Machine learning the enhancement chains arising by degenerating Calabi-Yau manifolds

Author:
Ruben Meijs BSc

Supervisors:
dr. Thomas Grimm
dr. Gil Cavalcanti

Utrecht University

Faculty of Science
Utrecht University

June, 2020

Abstract

We propose a machine learning algorithm, based on decision trees and a bagging procedure that helps us to study the allowed combinations of degeneration types in Calabi-Yau threefolds. Many Calabi-Yau manifolds have been constructed and classifying them is an important question. Recently a classification method for Calabi Yau threefolds has been proposed, based on studying the degeneration limits of the manifold. The degenerations correspond to taking the limit to the boundary of the Kähler moduli space. At a boundary, so-called limiting mixed Hodge structures can be defined, which are uniquely represented by 2 numbers. Determining these for all possible limits of the Kähler moduli space gives a graph-like structure, that can be used to classify the manifold. These graphs consist of enhancement chains, which are sequences of subsequent limits together with the corresponding type and value of the limiting Hodge structure. The rules for the allowed single enhancements chains are explicitly known. The machine learning algorithm developed in this work is able to learn most of these known rules from data on Calabi-Yau threefolds. The algorithm is used to learn the rules on the, previously unknown double enhancements. We find strong evidence that the proposed algorithm is able to learn rules on the allowed single and double enhancements in Calabi-Yau threefolds given enough input.

Contents

Chapter 1

Introduction

The study of string theory has had huge impact on both physics and mathematics [1]. One important mathematical object for which string theory has provided many new insights are Calabi–Yau manifolds. These manifolds can be used to so called compactify the 10 dimensional theory that arises in string theory. String theory is a (and perhaps the) theory of quantum gravity and has many fascinating, useful features. However, for string theory to be a proper, consistent theory it turns out that we need 10 dimensions to describe our world. This is of course not how we perceive our world, where we only have 3 spacial and 1 time dimension. The idea behind compactification is to curl up these six extra dimensions, to make them so small that we cannot observe them. How we compactify turns out to greatly influence the effective theory at low energy. The reason to use Calabi–Yau manifolds is that they keep precisely the right amount of supersymmetry [2]. This leads to a major research question: which low energy, effective theories can emerge from string theory? Finding an answer to this question is done in the so called *Swampland program* which tries to identify all effective theories which are and are not consistent with quantum gravity. All the effective field theories that are consistent with string theory are in the *landscape* while the rest is in the *swampland* [3].

An approach to study the landscape and swampland would be to try all possible compactifications on Calabi–Yau manifolds. Unfortunately this is not possible due to the huge number of possible Calabi–Yau manifolds. The number of constructed Calabi–Yau manifolds by Kreuzer and Starke is $473,800,776$ [4]. Working through all these possible compactifications is not possible due to the computational complexity [5]. Therefore finding criteria for when theories are in the swampland or not is a big open problem. There are many criteria stated as swampland conjectures [3], an example being the *Swampland Distance Conjecture*. To study these conjectures it is helpful to classify Calabi–Yau manifolds. It is however very difficult to classify the Calabi–Yau manifolds, since the data is either very difficult to handle and basis dependent or to generic and captures only a little bit of information about the manifold.

In recent work [6, 7, 8] a new method has been proposed to classify Calabi–Yau threefolds. This method is based on the mixed Hodge structure and Deligne splitting [9, 10]. The mixed Hodge structure is a refinement to the Hodge structure and it can be studied at the boundaries of the moduli space of a Calabi–Yau manifold, where the normal Hodge structure is not well defined any more at these boundaries. It can be used to classify the degeneration limits [11]. This classifica-

tion and especially the allowed types of subsequent degenerations by normally intersecting divisors will be the main study of this thesis. The construction of the classification data is based on advanced mathematics but although difficult to fully understand, computing the actual classification is simple and can be efficiently done on a computer. Due to this efficient way of classifying the Calabi–Yau manifolds, we can use this on two data sets of constructed Calabi–Yau manifolds, one set classified by Kreuzer and Skarke and the other set being the complete intersection Calabi–Yau manifolds (CICY).

In these modern days, having lots of data available is almost always linked with using machine learning. Since 2012, when the famous paper by Krizhevsky, Sutskever and Hinton [12] was published, the interest in machine learning has been ever increasing. This can also be seen by the number of physics papers that have been published in recent years using the techniques of machine learning. In 2017 the first applications of machine learning in string theory were published by four independent groups [13], [14], [15] and [16]. Although the term machine learning has had a lot of attention since 2012, it has been around for much longer and is a broader field than the study of (deep) neural networks alone. For instance in 1984 *Probably Approximately Correct* (PAC) learning, which is a general formal learning model [17] was already introduced by Valiant [18].

The goal of this thesis is to apply machine learning techniques on the vast amount of available data on Calabi–Yau manifolds. We want to study the enhancements of the classification of Calabi–Yau manifolds at their infinite volume limits. To do this with machine learning techniques, we have developed an algorithm. This algorithm uses decision trees together with a bagging procedure from Positive Unlabelled learning, concepts that will be introduced in this thesis. With this algorithm we are able to study the enhancements patterns obtained from Calabi–Yau manifolds and find the known enhancement rules as presented in [6]. With this algorithm we want to give new insights into the rules of the allowed enhancements as presented by [11] and hope to answer the question raised in [6] on the allowed combinations of two Deligne splittings.

Throughout this thesis we assume that the reader is familiar with complex geometry and specifically the study of Calabi–Yau manifolds. Terms such as complex structure, moduli space, Hodge decomposition and cohomology should be known. For the purpose of this thesis, it would have become too elaborate to also explain the basic complex geometry. If the reader feels uncertain about any of these objects we would recommend [19] for an introduction into complex geometry (or [20] for a physicist approach), [21] for learning about Calabi–Yau manifolds in string theory and [22] for an introduction to Hodge structures.

This thesis is structured as follows. In Chapter 2 we present the theory behind the classification of the degeneration at the large volume point. This is done by introducing the Mixed Hodge Structure and the Nilpotent orbit theorem. In Chapter 3 we will give general introduction to machine learning and then discuss the specific decision tree algorithm which is the basis of our method. To use this on the available data we have to use so called Positive Unlabelled machine learning, which we also introduce in this chapter. Next we present the algorithm used to find the enhancement rules in Chapter 4 and also discuss the validity of

the algorithm. In Chapter 5 we present and discuss the results of the algorithm on the enhancement rules.

Chapter 2

Hodge theory and classifying the limits

In this chapter we will provide the mathematical tools which are necessary to classify the Calabi–Yau threefolds (Y_3). This classification is done by studying the moduli space of the Y_3. We first introduce the period vector $\mathbf{\Pi}$ and its behaviour at the boundary of the complex structure moduli space. This is done by working with the monodromy of the period vector, giving us the monodromy matrix. Next we present the Nilpotent Orbit theorem, in which this matrix is used. This theorem provides an expression for the period vector at the large volume point up to exponential corrections. We then introduce the Mixed Hodge structure and Deligne splitting and we use the approximation of the period vector at the large volume point to study these structures. By the Deligne splitting we can finally classify the Calabi–Yau manifolds, due to the restrictions of the values in the splitting. In the last part of this chapter, we explain how to explicitly calculate the Deligne splitting for Calabi–Yau threefolds. This chapter is based on [6] and [7], which are based on the work presented in [11] where the used notation has been introduced.

2.1 The period vector

We work with Y_3, for which we have a complex structure and Kähler moduli space. The goal is to study the complex structure moduli space $\mathcal{M}_{cs}$, especially the boundaries of this space. Via mirror symmetry this can all be mapped to the Kähler moduli space and thus the presented method can also be used to study $\mathcal{M}_K$, we will shortly discuss this but refer to for instance [6] for a more detailed story. The boundaries (or singularities) of the moduli space correspond to the degeneration of the Calabi–Yau manifold. The boundaries can be studied using Mixed Hodge structures. We will first focus our discussion on $\mathcal{M}_{cs}$ but we keep in mind that this can be directly related to $\mathcal{M}_K$ by mirror symmetry.

Since $\mathcal{M}_{cs}$ is also a Kähler manifold, it has a Kähler potential which can be locally determined by

$$K(z,\bar{z}) = -\log\left[i\int_{Y_3}\Omega\wedge\bar{\Omega}\right], \tag{2.1.1}$$

with Ω a $(3,0)$-form. This form can be expanded into a real integral basis $\gamma_{\mathcal{I}}$, $\mathcal{I}=1,...,2h^{2,1}+2$

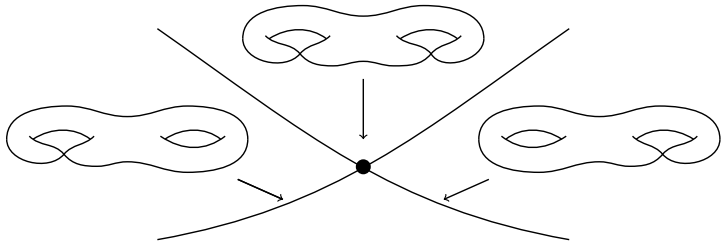


Figure 2.1 – Three degenerate complex manifolds, represented by a genus-two Riemann surface. The two divisors of the discriminant locus Δ intersect normally and at this intersection the singularities of the Calabi–Yau manifold worsens.

$$\Omega = \Pi^I \gamma_{\mathcal{I}}, \quad \eta_{\mathcal{I}\mathcal{J}} = -\int_{Y_3} \gamma_{\mathcal{I}} \wedge \gamma_{\mathcal{J}}. \tag{2.1.2}$$

So plugging this into Equation 2.1.1 gives $K(z,\bar{z}) = -\log\left[i\bar{\pi}^{\mathcal{I}}\eta_{\mathcal{I}\mathcal{J}}\pi^{\mathcal{J}}\right]$.

For us, it's crucial that we can expand Ω into a real integral basis since the π^I terms, called the periods can be studied using powerful mathematical techniques. As a notation we use the bold lettered π to indicate the vector as

$$\mathbf{\Pi} = \begin{pmatrix} \Pi^1 \\ \vdots \\ \Pi^{2h^{1,1}+2} \end{pmatrix}.$$

For now we only need to know that the period vector $\mathbf{\Pi}$ is a multi-valued function that experiences monodromies [7]. When a function experiences a monodromy this means that when it starts at a point z and is moved around a singular point z^* back to z, it has a different value. This means that the behavior of $\mathbf{\Pi}$ tells us something about the singularities of $\mathcal{M}_{cs}$ and these are exactly the points we want to study. We will later study the period vector in more details.

The points in $\mathcal{M}_{cs}$ for which the underlying Y_3 becomes singular form the discriminant locus Δ_s. By moving on Δ_s, the singularity of the Y_3 changes and can get worse, see Figure 2.1. This will be very useful for us. However Δ_s can have many intersecting components, making its structure non-trivial and therefore difficult to study. It can be shown ([7] and references therein) that there exists a resolution of Δ_s to more well behaved discriminant locus Δ , for which $\Delta = \cup_k \Delta_k$ such that all Δ_k intersect normally. In essence this means that we can study how the Y_3 becomes more degenerate when taking the intersection with more and more resolutions.

Since $\mathbf{\Pi}$ experiences monodromies, we can study this when moving around Δ. This can be done by introducing local coordinates z^I such that Δ_k is given by $z^k = 0$. For these coordinates, moving around Δ_k corresponds to $z^k \to e^{2\pi i} z^k$. This also directly gives us a way to study the behavior of a selection of the Δ_k normally intersecting divisors by taking for instance only $z^n = z^m = 0$ which gives the intersection of Δ_n and Δ_m.

When we take $z^k \to e^{2\pi i} z^k$, we get

$$\mathbf{\Pi}(\dots, e^{2\pi i} z^k, \dots) = T_k^{-1} \mathbf{\Pi}(\dots, z^k, \dots), \tag{2.1.3}$$

where T_k is the monodromy operator which acts on the integral basis of 3-forms. These operators are unipotent and we can define nilpotent matrices from these by

$$N_k = \log(T_k), \tag{2.1.4}$$

which is called the log-monodromy matrix. These matrices will play a crucial role in classifying the Y_3.

2.1.1 Nilpotent Orbit theorem

Now that we have discussed the log-monodromy matrix, we can further study the period vector for which we need the next crucial ingredient; the Nilpotent Orbit theorem. This theorem gives an approximation of $\mathbf{\Pi}$ on a local patch which contains the discriminant locus Δ. This can be made in such a way that $\mathbf{\Pi}$ is studied near the intersection of n_ϵ divisors Δ_i but away from any other intersection. In local coordinates this then corresponds to

$$z^{\mathcal{I}} = (z^i, \zeta^k) \;\text{ s.t. }\; \Delta_i = 0 \;\text{ is given by }\; z^i = 0. \tag{2.1.5}$$

The Nilpotent orbit theorem states that [7]

$$\mathbf{\Pi}_{\text{nil}} = \exp\Big[\sum_{j=1}^{n_\epsilon} -\frac{1}{2\pi i}(\log z^j)N_j\Big]\mathbf{a}_0(\zeta) \equiv \exp\Big[\sum_{j=1}^{n_\epsilon} -t^j N_j\Big]\mathbf{a}_0(\zeta) \tag{2.1.6}$$

approximates the period $\mathbf{\Pi}$ up to polynomial corrections in z^i near $z^i = 0$, where N_j are the log-monodromy matrices. $\mathbf{a}_0(\zeta)$ is a holomorphic function and for later use we have defined the coordinates $t^i \equiv \frac{1}{2\pi i}\log(z^i)$. In these coordinates the Nilpotent orbit approximations holds up to exponential corrections. It is important that $\mathbf{\Pi}_{nil}$ has the same transformation as $\mathbf{\Pi}$. As we will see shortly, at the large complex structure [1] the Nilpotent orbit approximation can be associated to a limiting, polarized Mixed Hodge structure. This structure will be used to classify the Y_3.

2.2 Hodge Structure

The Mixed Hodge Structure (MHS) is a powerful mathematical tool and a refinement to the Hodge Structure (HS). A HS is a decomposition of a vector space, where the decomposition has to obey some relation. For the purpose of this thesis we will not discuss the underlying Hodge theory fully, for this we recommend [19], or more focused on our purpose [6],[7]. From Hodge theory we know that the third cohomology, for a given complex structure, splits as

$$H^3(Y_3, \mathbb{C}) = H^{3,0} \oplus H^{2,1} \oplus H^{1,2} \oplus H^{0,3}, \tag{2.2.1}$$

which is a Hodge decomposition of weight 3. The decomposition is heavily tied to the complex structure since the spaces $H^{p,q}$ consists of (p,q) forms

$$\alpha^{p,q} = a_{i_1\dots i_p \bar{j}_1\dots\bar{j}_q} dz^{i_1} \wedge \cdots \wedge dz^{i_p} \wedge d\bar{z}^{\bar{j}_1} \wedge \cdots \wedge d\bar{z}^{\bar{j}_q}.$$

[1] When not at the large complex structure you might also need $a_1(\zeta)$

When moving through $\mathcal{M}_{cs}$, although the space $H^3(Y_3, \mathbb{C})$ stays the same, the decomposition of the (p, q) forms change since the complex structure changes. This can be explicitly seen from the fact that $H^{p,q}$ are vector spaces and the period vector $\mathbf{\Pi}$ spans the space $H^{3,0}$. The period vector is a function of the complex structure moduli and thus if the moduli change, then this function changes and therefore $H^{3,0}$ changes. By moving towards and going to the boundaries of $\mathcal{M}_{cs}$, this decomposition breaks down since at the boundaries of $\mathcal{M}_{cs}$ the complex structure of the Y_3 degenerates and this decomposition of forms is not well defined any more. Again this can also explicitly be seen from the period vector, since $\mathbf{\Pi}_{nil}$ breaks down when sending $z^j \to 0$. Therefore the MHS is needed, which gives extra structure which is preserved even when the Y_3 becomes singular.

First we need to study how 2.2.1 changes when moving through $\mathcal{M}_{cs}$, for which we can best use the filtration definition of the HS. Suppose we have a Hodge structure H of weight n. The Hodge filtration F is given by [19]

$$F^i H_{\mathbb{C}} \equiv \oplus_{p \geq i} H^{p,q},$$

and

$$H_{\mathbb{C}} = F^0 \supset F^1 \supset ... \supset F^k \supset \{0\}.$$

In the case of Y_3 this gives us the following

$$F = (F^3, F^2, F^1, F^0)$$

with

$$F^3 = H^{3,0}, \qquad F^2 = H^{3,0} \oplus H^{2,1}$$
$$F^1 = H^{3,0} \oplus H^{2,1} \oplus H^{1,2}, \quad F^0 = H^{3,0} \oplus H^{2,1} \oplus H^{1,2} \oplus H^{0,3}$$

It is important to note that $H^{3,0}$ is the space spanned by $(3, 0)$-forms and that Ω is an unique $(3, 0)$-form. Therefore F^3 is encoded by Ω.

