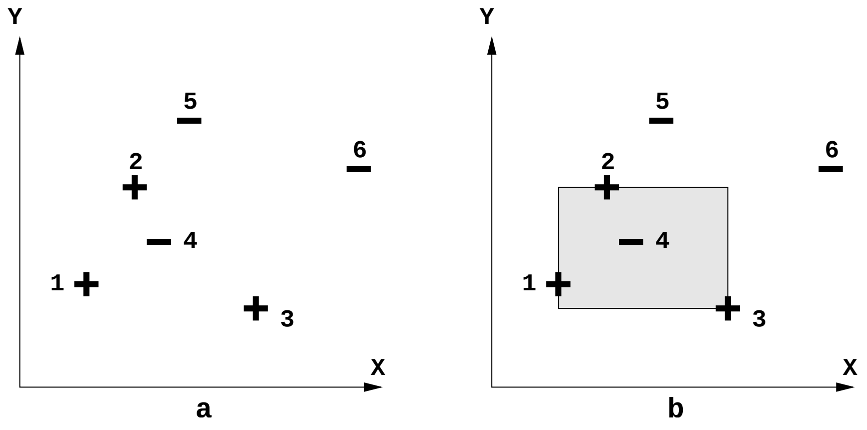

Figure 4.6: Example - Stage 1

$$Y:\ [Y_3, Y_2]$$

There are two possibilities of enlarging the box:

1. First adjust $Y$ dimension and then $X$:

   - Expand $Y$, *max* value (Figure 4.7 a): I'm looking for the nearest $Y$ value of a negative instance that have $X$ value inside current $X$ interval. We found point 5 ($X_6$ is outside interval $[X_1,\ X_3]$ that's why we didn't stop searching at point 6).

     The new interval for $Y$ dimension becomes $[Y_3,\ Y_5]$, so the new box (Figure 4.7 b) is delimited by these intervals:

     $$X:\ [X_1, X_3] \text{ - remains the same}$$
     $$Y:\ [Y_3, Y_5]$$

   - Expand $Y$, *min* value (Figure 4.8 a): there is no negative instance with $Y$ value smaller than $Y_3$ so we can extend it to $-\infty$. The new interval for $Y$ dimension becomes $(-\infty,\ Y_5]$.

     So far the new intervals are (Figure 4.8 b):

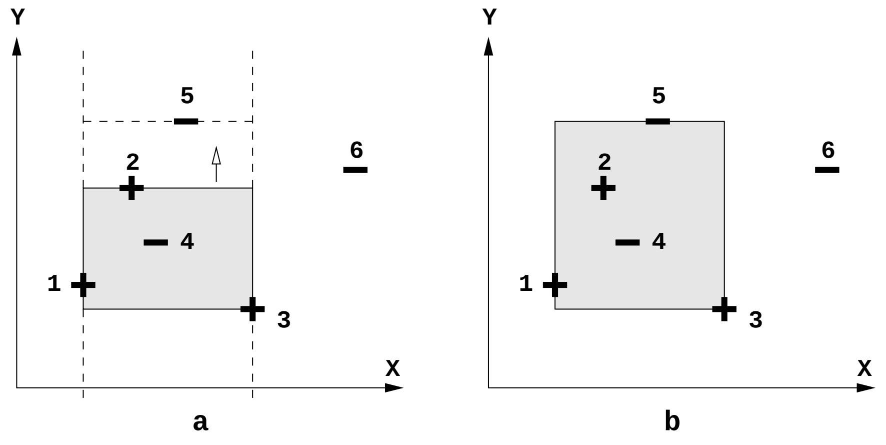

Figure 4.7: Example - expanding $Y$ interval (*max* limit), $Y$ first

$$X : \ [X_1, \ X_3] \text{ - remains the same}$$
$$Y : \ (-\infty, \ Y_5]$$

- Expand $X$, *max* value (Figure 4.9 a): point 6 is the nearest negative point with $X$ value greater then $X_3$ and $Y$ value inside $Y$ interval, which is $(-\infty, \ Y_5]$. So the new interval for $X$ is $[X_1, \ X6]$, and the box becomes (Figure 4.9 b):

$$X : \ [X_1, \ X_6]$$
$$Y : \ (-\infty, \ Y_5] \text{ - remains the same}$$

- Expand $X$, *min* value (Figure 4.10 a): find nothing so expand to $-\infty$. The new interval for $X$ is $(-\infty, \ X_6]$. The bounding box becomes (Figure 4.10 b):

