Universiteit Leiden
The Netherlands

Opleiding Informatica

A Type System
for Machine Learning Pipelines

Huib Sprangers

Supervisors:
Henning Basold and Mitra Baratchi

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

06/06/2024

Abstract

In this thesis a type system is introduced intended to help in the creation of machine learning pipelines. This type system makes use of lenses in order to allow for the definition of forward- and back-propagation within machine learning algorithms within this system.

An implementation of this type system was made in the Haskell programming language, which showcases that that this system can be used to implement machine learning algorithms, using linear regression as an example.

Contents

1 Introduction

In modern machine learning, it is often desired to be able to test multiple different types of algorithms with different sets of parameters on the same dataset, to see which combination obtains the best results when applied to said dataset. In order to standardize the processes which make up machine learning, pipelines were created, which give generalized structures for machine learning processes. These pipelines are intended to be used to test different configurations of different algorithms using a particular structure which can be applied to any number of datasets with minimal input from the user.

One possible framework from which these pipelines can be derived is RECIPE [dSPOP17], which uses specially defined grammars in order to set up various types of valid pipelines. These pre-programmed grammars are written specifically to only allow pipeline structures that suit the particular type of data for which said grammar is written. While this method has the advantage of being able to cater to any particular dataset's specific intricacies, but having to write the each of these possible grammars on a case by case basis means there is little flexibility to each of these grammars.

This thesis seeks to research the viability of using such a type-based machine learning pipeline, which aims to be less reliant on pre-programmed grammars to not require creating specific use-cases for each dataset. In addition, this model is intended to be modular, with the ability to easily add components through composition. The aim is to create an efficient machine learning pipeline that can easily be expanded and requires minimal need to adapt the created machine learning pipeline for each dataset it is used on.

For this research, the question we seek to answer is whether it is possible to create a type system for machine learning pipelines, as well as find out the benefits and drawbacks of using such a system, compared to previous methods.

A type system is defined for this, which makes use of bidirectional maps called lenses in its structure. The reason we decided to use types to try and improve upon previous pipeline creation frameworks is because types simplify error checking and are easily translated into code for many programming languages, which are themselves often type based. Type systems can also easily be made modular using compositional reasoning, which involves using generic building blocks which can be composed together using generically defined parts to form pipelines for our purposes. These pipelines can then easily be tested on their validity through type checking parts of the pipeline and parts of its composed blocks.

The type system has been implemented in the functional programming language called Haskell. An attempt was first made to implement this system in the the machine learning library known as Tensorflow [Goo] [Ten] in order to demonstrate its effectiveness and potential, but this ended up not being viable. A second implementation was then made more generically within haskell, which was used to successfully implement a linear regression algorithm.

In Section 2, background knowledge is provided, mainly partaining to type systems, machine learning and lenses. Section 3 focuses on the type system, it's semantics and implementations for particular machine learning algorithms. Afterwards, Section 4 showcases how machine learning algorithms can be implemented in our type system. Finally, Section 5 details the conclusions that can be reached about the system, and recommends possibilities for further research.

2 Background

2.1 Type Systems

In modern software engineering, there have been many formal methods which help ensure that a system runs correctly, in accordance with its desired specification. A type system is one such formal method that uses types and terms in order to ascribe properties to a system and use the principle of compositionality in order to allow those properties to maintain internal consistency within this system, in order to prove the absence of particular program behaviors through the values they compute. This system is often used as the logical base for programming languages due to it using well-defined tools for reasoning about the syntax and semantics of a program.

Within a type system, every term has a type, usually written as $t : A$ within type theory, which is read as "term t has type A". For example, there can be the term $4 : nat$, which asserts the number 4 as being of the type 'nat', here standing for 'natural number'. Terms that represent a set value of a particular type are often called base terms. For the natural numbers, each and every natural number (1,2,3,...) can be defined as a base term, though it is also possible to define the natural numbers with only 0 as a base term and a successor base term to define subsequent natural numbers. For Booleans, the only base terms are true and false.

Simply defining which base terms inhabit a type is not enough, however. The type system also needs to define how these terms interact with each other. For this constructors can be created, terms which take one or more other terms as arguments, which evaluate to a valid term within the type system themselves, acting as transformations of the terms they are given arguments. These constructors are used to define the properties and relations of terms within a type. An example of this for a type that defines natural numbers as given earlier would be 'succ n', a constructor that takes a natural number n as argument and gives the successive natural number, which evaluates to a base term of function type $nat \to nat$, defining how the natural numbers are ordered.

To give an illustration of this type system for natural numbers, we can define the following types:

$A ::=$	*types:*
nat	*natural numbers*

And then define the terms we described earlier:

$t ::=$	*terms:*
0	*zero*
$\mid$ **succ** t	*successive terms*

Term constructors can in this way be used to define the properties of the terms within a given type system. But even then, exactly what these constructors do will often need to be defined. For this, type rules will need to be given which formalize a static abstraction of the system's behavior. For example, the previously given constructor 'succ n' needs to have it be defined that if $n : nat$, that the result from using succ is that **succ** $n : nat$. This type rule thus formalizes the property of

natural numbers that for any natural number, there is a successive natural number that is, in itself, also a natural number. Do note that the exact implementation of succ is not given here, simply its property of always giving another natural number when a natural number is given, if a different type of number is given, succ is meant to fail due to this behavior being undefined in this example.

A type rule definition for the successive term of a natural number for this given type system would be as follows:

$$\frac{n : \mathbf{nat}}{\mathbf{succ}\ n : \mathbf{nat}}\ \text{succ}$$

What this rule essentially says is that for any term of type 'nat', there is a successive term of the same type. The actual behavior of 'succ n', that the successive term of 0 is 1, is not defined by the system itself.

For how to actually implement the system we talk about in this thesis as a piece of code, we'll first need to talk about Machine Learning, and the concepts that are applicable to what we're attempting to accomplish.

2.2 Machine Learning

Machine learning is, in essence, the creation of algorithms which can learn from data without being explicitly programmed. Various types of algorithms have been created over the years to accomplish this, with various goals and methodologies.

This thesis generally assumes basic knowledge about the machine learning concepts listed in this Section and some of their inner workings, though exact knowledge of every algorithm in not needed. This thesis will thus not go into too much detail on such matters. Further reading is recommended for any reader who is unfamiliar with the following concepts [Fla12]:

- Regression. These types of algorithms seek to find a continuous value from the provided input in order to predict certain values in a relation. Examples include Linear Regression, Logistic Regression and Polynomial Regression, whose differences generally lie in what kind of relation the continuous value is expected to take by the algorithm, with linear regression expecting a linear relation, logistic regression expecting a logarithmic relation, etc.

- Classification. These types of algorithms seek to find discrete values from the provided input, usually in the form of categories or labels. Examples include Decision Trees, Random Forests and K-nearest Neighbours algorithms. These types of algorithms typically try to create some sort of criteria for when a certain label should be applied, creating decision trees and forests out of the features of the data and how they could relate to a certain label or simply checking against a certain amount of the most similar cases to a current data point in the case of K-nearest Neighbours.

- Neural Networks. These types of algorithms model themselves on the neurons of the human brain, creating connections between different nodes of the networks and assigning weights to each connection and node to adjust the learning process. Neural Networks have seen use in many different disciplines, examples of this include Deep Learning being applied for speech recognition and machine learning, amongst other uses, and the Large Language Models (LLM) which have in recent years spawned the notorious Generative pre-trained transformers (GPT).

With so many different algorithms, it can sometimes be hard to figure out which would best fit the data. In addition, it can be difficult to find out what parameters would make a certain algorithm work best for said data. Testing out each possibility would be inefficient, not to mention time consuming if done by hand. For the purpose of providing a framework to automatically checking multiple of these algorithms on a particular dataset, machine learning pipelines were created.

2.2.1 Pipelines

A machine learning pipeline is a series of interconnected data processing steps and machine learning models arranged in a sequential order to automate and streamline the process of building, training, and deploying machine learning models. The main purpose of such a pipeline is to organize and structure the workflow involved in developing a model, from data pre-processing to model evaluation. A typical, though fairly simplified, workflow for a machine learning pipeline can be seen in Figure 1, the backwards arrow from Model Evaluation to Feature Selection is there to indicate that once a previous model is evaluated, it will most the time be used to further optimize the previous steps.

