NTNU – Trondheim
Norwegian University of
Science and Technology

Learning Distance Functions in *k*-Nearest Neighbors

Sigurd Fosseng

Master of Science in Informatics
Submission date: Januar 2013
Supervisor: Agnar Aamodt, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Normally the distance function used in classification in the k-Nearest Neighbors algorithm is the euclidean distance. This distance function is simple and has been shown to work on many different datasets. We propose a approach where we use multiple distance functions, one for each class, to classify the input data. To learn multiple distance functions we propose a new distance function with two learning algorithms. We show by experiments that the distance functions that we learn yields better classification accuracy than the euclidean distance, and that multiple distance functions can classify better than one.

Sammendrag

Normalt brukes Euclids distanse som distansefunksjonen under klassifisering med k-Nearest eighbours algoritmen. Denne distansefunksjonen er enkel og har blit vist at fungerer på mange forskjellige dataset. Vi foreslår en fremgangsmåte der man brker flere distansefunksjoner, med en funksjon per klasse, for å klassifisere inndataen. For å lære flere forskjellige distansefunksjoner legger vi frem en ny distansefunksjon med tilhørende to læringsfunksjoner. Vi viser med eksperimenter at distansefunksjonene vi lærer gir bedre klassifiserings nøyaktighet enn ved bruk av euclids distanse, og at det å bruke flere distansefunksjoner samtidig kan klassifisere bedre enn ved bruk av bare en.

Acknowledgements

This thesis is submitted to the Norwegian University of Science and Tehcnology (NTNU) for fulfilment of a Masters Degree.

When I started on this thesis, Sigve Hovda at Verdande Technology had already started implementing a system that was able to do classification that in early testing seemed to classify better than k-Nearest Neighbour algorithm. This system was used as a base for this thesis. A modified re-implementation of this system is presented in Section 3.3.1

The first semester of doing work on this thesis I worked with Andreas Ståleson Landstad, who delivered his thesis in June 2012. Some of the work presented in this thesis is a result of cooperation with him.

The main supervisor was Agnar Aamodt until June 2012, and due to him taking sabbatical leave from August 2012, the main supervisor became Helge Langseth for the remainder of the thesis.

Contents

List of Tables

List of Figures

Glossary

***k*NN** *k*-Nearest Neighbours. v, vii, 4–6, 18, 19, 23, 24, 30, 43–47

GA Genetic Algorithm. v, vii, 6–9, 11, 12, 14, 15, 30, 33–35, 37, 41, 44–46

GP Genetic Programming. vii, 11, 12, 14–16, 33, 35, 44–46

MD Generalized Mean over differences. 24–26

WKNN Weighted *k*-Nearest Neighbours. 45

Chapter 1

Introduction

Classification is the task of giving correct labels to input data, and a classifier can be described as a computer system that is able to correctly label the input data. For example a classifier can be given pictures of cars and trucks, and that classifier should return the correct label for the input picture.

Supervised learning for classification, gives the classifier a set of examples to induce rules to use during the classification. Many approaches for classification use distance functions during the classification task, and often the distance function used is the euclidean distance. The advantage of using this distance function is because it is simple and can work on many different classification problems.

However the euclidean distance can be overly simplistic, because it assumes that the same distance function works equally well when the input to the classification is pictures of cars and handwritten letters. Several approaches in the literature has been focused on adopting the distance function to get better classification results. The goal of distance function learning is to induce a better distance function from examples.

In this thesis we introduce two approaches for learning distance functions for classification. By structuring the distance function as a tree we are able to induce logical structures in the problem to get better classification results.

In literature approaches that use distance functions use the same distance function for every class during classification. We believe that we can increase the performance of the classification by learning separate distance functions for each class. The distance function learners we introduce can also learn one distance function for each class.

1.1 Problem Definition

The aim of this thesis is to create distance function learners that is able to create distance functions for usage in the k-Nearest Neighbours algorithm that increases the classification accuracy.

We also aim to explore if the usage of multiple distance functions where each distance function is used for calculating distances to one class.

1.1.1 Research Questions

- Is class specific distance metrics viable?
- How can distance functions be learned?

1.2 Overview of report

Chapter 1 introduces the problem and research questions.

Chapter 2 gives an introduction to the K-Nearest Neighbour algorithm, with a focus on previous research on the distance measure used in the algorithm. It also includes descriptions of the evolution based algorithms used in this thesis.

Chapter 3 introduces the concept and implementation of the solutions presented in this thesis.

Chapter 4 shows the results of running the solutions presented and discusses the results.

Chapter 5 contains the conclusions that we can draw from the discussion and presents possible future improvements to the solutions presented.

Chapter 2

Background and Motivation

This chapter gives a theoretic basis needed to understand this thesis. The first sections presents classification and *k*-Nearest Neighbours algorithm for classification. Then a explanation of Genetic Algorithms and Genetic Programming.

2.1 Supervised Learning for Classification

Classification is the problem of giving the correct class label to the input data. For example, a system is given the task of classifying pictures of *cars* and *trucks*. The pictures are the input data, when given a picture the classification system should return the correct class label of either *car* or *truck*.

In supervised learning, the system is first given a training set with input data and class labels to learn from, for example a set of pictures with their corresponding class labels (e.g. *car* or *truck*). Then after the system has learned by example what a *car* and *truck* looks like, give the system new pictures and have the system classify it with the correct labels.

More formally supervised learning is defined as follows:

Let $\{(\vec{x}_i, c_i)\}_{i=1}^{h}$ denote a training set of h examples, usually with inputs as n-dimensional vectors $\vec{x} \in \mathbb{R}^n$ with discrete class labels c_i.

Supervised learning for classification is the problem of inferring a function $c = f(\vec{x})$ based on the training set. The obtained function is evaluated on how well it generalizes. Evaluation of generalization can be done by measuring the accuracy of classification of new data assumed to follow the same distribution as the training data.

2.1.1 *k*-Nearest Neighbours classification

The *k*NN classifier [18, 6], is one of the oldest and simplest methods used for classification, and is a supervised learner. Even though it is simple it often yields competitive results[5]. The *k*NN classifier classifies unlabeled pattern by the majority label among its *k*-nearest neighbours.

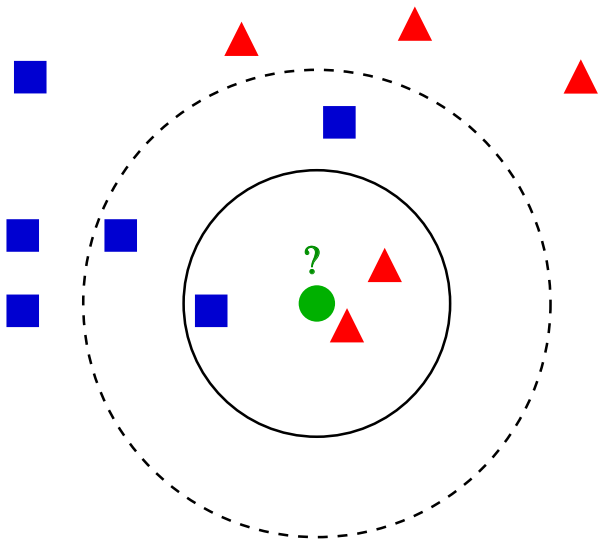


Figure 2.1: An example [1] of *k*NN with k neighbours with $k = 3$ (solid line circle) and $k = 5$ (dashed line circle)

In Figure 2.1 we show a *k*NN example with red and blue known samples, and a green unknown sample. The samples are placed in a two dimensional feature space, where each feature is one dimension. To classify the unknown sample as either red or blue, the algorithm uses a distance function (see Section 2.1.1.1) to find the k nearest neighbours of the unknown sample. Then by finding the majority (see Section 2.1.1.3) of red or blue labels among the k nearest neighbours, it predicts the label of the green sample. In this case, when $k = 3$ the unknown sample is predicted to be red, and when $k = 5$ it is predicted to be blue.

2.1.1.1 Distance Function

The *k*NN algorithm finds the k nearest neighbours by measuring the distance between the unknown sample and the training data samples.

In the literature there has been used several difference distance measurements for classification, but the most commonly used distance function is the Euclidean

distance. Given two samples $\vec{x}$ and $\vec{y}$ where $\vec{x} \in \mathbb{R}^n$ and $\vec{y} \in \mathbb{R}^n$ where n is the dimensionality, the Euclidean distance between the two samples are defined as:

$$d(\vec{x}, \vec{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{2.1}$$

Another commonly used distance function in *k*NN is the Manhattan distance:

$$d(\vec{x}, \vec{y}) = \sum_{i=1}^{n} |x_i - y_i| \tag{2.2}$$

Given $p \in \mathbb{R}$ these distances can be generalized by the Minkowski distance [24]:

$$d(\vec{x}, \vec{y}, p) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}}, p \geq 1 \tag{2.3}$$

When $p = 2$ the Minkowski distance is equal to the Euclidean distance and when $p = 1$ the Minkowski distance is equal to the Manhattan distance.

Doherty et al. [7] explored the properties of the Minkowski distance with different powers in *k*NN and did show that by using different power p one can increase the accuracy of the classification. However, the power is likely to be dependent on the data it tries to classify. We explore powers further in Section 3.2.3.

The distance function in *k*NN is defined to be a metric, Equation 2.1, 2.2 and 2.3 are all examples of a metric. Metrics have the following properties:

1. $d(\vec{x}, \vec{y}) \geq 0$; *non-negativity*
2. $d(\vec{x}, \vec{y}) = 0$ if and only if $x = y$; *identity*
3. $d(\vec{x}, \vec{y}) = d(\vec{y}, \vec{x})$; *symmetry*
4. $d(\vec{x}, \vec{y}) \geq d(\vec{x}, \vec{y}) + d(\vec{y}, \vec{z})$; *triangle inequality*

2.1.1.2 Feature Scaled Distance Functions

Kelly and Davis [12] showed that by replacing the Euclidean distance (Equation 2.1) with the weighted Euclidean distance in Equation 2.4 they could increase

the accuracy of *k*NN:

$$d(\vec{x}, \vec{y}) = \sqrt{\sum_{i=1}^{n} w_i(x_i - y_i)^2} \tag{2.4}$$

where w_i is the weight for the i-th feature. Intuitively this can be understood as defining that one feature is more important than others when comparing two samples. The weights vector $\{w_1 \dots w_k\}$ needs to be learned before classifying unknown samples. In their approach [12] the weights are learned by using a genetic algorithm. Genetic algorithms are explained further in Section 2.2

2.1.1.3 Majority Vote Classification Rule

One computes the *k*-nearest neighbours of the unknown sample, and does a vote among the neighbours where the neighbours votes on their own class label. The class label with the most votes wins, thus the unknown sample estimated to be of the class label with the most votes.

