Multi-objective Optimization for Support Vector Machines

by

Daan van Turnhout (SNR: 2051976)

A thesis submitted in partial fulfillment of the requirements for the degree of Bachelor in Econometrics and Operations Research

Tilburg School of Economics and Management
Tilburg University

Supervised by: Juan C. Vera

Date: January 14, 2024

Abstract

Support vector machines (SVM) are a popular method for binary linear classification problems, but applications for non-linear classification is also possible with the use of the Kernel Trick. In essence, the SVM problem is a bi-objective optimization problem that can be solved in different ways. In this thesis we will explore a Genetic Algorithm (GA) that can generate solutions that are located on the Pareto Front for the SVM problem. Solutions on this front are called pareto optimal and have the property that you can only improve one of the objective functions by deteriorating at least one other objective function. The proposed algorithm with the choices that are made in this thesis are then tested on two different datasets to asses its performance. Sensitivity analysis is done on the algorithm choices in order to observe the effects on the SVM problem. And we find that the proposed GA can definitely give good solutions to the SVM problem, but much more research needs to be done on the algorithm choices before we can obtain the true Pareto Front.

Contents

Chapter 1

Introduction

This thesis is about multi-objective model selection for the support vector machine optimization problem. Multi-objective optimization, abbreviated as MOO, is part of optimization theory in which a specific problem has multiple objective functions that need to be optimized, but they might be in conflict with each other. Increasing the objective value of one of the objectives might worsen the objective value of another one. To tackle this problem, we will be looking at pareto optimal solutions. The pareto solutions form a set called the Pareto Front and solutions in this set have the property that you can only improve one of the objective functions by deteriorating at least one other objective function. In the first chapter we give a short introduction into what a support vector machine is and what its applications are. In the second chapter we give the mathematical formulation for the support vector machine problem. In the third chapter we take a look at a more complex variant of the support vector machine problem by introducing a second objective function and we will introduce MOO theory. In the fourth chapter we will be looking at an algorithm to approximate the pareto front and show what the impact of the algorithm choices are in order to select the appropriate support vector machine model.

1.1 Support Vector Machines

Support vector machines (SVM) are used in machine learning to classify data into two categories, this is called a binary classification model. SVM are supervised learning models, this means that the model is trained on sample data together with the desired output to approximate a function. This function can then map new unlabeled data to one of the two output classes. Hence, the approximated function can classify unlabeled data into the two categories.

In general, the problem we are trying to solve with SVM is the following: Given certain data $x_i \in \mathbb{R}^n$ and two disjoint sets A and B. We want to find the decision rule that classifies x_i into either set A or set B. We do this by

separating all x_i with a $(n-1)$-dimensional hyperplane such that all vectors that belong to set A are on one side of the hyperplane and all vectors that belong to the set B are on the other side of the hyperplane. After we have trained the model with our training data, we can classify new unlabeled data with the hyperplane as the decision boundary, meaning that new data x_j will be classified into one of the two sets depending on the relative position to the hyperplane. For example, in 2-dimensional space, this means that the new data x_j will belong to set A if the point is above the hyperplane and new data x_j will belong to set B if the point is below the hyperplane.

The construction of the SVM strongly depends on the training data that we use. We can have three different types of training data X. Data in which we can draw a hyperplane that separates the two sets, as shown in figure (1.1). Data in which we can draw a curve that separates the two sets, as shown in figure (1.2). And data which is not separable into two disjoint sets, as shown in figure (1.3).

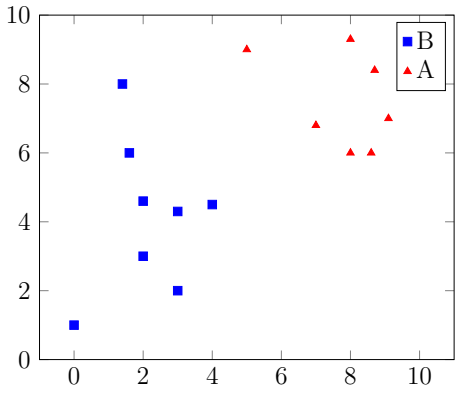

Figure 1.1: Separable by a straight line

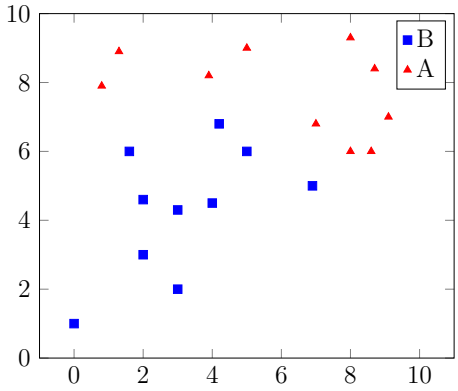

Figure 1.2: Separable by a curve

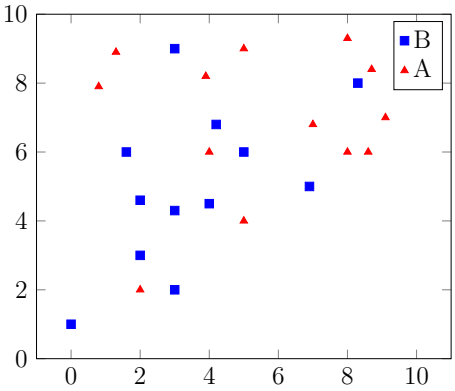

Figure 1.3: Not separable

Now comes the question, where do we draw the hyperplane H, this can be done in many different ways as shown in figure 1.4. All are valid decision boundaries, but some might be *better*. To determine which hyperplane we should use, we look at the generalization error of the model. The generalization error is the ability of the model to predict the class of new data. We want to set H such that the distance to the *closest* two datapoints of the two sets is maximum, this distance is called the margin. Determining H in such way implies that the distance between the two sets is maximum and such hyperplane is called a maximum-margin hyperplane.

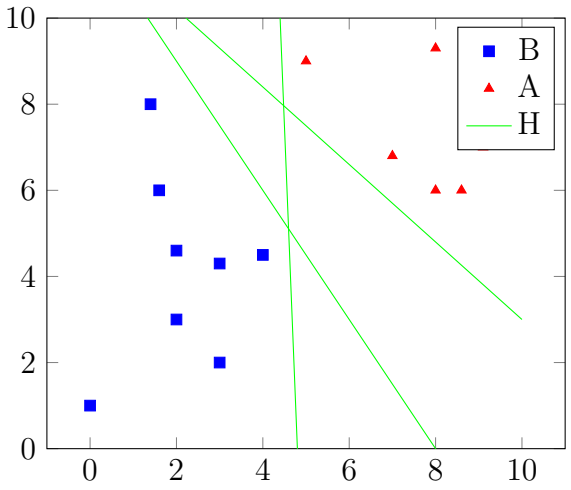

Figure 1.4: Possible locations for H

We also define two other distinct hyperplanes S_A and S_B, which will be called the support vectors of set A and B respectively. These hyperplanes are parallel to H and both have the same distance γ from H.

1.2 Applications of SVM

Support vector machines are a popular technique for (non) linear binary classification. Which has many applications. For example, SVM can be used in text categorization. Sorting vast amounts of text documents into predefined categories can reduce the retrieval time for users in large databases and help users find similar documents (Basu, Walters, & Shepherd, 2003). Another application of SVM is the recognition of handwritten digits, and specifically strings of handwritten digits. Which is generally harder than single digit recognition because of overlapping digits or the fact that there is an unknown number of digits in the string (Oliveira & Sabourin, 2004). Support vector machines can also be used for facial expression recognition, which can be used to determine a persons inner emotions or intentions. (Doiphode & Sapkal, 2018) Multiple papers have been written about improving the complexity and precision of the SVM, namely genetic algorithms such as *nondominated sorting genetic algorithm II* , that has shown in simulations to be more precise than conventional approximation methods. (Yuan & Jiangang, 2005)

Chapter 2

Linear separable SVM

In this chapter, we will only consider training data that can be linearly separable by a hyperplane as shown in figure (1.1).

2.1 Model Formulation

Let $H : w^\intercal x + b = 0$ be the decision boundary
Let X denote the training data consisting of vectors $x_1, ..., x_p$

Then for each $x_i \in X$ we can calculate the distance d_i from the decision boundary H with:

$$d_i = w^\intercal x_i + b \tag{2.1}$$

Let y_i be such that

$$y_i = \begin{cases} 1 & \text{if } x_i \in A \\ -1 & \text{if } x_i \in B \end{cases} \tag{2.2}$$

We define the margin γ_i of $x_i \in X$ as

$$\gamma_i = y_i(w^\intercal x_i + b) \tag{2.3}$$

Now we can define the *closest* vector to the hyperplane H as $\gamma = \min_i \gamma_i$
We want to find a hyperplane such that γ will be maximum, hence this will be our objective function. The only constraint we have is that all margins of our training data x_i should be larger than the margin of the vector that is *closest* to H. So the model becomes:

$$\max_{\gamma,w,b} \quad \gamma \tag{2.4}$$

$$\text{s.t.} \quad y_i(w^\intercal x_i + b) \geq \gamma \qquad \forall i \tag{2.5}$$

This formulation still has a problem, if we scale w with a constant $a \in \mathbb{R}$ in (2.3) we get:

$$y_i((aw)^\intercal x_i + b) = y_i(aw^\intercal x_i + b) = ay_i(w^\intercal x_i + b) = a\gamma_i$$

Hence, if we scale w with a constant $a \in \mathbb{R}$, then the margin changes to $a\gamma_i$. And this will also affect γ, which is the objective value of our model. So the current optimisation model will scale w to obtain a higher objective value. To prevent this, we use the normalised w in the calculations for the margins by substituting $w = \frac{w}{||w||}$ into the model. This means that the norm of w will always be equal to one.

The two support vectors are now defined as $S_A : w^\intercal x + b = 1$ and $S_B : w^\intercal x + b = -1$, because we normalized w. Then the followings holds:

$$w^\intercal x_i + b \geq 1 \quad \forall x_i \in A \tag{2.6}$$

$$w^\intercal x_i + b \leq -1 \quad \forall x_i \in B \tag{2.7}$$

Substituting y_i from equation (2.2) in (2.6) and (2.7) results in:

$$y_i(w^\intercal x_i + b) \geq 1 \quad \forall x_i \in X \tag{2.8}$$

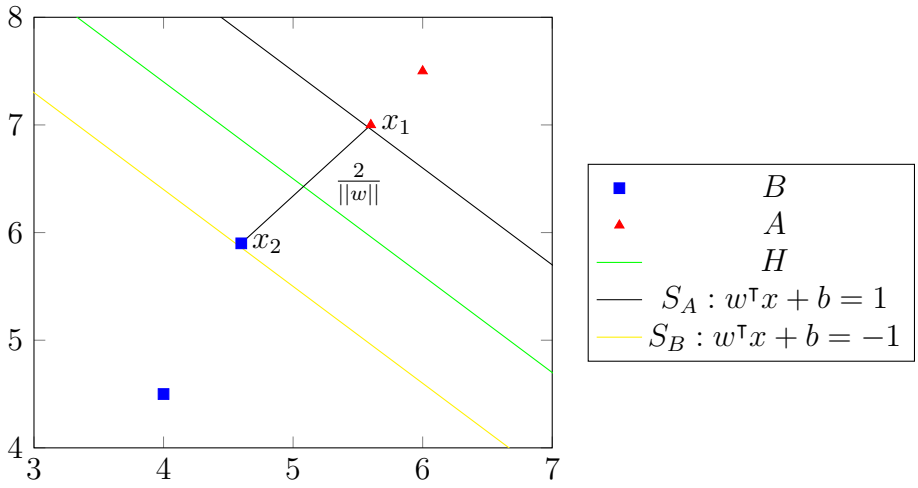

Figure 2.1: SVM example

Assume we have two points x_1 on S_A and x_2 on S_B as shown in figure 2.1
Then $x_1 = x_2 + 2\gamma\frac{w}{||w||}$
We also know from the definition of S_A:

$$w^\intercal x_1 + b = 1 \tag{2.9}$$

Then substituting $x_1 = x_2 + 2\gamma\frac{w}{||w||}$ in (2.9) results in:

$$w^\intercal(x_2 + 2\gamma\frac{w}{||w||}) + b = 1$$

$$w^\intercal x_2 + b + 2\gamma\frac{w^\intercal w}{||w||} = 1$$

$$-1 + 2\gamma\frac{w^\intercal w}{||w||} = 1$$

$$\implies \gamma = \frac{||w||}{w^\intercal w} = \frac{1}{||w||}$$

From optimization theory we know that maximizing γ is equivalent to minimizing $||w||$ which is equivalent to minimizing $\frac{1}{2}||w||^2$. Now the model becomes:

$$\min_{w,b} \quad \frac{1}{2}||w||^2 \tag{2.10}$$

$$\text{s.t.} \quad y_i(w^\intercal x_i + b) \geq 1 \qquad \forall i \tag{2.11}$$

This model is called SVM with hard constraints.

2.2 Evaluating the model

We can evaluate the model using cross-validation. Typically this means that you divide the data X into two parts, the training set and the validation set. First you train the SVM model using only the training data, and then you evaluate the model using only the validation set. Cross validation is used to test the ability of the model to predict the unseen data. To visualize this we use the *confusion* matrix:

	Predicted Class	
Actual Class	A	B
A	True Positive (TP)	False Negative (FN)
B	False Positive (FP)	True Negative (TN)

Table 2.1: Confusion matrix

Where $|A| = P$ and $|B| = N$

Then we define the following:

$$\text{sensitivity(TPR)} = \frac{TP}{P}$$

$$\text{specificity(TNR)} = \frac{TN}{N}$$

$$\text{precision(PPV)} = \frac{TP}{TP + FP}$$

$$F_1\text{-score} = 2 * \frac{PPV * TPR}{PPV + TPR} \in [0, 1]$$

We then evaluate the model based on the F_1-score, which is a harmonic mean of the precision and the sensitivity of the model. The higher the F1 score, the better the model is.