$$X : \ (-\infty, \ X_6]$$
$$Y : \ (-\infty, \ Y_5] \text{ - remains the same}$$

2. First adjust $X$ dimension and then $Y$:

   - Expand $X$, *max* value (Figure 4.11 a): no negative instances are found

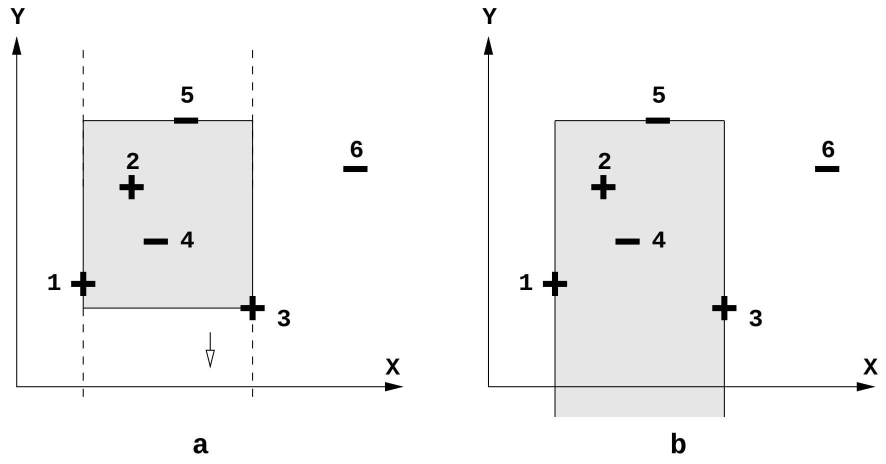

Figure 4.8: Example - expanding $Y$ interval (*min* limit), $Y$ first

so the interval for $X$ becomes $[X_1,\ +\infty)$. The bounding box is now (Figure 4.11 b):

$$X:\ [X_1,\ +\infty)$$
$$Y:\ [Y_3,\ Y_2]\text{ - remains the same}$$

- Expand $X$, *min* value (Figure 4.12 a): again, no instances found so the interval for $X$ becomes $(-\infty,\ +\infty)$ and the bounding box (Figure 4.11 b):

$$X:\ (-\infty,\ +\infty)$$
$$Y:\ [Y_3,\ Y_2]\text{ - remains the same}$$

- Expand $Y$, *max* value (Figure 4.13 a): expand to point 6, so the interval for $Y$ becomes $[Y_3,\ Y_6]$. The bounding box is now (Figure 4.13 b):

$$X:\ (-\infty,\ +\infty)\text{ - remains the same}$$
$$Y:\ [Y_3,\ Y_6]$$

- Expand $Y$, *min* value (Figure 4.14 a): $Y:\ (-\infty,\ Y_6]$. And the box becomes (Figure 4.14 b):

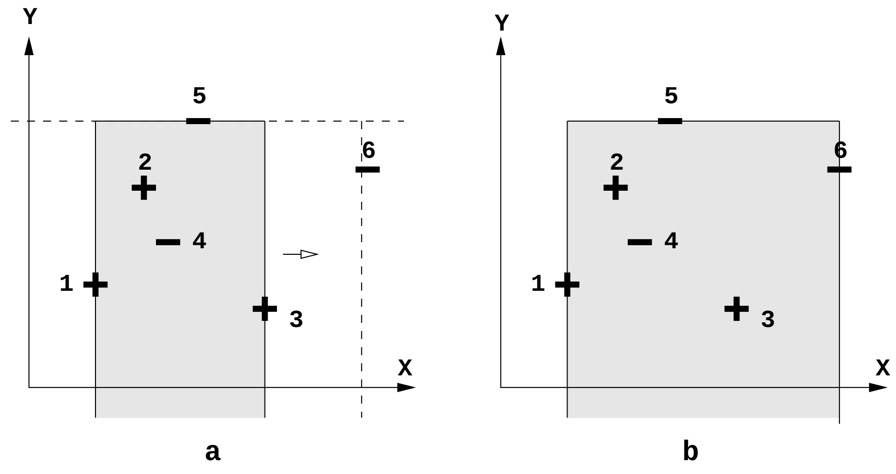

Figure 4.9: Example - expanding $X$ interval ($max$ limit), $Y$ first

$$X: \ (-\infty,\ +\infty) \text{ - remains the same}$$
$$Y: \ (-\infty,\ Y_6]$$

If we compare the bounding boxes after adjusting dimensions using order [1] and [2] (Figure 4.15) we can see that they are not the same. Moreover, it is not easy to say which box is better, because some intervals might not be (and, in our example they aren't) finite. If we want for sure the maximal box then we have to test all possibilities of ordering dimensions, using backtracking or branch-and-bound method.

In the beginning we assumed that a larger bounding box is preferred but, fortunately, we don't know *for sure* that the biggest one would give us the *best* results. So, considering that finding the biggest one is time consuming we prefer, in this implementation, to choose a random ordering of dimensions, let's say the natural one: $feature_1$, $feature_2$, ..., $feature_n$.

