Inferring Arithmetic Expressions from Data

Siamak Hajizadeh (s.hajizadeh@student.tudelft.nl)
MSc thesis under supervision of Dr. David M.J. Tax

Delft University of Technology
Departement of Electrical Engineering, Mathematics, and Computer Sciences
July 6, 2012

Preface

This work presents a framework for learning arithmetic expressions from a set of observations. Our intention is to introduce a Bayesian method for what is known as *equation discovery*. Our method is based on measuring a degree of belief (posterior probability) for a set of hypothesized expressions to find those which best explain the observed data. This measure is used as the basis for choosing one hypothesis over another. In our work we distinguish two tasks in the process of equation discovery, namely: the task of exploring the space of arithmetic expressions and that of evaluating the degree that an expression describes the data. Separating these two, allows us to investigate them independently.

For the first task, we use a context-free grammar to construct a large set of expressions which we take as our *hypothesis space*. The set contains a large number of hypotheses (each an arithmetic expression) that should be tested against the data. We also evaluate complexity of for each expression using the grammar. The complexity is presented to the model in the form of a prior probability. Our main focus here is the second task: the posterior evaluation using a Bayes formulation. The method tests a hypothesized expression against a set of provided samples that have quantitative input features. It calculates a likelihood probability which expresses the degree that a hypothesis describes the data. A final posterior probability is calculated based on the prior and the likelihood, that is the measure of qualification for each expression.

Contents

Chapter 1

Introduction

1.1 General problem description

If one is provided with a set of data samples, each with one or more features as input values and one output value, one may be interested in a model that when provided with the inputs, can predict the output. This is a very general idea behind many regression and classification tasks (see for example [16, 10, 28]). What is common between many of methods in the field of machine learning is that they learn weights, parameters, or adjustments for a predefined structure. Take any learning algorithms: Gaussian processes [21], Support Vector machines [5, 9], etc. for example; the learner has one fixed structure that updates it (i.e. its parameters, etc) when it is exposed to observations (Gaussian processes adjust their distributions, while SVM, applying a kernel, searches for maximum margins). Many of prevalent machine learning approaches look for adjustments of the tool model that they already have, in order to best predict the outputs.

However, there is another way to posit the prediction task under a different type of learning: the learning of concepts. In a domain of data samples, a concept defines a subset of these samples that share a common property [2, 20, 23]. *Concept* thus by itself is a very general notation that can be reduced to more concrete specifications. In the domain of folk biology for example, given the set of all animals the subset including all birds defines the concept *bird*, while *animal* is a more general concept that includes birds as well as the rest of animal kinds. See [24, 13] for a wealth of similar instances.

A concept can also be a logical relation (i.e. one with logical variables and operators) [27, 12, 8]. This relation is sometimes defined as a rule among the Boolean (or Fuzzy [4]) elements of an input vector [23, 7]. For the domain of folk biology, “Has wings” and “Lays eggs” are two Boolean features that have the value *True*, and “Breastfeeds” is an example of one with *False* value, for the class of birds. A rule defining the concept of *bird* then can be stated as:

$$\text{“(Has wings)} \wedge \text{Lays eggs} \wedge \text{not(Breastfeeds)”.}$$

Goodman *et al.* in [7] propose a method for learning such formulas (as rules) [1] that is based on Bayesian rule learning. We will discuss more details on this as it is one of the main sources that this work is based upon.

Here, we look to apply a similar method to the domain of arithmetic expressions (as compared to logical expressions). Our goal is to find an expression such that it fits best to a set of observations. These observations include a set of quantitative inputs and one quantitative or labeled output. A solution is an arithmetic expression that is a function from the input set to a resulting output. The closer that result is to the actual output, the better fit is that expression. In case of observations with labeled outputs, we do not have the result as a numeric value. We will later explain the challenges we face in this regard.

We define a grammar for the task of generation of arithmetic expression hypothesis space. Our grammar is a standard arithmetical expression grammar that parses simple expressions consisting of basic math operators. The grammar is limited to addition, subtraction, multiplication, division, constant-valued powers, and brackets. It is always possible to extend the grammar to parse expressions that contain other, perhaps more complex operations.

There is a drawback to this way of using the grammar however, and that is the more symbols the grammar contains the wider becomes the range of producible expressions. As a result, it requires more time and space to produce all expressions up to certain depth of the parse tree. We will introduce some possible improvements on exploration of the hypothesis space later in chapter 4. These approaches are not implemented in our work. Here we only confine ourselves to a time (or iteration count) criterion, to produce a reasonable (but not exhaustive) number of expressions as our hypothesis space. We then remove redundant expressions and we estimate constant values inside generated expressions using a Simplex optimization module, all as steps of generating the hypothesis space.

By the means of the grammar we also calculate a complexity for each expression and we use it as the prior probability to the Bayesian posterior evaluation. Also a likelihood probability is calculated for each expression

[1]By *formula* we infer the same as *rule*, *expression*, and *function*. Throughout this document, they are often used interchangeably.

that expresses its ability to explain the observed data. We finally calculate a posterior probability for all expressions. We call this process *Bayesian posterior evaluation.* Based on this evaluation, we judge a *relative* degree they fit to the data. Figure 1.1 shows an overview of the steps taken for achievement of the posterior values.

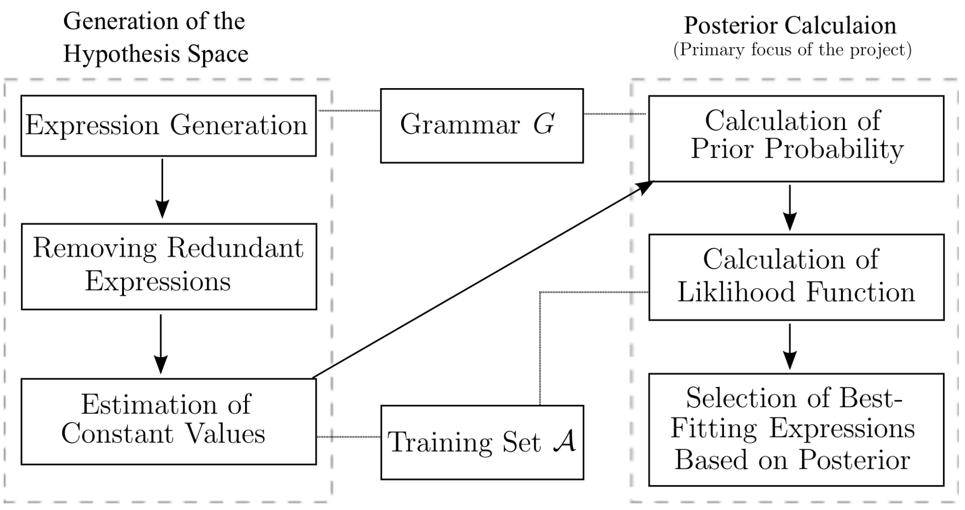
Generation of the Hypothesis Space
Posterior Calculaion
(Primary focus of the project)
Expression Generation
Removing Redundant Expressions
Estimation of Constant Values
Grammar G
Training Set $\mathcal{A}$
Calculation of Prior Probability
Calculation of Liklihood Function
Selection of Best-Fitting Expressions Based on Posterior

Figure 1.1: Figure shows general step taken for construction of the hypothesis space and calculation of the posterior values per expression. A grammar G and a set of observed samples $\mathcal{A}$ (the training set) are given to the model.

We have kept the two phases illustrated in Figure 1.1, as much independent as possible. This allows the exploration of the hypothesis space to be studied as a separate problem. The posterior calculation phase is an evaluator for any generated hypothesis, no matter how these hypotheses are produced. As we shall discuss, our method of exploring the hypothesis space is a naive one. We suggest some other alternative ways for exploration of the hypothesis space later. Our main target of interest during this work hass been the second phase: the *posterior calculation.*

In real world, there are cases where a relation function is adjusted to explain the behavior of a response variable with regard to a number of explanatory variables. An instance of this is when a researcher has gathered a number of observation from a physical phenomenon and is now trying to understand what might be the mathematical relation that binds the input to the output variables. Our method is a general framework that can be applied to similar domains of same idea for what is known as *Equation discovery.* We use hand-seeded data for both training and testing purposes in all of our experiments. It should be mentioned that our data is noisy. This is an important assumption that we make while looking for the expression which best describes the data. In fact without this assumption, the problem reduces to simply looking for the expression that exactly predicts most of the observations.