Chapter 3

Non linear separable data

In the previous chapter we discussed the hard margin model as formulated below, in which we only want to maximize the margin.

$$\min_{w,b} \quad \frac{1}{2}||w||^2 \tag{3.1}$$

$$\text{s.t.} \quad y_i(w^\intercal x_i + b) \geq 1 \qquad \forall i \tag{3.2}$$

But now we will be considering the case in which the training data is not separable by a linear hyperplane, as shown in figure (3.1). We still want to draw a hyperplane through the data, but this will always cause some points to be miss classified. We need to change the model in order to allow for miss classifications, because constraints (3.2) will not be satisfied if there is a data point that is on the wrong side of the hyperplane H. In order to do that, we have two assumptions to make. How do we define a miss classification and how to penalize a miss classification.

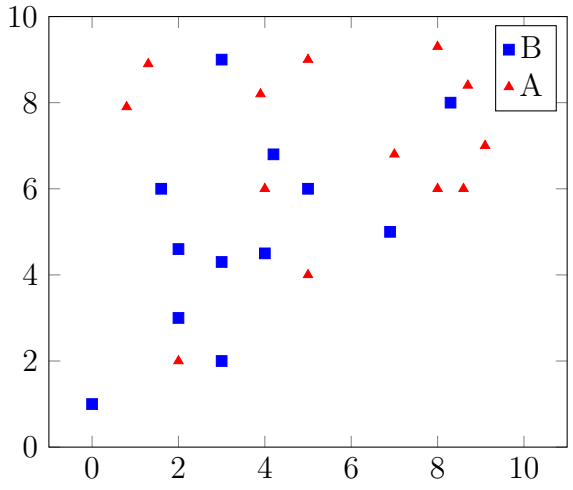

Figure 3.1: Not separable

3.1 Soft margin

In order to allow for miss classifications, we first have to change the definition of the margin. In the previous chapter the margin represented the euclidean distance from the *closest* data point to the hyperplane H. Now we define the margin to be the euclidean distance from the pair of *closest and correctly classified* data points, one from set A and one from set B both on opposite sides of the hyperplane H, such that H sits exactly in the middle of the pair of data points. These *closest and correctly classified* data points will be called the support vectors of the model.

We will introduce a new objective for our SVM, we want to minimize the number of miss classifications while still keeping the margin as large as possible. For now we assume that every miss classification is equally bad and has the same penalty. We can use constraints (3.2) to determine if datapoint x_i has been miss classified.

Assume that the hyperplane H has been determined.
Then if x_i is correctly classified the following holds:

$$w^\intercal x_i + b \geq 1 \quad \forall x_i \in A \tag{3.3}$$

$$w^\intercal x_i + b \leq -1 \quad \forall x_i \in B \tag{3.4}$$

Hence x_i is miss classified if $y_i(w^\intercal x_i + b) < 1$
Let ψ_i be called the *0-1* loss function which is equal to one if x_i has been miss classified, otherwise it is equal to zero.

$$\psi_i = \begin{cases} 0 & \text{if } y_i(w^\intercal x_i + b) \geq 1 \\ 1 & \text{if } y_i(w^\intercal x_i + b) < 1 \end{cases} \tag{3.5}$$

We want to minimize the number of miss classifications, so we minimize $\sum \psi_i$. We also change the constraints (3.2) to

$$y_i(w^\intercal x_i + b) \geq 1 - M\psi_i \quad \forall i \tag{3.6}$$

If $\psi_i = 0$, then we know $y_i(w^\intercal x_i + b) \geq 1$. So (3.6) also holds $\forall M \in \mathbb{R}$.
If $\psi_i = 1$, then $y_i(w^\intercal x_i + b) < 1$, and $y_i(w^\intercal x_i + b)$ can get arbitrarily negative depending on the severity of the miss classification. So by bounding the miss classification distance by a number $M \in \mathbb{R}$. We force the model to select w, b, ψ such that the hyperplane is relatively close by the data and we cannot get that $y_i(w^\intercal x_i + b)$ gets arbitrarily negative.

Now the model becomes:

$$\min_{w,b} \quad \frac{1}{2}||w||^2 \tag{3.7}$$

$$\min_{\psi} \quad \sum_i \psi_i \tag{3.8}$$

$$\text{s.t.} \quad y_i(w^\intercal x_i + b) \geq 1 - M\psi_i \qquad \forall i \tag{3.9}$$

$$\psi_i \in \{0, 1\} \qquad \forall i \tag{3.10}$$

This problem formulation is a binary integer quadratic programming problem.

3.2 Loss functions

Our assumption to use the *0-1* loss function (3.6) is a good way to minimize the number of miss classifications. But introducing this function brings some problems with it. Firstly, the optimization problem is now a binary integer quadratic programming problem, which are in generally harder to solve than a quadratic programming problem, unless we can find some properties of the SVM formulation that we can utilize. And more importantly, we have no way to distinguish between the different severity's of miss classifications, since we assumed that all miss classifications are equal.

3.2.1 Hinge Loss

We now no longer consider that all miss classifications are equal. So we need to change the loss function.
Let ψ_i be called the *hinge* loss function which represents the euclidean distance from x_i to the support vector of the corresponding set of x_i. ψ_i now penalizes miss classifications that are far away from the support vector more heavily than miss classifications that are close to the support vector.

$$\psi_i = \begin{cases} 0 & \text{if } y_i(w^\intercal x_i + b) \geq 1 \\ 1 - y_i(w^\intercal x_i + b) & \text{if } y_i(w^\intercal x_i + b) < 1 \end{cases} \tag{3.11}$$

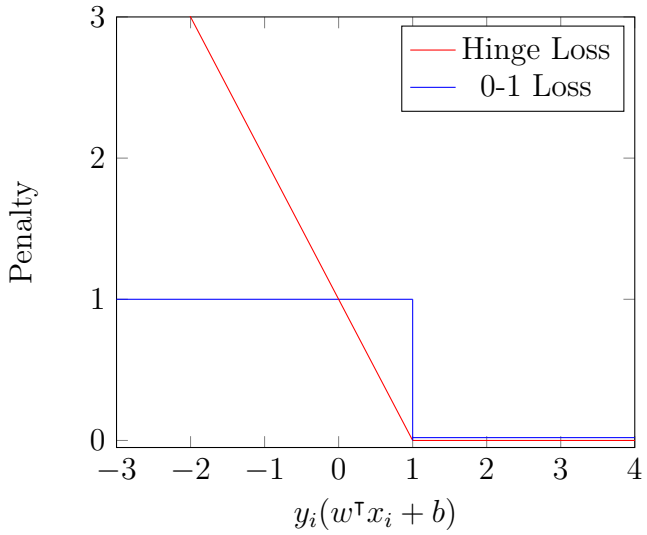

Figure 3.2: Different Loss functions

We can rewrite (3.11) into $\psi_i = \max\{0, 1 - y_i(w^\intercal x_i + b)\}$, which means that our loss function is no longer binary and it is also convex. So if we use the hinge loss function, we no longer need to solve an integer optimization problem. Now $\sum_i \psi_i$ no longer represents the total number of miss classified points, but it represents the total distance from all miss classified points to the support vectors. We now also do not need $M \in \mathbb{R}$ in order to control

the constraints (3.6). Since ψ_i will become arbitrarily large if $y_i(w^\intercal x_i + b)$ becomes very negative. So the model becomes:

$$\min_{w,b} \quad \frac{1}{2}||w||^2 \tag{3.12}$$

$$\min_{\psi} \quad \sum_i \psi_i \tag{3.13}$$

$$\text{s.t.} \quad y_i(w^\intercal x_i + b) \geq 1 - \psi_i \qquad \forall i \tag{3.14}$$

3.2.2 Exponential Loss

We also consider the exponential loss function, since the hinge loss function is not differentiable at $y_i(w^\intercal x_i + b) = 1$. So it cannot be used with the meta heuristic technique Gradient Descent.
Let ψ_i be called the *exponential* loss function which indirectly grows proportional to the euclidean distance from x_i to its support vector. So this loss function really penalizes miss classifications that are far away from the corresponding support vector, but it is really sensitive to outliers.

$$\psi_i = e^{-y_i(w^\intercal x_i + b)} \tag{3.15}$$

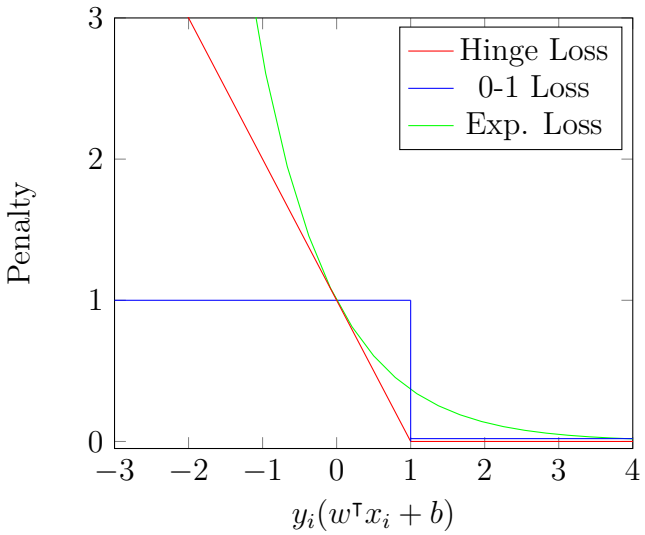

Figure 3.3: Different Loss functions

Unfortunately, $\sum_i \psi_i$ no longer has a geometric meaning. But we still want to minimize the sum. So the model becomes:

$$\min_{w,b} \quad \frac{1}{2}||w||^2 \tag{3.16}$$

$$\min_{\psi} \quad \sum_i \psi_i \tag{3.17}$$

$$\text{s.t.} \quad y_i(w^\intercal x_i + b) \geq 1 - \psi_i \qquad \forall i \tag{3.18}$$

3.3 Multi-objective Optimization

We now need to combine the objective functions into one, which might be problematic if the two objectives are conflicting with each other. For example, improving one of the objectives might deteriorate the other objective. Combining the objective functions can be done in many different ways, but first a few definitions. We also assume that all objective functions need to be minimized, as is the case with our current SVM model.

Let $\mathcal{F}$ be the set of feasible solutions. Then a generalized model becomes:

$$\min_x \qquad F(x) = \begin{pmatrix} f_1(x) \\ \vdots \\ f_p(x) \end{pmatrix} \tag{3.19}$$

$$\text{s.t.} \qquad x \in \mathcal{F} \tag{3.20}$$

Where $F : \mathbb{R}^n \to \mathbb{R}^p$

Definition 3.3.1 (Dominance)**.**
Consider $x_1, x_2 \in \mathcal{F}$. x_1 is dominating x_2 if
x_1 is always better or equal than x_2 in any objective

$$f_i(x_1) \leq f_i(x_2) \quad \forall i = 1, .., p$$

x_1 is strictly better than x_2 in at least one objective

$$\exists i \text{ s.t. } f_i(x_1) < f_i(x_2)$$

Notation: x_1 dominates x_2: $F(x_1) \prec F(x_2)$.

Definition 3.3.2 (Pareto Optimality)**.** $x^* \in \mathcal{F}$ is pareto optimal if it is not dominated by any other $x \in \mathcal{F}$

$$\nexists x \in \mathcal{F} \text{ s.t. } F(x) \prec F(x^*)$$

Definition 3.3.3 (Weakly Pareto Optimality)**.** $x^* \in \mathcal{F}$ is weakly pareto optimal if

$$\nexists x \in \mathcal{F} \text{ s.t. } \forall i = 1, .., p : \quad f_i(x) < f_i(x^*)$$

Then the pareto optimal set

$$P^* = \{x \in \mathcal{F} \quad | \quad \nexists x \in \mathcal{F} : F(x) \prec F(x^*)\} \tag{3.21}$$

And the pareto front

$$PF^* = \{f(x^*) \quad | \quad x \in P^*\} \tag{3.22}$$

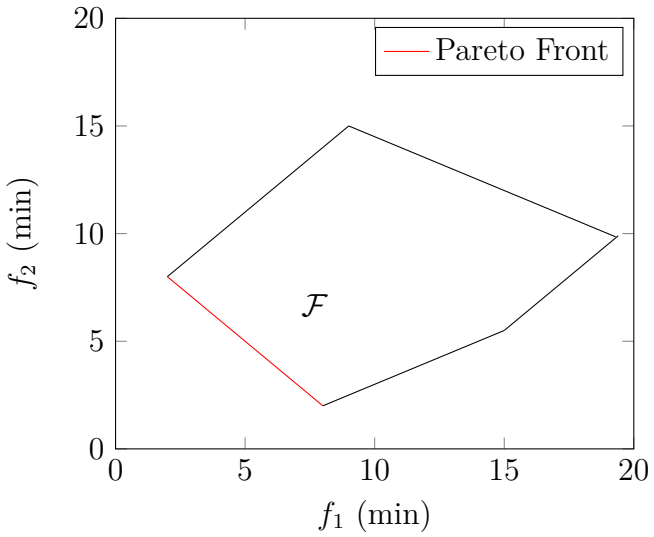

Figure 3.4: Example of Pareto front

We now have two spaces that we will be looking at, the feasible solution space which represents the decision variables x and we have the objective space which represents $F(x)$. For the sake of simplicity we say that a solution x^* to the optimization problem includes the optimal decision variables x and the optimal objective value $F(x)$. We want to find a solution x^* to the optimization problem that are located on the pareto front, we want that $x^* \in PF^*$ so we can call x^* the optimal solution to the multi-objective optimization problem. x^* might not be an unique solution as shown in figure (3.4), all solutions that are on the pareto front are optimal with respect to the multi-objective optimization problem.