This is very useful since we can obtain the other spaces F^p by taking derivatives on F^3. This is done by using the Gauss Manin connection, $\nabla_I \equiv \nabla_{\partial/\partial z^I}$ by which we have $\nabla_I F^p \subset F^{p-1}$ and for Y_3 we even have that all elements of F^p for $p < 3$ are obtained by derivatives of F^3 [6]. In this manner Ω contains all the information on the filtration and with the result from the Nilpotent Orbit theorem we have a beautiful and neat way to study Ω.

2.2.1 Polarized Hodge structure

Although not needed at this point, we also introduce the *polarized Hodge structure*, which is a Hodge structure H with the extra requirement that it has a bilinear form $S(\dot{;})$ on $H_{\mathbb{C}}$ such that the following conditions are satisfied

$$S(H^{p,q}, H^{r,s}) = 0 \qquad \text{for } p \neq s,\ q \neq r, \tag{2.2.2}$$
$$i^{p-q} S(v, \bar{v}) > 0 \quad \text{for any non-zero } v \in H^{p,q}. \tag{2.2.3}$$

We have already introduced the anti-symmetric matrix $\eta = \eta_{\mathcal{I}\mathcal{J}}$, which we can use to define the anti-symmetric bilinear form

$$S(v,w) \equiv S(\mathbf{v},\mathbf{w}) = \mathbf{v}^T \eta \mathbf{w} \equiv -\int_{Y_3} v \wedge w. \tag{2.2.4}$$

This bilinear form gives the Hodge structure coming from the third cohomology its polarization. We will need this to determine the type of Deligne splitting that we are going to introduce next.

2.3 Deligne Splitting

The MHS is related to the Deligne splitting, which gives a more straightforward way to relate objects to the points in Δ. The Deligne splitting can be seen as a refinement of the Hodge decomposition 2.2.1 which keeps structure even when Y_3 has become singular. The Deligne splitting is given by $I^{p,q}$, $p,q = 0,\dots 3$ and pictorially it is given as

$$(H^{3,0},\ H^{2,1},\ H^{1,2},\ H^{0,3}) \quad \xrightarrow{\text{move to } \Delta} \quad \begin{array}{ccccccc} & & & I^{3,3} & & & \\ & & I^{3,2} & & I^{2,3} & & \\ & I^{3,1} & & I^{2,2} & & I^{1,3} & \\ I^{3,0} & & I^{2,1} & & I^{1,2} & & I^{0,3} \\ & I^{2,0} & & I^{1,1} & & I^{0,2} & \\ & & I^{1,0} & & I^{0,1} & & \\ & & & I^{0,0} & & & \end{array} \tag{2.3.1}$$

This splitting can be studied by taking the filtration F on a point in Δ. In local coordinates we have that Δ_k is given by $z^k = 0$. The easiest situation is when only one z^k is sent to zero, so we will look at $z^1 = 0$ while all other coordinates are $z^k \neq 0$. This means that we look at the points of Δ_1 which are not in any other Δ_k. We sent $z^1 \to 0$ or in the t coordinates as introduced in the Nilpotent orbit theorem we have $t \to i\infty$, which gives us

$$\lim_{t^1 \to i\infty} e^{-t^1 N_1} F^p.$$

It is possible to associate

$$(F(\Delta_1^o), N_1) \to \{I^{p,q}(\Delta_1^o)\}.$$

To do this we have to use the monodromy weight filtration W_i, $i = 0,\dots 6$ which is a set of vector spaces that can be associated to N. For Y_3 these spaces can be explicitly determined from images and kernels of N, N^2 and N^3 (the

log-monodromy matrices in the Nilpotent orbit) as

$$\begin{array}{rcl} W_6 & = & V, \\ \cup & & \\ W_5 & = & \operatorname{Ker} N^3, \\ \cup & & \\ W_4 & = & \operatorname{Ker} N^2 + \operatorname{Im} N, \\ \cup & & \\ W_3 & = & \operatorname{Ker} N + \operatorname{Im} N \cap \operatorname{Ker} N^2, \\ \cup & & \\ W_2 & = & \operatorname{Im} N \cap \operatorname{Ker} N + \operatorname{Im} N^2, \\ \cup & & \\ W_1 & = & \operatorname{Im} N^2 \cap \operatorname{Ker} N, \\ \cup & & \\ W_0 & = & \operatorname{Im} N^3. \end{array} \tag{2.3.2}$$

This is where the MHS shows up, the weight filtration is the extra data needed to define from a HS the MHS. To associate F_Δ and N to $I^{p,q}$ we have

$$I^{p,q} = F^p_\Delta \cap W_{p+q} \cap \left(\bar{F}^q_\Delta \cap W_{p+q} + \sum_{j \geq 1} \bar{F}^{q-j}_\Delta \cap W_{p+q-j-1} \right). \tag{2.3.3}$$

For the classification of the Y_3 we need the following definitions and properties. The first very important feature of the nilpotent matrices N is how they act upon the filtrations F and W, $NF^N_\Delta \subset F_\Delta p - 1$ and $NW_i \subset W_{i-2}$. This gives us

$$NI^{p,q} \subset I^{p-1,q-1}. \tag{2.3.4}$$

Furthermore we define the primitive parts

$$P^{p,q} = I^{p,q} \cap \ker N^{p+q-2}, \tag{2.3.5}$$

which can be used in the following

$$I^{p,q} = \bigoplus_{i \geq 0} N^i (P^{p+i,q+i}). \tag{2.3.6}$$

Since the MHS comes from HS as the decomposition of the third cohomology of a Y_3, which has a polarization, the MHS has the same polarization and is therefore a polarized MHS. This gives the following properties

$$S(P^{p,q}, N^l P^{r,s}) = 0 \text{ for } p+q = r+s = l+3 \text{ and } (p,q) \neq (s,r), \tag{2.3.7}$$

$$i^{p-q} S(v, N^{p+q-3} \bar{v}) > 0 \text{ for } v \in P^{p,q}, v \neq 0. \tag{2.3.8}$$

Now $I^{p,q}$ is clearly dependent on the point considered in Δ and thus differs when taking (more) intersections.

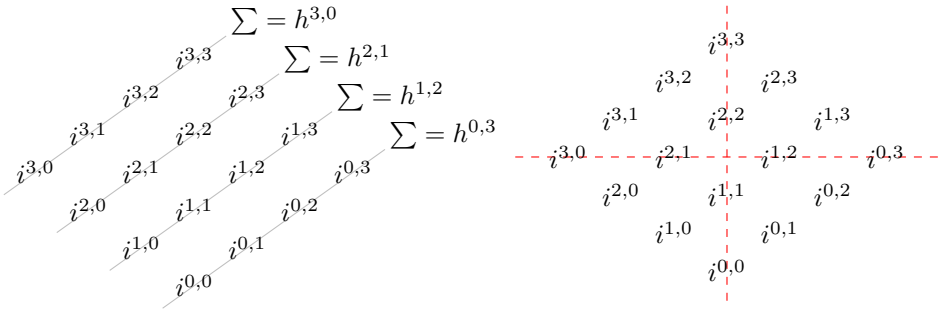


Figure 2.2 – In the left diamond, the summation properties of the diagonal rows is shown. The rows all sum to $h^{p,q}$. In the left diamond the symmetry axis for the diamonds are shown.

To use the Deligne splitting as classification it is useful to introduce the limiting Hodge diamond.

$$\begin{array}{ccccccc} & & & i^{3,3} & & & \\ & & i^{3,2} & & i^{2,3} & & \\ & i^{3,1} & & i^{2,2} & & i^{1,3} & \\ i^{3,0} & & i^{2,1} & & i^{1,2} & & i^{0,3} \\ & i^{2,0} & & i^{1,1} & & i^{0,2} & \\ & & i^{1,0} & & i^{0,1} & & \\ & & & i^{0,0} & & & \end{array} \quad , \qquad i^{p,q} = \dim_{\mathbb{C}} I^{p,q} \ . \tag{2.3.9}$$

By the relation between $I^{p,q}$ and $H^3(Y_3, \mathbb{C})$ we get the following equality

$$h^{p,3-p} = \sum_{q=0}^{3} i^{p,q}, p = 0, \dots 3, \tag{2.3.10}$$

see Figure 2.2. Since $h^{3,0} = 1$ for Y_3, the number of shapes of the limiting Hodge diamonds is reduced to 4, one for each value of $i^{3,d}$ with $d = 0, \dots, 3$. These are labelled with Roman numbers as I, II, III and IV (see Table 2.1). This notation has been introduced by [11]. Furthermore we have

$$i^{p,q} = i^{q,p} = i^{3-p,3-q},$$

thus $i^{2,2} = i^{1,1}$ and $i^{1,2} = i^{2,1}$. This gives mirror symmetries in the limiting Hodge diamond as given in Figure 2.2. Due to the symmetries there is only one independent value $i^{p,q}$ which are denoted as a,b,c and d. This gives the following $4m$ possible Deligne splittings

$$\begin{aligned} &\mathrm{I}_a \ , && a = 0, ..., m \ , \\ &\mathrm{II}_b \ , && b = 0, ..., m-1 \ , \\ &\mathrm{III}_c \ , && c = 0, ..., m-2 \ , \\ &\mathrm{IV}_d \ , && d = 1, ..., m \ , \end{aligned} \tag{2.3.11}$$

where m is the dimension of the considered moduli space. In our case this will be the Kähler moduli space and thus $m = h^{1,1}$.

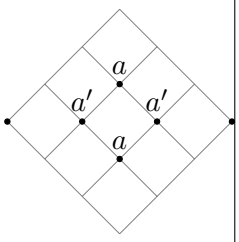
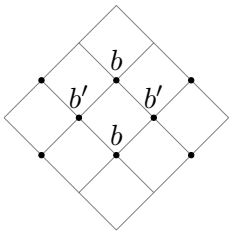
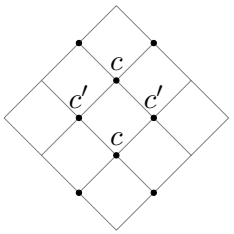
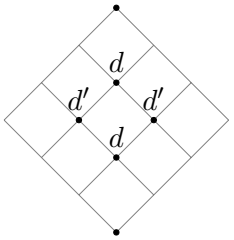
name	Hodge diamond	labels	N, η
I_a		$a + a' = m$ $0 \leq a \leq m$	$\mathrm{rank}(N, N^2, N^3)$ $= (a, 0, 0)$ ηN has a negative eigenvalues
II_b		$b + b' = m - 1$ $0 \leq b \leq m - 1$	$\mathrm{rank}(N, N^2, N^3)$ $= (2 + b, 0, 0)$ ηN has b negative and 2 positive eigenvalues
III_c		$c + c' = m - 1$ $0 \leq c \leq m - 2$	$\mathrm{rank}(N, N^2, N^3)$ $= (4 + c, 2, 0)$
IV_d		$d + d' = m$ $1 \leq d \leq m$	$\mathrm{rank}(N, N^2, N^3)$ $= (2 + d, 2, 1)$

Table 2.1 – The possible different types and values of Deligne splitting, together with their corresponding limiting Hodge diamond. The classification of the diamonds by type and value was introduced by [11]. The value m corresponds to the dimension of the moduli space we are working in. In the diamonds, the intersections without a dot represent a zero and those with a dot but without a letter represent an one. In the third column the relation and restriction on the values in the diamond are given. In the fourth column a specific way to determine the type of Deligne splitting is given.[7]

2.3.1 Determining the Deligne splitting

To be able to use the Deligne splitting to classify the Y_3, we want to be able to explicitly determine the splitting easily. From Equations 2.3.4,2.3.6, 2.3.7 and 2.3.8 we can determine that the ranks of N^k are different for the different types of Deligne splitting.

As an example, we could look at the I_a. In this case we have $i^{3,0} = i^{0,3} = 1$, $i^{2,2} = i^{1,1} = a$ and $i^{2,1} = i^{1,2} = a'$, so all other $i^{p,q}$ are zero and thus their corresponding spaces are empty. So if we apply N to the non-empty spaces we see that

$$NI^{2,2} \subset I^{1,1},\ NI^{1,1} \subset I^{0,0} \text{ and } N^2 I^{2,2} \subset I^{0,0} = \emptyset.$$

So we see that the $\text{rank}(N^2) = 0$ (and therefore also $\text{rank}(N^3) = 0$). Similar analysis can be done for the other 3 types. However, for both types I and II $\text{rank}(N^2) = 0$, thus to distuinguish between those types we also need to determine the sign of the eigenvalues of ηN. This can be done using the polarization condition.

Up until now our discussion has taken place in $\mathcal{M}_{cs}$, but we will now use mirror symmetry to explicitly calculate the Deligne splitting in the Kähler moduli space. By mirror symmetry we know that for every Y_3 there is a mirror manifold $\tilde{Y}_3$ related to it. The association is via the large complex structure point, which is mapped to the large volume point in the mirror Calabi–Yau manifold [6]. A full discussion of how to do this properly can be found in [6, 7], but for us it is enough to know that we can do this mapping and apply the whole mathematical theory we just discussed to the large volume points in Kähler moduli space.

Now we only need an explicit way to compute N and η for a point in Δ to determine the Deligne splitting at this point. It can be shown [6] that the period vector $\mathbf{\Pi}$ can be written in terms of the topological terms

$$\mathcal{K}_{IJK} = \int_{Y_3} \omega_I \wedge \omega_J \wedge \omega_K, \quad c_I = \frac{1}{24} \int_{Y_3} \omega_I \wedge c_2(Y_3), \text{ and } \chi = \int_{Y_3} c_3(Y_3), \tag{2.3.12}$$

where $\mathcal{K}_{IJK}$ are the triple intersection numbers and $c_2(Y_3)$ and $c_3(Y_3)$ are the Chern classes, as [6]

$$\mathbf{\Pi} = \begin{pmatrix} 1 \\ t^I \\ \frac{1}{2}\mathcal{K}_{IJK} t^J t^K + \frac{1}{2}\mathcal{K}_{IJJ} t^J - c_I \\ \frac{1}{6}\mathcal{K}_{IJK} t^I t^J t^K - (\frac{1}{6}\mathcal{K}_{III} + C_I) t^I + \frac{i\zeta(3)\chi}{8\pi^3} \end{pmatrix}. \tag{2.3.13}$$

With the expression for the period vector (2.3.13) we can now determine an explicit form for the monodromy transformation matrices by apply the shift $t^I \to t^I + 1$ to $\mathbf{\Pi}$, which is equivalent to taking $z^k \to e^{2\pi i} z^k$ to Equation 2.1.3. This gives

$$T_A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -\delta_{AI} & \delta_{IJ} & 0 & 0 \\ 0 & -\mathcal{K}_{AIJ} & \delta_{IJ} & 0 \\ 0 & \frac{1}{2}(\mathcal{K}_{AAJ} + \mathcal{K}_{AJJ}) & -\delta_{AJ} & 1 \end{pmatrix}, \tag{2.3.14}$$

and from this we get, using Equation 2.1.4

$$N_A = \begin{pmatrix} 0 & 0 & 0 & 0 \\ -\delta_{AI} & 0 & 0 & 0 \\ -\frac{1}{2}\mathcal{K}_{AAI} & -\mathcal{K}_{AIJ} & 0 & 0 \\ \frac{1}{6}\mathcal{K}_{AAA} & \frac{1}{2}\mathcal{K}_{AJJ} & -\delta_{AJ} & 0 \end{pmatrix}. \tag{2.3.15}$$

It is also possible to determine an explicit expression for η [7]

$$\eta = \begin{pmatrix} 0 & -\frac{1}{6}\mathcal{K}_{JJJ} - 2b_J & 0 & -1 \\ \frac{1}{6}\mathcal{K}_{III} + 2b_I & \frac{1}{2}(\mathcal{K}_{IIJ} - \mathcal{K}_{IJJ}) & \delta_{IJ} & 0 \\ 0 & -\delta_{IJ} & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}. \tag{2.3.16}$$

To determine the type of Deligne splitting, we need to determine the rank of N_A, N_A^2 and N_A^3. Due to the triangular structure of N_A this will turn out to be rather simple. In the next section we will discuss how the Deligne splitting can be determined when taking the normal intersection of multiple divisors, which is the more general case. We will present the explicit calculation that we will use to determine the Deligne type.

2.4 Enhancement rules

Having introduced a way to associated the Deligne splitting to specific limit in the moduli space, we want to apply this to classify the Calabi–Yau manifolds. To do this, we will take all combinations of coordinates in the Kähler moduli space and determine the type of Deligne splitting when sending the coordinates in the combination to $i\infty$, keeping the real part constant. Roughly this means that we take all the limits in which one coordinate t^{i_1} is send to $i\infty$ and determine the resulting Deligne splitting, than we take all the limits in which two coordinates t^{i_1}, t^{i_2} are both send to $i\infty$ and determine the Deligne splitting. This process is continued until all the possible limit combinations are classified. To do this properly, we would have to introduce an index set $\mathcal{I} = (i_1, \ldots, i_n)$ and define a growth sector. However, for us it is sufficient to have this picture of sending groups of coordinates to infinity. For a detailed explanation we refer to [11] for a more rigid approach and we refer to [6] for a approach more applied to our case. We will just discuss the results which are most important for our discussion.