1.2 Related works

Our approach toward discovering arithmetic expressions is sometimes called *learning of rules*. We see arithmetic expressions as rules (similar to Boolean expression example) that binds the the input and output observations. There has been a few other equation discovery techniques in the literature. The field is introduced by pioneering Langley et. al. [14] who used heuristics to extract the equations from data. Also, we mostly benefit from the work by Todorovski et. al. [26] where they introduce a grammar to parse arithmetic expressions. They argue that because their domain of expression is limited to equations that are specific to a certain domain (i.e. equations of a certain structure applied to a certain type of data), a biased grammar helps them search only a sub-domain of all expressions. That sub-domain can be proportionally very small compared to the domain of all expressions parsed by a non-specific grammar. They call this speacial grammar the *declarative bias*. We however, allow all types of expressions using a general grammar.

In their work Todorovski et. al. also use simplex optimization for evaluation of the constants in produced equations. This is again, what we do with all arithmetic expressions in the hypothesis space. We evaluate the constants with regard to a error function that is to be minimized. Unlike their approach though, our work is not limitted to a special type of equations. It can be applied to any arithmetic expression only by adding the necessary mathematical operators to the grammar.

The rule-based view point in our work is inspired by the work of Goodman et. al. [7]. They use a set of features (or criteria) to specify the range of sub-region in the problem space. For the birds example, these can be a pair of "tapered wings" or a "cone-like beak" with a "short-sized neck", which limit to birds such as sparrows and warblers.

There are a number of other approaches that build rules of similar structures. The well-known Nosofsky *et al.* RULEX model [19] for instance, is designed based on the idea that humans often learn rules and *memorize* cases that are exceptions. They present a general scheme that searches for a general rule describing a concept. It then adds exceptions (themselves as rules) to the general rule, as it encounters with the exceptions over which a compliance failure occurs.

Because our framework takes many ideas from them, let us now describe how Goodman *et al.* have proposed their own model of rule-based concept learning. In general, they take the rules to be a disjunctive normal form (DNF) logical expression with Boolean-valued features as binary variables. If "the ith feature of a concept C has value equal to 1" then it can be noted as: $f_i(C) = 1$. A formulation such as $(f_1(C) = 0 \wedge f_2(C) = 1) \vee (f_2(C) = 0 \wedge f_3(C) = 1)$ is a basic definition of a concept. In order to generate the logical expressions (each a hypothesized concept) that shape the hypothesis space, Goodman suggests a context-insensitive grammar that can derive DNF formulas. Starting from an initial symbol, each formula is resolved to a number of predicates and then to terminal symbols.

The grammar represents the whole hypothesis space in a concise structure. In fact, having such automaton is like having the whole set of hypotheses. What remains up to the learner given the samples, is that it should try to find the fittest hypothesis; here a Boolean expression. We use the same idea in our domain using a grammar that parses arithmetic expressions. In both cases, we search for a parse that most probably identifies the concept containing all the samples.

Having the hypothesis space expressed by the means of the grammar, next is to calculate a prior probability and a likelihood function. Since the grammar can run infinitely, the expressions while always well formed (and meaningful), can grow too large and sophisticated to be expressive. Goodman et. al. suggest that this process needs to be in control. They induce this control (as we do) by the means of the prior probability calculation which favors simpler expressions. This priori is a measure of complexity for each Boolean expression and it is based on the derivation steps that leads to formation of that expression. Each derivation in their grammar, has a certain probability of being selected. These probabilities are defined in a way to be *least informative* [11]. That is, when there is no reason to believe that one derivation is more probable that another, equal probabilities are assigned to them. The final prior probability for an expression thus, is directly affected by the number of derivation that has resulted in that expression. The more the number of derivations and hence the complexity, the less is the the final prior probability evaluation for that expression.

Another influential factor in Goodman's work is the likelihood function. The likelihood function measures the degree a hypothesized expression explains the observed data. It is defined formally for a hypothesized expression, as the probability of observing the data we have observed, given that the expression is actually the true underlying function. We also use a similar definition for evaluating our arithmetic expressions in our work.

For the rest of this report, we first start by describing how we construct our hypothesis space using a defined context-free grammar and our approach to calculation of a Bayesian posterior in Chapter 2. In Chapter 3 we will perform some experiments on the framework and evaluate it from different aspects. Finally we conclude in Chapter 4 and also suggest some future directions of further improvements and extensions.

Chapter 2

Learning Arithmetic Expressions as Rules

This chapter explains our approach of learning arithmetic expressions from the provided training data. Our method involves generation of a hypothesis space of expressions and calculating a measure of expressiveness for all expressions while regarding the data.

Having observed a set of samples $\mathcal{A} = \{s_i = (\mathbf{x}_i, y_i); i = 1, \cdots n\}$ of numeric feature values we assume that the data follows an arithmetic relation that holds between the input feature vector $\mathbf{x}_i$ and the output y_i. We search a hypothesis space of arithmetic expressions for such relation. We hold the assumption that the observations are meant to be sampled from one *true* relation f according to which input vector $\mathbf{x}_i$ relates to the output value y_i. To find the a best fitting expression, we introduce a Bayesian evaluation model that identifies these expressions from a set of hypothesized arithmetic expressions called the *hypothesis space*. Such space is characterized by a grammar G that parses arithmetic expressions of some certain complexity.

We start by construction of the hypothesis space using the grammar. Later we calculate the prior probability and the likelihood given the training set. Calculation of posterior value per expressions is an independent task of the hypothesis space and hence, independent of the grammar. It is therefore possible to use the idea over other similar rule-based domains. It should be noted that our approach is mainly inspired by the work of Goodman et. al. [7] which propose a similar framework for the domain of Boolean expressions.

By means of the grammar we construct the hypothesis space $\mathcal{H}$ of arithmetic expressions. Each expression $r \in \mathcal{H}$ is regarded as a plausible hypothesis that might explain the observed data. Using the model, we look to estimate this plausibility as probability $P(r \mid \mathcal{A})$ that expresses the degree of belief in that the expression r is in fact that underlying true relation f. We assume that all observed samples in training set comply with the true function f (with an added noise) and therefore *positively exemplify* the concept rule that is expressed by f. This is sometimes called *strong sampling* [25]. In this chapter we first explain the generation of a hypothesis space which is our approach to exploration of the domain of arithmetic expressions. We then continue to the details of the training phase.

2.1 Generation the hypothesis space

We use a context-free grammar to generate a large set of expressions. Each expression consists of variables, operators, and constant coefficients. The number of variables in the grammar is the same as the number of variables in data. Our grammar also locates constant coefficients in expressions. Figure 2.1 shows the CFG grammar that we use. Both constant value terminal symbol (`c`) and non-terminal variable symbol `V` can be directly derived from non-terminal `T` which is an indicator of a term. By replacing a term with a constant coefficient `c`, the grammar determines that a constant value should be inserted somewhere in the expression.

$$
\begin{aligned}
\mathtt{E} &\rightarrow \mathtt{F} \mid -\mathtt{F} \mid \mathtt{E}+\mathtt{F} \mid \mathtt{E}-\mathtt{F}\\
\mathtt{F} &\rightarrow \mathtt{T} \mid \mathtt{F}*\mathtt{T} \mid \mathtt{F/T}\\
\mathtt{T} &\rightarrow \mathtt{V} \mid \mathtt{c} \mid \mathtt{V}\hat{}\,\mathtt{c} \mid (\mathtt{E}) \mid (\mathtt{E})\hat{}\,\mathtt{c}\\
\mathtt{V} &\rightarrow \mathtt{x_1} \mid \mathtt{x_2} \mid \cdots \mid \mathtt{x_n}
\end{aligned}
$$

Figure 2.1: A transition function-based representation of the context-free grammar to parse and generate arithmetic expressions is illustrated. The left hand side of each transition arrow ($\rightarrow$) is a non-terminal symbol in the grammar. The right hand other side of the arrow list all possible productions of that non-terminal, separated by a $\mid$ sign. `E` is the so-called initial symbol of the grammar.