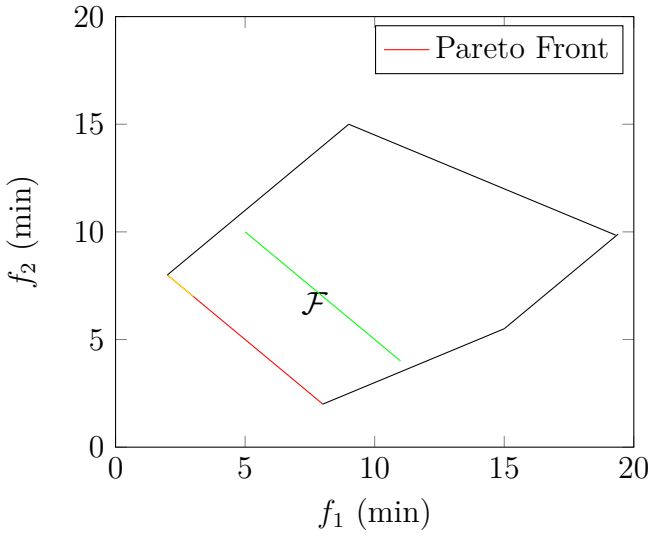

Figure 3.5: Multi-objective algorithms

Ideal multi-objective algorithm

Generally speaking, there are two goals that any multi-objective optimization algorithm should take into consideration.

1. Convergence to the pareto optimal front
2. Maintaining a diverse distribution on the pareto front

Looking at figure (3.5). Assume that an algorithm produced a set of solutions that is represented by the green line in the figure. We see that this line is nicely distributed in the feasible region, but no point on the line is close to the pareto front, so no point is close to being pareto optimal. Now assume that another algorithm produced a set of solutions that is represented by the yellow line, we see that this line converges to the pareto front, so the points are all pareto optimal, but they are all concentrated on a small part of the pareto front. So both the green and yellow line are not good to represent the pareto front.

A fundamental difference between single and multi objective optimization is the number of *optimal* solutions. After all, the user that uses an MOO algorithm only needs one solution. But we have shown that in the case of MOO their might exist more than one pareto optimal solution to the problem. Now the question arises, which solution to choose from this pareto optimal set. The answer to this question is non-trivial, "It involves higher-level information which is often non-technical, qualitative and experience-driven"(Deb, 2001). For this reason we will only be looking at finding the pareto optimal set by considering that all objective functions are important. And after such set of trade offs has been found, the user can use the *higher-level information* to make a choice from this set.

3.3.1 Parametric methods

With parametric methods we find pareto optimal solution by computing many single objective optimizations, and each time we only find one pareto optimal solution. This is achieved by converting the multiple objectives into a single objective with some parameters, and by changing the parameters we can obtain different pareto optimal solutions. Below we will discuss the weighted sum parametric scalarizing approach.

Weighted Sum approach

Consider the SVM problem, where we only have two objective functions to minimize. The most intuitive way to combine the objective functions into one, is to use a linear combination of the objective functions using weights. We give each objective function f_i a weight $w_i \in \mathbb{R}$. Then the objective function F in (3.19) becomes:

$$F(x) = w_1 f_1 + w_2 f_2$$

Where f_1 corresponds to (3.7) and f_2 corresponds to (3.8).

This represents a straight line with the slope $\frac{w_1}{w_2}$. Minimizing F results in dragging this line over the feasible region $\mathcal{F}$ until this line is at the boundary of $\mathcal{F}$ while still on the pareto front as shown in figure (3.6).

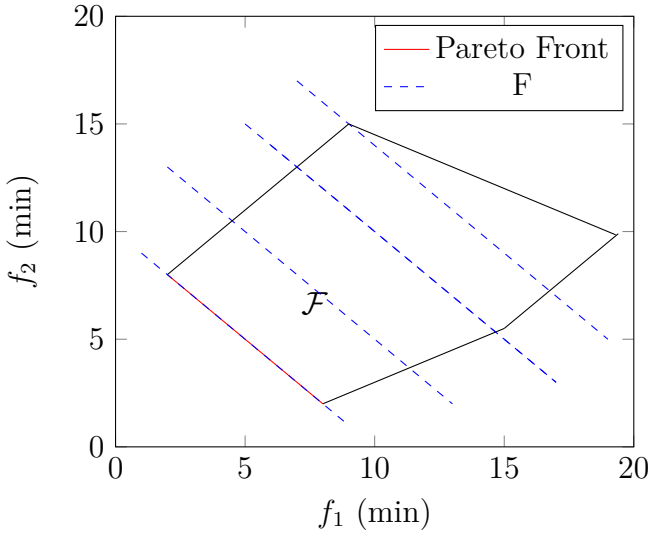

Figure 3.6: Example of Pareto front

Given a set of weights, the weighted sum approach gives us a pareto optimal solution to the SVM problem. But in general there are some difficulties with this approach. We still need to pick the weight ourselves, and we might not find all pareto optimal solutions if the feasible region is not convex. As can be seen in figure (3.7), all solutions on the pareto front between points A and B cannot be reached with the weighted sum approach no matter what weights we pick.

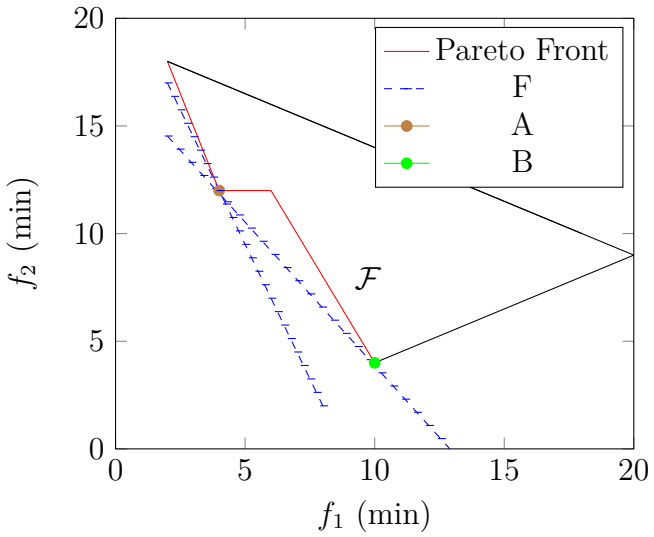

Figure 3.7: Non-convex $\mathcal{F}$

Fortunately, the SVM problem has a convex feasible region, since the constraints that corresponds to the SVM problem are linear in the unknowns.

3.3.2 Standard SVM

Now we will consider the *Standard* formulation by using the weighted sum approach with weights $w_1 = 1$ and $w_2 = \lambda$. This is a very popular way of describing the SVM problem, where we can choose a number of different loss functions (Yuan & Jian-gang, 2005).
Then our optimization problem becomes:

$$\min_{w,b,\psi} \quad \frac{1}{2}||w||^2 + \lambda \sum_i \psi_i \tag{3.23}$$

$$\text{s.t.} \quad y_i(w^\intercal x_i + b) \geq 1 - M\psi_i \qquad \forall i \tag{3.24}$$

$$\psi_i \in \{0, 1\} \qquad \forall i \tag{3.25}$$

Where ψ_i is the *0-1* loss function defined in (3.5).

With this formulation the λ is now a control parameter, which controls the margin of the model. Increasing the λ results in a lower margin, and decreasing λ results in a higher margin.

Case 1: $\lambda \to \infty$

The objective function is increasing in the value of λ, so increasing λ increases the objective value. But we want to minimize the objective function. So if we increase λ we can only get a lower objective value by lowering the loss function ψ as defined in (3.5). This can be done by changing the hyperplane such that more data points are correctly classified. If we want to decrease the number of miss classified points, we have to select a data point

that is in between the current support vector and the hyperplane to be the new support vector and thus be correctly classified. And since the hyperplane is always in the middle of the two support vectors from each set, this would mean that the distance between the hyperplane and the support vector must decrease, since the new support vector is between the old support vector and the old hyperplane. So the margin of the model would decrease, since the margin is defined (3.1) as the distance to the *closest* correctly classified data point and the hyperplane.

Case 2: $\lambda \to 0$

The objective function is increasing in the value of λ, so decreasing λ decreases the objective value. So if we decrease λ, but we want to maintain the same objective value, we have to increase the margin. And looking again at our model, we want to maximize the margin. Increasing the margin can be done by changing the hyperplane such that the distance between the support vectors and the hyperplane is larger. A consequence of increasing the margin is that we might increase the number of misclassified points, which increases the objective value. So decreasing λ results in a higher margin of the model.

3.3.3 Why differ from the standard

In this thesis we are looking at the entire pareto front instead of the conventional formulation with weights 1 and λ as discussed in chapter 3.3.2. We do this because we want to have a method for obtaining all the pareto optimal solutions instead of only one. This can be helpfull in situations in which the user that wants to solve a SVM problem is not sure about the parameter choice. By giving the entire pareto front as the solution, the user does not has to choose the specific parameters for the SVM beforehand. The user can just pick a solution from the pareto front that the user likes, for example a solution that has a very low number of miss classified points, and the algorithm provides the corresponding parameters and the decision rule.

By not setting the parameters for the SVM, we enlarge the solution space relative to the solution space of the standard SVM problem. A larger solution space or feasible region might result in better objective values, so this approach might be able to obtain better solutions than using the standard weighted sum approach.

Chapter 4

Evolutionary Multi-objective Methods

4.1 Genetic Algorithms

Genetic algorithms (GA) are heuristics that can solve optimization problems by using natural selection principles. The idea behind GA is that the initial solutions are evolved into better solutions by using predefined operations. Each solution is called a member of the population and has some *characteristics* that can be manipulated by certain operations, for example, mutations or cross-over selection between multiple solutions. Solutions in the population are represented by an encoding scheme, for example in binary or real numbers. These encoding schemes are used such that the cross-over or mutation operations are more convenient to perform.

Usually the initial population of solutions is generated at random, and is improved on an iterative basis. Each iteration is called a generation, and in each generation we perform some operations on the current population to create new solutions and then select a subset of the new population that will go to the next iteration. This subset is determined by a fitness function that evaluates how close a solution is to the optimal solution, for example the objective function can be used as a fitness function. The algorithm is terminated if a predetermined number of iterations has been reached or if some other stopping criteria has been reached, for example, a desired fitness value.

In each generation we need to do two things:

- Create offspring
- Determine the fitness of each solution

We can create new solutions, called offspring, by changing some of the *characteristics* of the current solutions at random, called mutations, or by combining the *characteristics* of two solutions into one, called cross-over selection. After we have created the offspring, we need to calculate the fitness of each member of the population and then determine which members of the current generation can go to the next iteration. It is commonly used to keep the population size the same throughout every iteration. (Mitchell, 1998)

4.2 NSGA-II

Evolutionary multi-objective (EMO) methods are specialised Genetic Algorithms that can find multiple pareto optimal solution in a single simulation, by looking at multiple non-dominated and isolated solutions (Deb, 2001). This is an improvement over the parametric methods that we discussed in the previous chapter. Because parametric methods can only find a single pareto optimal solution in a simulation.

We look at the Non-Dominated Sorting Genetic Algorithm (NSGA-II) proposed by (Deb, Pratap, Agarwal, & Meyarivan, 2002) as our EMO method. We also use the dual formulation of the SVM with the *0-1*-loss function in combination with the radial basis function kernel as described in appendix A, since this formulation simplifies the procedure for NSGA-II (Deb et al., 2002). And now each solution to the SVM can be described by the parameters κ, λ and the Lagrange multipliers α_i.

The main principle in the NSGA-II algorithm is as follows (Deb et al., 2002):

1. Create offspring and a combined population
2. Rank and sort offspring based on non-dominating fronts
3. Take best members to create new population, accounting for a diverse spread of the population in the objective space

The two goals that we want to have for an ideal MO algorithm are convergence to the pareto front (1) and a diverse distribution along this pareto front (2). We want to converge to the pareto front, so we look at non-dominating solutions, since that is a necessary condition for pareto optimality, hence principle two of NSGA-II. And we also want to have a diverse distribution on the pareto front, hence principle three of NSGA-II.

We let P_t denote the population in generation t and Q_t denote the offspring in generation t. Then the combined population $R_t = P_t \cup Q_t$ in generation t.

We set $|P_t|$ equal to 100 in accordance with the standard setting for GA proposed by De Jong (De Jong, 1975). And since we want that the population size stays the same in each iteration, we need to pick 100 members of R_t that can go to the next iteration. Small population sizes can cause the NSGA-II algorithm to converge in early iterations, this decreases the search space of the algorithm, while large population sizes can increase the search space and might result in better solutions. But this may increase the run time of the algorithm.

Encoding scheme

We represent the solutions p in the following way

$$p = \begin{bmatrix} \kappa & \lambda & \alpha_1 & \alpha_2 & \cdots & \alpha_n \end{bmatrix} \tag{4.1}$$

This row-vector contains all the characteristics of the solution. κ is the parameter for the RBF kernel function as defined in (A.9), which is a non-negative real number. λ is the parameter for the objective function, which is a non-negative real number. And α_i is the Lagrange multiplier of data point i, which has a value between 0 and λ.