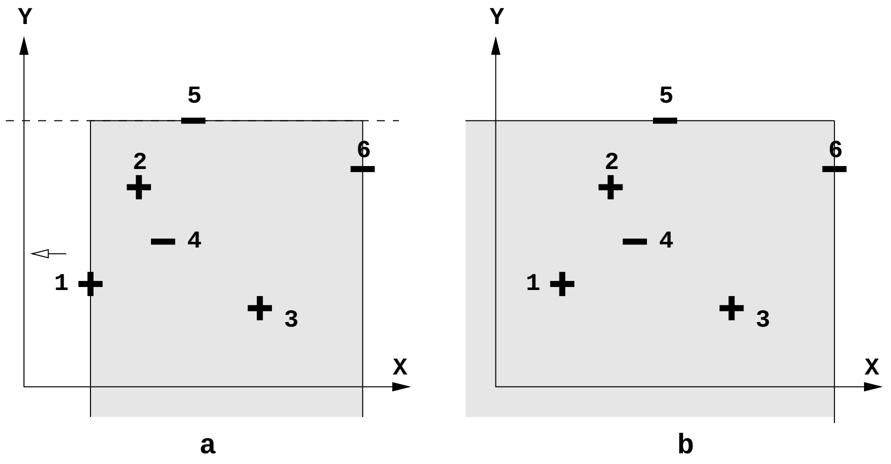

Figure 4.10: Example - expanding $X$ interval (*min* limit), $Y$ first

### 4.5.3 Threshold

While the first step of the algorithm bounds all positive instances in a box, the second stage expand this hyperbox by taking care not to include any addition negative instances in it. An obvious problem here is what happens if the bounding box include too many negative instances inside it? If the extreme values of the intervals that contain *all* instances (positive and negative) belong to positive instances then all negative instances will be inside those intervals. In this case the definition will perform poorly even when testing on the training data!

The most frequent case that results in poor performance of this method is the one where there are some "isolated" positive instances (which are positive instances surrounded by many negative instances), like the one in the Figure 4.16 a. By trying to include such an instance in a box, a lot of negative instances will also be included. A natural fix to this problem is to exclude such "isolated" instances from the training set. In this way, after expanding the box in stage 2, many of the negative instances will not be inside the box (Figure 4.16 b).

For a fine tuning we use 2 parameters here: one parameter for specifying the

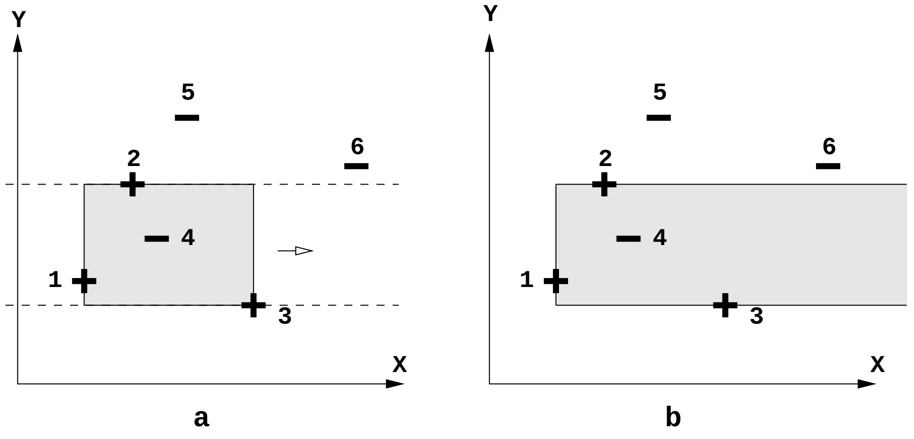

Figure 4.11: Example - expanding $X$ interval (*max* limit), $X$ first

size of the surrounding region and one for specifying the percentage of negative instances we can accept inside this region in order not to skip this positive instance. By adjusting these two parameters we can obtain better (or worse) results.

### 4.5.4 More refinement

To further reduce the number of negative instances inside the bounding box, we can attempt to exclude some negative instances from the box.

One way to do this is to partition the entire space into regular hyperboxes (Figure 4.17 a) (the number depends on the accuracy wanted) and to count the number of negative and positive instances inside each hyperbox (Figure 4.17 c). For $n$ instances in $d$-dimensional space, the space will be divided into $n^p$ $d$-dimensional boxes, where $p$ is that number in which we split one dimension). We use 2 counters for each box: one for positive instances and one for negative. If the number of negative instances in a box is much greater than the number of positive (much meaning a value given as a parameter) we conclude the entire box is “negative”, otherwise (empty box or with a great number of positive instances) is “positive”. All “negative” boxes will be skipped from the bounding box (Figure 4.17 b).

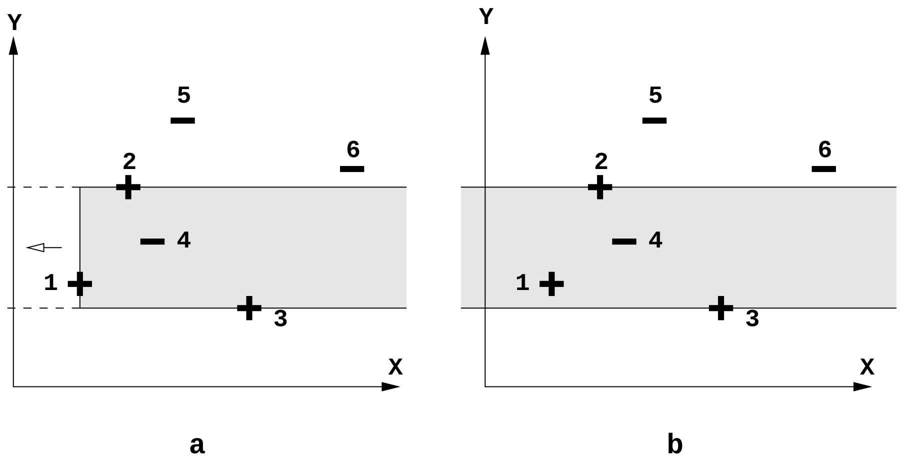

Figure 4.12: Example - expanding $X$ interval ($min$ limit), $X$ first

The total number of boxes is large but most of them are empty so we can memorize only the boxes with at least one of the two counters greater than zero. In the worst case we will have $n$ boxes (where $n$ is the total number of instances) - with each instance in a different box (in the extreme case when we fix $p$ to a high value for more accuracy). That means this improvement requires $O(d\ n)$ space complexity.