Besides a constant, a non-terminal `V` can reduce to any of the variable that each correspond to one of the sample features in the dataset. It can also reduce to a variable to the power of a constant (`V^c`), an expression in brackets and an expression to the power of a constant. The grammar starts parsing by the initial symbol `E` which indicates an expression. An expression is the divided to a number of factors that are added or subtracted. Factors in turn are made from multiplication and division of terms.

2.1.1 The construction process

As described, our hypothesis space consists of a set of arithmetic expressions. We generate these expression using the grammar. The algorithm used for the generation, traverses the grammar tree replacing each non-terminal by all its possible derivations. Figure 2.2 represents this algorithm.

```
Begin
    input Grammar $G$;
    output Set of expressions as hypothesis space $\mathcal{H}^*$;
    define Queue $Q$;
    $Q \leftarrow \{< \texttt{E} >\}$;
    repeat
        $s \leftarrow Q$.retrieve-from-head();
        if $s$ has a non-terminal sybol then
            $n \leftarrow$ first non-terminal symbol in $s$;
            for all $t \in$ Productions($n$, $G$) do
                $s' \leftarrow$ replace $n$ in $s$ with $t$;
                $Q$.add-to-end($s'$);
            end for
        else
            Add $s$ to $\mathcal{H}^*$;
        end if
    until time criterion is met;
    return $\mathcal{H}^*$;
End

function PRODUCTIONS(non-Terminal $n$, Grammar $G$)
    return a set of strings containing all right-hand-sides of productions of $n$ in $G$;
end function
```

Figure 2.2: Figure shows the algorithm which uses a queue to breadth-first search (BFS) the domain of simple arithmetic expressions, by exploring the grammar (Figure 2.1) tree. Until some time criterion, it repeatedly retrieves a string s from the head of the queue, extracts a non-terminal symbol n of that string and replaces it with all its immediate derivations.

Initially a symbol E (in the form of a string) is added to the container. Then iteratively a string is picked out of the container and one of non-terminal symbols in that string is chosen. Finally the non-terminal is replaced by all its immediate products and all resulting strings are added back to the container. At each iteration, if the string picked from the container does not have a non-terminal symbol in it (which indicates that it consists of only terminal) then it is a valid arithmetic expression and is added to the final list. These steps continue until some time criterion is met. The container used in the algorithm is a queue. It is used to retrieve a string from the head and append derived expressions to its end. This implements the *breadth-first* exploration in which as the algorithm runs further, more complex expressions are generated. It is worth mentioning here that a *depth-first* implementation is of no use here, simply because the grammar tree has an infinite depth and the algorithm will never get out of the first branch it enters.

There is a second method for exploring the grammar that assures all expressions with parse tree up until a certain depth are contained in the final results. Instead of adding only results from derivation of one non-terminal, in this method all non-terminals of the picked string are replaced by their immediate products in all combinations. For instance if a selected string is `F` $*$ `T` which has two non-terminals `F` and `T` (with 3 and 5 products), that set of all combinations will have 15 strings in it, all ultimately added to the end of the queue. Figure 2.3 shows this algorithm in more details.

```
Begin
    input Grammar $G$, integer $depth$;
    output Set of expressions as hypothesis space $\mathcal{H}^*$;
    define Queue $Q$;
    $i = 0$;
    $Q \leftarrow \{< \texttt{E} >\}$;
    repeat
        $s \leftarrow Q$.retrieve-from-head();
        if $s$ has a non-terminal sybol then
            $N \leftarrow$ the set of all non-terminal symbols in $s$;
            for all $n \in N$ and $t \in$ Productions($n$, $G$) do
                $S' \leftarrow$ all replacement combinaions of nonterminals;
                $Q$.add-to-end($S'$);
            end for
        else
            Add $s$ to $\mathcal{H}^*$;
        end if
        $i = i + 1$;
    until $i > depth + 1$;
    return $\mathcal{H}^*$;
End

function PRODUCTIONS(non-Terminal $N$, Grammar $G$)
```

```
    **return** a set of strings containing all right-hand-sides of productions
  of $N$ in $G$;
  **end function**
```

Figure 2.3: Similar to 2.2 the algorithm here performs a BFS on the grammar tree, with the exception that pushes all non-terminal symbols of a retrieved string forward one step at a time. In this way, it guarantees that all expressions up to a certain depth are produced and added to the final list.

By applying both of these algorithms a primary list of arithmetic expressions is created. Afterwards, a refinement procedure removes expressions that are redundant (i.e. have an equivalent expression in the list in any of the ways mentioned above). This procedure uses MATLAB Symbolic Math toolbox for determination of equivalence between the two expressions. We also use a hash table data structure to be able to quickly look for equivalent expressions and remove the redundancies.

2.1.2 Drawbacks of these algorithms

Generally speaking, the idea to have a grammar to produce the hypothesis space can be a practical way of exploring the hypotheses for many domains. For example the domain of Boolean expressions is a fine target for such way of solution generation. For the domain of arithmetical expressions though, this may be inefficient. There are two main reasons for that. Firstly, unlike Boolean expressions that can be converted to standard normal forms, there is not any normal form for math expressions. This complicates the generation process in that many redundancies can occur while generating the expression. For example, $x_1 \times (x_2 + x_3)$ is mathematically equivalent to $x_1 \times x_2 + x_1 \times x_3$. However, the two expressions are not grammatically equivalent. Another type of redundancy occurs when constant coefficients are let inside the expressions. As described, the grammar locates coefficients into the expressions. These coefficients are fitted later using an error minimization technique. As a result this, it can happen that two different expressions become equivalent. To give a simple example, $x_1 \times c$ and x_1/c are equal despite mathematical inequality. If c in the first expression is fitted to a value v then the second c will be fitted to $1/v$ due to them both being optimized for the same data. This is not an issue with Boolean expressions, since a grammar can be used to only parse expressions in disjunctive normal form (DNF).

There is a second problem that arises when using a grammar to produce the the set of expressions that is our hypothesis space. The number of expressions that can be generated is huge even when we want to limit ourselves to a certain depth. For the grammar above this number gets to the order of 5.5×10^{21} when all expressions with parse tree of depth less than or equal to 8 are retrieved [26]. This results a time consuming process for generation of the hypothesis space and compels us to limit ourselves to expressions of certain complexity (with parse tree up to certain depth). Although the target of this work is not to introduce an efficient method of arithmetic space exploration, we will propose some improvements to this process later in future directions section.

2.2 Estimation of the constant values

Having the list of expressions purified off the redundant ones, our method then includes training and testing phases. The training phase consists of estimation of constant values, calculation of the likelihood function, and calculation of the posterior value per expression. The two latter steps alongside the priori calculation, are parts of the Bayesian posterior evaluation for promoting fittest hypotheses (expressions) given the training data. The testing phase consists of exposing the trained module to the test data. We will investigate some tests and experiments in Chapter 3. For now we continue by describing the training phase.

2.2.1 Dataset structure

We continue by describing the structure of our target datasets that are used for both training and testing phases. For both tasks of training and testing we use a number of samples. The training samples are a subset of dataset samples that we use for training steps (e.g. evaluation of constant coefficients inside expressions and calculation of likelihood). The rest is used for testing of the model. We already denoted this set by $\mathcal{A}$. Each sample is represented by a number of inputs (features) denoted as a vector $\mathbf{x}_i$ and and an output value y_i. One such pair $(\mathbf{x}_i, y_i)$ locates a point in the quantitative problem space. As mentioned, we assume that there exists a true function f that relates $\mathbf{x}_i$ to y_i. The point since being a positive example of it, lies somewhere near the target function that we are searching for and we refer to as $f(\mathbf{x}_i)$. Because the data has noise, the samples seldom comply fully with the true function:

$$\begin{gathered} y_i = f(\mathbf{x}_i) + \varepsilon_i \\ \text{where} \\ \varepsilon_i \sim \mathcal{N}(0, \sigma_{\text{noise}}) \end{gathered} \tag{2.1}$$

The noise ε has a Gaussian distribution with mean equal to zero and standard deviation equal to σ_{noise}.