It can be shown [14] that the probability that a sample $\vec{x}$ has the label c_i can be defined as follows. Given k as the number of neighbours and k_i as the number of neighbours with the class label c_i.

$$P(c_i|\vec{x}) = \frac{k_i}{k} \tag{2.5}$$

Then the estimated label $\hat{c}$ for the sample $\vec{x}$ is the label with the highest probability.

$$\hat{c} = \arg\max_{c_i} P(c_i|\vec{x}) \tag{2.6}$$

2.2 Genetic Algorithms

GAs is heavily inspired by evolution by natural selection. In nature we observe that evolution is successful at adaption of biological systems. A good example of the adaptability of natural evolution is the rapid evolution of an Italian wall lizard species on an island in the Mediterranean.[11] A *population* of 10 lizards were left on the island and after approximately 36 years the lizards had reproduced over approximately 30 *generations* and as a result had adopted to a life on the

island. Initially the lizards were eating insects, but over the years adapted to a diet consisting largely of plants with changes in their bite strength, dietary system and head shape.

Natural evolution of a population is guided by natural selection. Natural selection is the process where individuals that are not well enough adopted (less fit) to the environment dies, while those more adopted (more fit) to the environment live longer and are able to create offspring. For instance in the case of the lizards, those not able to find edible food would not live long enough to reproduce, while some part of the population is able to find food and grow old enough to reproduce.

For natural selection to guide adaption to the environment, there must be variation in the fitness of individuals in the population. Variation in nature is generated by mutations and sexual reproduction. Mutation is a random change in one individual that might change the fitness of the individual. Sexual reproduction is the process of combining features of two parents into new offspring.

GAs [8][20] tries to reproduce the success of evolution by natural selection in a computer. By defining a problem they want to solve, they can create a population of candidate solutions to the problem, often called hypothesises. Then by mutating and reproduction of candidate solutions they can create variation in the population that they perform a selection mechanism on. Selection of individualsk is based on a fitness function that determines the fitness of an individual.

The set of all possible solution to the problem is often called the search space. And GAs tries to find or optimize the best solution in the search space.

2.2.1 Genotype

In nature the physical properties of individual creatures are determined by its genetic material. In GA the individual candidate solutions genetic material is some computer readable structure, often a binary string of ones and zeroes.

Candidate solutions should be encoded such that [8]:

- Recombination and mutation should have high likelihood of generating increasingly better candidate solutions
- The set of all possible candidate solutions should have high probability of covering the optimal solution.

2.2.2 Phenotype

In nature the phenotype of a creature is the physical properties of the creature that is encoded in the genotype. Similarly, in GA the phenotype is the candidate solution encoded in the genotype. A candidate solution can be anything that can be encoded by the genotype.

A simple example of a genotype is shown in Table 2.1. If we define the phenotype as the sum of ones in the genotype, the phenotype in the example is 2.

0	0	1	0	1

Table 2.1: An example genotype.

2.2.3 Initialization

In nature we do not often observe initialization, but in the case of the lizard on the island, the initial population was a collection of 10 different individuals. In GA this is equivalent to the initialization of the population, where one define the starting point for the evolution.

The initialization of the population in GA should be sufficiently large and diverse so that the candidate solutions have different fitness values. The size of the population in the literature is often ranging from a hundred to a few thousand.

The size of the population is dependant on the following [8]:

- The properties of the search space
- The cost of evaluating a candidate solution

In the case of genotypes represented as binary strings, initialization of the genotypes in the population is done by random generating the strings.

If one has knowledge about what a solution to the problem might look like, one can place these solutions into the initial population to speed up the evolutionary process.

2.2.4 Mutation

Mutations operate on single genotypes in the population, and is done by randomly changing a small part of the genotype. This operator should be designed in such

a way that every point in the genotype could be reached by the mutation operator. The change that a mutation creates should be so small that the discoveries in the previous solution is not lost.

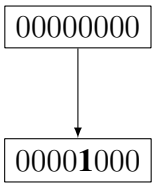


Figure 2.2: Point mutation

In Figure 2.2 we show an example of a simple genetic mutation operation. Every binary position is mutated with a probability of p_m, and in this example only one of the binary positions got mutated.

In literature p_m is typically around 0.01 at each position, which is much larger than we see in nature. The actual value should be chosen by inspecting the change the mutation takes and how it affects the final fitness value of the individuals mutated.

2.2.5 Crossover

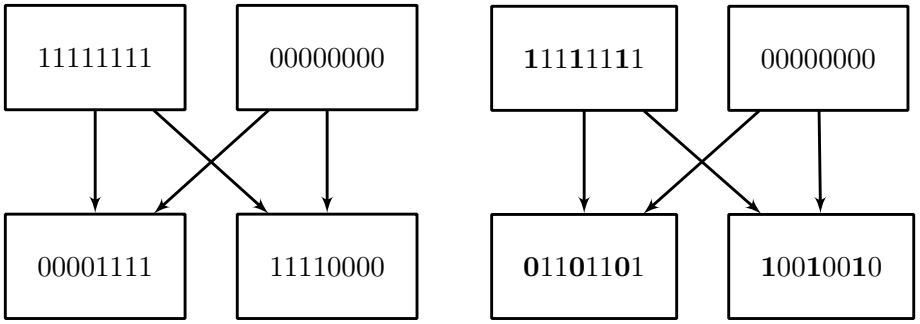
In nature Crossover is the sexual reproduction of two individuals in the population to create offspring. In GA this is done by selecting two parents and recombining them into one or two offspring genotypes.

The genetic crossover operator is also known as recombination [8]. The idea behind this is that combining subsolutions from the parents may yield better fitness values.

To validate that the crossover operator is beneficial, one can check whether the crossover operator consistently yields lover fitness than the parents. If this is the case, the assumption that combining subsolutions from parents can yield better fitness values is invalid, but that it is instead a large random mutation. When this is the case, one should skip the crossover step in the genetic algorithm.

In Figure 2.3a we show an example of a single point crossover. This is done by selecting a point of crossover, and taking the partitions around that point from different parents. This crossover method is commonly used on discrete and real valued genotypes.

In Figure 2.3b we show a example of a uniform crossover. This is done by randomly selecting n positions and swapping the genetic information from the par-



(a) Single Point Crossover (b) Uniform Crossover

Figure 2.3: Examples of crossover

ents at these positions.

2.2.6 Fitness Function

The task of the fitness function is to assign a numerical value to the phenotypes in the population. Two important aspects one must consider during the design of a fitness function is the choice of components in the fitness function and how the fitness is evaluated.

The components of a fitness function depends on the problem. For instance in design of a aircraft wing, several different factors should affect the fitness of the wing. For instance drag, lift, weight and the number of parts.

The choice of the fitness function is a difficult one. It could be based on knowledge about the search space and the relationships between the different components one must consider. However, the fitness function is often selected arbitrarily by trial and error and the experience of the designer.

2.2.7 Selection Mechanisms

Natural selection drives evolution in nature, while in the computer the selection mechanism does the same job. In nature selection is a combination of many things in the environment of the individuals: predators kill off some part of the population, some drown and others cannot find food. In the computer it is seldom this complex.

One method for selection is the *tournament selection* mechanism where one randomly selects a portion of the population and has a tournament among the selected

individuals. This is for instance used when one selects the parents for reproduction.

If we only selected the best individual for reproduction the children in the next generation would not have much variance and information contained in the previous generation that could lead to a better solution might be lost. The random selection of individuals for reproduction adds noise to the selection mechanism, and ensures that also average solutions has a possibility for reproduction, which keeps the population more diverse.

2.2.8 The Algorithm

The algorithm can be summarized as follows.

1. Initialize population of a given size of random candidate solutions
2. Evaluate: Compute fitness for each candidate solution in the population.
3. While <termination criteria> is not filled, do:
   (a) Select a fraction of the population kill them based on fitness.
   (b) Crossover: Select a fraction of the population, and recombine them as new candidate solution in the population.
   (c) Mutation: Select a fraction of the population and mutate them.
   (d) Evaluate: Compute fitness for each candidate solution in the population.

The *termination criteria* in the algorithm is some condition that ends the evolutionary process. This can be a variety of things, but often it makes the process end after a fixed number of iterations, or end the process when the population has converged to a fitness value.

2.3 Genetic Programming

GP is a specialization of GA described in Section 2.2. Rather than representing the candidate solutions as a string it is in GP represented as the syntax tree of the program. In Figure 2.4 we see the basic control flow of the algorithm, which contains the same steps as in an ordinary GA.

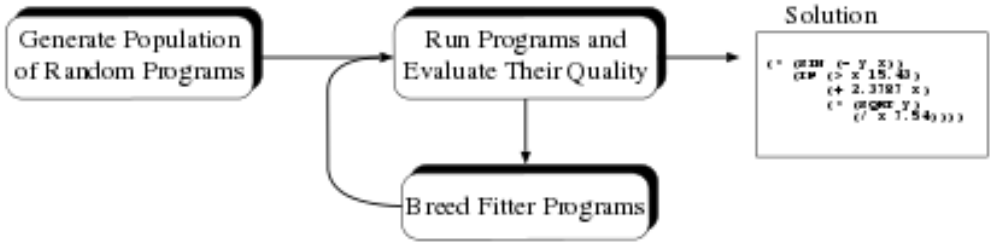


Figure 2.4: The basic control flow for genetic programming [22]

2.3.1 Representation

The program is represented as a syntax tree internally, and the genetic operators such as mutation and crossover works directly on this tree. In GP the leaf nodes are called *terminals* while the internal nodes are called *functions*. Functions have an *arity* which is the number of child nodes a function can have. And all nodes have a depth, which is defined as the number of edges one must traverse to reach the node from the root node. The root node has a depth of 0. The depth of a tree, is equal to the depth of the deepest node.