Initial population

We generate 100 random sets of parameter values $\kappa \in (0, 100], \lambda \in [0, 100]$, then for each set we must generate the Lagrange multipliers. This is done by selecting a number between 0 and the generated λ for each Lagrange multiplier α_i. The objective function of the SVM problem (3.24) and the performance measure F_1-score (2.2) can then be calculated by using the Lagrange multipliers as explained in appendix (A). The 100 generated solutions are the initial population P_0.

Cross-over selection

We create new solutions P_{new}, called offspring, by selecting two random members $P_{t,1}$ and $P_{t,2}$ from the population P_t and we randomly set $a \in \{0, 1\}^{n+2}$. Then we perform the following crossover

$$P_{\text{new}} = a * P_{t,1} + (\iota - a) * P_{t,2} \tag{4.2}$$

Where ι is a column vector consisting of only ones. This operation means that the new solution P_{new} has on each column either the value of $P_{t,1}$ or the value of $P_{t,2}$.

We create 100 new solutions such that Q_t has a size of 100. We choose 100 because then we have doubled our population which is in accordance with the standard setting for GA propsed by De Jong (De Jong, 1975). We now have a new population $R_t = P_t \cup Q_t$ of size 200, which we need to rank and select half to advance to the next iteration.

Mutation

The standard for genetic algorithms is to enlarge the solution space as much as possible in the early iterations to escape local optima, but at the end of the simulation, we should have convergence to the pareto front. Hence at the end of our simulation we do not want to introduce too much randomness with the mutations. So the mutation probability should be a decreasing function in the iteration counter t. Thus we set the mutation probability as:

$$p_{\text{mutation}} = 0.3 * \frac{1}{t} \tag{4.3}$$

Then for each solution in R_t we draw a random number from the uniform distribution UNIF(0,1). If that drawn number is greater than the p_{mutation}, we mutate nothing. If the drawn number is smaller or equal, we change the value of the characteristic to a random number accounting for the conditions that were set on the characteristics in subsection (4.2). Performing the mutation operation in this way implies that the probability of a mutation is equal to p_{mutation}.

Determine fitness with non-dominating sorting

After the offspring has been made, we will sort R_t based on non-dominating fronts, which is in line with the second principle of our algorithm (2). This means that we will group every solution in R_t based on the non-dominating property (3.3.1), as can be seen in figure (4.1). For example, all of the green dots are dominated by the red dots, and all of the red dots are dominated by the blue dots, but all dots with the same colour do not dominate each other. Set F_1 forms a pareto front, while F_2 forms a pareto front if you remove F_1, and F_3 form a pareto front if you remove F_1 and F_2. The procedure to sort the population R_t is given in appendix B.1.

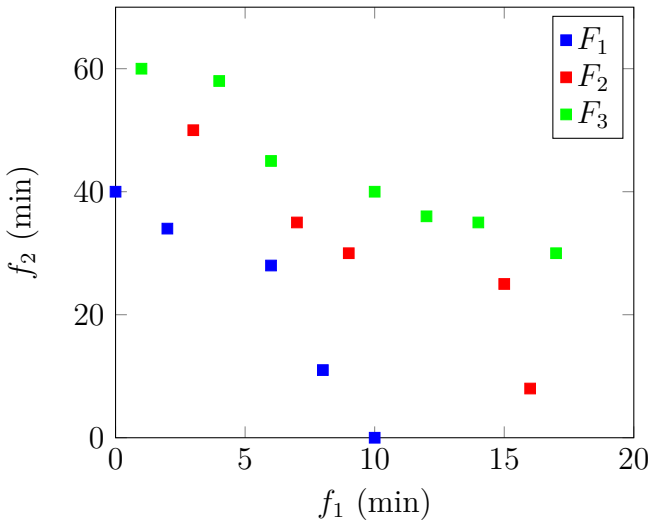

Figure 4.1: Non-dominating sorting

Determine fitness with crowding Distance Sorting

After we have sorted the population R_t based on non-dominating sorting, we rank the solutions based on the highest crowding distance (Deb et al., 2002). We do this because we also want that our solutions are distributed along the entire pareto front, and not clustered together on a small portion of the pareto front.

The crowding distance of solution i represents a square that is formed around the two *nearest* other solutions, $i-1$ and $i+1$ as can be seen in figure (4.2). And the crowding distance can be seen as the area of this square. Large values of the crowding distance imply that the solution is not close to any other solution. So picking the solutions with the highest crowding distance gives us solutions that are nicely distributed in the objective space, which is the third principle of NSGA-II (3). The procedure to sort the population R_t is given in appendix B.2

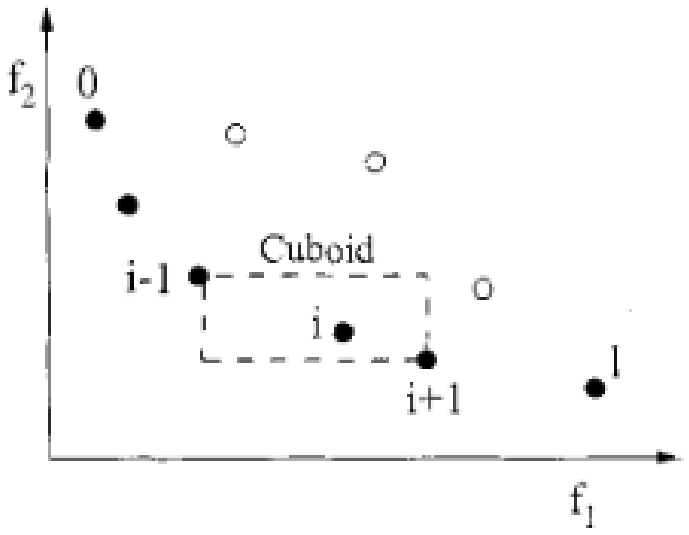

Figure 4.2: Distance Sort (Deb et al., 2002)

After we have calculated the fitness and ranked the population R_t, we can decide which solutions in the population can proceed to the next iteration. As can been seen in figure (4.3), first we rank the population R_t based on the non-dominated sorting principle, we then go through the groups, starting in F_1, and add it to the next iteration unless there is no more space. Which happens if adding a group results in more than 100 members in P_{t+1}. If this happens then we look at the crowding distance value for each solution in the group that doesn't fit, for example F_3. We then select the solutions in F_3 with the highest crowding distance value until the population P_{t+1} has 100 members.

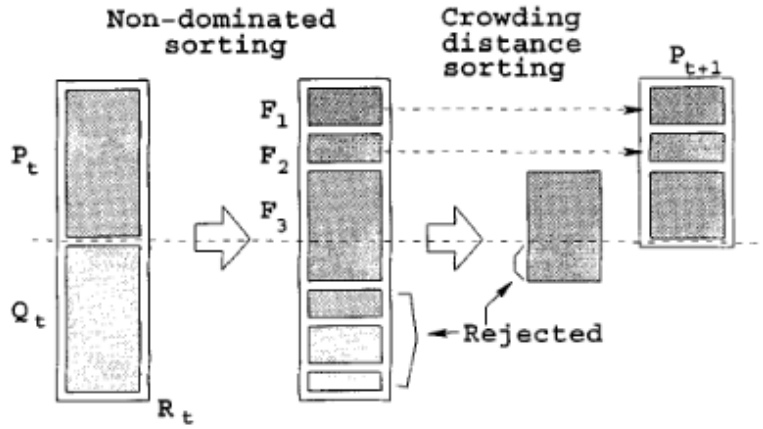

Figure 4.3: NSGA-II Procedure (Deb et al., 2002)

Stopping Criteria

We stop the simulation after 100 iterations in accordance with the standard setting for GA propsed by De Jong (De Jong, 1975).

4.3 Numerical Experiments

The experiment was performed in the following way

- load data or generate data depending on the dataset used.
- divide the data into the training and the validation set.
- perform the NSGA-II algorithm as described in section (4.2) on the training set.
- compute the performance measure F_1-score with the training set.

4.3.1 Data

The data used is a combination of user generated data and existing data that can be found online. The user generated data is called Normal. The normal dataset is comprised of five dimensional vectors for which each entry is generated from a normal distribution. Where data points from set A are i.i.d from $N(20, 20)$ and data points from set B are i.i.d. from $N(50, 20)$. This data does not represent anything in the real world. In total there are 10000 data points in this data set.

The other data, called UC Irvine, was retrieved from the UC Irvine Machine Learning Repository, this dataset represents default payments of credits cards from customers in Taiwan in 2005 (Yeh, 2016). Where the goal is to predict if a given customer is going to default on its payments or not. Each customer has the following information

- Y $\in \{0, 1\}$: defaulted on payments (0=no, 1=yes)
- X1: customers credit in NT dollars
- X2: gender
- X3: education
- X4: material status
- X5: age in years
- X6-X11: history of past payments in NT dollar
- X12-X17: amount of bill statements
- X17-X25: amount of previous payments in NT dollar

In total there is data from 30000 customers in the set. With no missing data entries.

4.3.2 Computer specifications

CPU	AMD Ryzen 5 1600 6 core processor
Clock speed	3.20 GHz
RAM	16 GB @1067 MHz
Python version	3.10.5
Pygurobi version	10.0.3
Numpy version	1.23.1

Table 4.1: Computer Specifications

Note: Python does only use a single core.

4.3.3 Results

Data Set	size	validation size	iterations	F_1-score	runtime (seconds)	figure
Normal	3000	300	100	0.632	579	C.5
Normal	3000	300	100	0.684	612	C.6
UC Irvine	3000	300	100	0.448	1368	C.8
UC Irvine	3000	300	100	0.492	1473	C.9

Table 4.2: results of NSGA-II algorithm

In table (4.2) some of the results can be seen. The *Data Set* column says which data as described in section (4.3.1) is used. The *size* column is the amount of data points that is used for training. The *validation size* is the amount of data points that is used for the performance measure. The training and validation sets are disjoint sets, so there is no overlap. The *iterations* column is the amount of generation there were run, which is the stopping criteria of the algorithm. The F_1*-score* is the average of all the F_1-scores of every solution in the last generation. the *runtime* is the amount of seconds that the algorithm took to finish, this does not include loading the dataset and splitting the data into a training and validation set, only the time it took for the NSGA-II algorithm and the performance measure to be computed. The *figure* column points to the corresponding figure in the appendix, again the F_1-score in the figures is the average of the F_1-scores of all the solutions in the generation t. All other results can be seen in appendix (C).

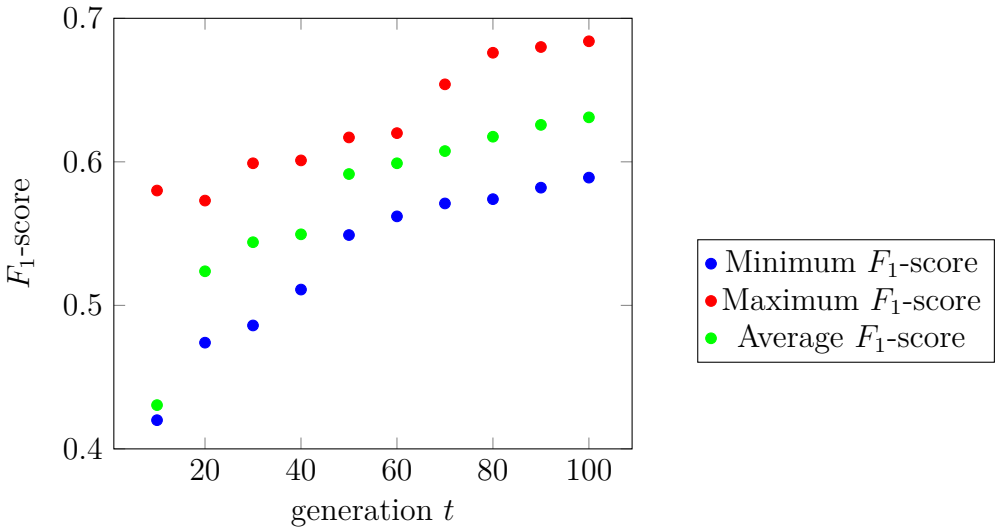

Figure 4.4: Statistics for simulations on the Normal dataset

We can see from figure (4.4) that there can be large differences in the obtained F_1-score between different simulations. This is displayed by the area in between the red and blue dots. But we can still draw some conclusions on our results. We see that the NSGA-II performs initially well on the data, between the initial generation and generation $t = 60$ there are large jumps in the F_1-score, hence the algorithm can rapidly improve the solutions. But after generation $t = 60$ we see a stagnation in this rapid improvement and see that the gap between the maximum and the minimum is closer than before generation $t = 60$. The overall F_1-scores still improves but the average F_1-score seems to converge. This can come from a number of reasons. Firstly, we cannot improve due to the data set and we have already achieved convergence to the pareto front. Secondly, the search space is exhausted due to a lack of introducing new solutions to the population. Changing the mutation probability or the stopping criteria might increase the search space such that more solutions can be found. We also see that in the initial generations ($t = 0$ to $t = 10$), the average and the minimum F_1-score are close to each other, while the maximum F_1-score is far above. This happened because in one simulation the initial solution was extremely good compared to the other simulations, and this is a consequence of the randomness in the construction of generation P_0.