2.2.2 Simplex optimization

First step of the training phase in our framework is the estimation of the constant coefficients that has been placed inside the formulas by the grammar. Before the expressions can be used for Bayesian posterior evaluation and testings, the constant values must be determined so that the expressions become meaningful. Testings and likelihood evaluation (a part of Bayesian posterior evaluation) need the expressions to only leave feature variables unassigned, so that they would be assigned from the dataset.

For evaluation of the constants we use Nelder and Mead (down-hill) Simplex algorithm [17] which is a widely used [26] nonlinear optimization technique. For optimization of an expression $r(x_1, , x_n, c_1, , c_m)$ (where x_is are the input variables and c_js are constant values that are to be estimated), the following sum of errors is minimized.

$$\min_{c_1 \dots c_m} \sum_{\mathcal{A}} |y_i - r_{\mathbf{x}_i}(c_1, ..., c_m)| \tag{2.2}$$

Here $r_{\mathbf{x}_i}(c_1, ..., c_n)$ is the expression $r(x_{i1}, , x_{in}, c_1, , c_m)$ with $\mathbf{x}_i$ assigned from samples in $\mathcal{A}$. The Nelder and Mead minimization module[1] then starts from an initial random assignment and iteratively adjusts the c_j's until either a maximum number of iterations is reached or the error measure has converged. In both case the final results are substituted inside expressions in place of the constant coefficients.

2.2.3 Constant evaluation with labeled samples

In addition to quantitative datasets, we use another type of datasets in our study and that is when numeric output values y_i's are replaced with labels l_i's which merely indicates whether the output of $f(\mathbf{x}_i)$ is positive or negative.

[1]We use Flanagan's Java library [6] to compute down-hill Simplex algorithm for minimization.

$$l_i = \text{sign}(f(\mathbf{x}_i) + \varepsilon_i), \quad \varepsilon_i \sim \mathcal{N}(0, \sigma_{\text{noise}}) \tag{2.3}$$

We use Simplex for constant estimation, in case of labeled data as well. Here, the function to be minimized is the number of samples that the predicted outcome $\text{sign}(r(\mathbf{x}_i))$ does not match the actual label l_i. Since the sign function returns 1 and -1 for positive and negative respectively, the summation in equation 2.2 does not change much for the case of labeled data:

$$\min_{c_1 \ldots c_m} \sum_{\mathcal{A}} |l_i - r_{\mathbf{x}_i}(c_1, ..., c_m)| \tag{2.4}$$

If a predicted label matches the true label, then the absolute value in the summation results a zero and otherwise results 2. Therefore, the summation is minimized when highest number of matches occur which is what we need. It is worth mentioning that although the Simplex module performs better on some types of functions (e.g. smooth, continuous, and with fewer local optimums), it is independent of the type of function that it is minimizing [17]. In cases that the minimization function is a non-smooth and non-continuous (as in equation 2.4) the module explores the parameter space by taking large steps from a random initialization. Depending on the amount of variation in the target function, the steps are adjusted as the solutions start to converge.

2.3 Evaluation best fitting hypothesized expressions

After estimation of the constant values in the expressions, we can now apply the Bayesian model to select bests among all expressions. The Bayesian posterior probability calculation involves calculation of the prior and likelihood probabilities, and, calculation of posterior based on *Bayes rule*. Our work is mainly inspired by the Bayesian approach in [7]. We start by describing the general Bayesian solution to the fittest hypothesis. We then discuss how we evaluate the prior probability and why we do it in this way. Finally, likelihood function and the difference in calculation of it, when dealing with label or numeric output data are presented.

2.3.1 Bayesian posterior evaluation

Given a set of samples (denoted by $\mathcal{A}$) that positively exemplifies the true arithmetic relation f underlying the samples, we are in search of an expression r that best fits the data. We described that having created a hypothesis space of expressions (hypotheses) our goal is to find the fittest among them. To achieve this, posterior probability for each expression in the hypothesis space is evaluated. The main goal of this work is to have posterior evaluation tuned in a way that it is actually a good measure of such compliance to the training data. We use Bayes rule to evaluate the posterior values. Equation 2.5 shows Bayes rule application in this context.

$$\begin{aligned} P(r \mid \mathcal{A}) &= \frac{P(\mathcal{A} \mid r)P(r)}{P(\mathcal{A})} \\ P(r \mid \mathcal{A}) &\propto P(\mathcal{A} \mid r)P(r) \\ \log P(r \mid \mathcal{A}) &\propto \log P(\mathcal{A} \mid r) + \log P(r) \end{aligned} \tag{2.5}$$

The first equation is the Bayes rule that relates the priori $P(r)$ and the likelihood function $P(\mathcal{A} \mid r)$ to the posterior $P(r \mid \mathcal{A})$. Since the denominator of the first equation is independent of the hypothesis that is being tested, it plays no role in comparison of posterior values of expressions and thus can be omitted. Calculation of prior and likelihood values often tends to result very small values. By taking the sum of logarithms we avoid dealing with such small numbers and subsequent data truncations in computations.

The posterior probability can be interpreted as an overall estimate of the degree of belief that expression r is actually the underlying relation f of the data. It is constituted of two factors. One is a priori knowledge about the hypotheses and how plausible is their occurrence. We intuitively find a large and complex expression less authentic compared to a rather simple one. It is also true that if two expressions describe the data equally well we prefer the simpler one over the more complex one.

The second factor is the likelihood function. Particularly, $P(\mathcal{A} \mid r)$ given that the expression r is in fact the underlying true function, is the probability that we observe all the samples we have in $\mathcal{A}$. That is, regardless of specifications of an expression, the likelihood expresses the degree that the expression explains the data. By combining these two factors the posterior value of an expression is influenced by both its qualifications regarding only the domain knowledge, and its expressiveness regarding the observations.

2.3.2 Prior probability

For each expression to be derived by the means of the grammar, a series of productions (each with single non-terminal as its left-hand and a string of terminals and non-terminals as its right-hand side) are applied (see Figure 2.1). Suppose that we want to generate an arithmetic expression. This starts from a string containing only the initial symbol `E` and then continues by randomly selecting a production rule for each non-terminal in the string. The process runs until no non-terminal symbol remains in the string. Since all productions of a non-terminal, have equal probabilities of being selected, this probability will be one divided by the number of productions. Because every string is uniquely parsed (i.e. it is result of a unique sequence of derivations) for a string r to be generated, a certain set of production rules (denoted as Δ_r) are applied. Therefore, the probability of r to be the result of our random walk, is equal to multiplication the probability of all productions δ in Δ_r.

$$P(r) = \prod_{\delta \in \Delta_r} P(\delta) \tag{2.6}$$

Replacement of a non-terminal by its production is called *transition*. Selection of equal probabilities for all productions of a non-terminal is set due to the principle of indifference [11]. It states that in case there is no reason to favor one output of an event over another, selecting the least informative prior is the best choice. The transition probability distribution is a uniform distribution over all possible outcomes of a non-terminal to favor all equally.

A consequence of such prior is that more complex expressions get less probability of being generated than simple ones. An expression (string) with higher number of transitions, have more probabilities multiplied together and thus its prior probability has shrunk more. In the set of expressions that we generate, there are expressions that are arithmetically equal while being syntactically different. We remove these expressions and we only let the simplest expression to remain. We also calculate the probability only based on the simplest expression, among all sets of arithmetically equal expressions. Figure 2.4 shows an example of prior probability calculation for a simple expression $\mathtt{c} * \mathtt{x}_1 + \mathtt{x}_2$.