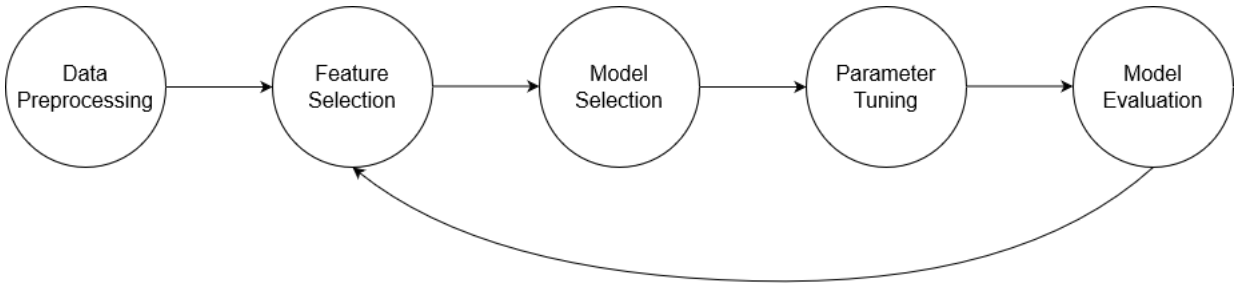

Figure 1: An example of a typical machine learning pipeline

The first step shown here is Data Preprocessing, which means cutting down, editing and modifying the data into a form which can be used by the model. Examples of this include cutting out or filling out missing chunks of data, standardizing the formats of fields such as dates, reformatting multiple different datasets into a single whole and changing the types of certain data fields, for example from strings to integers, to better work with a chosen model. While this is an important step, it is not covered by our model, which presumes that the data has already had preprocessing done.

The second step is Feature Selection, which is often expanded to also include steps like feature creation and preprocessing. Features, in machine learning, indicate the criteria by which a model is built for a dataset. If aiming to create a model that tries to predict the effectiveness of advertising using a certain key phrase, the number of sales and the days said advertisement was shown can be used to train a model, but this can result in leaving out other important factors. By creating more features that represent additional factors such as the frequency of advertisement in certain areas, as well as comparing the predictive abilities of different models that use different features, models can be compared in effectiveness with each other. This process can be largely automated through various steps.

Next is Model Selection, the act of testing which model to use on a certain dataset. Whether it be simple linear or logistic regression, a complex neural network or any of the other many types of models, the unpredictable nature of especially large datasets makes it hard to predict what sort of

machine learning model will create the best fit, without over-fitting the model to the data. This and the next step of the shown workflow are chief amongst the parts that the type system in this thesis is created to automate.

This next step is Parameter Tuning, which means to optimize the settings which determine the specific workings of a machine learning model. Many models also have their own parameters which determine its exact workings, like the maximum depth of a decision tree, or the learning rate. Tuning is done by testing variations of the same model with variance in the parameters, in order to reach the ideal combination for a given dataset.

Lastly, we have Model Evaluation, which is simply to evaluate whether the created model works to a satisfactory degree. Testing the accuracy, recall, whether over-fitting or under-fitting occurred are all part of this step. The type system created for this thesis does not cover this step by itself.

One issue these machine learning algorithms and pipelines all share is that, to a layman, they are difficult to understand and time consuming to implement. In addition, even for experts it can be hard to decide which algorithm will work best on a particular dataset, not to mention tuning the parameters of these algorithms to try and produce the best results. Clearly, there exists a vested interest in creating a fast and easy way to try out and test many different algorithms, and variants of these algorithms, on a particular dataset, with minimal need to understand the underlying mechanisms. For this purpose, AutoML was created.

2.2.2 AutoML

Automated Machine Learning is a research area that aims to automate the time consuming, iterative tasks required for machine learning development, allowing users to build Machine Learning models of great size and efficiency without sacrificing quality.

This field of research has three primary aims:

- Accessibility: Allowing for Machine Learning to be more easily used by people who are not proficient with it.
- Efficiency: Reducing the time that people will need to spend on fine tuning algorithms for use on a dataset.
- Speeding up Research: By allowing for pipelines to be run automatically, research on machine learning can be greatly accelerated.

The goal for AutoML is to be able to simplify a typical machine learning workflow to be run automatically, allowing users to solely focus on collecting data and telling the automated process how to use it, requiring minimal input from the user.

A typical AutoML process works by applying a large number of different machine learning algorithms to a given dataset, as well as many different configurations of these algorithms. The models these algorithms create are then tested against each other based on various criteria, defined by the user based on what problem the process aims to solve.

Examples of automated machine learning pipelines and their implementations include TPOT [OBUM16], which uses a tree based structure for pipeline optimization, and RECIPE [dSPOP17], a Grammar based approach. The model in this thesis will instead attempt to create a type system for this purpose.

2.3 Lenses

For the purpose of designing a type system which is capable of accurately modelling machine learning pipelines, it is important to use a structure where information is capable of flowing forwards and backwards. This is due to the fact that in order for a learning algorithm to function, it needs to be able to self-evaluate. And in order to self evaluate, the algorithm needs to have backpropagation, often this is to return updated parameters or gradients for the weights used to get closer to the ideal values used.

An example of an algorithm that uses such backpropagation would be a neural network, using gradient descend to update each of its nodes' weights in accordance with how well they perform. To this end of requiring a structure that can represent this type of behavior, the pipeline's design makes use of lenses, which are a bidirectional structure designed as seen in Figure 2.

A basic lens as shown in Figure 2 is generally considered to be a bidirectional mapping from $(A, A') \to (B, B')$, which consists of the functions (f, f'). The variables A, A', B and B' represent sets of data to be used as arguments or returned by these functions. f, which can also be referred to as the *forward* function of the lens, is a function $A \to B$, while f', referred to as the *reverse* or *backwards* function, is a function with the type $A \times B' \to A'$.

Often times, the type of A is the same as that of A', and the type of B is also identical to that of B', though they do not necessarily have to be for a lens. Nonetheless any such pair will generally still be referred to as X and X' to always give a clear indication that the two are separate objects.

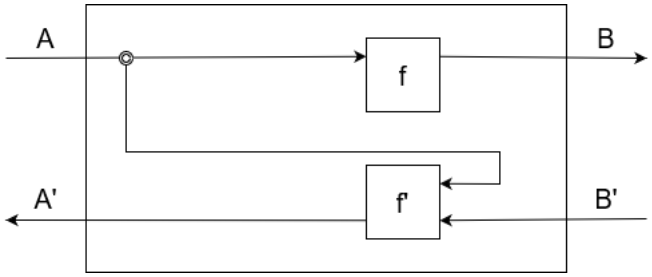

Figure 2: A detailed diagram of a Lens

Two lenses can also be composed together, either in sequence or in parallel. The composite of this pair of lenses can then be construed as a lens itself, with the functions from which these lenses consist forming functions within this new lens.

In Figure 3 we see a sequential composition of the lenses $(A, A') \to (B, B') = (f, f')$, called lens f for short, and $(B, B') \to (C, C') = (g, g')$, called g. Note that it is important that the type of (B, B') within f match those in g, as this type of composition is impossible otherwise. This will henceforth be referred to as the composition $g \circ f = (h, h')$, which can be considered onto itself to be the lens h with type $(A, A') \to (C, C')$. With this we can also define its forward and reverse functions h and h':

$$h(x) = g(f(x))$$
$$h'(x, y) = f'(x, g'(f(x), y))$$

In Figure 4 we see a parallel composition of the lenses $(A, A') \to (B, B') = (f, f')$, called lens f for short, and $(C, C') \to (D, D') = (g, g')$, called g. This will henceforth be referred to

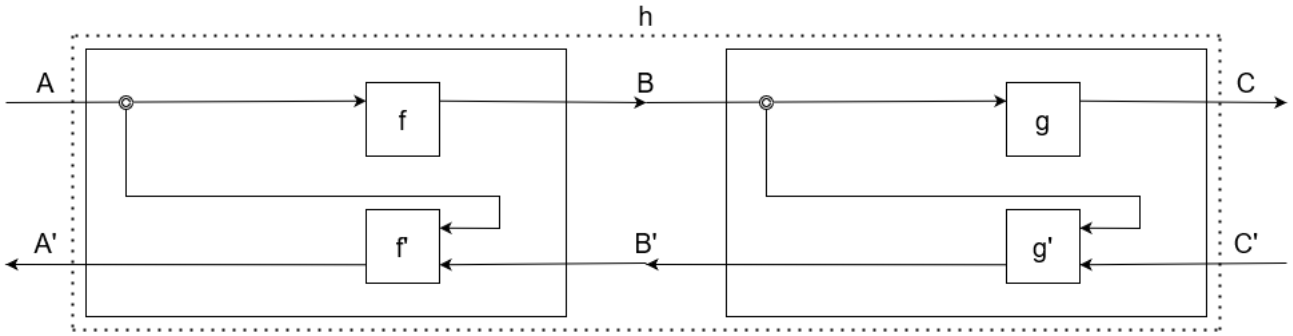

Figure 3: A diagram of two Lenses composed in sequence

as the product $g \times f = (h, h')$, which can be considered onto itself to be the lens h with type $(A \times C, A' \times C') \to (B \times D, B' \times D')$. Here a type being defined as $A \times C$ could create some issues with how the two types are defined. With the assumption that both A and C are fixed sets of real numbers, we can state that $A \times C$ is the product of sets A and C. For example, if $A = \mathbb{R}^n$ and $C = \mathbb{R}^m$, then the product $A \times C$ has dimensions $n \times m$ and is therefore isomorphic to $\mathbb{R}^{n+m}$. In this case, however, since both lenses will likely require different functions to be ran over their own data, it is a necessity to be able to separate $A \times C$ back into A and C with their correct dimensions and data.