An example representation of a candidate solutions in GP is shown in Figure 2.5. The program in this case is a math formula, shown in Equation 2.7. The representation shown in Figure 2.5 is the syntax tree. The nodes a, b, c are the leaf nodes or the *terminals* in the program, while $+, -, \sqrt{}$ are the functions.

$$a + b - \sqrt{c} \tag{2.7}$$

2.3.2 Initialization

Similar to GA approach, in GP the population is randomly initialized. However, because the representation in GP are trees, one can not simply use a randomly generated string as the representation, and another approach is necessary.

The simplest methods of generating trees are the **grow**, **full** and **ramped half-and-half**.

The **full** method is illustrated in Figure 2.6. This method ensures that all the *terminals* are at a predefined depth. The tree is generated by adding random children until one has reach the predefined depth. This allows for randomly generated

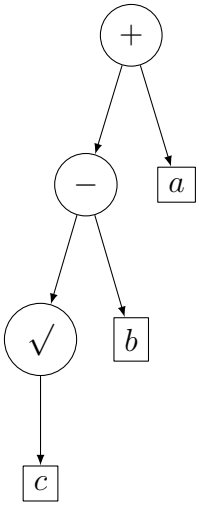


Figure 2.5: Example representation

trees. When the *functions* of a program have different *arity*, we cannot know the number of *terminals* the tree will end up having.

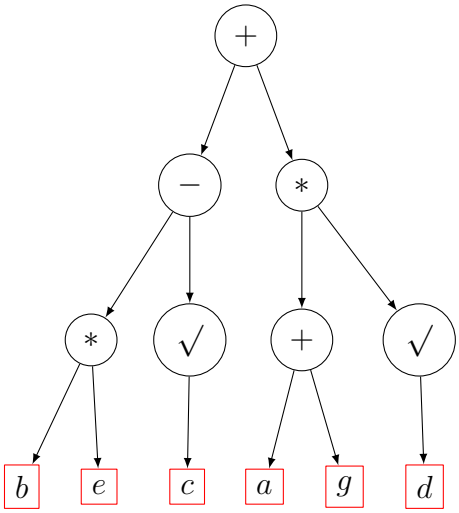


Figure 2.6: Full Tree of depth 3

In Figure 2.6 we have generated a tree using the full method, where all terminals are at a depth of 3. Note that we could not have known in before hand how many terminals the tree must end up having, because $\sqrt{}$ and $+$ have different arity.

The **grow** method is similar to the full method, but it does not promise that all terminals are all at the same depth. Instead it promises that the depth of the tree never exceeds a predefined depth, and allows terminals to be at any depth lower

than or equal to the predefined depth. An example of a tree initialized using the grow method with max depth of 3 is illustrated in Figure 2.7, where the red node is forced to be a terminal so that it does not violate the max depth limitation.

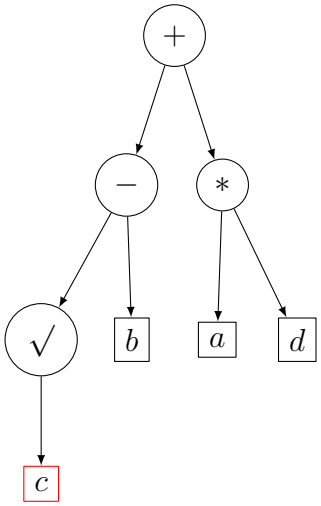


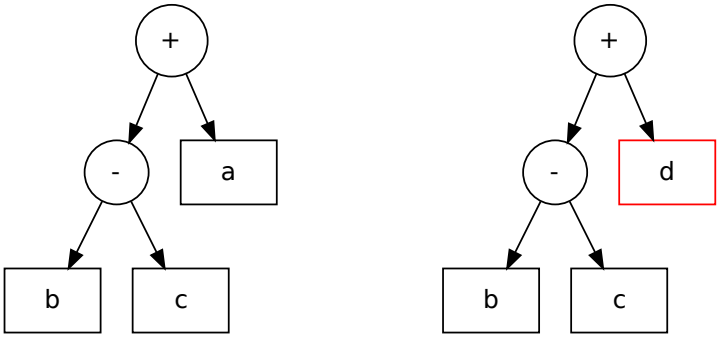
Figure 2.7: Growing Tree of depth 3

One weakness of these methods is that both methods have limitations on the trees they can represent. If all the instances in the population was generated using the grow method, the population would have mostly unbalanced trees. And when the population is generated using the full method, the population does not contain any unbalanced trees and only trees at the predefined depth. By generating half of the population with the grow method and the other half with the full method one can combat this limitation. This method is called **ramped half-and-half**.

2.3.3 Mutations

In GP one common method of mutation is a point mutation, it is roughly equivalent to the point mutation in GA (See Section 2.2.4). An illustration of this form of mutation is shown in Figure 2.8. The tree is mutated by looping through each node, and the node is mutated with a probability of p_m. In the example, the terminal a is mutated into another terminal d.

The most commonly used form of mutation in GP is the subtree mutation illustrated in Figure 2.9. This is done by randomly selecting a node in the tree, and replacing that node with a new randomly generated tree. This can be implemented by randomly initializing a new tree using the grow method from Section 2.3.2, and



(a) before (b) after

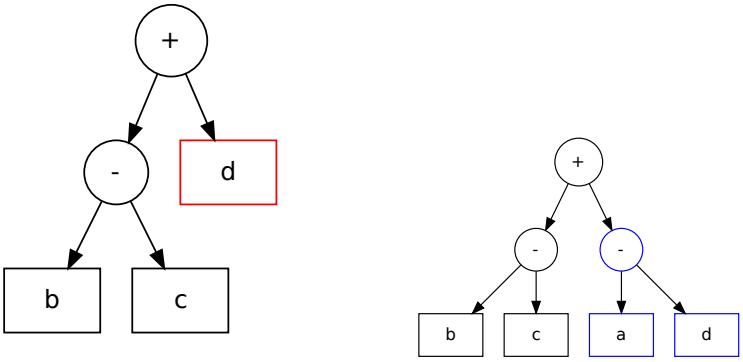
Figure 2.8: Point mutation in GP

replacing a random node with the newly created tree. In Figure 2.9 the node *d* is replaced by a newly generated subtree marked in blue.

2.3.4 Crossover

Crossover in GP is also similar to that of GA. In GP it is done by having two parents, and recombining them into children. However differently from GA crossover points are *not* selected at random. This is because the number of terminals is over represented in the tree, and when swapping leaf nodes very little from the parents are actually swapped. To counter this, it is suggested [22] that one should choose functions as the point to swap in the parents 90% of the time. Typically the portion of individuals in the new generation created by doing a crossover operation is around 90%. [13]

An example of a crossover operation is shown in Figure 2.10. In in both the mother and the father trees we select a node (according to the rules in the previous paragraph) to be swapped in their children. In both cases the internal node selected is the *subtract* function and the sections to be swapped is marked in red and blue. Then we create children by taking the root node from one parent and replacing the selected subtree with the selected subtree in the other parent. This can be done twise to create two children (shown in Figure 2.10c and Figure 2.10d)



(a) before (b) after

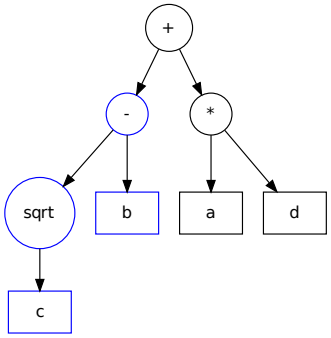
Figure 2.9: Subtree mutation in GP

2.3.5 Bloat

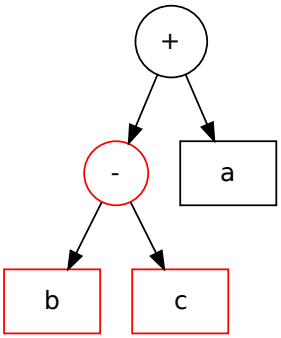
Researchers have noticed that after a certain number of generations the programs in the population starts to grow without the fitness of the individuals increasing significantly. There are situations where big programs are a good thing, because GP often start with small programs that evolve into fit bigger programs. Bloat occurs when the program grows without significant increase in the fitness. [22]

Bloat can lead to programs that generalizes badly by over fitting to the problem it tries to solve. Big programs may also be computationally expensive and hard to interpret.

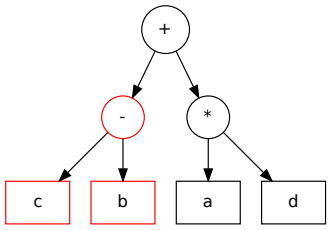
Many approaches to controlling bloat have been proposed, but a common approach is to limit the depth and size of the tree. This approach ensures that the tree never exceeds the constrains and prevents bloat, but one of the drawbacks is that the population often reaches that limit and is constrained from evolving to better solutions. So when one limits the tree one has to be sure that a good solution can be reached within the limits.



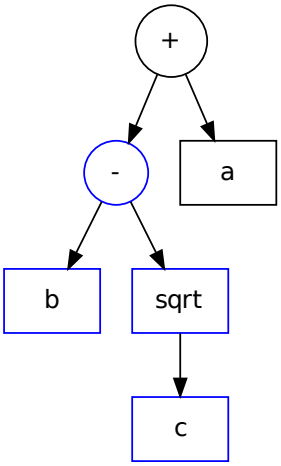
(a) Mother



(b) Father



(c) First Child



(d) Second Child

Figure 2.10: Single point crossover in genetic programming

2.4 Evaluating Classifiers

2.4.1 Cross Validation

Cross Validation [15] an empirical accuracy based method for evaluating classifiers. It is classifier neutral in that it does not need to know how the classifier is implemented. The accuracy is measured by calculating

$$\text{accuracy} = \frac{\text{correctly classified}}{\text{number of samples}} \tag{2.8}$$

Cross validation is done by partitioning the dataset $\mathbb{T}$ into n subsets $\mathbb{T}_1, .., \mathbb{T}_n$. Then loop over the subsets and use the subset as testing data and all other subsets as training data.

The special case of cross validation where $n = |\mathbb{T}|$ is called *leave one out*. It is computationally quite heavy because the classifier has to be run n times to get the classification accuracy, but it also yields the best prediction of how well the classifier is able to classify on the dataset because the classification accuracy is measured with $|\mathbb{T}| - 1$ samples in the training data. However, commonly a 10 fold cross validation is used when testing classifiers.

2.4.1.1 Overfitting

Overfitting occurs when the classifier describes the noise in the dataset instead of the underlying relationships. The assumption is that when the classifier learns how to classify on the training data that it will also learn how to classify new unknown samples, thus generalizing the solution to future classification tasks. However when the classification increases on the training data and decreases on the testing data the classifier is said to overfit.