In Figure 2.4 all derivation steps for a simple expression are illustrated. The final prior probability will be equal to $0.25^2 \times 0.33^3 \times 0.2^3 \times 0.5^2 \simeq 4.49 \times 10^{-6}$. As observable in the figure the larger the parse tree grows the more smaller-than-one numbers are multiplied into the prior probability.

$$\begin{tikzcd}
 & & \mathtt{E} \arrow[d, "0.25"] & & \\
 & & \mathtt{E} + \mathtt{F} \arrow[dl, "0.25"'] \arrow[dr, "0.33"] & & \\
 & \mathtt{F} \arrow[d, "0.33"'] & & \mathtt{T} \arrow[dr, "0.2"] & \\
 & \mathtt{F} * \mathtt{T} \arrow[dl, "0.33"'] \arrow[dr, "0.2"] & & & \mathtt{V} \arrow[d, "0.5"] \\
\mathtt{T} \arrow[d, "0.2"'] & & \mathtt{V} \arrow[d, "0.5"] & & \mathtt{x}_2 \\
\mathtt{c} & & \mathtt{x}_1 & &
\end{tikzcd}$$

Figure 2.4: Figure shows probabilistic derivation of expression $\mathtt{c} * \mathtt{x}_1 + \mathtt{x}_2$. Multiplying all single step transition probabilities will result prior probability of the whole expression.

There are however some problems with this method of prior probability calculation. We use prior probability as a measure of complexity of the expressions. The idea is to favor simpler hypotheses opposed to the more sophisticated ones. Using a grammar to determine such simplicity is not an immediately clear way to determine the arithmetic behavior of function though. For example prior probability for expression $\mathtt{c} * \mathtt{x}_1 * \mathtt{x}_2$ with the same method of calculation will be $0.25 \times 0.33^3 \times 0.2^3 \times 0.5^2 \simeq 1.8 \times 10^{-5}$. Arithmetically though this expressions is not less complicated than the one in Figure 2.4, if not more.

As a result this measure of prior calculation is not precise. However, it fulfils our requirements to some extent. We want the two above-exampled expressions to be equal in measure of simplicity and they are in a way, approximately. The difference between them becomes insignificant when we use log calculations for final evaluation of posterior value. Compared to a large expression with several terms and hence with a very small prior probability these simple expressions are still favored enough in terms of simplicity. It is however, still not precise.

2.3.3 The likelihood function

We discussed that the likelihood is a measure of expressiveness of an expression regarding the set of observed samples. As the number of observed samples (i.e. the size of the training set) grows the likelihood tends to decrease. We will first demonstrate how the likelihood is calculated based on one observation (sometimes called a stimulus). For now lets focus on the case that the output feature is a numeric value. If the observed sample $s : (\mathbf{x}, y)$ lies exactly on the sub-space of expression r we expect the likelihood probability to be at its maximum. This occurs where r completely complies with s. In contrast, the further s is located from the sub-space corresponding to r, the less becomes the likelihood $P(s \mid r)$. To model this reverse relation with distance, we use a Gaussian distribution:

$$\begin{aligned}
P(s \mid r) = P((\mathbf{x}, y) \mid r) &= P(y \mid \mathbf{x}, r) \\
&= \mathcal{N}_{r(\mathbf{x}),\hat{\sigma}_{\text{noise}}}(y) \\
&= \frac{1}{\hat{\sigma}_{\text{noise}}\sqrt{2\pi}} \exp\left(-\frac{(y - r(\mathbf{x}))^2}{2\hat{\sigma}^2_{\text{noise}}}\right)
\end{aligned} \tag{2.7}$$

where $\mathcal{N}_{r(\mathbf{x}),\hat{\sigma}_{\text{noise}}}$ is the Gaussian distribution function with mean and standard deviation equal to $r(\mathbf{x})$ and $\hat{\sigma}_{\text{noise}}$ respectively. Figure 2.5 illustrates an example where $\mathbf{x}$ is assumed to be a vector of only one input.

Figure 2.5 shows an example case where only one feature is shown as x'. Notice that the point (x', y') is not located exactly on the top of the true function f. This is due to the noisiness of the outcome values in the dataset. The shaded bell in the graph is the Gaussian probability density function $\mathcal{N}_{r(x'),\hat{\sigma}_{\text{noise}}}$ which is centered on $r(x')$ and has the standard deviation equal to $\hat{\sigma}_{\text{noise}}$. As the distance between $r(x')$ and y' grows the probability density decreases. The highest density occurs when the distance between the two point becomes zero.

What is important to note here, is that we choose $\hat{\sigma}_{\text{noise}}$ a constant value, regardless of the dataset on which we are conducting the experiments. As a result, the likelihood probabilities are not accurate measurements of the exact probabilities and are not to be referred to judge utility of a single hypothesized expression. If one would need such information, one would have to find a way to estimate σ_{noise} by some method first and then use it in calculation of the likelihood. Here the likelihood is only a basis for comparing hypothesized expressions. Since we take a training set and evaluate likelihood for all expressions over that set with same $\hat{\sigma}_{\text{noise}}$, the resulting likelihood values are a good metric for comparison. If we were to estimate the noise variance σ_{noise} from the data, one way would be to formulate it using outcome values of samples with similar input features. When there are a rather large number of samples provided for training, there are statistical methods that can help choosing a better estimation for σ_{noise} than merely putting a constant value in its place. However since the current approach

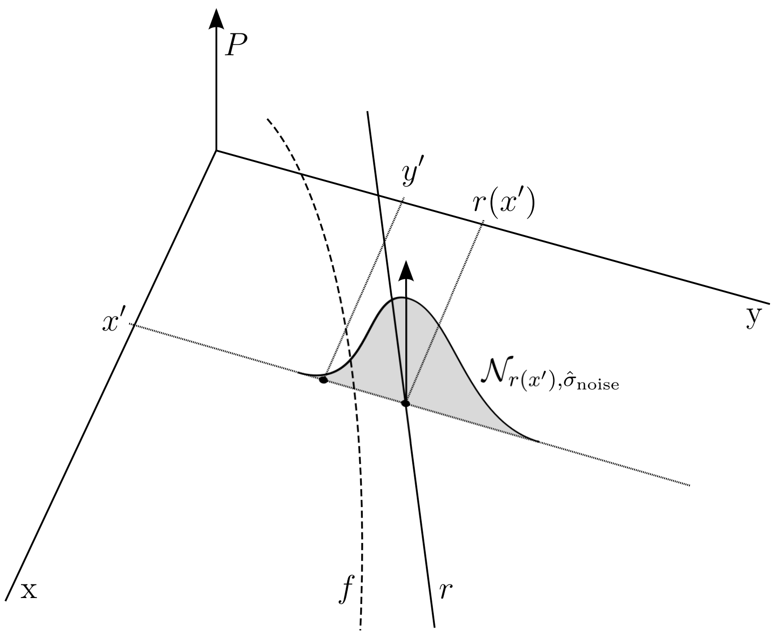

Figure 2.5: Figure shows the probability distribution of the likelihood around the predicted output value $r(x')$ where r is the hypothesized expression. The vertical axis P represents the probability density. Note that output value y' does not lie on top of the true function f because of the noise that we assume we have in our observation.

fulfills our goal for the case of multiple training samples as well, here we do not investigate any such methods. We will suggest some improvement to this process in our future directions.

2.3.4 Extending likelihood to the whole training set of samples

The likelihood is calculated with a similar logic when there are more than one sample provided in the training set. In this case again it interprets to the probability that we observe all the samples given that r is the true function. We hold the assumption that observing each of the samples is independent of observation of the other samples[2]. Because of this, we simply multiply all probabilities to evaluate the overall probability of the observing all samples.

[2]This might not be true in general. Here a set of all observation are provided to the learner at once. Since they are all supposed to be positively exemplifying the function that we are looking for, we do not estimate any sort of joint probability for occurrence of the specific samples. In many learning methods (e.g. prediction with time series, etc.) these assumptions does not hold.

$$P(\mathcal{A} \mid r) = \prod_{s_i \in \mathcal{A}} P(y_i \mid \mathbf{x}_i, r) = \prod_{s_i \in \mathcal{A}} \mathcal{N}_{r(\mathbf{x}_i),\hat{\sigma}_{\text{noise}}}(y_i) \tag{2.8}$$

Again it should be noted that this multiplication is valid based on the simplifying assumption that observing a sample s_1 does not change the probability of observing another sample s_2, or in general: $P(s_i \mid s_j, r) = P(s_i \mid r)$ for all $i, j \in \{1, , n\}$.