With this we can define the functions h and h' as follows:

$$h(x, y) = (f(x), g(y))$$
$$h'(v, w, x, y) = (f'(v, w), g'(x, y))$$

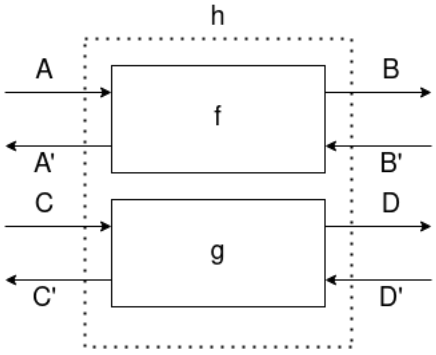

Figure 4: A diagram of two Lenses composed in parallel

2.3.1 Parametric Lenses

For the purpose of learning parameters, a simple lens is not enough, the data and the parameters for the machine learning algorithm to learn need to be managed separately. Thus, we make use of the parametric lenses introduced in [FST19] and elaborated on in [CGG+22].

Essentially, a parametric lens is a Lens whose dimensions now include an extra value in its input and output. Compare the lens shown in Figure 2 to the parametric lens shown in Figure 5. For the parametric lens, the dimensions of the map change to $(P, P') \times (A, A') \to (B, B') = (f, f')$, the forward function f changes to $P \times A \to B$ and the reverse function f' changes to $P \times A \times B' \to P' \times A'$.

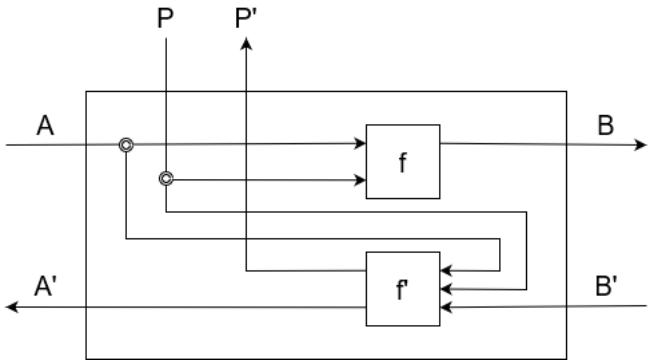

Figure 5: A detailed diagram of a Parametric Lens

An example of a machine learning concept that can be represented by a parametric lens would be this approximation for a loss function using quadratic error, where $f_{loss}(p, a)$ is the forward function and $f'_{loss}(p, a, b)$ is the reverse. For these functions, the input p is given through the parameter space, a is given through the input space and b is given through the output space, in the same manner as in Figure 5

$$f_{loss}(y_t, y_p) = 1/2 * \sum_{i=1}^{y} ((y_p)_i - (y_t)_i)^2$$
$$f'_{loss}(y_t, y_p, \alpha) = \alpha * (y_p - y_t, y_t - y_p)$$

Here, y_t and y_p stand for the true value of the equation and the predicted value calculated by the model, and α stands for the learning rate. In this case, the forward function's result would be sent along to another lens to presumably calculate the loss function, which is then sent along to the reverse function, whose result is then returned to the model's own reverse function.

But in order for the input and output of this loss lens to be properly given to other components, we must first define the ways different lenses can be composed together. In addition, we need to define how the lens that results for such a composition has its forward and reverse functions defined.

Compared to regular lenses the sequential composite of two parametric lenses now also requires the parameter space, as seen in Figure 6. The lens f defined as $(P, P') \times (A, A') \to (B, B') = (f, f')$ and the lens g defined as $(Q, Q') \times (B, B') \to (C, C') = (g, g')$ when composed together as $g \circ f = (h, h')$ form the lens h with dimensions $(P \times Q, P' \times Q') \times (A, A') \to (C, C')$. The definitions for h and h' are given as follows:

$$h(x, y, z) = g(z, f(y, x))$$
$$h'(v, x, y, z) = f'(y, x, g'(v, f(y, x), z))$$

Since the parameters for f and g can be different in size and exact type, the two need to be combined together for h, but able to be separated to properly fit into f and g.

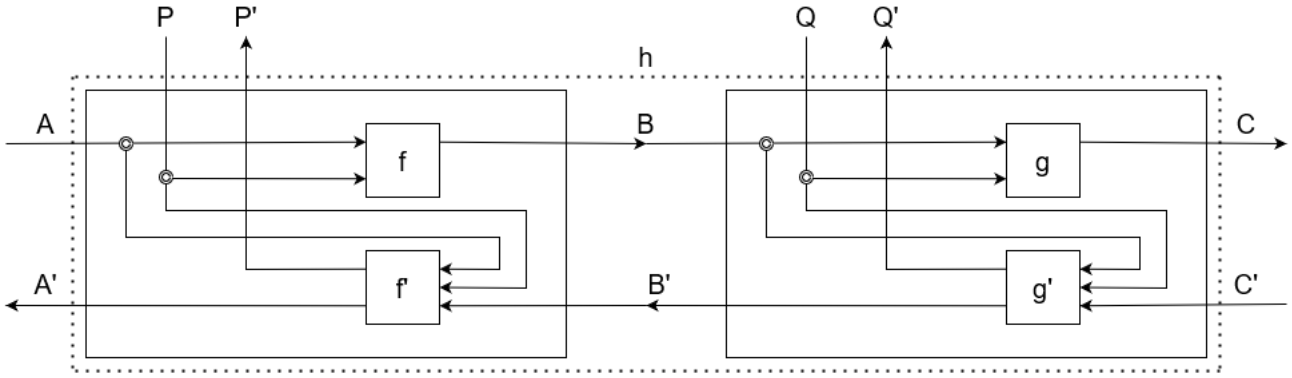

Figure 6: A diagram of two Parametric Lenses composed in sequence

As seen in Figure 7, the parallel composition of two parametric lenses is similar to that of the regular lens in Figure 4, but with the parameters added. This gives the parametric lenses $(P, P') \times (A, A') \to (B, B') = (f, f')$, called lens f for short, and $(Q, Q') \times (C, C') \to (D, D') = (g, g')$, called g. This will henceforth be referred to as the product $g \times f = (h, h')$, which can be considered onto itself to be the lens h with type $(P \times Q, P' \times Q') \times (A \times C, A' \times C') \to (B \times D, B' \times D')$. The definitions for the functions h and h' are given accordingly:

$$h(w, x, y, z) = (f(w, x), g(y, z))$$
$$h'(u, v, w, x, y, z) = (f'(u, v, w), g'(x, y, z))$$

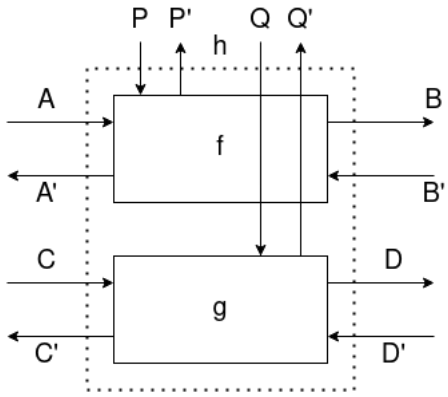

Figure 7: A diagram of two Parametric Lenses composed in parallel

Sometimes it is required to compose a lens whose function is to manage changes to the parameters, in this case we can compose two parametric lenses in a manner as shown in Figure 8. This uses the parametric lenses $(P, P') \times (A, A') \to (B, B') = (f, f')$, called lens f for short, and $(Q, Q') \times (0, 0) \to (P, P') = (g, g')$, called g.

For indicating such a composition over the parameter space, the symbol $\circ_p$ is introduced. This creates the parametric composition $g \circ_p f = (h, h')$ shown in Figure 8, where the lens h has the type $(Q, Q') \times (A, A') \to (B, B')$.

Note that $(\mathbb{R}^0, \mathbb{R}^0)$ indicates this field gets an empty, or at least not used, field as input, meaning the product $A \times \mathbb{R}^0$ which would serve as input for the lens h does not have its dimensions expanded and evaluates simply to A. This is required for this composition due to the difficulty with mixing extra data into input spaces like (A, A'), which usually contain large data of a specific kind, compared to the often mixed parameter space. This will need to be accounted for in any lens g that is created for this kind of composition, any sort of data will need to be run through (Q, Q').