To determine the Deligne splitting, we have to find the log-monodromy matrix for each limit. It has been shown that this matrix for a specific limit

$$t^{\mathcal{I}} \equiv (t^{i_1}, \ldots, t^{i_n}) \to i\infty,$$

is given by

$$N_{(\mathcal{I})} = N_{i_1} + \cdots + N_{i_n},$$

where $\mathcal{I}$ is previously introduced the index set and N_{i_k} is the log-monodromy matrix belongs to the corresponding coordinate sent to $i\infty$. In fact it was shown in [23] that any positive linear combination of N_{i_k} can be used. With the log-monodromy matrix we are able to determine the Deligne splitting at the limit. By staring with an index set $\mathcal{I}$ with only one index and adding one extra coordinate at the time, we can determine so called *enhancement chains* [6] of the form

$$\mathrm{I}_0 \xrightarrow{t^{i_1}\to i\infty} \mathsf{Type\ A}_{(i_1)} \xrightarrow{t^{i_2}\to i\infty} \mathsf{Type\ A}_{(i_2)} \xrightarrow{t^{i_3}\to i\infty} \ldots \xrightarrow{t^{i_n}\to i\infty} \mathsf{Type\ A}_{(i_n)}\,.$$

Taking a certain index set corresponds to taking the intersection of the corresponding divisors of the discriminant locus Δ.

It turns out that there are some restrictions to which enhancements are allowed after each other in such a chain [11]. The first thing is that the type can only increase, meaning that for instance enhancemeents of the form IV $\to$ II are not possible. The full list of known enhancement rules is given in Table 2.2. The rules given in Table 2.2 only give a restriction on the chains, but not on the combination of limits. For instances when individually sending $t^{i_n} \to i\infty$ and $t^{i_m} \to i\infty$, we

starting type	enhanced type
I_a	$\mathrm{I}_{\hat{a}}$ for $a \leq \hat{a}$
	$\mathrm{II}_{\hat{b}}$ for $a \leq \hat{b}$, $a < m$
	$\mathrm{III}_{\hat{c}}$ for $a \leq \hat{c}$, $a < m$
	$\mathrm{IV}_{\hat{d}}$ for $a < \hat{d}$, $a < m$
II_b	$\mathrm{II}_{\hat{b}}$ for $b \leq \hat{b}$
	$\mathrm{III}_{\hat{c}}$ for $2 \leq b \leq \hat{c}+2$
	$\mathrm{IV}_{\hat{d}}$ for $1 \leq b \leq \hat{d}-1$
III_c	$\mathrm{III}_{\hat{c}}$ for $c \leq \hat{c}$
	$\mathrm{IV}_{\hat{d}}$ for $c+2 \leq \hat{d}$
IV_d	$\mathrm{IV}_{\hat{d}}$ for $d \leq \hat{d}$

Table 2.2 – List of all allowed enhancements of degeneration types as determined in [11], where m is the dimension of the moduli space. Table is taken from [6].

get two Deligne splittings denoted as Type A_{i_n} and Type A_{i_m}. It is not known how Type A_{i_n} and Type A_{i_m} restrict the Deligne splitting found when sending both $t^{i_n}, t^{i_m} \to i\infty$. Some first results for specific cases have been found [11], but the full rules remain to be unknown. In thesis we will call these unknown rules the double enhancement rules, reflecting the fact that they govern how two limits meet. The known rules are called the single enhancement rules.

For the large volume regime we only need the intersection numbers to determine the log-monodromy matrix. This is very convenient since this data is available for Calabi–Yau manifolds. When taking a certain index set $\mathcal{I}$ the corresponding log-monodromy matrix is determined as [6]

$$N_{(\mathcal{I})} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ -\sum_{i\in\mathcal{I}} \delta_{iI} & 0 & 0 & 0 \\ -\frac{1}{2}\sum_{i\in\mathcal{I}} \mathcal{K}_{iiI} & -\sum_{i\in\mathcal{I}} \mathcal{K}_{iIJ} & 0 & 0 \\ \frac{1}{6}\sum_{i\in\mathcal{I}} \mathcal{K}_{iii} & \frac{1}{2}\sum_{i\in\mathcal{I}} \mathcal{K}_{iJJ} & -\sum_{i\in\mathcal{I}} \delta_{iJ} & 0 \end{pmatrix} . \tag{2.4.1}$$

To simplify the notation we define, as is done in [6]

$$\mathcal{K}_{IJ}^{(\mathcal{I})} \equiv \sum_{i\in\mathcal{I}} \mathcal{K}_{iIJ} \,, \qquad \mathcal{K}_{I}^{(\mathcal{I})} \equiv \sum_{i,j\in\mathcal{I}} \mathcal{K}_{ijI} \qquad \text{and} \qquad \mathcal{K}^{(\mathcal{I})} \equiv \sum_{i,j,k\in\mathcal{I}} \mathcal{K}_{ijk} \,. \tag{2.4.2}$$

With these quantities it is straightforward to compute $N_{\mathcal{I}}^2$ and $N_{\mathcal{I}}^3$, which results in

$$N_{(\mathcal{I})}^2 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \mathcal{K}_I^{(\mathcal{I})} & 0 & 0 & 0 \\ 0 & \mathcal{K}_J^{(\mathcal{I})} & 0 & 0 \end{pmatrix} , \qquad N_{(\mathcal{I})}^3 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -\mathcal{K}^{\mathcal{I}} & 0 & 0 & 0 \end{pmatrix} . \tag{2.4.3}$$

We see that the only possible non-zero term in $N_{\mathcal{I}}^3$ is $-\mathcal{K}^{\mathcal{I}}$. When this term is non-zero the Deligne splitting is of type IV, but when this term is zero, it is clear that the rank of $N_{\mathcal{I}}^3$ is also zero. In this case we directly know that the type of Deligne splitting is not IV. In a similar way we can compute the ranks of $N_{\mathcal{I}}^2$ and $N_{\mathcal{I}}$ and use this to determine the Deligne splitting. In the case that both $N_{\mathcal{I}}^3$ and $N_{\mathcal{I}}^2$ have rank 0, we also need to compute the sign of the eigenvalues of $\eta N_{\mathcal{I}}$. This is needed to distinguish between a type I and II. However, a singularity of type I_a will never occur in the large volume regime, thus we do not have to check this type [6].

Combining the expressions in Equations 2.4.1 and 2.4.3 with the information in the last column of Table 2.1, we can now readily determine the Deligne splittings only using the intersection numbers. This gives us the conditions presented in Table 2.3.

Type	$\mathrm{rk}\mathcal{K}^{\mathcal{I}}$	$\mathrm{rk}\mathcal{K}_I^{(\mathcal{I})}$	$\mathrm{rk}\mathcal{K}_{IJ}^{(\mathcal{I})}$
II_b	0	0	b
III_c	0	1	$c+2$
IV_d	1	1	d

Table 2.3 – List of types in the large volume regime in the limit $t^{\mathcal{I}} = (t^{i_1}, ..., t^{i_n}) \to i\infty$. For numbers and vectors, we define the ranks $\mathrm{rk}(\mathcal{K}^{(\mathcal{I})})$ and $\mathrm{rk}(\mathcal{K}_I^{(\mathcal{I})})$ to be either 0 or 1, depending on whether $\mathrm{rk}(\mathcal{K}^{(\mathcal{I})}) = 0$ and $\mathcal{K}_I^{(\mathcal{I})} = 0 \ \forall \ I$. [6]

Chapter 3

Machine learning

As explained in Section 2.4 we want to find rules for the allowed double enhancements using machine learning techniques. In this chapter, we will present a short, general introduction to machine learning. We will discuss a definition of machine and outline the general learning model. For this we will explain what is meant with *task*, *performance* and *experience*. After this more theoretical discussion, we will present the more practical side of machine learning in Section 3.2. We explain how we can actually make a computer program perform better at a task. This discussion will also lead us to the discussion on how to validate the performance of the machine learning method and how to prevent under and overfitting. In Section 3.3 we will then discuss the specific machine learning method of decision trees, which form the basis of our final algorithm. The last section of this chapter is devoted to a specific area of machine learning, called Positive Unlabelled (PU) learning. Due to the nature of our data we need to use this specific variant of machine learning.

3.1 General Machine Learning

Traditional computer programs and algorithms are instructions which the computer will exactly follow. Everything the computer has to do, has to be added to the instructions, from storing data to performing computations. This also means that we can exactly tell what the computer is doing, since it has been exactly told what to do. This works very well for tasks for which we know how to perform, or in computer language for which we know how to write code for it.

However, when we encounter a problem for which we don't know how to exactly perform the task, it is seemingly impossible to write an computer program for it. An example of this is classification of objects, for example deciding whether something is a dog or a cat. Deciding whether an animal is a dog or a cat is for many humans an easy task. Somehow we intrinsically know when an animal is a dog or cat. Most humans have learned this at some point in their life, probably just by observing and learning from the people around them. However, despite being a seemingly very simple task, it is very difficult to state exactly the difference between a cat and a dog and how humans are able to classify them. For instance we might say that an animal having a whisker is a cat. But in most cases we are also able to distinguish between cats and dogs when we do not see their head, or we encounter a cat whose whiskers have been cut of. Thus there are more

features which distinguish cats from dogs, but it is very hard (if not impossible) to exactly write those down. Especially, because there appear to be exceptions to each of the rules. To have a computer algorithm classify cats and dogs, we would have to program exactly those distinguishing features for all the cases. Therefore it is simply too difficult to write a (regular) computer algorithm to classify cats and dogs. This is where machine learning comes in handy. Machine learning is a method to make a computer perform a task, without explicitly telling the computer how to perform the task. It will learn how to perform this task by itself, just like human learns how to classify cats and dogs.

3.1.1 Machine learning definition

Machine learning can be defined in a formal way, although it is difficult to capture the vast amount of different machine learning variants. A proper definition of learning by a computer is given in [24]:

Definition 3.1.1.
A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

This is a very broad definition, due to the fact that machine learning is also a very broad topic. An insightful way to think about this definition is to see experience E as some input data, which the algorithm uses to learn to perform a task. The task T can be anything we want the computer to do with the data, ranging from classifying different objects to driving a car. The measure P is used to tell the algorithm if it is performing the task correctly. The main idea of machine learning is to allow the algorithm to alter itself such that the performance improves.

In the case of deciding if an animal is a cat or dog, the experience E (the data) could be a lot of pictures of cats and dogs or a list of features for all encountered instances. The task T would be to classify whether a given data point is a cat or dog. The measure P could be simply the percentage of correctly classified animals, but it can also be another, more difficult function on the performance of the algorithm. Common nomenclature is to say that the machine learning algorithm is trained on the (training) data to learn to perform the task. In the end the goal is to have a machine learning algorithm which has learned to perform a task well on data which it has not seen before, this is called generalization. Thus in the cat and dog classification case it would have learned based on a number of pictures such that it can (almost always) classify new cases correctly.

3.1.2 Task

The goal of machine learning is to perform a task and the learning is the method to make the algorithm better at performing the task. The task can in general be described by what the algorithm should do with samples or data points. Samples are given in terms of features, which are (often) represented as vectors in a n dimensional space $\mathbb{R}^n$, called the feature space. See Table 3.1 for some example samples in a 4 dimensional feature space. Every sample can be written as a vector, for instance $\mathbf{x}_1 = (7.5, 0, 1, 0.4, 0.8)$ and the task can be described by a function t

with

$$t : \mathbb{R}^4 \to \{0, 1\},$$

where 0 represents a cat and 1 a dog. The function takes the feature functions and maps them to the correct animal.

	weight	whiskers	fur	tail length	ear length
sample 1	7.5	0	1	0.4	0.8
sample 2	2.3	1	1	1.2	0.2
sample 3	1.7	0	1	1.7	0.2
sample 4	3.3	0	1	0.7	0.6

Table 3.1 – The features of four example samples which can be used to classify wether an animal is a cat or dog.

Since many problems can be solved with machine learning, it is impossible to name all the different tasks. However to get some idea about the vast amount of possabilities and variety we will discuss some common, general types of tasks. These are classification, regression, data clustering and dimensionality reduction [25], [17].

Classification is the task of determining from the input what type of object this is. Image recognition, such as the one for cats and dogs belongs to this type task and it is characterized by the discrete classes. An object either belongs to a class or not. The continuous version of this is regression, in which the algorithm has to learn to predict a (numerical) value given some input. Data clustering is the task of getting a set of data points and finding common groups within them. For this task, we do not necessarily know or have different types of object. It can be used to find different types or groups within the data. Dimensionality reduction is the task of finding the features which are most distinguishing for the objects. In the case of finding the difference between cats and dogs, this might result in learning that a feature such as having a fur will be less important to classify the animal compared to having whiskers.

3.1.3 Performance

The performance measure is a key aspect of machine learning and it is described by the loss function. The goal of machine learning is to train the algorithm in such a way that it performs the given task as well as possible. The loss function is a measure of how well the task is performed. Due to the wide range of task, it is not possible to define one unique loss function but in general the loss function should be a map of the following form [17]

$$l : \mathcal{H} \times \mathcal{D} \to \mathbb{R}_+,$$

where $\mathcal{H}$ is called the *hypothesis space* and $\mathcal{D}$ the data on which the algorithm performs the task. The hypothesis space is the space containing all the possible solutions for the task which the algorithm can find. This space is chosen before starting the machine learning. It can be anything from ten linear functions to all functions that can be made from all combinations of continuous functions.

The data $\mathcal{D}$ can be either only the feature space $\mathbb{R}^n$ or $\mathbb{R}^n \times \mathcal{Y}$ where $\mathcal{Y}$ is the space of labels given to the points in $\mathbb{R}^n$. In the case of cats and dogs $\mathcal{Y}$ thus contains whether an instance is a cat or a dog and the loss function could simply be the percentage of correctly classified samples. In some cases, for instance deep neural networks, the used loss function has a huge effect on the capability of the algorithm to learn.

To get some intuition about the loss function, it is useful to work with the example of classifying cats and dogs. The task of this machine learning algorithm is to correctly classify the different animals. In the end we want to get as many correct predictions as possible, thus naively this should be our performance and the loss function could be the fraction of incorrectly labelled data. So lets assume we have some training set $\mathcal{X} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_1, y_1)\}$ and some classifier $h \in \mathcal{H}$. Then taking as a error function the fraction of incorrectly labelled would give the following

$$\mathcal{H} \times \mathcal{D} \to \mathbb{R}_+, (h, T) \mapsto \frac{|\{x_i | h(x_i) \neq y_i, x \in \mathcal{X}\}|}{|\mathcal{X}|}$$

The problem with this type of loss function is that it is not a smooth function. When taking a new $h' \in \mathcal{X}$ close to h might not have any effect on this fraction, since it classifies all objects in the same way. Only when the classifier classifies points differently the loss function, but the new function might be closer to the optimal solution in the hypothesis space and thus beneficial to finding the optimal solution. The goal of the algorithm is to find the solution in $\mathcal{H}$ for which the loss function is as low as possible. This is done in the learning process, by determining the value of the loss function of the current solution h and updating it to h' such that $l(h, \mathcal{X}) > l(h', \mathcal{X})$. An example for a smooth loss function would be the logarithmic loss, which can be used for a classification task where the algorithm outputs probabilities for a instance to belong to a certain class. In the case of cats and dogs, the cats are labelled as 0 and the dogs as 1. The log loss is then defined as

$$\sum_{x_i \in \mathcal{X}} -(y_i \log(p_i) + (1 - y_i) \log(1 - p_i)),$$

where p_i is the predicted probability of the instance x_i to be a cat and y_i its true label [26].

3.1.4 Experience

Due to the wide range of machine learning tasks, it is useful to distinguish between different types of machine learning. Often, machine learning is classified into two main classes **supervised** and **unsupervised** [25] but there is also third sub-type called **reinforcement** learning [17] as also discussed in [27]. The key difference between the types of machine learning is based on what kind of data is used to make the algorithm learn. The experience which a machine learning algorithm uses to learn is a (big) dataset. This dataset stores the features of the different instances but it can also store the class to which it belongs. The term dataset should be interpret rather generic, since the algorithm can also learn on data created by its own behaviour. In many types of machine learning the data points are not used just once, but rather the algorithm performs its task on the data and

uses its performance to change its own behaviour (this is the learning aspect). In the next step the same data is again used to check how the algorithm performs with this change.