2.5 Related Work

The *k*NN algorithm is a special case of the Variable Density Estimation. The Variable Density Estimator[23] can be used as a statistical approach for classification where they use different kernel functions to classify a test sample.

Several approaches has been done to rescale the input data. Two methods from the literature are the Large Margin Nearest Neighbour algorighm [26] and Neighbourhood Component Analysis (NCA)[10]. The former method learns the Mahalanobis distance for classification. The Mahalanobis distance differs from euclids distance in that it is elliptical. NCA creates a transformation of the input data in a way that minimizes the average *leave one out* cross validation accuracy on the test data.

Research has also been done to speed up the algorithm. One approach is to sort the training data in some way before classification to reduce the number of calculations needed to determine a class. Two such sorting algorithms and data structures are kd-Trees and Voronoi diagrams [16].

Normally when classifying a sample in kNN one must the distance from the sample to every point in the training data. By dataset reduction one can reduce the number of calculations needed to classify a sample. This is done by finding data points in the training data that can be removed without impacting the performance of the classifier. [3][27]

Chapter 3

Concept and Implementation

One of the limitations of the majority votie classifiaction rule in Section 2.1.1.3 is that they use the same distance function for all the classes. For instance if we are to classify *cars* and *trucks* it is not given that one single distance function yields the best classification accuracy. We know that cars almost always have four wheels, while trucks can have a wide range of possible wheel combinations. If we use the weighted Euclidean distance we cannot express that the number of wheels is more important when classifying cars than it is for trucks. Using the k-Mean classification rule, defined in te next section, it is possible to have different distance functions.

This leads us to the first research question:

- **RQ1** Are class dependant distance functions viable?

To be able to create an automatic classifier with class dependant distance functions we must be able to learn distance functions from the data. This leads to our second research question:

- **RQ2** How can distance functions be learned?

3.1 *k*-Mean Classification Rule

Landstad [14] showed during experiments that by using this classification rule when learning distance functions the genetic algorithm converged quicker while the accuracy of the classification was not affected.

This classification rule finds the *k*-nearest neighbours in each class to the unknown

sample, computes the mean distance to each class, and labels the unknown sample as the label with the shortest mean distance.

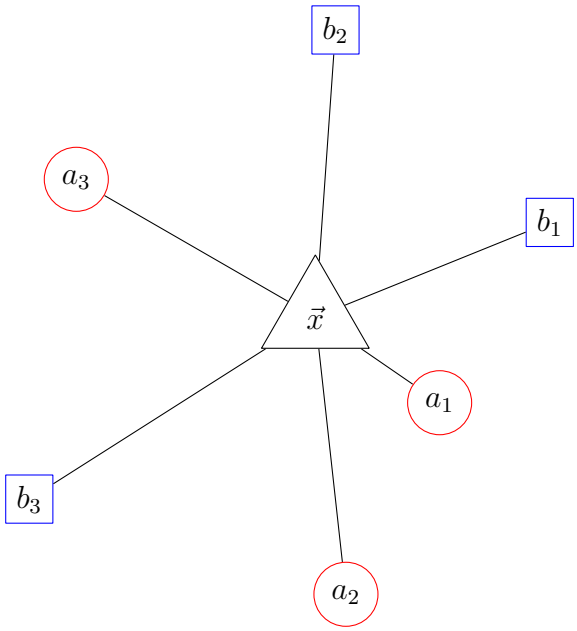


Figure 3.1: The k-Mean classification rule in a two class problem and $k = 3$. The edges are distances and the nodes are known samples. The triangle is the unknown sample

In the example shown in Figure 3.1 we have a two class classification problem with the classes $c = \{a, b\}$. When we try to classify an unknown sample $\vec{x}$ we collect the k nearest neighbours in each class shown as circles with their class label and subscript annotating their membership. The edges in the figure illustrate the relative distance to the unknown sample. To classify the sample we calculate the mean distance to the k nearest neighbours for each class, and assign the unknown sample the label with the lowest mean distance. In the example, $\vec{x}$ will be given the label a because the mean distance to a is the lowest of the two mean distances.

The mean distance to c_i given the unknown sample $\vec{x}$ can be expressed as follows. Given a distance function $d(\cdot, \cdot)$ and a matrix of the k nearest neighbours in for each class m.

$$d_i(\vec{x}) = \frac{\sum_{j=1}^{k} d(\vec{x}, \vec{m}_{ij})}{k} \tag{3.1}$$

Then one can express the k-Mean classification rule as follows.

$$\hat{c} = \arg\min_{c_i} d_i(\vec{x}) \tag{3.2}$$

3.2 A New Distance Function

Classification error in classification with *k*-Nearest Neighbours (*k*NN) decreases when the number of training samples approaches infinity [6]. So to increase the classification accuracy of *k*NN one can add more training data, but when more training data is not available other methods must be considered.

By changing the distance function in *k*NN we can increase the accuracy of the classifier. Normally the distance function in *k*NN is the Euclidean distance, where each feature is equally important. By applying weights, so that some features are more important than others, we can increase the accuracy of the classifier.[12]

In this section we define a new distance function that can be used in *k*NN.

3.2.1 Generalized Mean as Distance Function

The generalized mean, also known as power average, can be defined as follows[17][28]. Let $\vec{x}$ be a n dimensional vector of positive real numbers then the generalized mean is:

$$M_p(x_1, \ldots, x_n) = \left(\frac{1}{n} \sum_{i=1}^{n} x_i^p \right)^{\frac{1}{p}}, p \neq 0 \tag{3.3}$$

Remember that the Minkowski distance from Section 2.1.1.1 is as follows given two input n dimensional vectors $\vec{x}$ and $\vec{y}$ of positive real numbers is defined as follows where $1 \leq p \leq \infty$:

$$\text{dist}(\vec{x}, \vec{y}, p) = \left(\sum_{i=1}^{n} |x_i - y^i|^p \right)^{\frac{1}{n}} \tag{3.4}$$

k	p	md	dist
1	1	84.3%	84.3%
1	2	77.0%	77.0%
3	1	79.2%	79.2%
3	2	74.2%	74.2%

Table 3.1: Mean and Distances with the same power on the wine dataset

By defining a distance function for *k*NN with means where $1 \leq p \leq \infty$

$$\text{md}(\vec{x}, \vec{y}, p) = \left(\frac{1}{n} \sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}} \tag{3.5}$$

we do not loose classification accuracy when we replace it with the Minkowski distance with the powers $p = 1$ and $p = 2$.

In Table 3.1 we show this by classifying the wine dataset from the UCI Machine Learning Repository[9] with different powers does yield exactly the same classification accuracy. We use the powers $p = 1$ and $p = 2$ because they are the most used distance functions in *k*NN.

3.2.2 Properties of the Generalized Mean

By exploring the limits of the Generalized Mean over differences (MD) in Equation 3.5 it becomes easier to reason about distances.

When p approaches infinity the MD is the maximum difference between the features in two vectors [28]

$$\lim_{p \to \infty} \left(\frac{1}{n} \sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}} = \max_{i=1}^{n}(|x_i - y_i|) \tag{3.6}$$

We understand this as an **and** relationship between the differences. This is because when we let the maximum difference be a threshold t:

$$t = \max_{i=1}^{n}(|x_i - y_i|) \tag{3.7}$$

Then this holds true:

$$(|x_1 - y_1| \leq t) \wedge \cdots \wedge (|x_n - y_n| \leq t) \tag{3.8}$$

When p approaches negative infinity the MD is the minimum difference between the features in the two vectors[28].

$$\lim_{p\to-\infty}\left(\frac{1}{n}\sum_{i=1}^{n}|x_i-y_i|^p\right)^{\frac{1}{p}}=\min_{i=1}^{n}(|x_i-y_i|) \tag{3.9}$$

We understand this as an **or** relationship between the differences. This is because when we let the minimum difference be a threshold t:

$$t=\min_{i=1}^{n}(|x_i-y_i|) \tag{3.10}$$

Then this holds true:

$$(|x_1-y_1|\leq t)\vee\cdots\vee(|x_n-y_n|\leq t) \tag{3.11}$$

When p approaches 0 the MD becomes the geometric mean[28], and often one define the MD with $p=0$ to be the geometric mean.

$$\lim_{p\to0}\left(\frac{1}{n}\sum_{i=1}^{n}|x_i-y_i|^p\right)^{\frac{1}{p}}=\left(\prod_{i=1}^{n}|x_i-y_i|\right)^{\frac{1}{n}} \tag{3.12}$$

When $p=-1$ the MD is the harmotic mean [17]:

$$\text{md}(\vec{x},\vec{y},p)=\frac{n}{\sum_{i=1}^{n}\frac{1}{|x_i-y_i|}} \tag{3.13}$$

When $p=1$ the MD is the arithmetic mean [17][28]:

$$\text{md}(\vec{x},\vec{y},p)=\frac{1}{n}\sum_{i=1}^{n}|x_i-y_i| \tag{3.14}$$

When $p=2$ the MD is the Euclidean mean [17][28]:

$$\text{md}(\vec{x},\vec{y},p)=\sqrt{\frac{1}{n}\sum_{i=1}^{n}|x_i-y_i|^2} \tag{3.15}$$

3.2.3 A Set of Distance Functions

The distance functions we use in our approach is the Generalized Mean over distances in Equation 3.5. And with the properties of this function in mind we are able to define a set of distance functions that we will use in our approach.