The definitions for h and h' are as follows:

$$h(x, y) = f(g(y, 0), x)$$
$$h'(x, y, z) = g'(y, 0, f'(g(y, 0)), x, z))$$

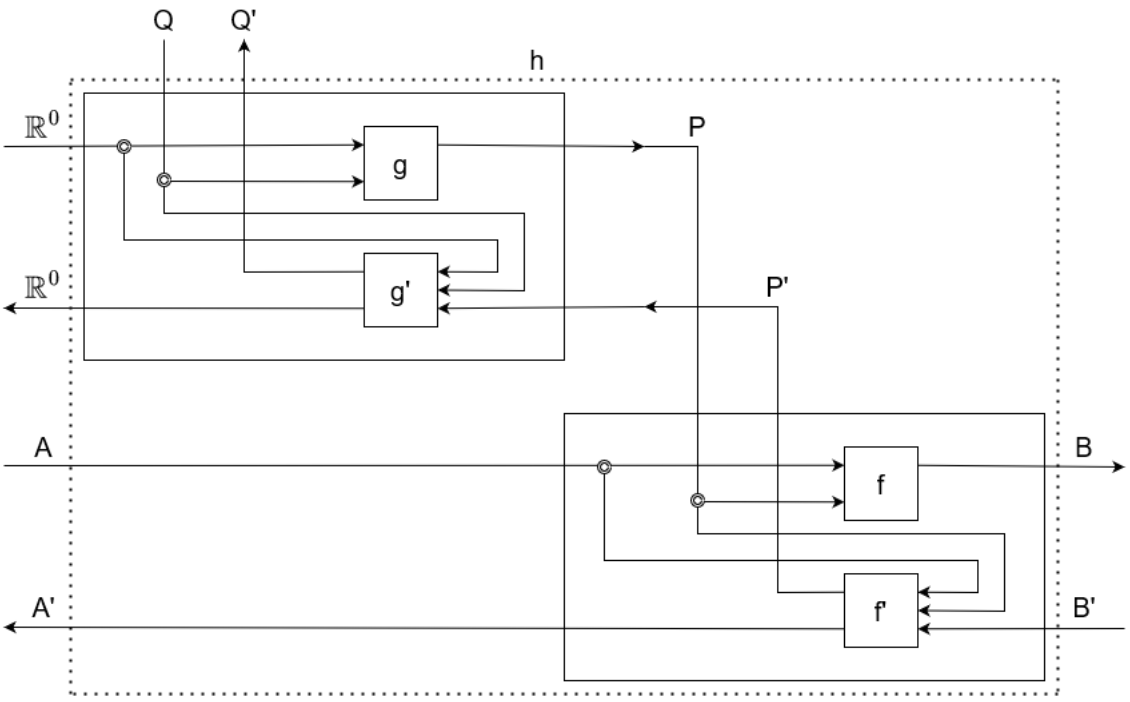

Figure 8: A diagram of two Parametric Lenses composed over the parameter space

2.4 Model

The following figures are diagrams used to demonstrate typical machine learning pipelines within bounds of the created type system, where each component represents a lens. Figure 9 contains explicit parameters, which are used in the calculations within the system's components. Figure 10 only has these parameters implicitly, meaning that while the parameters are still present, they cannot be accessed by the lens' functions.

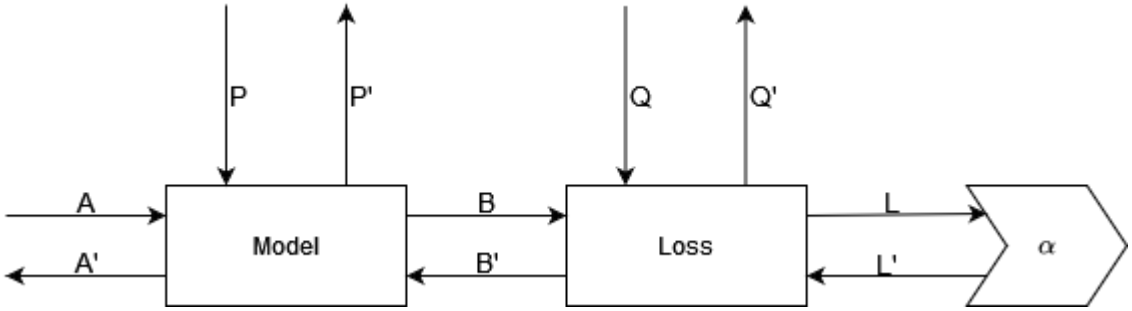

Figure 9: A diagram of a typical machine learning pipeline within the created type system with explicit parameters

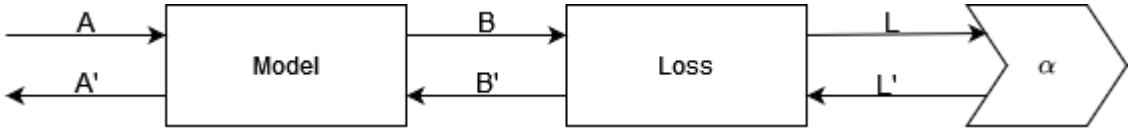

Figure 10: A diagram of a typical machine learning pipeline within the created type system with implicit parameters

In both figures, we see three components through which the data is run. Model, which calculates the data that will be used by the loss function as its forward function, and returns the changed data for a single run of the algorithm. Loss, which handles the loss function, calculating how much information is left incorrect each run. And the learning rate α, which simply determines the degree to which each learning step can change the program. Often the learning rate is dynamically altered, but for the sake of keeping this pipeline simple, it is kept constant for our purposes.

The pipelines shown here represent the simplest forms of a machine learning pipeline. When the type system is used to implement real algorithms, more components will often be needed. When they are, whether to edit the data or the parameters, they can simply be composed onto the base diagrams shown earlier through regular or parametric composition.

3 Type System Definition

Each of the shown components of which the pipelines shown in the diagrams of Figure 9 and Figure 10 consist are defined within the bounds of the following type system:

Types are generated according to the following grammar, where each variable a,a',b,b',p,p' is a natural number:

$$
\begin{array}{rr}
A ::= & \textit{types:} \\
\quad (p, p') \times (a, a') \to (b, b') & \textit{explicit} \\
\quad \mid (a, a') \to (b, b') & \textit{implicit}
\end{array}
$$

The explicit type represents a component modeled as a parametric lens like shown in Figure 5 while the implicit type represents a component modeled as a basic lens like shown in Figure 2.

Note that within type A, the lowercase letters represent natural numbers of a particular dimension. For example, space (a, a') representing the input data space can have the value $(2, 2)$, which would mean both input and output would be a collection of two numbers. Theoretically this system could be expanded to accept more kinds of input, but that is beyond the scope of this thesis. The intent is to formalise lenses with only $\mathbb{R}^n$ on the interface, thus only dimensions are used in the type definitions.

For this type system, the following terms are given:

$$
\begin{array}{rr}
t ::= & \textit{terms:} \\
\quad \textbf{id} & \textit{identity function} \\
\quad \mid \textbf{hide}\ t & \textit{type conversion} \\
\quad \mid \textbf{switch}\ t & \textit{output switching} \\
\quad \mid t \times t & \textit{parallel composition} \\
\quad \mid t \circ t & \textit{sequential composition} \\
\quad \mid t \circ_p t & \textit{parametric composition} \\
\quad \mid b \in B & \textit{collection of base terms}
\end{array}
$$

Here, **id** represents the identity function, a term that, when sequentially composed to any term, gives the same term as output. **hide** is used to convert from the explicit lens type within A to the implicit type, because of this, **hide** can only be applied to a term t if t is of the explicit type. The term constructors $\times$, $\circ$ and $\circ_p$ represent the various types of composition of basic and parametric lenses shown in Section 2.3. Parallel and sequential composition can be applied with both the explicit and implicit types, while the parametric composition can only be applied using explicit lenses, due to requiring the usage of the parameters.

The **switch** term can be applied to both explicit and implicit types. This term is meant to allow for the reordering of input and output, switching which data flow would enter which component in a parallel composition. An example of this for implicit lenses can be found in Figure 11. For explicit lenses this term operates much the same, the parameters for the explicit **switch** are considered to be $\mathbb{R}^0$ in this case, or at the very least unused when **switch** is applied to an explicit lens.

Lastly, the collection of terms B stands for the collection of base terms that are specific to each algorithms that can be implemented in this type system. Each $b \in B$ stands for a base term, while exactly what terms the collection B consists of depends on the pipeline in question.

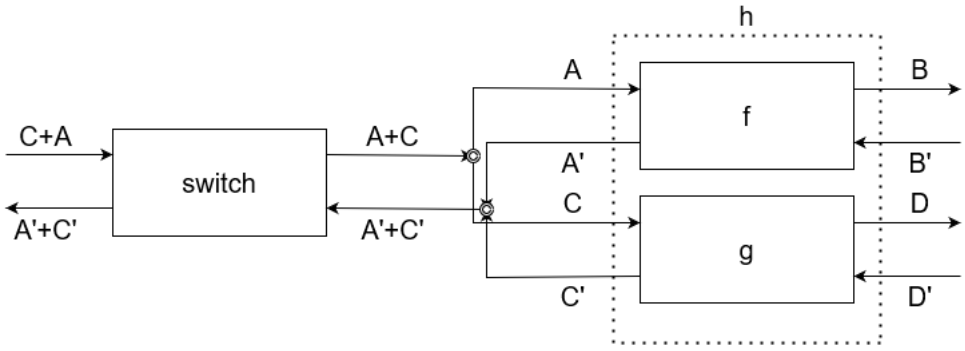

Figure 11: A diagram showing the **switch** term used to switch the inputs for a parallel composition

3.1 Type Rules

In order for the behavior of a type system to be defined, we must define certain rules which its terms act in accordance with, primarily in order to define what sort of input types these terms will and will not accept as well as the type which these terms simplify to. Taking the first rule showcased here, $A1 - comp$, as an example: this rule define the behavior of the sequential composition which takes two explicitly typed lenses, forming an explicit lens in turn.