Supervised learning
Labelled data is used for supervised learning and this could for example be learning to recognise emails on whether they are spam or not. The data would be a set of emails that are labelled as either spam or not. Supervised learning can also be used for regression tasks, in which the output is continuous instead of discrete. The name supervised learning comes from the fact that the algorithm is told when it is correct or not, since the true labels are known for the data, thus we have some kind of abstract teacher who teaches the algorithm. In this case the training set contains the features as well as the label of the samples, $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$, where N is the number of samples. In mathematical terms, the goal of supervised learning is to find a function $f : \mathcal{X} \to \mathcal{Y}$ such that

$$f(\mathbf{x}_i) = y_i$$

Theoretically it is possible to give an estimate on the performance of a supervised machine learning model, see for instance [17].

Unsupervised learning
When the data is not labelled, unsupervised machine learning is used. In this case the teacher is not present and the algorithm is not told if it has performed the task correctly. This can be used for example to learn to recognise patterns in the data often to cluster different samples or find outliers. In the case of the spam recognition, the algorithm will learn to recognise unusual emails or that certain emails are always kept unread. Mathematically unsupervised learning is based on finding the probability distribution, thus finding models as [27] $p(\mathbf{x}_i, \theta)$ where θ are random samples from the underlying probability distribution which are thus the observed points present in the dataset.

Reinforcement learning
The third sub-class is reinforcement learning and can be seen as semi-supervised learning. This type of machine learning often occurs in (computer) games, in which a reward is given for winning or completing a level but the algorithm has to completely figure out how this reward is reached. The data on which the algorithm learns doesn't contain labels but the algorithm is told when it has performed good or bad. The algorithm observes its environment and the task is often to perform an action on this environment [24]. This is done in a sequential fashion. This makes reinforcement have some important similarities with supervised learning and therefore it is not always seen as a separate type. An common example of reinforcement learning is learning to play a game, for instance chess. A reinforcement algorithm takes actions, in this case performing a move and reacts on the environment since the opponent also performs a move. Not after every move the algorithm is told how it performed, but only at some certain points. For instance by giving it a reward for capturing a piece or when it has won the game. This is an important aspect in which reinforcement learning differs. In [24] this is called *delayed reward*.

3.2 Machine learning methodology

Definition 3.1.1 only defines machine learning, it does not discuss the procedure. To get a good understanding of machine learning it is necessary to also know how a general algorithm can use its experience and performance measure to learn to better perform the given task. In [24] 4 procedural steps are given to develop a machine learning program. The first step is to choose the training, the second step is picking the type of target function. Next one has to choose a model to approximate this target function and in the last step the training takes place to optimise the model to perform the task on the training data.

Suppose we have chosen a task that we want to perform, as example we will stick to determining whether an animal is cat or dog. Another example we will often use is training a machine to play chess. The first step described in [24] is to determine the experience we want to learn on. This could be for instance pictures of cats and dogs but might also be lists of samples for which the values of certain features is stored. Furthermore there is a choice of using labelled data or not. When working with classification the type of labelling is obvious, these are the classes. There are cases when this is not so straightforward, for instance when learning to play chess. In this case the user has a wide variety of labels such as the best move, whether the game was eventually won or lost or give scores to the board state.

The next step is to determine the target function the algorithm will learn. This target function is used to perform the task, so for instance when classifying cats and dogs this could be a function that assigns a probability to an instance of being a cat or a dog. With classification the choice of target function is rather limited, but with learning to play chess there are some options for the target function. An example target function would be a function that takes a board state and outputs the current best move. Another target could be giving a score to the current state of the board. So picking the target function essentially boils down to picking what kind of output we want.

When the target function has been chosen, the next step is deciding which representation to use to approximate this target function with. This choice heavily depends on the task and the target function. The representation can be best seen as a class of functions, from which the algorithm can pick approximations. An example would be the class of linear functions $y = ax + b$, where the algorithm has to learn the values of a and b to best approximate the target function. This is the step where the machine learning method has to be chosen. In recent years neural networks have become the prime example of such a method. But there are many more such as, but not limited to k-means, decision trees and principle component analysis. In the case of classifying cats and dogs we could for example let the algorithm learn rules to check the input features and use these to classify. The last step is to pick the specific function from the representation that best approximates the target function. A machine learning method will consist of certain parameters which will influence the behaviour of the algorithm. The learning should make sure that the parameters are chosen such that the algorithm performs as well as possible and thus approximates the target function. In the case of a (linear) regression, the parameters are the values of the coefficients in the (linear) regression. Finding the best parameter values is done by computing the

performance via the loss function on the training data. The goal is to change the values such that the loss function is lowered. This can often be done by computing the gradient of the loss function with respect to the parameters. By using the gradient it can be determined on how to change the parameter values to lower the loss function. By changing the parameters, we get a new version of the algorithm. This will again be tested against the data and should perform slightly better. This repeated until the algorithm performs as desired. This phase is called the training of the algorithm.

3.2.1 Performance of the final algorithm

It is also important to have an idea of how well the algorithm performs on unseen data. Therefore machine learning often also has a testing phase. During this phase the algorithm is presented with data which it has not seen during the training. By computing the loss function on the unseen data, called the testing data, the testing error can be computed. This error gives an estimate on the performance of unseen data. To be able to compute this testing error the available data $\mathcal{D}$ has to be split into training and testing data. There is no fixed rule on how big the two separate sets should be. By keeping the training set large, the algorithm should be able to learn more and thus perform better. But the pay off is that the results on the testing data become less certain.

Performing very well on the training set doesn't guarantee that the algorithm generalizes well. There can be many causes for this, for instance working with an unsuitable model but two causes are important to discuss namely *over-* and *underfitting*, see 3.1. Data will have some noise and therefore when the data comes from a certain relation, points will slightly deviate from this relation. Overfitting occurs when the algorithm is completely fitted to the training data and thus following the random perturbation as well. This can be the case when the data comes from a linear or quadratic relation, whereas the algorithm is able to learn any polynomial expression. If the training is continued long enough, the algorithm will eventually perfectly fit the data. Underfitting occurs when the capacity of the algorithm is too small. The capacity of an algorithm is the amount of different mappings it can learn [25]. Using the same example, the capacity of a model is too small when the data comes from polynomial of degree 10 whereas the algorithm is only capable of learning linear functions. It doesn't matter how long the model is trained, since it will never be able to approximate the fit well due to the small capacity.

3.2.2 Hyperparameters

Besides the parameters which are optimized in the training phase, there will often also be so called hyperparameters which are parameters which influence the performance but the algorithm is not allowed to change. For instance, when using a regression with polynomials it could be the maximum degree of the polynomials. Sometimes hyperparameters could possibly also be trained, but often these are fixed as hyperparameters to limit the search space or to make the algorithm simpler. Choosing the values of the hyperparameters is important and should be done with care. Determining the best hyperparameters is done through validation.

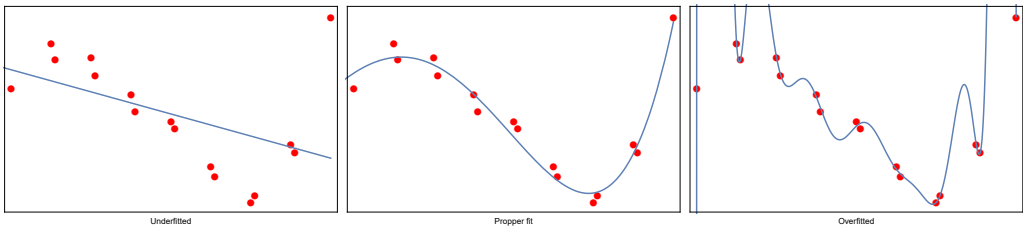


Figure 3.1 – Examples of over and underfitting. The data comes from a polynomial of degree 4 with some random noise. In graph on the left the data is approximated by a linear function, which does not capture the data properly. In the graph on the right the data is approximated by a polynomial of degree 14, which exactly goes through all points but which is not a good approximation of the data. In graph in the middle a polynomial of degree 4 is used to approximate the data, which clearly does follow the data properly.

Validating the algorithm is done by computing the loss function of the performance on some subset of the data.

To be able to compute a true testing error, the testing data cannot be used during the training and validation of the algorithm. By using the testing data to validate and pick the best hyperparameter values we essentially also use this data to learn. Therefore the data for the validation has to be taken from the training data, giving three sets of data used in the development and testing of a machine learning algorithm. To validate different hyperparameter values, the algorithm is trained on the training data and than the error is computed on the validation data. Doing this for different values of the hyperparameters gives a way to compare their predictive power. The hyperparameter settings for which the algorithm performs best on the validation data should be picked. We want to train the algorithm on as much data as possible and therefore it is best to than retrain the algorithm on the combined data form the training and validation set. In this way the algorithm can learn on as much data as possible. The validation process can be improved by doing cross validation [27], where the training data is split into K subsets of equal size. The algorithm than trains on $K-1$ subsets and can be validated on the remaining subset. This can than be done K times. The validation is than based on the average validation score across the K subsets.

3.3 Decision trees

In this section we will explain decision trees, which is the machine learning method that forms the basis of our algorithm. For this we will be using the implementation from Scikit learn [28]. This implementation is based on the algorithm developed by Breiman et all [29] in 1984, called Classification And Regression Trees (CART) . Other well known algorithms are C4.5 [30] (1993) and ID3 [31] (1986) both developed by Quinlan. A decision tree is a so called *white box* algorithm, since they are easy to interpret and their decision making process can be easily understood. The opposite to this are *black box* algorithms, for which it is (almost) impossible to follow the decision making process. An example for a black box algorithm are neural networks. Being able to interpret the algorithm is the main reason for using decision trees to learn the enhancement rules, as explained in Section 2.4.

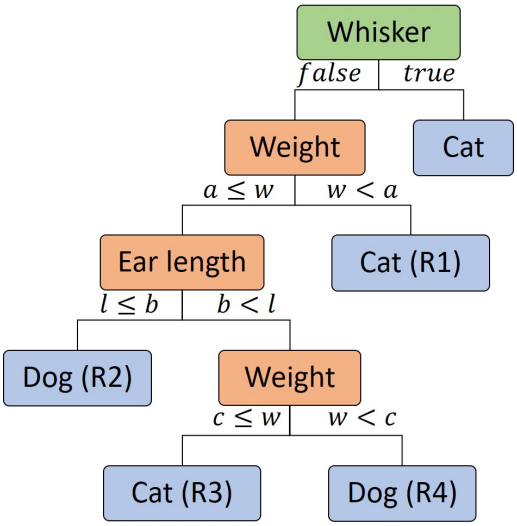


Figure 3.2 – An example of a decision tree based on classifying cats and dogs. The first node, the box indicated in green, is the root. This is where all the data comes in. The boxes in red correspond to nodes, which split the data based on the value of a specific feature. The blue boxes are the leaves, at which the objects are labelled. In this case, there were never any dogs with whiskers encountered.

A decision tree is a machine learning method which tries to mimic a human approach to making a decision. It can be represented by a tree like structure and used as a flowchart. To make a decision on a sample, the values of certain features are checked successively depending on the preceding checks. See Figure 3.2 for an example, which is based on classifying cats and dogs and we will use this throughout this section.

In this case, the first thing we might check is whether the animal has whiskers or not, if this is the case we conclude that the animal is a cat (since we have only ever seen cats with whiskers and no dogs with whiskers). If this is not the case we are still uncertain, so the next check could be the weight of the animal. Depending on the animals we have observed, when the animal is below a certain weight we again conclude that it is a cat. For the animals for which we are still uncertain, we can keep on asking questions and might even check the same features again at different values. In essence a decision tree tries to mimic this decision making process. Decision trees can be used both for classification and regression tasks. We will be using decision trees to classify and thus we will explain the algorithm for this task.

3.3.1 Elements of decision tree

A decision tree consists of nodes, leaves, branches and a root. The root of a decision tree is the first node and does not have any in going branches. This is the point at which all the input data is given and the first data split is made, in the case of the cat and dog classification this is the question if the animal has whiskers. The root is indicated by the green color in Figure 3.2. Nodes reassemble further questions, indicated by red in Figure 3.2, thus in our cat and dog example this is for example the question how much the animals weights. Nodes have one incoming branch and

at least two outgoing branches (in general nodes can have arbitrary number of outgoing branches). The algorithm we will use only has two outgoing branches, this is called a binary decision tree. The leaves are the points at which the tree stops and here we have thus classified our input, these are indicated by blue in Figure 3.2.

The algorithm makes such a structure of questions (called splits), based on the training data it has been given by finding features that separate the data as much as possible. We will use the classification of the cats and dogs as an example to explain the general algorithm. The training data is a set of points in some space $\mathbb{R}^n$, where n is the number of features. Taking the features from Table 3.1, we have $n = 4$. While the space in general is $\mathbb{R}^n$, the features do not have to be able to attain all values in $\mathbb{R}$. Some features might have discrete values while others can be defined on a specific interval only. Examples for this would be the animal having whiskers or not, which can be stored as an one or zero or the weight of the animal, which is defined on $\mathbb{R}_{>0}$.

3.3.2 How to split the data

The objective of a (classification) decision tree algorithm is to create nodes that split the data in such a way that the impurity of the outgoing sets decreases as much as possible. This means that they contain mainly one class of objects, thus in our case mainly cats or dogs. The splits are made by grouping the set according to the value of a certain feature. Thus as an example, see Figure 3.3, the cats and dogs are split by their weight, one set containing animals of weight less or equal to a and one set with weight above a. At the next node such a split is made again, based on the samples that reach this node.

Deciding on which feature and value to split is an essential part of the method and this is done by choosing the split that has the highest reduction in normalized cost. The reduction in normalized cost is defined as [27]

$$\Delta := \text{cost}(\mathcal{D}) - \left(\frac{|\mathcal{D}_L|}{|\mathcal{D}|}\text{cost}(\mathcal{D}_L) + \frac{|\mathcal{D}_R|}{|\mathcal{D}|}\text{cost}(\mathcal{D}_R) \right), \tag{3.3.1}$$

where $\mathcal{D}$ is the incoming data of the node and $\mathcal{D}_L$ and $\mathcal{D}_R$ are the two splitted sets. The cost function gives a measure on the impurity of a set. The impurity is a quantity on how mixed the set is with different types of objects. There exist different cost functions, the implementation we use for the decision trees is the CART algorithm [29] which uses the Gini index. Other types of cost functions are the *misclassification rate* and the *entropy* [27].

The Gini index is defined as

$$G = \sum_{c=1}^{C} p_c(1 - p_c), \tag{3.3.2}$$

where the sum is over all the different classes of samples in the set and p_c is the percentage of a certain class being present in the set. The Gini index gives the expected error of misclassification when classifying in the following way. Take a sample randomly from the set (with probability p_c) and randomly label it according to the distribution of the classes in the set. The probability of a wrong

classification is $1 - p_c$. In our algorithm the Gini index is used as the cost function in equation 3.3.1 and the goal is to find data splits such that Δ is minimized. This is done by computing the value of Δ using the Gini index for different splits and finding the lowest value.

As an example of the Gini index we look at the set of all observations in Figure 3.3, which we will call R. The objective is to split this set into two subsets, such that Δ in equation 3.3.1 is maximal. In the given example, the first split is made on the weight at value a. This makes the subset R_1 pure ($G = 0$) and the subset $R_2 \cup R_3 \cup R_4$ has Gini index $G = 0.38$. The Gini index of R is

$$G_R = \frac{16}{25} \cdot \frac{9}{25} + \frac{9}{25} \cdot \frac{16}{25} = 0.461$$

.
This gives

$$\Delta = 0.461 - (0 + \frac{15}{25} \cdot 0.38) = 0.237,$$

which is the lowest value of Δ on R.

If we would have split the data for instance on the value b for ear length, this would give

$$\Delta = 0.461 - \left(\frac{9}{25} \cdot 2 \cdot \frac{4}{9} \cdot \frac{5}{9} + \frac{16}{25} \cdot 2 \cdot \frac{12}{16} \cdot \frac{4}{16} \right) = 0.461 - 0.418 = 0,043.$$

This gives a much smaller reduction of cost. In a real decision tree, the algorithm computes the cost reduction for all splits and then selects the best split by picking the highest cost reduction.

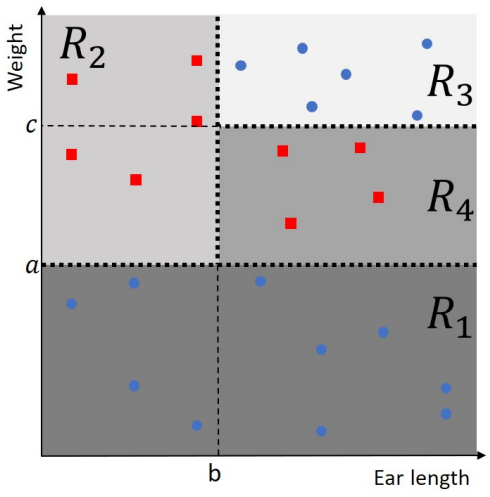


Figure 3.3 – An example of how data is split using the Gini index with a decision tree. The color correspond to a specific label.