Because when p approaches 0 the MD becomes the geometric mean we define our distance function $\text{md}(\cdot, \cdot, 0)$ to be the geometric mean.

$$\text{md}(\vec{x}, \vec{y}, p) \text{ defined to be } \left(\prod_{i=1}^{n} |x_i - y_i| \right)^{\frac{1}{n}} \text{ when } p = 0 \tag{3.16}$$

Thus the distance function we use in our approach is defined as follows. Given two vectors $\vec{x} \in \mathbb{R}^n$ and $\vec{y} \in \mathbb{R}^n$ where n is the dimensionality of the vectors and with powers p as $p \in \mathbb{R} \cup \{-\infty, \infty\}$.

$$\text{md}(\vec{x}, \vec{y}, p) = \begin{cases} (\prod_{i=1}^{n} |x_i - y_i|)^{\frac{1}{n}} & \text{if } p = 0 \\ (\frac{1}{n} \sum_{i=1}^{n} |x_i - y_i|^p)^{\frac{1}{p}} & \text{if } p \neq 0, p \in \mathbb{R} \\ \max_{i=1}^{n}(|x_i - y_i|) & \text{if } p = \infty \\ \min_{i=1}^{n}(|x_i - y_i|) & \text{if } p = -\infty \end{cases} \tag{3.17}$$

Also with the properties of the MD in mind from the previous section we also define a set of interesting powers p the distance function can take. We use integer valued p where $-1 \leq p \leq 2$ to allow for the distance function to be normal means. We use $p \in \{-\infty, \infty\}$ to have *and* and *or* relationships. Thus the set of all interesting values of p in our approach is defined to be:

$$p \in \{-\infty, -1, 0, 1, 2, \infty\} \tag{3.18}$$

When $p = -1$ it is a possibility for the distance function to become undefined. This happens when the difference between two features is equal to zero, and is because you cannot divide by zero in Equation 3.12. When this happens, to avoid fatal exceptions in the implementation we say that the distance between the two vectors is 0.

Note that when $p < 1$ we do not fulfill the identity requirement for metrics, but $d(\vec{x}, \vec{x}) = 0$ still holds. Also when $p < 1$ the triangle inequality property is violated. These properties still hold true, making it a pseudosemimetric:

- $d(\vec{x}, \vec{y}) \geq 0$; *non-negativity*
- $d(\vec{x}, \vec{x}) = 0$; *identity*
- $d(\vec{x}, \vec{y}) = d(\vec{y}, \vec{x})$; *symmetry*

3.2.4 Distance Tree

Normal distance functions like the Euclidean distance are not able to take into account *and* and *or* relationships. We propose to create a tree of distance functions that is able to encode those relationships.

For our tree to encode these relationships we use the set of distance functions from Section 3.2.3 as internal nodes in the tree, while using features as leaf-nodes in the tree. This can also be seen as a syntax tree of a program where the distance functions are *functions* while features are *terminals*.

For instance with two distance functions *Euclidean mean* and *arithmetic mean* and three features f_1, f_2 and f_3, a distance tree can be expressed as shown in Figure 3.2. Then the Euclidean mean in the figure is calculated as flows, where it only uses the features f_2 and f_3.

$$d(\vec{x}, \vec{y}) = \sqrt{\frac{1}{2} \sum_{i=2}^{n} |x_i - y_i|^2} \tag{3.19}$$

When one calculates the Manhattan distance in the figure, one calculates with the distance given by the Euclidean distance and the difference in f_1. Given the calculated Euclidean distance z the distance of the tree is:

$$d(\vec{x}, \vec{y}) = \frac{z + |x_1 - y_1|}{2} \tag{3.20}$$

3.2.5 Distance Tree Example

If we want to express similarity between a car and an unknown sample we can use the features: number of wheels, number of windows and color. We know that most cars have 4 wheels, have some number of windows and often share color with other cars. With this information it is possible to state that a something is similar to a car if it has similar number of wheels and has similar number of

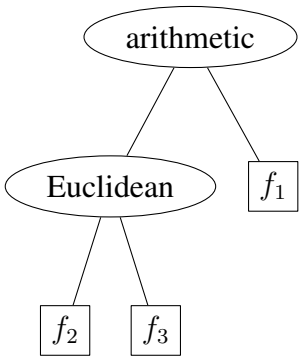


Figure 3.2: A distance tree

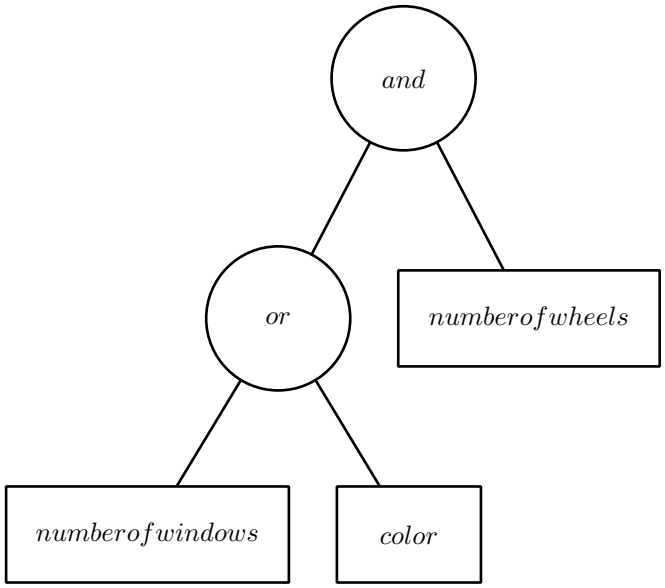
windows or similar color. This can be expressed as a simplified logical expression as follows:

$$\textit{number of wheels} \wedge (\textit{number of windows} \vee \textit{color code}) \tag{3.21}$$

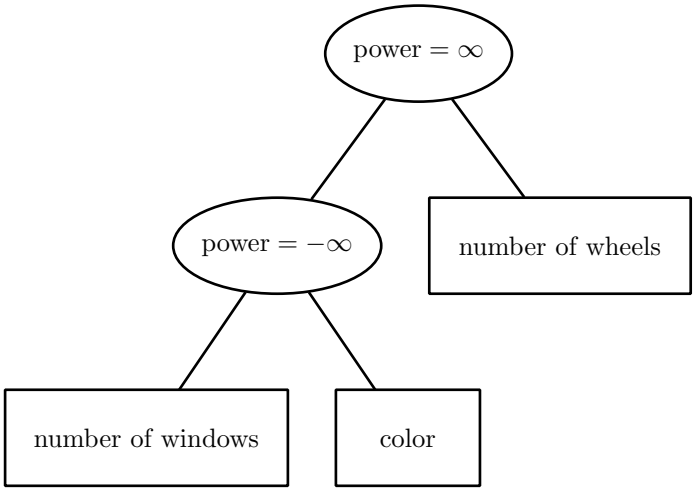
The expression above can be visualized as a tree, where the $\wedge$ and $\vee$ are internal nodes and the features is leaf nodes in the tree as shown in Figure 3.3a. By using powers of ∞ and $-\infty$ from Section 3.2.3 to express *and* and *or* relationships the same figure can also be used expressed with powers as shown in Figure 3.3b.

Figure 3.3b is the visual expression of the *and* and *or* relationships of the features, and roughly translates to the following distance function. Given $d_1 =$ *difference in number of wheels*, $d_2 =$ *difference in number of windows* and $d_3 =$ *difference in color*.

$$dist(car, sample) = max(d_1, min(d_2, d_3)) \tag{3.22}$$



(a) Tree Representation



(b) Distance Tree Representation

Figure 3.3: Equation 3.21 expressed as a tree and a distance tree

3.3 Learning Distance Trees

Our approach is similar to in that we optimize a distance function for each class in the dataset.

Classification in *k*NN is done by computing distances from the unknown sample to all known samples. And as in if we use one distance function for each class, the distances from known samples to the unknown sample is not necessarily comparable. However, we can mitigate this problem by optimizing distance functions for each class that *together* generalizes well on the dataset. Testing of generalization is done by testing the accuracy of the classification.

The problem we seek to solve is to find a set with one distance function for each class that together generalizes well on the dataset.

3.3.1 Genetic Algorithm

The genetic algorithm was implemented in the Java programming language with the genetic algorithm framework Jgap [19] and a machine learning framework called JavaML [4].

One methods for learning distance trees implemented in this report is using a Genetic Algorithm (GA).

3.3.1.1 Genotype

The genotype in the implemented solution is shown in Table 3.2. Each cell in the table contains an integer that is used to encode discrete values. The f_i represents features in the i-th dimension. And the number of features is dependent on the number of features in the training data. While p_g encodes the power in the g-th group. The superscript indicates which class each power and feature belongs to.

f_1^1	f_2^1	f_3^1	$\mathbf{p}_1^1$	$\mathbf{p}_2^1$	f_1^2	f_2^2	f_3^2	$\mathbf{p}_1^2$	$\mathbf{p}_2^2$

Table 3.2: Definition of the genotype for a two class problem with three features with two groups.

3.3.1.2 Phenotype

The phenotype of the genotype defined in the previous section is a set of distance trees with one distance tree for each class. A example genotype is shown in Table 3.3 where each **p** refers to a power in Equation 3.18 and each f represents which group the feature belongs to. In the example genotype, there are two groups, but in our implementation the number of groups g can be much larger: $g \in \mathbb{N}^+$ and $0 < g \leq n/2$. Where n is the dimensionality of the dataset.

In Figure 3.4 we show a graph representation of the two distance functions in Table 3.3. $\mathbf{p}_1$ is the power of the root node and $\mathbf{p}_2$ is the power of the subgroup. Each **p** node can have features or groups as children. When there is no features in the subgroup, simply that group is ignored in the phenotype.

f_1^1	f_2^1	f_3^1	$\mathbf{p}_1^1$	$\mathbf{p}_2^1$	f_1^2	f_2^2	f_3^2	$\mathbf{p}_1^2$	$\mathbf{p}_2^2$
0	1	1	0	6	0	0	0	2	3

Table 3.3: An instantiated chromosome for a two class problem with three features with two groups.

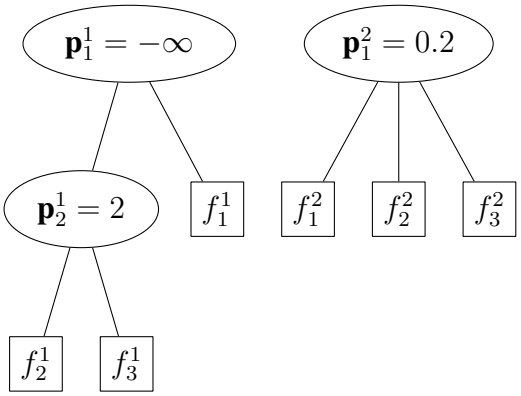


Figure 3.4: The phenotype of the instantiated genotype in Table 3.3 expressed as a Minkowski tree.