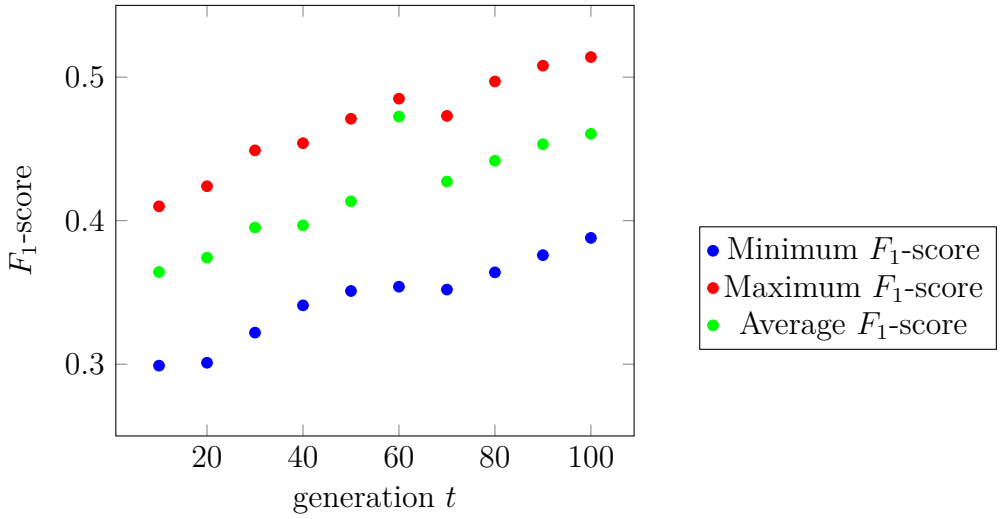

Figure 4.5: Statistics for simulations on the UC Irvine dataset

Just as with the Normal dataset, we can see from figure (4.5) that there is a big difference between the maximum and the minimum F_1-score. Indicated as the area between the red and blue dots. In contrast with the Normal dataset, we now see no big jumps in the F_1-score, except around generation $t = 60$. This sudden increase in the average F_1-score is the result of an coding error that caused the calculation of the performance to fail at this generation, hence we should ignore the results of generation $t = 60$ in this figure. We only see that the average, the maximum and the minimum scores increase gradually with no big improvements as we saw with the Normal dataset. This can be caused by a number of things. Firstly, the search space does not increase enough to find better solutions, so we need to increase the mutation probability to allow for a larger solution space. One other reason is that this dataset has 25 dimensions and there really is no good way to separate these data points with the current setup of the algorithm, so a different cross-over selection or stopping criteria has to be chosen. We also see that the average stays nicely in between the middle of the minimum and the maximum F_1-score, which is different compared to the Normal dataset. There we saw that the maximum and minimum form a hourglass around the average, the difference between the maximum and minimum gradually decreases until generation $t = 60$, after which the difference increases again. But we did not see that pattern with the UC Irvine dataset.

Changing the mutation probability

Our current mutation probability is $p_{\text{mutation}} = 0.3 * \frac{1}{t}$. But we want to change this in order to see if we can increase the search space and obtain better results. Therefore we introduce the following mutation probability:

$$p_{\text{mutation}} = r * \frac{1}{t} \quad , \quad r \in [0.3, 0.8] \tag{4.4}$$

We chose this because in order to increase the search space, we need to allow for more mutations so the mutation probability has to increase. The mutation probability still decreases as the number of iteration increases. We ran multiple simulations to obtain the minimum, maximum and average performance scores.

The same datasets are used as before, the normal and the UC Irvine with training size of 3000 and validation size of 300. The stopping criteria is still 100 iterations. Because we now have three variables to display in the figures, the iteration counter t, the mutation probability parameter r and the F_1-score, we decided to only display the figures for a select number of iterations. Namely iteration $t = 20$, $t = 60$ and $t = 100$, in order to keep using 2D-figures. Below are the figures for iteration $t = 100$, the other results can be seen in appendix (C).

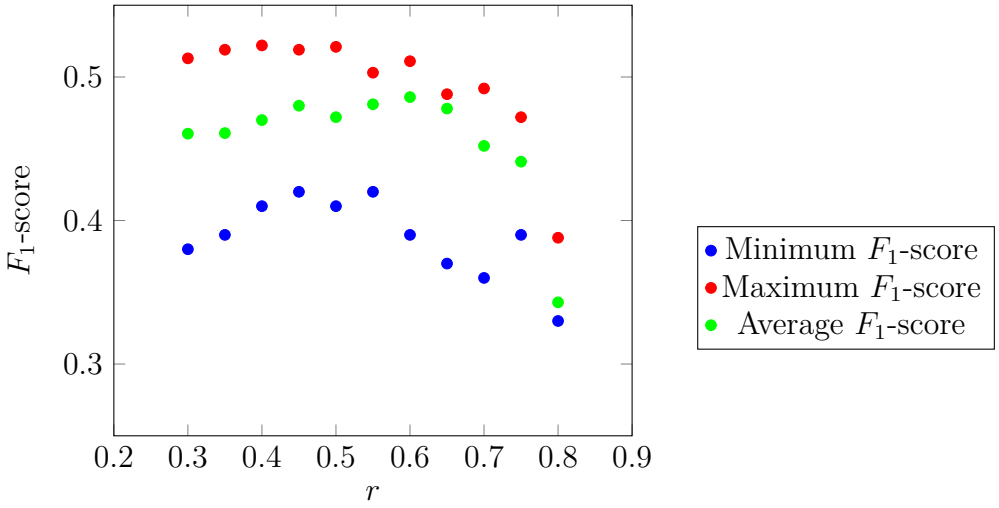

Figure 4.6: UC Irvine dataset at iteration $t = 100$

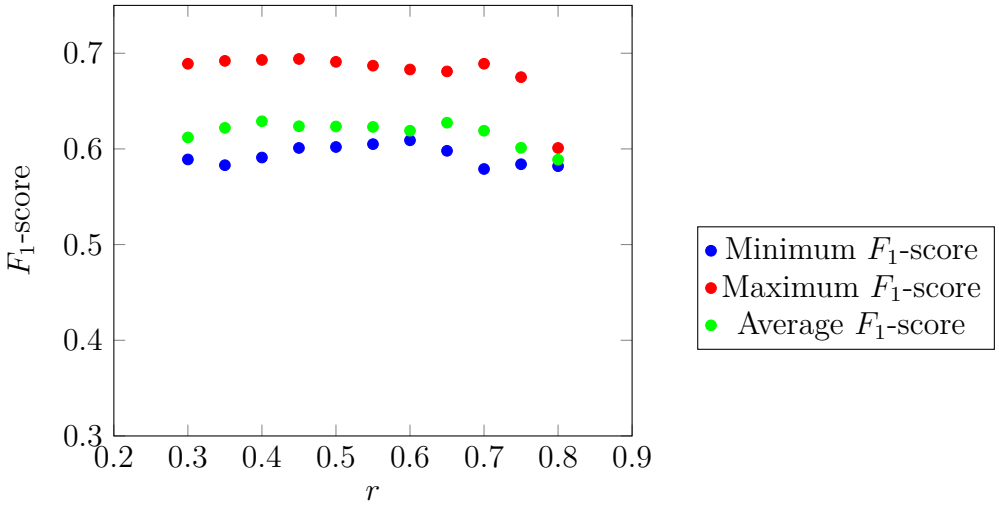

Figure 4.7: Normal dataset at iteration $t = 100$

From the figures in appendix (C) we can conclude that the F_1-score for the normal dataset and the UC Irvine dataset almost always drops for $r \geq 0.7$. Which is something we did expect to happen. Since the mutation probability is increasing in the value of r, so for large values of r an increase in the mutation probability can drastically change almost all solutions in the corresponding iteration, since then almost all solutions get randomly changed, even the *best* solutions of the iteration.

For the normal dataset, we see that changing the mutation probability does not really effect the F_1-score in the final iteration. As can be seen in figure (4.7), the minimum, the maximum and the average F_1-scores stay the same over all values of r. Hence changing the mutation probability to increase the solution space did not work as intended to achieve a higher F_1-score in the final generation. So in order to increase the F_1-score, we must change other parts of the algorithm, for example the stopping criteria or cross-over selection. It can also be the case that we already achieved the convergence to the pareto optimal solutions, and hence we cannot improve the F_1-score.

For the UC Irvine dataset, we see that the value for r can drastically change the F_1-score in the final generation. We see in figure (4.6) that around $r = 0.6$ we obtain the highest average F_1-score, so the change in mutation probability can increase the performance of the algorithm. But we also see that the maximum and minimum F_1-score are lower than in the original $r = 0.3$ case. This means that the solutions at $r = 0.6$ are more consistent in performance. From figure (C.14) we see a strange pattern, for values of $r \geq 0.55$ we see that the difference between the maximum and minimum increases rapidly, most likely due to the erratic behaviour of the randomness that comes from a high mutation probability. Which is not something that we want, we prefer consistent solutions both in the minimum and maximum performance scores.

Changing the stopping criteria

The current stopping criteria is that the NSGA-II algorithm stops after 100 iterations. We will now change this in order to test if we can increase the performance of the algorithm. Therefore we will allow for the algorithm to run longer, because it can happen that currently our algorithm just needs more time to escape local optima and actually starts to increase the F_1-score. We will test the following maximum iterations on both datasets: 100,200 and 300 iterations.

The same datasets are used as before, the normal and the UC Irvine with training size of 3000 and validation size of 300. Below are the figures for the 300 iterations, but they can also be used to see the performance at 100 and 200 iterations. Also the original mutation probability of $p_{\text{mutation}} = 0.3 * \frac{1}{t}$ is used.

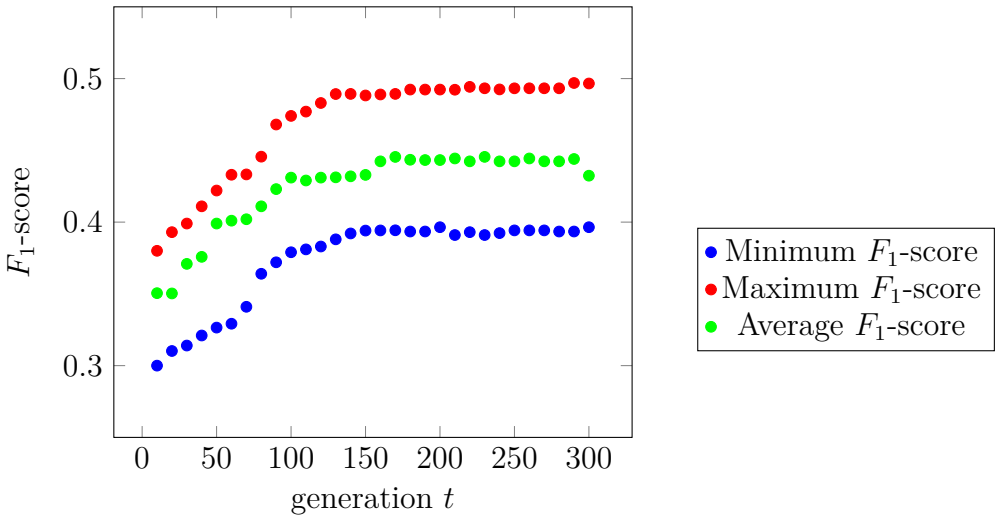

Figure 4.8: Statistics for simulations on the UC Irvine dataset

As we can see from figure (4.8), after the 150-th iteration we no longer see any improvements in the minimum, maximum or average F_1-score on the UC Irvine dataset. This also means that the algorithm cannot obtain more consistent solutions, as this would imply that the maximum and minimum values would become closer to each other. As before, we see large jumps in the performance in the early iterations that gradually stop the longer the algorithm continues. So increasing the number of iterations does not work for the UC Irvine dataset. This can happen because we already achieved convergence to the optimal pareto front but more experimentation is needed to conclude this. For example, changing the cross-over selection or changing the stopping criteria to a target F_1-score instead of a constant value for the iterations.

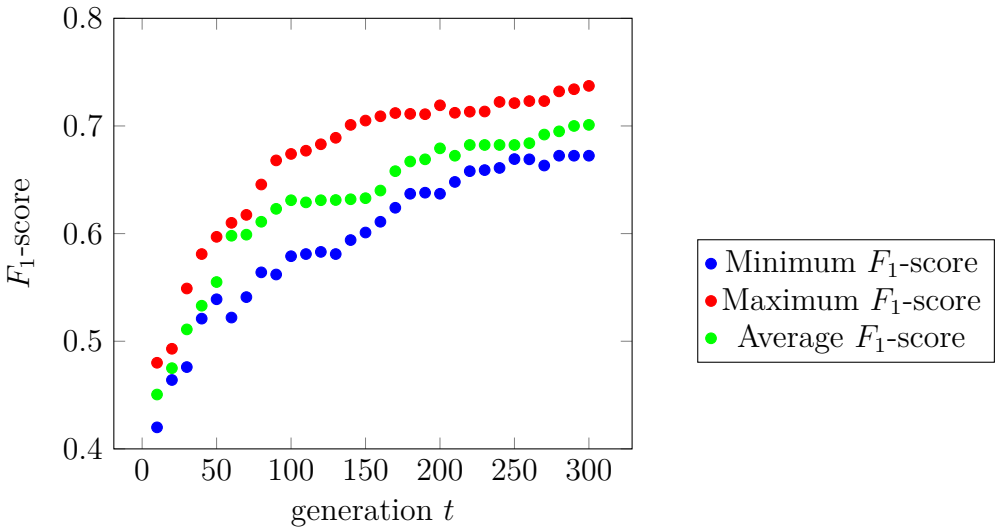

Figure 4.9: Statistics for simulations on the Normal dataset

As we can see from figure (4.9), that increasing the number of iterations does improve the F_1-score of the algorithm. Just as before we see that the algorithm makes large jumps in the F_1-score in the early iterations. Between the 80th and 120th iteration we see little to no improvement, but after the 120th iteration see can again see big jumps in the performance. After the 200th iteration we see that the F_1-score stagnates but the maximum and minimum F_1-score become closer to each other, hence we are able to obtain more consistent solutions as a result of increasing the number of iterations.