While $A2 - comp$ does the same for the sequential composition of two implicitly typed lenses, there is no definition for the sequential composition of an explicitly typed lens and an implicitly typed one, therefore such behavior is invalid within this type system. In this way, what sort of input is and is not given a type rule defines the behavior of the type system.

The terms given in the previous section are defined according to the following type rules:

$$\frac{s : (p, p') \times (a, a') \to (b, b') \qquad t : (q, q') \times (b, b') \to (c, c')}{t \circ s : (p + q, p' + q') \times (a, a') \to (c, c')} \text{ A1-comp}$$

$$\frac{s : (a, a') \to (b, b') \qquad t : (b, b') \to (c, c')}{t \circ s : (a, a') \to (c, c')} \text{ A2-comp}$$

$$\frac{s : (p, p') \times (a, a') \to (b, b') \qquad t : (q, q') \times (0, 0) \to (p, p')}{t \circ_p s : (q, q') \times (a, a') \to (b, b')} \text{ P-comp}$$

$$\frac{t : (p, p') \times (a, a') \to (b, b')}{\textbf{hide}\ t : (a, a') \to (b, b')} \text{ hide}$$

$$\frac{t : (0, 0) \times (a + a', b) \to (a + a', b)}{\textbf{switch}\ t : (0, 0) \times (a + a', b) \to (a' + a, b)} \text{ E-switch}$$

$$\frac{t : (a + a', b) \to (a + a', b)}{\textbf{switch}\ t : (a + a', b) \to (a' + a, b)} \text{ I-switch}$$

$$\frac{s : (p, p') \times (a, a') \to (b, b') \qquad t : (q, q') \times (c, c') \to (d, d')}{s \times t : (p + q, p' + q') \times (a + c, a' + c') \to (b + d, b' + d')} \text{ A1-prod}$$

$$\frac{s : (a, a') \to (b, b') \qquad t : (c, c') \to (d, d')}{s \times t : (a + c, a' + c') \to (b + d, b' + d')} \text{ A2-prod}$$

$$\frac{}{\mathbf{id} \ : (0, 0) \times (a, a') \to (a, a')} \text{ id}$$

For the construction of a pipeline, a collection of base terms B is created and composed according to the preceding type rules. Each collection contains components of type A. For different machine learning algorithms, different base terms are defined in accordance to the potential needs and wants of said algorithm.

What follows is a type rule that allows the creation of base terms that are from then onward considered part of the collection of base terms B for that particular application of the type system:

$$\frac{b : (p, p') \times (a, a') \to (b, b) \in B}{b : (p, p') \times (a, a') \to (b, b)} \text{ A1-basic term}$$

To demonstrate the way this type system works, we can apply the following definitions to the base terms which form the collection B_{basic} for the components of the basic pipeline shown in Figure 9:

$$Model : (p, p) \times (n, n) \to (m, m)$$
$$Loss : (q, q) \times (m, m) \to (1, 1)$$
$$\alpha : (0, 0) \times (1, 1) \to (0, 0)$$

Each of these base terms represents a parametric lens. More complex algorithms will naturally require more components to operate and thus a larger collection of base terms. Most algorithms will make use of the Loss and α components, though their exact definitions may differ depending on the nature of the algorithm.

When composing the lenses of the basic pipeline together, the following type derivation is found:

$$Loss \circ Model : (p \times q, p \times q) \times (n, n) \to (1, 1)$$
$$\alpha \circ Loss \circ Model : (p \times q, p \times q) \times (n, n) \to (0, 0)$$

Note that our composition operations $\circ$ and $\circ_p$ are right associative, meaning that $\alpha \circ Loss \circ Model$ is equivalent to $\alpha \circ (Loss \circ Model)$. This is demonstrated for the base terms of collection B_{basic} in the following derivation tree:

$$\frac{\alpha \in B_{basic} \qquad \dfrac{Loss \in B_{basic} \qquad Model \in B_{basic}}{Loss \circ Model : (p + q, p + q) \times (n, n) \to (1, 1)} \text{ A1-Comp}}{\alpha \circ Loss \circ Model : (p + q, p + q) \times (n, n) \to (0, 0)} \text{ A1-Comp}$$

3.2 Semantics

The following semantics can be given for use with any system that implements the type system A. First, we provide semantics for the two possible types of A. Note that it is a requirement for f to be differentiable and f' to be linear, due to certain models requiring f to be differentiated to find f'.

$$[\![(p, p') \times (a, a') \to (b, b')]\!] = \{(f, f') \mid (f, f') \colon (\mathbb{R}^p, \mathbb{R}^{p'}) \times (\mathbb{R}^a, \mathbb{R}^{a'}) \to (\mathbb{R}^b, \mathbb{R}^{b'}) \text{ lens}\}$$

$$[\![(a, a') \to (b, b')]\!] = \{(p, p', f, f') \mid (f, f') \colon (\mathbb{R}^p, \mathbb{R}^{p'}) \times (\mathbb{R}^a, \mathbb{R}^{a'}) \to (\mathbb{R}^b, \mathbb{R}^{b'}) \text{ lens}\}$$

The sequential composition of the following parametric lenses (f, f') and (g, g') happens according to the model shown in Figure 6:

$$\begin{gathered}
[\![\vdash s : (p, p') \times (a, a') \to (b, b')]\!] = (f, f') \\
f : \mathbb{R}^p \times \mathbb{R}^a \to \mathbb{R}^b \\
f' : \mathbb{R}^p \times \mathbb{R}^a \times \mathbb{R}^{b'} \to \mathbb{R}^{p'} \times \mathbb{R}^{a'} \\
[\![\vdash t : (q, q') \times (b, b') \to (c, c')]\!] = (g, g') \\
g : \mathbb{R}^q \times \mathbb{R}^b \to \mathbb{R}^c \\
g' : \mathbb{R}^q \times \mathbb{R}^b \times \mathbb{R}^{c'} \to \mathbb{R}^{q'} \times \mathbb{R}^{b'} \\
[\![t \circ s]\!] = [\![t]\!] \circ [\![s]\!] = (f, f') \circ (g, g')
\end{gathered}$$

The sequential composition of the following lenses (f, f') and (g, g') into (h, h') shown in as follows happens according to the model shown in Figure 3, though due to the properties of Implicit Parametric Lenses, the parameters are added into the equation, turning (f, f') into (p, p', f, f'), (g, g') into (q, q', g, g') and due to the composition rules, (h, h') into $(p + q, p' + q', h, h')$:

$$\begin{gathered}
[\![\vdash s : (a, a') \to (b, b')]\!] = (p, p', f, f') \\
p, p' \in \mathbb{N} \\
f : \mathbb{R}^a \to \mathbb{R}^b \\
f' : \mathbb{R}^a \times \mathbb{R}^{b'} \to \mathbb{R}^{a'} \\
[\![\vdash t : (b, b') \to (c, c')]\!] = (q, q', g, g') \\
q, q' \in \mathbb{N} \\
g : \mathbb{R}^b \to \mathbb{R}^c \\
g' : \mathbb{R}^b \times \mathbb{R}^{c'} \to \mathbb{R}^{b'} \\
[\![t \circ s]\!] = (p + q, p' + q', (g, g') \circ (f, f'))
\end{gathered}$$

Next we look at the parametric composition of two parametric lenses shown in Figure 8:

$$\llbracket \vdash s : (p, p') \times (a, a') \to (b, b') \rrbracket = (f, f')$$
$$f : \mathbb{R}^p \times \mathbb{R}^a \to \mathbb{R}^b$$
$$f' : \mathbb{R}^p \times \mathbb{R}^a \times \mathbb{R}^{b'} \to \mathbb{R}^{p'} \times \mathbb{R}^{a'}$$
$$\llbracket \vdash t : (q, q') \times (0, 0) \to (p, p') \rrbracket = (g, g')$$
$$g : \mathbb{R}^q \times \mathbb{R}^0 \to \mathbb{R}^p$$
$$g' : \mathbb{R}^q \times \mathbb{R}^0 \times \mathbb{R}^{p'} \to \mathbb{R}^{q'} \times \mathbb{R}^0$$
$$\llbracket t \circ_p s \rrbracket = \llbracket t \rrbracket \circ_p \llbracket s \rrbracket = (f, f') \circ_p (g, g')$$