3.3.2 Mutation

The mutation of the genotype is mutated to a random valid value by a probability p_m. The valid values for features f when we have two groups are $f \in \{0, 1\}$ and the valid values of **p** are defined in Equation 3.18.

p_m in our implementation is 0.08, which is the default value in Jgap.

f_1^1	f_2^1	f_3^1	$\mathbf{p}_1^1$	$\mathbf{p}_2^1$	f_1^2	f_2^2	f_3^2	$\mathbf{p}_1^2$	$\mathbf{p}_2^2$
0	**0**	1	0	6	0	0	0	**9**	3

Table 3.4: The genotype in Table 3.3 with two mutations

A example mutation of the genotype in Table 3.3 into a mutated genotype is shown in Table 3.4. There are two mutations marked in bold. The tree representation of the trees in the genotype is shown in Figure 3.5.

As we can see from Figure 3.5 there is a group with only one feature. Groups with only one feature are legal but the internal node could be deleted without loss of information in the phenotype and the child node can be moved to the parent. This is because the distance is equal to the absolute value of the difference when the dimensionality of the vector is 1 as seen in Equation 3.23.

$$dist(\vec{x}, \vec{y}, p) = |x_1 - y_1| \tag{3.23}$$

Which is true if $\vec{x} \in \mathbb{R}^1$ and $\vec{y} \in \mathbb{R}^1$

However we do not delete nodes in the trees after mutations, because we do not want to remove information from the genotype about the previous solution. If at a later stage a node is mutated into the group again we still have information about another node in the group and the power of that group.

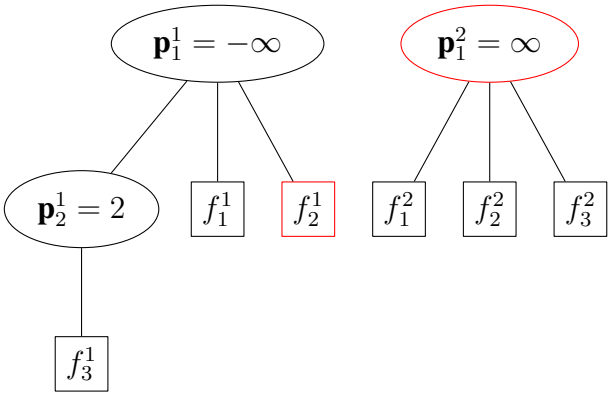


Figure 3.5: The phenotype of the mutated genotype in Table 3.4 expressed as a tree. The nodes altered are shown in red.

3.3.2.1 Crossover

The crossover operator chosen in our approach is the Single Point Crossover Operator (see Section 2.2.5), which selects a random point in the genotype of the mother in their children swaps the information in the genotype in around that crossover point. The crossover rate p_c is in our implementation 35%.

A example of how the crossover is done in our application is shown in Table 3.5 and their phenotype representation in Figure 3.6. We select two parents in the population as the mother and father marked as red and blue in the table. Then replace the parents in the population with their children. The source of the genetic material in the genotype of the children is marked in red and blue, and the crossover point is randomly selected to be between f_1^2 and f_2^2.

Definition:	f_1^1	f_2^1	f_3^1	$\mathbf{p}_1^1$	$\mathbf{p}_2^1$	f_1^2	f_2^2	f_3^2	$\mathbf{p}_1^2$	$\mathbf{p}_2^2$
Mother :	0	0	0	1	1	0	0	0	9	2
Father :	0	1	1	0	6	1	1	0	0	3
Child 1:	0	0	0	1	1	0	1	0	0	3
Child 2:	0	1	1	0	6	1	0	0	9	2

Table 3.5: A example of a crossover operation in GA

3.3.2.2 Fitness Function

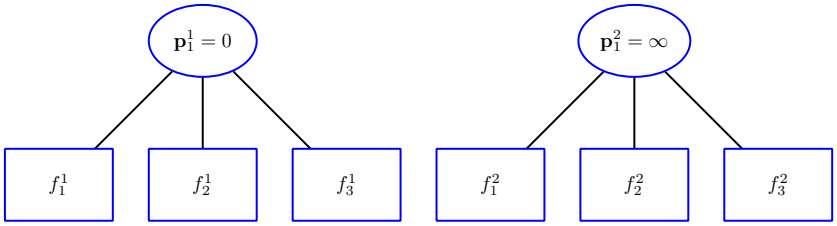
The fitness function used in this implementation is a 10 fold Cross Validation on the training data (See Section 2.4.1). The classification rule used during training was the *k*-Mean classification rule (See Section 3.1).

3.3.3 Genetic Programming

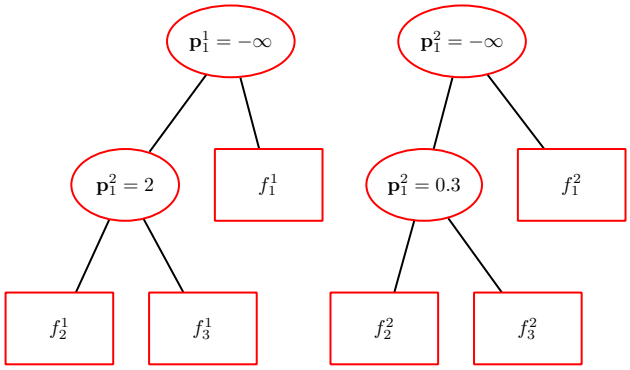
The Genetic Programming (GP) implementation was implemented in the Python programming language with a framework for GP called Pyevolve [21] and a machine learning library called mlpy [2].

Many variations of representation, limits and genetic operatiors in geneitc programming was tried, and the solution described in this section is the solution that produced the best resutls during development.

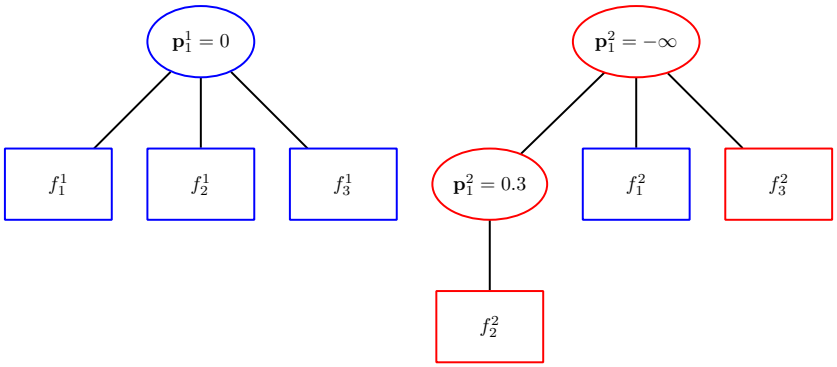
The trees that our GA solution is able to represent are limited to trees of depth 2. However the optimal solution might not be in that search space as with logical



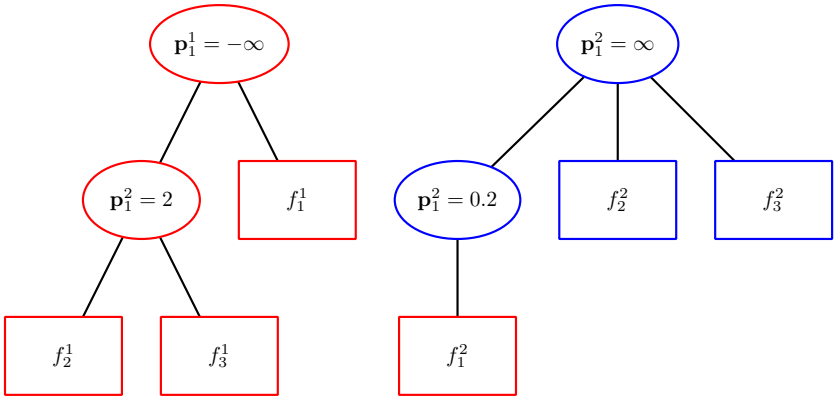
(a) Mother



(b) Father



(c) First Child



(d) Second Child

Figure 3.6: Single point crossover in the GA implementation

expressions one can have depths beyond two. In this Section we present a GP solution that tries to explore possible solutions with higher trees. The reason for it being a GP is that it is hard to represent higher trees with a two dimensional array.

3.3.3.1 Tree representation

Internally the tree is represented with lists and tuples, but logically they are of the same form as the trees in the GA solution. However we are able to represent trees of depth larger than two as shown in Figure 3.7.

In this implementation of the trees we do not require all the features to be present in the tree, and features might also appear several places in the tree.

To combat bloat we limit functions and terminals, in the implementation the maximum number of allowed internal nodes is 5, while the maximum number of allowed children of a node is term $= n$ where n is the dimensionality and a maximum depth of the tree of three.

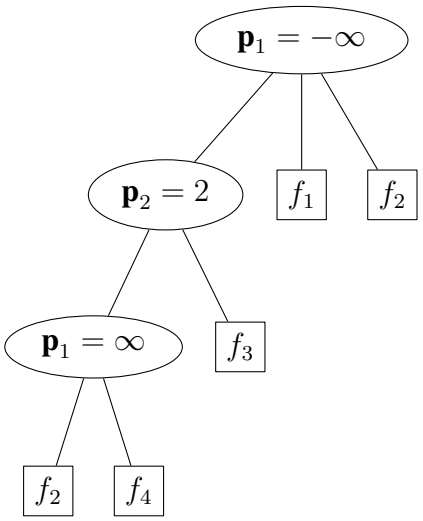


Figure 3.7: A example of a tree in our GP solution

3.3.3.2 Population initialization

The grow initialization method in our implementation differs from the normal initialization method in Section 2.3.2 in that it also limits the arity and maximum

number of functions. This is done by randomly giving the function an arity between 2 and $n/2$. We also limit the number of functions the tree can have to 5. The tree in Figure 3.7 is a example of a tree initialized with the grow method.

The reason for not choosing the full initialization method is that we wish to limit the internal nodes of the tree to something smaller than can be represented when initializing a full tree. When we set the depth to initialize to 3 we would need $2^3 - 1 = 7$ internal nodes when the minimum arity of a initialized function is 2. By omitting the full method we have more freedom in choosing limits to the tree.

3.3.3.3 Mutation

The mutation operator chosen in our implementation is the single point mutation, where each node in the tree is mutated to another node with a probability p_m. In our implementation $p_m = 0.1$. Internal nodes get a new randomly chosen power while feature nodes are randomly mutated into another feature.

The reason for not choosing the subtree mutation from Section 2.3.3 is because this has the possibility of adding a lot of new feature nodes that would lead to bloat. Also adding a new subtree to the solution is often a very large mutation that changes the fitness of the solution much that leads to a loss of the information in the previous tree.