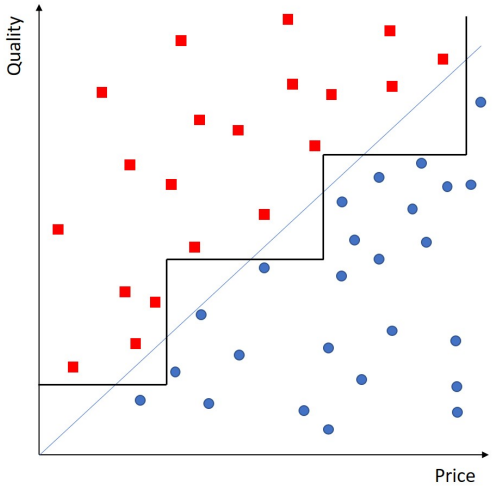


Figure 3.4 – An example of data for which a decision tree finds a correct way to split the data, but this is rather cumbersome. A diagonal boundary between the data is much more optimal. The color correspond to a specific label.

3.3.3 Greedyness, sub-optimal solution

The decision tree algorithm we are considering is called a greedy algorithm, this means that at every node the algorithm will try to decrease the current impurity as much as possible but it will not look at future nodes. Therefore, the solution that we might find can be a local optimal solution but not the best fit for the data. The construction of the best binary tree with the minimal number of questions is a NP-complete problem [32]. Since NP-complete problems become unsolvable on large amount of data, using a greedy algorithm is preferred even though this can lead to sub-optimal solutions. For the algorithm presented in the next chapter, this does not seem to be a problem due to the small decision trees we use.

3.3.4 Obliqueness

Another important thing about decision trees is the fact that in general they only look at the values of the individual input features. Such a decision tree is called univariate. This means that the algorithm will not make splits based on comparing values. This short coming is very important for our work and therefore we shall explain it in more detail. Lets say we want to buy a new computer and we are only interested in how much it costs and the quality of the processor. We are only willing to buy a computer of a certain quality processor if it is of certain price and we are willing to pay more if the quality is higher, see Figure 3.4.. Thus the computers that we classify as *want to buy* and those which we classify as *do not want to buy* are dependent on the ratio between the two features and not on either of the individual features.

The regular CART algorithm only looks at single features to make a split and is therefore not capable of directly learning such a ratio. It will create a large tree with nodes that alternate their selection criterion, see Figure 3.4. If the algorithm were capable of comparing two features, then it could directly learn than the ratio should be above some threshold value. There are extensions to decision tree algorithms which make them multivariate, meaning that they do look at the multiple input values. A way to make decision trees learn these ratios is by allowing them to learn on linear combinations of the features [29].

3.3.5 Overfitting

Furthermore the final tree is highly dependent on the data we feed the algorithm. Decision trees are very prone to overfitting, due to the fact that splits will be added until all data is split into sets of separate classes. To solve this it is also important to have some stopping criteria on when to stop growing the tree. One method to prevent this is to limit the size of the tree. Another method is called pruning, which is also part of the CART algorithm. In pruning the tree is first completely grown after which some nodes and their sub-nodes and leaves are replaced by one leave, which labels according to the majority vote. For the details see [29].

Since decision trees can easily overfit and are very much influenced by the data presented there exists an extension called decision forests. This classifier then consists of many decision trees, all trained on different subsets of the data. The combined classification of all trees than gives a true classification. In this way,

the randomness of the decision tree can be dealt with. A similar approach is used in our algorithm, as will be presented in Section 4.2.

3.4 Positive Unlabeled learning

Decision trees are the basis for our algorithm, since their decision making process is interpretable. However the data on which we want to learn is not suitable for any of the three main machine learning techniques presented in Section 3.1. To understand this, we first have to explain the data and how it is represented.

As discussed in the introduction and chapter 2 we want to learn the rules for the enhancements of the Deligne splitting. We want to do this by the data from the observed classifications of the degenerations for Calabi–Yau manifolds. Despite having around 20000 Calabi–Yau manifolds, the data set for a single enhancement will always be of the order $h^{1,1}{}^2$. This is due to the fact that for a given Y_3 there are only $4h^{1,1}$ possible Deligne splittings when working in the Kähler moduli space. Furthermore, the data is discrete since the classification of the degeneracies is done by two numbers. So, when working with all the single enhancements chains, all the data can be stored in as an array of length 4, where the entries have the type and value of the two degeneracies. For the double enhancements we can, in a similar fashion store the data in a 6 dimensional array. This means that our data is discrete, finite and all possible data points (not taking into account if they can occur) are known.

The reason why conventional machine learning is not possible is two fold. The first problem comes from the fact that we are only able to tell which points are certainly allowed, since these are the points that we observe in our data. We do not know anything for certain about the points that we do not observe and therefore we do not have any data on enhancements that are not allowed. This means that we do not have negative training data and therefore traditional supervised machine learning cannot learn from this data. Unfortunately also unsupervised learning is not able to discover some structure in the observed data points, since all the points lie on a lattice, with precisely the same distance between them. So the data has a lot of structure and the points do not group depending on whether they are allowed or not.

To overcome these problems, we might naively argue that not observing a combination means that this not allowed especially since we check so many Calabi–Yau manifolds compared to the number of possible enhancements. However, if we look at the single enhancements from IV to IV, we never observe $IV_1 \to IV_1$ but this is a possible enhancement by the rules in [6]. This shows that this assumption does not hold and something else is needed.

There is a sub class of supervised machine learning that deals with these kind of problems, which is called Positive Unlabelled (PU) machine learning [33]. In PU learning we only have points which are labelled positive and no points that are labelled as negative but there are a lot of unlabelled points available. There can be several reasons why only positive data is present, for instance because it is much more difficult or expensive to label a point as negative.

An example is linking a gene with a disease. It is much easier to label a gene as a predictor or cause for a certain disease than excluding a certain gene from having an effect. Another example is a simplified version of recommendations of movies or series on Netflix or youtube. These sites only know what you have watched but they don't know anything on what you have not watched. It can than be assumed that when you have watched something you like it. However not having watched something doesn't mean that you do not like it, you simply have not watched it yet.

PU machine learning is based on this type of data and uses the unlabelled data together with the positive data to still perform the task. This type of data is exactly the data we deal with, since we can only label the observed enhancements as possible.

3.4.1 PU learning methods

There are several ways to perform PU learning, which can be used for different learning problems. The first naive approach is to just take all the unlabelled points as negative and use a supervised machine learning algorithm, which is an approach taken by Neelakantan, Roth, and McCallum 2015 [34]. Another approach would be to only train on the positive samples and use this to rank the unlabelled samples by how similar they are to the positive samples [35]. These approaches are studied and used but more sophisticated methods exist and these approach are not truely seen as PU algorithms. The PU algorithms can be divided into three different types: two-step techniques, biased learning and class prior incorporation [33]. Since we use the first two types, we will not discuss the third type.

3.4.2 Biased PU learning

In biased PU learning, the unlabelled data is treated as mainly truly negative with some hidden positive points, which are seen as noise. There are several approaches on how to deal with these contaminated unlabelled data, as discussed in [33]. The method used in this thesis is based on [35], which uses bootstrapping create different training sets. These training sets contain all the positive data points and a subset of the unlabelled points, which are randomly sampled and regarded as negative data. On every training set a single classifier is trained and this classifier than classifies the unseen data points from the unlabelled set to give these points a score. By using enough pairs of training sets and classifiers, all points from the unlabelled set will have been scored. After this has been done, the scores are aggregated and a final label is given to the unlabelled data. This method is based on the bagging (bootstrap aggregating) procedure as suggested by Breiman [36]. The bagging procedure and how exactly it is implemented in this thesis, will be explained in Section 4.2.

3.4.3 Two-step

The two-step techniques are based on the idea of first finding reliable negatives and then applying a machine learning algorithm to the positive data and the

reliable negatives and possibly the unlabelled data [33]. There are many different methods for both steps, as discussed in [37] where the focus lies on text classification. Our method is based on the Positive examples and Negative examples Labeling Heuristic (PNLH) [38]. This method essentially takes the first step further by also searching for reliable positives. In the PNLH method, first the reliable negatives are found and with this information the reliable positives our found. In our used method we do this in the same step, but it is essential to find reliable positives. In the second step, a regular classification algorithm is trained on the positive data and the reliable negatives and positives. For the first step we use the bagging algorithm and for the second step we use a regular decision tree.

Chapter 4

Algorithm

In this chapter we will discuss the developed algorithm used to learn the double enhancement rules. First we will discuss the data on which the algorithm will be applied, this is done in Section 4.1. In Section 4.2 we will discuss the basis and main part of the algorithm, which is the bagging procedure based on the methods explained in Section 3.4. During the development of the algorithm, we discovered that the basis, main algorithm is not always able to perform correctly. To overcome this we implemented two extensions of the algorithm, these extensions are discussed in Section 4.3. In the last part of this chapter, a validation procedure of the algorithm will be presented. The validation is based on the single enhancements since we know the true, exact rules for these enhancements.

4.1 The data

The enhancement rules are learned from Calabi–Yau manifolds, which are constructed by the methods from [4] and [39]. With the intersection numbers of these Calabi–Yau manifolds, we can determine the Deligne splitting by the calculations presented in 2.3.1 for the different degeneracies. This gives us the graph like structures as presented in [6] from which we can read of the single and double enhancements.

The goal is to learn the rules for the double enhancements. For our discussion it is useful to use some slightly different language than in [6]. By the type of an enhancement we mean the value of the Roman number, thus I, II, III or IV and with the value we mean the value of the numbers a, b, c and d in I_a, II_b, III_c and IV_d. This distinction is important since the rules for the allowed enhancements can be grouped per combination of types. To simplify the interpretation of the decision trees, it is therefore also good to apply the algorithm to every possible combination of types separately. As a general notation we denote a single enhancement chain by

$$\texttt{Type}_a \to \texttt{Type}_b.$$

The $\texttt{Type}_a$ degeneration corresponds to sending $t^{\mathcal{I}_a} \to i\infty$ with the ordered index set $\mathcal{I}_a = (i_1, ..., i_{n_a})$ which specifies the growth sector [6] and similarly $t^{\mathcal{I}_b} \to i\infty$ with the growth sector $\mathcal{I}_b = (i_1, ..., i_{n_a}, i_{n_a+1})$, which thus contains one extra coordinate. In a similar fashion the double enhancements are given by

$$\texttt{Type}_a + \texttt{Type}_b \to \texttt{Type}_c, \tag{4.1.1}$$

where we have a ordered index set $\mathcal{I}_a = (i_1, ..., i_{n_a})$ and $\mathcal{I}_b = (i_1, ..., i_{n_b})$ which specify the growth sector and $\mathcal{I}_a \backslash \mathcal{I}_b = \{i_x\}$ and $\mathcal{I}_b \backslash \mathcal{I}_a = \{i_y\}$ with $i_x \neq i_y$. Furthermore $\mathcal{I}_c = \mathcal{I}_a \cup \mathcal{I}_b$.

For every allowed combination of types (meaning the roman numbers will never decrease) in an enhancement, we will perform our analysis by the algorithm. The input data for the algorithm are the observed combinations of values. So for instance when learning the rules for IV $\rightarrow$ IV the specific enhancement $\text{IV}_1 \rightarrow \text{IV}_3$ is stored as a two dimensional array $[1, 3]$ or in general as $[a, b]$. For the double enhancements the data is stored as $[a, b, c]$.
Learning on specific combinations of types would mean for the single enhancements that we perform the algorithm 10 times on the data corresponding to the types. However, since we never encounter any type I except for when no coordinate is send to infinity we are not able to learn any of the rules regarding this type. The same holds for the enhancements from II to II, which also do no occur in Calabi–Yau manifolds. In the end, the algorithm is only run on 5 combinations of types.

4.2 The bagging algorithm

To deal with the PU data a bootstrap aggregating (bagging) algorithm using decision trees has been implemented. As the name suggests, this algorithm is based on a bootstrap procedure. This essentially means that the algorithm will use random sampling with replacement to improve the performance of the algorithm. Bootstrapping is a common method in statistics when working with random variables from an unknown distribution. By using a set of samples of the random variable and sampling from this set a normal distribution is recovered and in this way the methods and tools known for a normal distribution can be used.

The algorithm has three steps which are repeated a number of times. The first step is to make a training set. The second step is training a decision tree on the training set and the third and final step is to classify points which are not in the training set.

To perform the training of a classification decision tree, the training data needs data point from the different classes. In the case of enhancements these are the allowed (positively labeled) and not allowed enhancements (negatively labeled). Since we can only observe the allowed enhancements, there is no natural way to get negatively labeled data. The set of all observed points are called $\mathcal{P}$ and the set of unobserved points are called $\mathcal{U}$ In the first step a training set X_t is created. This is done by bootstrapping a subset $\mathcal{U}_t$ of size b from the set $\mathcal{U}$. The points in $\mathcal{U}_t$ are labelled as negative and added to the training set. The points in $\mathcal{P}$ are obviously allowed and therefore labelled as positive and added to the training set X_t, thus $X_t = \mathcal{P} \cup \mathcal{U}_t$. We will refer to the points in $\mathcal{U}_t$ as the bagged or bootstrapped elements.

In the next step, a decision tree f_t is trained on the training data X_t. It is important to note that only the labels for $\mathcal{P}$ are completely certain. There might be points in $\mathcal{X}_t$ which are hidden positives, meaning that their current label for the training is incorrect. In the last step the points which were not in the training

set, thus the points in $\mathcal{U}\backslash\mathcal{U}_t$, are classified by f_t. The classification as done by the decision tree of the current time step is stored.

These three steps are repeated N times. Since the samples in $\mathcal{U}_t$ are selected at random for every iteration, the trained decision tree for every iteration step t can be different. After N iterations, the average classification for all points in $\mathcal{U}$ are computed and this gives a probabilistic classification for their true label. After the N iterations a last decision tree is trained on all the data, thus on the sets $\mathcal{P}$ and $\mathcal{U}$. The points in $\mathcal{U}$ have been given a label according to the probabilistic classification, in which the label is the one with the highest probability. This last decision tree is then used to distill the enhancement rules.

Algorithm 1: PU bagging algorithm

Input : $\mathbf{T}_X$: data of observed points
$\mathbf{Y}$: data of points to be classified
$\mathbf{L}_x$: labels of observed points
N_{bag}: number of samples taken from $\mathbf{Y}$
N_{run}: number of times classification is done

Output : $\mathbf{C}$: predicted classification on $\mathbf{Y}$

Create $\mathbf{A}$ used to store the predicted labels per element of $\mathbf{Y}$

for *i = 1* **to** N_{run}**:**
  Create T_Y to store samples from $\mathbf{Y}$ used in the training process.
  for *j = 1* **to** N_{bag}**:**
    Pick a random $y \in \mathbf{Y}$. Add y to T_Y
  end
  $\mathbf{L}_Y$:= Array of N_{bag} false labels
  $\mathbf{T} := \mathbf{T}_X \cup \mathbf{T}_Y$
  $\mathbf{L} := \mathbf{L}_X \cup \mathbf{L}_Y$
  $\mathbf{V} := \mathbf{T}_X \backslash \mathbf{T}_Y$
  Train a decision tree on $\mathbf{T}$ and $\mathbf{L}$.
  Predict the labels of the elements of $\mathbf{V}$ with the decision tree and store in $\mathbf{A}$.
end

Compute the average labels of every point in $\mathbf{Y}$ and store in $\mathbf{C}$

4.2.1 The performance of the bagging procedure

To check the learning abilities of the algorithm we first tried to learn the rules for two examples, the enhancements from IV to IV and the enhancement from II to III on data obtained from Calabi–Yau manifolds with $h^{1,1}$ upto 5. However, by applying the algorithm directly to the data we encountered a problem.

As discussed in Section 3.3.4, decision trees are univariate algorithms and this means that only they learn splitting criteria which are parallel to the axes. If we take the known enhancement rules from [6], then we can see that these rules depend on the relative values between the two limits. So for the single enhancement case, learning rules parallel to the axes will make learning more difficult and also the resulting decision tree will be much harder to interpret. When applying the algorithm directly to the data of the form $[a, b]$ the final classification of the

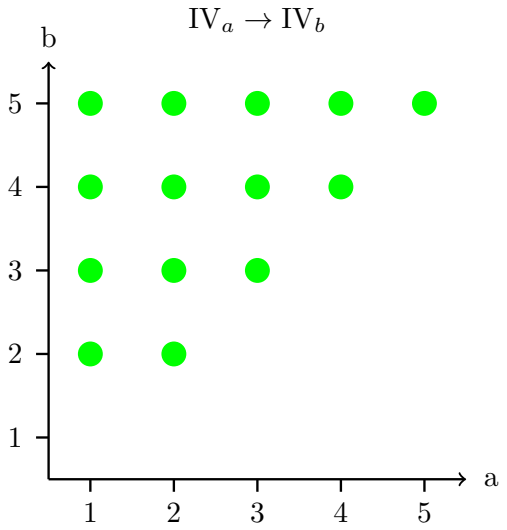


Figure 4.1 – The observations of the enhancement $\mathrm{IV}_a \to \mathrm{IV}_b$ for the Kreuzer Skarke database [4] for $h^{1,1} \leq 5$.

points from $\mathcal{U}$ is prone to mistakes. This makes the algorithm unable to learn the correct enhancement rules.