The product semantics for explicitly typed parallel composition shown as follows happen according the the model shown in Figure 7 for two parametric lenses:

$$\llbracket \vdash s : (p, p') \times (a, a') \to (b, b') \rrbracket = (f, f')$$
$$f : \mathbb{R}^p \times \mathbb{R}^a \to \mathbb{R}^b$$
$$f' : \mathbb{R}^p \times \mathbb{R}^a \times \mathbb{R}^{b'} \to \mathbb{R}^{p'} \times \mathbb{R}^{a'}$$
$$\llbracket \vdash t : (q, q') \times (c, c') \to (d, d') \rrbracket = (g, g')$$
$$g : \mathbb{R}^q \times \mathbb{R}^c \to \mathbb{R}^d$$
$$g' : \mathbb{R}^q \times \mathbb{R}^c \times \mathbb{R}^{d'} \to \mathbb{R}^{q'} \times \mathbb{R}^{c'}$$
$$\llbracket t \times s \rrbracket = \llbracket t \rrbracket \times \llbracket s \rrbracket = (f, f') \times (g, g')$$

The product semantics for implicitly typed parallel composition shown as follows happen according the the model shown in Figure 4 for two lenses:

$$\llbracket \vdash s : (a, a') \to (b, b') \rrbracket = (p, p', f, f')$$
$$p, p' \in \mathbb{N}$$
$$f : \mathbb{R}^a \to \mathbb{R}^b$$
$$f' : \mathbb{R}^a \times \mathbb{R}^{b'} \to \mathbb{R}^{a'}$$
$$\llbracket \vdash t : (c, c') \to (d, d') \rrbracket = (q, q', g, g')$$
$$q, q' \in \mathbb{N}$$
$$g : \mathbb{R}^c \to \mathbb{R}^d$$
$$g' : \mathbb{R}^c \times \mathbb{R}^{d'} \to \mathbb{R}^{c'}$$
$$\llbracket t \times s \rrbracket = (p + q, p' + q', (g, g') \times (f, f'))$$

The semantics for **switch** are given as follows, first for the implicit type:

$$\begin{gathered}
[\![\vdash s : (a + a', b) \to (a + a', b)]\!] = (f, f') \\
f : \mathbb{R}^{a+a'} \to \mathbb{R}^{a+a'} \\
f' : \mathbb{R}^{a+a'} \times \mathbb{R}^{b} \to \mathbb{R}^{b} \\
[\![\vdash \textbf{switch } s : (a + a', b) \to (a' + a, b)]\!] = (f, f') \\
f : \mathbb{R}^{a+a'} \to \mathbb{R}^{a'+a} \\
f' : \mathbb{R}^{a+a'} \times \mathbb{R}^{b} \to \mathbb{R}^{b}
\end{gathered}$$

Next we give the semantics for **switch** with the explicit type:

$$\begin{gathered}
[\![\vdash s : (0, 0) \times (a + a', b) \to (a + a', b)]\!] = (f, f') \\
f : \mathbb{R}^{0} \times \mathbb{R}^{a+a'} \to \mathbb{R}^{a+a'} \\
f' : \mathbb{R}^{0} \times \mathbb{R}^{a+a'} \times \mathbb{R}^{b} \to \mathbb{R}^{0} \times \mathbb{R}^{b} \\
[\![\vdash \textbf{switch } s : (0, 0) \times (a + a', b) \to (a' + a, b)]\!] = (f, f') \\
f : \mathbb{R}^{0} \times \mathbb{R}^{a+a'} \to \mathbb{R}^{a'+a} \\
f' : \mathbb{R}^{0} \times \mathbb{R}^{a+a'} \times \mathbb{R}^{b} \to \mathbb{R}^{0} \times \mathbb{R}^{b}
\end{gathered}$$

Lastly, **hide**, the type rule to transform an explicit lens into an implicit one is given the following semantics:

$$\begin{gathered}
[\![\vdash s : (p, p') \times (a, a') \to (b, b')]\!] = (f, f') \\
f : \mathbb{R}^{p} \times \mathbb{R}^{a} \to \mathbb{R}^{b} \\
f' : \mathbb{R}^{p} \times \mathbb{R}^{a} \times \mathbb{R}^{b'} \to \mathbb{R}^{p'} \times \mathbb{R}^{a'} \\
[\![\vdash \textbf{hide } s : (a, a') \to (b, b')]\!] = (p, p', g, g') \\
p, p' \in \mathbb{N} \\
g : \mathbb{R}^{a} \to \mathbb{R}^{b} \\
g' : \mathbb{R}^{a} \times \mathbb{R}^{b'} \to \mathbb{R}^{a'}
\end{gathered}$$

Using these semantics, it becomes possible to demonstrate that for every well typed term t, that any such term can be proven to be part of our type, shown by the following theorem:

$$\frac{[\![\vdash t : A]\!]}{[\![t]\!] \in [\![A]\!]}$$

4 Implementation

4.1 Example: Logistic Regression

The semantics shown here for Logistic Regression are based on one of the grammars used in RECIPE [dSPOP17]. This Section mainly serves as an example for how the type system that has been previously defined can be used to model and compose an algorithm and its variants, along with its limitations.

Firstly we will define some of the possible parameters for the generation of the model in question as the following terms:

$$
\begin{gathered}
< boolean > ::= \ 'True' \mid 'False' \\
fit\_intercept ::= < boolean > \\
warm\_start ::= < boolean > \\
max\_iter\_lr ::= \ '10' \mid '100' \mid '150' \mid '300' \mid '350' \mid '400' \mid '450' \mid '500' \\
solver\_lr\_options ::= \ 'liblinear' \mid 'sag' \mid 'newton - cg' \mid 'lbfgs' \\
class\_weight ::= \ 'balanced' \mid 'None' \\
tol ::= ' \ RANDFLOAT(0.0000000001, 0.1)'
\end{gathered}
$$

The boolean term is used twice, to both decide if the model will reuse the previous fit to initialize with warm_start and whether the decision function contains an intercept with fit_intercept. max_iter_lr indicates the maximum number of iterations for testing this model. solver_lr_options picks between 4 different optimization methods. class_weight determines whether to give all classes a weight of 1, or whether to 'balance' the classes by adjusting the weights inversely proportional to class frequency. Lastly, tol determines the stopping criteria.

These parameters are then used to set up the following Logistic Regression model:

$$
\begin{gathered}
[\![LogRegModel]\!] = (f, f') \\
f : \mathbb{R}^2 \times \mathbb{R}^n \to \mathbb{R}^n \\
f([\beta_1, \beta_0], x) = \frac{1}{1 + e^{-(\beta_1 * x + \beta_0)}} \\
f' : \mathbb{R}^2 \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^2 \times \mathbb{R}^n \\
f'([\beta_1, \beta_0], x, v) = (J[f([\beta_1, \beta_0], x)]^T * v, x) = \left(\begin{bmatrix} \frac{\partial f([\beta_1, \beta_0], x_1)}{\partial \beta_1} & \frac{\partial f([\beta_1, \beta_0], x_1)}{\partial \beta_0} \\ \vdots & \vdots \\ \frac{\partial f([\beta_1, \beta_0], x_n)}{\partial \beta_1} & \frac{\partial f([\beta_1, \beta_0], x_n)}{\partial \beta_0} \end{bmatrix}^T \cdot v, x \right)
\end{gathered}
$$

Here, $J[f]$ stands for the Jacobian of the function f, we will go into more detail on what this entails in the next Section.

In Figure 12 we see the 2 different ways the model can be added to through added lenses based on terms such as the ones shown earlier.

Firstly, for changes to the dataset such as the ones made by *warm_start*, we can place a component which takes the data and performs necessary actions on it. In the case of *warm_start*, this would either mean 'none', simply letting model use the data unedited, or 'balanced', which would adjust class weights according to the number of samples containing it [Lea].

Secondly, when the parameters need to be adjusted either before they are given to the model or afterwards to adjust the parameters for the purpose of finding their ideal values, a component composed like the Optimizer in Figure 12 can be used. The main example of this amongst the terms given earlier would be the optimizers given as options for *solver_lr_options*. $S \times P$ side of the lens for the Optimizer represents that certain solvers also require a change in state S to function, but this is not always the case. In those other cases $S \times P$ can be simplified to P.

An example of a component that would be composed in a similar manner to the Optimizer would be one representing the $fit\_intercept$, since this component would only need to make sure the β_0 parameter of the logistic regression model is set to 0.