Nevertheless, the size of the new definition will grow but the accuracy will be greater.

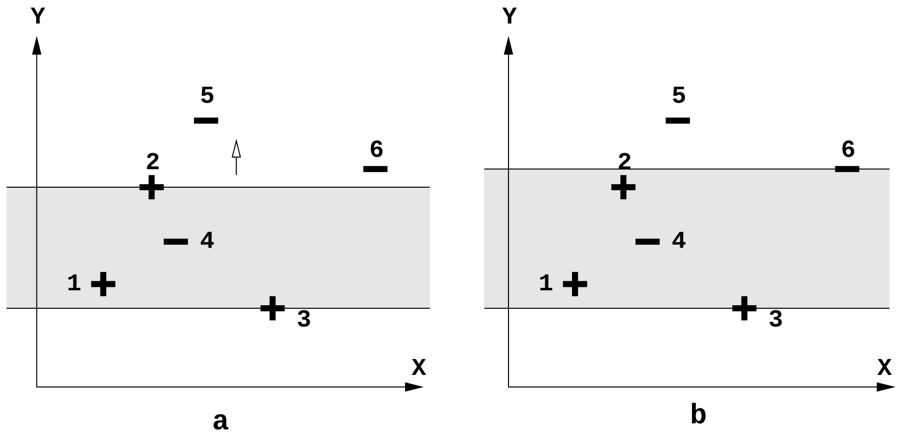

Figure 4.13: Example - expanding $Y$ interval (*max* limit), $X$ first

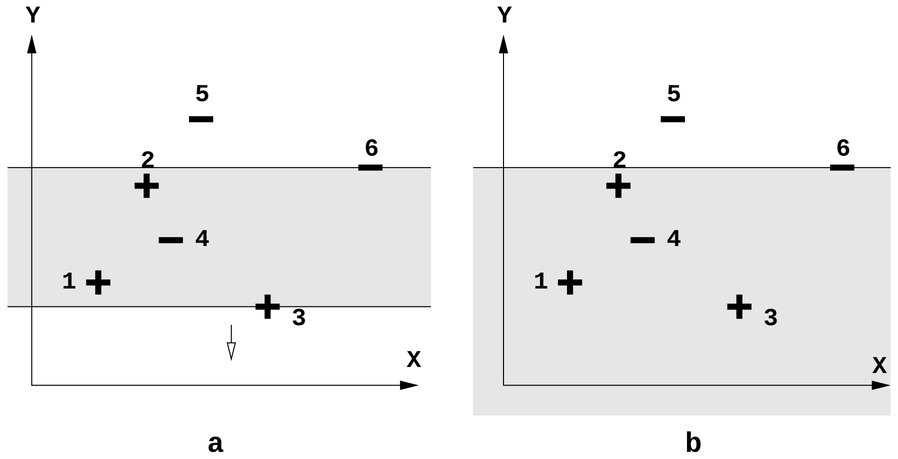

Figure 4.14: Example - expanding $Y$ interval (*min* limit), $Y$ first

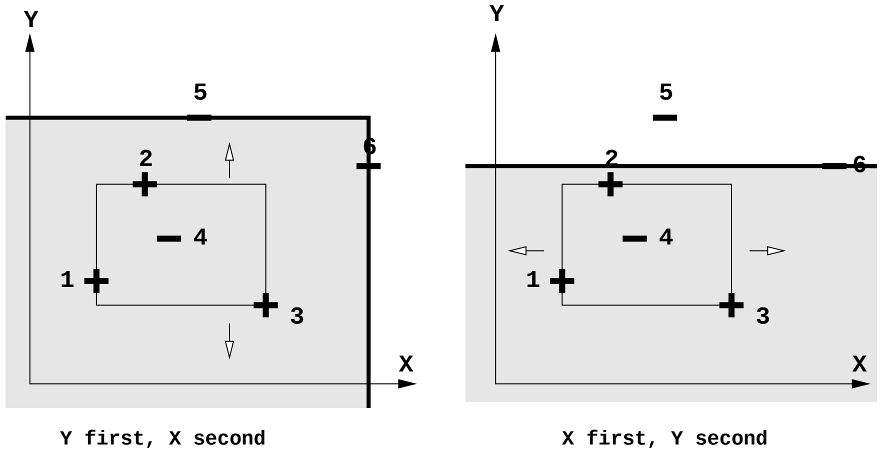

Figure 4.15: Comparison

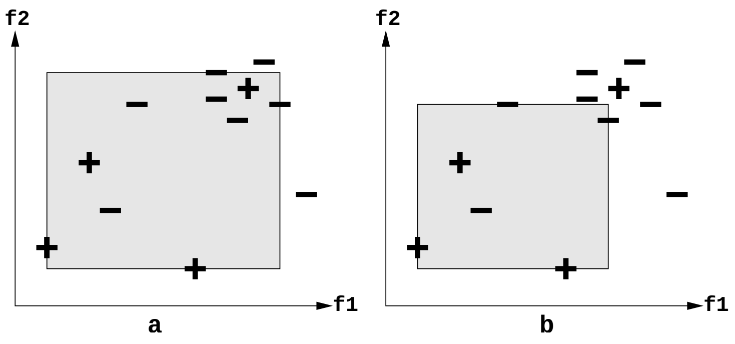

Figure 4.16: Threshold

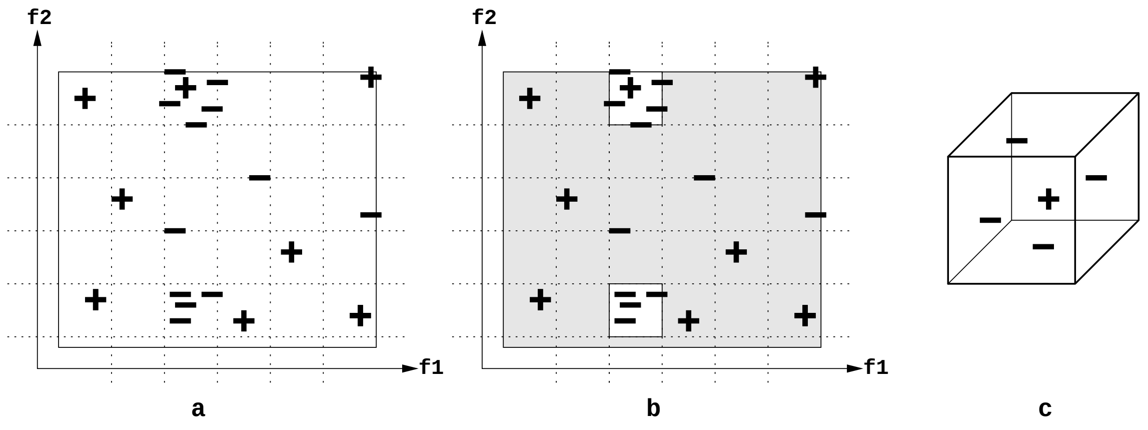

Figure 4.17: Removing negative boxes

# Chapter 5

# ... and Implementation

This chapter gives an overview of the main files of the project. Details of implementation can be found in sources.

## 5.1 Files

These are the CPL original files, each with a brief description:

- CPL.HS - contains the main function for searching a pattern a set of data; some examples of pattern's definitions are also included.
- COMPANYNAMES.HS - contains a definition of an array with 40 names of input files, one file for each company.
- MUTABLES.HS - contains:
  - definitions if indicators: *open, close, high, low, volume*
  - functions for loading data, both for ASCII and binary format
  - various declarations of instances

- OPERATORS.HS - defines various operators and their priorities; also contains some definitions of instances.
- PATTERN.HS - defines a class for *landmark*; also defines *followed-by* and *constraint-by* operations.
- PRIMITIVES.HS - contains definitions for *up* and *down* patterns
- TYPES.HS - here is definition of the *Ind* class, definition of *Patt* and *PattT* types and also some basic types (like *Slope*, *Bar*, *Price*, *Volume* etc.)