There are several ways to overcome this problem. One might use so called multivariate or oblique decision trees. However these algorithms are much more difficult to make, have longer computation time and are more difficult to interpret. Furthermore, there are only a very few readily available implementations of such decision tree algorithms and these are often not very good or insightful.

For now we will change the data based on the known rules to make the algorithm perform well. Working with the example of the enhancement $\mathrm{IV}_a \to \mathrm{IV}_b$, we change the input vectors from $[a, b]$ to $[b - a, a + b]$. This gives a much better performance and makes the decision trees much smaller. Thus in general we could perform an operation on the input vector, which essentially is a basis transformation to boost the learning process. Since we do not know the rules for the double enhancements we also do not know which basis transformation we should use. Therefore we first have to find the best basis transformation and then start the learning process. To find the best basis transformation, we developed an adaptation of the already proposed PU algorithm, which is presented in Section 4.3. This algorithm we will call the feature selector and it is again an iterative algorithm.

Example 1

For all the CYs from the KS database upto $m = 5$, we took the observations of the enhancements of $\mathrm{IV}_a \to \mathrm{IV}_b$ which are presented in Figure 4.1 by a green dot. So the only not observed, but allowed enhancement is $\mathrm{IV}_1 \to \mathrm{IV}_1$ thus this combination is part of the set $\mathcal{U}$. In this first example we will show that the algorithm is capable of classifying all other value combinations as not possible. The algorithm has two hyperparameters: the number of iterations and the number of samples being bootstrapped. In this example, there are 11 unlabelled points. In the case that we always bootstrap 10 points, we will always have at least 1 point being classified per iteration. Assuming that only one element is classified (due to

sampling with replacement, an element might be sampled twice in one iteration), we need to run the algorithm 110 times to have a lower bound of on average 10 classifications per unobserved sample. For this example we decided to have 1100 iterations. This gives a lower bound on the average number of classifications per sample of 100 but still has a low computation time of only a few seconds, enabling us to test different parameter values and settings.

The results are presented in Table 4.1, where we can clearly see that for this case increasing the number of bootstrapped elements increases the accuracy of the algorithm. This can be explained by the fact that for this case only one unlabelled sample is still possible. So bootstrapping all other elements and thus labelling them as not possible indeed gives them, for this specific enhancement, the correct label. Since in general we do not know the true label, it might be beneficial to have some more fluctuations in the model. In Section 4.4 we will discuss the best values for the hyperparameters.

	$n = 1$	$n = 5$	$n = 10$
$\text{IV}_1 \to \text{IV}_1$	1.000	1.000	1.000
$\text{IV}_2 \to \text{IV}_1$	0.395	0.017	0.000
$\text{IV}_3 \to \text{IV}_1$	0.099	0.000	0.000
$\text{IV}_3 \to \text{IV}_2$	0.398	0.019	0.000
$\text{IV}_4 \to \text{IV}_1$	0.098	0.000	0.000
$\text{IV}_4 \to \text{IV}_2$	0.100	0.000	0.000
$\text{IV}_4 \to \text{IV}_3$	0.399	0.019	0.000
$\text{IV}_5 \to \text{IV}_1$	0.100	0.000	0.000
$\text{IV}_5 \to \text{IV}_2$	0.098	0.000	0.000
$\text{IV}_5 \to \text{IV}_3$	0.099	0.000	0.009
$\text{IV}_5 \to \text{IV}_4$	0.395	0.018	0.010

Table 4.1 – Probability of the enhancement from IV to IV being allowed computed by the bagging algorithm with decision trees. n is the number of elements being bootstrapped.

Example 2

In this second example, we will look at the enhancements II $\to$ III where the training data again comes from Calabi Yau manifolds with $h^{1,1} \leq 5$. There are several differences compared to the previous example. First of all, there are less allowed enhancements (only 9). Secondly we also observe a smaller portion, namely just 6 of the 9 allowed enhancements. Lastly, we have a slightly different enhancement rule since it is dependent on the value of the type II and not only on the difference between the values. To show the need of using some kind of multivariate decision tree, the results of the algorithm are presented for different basis transformations. The number of bootstrapped samples is 10 for all different inputs. The results for the algorithm are presented in Table 4.2. The enhancement patters are stored as $[a, b]$ before the basis transformation. The different transformations are also given in the table, note that we have also performed the algorithm on a over determined system (4 input features).

		$[a-b, a+b]$	$[a, b]$	$[a, b, a-b, a+b]$	$[a-b, a]$
$II_0 \to III_0$	not possible	0.104	0.011	0.12	0.139
$II_0 \to III_1$	not possible	0.002	0.033	0.031	0.0
$II_0 \to III_2$	not possible	0.002	0.078	0.034	0.0
$II_0 \to III_3$	not possible	0.035	0.002	0.0	0.0
$II_1 \to III_0$	not possible	**0.889**	0.012	0.327	0.373
$II_1 \to III_1$	not possible	**0.576**	0.033	0.168	0.141
$II_1 \to III_2$	not possible	0.146	0.076	0.04	0.0
$II_1 \to III_3$	not possible	**0.494**	0.002	0.0	0.0
$II_2 \to III_3$	possible	**0.617**	0.523	0.318	0.975
$II_3 \to III_0$	not possible	0.145	0.989	0.343	0.445
$II_3 \to III_3$	possible	0.971	0.367	0.464	1.0
$II_4 \to III_0$	not possible	0.190	0.225	0.222	0.199
$II_4 \to III_1$	not possible	0.182	0.862	0.272	0.198
$II_4 \to III_3$	possible	0.984	0.285	0.526	1.0

Table 4.2 – Probability of the enhancement from *II* to *III* being allowed computed by the bagging algorithm with decision trees.

From the results in Table 4.2 it is clear that the algorithm needs the correct transformation to correctly classify, but it is still not able to identify all points correctly. For some samples the probability of it being positive or negative is around 0.5, which gives a very uncertain result. Therefore another extension, besides the feature selector has to be implemented to give all the points a label with a high certainty.

To obtain the rules for the double enhancement, we will need 10 different decision trees, one for each combination of types. Since we fix the type before training, we only have to store the values of the different instances. For the double enhancements this means that we can store a point as a 3 dimensional vector and for the single enhancements we could store the points as a 2 dimensional vector. If we look at for example the single enhancement $IV_a \to IV_b$, we can store the input as $[a, b]$ where a is the value of the begin node and b the value of the end node. For the double enhancement, we can thus similarly store the input as $[a, b, c]$.

4.3 Extension of the algorithm

We will now discuss two extensions of the algorithm to boost its learning. The first extension is used to prepare the data and find the basis transformation to enable the best learning. The second extension is to have more points from the unlabeled data get a label with a high certainty.

4.3.1 Feature selector

The preparation step is to choose which basis transformations we want to test. This can be any computation on different elements from the input vector. The current algorithm is made in such a way that the transformations have to be given by the user.
These computations are then executed on the input vectors, creating new vectors

which can be used as input for a decision tree learning algorithm. The elements of these vectors are called features and the vectors will in general be of higher dimension than the original input vectors. An example transformation would be

$$a, b \to [a, b, b - a, a - b, b + a].$$

After having applied the transformation on the data, we want to find the features that predict the labels the most. In regular supervised machine learning this can be done with various algorithms from [28]. However we are dealing with PU-learning and thus we do not have negatively labeled data. Furthermore, these algorithms are often used on high dimensional data sets. For machine learning purposes our data is not high dimensional and we even have very few data points. This makes the conventional methods not suitable for our data. Therefore we use an algorithm, which is highly based on the bagging idea. We call the algorithm the feature selector. It performs many iterations and each iteration consists of three steps.

The first step of the iteration is to perform the bagging procedure, similar to the first step described in 4.2. Next, a decision tree is trained on the created training data also similar to the second step of the bagging algorithm. The last step is different, the algorithm doesn't classify the unused points but it checks which features from the transformed input data the decision tree has used to split the data. This data is stored at every iteration step. After all the iterations, the algorithm computes how many times every feature (computation on the data) has been used to split the data.

We classify the features by how many times they are used as a selection criterion, where the best feature is the feature which has been used the most as split. There are two options to select the best features to learn on. Either the M most chosen features can be selected directly from performing the procedure once or only the most chosen feature is selected and the whole feature selector algorithm is run again, but without this one feature. This last procedure is the one implemented for this code, since in the test phase we observed that in most cases there is only a big difference between the best feature and the rest, while the other features are chosen approximately the same amount of times. By removing the most chosen feature this changes and again one feature is clearly the most chosen. See algorithm 2 for pseudo-code on the feature selector.

Algorithm 2: Feature selection algorithm

Input : $\mathbf{T}_X$: data of observed points
$\mathbf{Y}$: data of unobserved points
$\mathbf{L}_x$: labels of observed points
N_{bag}: number of samples taken from $\mathbf{Y}$
N_{run}: number of times classification is done
$\mathbf{F}$: list of manipulations.

Output : $\mathbf{S}$: scoring of features to split upon

Create $\mathbf{F}$ used to store the scoring on number of times feature was used as splitting criteria.
Apply the manipulations of $\mathbf{F}$ to $\mathbf{T}_X$ and $\mathbf{Y}$ to create all the features used in training.

for $i = 1$ **to** N_{run}**:**
    Create T_Y to store samples from $\mathbf{Y}$ used in the training process.
    for $j = 1$ **to** N_{bag}**:**
        Pick a random $y \in \mathbf{Y}$. Add y to T_Y
    end
    $\mathbf{L}_Y$:= Array of N_{bag} false labels
    $\mathbf{T} := \mathbf{T}_X \cup \mathbf{T}_Y$
    $\mathbf{L} := \mathbf{L}_X \cup \mathbf{L}_Y$
    Train a decision tree on $\mathbf{T}$ and $\mathbf{L}$.
    Store the used features for the splits of the decision tree in $\mathbf{F}$.
end

Determine how often each feature stored in $\mathbf{F}$ was used, output this score in $\mathbf{S}$

4.3.2 Adding points

As has become clear from the results presented in the previous section, the algorithm is not yet capable of finding all the allowed and not allowed enhancements with very high certainty (see Table 4.2. However, for some unobserved samples it is very certain, meaning that it (almost) always gives them the same classification in every round of the bagging procedure. These very certain points can be used to make the algorithm more robust in the following way. After the probabilities for all unobserved samples are computed, the samples for which this probability is above some predefined threshold to be a certain class are added to the list of observed samples. This means that the samples are added to the set $\mathcal{P}$ and removed from the set $\mathcal{U}$. In this way, they are not taken into account for the bootstrap procedure. This gives less fluctuations and with the assumption this previously computed probability is correct, the algorithm should be better able to classify correctly. From this point on the set $\mathcal{P}$ does not contain only points which are for sure positive, it might contain negatively labeled points and extra positive points which are estimated labels. See algorithm 3 for pseudo-code on this algorithm.

Adding this extension to the algorithm and keeping the iteration going until all samples have a probability above the threshold value gives for both examples the correct classification to all samples and gives easily interpretable decision trees

which give the same rules as presented in [6].

Algorithm 3: Adding points to classified data

Input : $\mathbf{Y}$: data of unclassified points
$\mathbf{X}_0$: data of classified points
$\mathbf{C}$: predicted probability of class on $\mathbf{Y}$
t: threshold value to add points to classified data

Output : $\mathbf{X}$: updated classified points

for $y \in Y$**:**
  Get probability p_y of point y to be *True* from $\mathbf{C}$.
  if $p_y > t$**:**
    Add y to $\mathbf{X}$ with label *True*.
    Remove y from $\mathbf{Y}$
  end
  elif $p_y < 1 - t$**:**
    Add y to $\mathbf{X}$ with label *False*.
    Remove y from $\mathbf{Y}$
  end
end

4.4 Validating the algorithm

The disadvantage of using machine learning is that we can not present results with hundred percent certainty. Therefore it is important to validate and test the algorithm in order to find the best values for the hyperparameters and to get a measure on how well the algorithm performs. In the development of the algorithm we have worked with two enhancements for which the rules are known, from II to III and from IV to IV. These two enhancements are also used to determine the values for the hyperparameters. With these values, we can test the algorithm on the other known enhancements and thus getting a performance measure. In this section we will discuss the validation process and the final choice of hyperparameters.

Since we are not interested in the classification of individual data point but in the classification rules, we do not want to be correct on most data points but on all. Therefore we have to validate our algorithm slightly differently compared to a regular machine learning algorithm. In a regular validation procedure, the algorithm will be trained with different settings of hyperparameters on a subset of the input data and validated on the rest of the input data. The setting of hyperparameters with the best validation score will be picked as the final algorithm. Another reason why we need a different validation procedure is that our sample space is finite and very small compared to usual machine learning standards. We know all the points that are an option and we only need to label whether this is option is indeed possible. For the algorithm to learn, it needs all the options, since the bagging procedure makes use of the unlabeled points. Because of these two reasons, the algorithm cannot simply be validated by taking points out of sample set and used as a validation as is usually done in machine learning.

There are several methods we can use as validation. The most straightforward way is to use the observed data to check if the algorithm can find the rules for the single enhancements with this data. This procedure tells us something about the capacity of the algorithm but it also tells something about whether there is enough observed data to find the correct rules. Even a perfect algorithm, capable of finding the correct rules will need enough data to find the rules. So while this might give some interesting results, if the algorithm cannot find the correct rules it is hard to determine the cause.

As an example, when looking at the data obtained from CYs with $h^{1,1} \leq 5$ for the enhancements from III to III, we never encounter an enhancement from III_a (with $a = 0, 1, 2, 3$) to III_3, which is however an allowed enhancement according to [11]. The algorithm is likely to end up with rules such that it is never allowed to enhance to III_3, since it never encounters such enhancement. If we however add only one enhancement to III_3, lets say $\text{III}_1 \to \text{III}_3$. The algorithm always finds the rule from [11]. Thus validating only with the observed points might teach us more about our observations and data than about the predictive power of the algorithm. CYs might not show all allowed enhancements [7]. Of course, to justify our results we also have to know the quality of the data set.

Therefore we use a slightly different method. By using the fact that we know the exact rules for the single enhancements, we can create multiple, artificial datasets. This means that we pick two types, for example II and III and compute all allowed enhancements from II to III. This gives the complete data and we can take a random subset of the allowed points as training input. On these artificial datasets, we can apply the algorithm and check if it gives the desired result. Using many different artificial datasets with varying size, we can check how much data the algorithm needs in order to learn the correct rules. We can than also check the settings for which the algorithm has the highest probability to give the correct result. We believe that this this is the best way to validate the algorithm.

4.4.1 Validation procedure

The following hyperparameters are fined-tuned by the different validation approaches.

- threshold value
- runs feature selector
- runs labeling
- bagging size
- number of features to select

Validation one: threshold
The first test is done on the enhancements from II to III and IV to IV, cross validated over different input sets of varying size. For II to III these are sets of $6, 7$ or 8 elements and for IV to IV $9, 10, 11, 12$ and 13 elements. For every input size we have created 50 random training sets. For the enhancement from II to III there are 20 possibilities for which 9 are allowed and for the enhancement from

threshold	6	7	8	avg
60	36.96	76.88	145.92	86.6
70	89.76	124.36	176.96	130.4
75	95.88	133.0	188.32	139.1
80	98.52	138.68	217.04	151.4
85	100.44	137.96	241.08	159.8
90	99.8	136.64	245.28	160.6
95	72.68	135.64	247.04	151.8

Table 4.3 – The number of perfect classifications for the enhancement from II to III for 50 sets per percentage of input data with 5 tests per input set.

IV to IV there are 25 possibilities of which 15 are allowed.

The best values for number of runs for both the feature selector as the labeling procedure are determined by testing the algorithm against random input sets of varying size. The number of runs are chosen from the following values $[50, 100, 200, 500, 1000]$, where every combination for N_f and N_b is picked, giving 25 different combinations for the hyperparameters. For every combination, the algorithm is run on 50 different input sets and for every training set, the algorithm is run 5 times. This makes a total of 250 runs per combination of hyperparameters.

For the enhancement II to III, the tests where done on input sets of size $6, 7$ and 8. On sets with only 5 elements, the algorithm is in very rare cases also capable to learn the perfect classification. The results from this test suggest using a threshold value of 90 instead of 80 when looking at the best performing hyperparameters in Table 4.5. The average value in Table 4.3 also suggests that a threshold value of 85 of 90 gives a better result. Also a very high threshold value of 95 gives very good results when almost all data is known, but it performs worse when the algorithm is only presented with a small portion of the possible data.