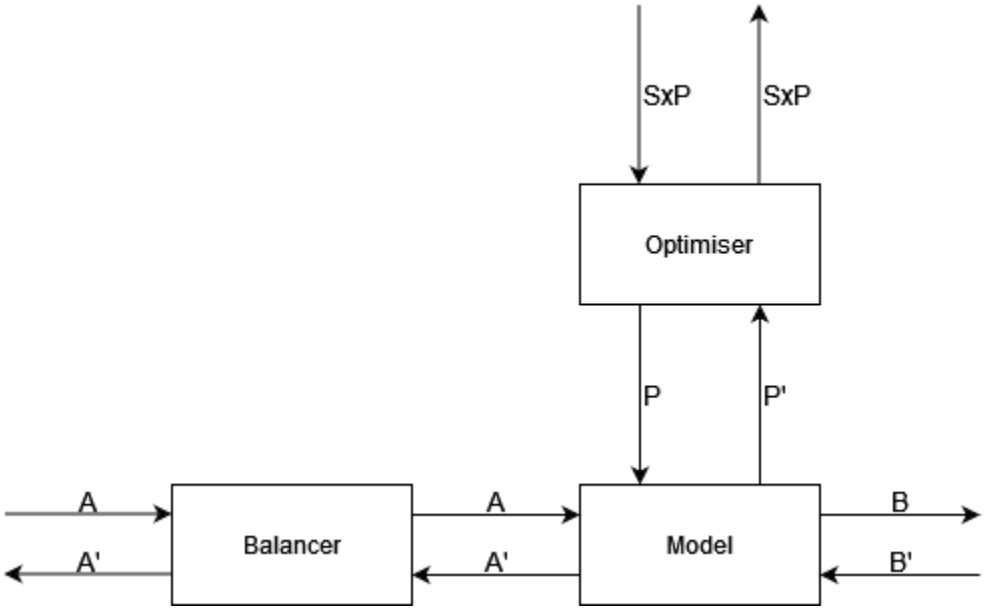

Figure 12: A diagram of the composition of the optimizer and class balancer relative to the model

There are also, however, some terms that cannot be composed with the model itself. Examples from the shown terms are *max_iter_lr* and *tol*, due to both of these not affecting the workings of the model itself, but its end conditions. These terms decide the maximum number of loops for the algorithm and the tolerance of the stopping criteria, respectively.

Due to the way lenses are composed and how they run through the entire composition whenever it is called, it is not possible, or at least undesirable to include components in the model that impact the amount of times the program runs, or otherwise try to interfere with the way the overall model runs instead of adding to the calculations. End conditions and such are best kept track of outside of the type system.

These parameters, along with the logistic regression model, loss and α components allow us to create the following collection of base terms for components in logistic regression:

$$B_{log} = \{LogRegModel, \alpha, loss, fit\_intercept, solver\_lr, balancer\}$$

$$LogRegModel : (p, p') \times (a, a') \to (b, b')$$
$$\alpha : (0, 0) \times (1, 1) \to (0, 0)$$
$$loss : (b, b') \times (b, b') \to (1, 1)$$
$$fit\_intercept : (0, 0) \times (p, p') \to (p, p')$$
$$solver\_lr : (p, p') \to (p, p')$$
$$balancer : (a, a') \to (a, a')$$

4.2 Linear Regression

For linear regression over n functions in the form $y = a \cdot x + b$, the type of the Model would be $(1 + 1, 1 + 1) \times (1 \times n, 1 \times n) \to (n, n)$, due to using only a and b as parameters and having only a single x parameter, meaning the number of x values for which is solve must be as large as the vector y values. This then simplifies to $(2, 2) \times (n, n) \to (n, n)$.

With this in mind, the Model component for such a linear regression pipeline can be defined for our type system as follows, creating a smoother matrix as the data it returns from its reverse function [CGG+22]:

$$[\![LinRegModel]\!] = (f, f')$$
$$f : (\mathbb{R}^2 \times \mathbb{R}^n) \to \mathbb{R}^n$$
$$f([a, b], x) = a * x + b * \vec{1}$$
$$f' : (\mathbb{R}^2 \times \mathbb{R}^n) \times \mathbb{R}^n \to (\mathbb{R}^2 \times \mathbb{R}^n)$$
$$f'([a, b], x, v) = (J[f([a, b], x)]^T * v, x) = \left(\begin{bmatrix} \frac{\partial f([a,b],x_1)}{\partial a} & \frac{\partial f([a,b],x_1)}{\partial b} \\ \vdots & \vdots \\ \frac{\partial f([a,b],x_n)}{\partial a} & \frac{\partial f([a,b],x_n)}{\partial b} \end{bmatrix}^T \cdot v, x \right)$$

In the above, $J[f(x, a, b)]$ stands for the Jacobian of matrix f, which is the matrix of all first-order partial derivatives of a multi-variable function.

For the purpose of regression the transpose of this Jacobian multiplied by the errors in the outputs given by the reverse side of the loss function in v gives the error in each individual parameter, allowing for correction of the individual weights. For the output of f', since the data does not need to be altered for linear regression, x is returned unchanged while the errors in the parameters are output for use in an optimizer component.

The loss component can be defined for this model of linear regression as follows, applying quadratic error as is standard for a smoother [CGG+22]:

$$[\![Loss]\!] = (loss, loss')$$
$$loss : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$$

$$loss(x_t, x_p) = \frac{1}{2} * \sum_{i=1}^{x} ((x_p)_i - (x_t)_i)^2$$

$$loss' : (\mathbb{R}^n \times \mathbb{R}^n) \times \mathbb{R} \to (\mathbb{R}^n \times \mathbb{R}^n)$$

$$loss'(x_t, x_p, n_\alpha) = (n_\alpha * x_t, n_\alpha * (x_t - x_p))$$

Here, x_t stands for the true values the expression would equal to, while x_p stands for the predictions given by the model. Lastly n_α stands for the learning rate. In this case, since no changes are made to the true output values, it is returned unchanged through the parameter space, while for the input space it is instead returned as the difference between true and predicted times the learning rate.

The next component to be added is the simple component giving the alpha, whose only purpose for this pipeline, which uses a constant for the learning rate instead of dynamically altering it, is to return said learning rate in its reverse function. It is given the following definition:

$$[\![Alpha]\!] = (alpha, alpha')$$

$$alpha : \mathbb{R}^0 \times \mathbb{R} \to \mathbb{R}^0$$

$$alpha(p, a) = 0$$

$$alpha' : (\mathbb{R}^0 \times \mathbb{R}) \times \mathbb{R}^0 \to (\mathbb{R}^0 \times \mathbb{R})$$

$$alpha'(p, a, b) = (0, n_\alpha)$$

The inputs p, a and b are irrelevant in this implementation of the learning rate. n_α is to be replaced by whatever constant is decided upon to serve as the learning rate. For our later practical implementation this is consistently set as 0.001. The empty brackets indicate both that the output given here is unused and that said values are intended to be empty lists in later implementations shown.

The last component to be added to the linear regression model is the final step of the gradient descent, updating the parameters that were originally given to the model. The definition for this component is as follows:

$$[\![Grad]\!] = (grad, grad')$$

$$grad : \mathbb{R}^2 \times \mathbb{R}^0 \to \mathbb{R}^2$$

$$grad(p, a) = p$$

$$grad' : (\mathbb{R}^2 \times \mathbb{R}^0) \times \mathbb{R}^2 \to (\mathbb{R}^2 \times \mathbb{R}^0)$$

$$grad'(p, a, b) = (p + b, 0)$$

Like with the alpha component, the empty brackets indicated that the output into the input space for the reverse function grad' is an empty list, so as not to interfere when this component is parametrically composed with the model. The forward function is not used for this component, simply forwarding the parameters, while the reverse function is used to update the parameters with the results of reverse function in the model, slight getting the values of the parameters closer to their true fit.

Now that all components necessary for the linear regression pipeline are defined, it will be shown how these components are composed together. This starts with the parametric composition of LinRegModel and Grad:

$$[\![Grad \circ_p LinRegModel : (p, p') \times (a, a') \to (b, b')]\!]$$
$$(p \times a) \to b : (\mathbb{R}^2 \times (\mathbb{R}^0 + \mathbb{R}^n)) \to \mathbb{R}^n = (\mathbb{R}^2 \times \mathbb{R}^n) \to \mathbb{R}^n$$
$$(p \times a) \times b' \to (p' \times a') : (\mathbb{R}^2 \times (\mathbb{R}^0 + \mathbb{R}^n)) \times \mathbb{R}^n \to \mathbb{R}^2 \times (\mathbb{R}^0 \times \mathbb{R}^n) = (\mathbb{R}^2 \times \mathbb{R}^n) \times \mathbb{R}^n \to (\mathbb{R}^2 \times \mathbb{R}^n)$$

The empty input from the Grad component means that the final dimensions of the composition end up the same as the dimensions for the LinRegModel. For the sake of brevity $[\![Grad \circ_p LinRegModel]\!]$ will be shortened to $[\![GradModel]\!]$ from this point.