2.3.5 Likelihood of the labeled data

As we explained, this method of calculation of likelihood has the advantage that it allows dealing will noise and possible outlying samples. We however, use another type of datasets that do not contain a numeric output value and this introduces a problem when using the same method for evaluation of the likelihood. Here instead of numeric outputs, each samples s_i has an output label l_i which comes from a binary set of possible labels $\{-1, 1\}$ indicative of negative and positive outputs. The predicted outputs are also labels -1 and 1. As a result the likelihood will be either equal to 1 when there is a match between the predicted output label $r(\mathbf{x})$ and the actual label l, and 0 when there is not a match. For the case of one sample we will have:

$$P(s \mid r) = P(l \mid \mathbf{x}, r) = [l = r(\mathbf{x})] = \begin{cases} 1 & \text{if } l = r(\mathbf{x}) \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

and extending it to a set of samples (the training set):

$$P(\mathcal{A} \mid r) = \prod_{s_i \in \mathcal{A}} P(l_i \mid \mathbf{x}_i, r) = \prod_{s_i \in \mathcal{A}} [l_i = r(\mathbf{x}_i)] \tag{2.10}$$

where $[Q]$ is the Iverson bracket defined as:

$$[Q] = \begin{cases} 1 & \text{if } Q \text{ is true;} \\ 0 & \text{otherwise.} \end{cases} \tag{2.11}$$

The problem becomes apparent in the case that we extend the calculation of the likelihood to whole number of samples in the training set. It will take only one mismatch (for a single sample) between the predicted and actual labels for the whole calculation of the likelihood to produce a zero. Thus it is either 1 when the expression r has predicted all the labels truly, or 0 otherwise. There is no way to compare the expressions with partially correct predictions using this method. Also since the learner is not provided with noise-free data, a full set of correct predictions is rare.

To solve this we use the approach by Goodman et. al. [7] in which they allow a certain probability of outlying-ness for each sample. This is a way that allows expressions to explain part of the training set and yet remain plausible as a hypothesis. In the case of numeric output values the Gaussian density function works as a similar mechanism that is based on the difference of the predicted and actual outcomes. Here though, since no such difference estimate is available, this method helps avoiding exclusion of possibly well-fitting expressions.

To arrange this into the model, a probability of e^{-b} is subsumed for sample to be an outlier. Therefore, the likelihood probabilty for the case that there is only one sample will be multiplied by a the probability $1 - e^{-b}$ which then defines the probability of non-outlying-ness.

$$P(s \mid r) = P(l \mid \mathbf{x}, r) = (1 - e^{-b})[l = r(\mathbf{x})] \tag{2.12}$$

This function now result 0 in case of a mismatch, and $1 - e^{-b}$ when the two label are the same. Now for extending this over the whole training set, we avoid resulting a zero by only multiplying the likelihood over the set of samples $\mathcal{S}$ which are non-outliers. First, let us demonstrate how the likelihood probability evaluates while $\mathcal{S}$ is given. This is the case that we assume the set of non-outlying samples is known. Equation 2.13 shows this.

$$P(\mathcal{A} \mid \mathcal{S}, r) = \prod_{s_i \in \mathcal{S}} P(s_i \mid r) = \prod_{s_i \in \mathcal{S}} P(l_i \mid \mathbf{x}_i, r) = \prod_{s_i \in \mathcal{S}} [l_i = r(\mathbf{x}_i)] \tag{2.13}$$

Note that since we keep that assumption that all samples in $\mathcal{S}$ are non-outliers (and that all samples in $\mathcal{A} - \mathcal{S}$ are outliers) we do not involve the probability of outlying-ness here anywhere. However, because we have no prior knowledge on which samples to take as non-outliers[3] we have to av-

[3]If we had such knowledge we would not need to assign a probability of outlying-ness to each sample.

erage equation 2.13 over all subset $\mathcal{S}$ of $\mathcal{A}$ by marginalizing $\mathcal{S}$ out of the equation 2.13.

$$
\begin{aligned}
P(\mathcal{A} \mid r) &\propto \sum_{\mathcal{S} \subseteq \mathcal{A}} (e^{-b})^{|\mathcal{A}-\mathcal{S}|}(1-e^{-b})^{|\mathcal{S}|} P(\mathcal{A} \mid \mathcal{S}, r) \\
&= \sum_{\mathcal{S} \subseteq \mathcal{A}} (e^{-b})^{|\mathcal{A}-\mathcal{S}|}(1-e^{-b})^{|\mathcal{S}|} \prod_{s_i \in \mathcal{S}} [l_i = r(\mathbf{x}_i)] \\
&= \sum_{\mathcal{S}^* \subseteq \mathcal{A}_{[l=r]}} (e^{-b})^{|\mathcal{A}-\mathcal{S}^*|}(1-e^{-b})^{|\mathcal{S}^*|} \\
&= e^{-b|\mathcal{A}-\mathcal{A}_{[l=r]}|}
\end{aligned}
\tag{2.14}
$$

Here $|\mathcal{S}|$ indicates the cardinality of a set $\mathcal{S}$, etc. We start by summing over all subsets of $\mathcal{A}$ taking each as the set of non-outliers at a time. For proportionality in the first line to become an equality, one can divide the right hand side by total number of sum of all weights $(e^{-b})^{|\mathcal{A}-\mathcal{S}|}(1-e^{-b})^{|\mathcal{S}|}$ over all subsets $\mathcal{S} \subseteq \mathcal{A}$. However, we do not need to normalize the summation through dividing it by a total sum denominator, because such denominator is not dependent to r. In the third line of equation 2.14 the subsets $\mathcal{S}^*$ are selected not from the whole training set $\mathcal{A}$ but only from the set of samples s_i that the predicted label $r(\mathbf{x}_i)$ match the actual label l_i. We denote this set by $\mathcal{A}_{[l=r]}$. Because any other subset of $\mathcal{A}$ which is not a subset of $\mathcal{A}_{[l=r]}$ as well, will produce a mismatch between that predicted and actual labels and result to zero in production and so add nothing to the summation. The final step, which significantly reduces the computation load, follows from the Binomial Theorem [7, 1]. It is worth noting that $\mathcal{A} - \mathcal{A}_{[l=r]}$ represents the set of all samples of which the predicted and actual labels do not match.

To show how this method of likelihood calculation helps selecting an expression that better explains the data, we continue by an example. Assume that the learner is provided with a set of 10 samples, we compare the likelihood evaluation for three hypothesized expressions r_1, r_2 and r_3. Table 2.1 shows the actual versus the predicted labels of the three expressions.

The first expression r_1 predicts the second, the sixth and the ninth sample incorrectly, whereas the second expression r_2 predicts all correctly except the seventh one, and the third expression predicts only three of them correctly. Therefore $|\mathcal{A}-\mathcal{A}_{[l=r_1]}|$, $|\mathcal{A}-\mathcal{A}_{[l=r_2]}|$ and $|\mathcal{A}-\mathcal{A}_{[l=r_3]}|$ will be equal to 3, 1 and 7 respectively, each representing the count of incorrect predictions. Thus the final measure of likelihood for $b = 5$ will become 3.06×10^{-7}, 6.73×10^{-3}, 6.31×10^{-16}. We mentioned earlier that to evaluate the final posterior we take log of the likelihood and prior values (see equation 2.5). Having done

Table 2.1: Actual labels vs. predicted ones of three hypothesized expressions: r_1, r_2 and r_3.

i	1	2	3	4	5	6	7	8	9	10
l_i	1	1	1	1	1	1	-1	-1	-1	-1
$r_1(\mathbf{x})$	1	-1	1	1	1	-1	-1	-1	1	-1
$r_2(\mathbf{x})$	1	1	1	1	1	1	1	-1	-1	-1
$r_3(\mathbf{x})$	-1	1	-1	-1	1	-1	1	1	1	-1

so, likelihood contribution to the postrior, in case of labeled samples reduces to the linear term $-b|\mathcal{A} - \mathcal{A}_{[l=r]}|$. In the experiments chapter, we will show how we manage to choose a proper b. It is worth mentioning here though, that since b is positive, the relation can be simply explained as: the higher the number of mismatching samples grows, the less becomes the likelihood.