For the enhancement from IV to IV, we have taken input sets of size 9 to 13. These results are less decisive, since almost all threshold values seem to perform approximately the same. There is only a small noticeable difference for the results with threshold value 95, which as with the enhancement II to III performs badly when presented with little information. This is the case for both enhancements and can be explained by the fact that the algorithm will never be very certain when presented with little data. Therefore, very few points will ever reach this level of certainty and thus most will never receive a label for a next round.

Validation two: number of runs
To determine the optimal number of runs, the number of runs is varied while keeping the other hyperparameters fixed. In Figures 4.2 and 4.3, the average number of perfect classifications per combination of N_b and N_f is given. The average is taken over all values for the threshold and input size. The maximum number of perfect classifications is 250. For the enhancement from II to III we can clearly see an improvement in the results when the algorithm is allowed to have more runs. This is expected but unfortunately not supported by the results for the enhancement from IV to IV. From Figure 4.2 it seems as if the algorithm performs slightly better when using $N_b = 500$. But the difference between the performance

threshold	9	10	11	12	13	avg
60	182.28	195.2	238.76	248.88	249.96	223
70	183.64	197.92	240.24	249.76	250.0	224.3
75	180.84	198.68	240.2	249.6	250.0	223.9
80	181.0	199.48	239.96	249.68	250.0	224
85	180.44	198.0	237.28	249.88	250.0	221.4
90	173.28	192.88	234.84	249.84	250.0	219.3
95	154.96	185.84	235.08	249.64	250.0	214.8

Table 4.4 – The number of perfect classifications for the enhancement from IV to IV for 50 sets per percentage of input data with 5 tests per input set.

for $N_b = 500$ and 1000 is very small and is most likely due to randomness. In Figure 4.3 there is no clear improvement which can also be explained besides random fluctuations in the performance. Essentially we think that the algorithm performs similarly regardless of the number of runs .

The idea of these random fluctuations is supported by the results presented in Table 4.6, where we see that the top 5 best performing algorithms perform slightly above average. This shows that these are just some random fluctuations. From Table 4.5, for the enhancement from II to III, we can conclude that for this enhancement the algorithm does perform better with certain hyperparameters. These results both support the idea that a threshold value of 85 of 90 is optimal and that the algorithm is more stable when increasing N_b and N_f. This can also very well be seen by the results in Figure 4.3, which are almost equal when $N_b = 1000$, showing that the algorithm has come to a stable solution. This means that the algorithm will give the same result every time the algorithm is run, which is the situation we are interested in. We do not want to find results which come from a random fluctuation. We want to get the same result every time. Of course increasing the number of runs doesn't keep improving the algorithm.

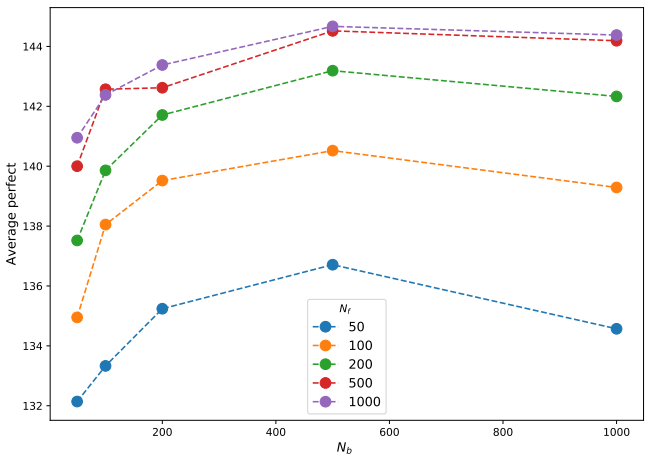


Figure 4.2 – Mean perfect score per N_f and N_b value for the enhancement II to III with $Th = 0.8$. The algorithm is run on input sets of 6,7 and 8 elements. For each size 50 random sets are created and for every set 5 tests are done. This makes 250 tests per size. The average over the size is taken.

N_f, N_b, Th	perfect
$(1000, 500, 85)$	163.7
$(500, 500, 90)$	164.0
$(1000, 500, 90)$	164.3
$(1000, 1000, 90)$	164.3
$(500, 1000, 90)$	165.3

Table 4.5 – The top 5 of perfect validations together with their mean validation score values on the enhancement from II to III with 50 fixed sets. For every set the algorithm was run 5 times.

N_f, N_b, Th	perfect
$(1000, 500, 70)$	224.3
$(200, 1000, 70)$	224.6
$(1000, 50, 80)$	225.0
$(1000, 50, 75)$	225.3
$(500, 100, 75)$	225.6

Table 4.6 – The top 5 of perfect validations together with their mean validation score values on the enhancement from IV to IV with 50 fixed sets. For every set the algorithm was run 5 times.

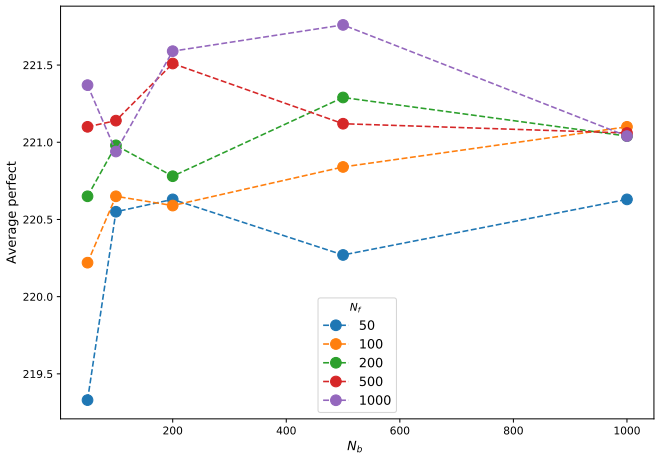


Figure 4.3 – Mean perfect score per N_f and N_b value for the enhancement IV to IV with $Th = 0.8$. The algorithm is run on input sets of 9,10,11,12 and 13 elements. For each size 50 random sets are created and for every set 5 tests are done. This makes 250 tests per size. The average over the size is taken.

Validation three: bagging size
To determine the best bagging size, the settings from the previous tests with the best scores are chosen. The bagging sized is varied and again tested against 50 random sets where each set is checked 5 times. The threshold value was taken to be 90. The results are not really clear to which amount is the best to pick as bagging size. When looking at Table 4.7 we see that the performance increases when increasing the bagging size but then suddenly drops when going to 11 data points. Similar behavior is seen for the enhancement from IV to IV, where the performance is much less when adding all but one element in the bagging procedure.

bagging size	perfect	mean score
1	167.3	0.921
3	171.7	0.917
5	175.3	0.918
7	189.7	0.929
9	188.7	0.938
11	102.3	0.829

Table 4.7 – The number of perfect classifications and mean validation score against the number of bagged elements for the enhancement from II to III. This was computed for 50 random sets per percentage of input data, each with 5 tests per set. The average scores were taken over the percentage of input data.

bagging size	perfect	mean score
1	169.5	0.905
3	205.125	0.945
5	218.875	0.961
7	224.75	0.97
9	223.875	0.969
11	215.5	0.96
13	159.0	0.774

Table 4.8 – The number of perfect classifications and mean validation score against the number of bagged elements for the enhancement from II to III. This was computed for 50 random sets per percentage of input data, each with 5 tests per set. The average scores were taken over the percentage of input data. The input data with 13 elements was not used.

Validation four: number of features
To determine the best number of features to select upon, the settings from the previous tests with the best scores are chosen. The number of features selected is varied and again tested against 50 random sets where each set is checked 5 times. The value for the threshold was taken as 90 (but using 80 gives very similar results). From Tables 4.9 and 4.10 it is clear that using only one feature makes it impossible for the algorithm to learn. As can be expected, using 2 features will give good results, but only slightly better than using 3 or 4 features to select. The difference between the results are big enough to make conclusive statements. To give more details, it might be insightful to not only give the average over the data size but look at the results per input size. Since taking more features makes the input overdetermined in a linear algebra sense, we could still argue against using more than needed for a new basis. In this test we have not looked at many different features, but only at taking the original values of the nodes and their sums and differences thus $[a, b, b-a, b+a]$.

No features	perfect	Mean score
1	0.0	0.61
2	165.7	0.92
3	163.3	0.92
4	147.3	0.89

Table 4.9 – The number of perfect classifications and the mean validation score per number of features to select for II to III with $Th = 0.9$, $N_f = 1000$ and $N_b = 1000$. 50 random sets each 5 tests per percentage of data available.

No features	perfect	Mean score
1	22.5	0.69
2	214.3	0.97
3	212.8	0.96
4	206.8	0.96

Table 4.10 – The number of perfect classifications and the mean validation score per number of features to select for IV to IV with $Th = 0.9$, $N_f = 1000$ and $N_b = 1000$. 50 random sets each 5 tests per percentage of data available.

Final hyperparameter choice
For the results presented in the next chapter we use the following settings:

- $N_b = 1000$,
- $N_f = 1000$,
- $Th = 0.9$,
- bagging size is three less than the total number of unlabelled points,
- number of features selected 2 for single enhancements, 3 for double enhancements.

With the results in this section we can also conclude that the algorithm is clearly capable of finding the correct rules if it presented with enough data.

Chapter 5

Results

In this chapter, we present and discuss the results from the algorithm as explained in Chapter 4. In Section 5.1 we present our findings on the single enhancements and compare those with the known rules from [6]. Furthermore we discuss the effect of the size of the observed data on the outcome of the algorithm. In Section 5.2 we present the results of the algorithm on the double enhancements. The data used in this chapter comes from the Kreuzer Skarke list [4] (upto $h^{1,1} = 5$) and the CICY [39] (upto $h^{1,1} = 7$).

5.1 Results single enhancement

The developed algorithm was run on the observed single enhancements, with the 1000 iterations in both the feature selection and the bagging / classification procedure. The threshold value to add points to the observed points was at 0.9. The features to select were $[a, b, b - a, b + a]$. The starting value for the minimum number of not bagged elements is 3. When less than 3 points are still not classified, this value decreases such that always at least one point is bagged. The algorithm was stopped when either all points where classified or after 10 full cycles.

The algorithm was trained on data of at most $h^{1,1} = 7$. For every value of $h^{1,1}$ all the data obtained from Calabi–Yau manifolds with lower value of $h^{1,1}$ was also used in the training, since we know that rules for the allowed enhancements do not depend on the value of $h^{1,1}$. Of course the allowed splitting types do depend on the value of $h^{1,1}$, but these only increase in numbers. This was important since especially for $h^{1,1} = 6$ and 7 the data came from the CICY, which are very few Y_3's compared to the Kreuzer Skarke CYs. In Table 5.1 the number of observed points, the number of value combinations and the number of allowed enhancements are given. By the value combinations, we mean two enhancements types and values which can both exists but their enhancement might not be allowed. This could for instance be $IV_3 \to IV_2$ (which is not allowed) and $II_3 \to III_2$ (which is allowed). In Table 5.2 the rules as obtained from the algorithm are presented. The green cells indicate that the obtained rules are equal to those presented in [6]. The red cells contain rules that are incorrect. Two example decision trees from which the results are obtained are presented in Figure 5.1.

When comparing Tables 5.1 and 5.2, there are a several important thing to notice. First of all we see that the even when the data does not contain all the allowed single enhancements, the algorithm is able to learn the correct rules. This is

	$h^{1,1}=4$		$h^{1,1}=5$		$h^{1,1}=6$		$h^{1,1}=7$	
II $\to$ II	$0/16$	(10)	$0/25$	(15)	$0/36=0$	(21)	$0/49=0$	(28)
II $\to$ III	$3/16\approx 0.19$	(5)	$6/25=0.24$	(9)	$9/36=0.25$	(14)	$13/49\approx 0.27$	(20)
II $\to$ IV	$6/16\approx 0.38$	(6)	$10/25=0.4$	(10)	$14/36\approx 0.39$	(15)	$19/49\approx 0.39$	(21)
III $\to$ III	$3/9\approx 0.33$	(6)	$6/16\approx 0.38$	(10)	$7/25=0.28$	(15)	$8/36\approx 0.22$	(21)
III $\to$ IV	$6/9\approx 0.67$	(6)	$10/16\approx 0.63$	(10)	$15/25=0.6$	(15)	$21/36\approx 0.58$	(21)
IV $\to$ IV	$9/16\approx 0.56$	(10)	$14/25=0.56$	(15)	$15/36\approx 0.42$	(21)	$16/49\approx 0.33$	(28)

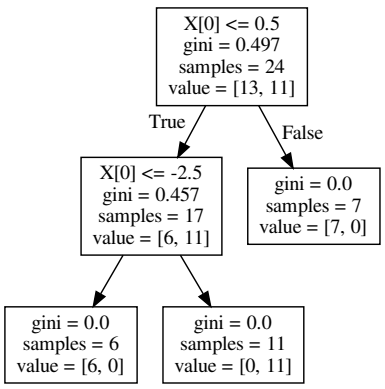
Table 5.1 – For different values of $h^{1,1}$ we have given the number of observed single enhancements and the total possible number of enhancements. To show if the data is equally valid for different values of $h^{1,1}$ we have expressed the fraction of observed against possible data points. In brackets we have given the total number of allowed enhancements, which we have computed by the rules presented in Table 2.2 [6].

for instance the case with the enhancement IV $\to$ IV, for $h^{1,1}\leq 6$. In the case of III $\to$ III the algorithm is never able to learn the correct rules. This can be explained by the number of observed points and the amount of hidden positives in the unobserved data. For $h^{1,1}=4,5$ the number of hidden positives is high , which explains why the algorithm is not able to learn. For the higher values of $h^{1,1}$ it gets even worse, since we only obtain one extra observed point.

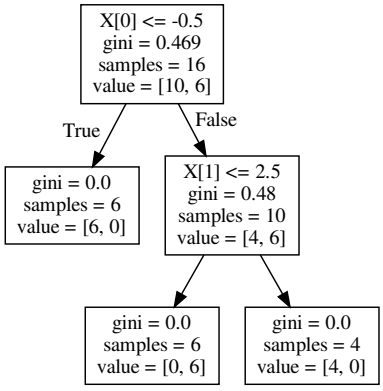
On the enhancement II $\to$ III, the algorithm is only able to obtain the rules for $h^{1,1}=6,7$ even though 5 out of 25 and 7 out of 36 unobserved points are possible. From this enhancement we clearly see that having more points gives the algorithm more capacity to learn. The percentage of observed points for this enhancement is slightly increasing when $h^{1,1}$ increases. This shows, together with the observation that for the enhancement IV $\to$ IV with $h^{1,1}=7$ this percentage has dropped compared to the other values and the algorithm does not learn the rules perfectly. So if this percentage drops when $h^{1,1}$ is increased, the algorithm seems to be unable to learn. A last interesting remark is the fact that when the algorithm is wrong, it is still close to the true rules. This is for instance the case with the enhancement $\text{III}_a \to \text{III}_b$, the algorithm is partly correct since the known rule is $b\geq a$. The algorithm however finds extra restrictions to this enhancement. This suggests that the rules for the enhancement chains in Calabi–Yau manifolds are more restrictive than the general case. Unfortunately the algorithm is not able to learn anything for III $\to$ III with $h^{1,1}=4$.

Enhancement	$h^{1,1} = 4$	$h^{1,1} = 5$	$h^{1,1} = 6$	$h^{1,1} = 7$
$\text{II}_a \to \text{III}_b$	$b - a \leq -1$ or $a \geq 2$	$a \geq 2$ and $b - a \geq -3$	$-2 \leq b - a \leq 0$ ✓	$b - a \geq -2$ and $a \geq 2$ ✓
$\text{II}_a \to \text{IV}_b$	$b - a \geq 1$ and $a \geq 1$ ✓	$b - a \geq 1$ and $a \geq 1$ ✓	$b - a \geq 1$ and $a \geq 1$ ✓	$b - a \geq 1$ and $a \geq 1$ ✓
$\text{III}_a \to \text{III}_b$	No rules	$b - a \geq 0$ and $b \leq 2$	$b - a \geq 0$ and $b \leq 2$ or $b = a = 3$	$b - a \geq 0$ and $a \leq 2$ or $b - a = 0$
$\text{III}_a \to \text{IV}_b$	$b - a \geq 2$ ✓	$b - a \geq 2$ ✓	$b - a \geq 2$ ✓	$b - a \geq 2$ ✓
$\text{IV}_a \to \text{IV}_b$	$b - a \geq 0$ ✓	$b - a \geq 0$ ✓	$b - a \geq 0$ ✓	$b - a \geq 0$ and ($a \leq 5$ or $b - a = 0$)

Table 5.2 – The rules obtained for the single enhancements from the decision trees trained on the data obtained by the algorithm presented in chapter 4. The aglorithm was run on data from Calabi–Yau manifolds with 4 different values of $h^{1,1}$. The cells in green show the cases for which the algorithm has been able to learn the enhancements. The cells in red indicate the cases in which the final decision tree was wrong. This was determined by comparing with the rules as presented in Table 2.2 [6]



(a) The final decision tree of the enhancement $\text{II} \to \text{III}$ on the data for $h^{1,1} = 6$. The array X contains the information on the enhancements with $X[0] = b - a$ and $X[1] = a$. This tree has learned on the data classified by the bagging algorithm.