An example of how the mutation operation in our solution works is shown in 3.8. Here there are two mutations one where the power is swapped for a new randomly selected power, and another mutation where the feature is swapped for a new randomly selected feature.

3.3.3.4 Crossover

Crossover of trees is done similarly to the solution shown in Section 2.3.4. But we also force children to follow the rules of the tree, so that the children do not exceed the maximum number of functions and a maximum depth.

This is done by selecting a node in the mother, and selecting a node in the father where the resulting children does not violate the max depth and the maximum number of functions. Implementation wise this is done by trying to find a valid crossover point in the father multiple times and if unsuccessful cancel the crossover operation.

The selection of nodes is not purely random, but internal nodes are selected 90% of the time. This is done to ensure that the crossover does not simply swap feature

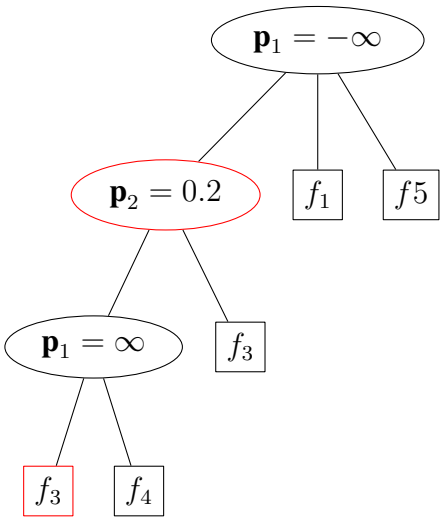


Figure 3.8: The tree in Figure 3.7 mutated

$\mathbb{T}^1$	$\mathbb{T}^2$	$\mathbb{T}^3$

Table 3.6: One tree for each class.

nodes as there are many more feature nodes in the tree than function nodes. By selecting internal nodes more often we will swap a larger portion of the tree from both the mother and the father.

3.3.3.5 Genetic Algorithm as a Container for Trees

To have one tree for each class in the solution we wrap the trees into a structure similar to the one used in our GA. An example of the genotype of this solution is shown in Table 3.6. Here each $\mathbb{T}$ represent a tree, and the genotype contains three trees.

The mutation operator works on this genotype, and mutates each tree with a probability of $p_m = 0.1$. Note that in each tree each node is also mutated with the probability p_m resulting in the true probability of a mutated node becomes 0.01 which is the same as the rate commonly used in literature.

The crossover rate in our implementation is 90% and we use a modified version of the uniform crossover from Section 2.3.4. We randomly select a mother and a father and then we randomly select a set of trees that should be crossed. For instance if we selected the set $\{\mathbb{T}^1\}$ to be crossed, we take $\mathbb{T}^1$ in the mother and crosses it with $\mathbb{T}^1$ in the father using the cross over operator from Section 3.3.3.4.

3.3.3.6 Fitness Function

The fitness function used in this implementation is a 10 fold Cross Validation on the training data (See Section 2.4.1). The classification rule used during training was the k-Mean classification rule (See Section 3.1).

Chapter 4

Evaluation and Discussion of Results

4.1 Datasets

To evaluate our distance tree learners we run them on different datasets from the UCI Machine Learning Repository [9]. We do not scale or pre-process the datasets before we learn the trees, but it has been shown that it can increase the classification accuracy by doing feature selection and scaling [14].

4.1.1 The MONKS-problem

The MONKS-problem [25] is a set training and test data used in a performance comparison of different learning algorithms in 1994. This is a set of binary classification problems, where each classification problem is generated by a logical rule. Each problem has a training set and a test set.

There are 6 discrete features describing robots annotated as f_i:

$$
\begin{array}{rcl}
f_1: & \text{head shape} & \in \{\text{round}, \text{square}, \text{octagon}\} \\
f_2: & \text{body shape} & \in \{\text{round}, \text{square}, \text{octagon}\} \\
f_3: & \text{is smiling} & \in \{\text{yes}, \text{no}\} \\
f_4: & \text{holding} & \in \{\text{sword}, \text{balloon}, \text{flag}\} \\
f_5: & \text{jacket color} & \in \{\text{red}, \text{yellow}, \text{green}, \text{blue}\} \\
f_6: & \text{has tie} & \in \{\text{yes}, \text{no}\}
\end{array}
$$

Instead of providing a complete class description, with all 432 possible examples of robots, the training set for the supervised classifier is a set of randomly selected robots.

4.1.1.1 Problem 1

$$(\text{head shape} = \text{body shape}) \wedge (\text{jacket color} = \text{red})$$

All robots that follow this rule have the *true* label, and all the robots not following this rule have the class label *false*. The training set consists of 124 randomly selected robots.

4.1.1.2 Problem 2

*exactly two of the six attributes have their **first** value*

For example a robot with *head shape* $=$ *body shape* $=$ *round* implies that the robot is not smiling, not holding a sword, jacket color is not red and has no tie.

All robots that follow this rule have the *true* label, and all the robots not following this rule have the class label *false*. The training set consists of 169 randomly selected robots.

4.1.1.3 Problem 3

$$((\text{jacket color} = \text{green}) \vee (\text{holding} = \text{sword})) \wedge ((\text{jacket color} \neq \text{blue}) \vee (\text{body shape} \neq \text{octagon}))$$

All robots that follow this rule have the *true*, and all the robots not following this rule have the class label *false*. The training set consists of 122 randomly selected robots, with 5% noise.

4.1.2 IRIS dataset

The iris dataset is a well known dataset in classification and is known as the Fisher iris dataset, that consists of 3 classes of the iris plant. There are 4 features of the plants that are length and width of pits spetal and petal. The dataset has 150 samples where there are 50 samples of each class.

	Problem 1		Problem 2		Problem 3	
distance functions	1	2	1	2	1	2
μ	96.99	97.38	78.40	79.73	95.05	97.41
σ	0.49	5.98	1.58	3.5	1.11	1.49
max	98.15	99.54	80.55	82.87	95.60	100

Table 4.1: Results from 20 runs on the MONKS-problem

4.1.3 Wine Dataset

The wine dataset is also a well known dataset in classification, and is collected from the UCI machine learning database[9]. It consists of 178 samples with 3 classes. The task is to classify the cultivars (the type of plant), of wine with 13 attributes of its chemical composition.

4.2 RQ1: Are class dependant distance functions viable?

To verify that using two distance functions might be viable we test it with two runs of the GA solution, where in one run we only learn one distance function for every class and another run where we learn two distance functions.

For every run of the GA we use a population size of 50, and evolve over 100 generations. The groups have the possible powers $p \in \{-\infty, -1, 0, 1, 2, \infty\}$ and a total number of functions in the tree to be 3 and $k = 3$.

The results shown in Table 4.1 is the classification average of 20 runs. We calculate the mean μ, and standard deviation σ of the classification on the test set. We also include the maximum classification accuracy produced by the algorithm.

4.2.1 Hypothesis test

By doing a two sample z-test on the problems, we can check whether using two distance functions can classify better than by using only one distance function for classification. We let the two distance function means classification accuracy be μ_1 and the mean classification accuracy from the one distance functions run be μ_2. We assume that the test results are Gaussian distributed.

Our claim is that classification with two distance functions is better than with only one $\mu_1 > \mu_2$. The null hypothesis H_0 and alternate hypothesis H_a is the following:

$$H_0 : \mu_1 - \mu_2 \geq 0 \tag{4.1}$$
$$H_a : \mu_1 - \mu_2 < 0 \tag{4.2}$$

We define a significance level $\alpha = 0.05$ with $z_\alpha = -1.645$, we reject the null hypothesis if $z < z_a$. We calculate z by:

$$z = \frac{\mu_1 - \mu_2}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}} \tag{4.3}$$

For instance when we calculate the z-value for the first problem:

$$z = \frac{97.38 - 96.99}{\sqrt{\frac{5.98}{20} + \frac{0.49}{20}}} \tag{4.4}$$
$$z = -0.747 \tag{4.5}$$

The z-value for each of the problems are:

- Problem 1: $z = -0.747$
- Problem 2: $z = -1.732$
- Problem 3: $z = -5.667$

In the first problem we cannot reject the null hypothesis with significance level $\alpha = 0.05$. Thus given the training and test set using one distance function is equally good or better on average than using two distance functions.

In the second and third problems we reject the null hypothesis with significance level $\alpha = 0.05$. This means that learning two distance functions in these setups is on average better than using only one.

4.2.2 Discussion

We know that we can represent any single distance function for a two class problem with two identical class dependant distance functions. Thus we know that if

there is a optimal single distance function for classification of the dataset, the optimal class dependant distance functions are equally good or better. The problem is finding the optimal distance function.

From Table 4.1 we see that the maximum classification accuracy differs between the two approaches, and that using two distance functions have a higher maximum classification accuracy. This also indicates to us that the optimal solution with two distance functions is better, however we can not be sure that the distance functions found by the algorithm is the optimal solutions.

In the results from the MONKS-problem, we see that the standard deviation is in general higher with two distance functions. This indicates that the algorithm has problems finding the optimal solution in the search space. The search space on this binary classification problem is doubled because we have two distance trees.

Specially in the first problem is the standard deviation high, during training the standard deviation was 1.3 with a mean of 98.4. This shows us that two distance functions when finding a good solution on the training data can create overly complex rules that can have problems in generalizing. This can be mitigated by reducing the complexity the trees can represent.

In the first problem the average classification accuracy for both of the approaches is about the same, we think that this is due to the fact that the rule is very simple and fits with the *and* and *or* relationships encoded in our trees.

In the third problem we have 5% noise, and the solution with one distance function looses accuracy close to this, as one would expect because the noise would affect the classification accuracy when incorrect data is evaluated during classification. Interestingly enough, two distance functions does not seem to be affected as much by noise in the training data, and the average classification accuracy on the training data is 94.63%. That it is able to classify better on the testing data can be explained by the fact that two distance functions are able to separate the classes better than a single distance function does.

4.3 RQ2: Learning distance trees

	Problem 1	Problem 2	Problem 3
KNN	83.10%	70.14%	85.65%

Table 4.2: *k*NN classification accuracy on the MONKS-problems

To benchmark the results on the MONKS-problems, we try run the k-Nearest Neighbours classifier on the same datasets. Shown in Table 4.2. The results was generated with our k-Mean classification rule and $k = 3$. And as shown, we beat *k*NN by more than 9% on every problem, which shows that the distance trees we learn with our GA approach are in fact able to increase the classification rate. And because the difference between the GA runs, with one and two distance functions, and *k*NN is so large, we believe that our new distances function is the main contributing factor for the good classification accuracies.