A question that comes to the mind at this point (for the case of labeled data) is that if the likelihood is just a measure to *compare* the expressions regarding the observed data, why bother going through these formulation steps and not only take the number of matched samples as a metric of likelihood. There are two reasons for this. Firstly, we want our approach for the case of labeled data to be consistent with the case of samples with numeric outputs. Assigning a probability of outlying-ness makes sense in terms of *likelihood probability* whereas merely counting the number of correctly predicted samples does not make such sence at least immediately. Secondly, although we do not practice it here, the notion of outlying-ness can be developed from a fixed constant to a dynamically estimated variable as an improvement for the future. In such case the integrity of the framework still persists.

2.3.6 Estimation of constants with labeled data

We discussed how calculation of the likelihood for the case of labeled data differs from the case of data with numeric output values. We also discussed earlier that how we evaluate constants of expressions when we are provided with labeled data, using the Simplex optimization. This leads us to the question that is it not necessary to apply the same rationale here (i.e. let some samples to be outliers). If there exists an outlying sample in the training set which we use for both tasks (constants and likelihood), it can influence the estimation of constants as well. One might suggest applying weights to samples from the training set in the function that we use for minimization of error (equation 2.4) for constant estimation.

The answer to this question is that applying such weights introduces no improvements to the Simplex optimization, because the weights will inevitably cancel from the formulation. Let us define the probability (weight) of a sample being an outlier to be P_0. Then the probability of a sample not to be an outlier will be $P_1 = 1 - P_0$. Also from the equation 2.4 we denote $|l_i - r_{\mathbf{x}_i}(c_1, ..., c_m)|$ as briefly d_i. The minimization target function of equation 2.4 will be:

$$\begin{aligned}
\sum_{\mathcal{S}\subseteq\mathcal{A}} (P_0)^{|\mathcal{A}-\mathcal{S}|}(P_1)^{|\mathcal{S}|} \sum_{\mathcal{S}} d_i = {} & \underbrace{P_0^{n-1}P_1 d_1 + \cdots + P_0^{n-1}P_1 d_n}_{\binom{n}{1}} + \\
& \underbrace{P_0^{n-2}P_1^2(d_1+d_2) + \cdots + P_0^{n-2}P_1^2(d_{n-1}+d_n)}_{\binom{n}{2}} + \\
& \cdots + \underbrace{P_1^n(d_1+d_2+\cdots+d_n)}_{\binom{n}{n}} \\
= {} & P_0^{n-1}P_1\left(\sum_{\mathcal{A}} d_i\right) + \\
& (n-1)P_0^{n-2}P_1^2\left(\sum_{\mathcal{A}} d_i\right) + \cdots + \\
& P_1^n\left(\sum_{\mathcal{A}} d_i\right) \\
= {} & \left(\sum_{i=1}^{n} \frac{i}{n}\binom{n}{i} P_0^{n-i}P_1^i\right)\sum_{\mathcal{A}} d_i
\end{aligned} \tag{2.15}$$

The first term in final line is a constant that is independent of the d_i and thus plays no role in the minimization of the whole function. It is also similarly justified that the same holds with numeric output values. Therefore, no improvement can be made in estimation of the constants, by assigning an outlying-ness probability to the training samples.

2.4 Testing our model

After completing the training step, all expressions are tested for their accuracy of prediction using 40 test samples. We use training sets of 20 samples maximum. The calculation process usually takes hours long above that size.

All expressions are evaluated by substituting their variables by input vector from a sample in test set. For the case of quantitative values a tolerance threshold is fixed that determines the range the outcome result of an expression can deviate from the one provided in the dataset. We set the threshold equal to the lowest noise variance among our datasets, which is 2.5. Since our aim here is only comparison we do not go into steps of finding optimal tolerance for each dataset. We define *accuracy* of an expression to be the percentage of correct prediction when tested over all samples of the test set. For datasets with quantitative outputs, we use a constant threshold of 2.5 to determine *correct prediction.* In case of labeled output, this correctness is simply defined as whether the predicted output label matches the one from the dataset.

Having measured the accuracy of each expression over the test data, the next step is to extract a list of top accurate expressions and finally compare it to a list of expressions with highest posteriors. The first list is simply obtained by testing all expressions and sorting them according to their accuracy. The second is likewise, obtained by calculating the posterior for all expressions and sorting them, this time according to their posterior probability. A number of *top* expressions are selected from both lists and are compared. The degree to which the two lists match is what we define as measure of *performance.* We do not compare the ordering of all expressions in the first list with that of the second list. This is due to accuracy falling to zero for the rest of expressions, after a number of top accurate expressions selected. Therefore it makes no sence to compare all thousands of expressions, where most of them have neither a significant accuracy nor a significant posterior.

2.4.1 A measure of similarity between the ordered lists of expressions

To determine the degree to which the lists of actual top accurate expressions and the list of expressions with highest posterior probability match we need a measure. There are techniques in the literature for comparing two ordered lists. They do not however exactly do the sort of comparison that we use. Yilmaz et. al. [29] for example, compare two orderings of a same list, whereas in our case, the two lists have intersection that is only covers parts of them both. Here thus, we propose a heuristic measure. At its ideal case, the expressions are ranked by the method in exact order as they accurately predict the data. That is, the list of expression sorted based on posterior probability matches exactly to the list of expressions sorted by their accuracy of prediction. Having a measure for this is a way of estimating the performance of our technique. It indicates to what degree those expressions that are marked as good fits of the data are actually good in predicting them.

The measure is based on two factors. Firstly it accounts for the degree to which the orderings of the expressions match each other across the two lists. It keeps track of the fact that a match on the most accurate expression in the lists (say top 5) is of higher importance than a match on less accurate ones (say top 20). It is more important to order accurate hypotheses correctly that it is to order the non-accurate one. Secondly, it considers that a match becomes more important if the difference between the accuracies of the best and worst expressions in the list of top accurate expressions is high. If such the difference is not significant, then the ordering of the posterior list and the way it matches to the accuracy list is less important either.

We denote the top expressions in the list ordered by the accuracy percentage with U and we call it the *accuracy set.* The cardinality of U is $|U|$. We denote the top $|U|$ expressions in the list ordered by the posterior probability with T and reffer to it as the *posterior set.* The two criteria described above are contained in the following performance measure:

$$M = \sum_{i=1}^{|U|} m_i \frac{D_{\max}}{D_i} \tag{2.16}$$

where $U = \{u_1, , u_{|U|}\}$ and m_i is the percentage to which a subset of the first i elements in $T = \{t_1, , t_{|U|}\}$ (the posterior set) covers the first i elements of U. Therefore it is simply the size of the intersection of the top i elements from the two sets. $D_{\max}$ is equal to $D_{|U|}$ and, D_i is the difference between the accuracies of the largest and the smallest elements in the first i elements of U. We define:

$$\begin{aligned} m_i &= |\{u_1, ..., u_i\} \cap \{t_1, ..., t_i\}|/i \\ D_i &= \text{max-accuracy}(\{u_1, ..., u_i\}) - \text{min-accuracy}(\{u_1, ..., u_i\}) \end{aligned} \tag{2.17}$$

In our formulation, m_i represents the first factor that takes account of the degree to which the orderings of the expressions match each other across the two sets. Note that the summation assures that the higher ranked are more important than the lower ranked. It does so by separately adding the matching ratio m_i for the first i expressions from the top, and to iteratively increasing i. The second factor is D_i which accounts for the difference between the accuracies of the first i elements in accuracy set. Consequently,

$D_{\max}$ is the distance between the top and the bottom of the whole accuracy set. The ratio $D_{\max}/D_i$ then is the inverse of the normalized D_i. This ratio is high when top i expressions have a small difference among themselves, meaning that they differ much from the rest of expressions and so have higher importance. The ratio is at its lowest (i.e. equal to 1) when the distance of the top i expressions are roughly the same as the overall distance from best (most accurate) to worst of the accuracy set. Hence their significance becoming small.