Chapter 5

Conclusion and Future Work

The main goal of this thesis was to find solutions to the support vector machine optimization problem where we have multiple conflicting objective functions. In the first chapter we explained what a support vector machine is and what its applications are. In the second chapter we showed how to derive the SVM model, and in the third chapter we introduced a more complex form of the SVM problem where we have multiple objective functions that need to be minimized. We saw in chapter three that an algorithm to find optimal solutions to a multi-objective problem should take two principles into consideration, convergence to the pareto optimal front and maintaining a diverse distribution on the pareto front. In chapter 4 we discussed the NSGA-II algorithm, which can be used to solve optimization problems that have multiple objective functions while also taking the two principles into consideration. We talked about the choices we made for the algorithm and what could happen if we would make different choices.

We then ran our experiment on the two datasets, the user generated *Normal* dataset and the *UC Irvine* dataset on credit card default payments. In both cases we saw that our algorithm could improve the performance measure in the early iterations, but the improvement would gradually stop at the end of the simulations. This could have a few causes, firstly the simulation stopped too early, secondly the search space of the algorithm is not sufficiently large enough to find better solutions and thirdly the algorithm has already reached convergence to the pareto optimal solutions. The best case scenario would be if we had already convergence to the pareto optimal solutions, but in order to test this we need to perform sensitivity analysis on the parameters and choices that we made for the algorithm in chapter three.

First we changed the mutation probability of the algorithm, because this probability has in general a large impact on the amount of new solutions that are introduced to our search space. For the *Normal* dataset we saw that changing the mutation probability did not effect the performance measure of the algorithm. But for the *UC Irvine* dataset we saw that changing the mutation probability did lead to better results, unless this probability was close too high.

We also changed the stopping criteria of the algorithm, but we only increased the amount of iterations that the algorithm would run. Here we saw that increasing the number of iterations did not increase the performance measure of the *UC Irvine* dataset, but for the *Normal* dataset it had a large impact on the performance of the algorithm. So by increasing the number of iteration we could expand the solution space such that we obtained better results for the *Normal* dataset.

The overall conclusion of the performance of the implemented NSGA-II algorithm is that it is possible to obtain good results from an implementation of GA. However the results strongly depend on the dataset that is used and requires more analysis on the algorithm choices, which we did only for the mutation probability and a simple stopping criteria. With the obtained results we cannot say for certain that the achieved solutions are really on the true Pareto Front or if the algorithm can still improve the solutions by looking at different solution spaces.

Due to the very long runtime of the implemented NSGA-II algorithm, we were not able to test more of our algorithm choices. For example, the stopping criteria of a constant number of iterations is not good, ideally we would like to have a target performance level as the stopping criteria, but this could be something for future works. Another thing that could be implemented in future works is *Multi threading*, this could significantly lower the runtime by performing calculations in parallel on different CPU threads. For example splitting the population in two subsets and performing the cross-over and mutation operations on both subsets simultaneously.

References

Basu, A., Walters, C., & Shepherd, M. (2003, Jan). Support vector machines for text categorization. In *36th annual hawaii international conference on system sciences, 2003. proceedings of the* (p. 7 pp.-). doi: 10.1109/HICSS.2003.1174243

Deb, K. (2001, 01). Multiobjective optimization using evolutionary algorithms. wiley, new york..

Deb, K., Pratap, A., Agarwal, S., & Meyarivan, T. (2002, April). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, *6*(2), 182-197. doi: 10.1109/4235.996017

De Jong, K. A. (1975). *An analysis of the behavior of a class of genetic adaptive systems.* (Unpublished doctoral dissertation). USA. (AAI7609381)

Doiphode, B. S., & Sapkal, S. D. (2018, June). Classifying facial expression using support vector machine based on bidirectional local binary pattern histogram feature descriptor. In *2018 second international conference on intelligent computing and control systems (iciccs)* (p. 1890-1895). doi: 10.1109/ICCONS.2018.8662980

Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. The MIT Press. Retrieved from `https://doi.org/10.7551/mitpress/3927.001.0001` doi: 10.7551/mitpress/3927.001.0001

Oliveira, L., & Sabourin, R. (2004, Oct). Support vector machines for handwritten numerical string recognition. In *Ninth international workshop on frontiers in handwriting recognition* (p. 39-44). doi: 10.1109/IWFHR.2004.99

Thurnhofer-Hemsi, K., López-Rubio, E., Molina-Cabello, M. A., & Najarian, K. (2020). *Radial basis function kernel optimization for support vector machine classifiers.*

Yeh, I.-C. (2016). *default of credit card clients.* UCI Machine Learning Repository. (DOI: https://doi.org/10.24432/C55S3H)

Yuan, G., & Jian-gang, L. (2005, Oct). A scheme of function approximation based on svm and nsga ii. In *2005 international conference on neural networks and brain* (Vol. 1, p. 512-516). doi: 10.1109/ICNNB.2005.1614665

Appendix A

Dual Formulation of SVM

We will use the dual formulation of the SVM problem in the algorithm that is discussed in chapter 4, since it is more convenient than the primal formulation (Deb, 2001).
The lagrange function for the SVM is

$$L(w, b, \psi, \alpha, \mu) = \frac{1}{2}||w||^2 - \sum_i \alpha_i \lambda [y_i(w^\intercal x_i + b) - 1 + M\psi_i] - \sum_j \mu_j \psi_j \quad \text{(A.1)}$$

Where ψ is the *0-1* loss function defined in (3.5)

The first order conditions are:

$$\frac{dL}{dw} = w - \sum_i \alpha_i y_i x_i = 0 \implies w = \sum_i \alpha_i y_i x_i \quad \text{(A.2)}$$

$$\frac{dL}{db} = -\sum_i \alpha_i y_i = 0 \implies \sum_i \alpha_i y_i = 0 \quad \text{(A.3)}$$

$$\frac{dL}{d\psi} = -\sum_i \alpha_i - \sum_j \mu_j = 0 \implies \sum_i \alpha_i = -\sum_i \mu_i \quad \text{(A.4)}$$

Substituting these results back into the lagrange function results in

$$L(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j x_i^\intercal x_j \quad \text{(A.5)}$$

Where α_i is the Lagrange multiplier for data point i.

Now the model becomes

$$\min_{\alpha} \quad \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j x_i^\intercal x_j \tag{A.6}$$

$$\text{s.t.} \quad \sum_i \alpha_i y_i = 0 \qquad \forall i \tag{A.7}$$

$$0 \leq \alpha_i \leq \lambda \qquad \forall i \tag{A.8}$$

Now the following holds for every data point i:

- $\alpha_i = 0 \implies$ the data point i is on the correct side of the hyperplane
- $0 < \alpha_i < \lambda \implies$ the data point i is a support vector
- $\alpha_i = \lambda \implies$ the data point i is on the wrong side of the hyperplane

Hence to compute the number of miss classified points, we need to look at the data points which have a lagrange multiplier equal to λ. To compute the margin, we can use equation (A.2).

A.1 Kernels

Looking at the dual formulation (A.6) we can see that the objective function now depends on the dot product between each pair of data points x_i and x_j. We can use *kernels* to transform this product to a space that might be more convenient, for example, resulting in a better decision boundary that separates the data and obtain a lower objective value (Thurnhofer-Hemsi, López-Rubio, Molina-Cabello, & Najarian, 2020). This transformation is denoted by $K(x_i, x_j)$.

We will be using the RBF kernel:

$$K(x_i, x_j) = e^{-\kappa ||x_i - x_j||^2} \tag{A.9}$$

With κ as the parameter that controls the width of the kernel function. The RBF kernel is a very general form for the kernelization of the SVM problem and is very popular in SVM applications (Thurnhofer-Hemsi et al., 2020). Now the model becomes

$$\min_{\alpha} \quad \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j K(x_i, x_j) \tag{A.10}$$

$$\text{s.t.} \quad \sum_i \alpha_i y_i = 0 \qquad \forall i \tag{A.11}$$

$$0 \leq \alpha_i \leq \lambda \qquad \forall i \tag{A.12}$$

Appendix B

Algorithms

B.1 Non-dominated sorting

Algorithm 1 set counters & lists

```
1: Input
2:         all solutions of the current iteration
3: Output
4:     dc_i  number of solutions that dominate solution i
5:     dl_i  the solutions that are dominated by solution i
6: for solution k = 1, 2, ... 200 do
7:     dc_k=0
8:     dl_k= empty list
9:     for solution l = 1, 2, ... 200 do
10:        if k != l then
11:            if f_1(k) ≤ f_1(l) and f_2(k) ≤ f_2(l) then
12:                dc_k = dc_k + 1
13:                dl_k add solution l
14:            end if
15:        end if
16:    end for
17: end for
```

Explanation of the code

For every solution i we set the dc_i and dl_i to zero. Then we loop over all other solutions k and see if the objectives for solution i are better (lower) than for solution k. If this is true then we say that solution i dominates solution j and we increase dc_k with one. We also add solution k to the list dl_i of solution i.

Algorithm 2 non-dominated sorting

```
1: Input
2:     dc_i  number of solutions that dominate solution i
3:     dl_i  the solutions that are dominated by solution i
4: Output
5:           all groups that are non-empty
6: put all solutions with dc_i= 0 in group 1
7: for group k = 2, 3, ... do
8:     for solution i in group (k − 1) do
9:         for solution l in dl_i do
10:            dc_l = dc_l - 1
11:        end for
12:    end for
13:    for solution l = 1, 2, ..., 200 do
14:        if dc_l=0 and not group 1, ..., (k − 1) then
15:            add solution l to group k
16:        end if
17:    end for
18:    terminate if all solutions are inside a group
19: end for
```

Explanation of the code

The idea is to put solution in groups. The first group we create by putting all solutions that are not dominated by any other solution, hence put all solutions i that have $dc_i = 0$ in group F_1. We then perform a loop that stops if all solutions are inside a group. In every loop we make a new group F_k. First we need to loop over all solutions that are inside group F_{k-1}, and for these solutions i we loop over their list dl_i, for every solution l that is inside this list dl_i, we decrease that counter dc_l with one. Because we now no longer look at solutions in previous groups, so the solutions in previous groups do no longer dominate the other solutions. After we have looped over all solutions in group F_{k-1} we can start to fill group F_k. We do this by putting all solutions l that have $dc_l = 0$ in group F_k. Then we repeat this cycle until all solutions are inside a group.

B.2 Crowding Distance Sorting

Algorithm 3 Crowding distance sorting

1: **Input**
2: all non-empty groups from algorithm 2
3: **Output**
4: dis_i the crowding distance of solution i
5: Set dis_i=0 for all i
6: **for** group $k = 1, 2, \ldots$ **do**
7: **for** objective function f=1,2 **do**
8: sort group k from lowest $f(i)$ to highest $f(i)$
9: Let f_m denote the minimum value of the group k w.r.t objective f
10: which corresponds to solution im
11: Let f_b denote the maximum value of the group k w.r.t objective f
12: which corresponds to solution ib
13: $dis_{im} = Inf$
14: $distance_{ib} = Inf$
15: **for** all other solutions in group k **do**
16: $dis_i = dis_i + \frac{f(i+1)-f(i-1)}{f_m - f_b}$
17: **end for**
18: **end for**
19: **end for**

Explanation of the code

We set the crowding distance $dis_i = 0$ for all solutions Then we loop over all the groups, starting with F_1. For these groups we also loop over every objective function. For each objective function we sort all the solutions from lowest to highest. The value of these solutions represents the position in the objective space and we want to try to calculate the distance a solution has relative to its neighbors. We saw that the solutions which have the highest and lowest objective value do not have two valid neighbors, so their crowding distance is set to Infinity. For all other solutions we calculate the crowding distance by applying the formula that is in line 16.