The composition for the GradModel and Loss components can now be defined as follows, based on the base terms, the semantics given earlier and the definitions for the components:

$$[\![Loss \circ GradModel : (p, p') \times (a, a') \to (b, b')]\!]$$
$$(p \times a) \to b : ((\mathbb{R}^2 \times \mathbb{R}^n) \times \mathbb{R}^n) \to \mathbb{R}$$
$$(p \times a) \times b' \to (p' \times a') : ((\mathbb{R}^2 \times \mathbb{R}^n) \times \mathbb{R}^n) \times \mathbb{R} \to ((\mathbb{R}^2 \times \mathbb{R}^n) \times \mathbb{R}^n)$$

Finally, we can obtain the completer pipeline by composing the Alpha component sequentially to the previous composition, note that the sequential composition represented by $\circ$ is right associative.:

$$[\![Alpha \circ Loss \circ GradModel : (p, p') \times (a, a') \to (b, b')]\!]$$
$$(p \times a) \to b : ((\mathbb{R}^2 \times \mathbb{R}^n) \times \mathbb{R}^n) \to \mathbb{R}^0$$
$$(p \times a) \times b' \to (p' \times a') : ((\mathbb{R}^2 \times \mathbb{R}^n) \times \mathbb{R}^n) \times \mathbb{R}^0 \to ((\mathbb{R}^2 \times \mathbb{R}^n) \times \mathbb{R}^n)$$

The shape of this composition can be shown using the following diagram, showing a model for linear regression in this type system:

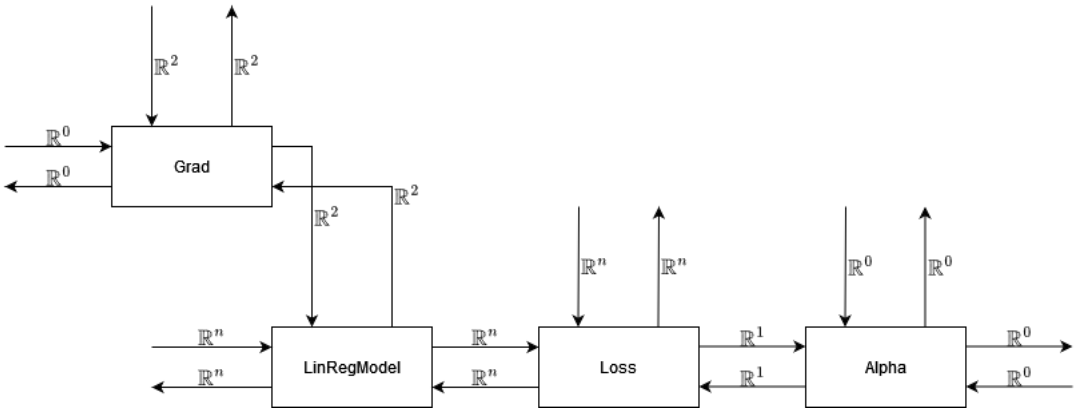

Figure 13: A diagram of the composition of the model built for linear regression.

4.3 Haskell Implementation

An implementation of the type system was made in the Haskell programming language to show its workings [1]. The reason this language was chosen is due to the ease with which it allows type systems to be translated into its code.

In the implementation vectors are used as the type for the lenses, due to vectors allowing for lists that are ordered and easily expanded, both requirement for the implementation of composition for parametric lenses. It additionally uses Haskell's arrow class in order to manage the computation of the composition, making sure that the correct data is given as arguments to the correct components. Some simple tests are included to show the results of calculation through the lenses.

The aforementioned repository also contains an implementation of linear regression according to the definition given earlier, in order to demonstrate its ability to model machine learning algorithms.

Additionally, a similar Haskell implementation was made for this type system using the haskell-tensorflow library [Ten]. It mainly features a test for the type system through implementing the linear regression model as described earlier[2]. Sadly, this implementation encountered an incompatibility between our type system and tensorflow. Namely, the type system requires it to be possible to concatenate and split its datatype into potentially empty or singleton variants, which is not allowed when using tensors in tensorflow.

After this attempt to implement a linear regression model failed to work, we made a second attempt to implement it, which ended up being far more successful, this implementation can be found in the initial repository for the type-system's implementation[3].

The implementation of linear regression is implemented in accordance with the linear regression implementation described in 4.2. As a result we can thus showcase that this type system is capable of, at the very least, implementing simple regression algorithms.

4.4 Results

The implemented model first generates one hundred random values for x, which is then used to generate the same number of values for y in a linear equation $y = a \cdot x + b$, where 'a' and 'b' are predefined constants. Then, the x values are given to our linear equation model as input, while a set of two zeros are given for the parameters, representing the slope and intercept that the linear regression model seeks to calculate, which should end up being calculated closer to their given equivalents 'a' and 'b'. Alongside these parameters, the true values for y are fed to the model through the parameter space, for the loss function.

When the model is run enough times, updating the slope and intercept with each run, a close approximation of their true values is found, within three decibels. We can thus showcase that this type system is capable of implementing, at the very least, linear regression as a machine learning algorithm.

[1] https://github.com/HuibSprangers-leiden/ML-Type-System

[2] https://github.com/HuibSprangers-leiden/type-system-tensorflow

[3] See footnote 1

5 Conclusions and Further Research

In this thesis we have created a type system designed for machine learning and given it specifications as well as a framework for its use in Haskell.

It has been shown that the type system can be used for linear regression, but more types of algorithms will need to be implemented and tested for better judgment of the system's merits and limits, as well as to be able to create a true AutoML framework for creating pipelines. Implementations of neural networks, for instance, can be made in a similar manner to specifications by Crutwell et al.'s python implementation of their categorical system [CGG+22]. Besides this, the algorithm can also be tested on its performance on particular datasets, to truly compare it to other algorithms.

One possible avenue of further research to improve this type system would be refinement types [RKJ08] [VBJ15], which is a kind of type that dynamically changes its function based on certain bounds for any given input. An example of this would be the following types:

$$\begin{aligned} type\ IntPos &= \{v : Int \mid 0 < v\} \\ type\ IntGE\ x &= \{v : Int \mid x < v\} \end{aligned}$$

Here, IntPos represents a type of integer $(v : Int)$ which is bound to always be positive $(0 < v)$ and otherwise give a type error. Similarly, IntGE is a type that is instead defined as an integer bound to always be greater than a certain value x. These types can then be used to verify certain criteria as follows:

$$\begin{aligned} addPos &:: Int \to IntPos \to Int \\ plus &:: x : Int \to y : Int \to \{v : Int \mid v = x + y\} \\ plusPos &:: x : IntPos \to y : IntPos \to \{v : Int \mid v = x + y\} \end{aligned}$$

The first of these theoretical functions, addPos, is used to add an integer from the IntPos type to a regular integer, but does not verify whether its output is a still a valid IntPos. The plus function adds two regular integers together, and uses refinement to verify its output, creating a new bound. Lastly, plusPos adds two IntPos together, guaranteeing that both integers given as input are positive, while also giving as output an integer whose bound is now given as being the addition of the input. Through functions such as this, bounds are kept dynamic, creating self-checking types.

Using refinement types can allow for automatic verification of input and output, a valuable property in machine learning, where bad data being inserted can lead an algorithm to give inaccurate results. For the type system defined in this thesis, refinements types could be used to validate the input and output of the various components.

Another possibility for further improvement lies with inhabitation machines [AB17, BDS13]. These are automata designed to be able to process an infinite alphabet of terms through storing data in its registers. Because of the properties of automata it is able to allow terms to loop into themselves, allowing complex traversals through the terms that make up the machine. This way, it is possible that inhabitation machines can be used to automate the exploration of ML pipelines.

In conclusion, while this type system is still in its initial stages and requires more testing, it should provide a possibility for designing machine learning pipelines that are more easily implemented, modified and checked for errors, in accordance with the merits of type systems.

References

[AB17] Sandra Alves and Sabine Broda. Inhabitation machines: determinism and principality. In *Workshop on Non-Classical Models for Automata and Applications*, 2017.

[BDS13] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013.

[CGG+22] G. Cruttwell, B. Gavranović, N. Ghani, P. Wilson, and F. Zanasi. Categorical foundations of gradient-based learning. In *European Symposium on Programming*, 2022.

[dSPOP17] A. de Sá, W. Pinto, L. Oliveira, and G. Pappa. Recipe: A grammar-based framework for automatically evolving classification pipelines. In *20th European Conference on Genetic Programming (EuroGP'17)*, 2017.

[Fla12] P. Flach. *Machine learning: the art and science of algorithms that make sense of data*. Cambridge University, 2012.

[FST19] Brendan Fong, David Spivak, and Rémy Tuyéras. Backprop as functor: A compositional perspective on supervised learning. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, 2019.

[Goo] Google. Tensorflow. https://www.tensorflow.org/ [Accessed 26-9-2023].

[Lea] Scikit Learn. Logistic regression. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html [Accessed 26-9-2023].

[OBUM16] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 485–492, New York, NY, USA, 2016. ACM.

[RKJ08] P. Rondon, M. Kawaguci, and R. Jhala. Liquid types. *ACM SIGPLAN Notices*, 43(6):159–169, 2008.

[Ten] Tensorflow. Tensorflow-haskell. https://github.com/tensorflow/haskell [Accessed 26-9-2023].

[VBJ15] Niki Vazou, Alexander Bakst, and Ranjit Jhala. Bounded refinement types. *CoRR*, abs/1507.00385, 2015.