CPL also comes with a graphical interface, written in Java, which facilitate the work of discovering patterns (Browser.java, CPL.java, CommandLine.java, Ghc.java, Graph.java, MyDate.java, MyReader.java).

Files added/modified by this project:

- CPL.hs
- CPLTuningBb.hs
- CPLTuningBf.hs
- Constraints.hs
- DeepSeq.hs
- Evaluations.hs
- Features.hs
- Makefile
- Utils.hs
- run-test.hs

- run-tuning.hs
- run.hs
- some scripts

A few words about each of these files can be found in the following sections together with some details for the most important functions.

### 5.1.1 CPLTuningBf.hs

This file contains the implementation of the *Bruteforce* algorithm. For a CPL user one single function is important here:

ADJUSTALL - its goal is to search a pattern in a training set of data and generate a better CPL definition; requires the following parameters, in this order:

1. pattern - a pattern definition, like the one that exists in CPL.HS.
2. constraints - a list of additional constraints that can be checked during the searching process; it was designed as a possible backdoor for quickly implementation of some filtering functions, if they are needed, but they are not wanted to be included in the new definition; usually this list is left blank.
3. evalFunction - a function that evaluates the "goodness" of an instance; this function needs to be previously defined in the EVALUATIONS.HS file.
4. featuresFunList - a list of functions for computing characteristics of the functions that are wanted to be adjusted; all the features from the list should be previously defined in FEATURES.HS file.
5. patternLength - a maximum possible length of an instance (specified by a number of day); this parameter is needed by the search function.