Appendix C

Extra Tables & Figures

Data Set	size	validation size	iterations	F_1-score	runtime (seconds)	figure
Normal	400	40	100	0.759	103	C.1
Normal	1000	100	100	0.763	264	C.2
Normal	3000	300	100	0.589	591	C.3
Normal	3000	300	100	0.619	622	C.4
Normal	3000	300	100	0.632	579	C.5
Normal	3000	300	100	0.684	612	C.6
UC Irvine	3000	300	100	0.388	1437	C.7
UC Irvine	3000	300	100	0.448	1368	C.8
UC Irvine	3000	300	100	0.492	1473	C.9
UC Irvine	3000	300	100	0.514	1410	C.10
UC Irvine	3000	300	100	multiple runs		C.11
Normal	3000	300	100	multiple runs		C.12
UC Irvine	3000	300	$p_{\text{mutation}} = r * \frac{1}{t}$ at $t = 20$			C.13
UC Irvine	3000	300	$p_{\text{mutation}} = r * \frac{1}{t}$ at $t = 60$			C.14
UC Irvine	3000	300	$p_{\text{mutation}} = r * \frac{1}{t}$ at $t = 100$			C.15
Normal	3000	300	$p_{\text{mutation}} = r * \frac{1}{t}$ at $t = 20$			C.16
Normal	3000	300	$p_{\text{mutation}} = r * \frac{1}{t}$ at $t = 60$			C.17
Normal	3000	300	$p_{\text{mutation}} = r * \frac{1}{t}$ at $t = 100$			C.18
UC Irvine	3000	300	300	multiple runs		C.19
Normal	3000	300	300	multiple runs		C.20