(b) The final decision tree of the enhancement $\text{III} \to \text{III}$ on the data for $h^{1,1} = 5$. The array X contains the information on the enhancements with $X[0] = b - a$ and $X[1] = a$. This tree has learned on the data classified by the bagging algorithm.

Figure 5.1

5.2 Results of double enhancement

The developed algorithm was run on the observed double enhancements, with the 1000 iterations in both the feature selection and the bagging / classification procedure. The threshold value to add points to the observed points was at 0.9. The features to select were $[a, b, c, c-a, c-b, b-c, , c+a, b+c, b+a]$. The starting value for the minimum number of not bagged elements is 3. When less than 3 points are still not classified, this value decreases such that always at least one point is bagged. The algorithm was stopped when either all points where classified or after 10 full cycles.

The observed data is shown in Table 5.3. By comparing the percentage of observed points between the different values of $h^{1,1}$ it is clear that the data from $h^{1,1} = 6$ and 7 contains much less observed points. The rules for the single enhancements do not depend on the value of $h^{1,1}$, so we would expect to observe approximately the same percentage of points for all values of $h^{1,1}$. With this reasoning, there should more allowed points in the not observed data. Since this is increasing we believer that we do not have enough data for $h^{1,1} = 6$ and 7. This idea is supported by the fact that for the single enhancement, the algorithm fails to learn the correct rules when the percentage of observed points drops. Therefore we only present the rules obtained on the data from Calabi–Yau manifolds with $h^{1,1} = 4$ and 5. These rules are presented in Table 5.4. In Figure 5.2 two example decision trees are shown.

The rules obtained from $h^{1,1} = 4$ and 5 show similarities for the different double enhancements, but (except for $\mathrm{II}_a + \mathrm{III}_b \to \mathrm{IV}_c$) are never exactly the same. In most cases, the rules are based on the same values of types or their difference. For instance when looking at the enhancement $\mathrm{II}_a + \mathrm{III}_b \to \mathrm{III}_c$, the algorithm finds that the value of c tells us if the enhancement is allowed. However the specific value for c differs for both sets of data. In some cases there are some extra restrictions found in one of the two data sets, as is for instance the case with the enhancement $\mathrm{II}_a + \mathrm{IV}_b \to \mathrm{IV}_c$. It also seems as if the rules depend on the value of $h^{1,1}$, for instance with the enhancement $\mathrm{IV}_a + \mathrm{IV}_b \to \mathrm{IV}_c$.

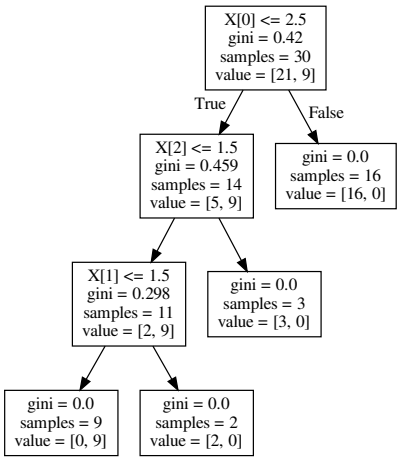
The results are indecisive, since they are different for the 2 values of $h^{1,1}$ and we do not have more results to compare. To improve the results, the most obvious step would be to add more data on the enhancements for $h^{1,1} = 6$ and 7. When we have more data for these Calabi–Yau manifolds we could also use this data. This would give us more rules and enable us to compare which rules depend on the value of $h^{1,1}$.

Enhancement	$h^{1,1} = 4$	$h^{1,1} = 5$	$h^{1,1} = 6$	$h^{1,1} = 7$
$\mathrm{II} + \mathrm{II} \to \mathrm{II}$	0	0	0	0
$\mathrm{II} + \mathrm{II} \to \mathrm{III}$	$4/9 = 0.44$	$10/23 = 0.43$	$16/46 = 0.35$	$26/80 = 0.33$
$\mathrm{II} + \mathrm{II} \to \mathrm{IV}$	0	0	0	0
$\mathrm{II} + \mathrm{III} \to \mathrm{III}$	$4/11 = 0.36$	$10/26 = 0.38$	$13/50 = 0.26$	$17/85 = 0.2$
$\mathrm{II} + \mathrm{III} \to \mathrm{IV}$	$7/14 = 0.5$	$16/30 = 0.53$	$26/55 = 0.47$	$41/91 = 0.45$
$\mathrm{II} + \mathrm{IV} \to \mathrm{IV}$	$10/20 = 0.5$	$20/40 = 0.5$	$24/70 = 0.34$	$29/112 = 0.26$
$\mathrm{III} + \mathrm{III} \to \mathrm{III}$	$4/14 = 0.29$	$7/30 = 0.23$	$7/55 = 0.13$	$8/91 = 0.09$
$\mathrm{III} + \mathrm{III} \to \mathrm{IV}$	$10/14 = 0.71$	$20/30 = 0.67$	$35/55 = 0.64$	$56/91 = 0.62$
$\mathrm{III} + \mathrm{IV} \to \mathrm{IV}$	$16/20 = 0.8$	$30/40 = 0.75$	$35/70 = 0.5$	$41/112 = 0.37$
$\mathrm{IV} + \mathrm{IV} \to \mathrm{IV}$	$19/30 = 0.63$	$34/55 = 0.62$	$35/91 = 0.37$	$56/140 = 0.4$

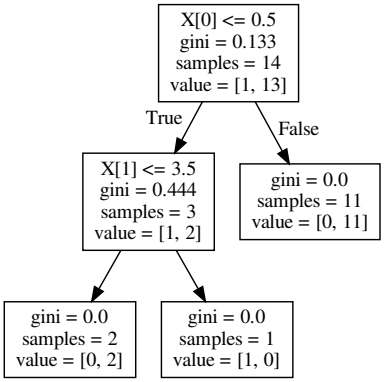
Table 5.3 – For different values of $h^{1,1}$ we have given the number of observed double enhancements and the total possible number of enhancements. To show if the data is equally valid for different values of $h^{1,1}$ we have expressed the fraction of observed against possible data points.

Enhancement	$h^{1,1} = 4$	$h^{1,1} = 5$
$\mathrm{II}_a + \mathrm{II}_b \to \mathrm{III}_c$	$c - a \leq 1$	$c \leq 2$
$\mathrm{II}_a + \mathrm{III}_b \to \mathrm{III}_c$	$c \leq 1$	$c \leq 2$
$\mathrm{II}_a + \mathrm{III}_b \to \mathrm{IV}_c$	$b \geq 1$	$b \geq 1$
$\mathrm{II}_a + \mathrm{IV}_b \to \mathrm{IV}_c$	$b \geq 2$	$a + b \geq 5$ or $c - a \leq 1$ and $b \geq 2$
$\mathrm{III}_a + \mathrm{III}_b \to \mathrm{III}_c$	$c \leq 1$	$c \leq 2$ and $c - a \leq 1$ and $c - b \leq 1$
$\mathrm{III}_a + \mathrm{III}_b \to \mathrm{IV}_c$	$a + b \geq 1$ or $c - a \leq 3$	$a + b \geq 2$ or $c - b \leq 3$ or $c - a \leq 3$
$\mathrm{III}_a + \mathrm{IV}_b \to \mathrm{IV}_c$	$c - b \leq 2$ or $c - a \leq 3$	$c - b \leq 2$ or $a + b \geq 3$ or $c - b \leq 3$
$\mathrm{IV}_a + \mathrm{IV}_b \to \mathrm{IV}_c$	$a + b \geq 4$ or $c - a \leq 1$	$a + b \geq 5$ or $c - a \leq 2$ or $c - b \leq 1$

Table 5.4 – The rules obtained for the double enhancements from the decision trees trained on the data obtained by the algorithm presented in chapter 4. The algorithm was run on data from Calabi–Yau manifolds with 2 different values of $h^{1,1}$.



(a) The final decision tree for the double enhancements from II and III to IV. The input vectors are $[V_{III}, V_{IV} - V_{III}, V_{II} + V_{III}]$. The value array gives the number of elements from each class at a node or leaf. The index (starting from 0) gives the class, where 0 means the enhancement is not possible and 1 means it is possible.



(b) The final decision tree for the double enhancements from III (L) and III (R) to IV. The input vectors are $[V_{III,L} + V_{III,R}, V_{IV} - V_{III,R}, V_{IV} - V_{III,L}]$. The value array gives the number of elements from each class at a node or leaf. The index (starting from 0) gives the class, where 0 means the enhancement is not possible and 1 means it is possible.

Figure 5.2 – The rules obtained for the double enhancements from the decision trees trained on the data obtained by the algorithm presented in chapter 4. The aglorithm was run on data from Calabi–Yau manifolds with 4 different values of $h^{1,1}$.

Chapter 6

Conclusion and outlook

In this thesis we developed an algorithm to study the enhancements of Deligne splittings in Calabi–Yau manifolds. This algorithm is based on machine learning techniques, in particular a bagging algorithm that uses decision trees . The algorithm is mainly based on the algorithm presented in [35].

The algorithm was used to study the limiting mixed Hodge structure that arises at the boundary of the moduli space of a Calabi–Yau manifold. To do this, we first briefly presented the mathematical theory behind this structure. For this we had to discuss the monodromies of the period vector in the complex structure moduli space. Furthermore we used the Nilpotent orbit theorem to get an approximation of the period vector upto exponential corrections. With this we were able to define the mixed Hodge structure and use this information to classify the limits at the boundary, in similar fashion as [6].

Furthermore we have presented a general discussion on Machine learning to help the reader properly understand the techniques needed for the algorithm. The main machine learning algorithm used is the decision trees, which we have favored due to their interpretability. Due to the special nature of the data, we were not able to use conventional machine learning techniques and therefore we resorted to a special type of machine learning named Positive Unlabeled learning. This type of learning was needed to properly classify all the allowed enhancements of the Deligne splittings.

The algorithm presented in this thesis has been able to rediscover the known single enhancement rules for some cases. However, due to a lack of data for some enhancement chains the algorithm has not been successful for all cases. It is however clear that the algorithm can learn the rules when it is presented with enough data. With this algorithm we have also discovered new enhancement rules, for when two divisors intersect. These rules are the main result of this thesis. They present some new insight in the enhancement rules, but the results are unfortunately not decisive yet. This is due to two reasons. First of all the rules discovered for $h^{1,1} = 4$ and 5 differ, making the results unclear. Secondly, we lack enough data for $h^{1,1} = 6$ and 7 to find a pattern in the rules to find mistakes in the results of the algorithm for different values of $h^{1,1}$. When having multiple results for the same enhancements, we can compare the resulting rules and find patterns.

The analysis presented in this thesis does give us some new insight in the limits that occur in the Calabi–Yau manifolds. For instance it is still unknown if all enhancements that are allowed actually occur in Calabi–Yau manifolds [7]. The rules presented in Table 5.2 might suggest that this is not the case, since we have learning different rules by learning with data coming from the Calabi–Yau manifolds compared to the general rules.

There remain several things that we can study further. First of all it would be great to have more data on $h^{1,1} = 6$ and 7, or even higher values. This would give the algorithm more data to learn and we might be able to find rules with a higher degree of certainty. Although we have shown that our algorithm is capable of learning, we might investigate this further. We could test the algorithm on completely different data sets as a proof of concept but we might also be able to study the algorithm more theoretical. In [33] some methods to measure the performance of a PU learning algorithm have been proposed. One part part of the algorithm can be definitely improved, which is the feature selector. We have biased this part towards the rules that are already known, making it not completely generic. Finding a better way to determine a proper basis transformation for the data would greatly improve the algorithm. However the results in this thesis have shown that a PU machine learning method is capable of learning enhancement rules.

Bibliography

[1] D. Tong, "Lectures on string theory," *arXiv preprint arXiv:0908.0333*, 2009.

[2] R. Blumenhagen, D. Lüst, and S. Theisen, *Basic concepts of string theory*. Springer Science & Business Media, 2012.

[3] E. Palti, "The swampland: introduction and review," *Fortschritte der Physik*, vol. 67, no. 6, p. 1900037, 2019.

[4] M. Kreuzer and H. Skarke, "Complete classification of reflexive polyhedra in four dimensions," 2000.

[5] F. Ruehle, "Data science applications to string theory," *Physics Reports*, vol. 839, pp. 1 – 117, 2020. Data science applications to string theory.

[6] T. W. Grimm, F. Ruehle, and D. van de Heisteeg, "Classifying calabi-yau threefolds using infinite distance limits." [1910.02963].

[7] T. W. Grimm, C. Li, and E. Palti, "Infinite distance networks in field space and charge orbits," *Journal of High Energy Physics*, vol. 2019, no. 3, p. 16, 2019.

[8] P. Corvilain, T. W. Grimm, and I. Valenzuela, "The swampland distance conjecture for kähler moduli," *Journal of High Energy Physics*, vol. 2019, Aug 2019.

[9] E. Cattani, A. Kaplan, and W. Schmid, "Degeneration of hodge structures," *Annals of Mathematics*, vol. 123, no. 3, pp. 457–535, 1986.

[10] W. Schmid, "Variation of hodge structure: the singularities of the period mapping," *Inventiones mathematicae*, vol. 22, no. 3-4, pp. 211–319, 1973.

[11] M. Kerr, G. Pearlstein, and C. Robles, "Polarized relations on horizontal sl(2)s," 2017.

[12] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," *Neural Information Processing Systems*, vol. 25, 01 2012.

[13] D. Krefl and R.-K. Seong, "Machine learning of calabi-yau volumes," *Physical Review D*, vol. 96, Sep 2017.

[14] F. Ruehle, "Evolving neural networks with genetic algorithms to study the string landscape," *Journal of High Energy Physics*, vol. 2017, Aug 2017.

[15] Y.-H. He, "Deep-learning the landscape," 2017.

[16] J. Carifio, J. Halverson, D. Krioukov, and B. D. Nelson, "Machine learning in the string landscape," *Journal of High Energy Physics*, vol. 2017, Sep 2017.

[17] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. Understanding Machine Learning: From Theory to Algorithms, Cambridge University Press, 2014.

[18] L. G. Valiant, "A theory of the learnable," *Commun. ACM*, vol. 27, p. 1134–1142, Nov. 1984.

[19] D. Huybrechts, *Complex geometry: an introduction*. Springer Science & Business Media, 2006.

[20] M. Nakahara, *Geometry, topology and physics*. CRC Press, 2003.

[21] B. Greene, "String theory on calabi-yau manifolds," *arXiv preprint hep-th/9702155*, 1997.

[22] S. A. Filippini, H. Ruddat, and A. Thompson, "An introduction to hodge structures," *Fields Institute Monographs*, p. 83–130, 2015.

[23] E. Cattani and A. Kaplan, "Polarized mixed hodge structures and the local monodromy of a variation of hodge structure," *Inventiones mathematicae*, vol. 67, no. 1, pp. 101–115, 1982.

[24] T. M. Mitchell *et al.*, *Machine learning*. McGraw-hill New York, 1997.

[25] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[26] C. M. Bishop, *Pattern recognition and machine learning*. Information science and statistics, New York, NY: Springer, 2006. Softcover published in 2016.

[27] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[28] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[29] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[30] J. R. Quinlan, *C4. 5: programs for machine learning*. Elsevier, 2014.

[31] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[32] L. Hyafil and R. L. Rivest, "Constructing optimal binary decision trees is np-complete," *Information Processing Letters*, vol. 5, no. 1, pp. 15 – 17, 1976.

[33] J. Bekker and J. Davis, "Learning from positive and unlabeled data: A survey," 2018.

[34] J. Bekker and J. Davis, "Estimating the class prior in positive and unlabeled data through decision tree induction," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[35] F. Mordelet and J.-P. Vert, "A bagging svm to learn from positive and unlabeled examples," *Pattern Recognition Letters*, vol. 37, pp. 201–209, 2014.

[36] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[37] B. Liu, Y. Dai, X. Li, W. S. Lee, and P. S. Yu, "Building text classifiers using positive and unlabeled examples," in *Third IEEE International Conference on Data Mining*, pp. 179–186, IEEE, 2003.

[38] G. P. C. Fung, J. X. Yu, H. Lu, and P. S. Yu, "Text classification without negative examples revisit," *IEEE Trans. on Knowl. and Data Eng.*, vol. 18, p. 6–20, Jan. 2006.

[39] P. Candelas, A. Dale, C. Lutken, and R. Schimmrigk, "Complete Intersection Calabi-Yau Manifolds," *Nucl. Phys. B*, vol. 298, p. 493, 1988.