6. rescaleFun - a function used for normalization of values of features; two functions (according to the two methods discussed in Section 4.4.3) are provided in Utils.hs.
7. fromFileName - name of the input file
8. fileName - name of the output file
9. constrName - a string specifying the name of the new definition
10. featuresString - a string specifying the name of the functions for computing features; we have a list of *functions* in another parameter but we also need the names of these because it should be included in the new definition.

### 5.1.2 CPLTuningBb.hs

This is the implementation of the *Bounding Box* method. Here is the main function:

adjustAllBb - its goal is to search a pattern in a training set of data and generate a better CPL definition; requires the following parameters, in this order:

1. pattern - the same as in Section 5.1.1
2. constraints - the same as in Section 5.1.1
3. evalFunction - the same as in Section 5.1.1
4. featuresFunList - the same as in Section 5.1.1
5. patternLength - the same as in Section 5.1.1
6. fromFileName - the same as in Section 5.1.1
7. fileName - the same as in Section 5.1.1

8. constrName - the same as in Section 5.1.1

9. featuresString - the same as in Section 5.1.1

10. p1 - this is the first parameter used by the threshold function; it is a percentage used for specifying the size of the region (see 4.5.3 for details); a value of *1* means the region is as large as possible (from the current point to the farthest, so $p1$ is a percentage of this maximum distance); low values (such as *0.001*) are recommended or even lower.

11. p2 - this is the second parameter used by the threshold function; it represents the ratio of the negative instances that needs to be inside the zone specified by the previous parameter in order to exclude the positive instance (see Section 4.5.3 for details); a value of *1* means no toleration, a positive instance is eliminated only in the case where all instances that surround it are negative; high values are recommended such as *0.9*.

12. p3 - the number of "chunks" in which each dimension will be split ($p$ value used in 4.5.4); recommended values: none, choose an appropriate number considering how big you want the new definition to be; should be a positive integer value.

13. p4 - it represents the ratio of the negative instances that needs to be inside the box in order to exclude a hyperbox from the bounding box (see Section 4.5.4 for more details about this); a value of *1* means that a box is eliminated only if all instances are negative; large values are recommended, such as *0.9*.

### 5.1.3 CPL.hs

In this file were added some functions that generates some statistics after a search process.

### 5.1.4 Constraints.hs

This file contains some examples of definitions generated by *bruteforce* and *bounding box* algorithms.

### 5.1.5 Evaluations.hs

This is the file in which functions for evaluating various patterns should be defined. These function will be used as parameters for *adjustAll* and *adjustAllBb* functions.

### 5.1.6 Features.hs

This is the file in which functions for computing features for various patterns should be defined. These functions will be used as parameters for *adjustAll* and *adjustAllBb* functions.

### 5.1.7 Utils.hs

Contains a set of auxiliary functions used by other modules, like some functions for working with lists, basic statistical functions and some functions for debugging.

### 5.1.8 Makefile

It is used for building binaries for searching, adjusting and testing patterns.

### 5.1.9 Other files

- run - bash script for running CPL using *ghci* interpretor
- run-test.hs - example of running test functions
- run-tuning.hs - example of running adjust functions

- run.hs - example of running search function

## 5.2 Additional scripts

- CONVERT.PL - perl script that converts a data file from Yahoo format [26] in CPL input format.
- ALL-IN-ONE - bash script that takes a file with tickers names, a start and an end date and automatically download the data for that period of time from Yahoo Finance website [26]; after downloading, the script converts the data in CPL format and runs a function for computing statistics.

# Chapter 6

# Testing Area

## 6.1 Bruteforce method

We tested the bruteforce method on a training set of data collected from *250* companies in a period of approximately *13 years* (1990-2003). A similar set of data, obtained from another *250* companies was used for testing the definitions generated by the algorithms.

It was determined that the training set contains *283* instances of the original definition, while the testing set contains *293* instances of the original definition. Also, we should know that from those *283* instances only 20 instances were found "positive" (they match the expected evolution) and from the last *293* instances *25* were positive.

We choose to perform the adjustment process on a *head-and-shoulder* pattern whose definition is provided together with CPL sources. The refinement process was based on 3 characteristics (features): $feature_1$, $feature_2$ and $feature_4$. It were tested all combinations of features.

The results obtains can be seen in Table 6.1.

The columns have the following meanings:

| | BF1.0 | BF1.1 | BF1.2 | BF2.0 | BF2.1 | BF2.2 |
|---|---|---|---|---|---|---|
| feature 1 | 32/3 | 32/3 | 32/3 | 25/2 | 25/2 | 25/2 |
| feature 2 | 22/0 | 22/0 | 22/0 | 21/0 | 21/0 | 21/0 |
| feature 4 | 21/2 | 21/2 | 21/2 | 13/2 | 13/2 | 13/2 |
| features 12 | 17/2 | 17/2 | 17/2 | 19/2 | 17/2 | 17/2 |
| features 14 | 24/4 | 22/6 | 26/6 | 13/1 | 6/3 | 8/3 |
| features 24 | 24/2 | 19/4 | 22/6 | 12/2 | 15/1 | 17/2 |
| features 124 | 32/3 | 14/5 | 14/5 | 10/1 | 4/3 | 17/2 |

Table 6.1: Bruteforce method - tests

BF1.0 = Bruteforce method 1, no rescaling

BF1.1 = Bruteforce method 1, rescaling method 1

BF1.2 = Bruteforce method 1, rescaling method 2

BF2.0 = Bruteforce method 2, no rescaling

BF2.1 = Bruteforce method 2, rescaling method 1

BF2.2 = Bruteforce method 2, rescaling method 2

A result $X/Y$ on a row *row* means that $X$ is the total number of instances found in the testing set of data using the definition generated after adjusting features mentioned in the first column on row *row*. $Y$ is the number of instances from $X$ that are "positive".