Table C.1: Results of NSGA-II

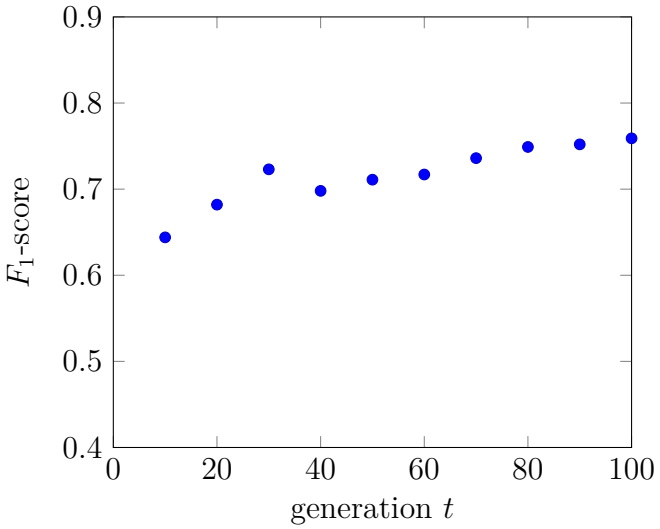

Figure C.1: Normal with 400 samples

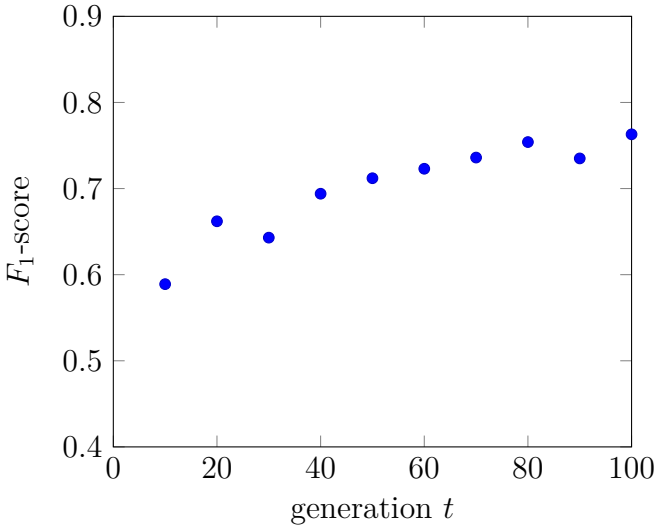

Figure C.2: Normal with 1000 samples

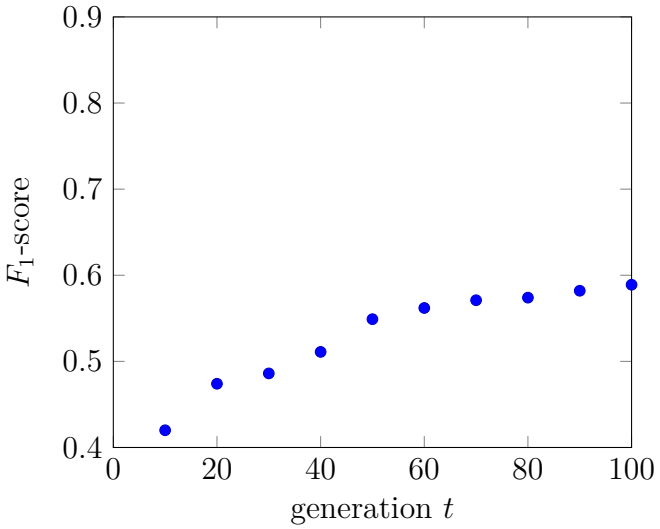

Figure C.3: Normal with 3000 samples

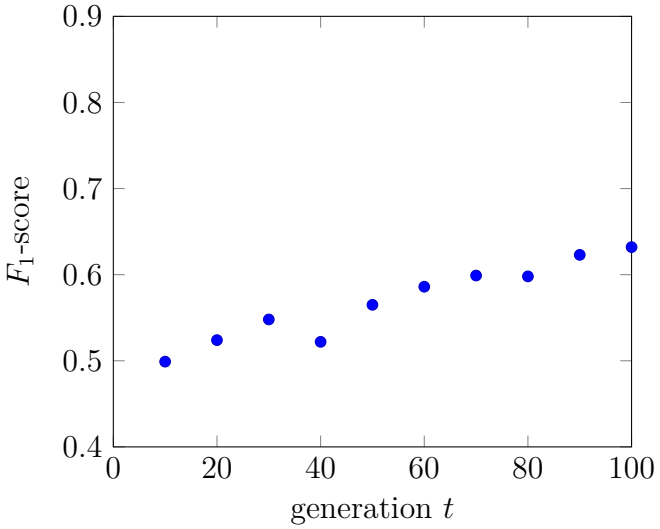

Figure C.4: Normal with 3000 samples

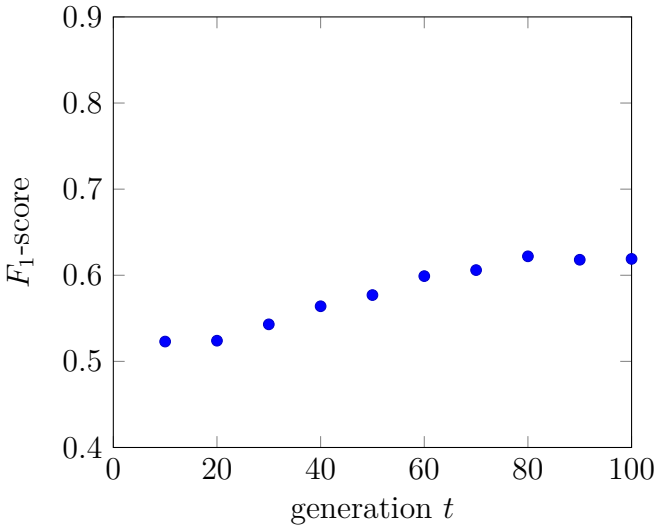

Figure C.5: Normal with 3000 samples

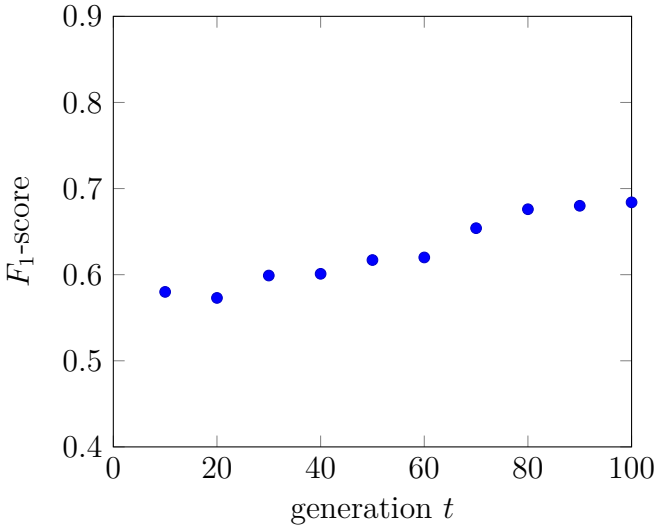

Figure C.6: Normal with 3000 samples

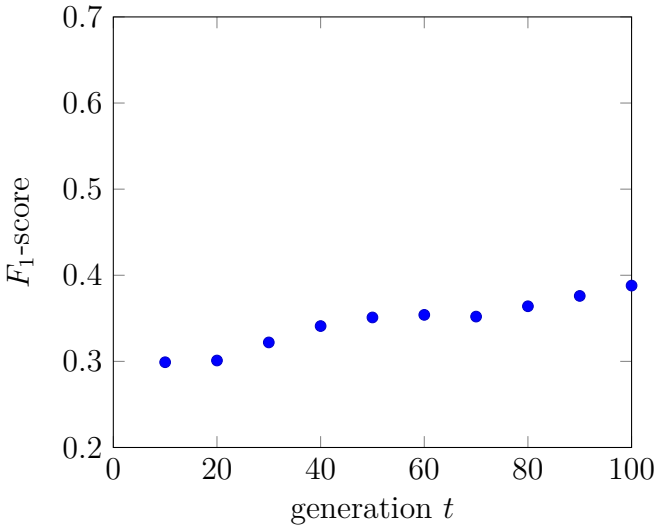

Figure C.7: UC Irvine with 3000 samples

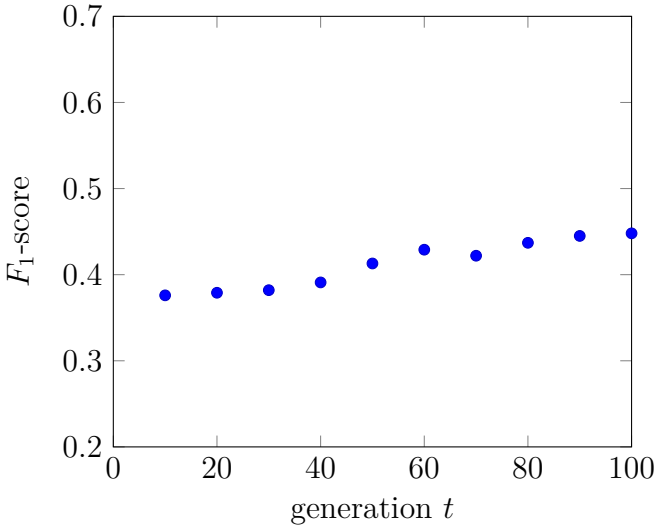

Figure C.8: UC Irvine with 3000 samples

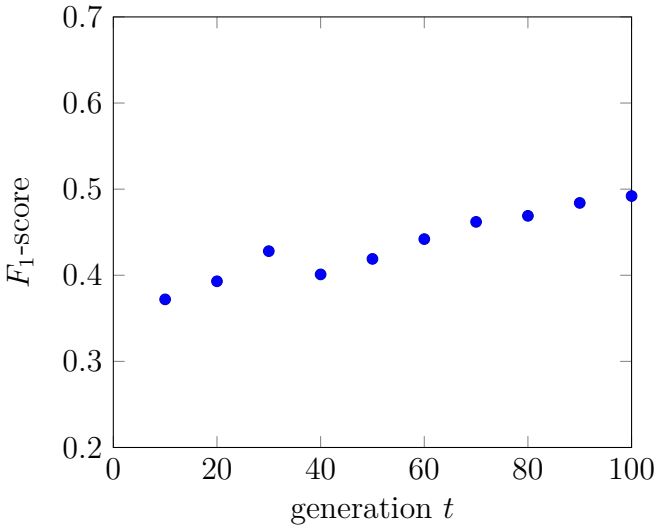

Figure C.9: UC Irvine with 3000 samples

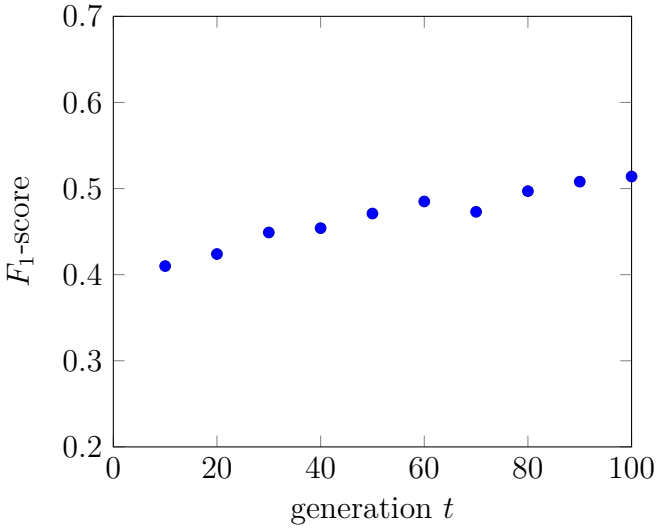

Figure C.10: UC Irvine with 3000 samples

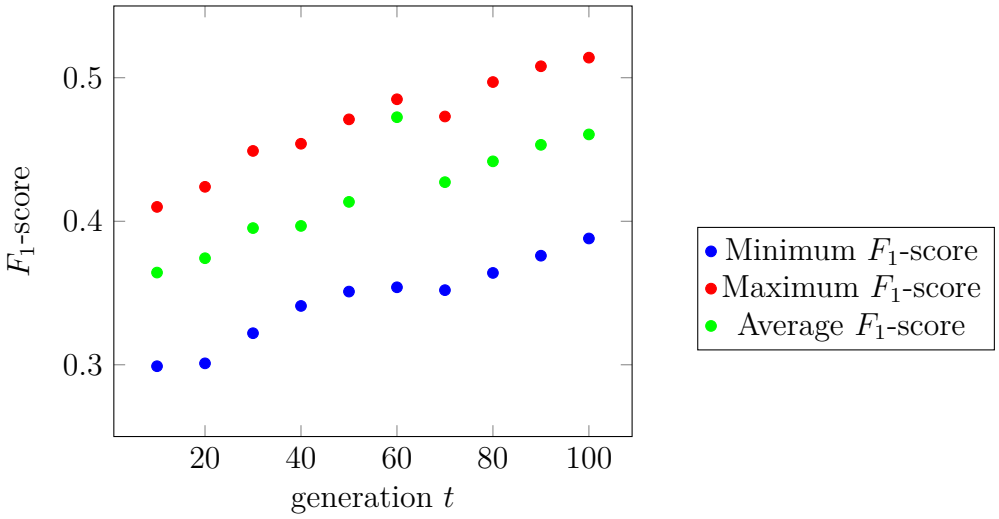

Figure C.11: Statistics for simulations on the UC Irvine dataset

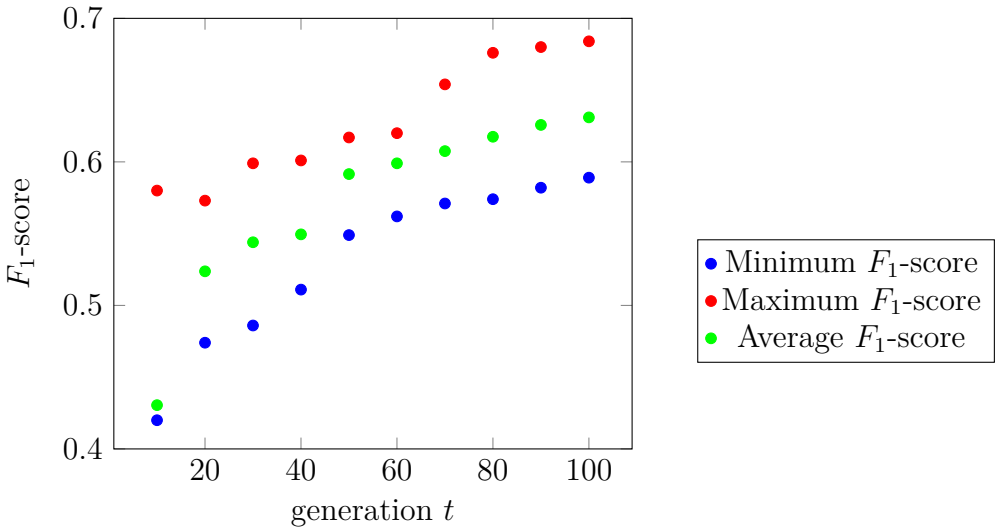

Figure C.12: Statistics for simulations on the Normal dataset

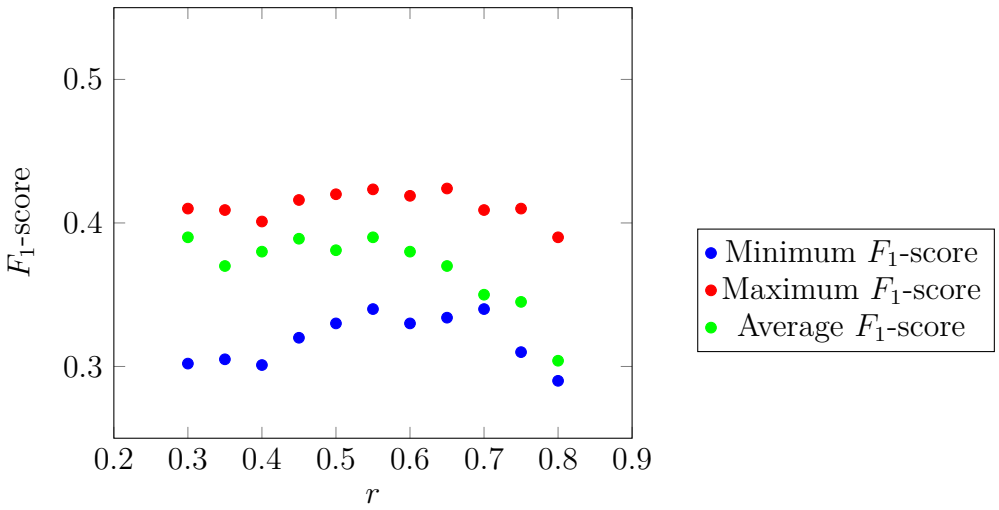

Figure C.13: UC Irvine 3000 dataset at iteration $t = 20$

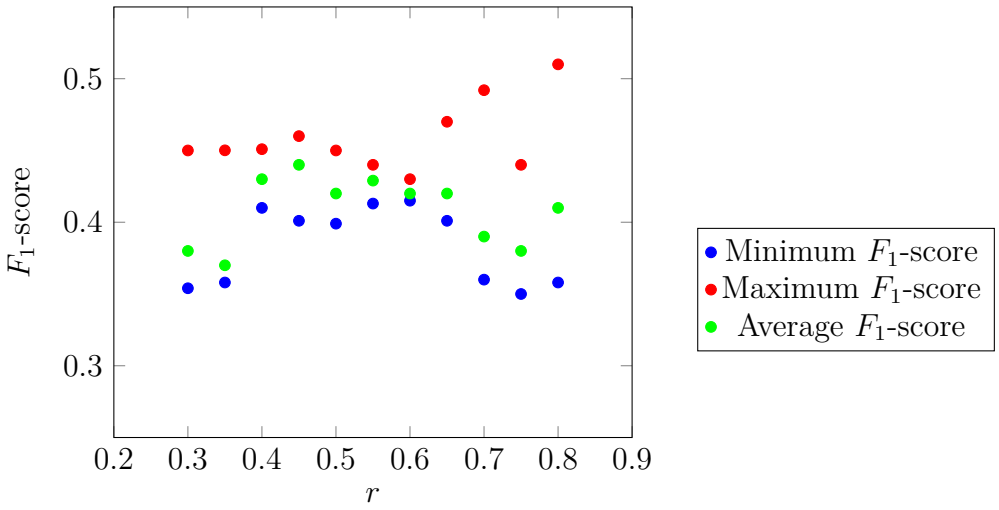

Figure C.14: UC Irvine 3000 dataset at iteration $t = 60$

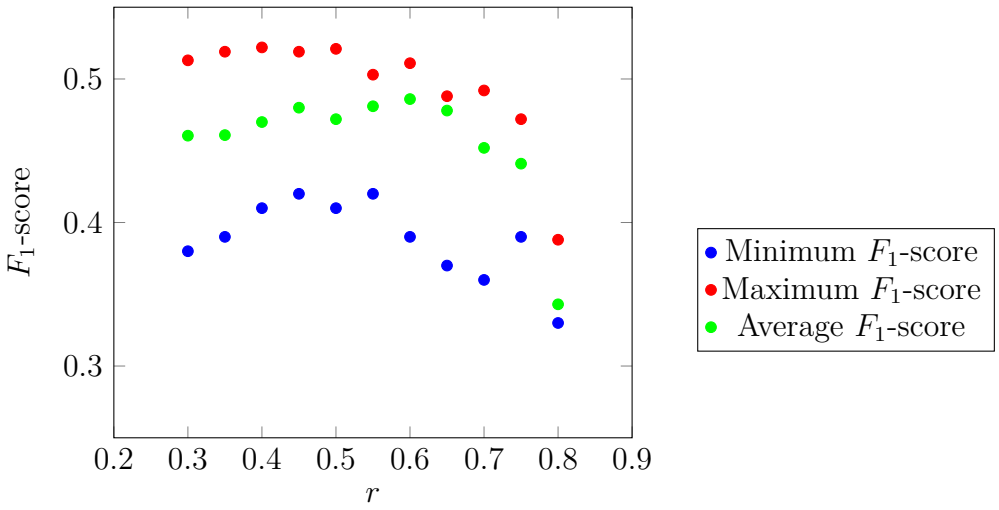

Figure C.15: UC Irvine 3000 dataset at iteration $t = 100$

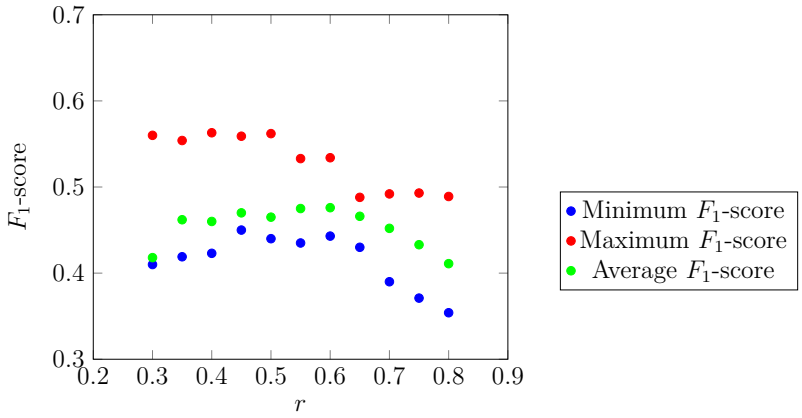

Figure C.16: Normal 3000 dataset at iteration $t = 20$

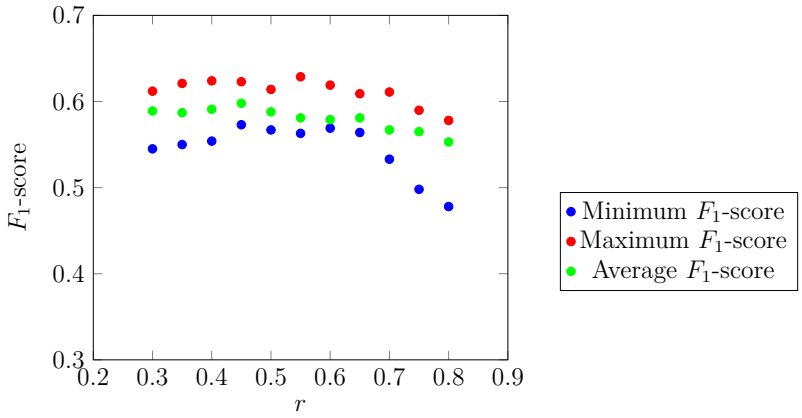

Figure C.17: Normal 3000 dataset at iteration $t = 60$

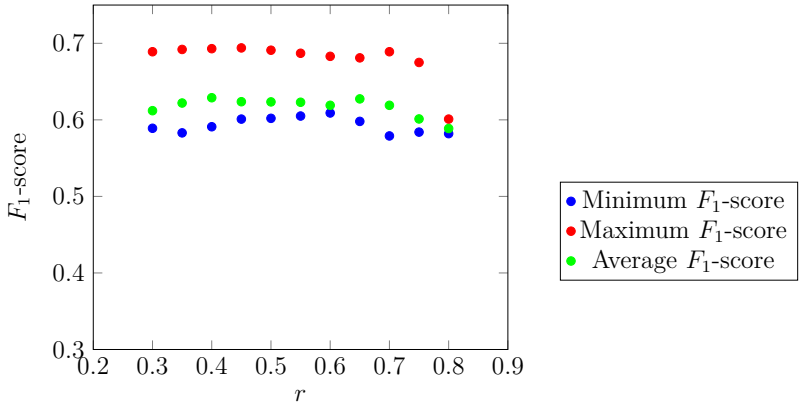

Figure C.18: Normal 3000 dataset at iteration $t = 100$

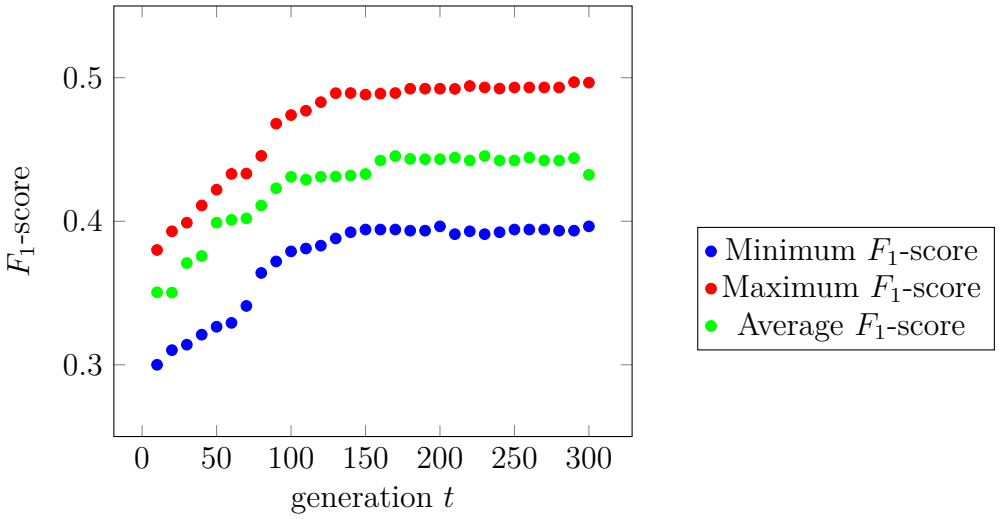

Figure C.19: Statistics for simulations on the UC Irvine dataset

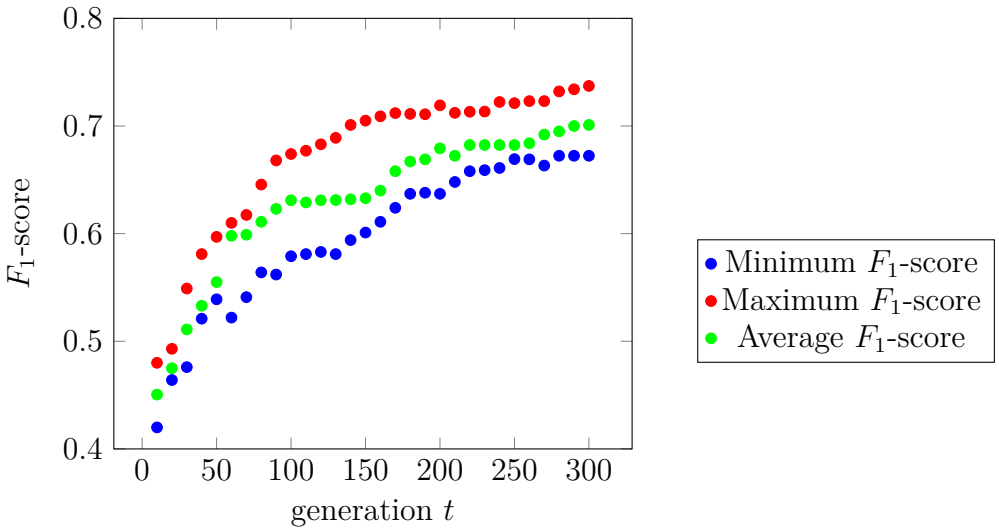

Figure C.20: Statistics for simulations on the Normal dataset