4.3.1 Runs on the wine and iris datasets

To compare the two learning algorithms GA and GP we run our algorithms on the wine and iris datasets. The accuracy measurements of the classifiers shown in Table 4.3 was generated from 3 runs of 3 fold cross validation. The trees were learned with a population size of 100 and 100 generations. The GA solution had max groups of 5 and the GP solution had a limit of 5 internal nodes and the max depth of the trees was set to 3. The results are shown in table 4.3

	Testset					Training			
	k	*k*NN	GA	GP	WKNN	KNN	GA	GP	WKNN
wine	3	71.0	87.3	90.6	96.7	71.0	94.7	97.4	99.5
iris	3	96.1	94.7	94.9	94.7	96.0	97.8	98.4	96.7

Table 4.3: Average classification results in percent.

As a benchmark for the classification we include two classifiers, the *k*NN classifier and a Weighted *k*-Nearest Neighbour Classifier (WKNN). Both classifier use our k-mean classification rule.

4.3.2 Weighted *k*-Nearest Neighbour Classification

Kelly and Davis [12] showed that by applying scales to the distance function in *k*NN one can improve the classification accuracy. In this Section we describe an algorithm in the spirit of their work.

The weighted Euclidean distance is defined as follows. Given two input vectors $\vec{x} \in \mathbb{R}^n$ and $\vec{y} \in \mathbb{R}^n$ and let $\vec{w}$ be a n dimensional vector of real numbers between

0 and 1. n is the dimesionality of the vectors.

$$d(\vec{x}, \vec{y}) = \sqrt{\sum_{i=1}^{n} w_i (x_i - y_i)^2} \tag{4.6}$$

The vector $\vec{w}$ is learned with a genetic algorithm where one optimizes for a classification accuracy.

4.3.2.1 Weight Learning using a Genetic Algorithm

The **genotype** of the application is in our implementation the weight vector of integers between 1 and 20, weights are later forced between 0 and 1 by dividing by 20. However unlike Kelly and Davis [12] we do not include k in the genotype.

Otherwise the implementation is exactly the same as our GA solution in Section 3.3.1, that is the same mutation rate, crossover rate, and fitness function.

4.3.3 Results and Discussion

The GA and GP solutions show a much higher accuracy rate than *k*NN on the wine dataset. This verifies that the distance trees we are able to learn from the data are also in this case better for classifying. However those classifiers are not able to beat the Weighted *k*-Nearest Neighbours (WKNN) classifier.

The difference in classification accuracy between our two learning methods for trees can be that in the GA solution a feature can appear several times in the tree which adds a possibility for extra weighting of some features in the tree. However, the GP solution is still able to represent exactly the same tree structure that is allowed by the GA solution, so we do not expect he optimal trees in the search space to classify any worse. The GP solution is also able to represent other tree configurations such as deeper trees, which makes the search space larger.

The larger search space in the GP solution can lead to more overfitting to the training data, which means that the trees it finds might not beat the GA solution in practice.

The GP solution is also able to do feature selection differently on each of the classes separately, so that one feature can be removed when measuring distances to one class, while having that feature in the distance tree for another class. This could possibly be a problem, because higher dimensional distance functions will

generally yield higher distances between two vectors. However we did not expect, and do not see from the results, that this had a negative effect on the classification accuracy. This is because we optimize for classification accuracy on the dataset, so the resulting distance trees are the ones that result in the best classification accuracy regardless of the dimensionality of the trees.

We see from the results that both classifiers have problems generalizing the results from training, in that the classification accuracy is much higher during training than it is on the test set. This is likely due to the fact that the trees are can become very complex, and fit to irrelevant information in the training data. Different distance functions for each class can also be a factor in over fitting, because different distance functions for each class might not be a better solution than having only one.

As we see from the results on the iris dataset, we also see the same problem with properties of our search space. Our distance trees are able to encode a distance function that should classify exactly the same as the *k*NN solution, but the trees it finds does not have the same classification accuracy. Judging by the training accuracy on the training data, we see that it finds trees that classifies *better* on the training data than a tree that we know generalizes well, and we have over fitting.

4.3.4 Evaluation of the running times

The time spent on the CPU for learning on the wine dataset was in the GA solution, implemented in Java, approximately 5 minutes. The dataset is very small with only 178 samples, so the time spent on the CPU seems unreasonable. The time spent can be explained because in every generation in our learned every distance function in the population must be evaluated by the fitness function. The fitness function we initially used was the leave one out cross validation that computes n^2 distances. With 100 generations and a population size of 100, the total number of distance calculations is $100 * 100 * n * n$ where n is the number of samples.

To reduce the number of calculations we replaced the fitness function with the 10 fold classification rule. This reduced the number of calculations with 10%.

The GP solution is implemented in Python and the distance calculations are very slow. In this case to speed up the process of generating results we created a parallelized implementation. Without the parallelization the calculation of of accuracy for one run would on the dataset takes 3 hours on my laptop, but by utilizing a powerful computer with 16 cores the algorithm runs 16 times faster. We also stop the evolutionary when the algorithm have not found a better solution the last 8 generations.

Chapter 5

Conclusion

In this thesis we show that by using class dependant distance functions we can increase the classification accuracy in *k*NN. We also introduce a new distance function that is a tree of multiple distance functions, that is able to encode logic and weighting of features. We show two evolution based learners for the new distance function, and both approaches are able to learn distance functions that increases the classification accuracy when used in *k*NN.

However, drawbacks with our approach are that it can over fit to the training data, and the distance functions are computationally expensive to learn.

5.1 Further Work

Optimization of speed of the learning algorithms should be explored further, so that it can be an alternative to other nearest neighbours related tasks. Parallelization of the algorithm over multiple machines could be a reasonable approach.

It would be interesting to explore the possibility for using learned distance functions in regression, and do something similar to linear regression. One can also split the dataset into several partitions and learn a distance function for each split.

Further analysis of the class dependant distance functions in regards to over fitting is needed, as we have only tested this algorithm on several small datasets with few classes.

It could also be interesting to implement more distance functions into this approach for a more complete set of logical functions. One could for instance implement a *not* operator into the set of possible distance functions.

Bibliography

[1] Antti Ajanki. Example of k-nearest neighbour classificationn, 2007. URL `http://commons.wikimedia.org/wiki/File:KnnClassification.svg`. [Online; accessed 7-december-2012].

[2] Davide Albanese, Roberto Visintainer, Stefano Merler, Samantha Riccadonna, Giuseppe Jurman, and Cesare Furlanello. mlpy: Machine learning python, 2012.

[3] F. Angiulli. Fast condensed nearest neighbor rule. In *Proceedings of the 22nd international conference on Machine learning*, pages 25–32. ACM, 2005.

[4] G.J. Badros. Javaml: a markup language for java source code. *Computer Networks*, 33(1):159–177, 2000.

[5] S.D. Bay. Combining nearest neighbor classifiers through multiple feature subsets. In *Proceedings of the Fifteenth International Conference on Machine Learning*, volume 3, pages 37–45. Citeseer, 1998.

[6] T. Cover and P. Hart. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, 13(1):21–27, 1967.

[7] KAJ Doherty, RG Adams, and N. Davey. Unsupervised learning with normalised data and non-euclidean norms. *Applied Soft Computing*, 7(1):203–210, 2007.

[8] D. Floreano and C. Mattiussi. *Bio-inspired artificial intelligence: theories, methods, and technologies*. The MIT Press, 2008.

[9] A. Frank and A. Asuncion. UCI machine learning repository, 2010. URL `http://archive.ics.uci.edu/ml`.

[10] J. Goldberger, S. Roweis, G. Hinton, and R. Salakhutdinov. Neighbourhood components analysis. 2004.

[11] A. Herrel, K. Huyghe, B. Vanhooydonck, T. Backeljau, K. Breugelmans, I. Grbac, R. Van Damme, and D.J. Irschick. Rapid large-scale evolutionary divergence in morphology and performance associated with exploitation of a different dietary resource. *Proceedings of the National Academy of Sciences*, 105(12):4792–4795, 2008.

[12] J.D. Kelly and L. Davis. A hybrid genetic algorithm for classification. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 645–650, 1991.

[13] J.R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112, 1994.

[14] Andreas Ståleson Landstad. Enhanced similarity matching by grouping of features. 2012.

[15] N. Lavesson. *Evaluation and Analysis of Supervised Learning Algorithms and Classifiers*.

[16] D.T. Lee. On k-nearest neighbor voronoi diagrams in the plane. *Computers, IEEE Transactions on*, 100(6):478–487, 1982.

[17] T.P. Lin. The power mean and the logarithmic mean. *The American Mathematical Monthly*, 81(8):879–883, 1974.

[18] D.O. Loftsgaarden and C.P. Quesenberry. A nonparametric estimate of a multivariate density function. *The Annals of Mathematical Statistics*, pages 1049–1051, 1965.

[19] K. Meffert, N. Rotstan, C. Knowles, and U. Sangiorgi. Jgap-java genetic algorithms and genetic programming package. *URL: http://jgap. sf. net*, 2008.

[20] Tom M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997. ISBN 0070428077.

[21] C.S. Perone. Pyevolve: a python open-source framework for genetic algorithms. *ACM SIGEVOlution*, 4(1):12–20, 2009.

[22] R. Poli, W.B. Langdon, and N.F. McPhee. *A field guide to genetic programming*. Lulu Enterprises Uk Limited, 2008.

[23] G.R. Terrell and D.W. Scott. Variable kernel density estimation. *The Annals of Statistics*, 20(3):1236–1265, 1992.

[24] A.C. Thompson. *Minkowski geometry*. Cambridge University Press, 1996.

[25] S.B. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. De Jong, S. Dzeroski, S.E. Fahlman, D. Fisher, et al. The monk's problems a performance comparison of different learning algorithms. 1991.

[26] K.Q. Weinberger, J. Blitzer, and L.K. Saul. Distance metric learning for large margin nearest neighbor classification. In *In NIPS*. Citeseer, 2006.

[27] D.L. Wilson. Asymptotic properties of nearest neighbor rules using edited data. *Systems, Man and Cybernetics, IEEE Transactions on*, (3):408–421, 1972.

[28] R.R. Yager. The power average operator. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 31(6):724–731, 2001.