Some remarks on the results:

- The feature number 2 is quit irrelevant, when we adjust only this feature we find not even one good instance! The other two features find 2 or 3 good instances which is also a small number (from 25). The best results seems to be those obtained by $feature_1$ - $feature_4$ combination.
- The results without rescaling are worse than with rescaling (as it is expected)

and when we take into consideration only one dimension the scaling method doesn't matter (also as expected).

## 6.2 Bounding box method

For testing this method we choose two definitions of *head-and-shoulder* pattern, *hdR* and *hns*, and two distinct sets of data, *data set 1* and *data set 2*. The results can be seen in Table 6.2.

| | Before | After |
|---|---|---|
| hdR data set 1 | 163/18 | 88/14 |
| hns data set 1 | 574/119 | 429/95 |
| hdR data set 2 | 120/13 | 54/10 |
| hns data set 2 | 481/99 | 385/88 |

Table 6.2: Bounding box method - tests

We can see the improvement in the total number of instances found by the new definitions: it was reduced to almost half while the number of positive instances is close to the original number. Some acceptable values for tuning bounding-method were chosen so these might not be the best results that can be obtained on these sets of data.

This test was performed with $feature_1$, $feature_2$ and $feature_4$, altogether.

## 6.3 Comparison

Here we perform some tests in which we adjust the same definitions on the same sets of data. The results are in Table 6.3.

For evaluation of the results we used three ratios:

| Definition and method | Before | After | $ratio_1$ | $ratio_2$ | $ratio_3$ |
|---|---|---|---|---|---|
| hdR, data set 2, BF1, rescale1 | 120/13 | 7/3 | 23% | 42% | 11% |
| hdR, data set 2, BF1, rescale2 | 120/13 | 7/3 | 23% | 42% | 11% |
| hdR, data set 2, BF2, rescale1 | 120/13 | 6/2 | 14% | 33% | 11% |
| hdR, data set 2, BF2, rescale2 | 120/13 | 6/2 | 14% | 33% | 11% |
| hdR, data set 2, bounding box | 120/13 | 54/10 | 77% | 19% | 11% |
| hns, data set 2, BF1, rescale1 | 481/99 | 119/31 | 31% | 26% | 20% |
| hns, data set 2, BF1, rescale2 | 481/99 | 123/31 | 31% | 25% | 20% |
| hns, data set 2, BF2, rescale1 | 481/99 | 66/19 | 19% | 28% | 20% |
| hns, data set 2, BF2, rescale2 | 481/99 | 64/18 | 18% | 28% | 20% |
| hns, data set 2, bounding box | 481/99 | 385/88 | 89% | 22% | 20% |

Table 6.3: Tests - comparison

- $ratio_1$ - the percentage of *positive* instances found by the *new* definition relative to the number of *positive* instances found by the *original* definition
- $ratio_2$ - the percentage of *positive* instance found by the *new* definition relative to the number of *total* instances found by the *new* definition
- $ratio_3$ - the percentage of *positive* instance found by the *original* definition relative to the number of *total* instances found by the *original* definition

$ratio_2$ and $ratio_3$ are directly comparable: in the case of *hdR* adjustment the percentage rose from 11% to $19 - 42\%$ while for *hns* definition it rose from 20% to $22 - 28\%$.

Chapter 7

# Conclusion

In conclusion, both methods we have implemented in this project improve the original definitions, but, *bounding method* have a small advantage in front of *bruteforce* because of the simplicity of the algorithm and the size of the definitions that are generated. It is hard to compare the results of these two methods, because, as we saw in Section 6.3, the *bounding box* method provides higher values for $ratio_1$ (the percentage of positive instances found by the new definition relative to the number of positive instances found by the original definition) while the *bruteforce* method improve $ratio_2$ to higher values then *bounding box* method.

One important issue for bruteforce method, as we discussed in Section 4.4.4 is how we handle with irrelevant features. The standard distance formula weighs each feature equally, and this can cause problems if only a few features are relevant in the classification task, so the method could be misled by similarities in many irrelevant dimensions.

Some methods for adjusting feature relevance that can be explored in the future are:

- methods for selection a set of features that are relevant (helpful only if the number of features that are wanted to be adjusted together are really high -

i.e. $> 20$):

  - Wrapper methods - generate a set of candidate features, run the algorithm with these features and then add or remove attributes in/from this set (John et al, 1994).
  - Methods based on decision trees - in addition to storing training cases use them to induce a decision tree. Features that do not appear in the decision tree are considered irrelevant for the learning task and can be discarded (Cardie, 1993).

- methods for computing weights for the features. These methods can be split in two categories:

  - *Global* methods which compute a single weight vector for the classification task. Daelemans et al. in 1999 and Cardie and Howe in 1997 proposed some algorithms.
  - *Local* methods which allow feature weights to vary for each training instance, for each test instance, or both (Wettschereck et al, 1997); Another method was proposed by Stanfill and Waltz in 1986.

# Bibliography

[1] PANKAJ K. AGARWAL AND JIRI MATOUŠEK: *Ray shooting and parametric search*, Proceedings of the 24th Annual ACM Symposium on Theory of Computation, 1992, pages 517-526

[2] SUNIL ARYA, DAVID M. MOUNT, NATHAN S. NETANYAHU, RUTH SILVERMAN, ANGELA Y. WU: *An Optimal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions*, Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms, 1994, pages 573-582

[3] SUNIL ARYA AND DAVID M. MOUNT: *Approximate Nearest Neighbor Queries in Fixed Dimensions* Proceedings of 4th Annual ACM-SIAM Symposium on Discrete Algorithms, 1993, pages 271:280

[4] MARK DE BERG, MARC VAN KREVELD, MARK OVERMARS, ORFRIED SCHWARZKOPF: *Computational Geometry. Algorithms and Applications*, Second Edition

[5] TIMOTHY M. CHAN: *Approximate Nearest Neighbor Queries Revisited* Proceedings of the 13th Annual ACM Symposium on Computational Geometry, 1997, pages 352-358

[6] KENNETH L. CLARKSON: *A randomized algorithm for closest-point queries*, SIAM Journal on Computing, 17(1988): pages 830-847

[7] KENNETH L. CLARKSON: *An Algorithm for Approximate Closest-Point Queries*, Proceedings of the 10th Annual Symposium on Computational Geometry, 1994, pages 160-164

[8] DAVID P. DOBKIN, RICHARD J. LIPTON: *Multidimensional search problems*, SIAM Journal on Computing, 5(1976): pages 181-186

[9] KEVIN Q. BROWN: *Voronoi Diagrams from Convex Hulls*, Information Processing Letters 9(5): pages 223-228 (1979)

[10] THOMAS N. BULKOWSKI: *Encyclopedia of Chart Patterns*, John Wiley & Sons, January 2000

[11] HERBERT EDELSBRUNNER, RAIMUND SEIDEL: *Voronoi Diagrams and Arrangements*, Discrete & Computational Geometry 1: pages 25-44 (1986)

[12] PIOTR INDYK, RAJEEV MOTWANI: *Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality* Proceedings of the 30th Annual ACM Symposium on Theory of Computing 1998, pages 604-613

[13] VICTOR KLEE: *On the complexity of d-dimensional Voronoi diagrams*, Archiv der Mathematik, 34: pages 75-80, 1980

[14] Jon M. Kleinberg: *Two Algorithms for Nearest-Neighbor Search in High Dimensions*, Proceedings of the 29th Annual ACM Symposium on Theory of Computing, 1997, pages 599-608

[15] Komei Fukuda: *Frequently Asked Questions in Polyhedral Computation*, http://www.cs.mcgill.ca/ fukuda/soft/polyfaq/polyfaq.html

[16] Eyal Kushilevitz, Rafail Ostrovsky, Yuval Rabani: *Efficient Search for Approximate Nearest Neighbor in High Dimensional Spaces* Proceedings of the 30th Annual ACM Symposium on Theory of Computing, 1998, pages 614-623

[17] Jiri Matousěk: *Reporting points in halfspaces*, Computational Geometry: Theory and Applications, 2 (1992), pages 169-186

[18] Stefan Meiser: *Point location in arrangements of hyperplanes*, Information and Computation, 106(1993): pages 286-303

[19] Tom M. Mitchell: *Machine Learning*, McGraw-Hill Science/Engineering/Math, March 1997

[20] Atsuyuki Okabe, Barry Boots, Kokichi Sugihara, Sung Nok Chiu: *Spatial Tessellations. Concepts and Applications of Voronoi Diagrams*

[21] Joseph O'Rourke: *Computational Geometry in C*, 1994 or 1998 edition

[22] Anand Saswat: *CPL: A Language for Programming Chart Patterns*, Masters Thesis, National University of Singapore, 2002

[23] Anand Saswat, Wei Ngan Chin and Siau Cheng Khoo: *Charting Patterns on Price History*, ACM SIGPLAN International

Conference on Functional Programming (ICFP'01), September 2001: pages 134-145

[24] Franco P. Preparata, Michael Ian Shamos: *Computational Geometry: An introduction*, Springer-Verlag, 1985

[25] Raimund Seidel: *On the Number of Faces in Higher-Dimensional Voronoi*, 1987

[26] Yahoo Finance: *Historical Prices*, http://chart.yahoo.com/d

[27] A.C. Yao and F.F. Yao: *A general approach to d-dimensional geometric queries*, Proceedings of the 17th Annual ACM Symposium on Theory of Computing, 1985, pages 163:168