To avoid division by zero in Equation 2.16, a small value can be added to both numerator and denominator of the fraction. Also to represent the measure as a percentage that expresses the amount of match between the two lists, and to make it independent of the number of expressions in the lists, we divide the value M in equation 2.16 by the maximum value possible when the lists are identical. The maximum for m_i values in equation 3.1 is 1 and it corresponds to the case that the lists contain exactly the same elements and same orders. Thus we have:

$$M_{\max} = \sum_{i=1}^{|U|} \frac{D_{\max}}{D_i} \tag{2.18}$$

To clarify how the measure is calculated, we continue with an example. Assume we have 9 expressions: r_1 to r_9 with accuracies on a given dataset, equal to $r_1 : 0.70$, $r_2 : 0.65$, $r_3 : 0.60$, $r_4 : 0.55$, $r_5 : 0.50$, $r_6 : 0.45$, $r_7 : 0.40$, $r_8 : 0.35$, and $r_9 : 0.30$. Also assume that where the accuracy set is equal to $U = \{r_1, r_2, r_3, r_4, r_5, r_6\}$ in descending order ($|U| = 6$), the posterior set has been calculated as $T = \{r_2, r_7, r_1, r_8, r_4, r_9\}$, again in decreasing order of posterior probability. Table 2.2 shows how the measure of similarity between the two sets is calculated.

Table 2.2: Example calculation of similarity measure between the accuracy and posterior sets.

i	1	2	3	4	5	6
m_i	0	0.5	0.666	0.5	0.6	0.5
D_i	0.01	0.05	0.10	0.15	0.20	0.25
$D_{\max}/D_i$	25	5	2.5	1.666	1.25	1

In the example, $D_{\max} = 0.25$. Finally we calculate $M = 6.25$ according to equation 2.16 and we divide it by $M_{\max} = 36.416$ (from equation 2.18) to obtain the value $M/M_{\text{max}} = 0.17$.

We test the performance of our framework in the next chapter. The measure M/M_{max} is used throughout chapter 3 to evaluate performance upon different settings. We perform a set of experiments and analyse the results using the measure. In all experiments the final results and numbers are average values obtained from 3-fold cross valiadations.

Chapter 3

Experiments and Results

This chapter presents an evaluation of our method. The goal is to understand to what extend our proposed solution is able to discover fit expressions from a set of many generated expressions which shape our hypothesis space. Also we are interested in how different parameter values and other setting influence our performance. We investigate this trying to answer three main questions about how the proposed method works. To answer each question we have designed and run experiments of different settings and parameterizations. We first describe the way the experiments were implemented and what steps we have taken to build the system and its compounding algorithms which has enabled us to put the theory into test. Next we present the outcomes and test results and try to interpret them.

3.1 General overview of experiments

All experiments and settings are implemented using Java and MATLAB. They consist of hypothesis space generation, training, and testing. In this section view describe details of each of these steps.

3.1.1 Construction of the hypothesis space

Our hypothesis space for all experiments described in following sections, consists of a set of expressions that are generated using the context-free grammar described in 2.1. Our algorithm uses a queue to traverse the space of all parse trees. Starting from the initial non-terminal of the grammar, at each step the string in the head of the queue is retrieved and one of its non-terminal symbols is replaced by all immediate derivations of that symbol. Then all the newly generated strings are put at the end of the queue. This process continues until some space or time criteria are met. The algorithm is implemented in way that it is possible to assure that all expressions of some certain parse tree depth are added to the queue. The reason for this is to make sure that the hypothesis space contains expressions of a certain complexity. It is however not a practical approach for extracting more complex expressions of higher parse tree depths because of the exponential space and time growth. Our hypothesis space therefore is limited mostly to fairly simple expressions.

After the generation step, there are many expressions that although are syntactically different, are mathematically equal. It would be a difficult task to adjust the grammar in way to generate only expressions that are not semantically redundant and yet exhaustive. We choose to check for redundancies and remove them after all expressions are generated by the algorithm. This is what we do using the MATLAB symbolic math toolbox[1]. Using this toolbox, generated expressions are simplified to their expanded form and compared. After redundancies are omitted, prior probability values are calculated by the means of the parse tree for all the rest of expressions. A procedure traverses the parse tree of each expression in breath-first order and acquires the final prior probability by multiplying all branching probabilities together.

For our experiments we used two grammars of 3 and 4 variable terminal symbols. Besides the variables which take their values from datasets during training and testing phases, the grammar also locates constant values in expressions. These constants are then evaluated in training phase.

3.1.2 Training and calculation of the posteriors

As described in the previous chapter training includes two steps. Firstly using a down-hill simplex algorithm [18] we evaluate the constant values given the training samples from the dataset. To do this, variables in an expression are substituted by quantitative values from the training set. Then each constant symbol in the expression string is treated as a target variable. The Simplex minimizes the difference between the evaluation of the expression with substituted values and the target output from the dataset. For calculation of the Simplex minimization we use M. T. Flanagan's Java scientific library. After best estimates of the constant values are retrieved, they are replaced in the expressions and saved per dataset, number of training samples, and number of grammar variables.

The second step is calculation of likelihood and posterior values. Rather

[1]http://www.mathworks.com/products/symbolic/index.html

than a modification to the expressions or any parameters, the learning is a shift from prior probabilities to posteriors. In our experiments we use both datasets with quantitative and label output values. For either case we use such output for both estimation of the constants and evaluation of the posterior. In the case of label outputs however likelihoods and hence posteriors are calculated in a different way than for the case of quantitative outputs. To obtain posterior values we take logarithms of the prior and likelihood and sum them. This is done because many of these probabilities shrunk as new samples are processed and they are multiplied by small single probability amounts to an order of some negative power of 10. Using log calculation helps avoiding computational inaccuracies that occur due to truncation of very small values. We use the calculated posterior values to predict and select expressions that best describe the data. We described the detailed theory of all calculations used in our experiments in Chapter 2. Here we continue by experimenting the performance of our method.

3.2 Results per detailed research questions

3.2.1 How richness of domain expression set affects their refinement performance?

By *richness* we mean the number of expressions in the hypothesis space that have a high accuracy in describing the data. As described, such performance is measured by the degree to which the list of expressions with highest posterior probability (the posterior set), matches the list of expressions that have actually performed best on the test data (the accuracy set).

In Figure 3.1, we have plotted the accuracy set of expressions (the blue dots) alongside a posterior set of expressions (the red dots) where $|U| = 50$. The vertical axis is the accuracy of prediction on the test set. To visually compare how they match, two power trendlines have been added to plot. The reason we have chosen power sketch here is that the prior probability is of such form. That is, the difference between two prior values is one or more multiplications by the constant factor of branching probabilities introduced in previous chapters. The trendline is added by Microsoft Excel which is not precise and quite reliable according to [15]. This is to just give a visual impression of the difference we are examining. We will apply the introduced performance measure here later.

The experiments are performed over sets of expressions of 3 and 4 variables in grammar. For each number of variables two datasets were generated artificially and summed with a Gaussian noise with a fixed variance. For all measurements here, only quantitative output values are used for the outcome to be predicted in both training and test sets in both datasets[2]. For the case of 3 grammar variables, Figures 3.1.a and 3.1.b show the result of such comparison for the two datasets. The target expression of the first hand-seeded dataset (Figure 3.1.a) is $x_1^2 x_2/x_3 + c_1 x_3^{c_2}$ where $c_1 = \pi$ and $c_2 = 0.5$. The second dataset (Figure 3.1.b) was generated according to the expression $x_1 - x_2^2 x_3/c_1$ with constant value c_1 set to 9.816, the standard gravity acceleration[3].

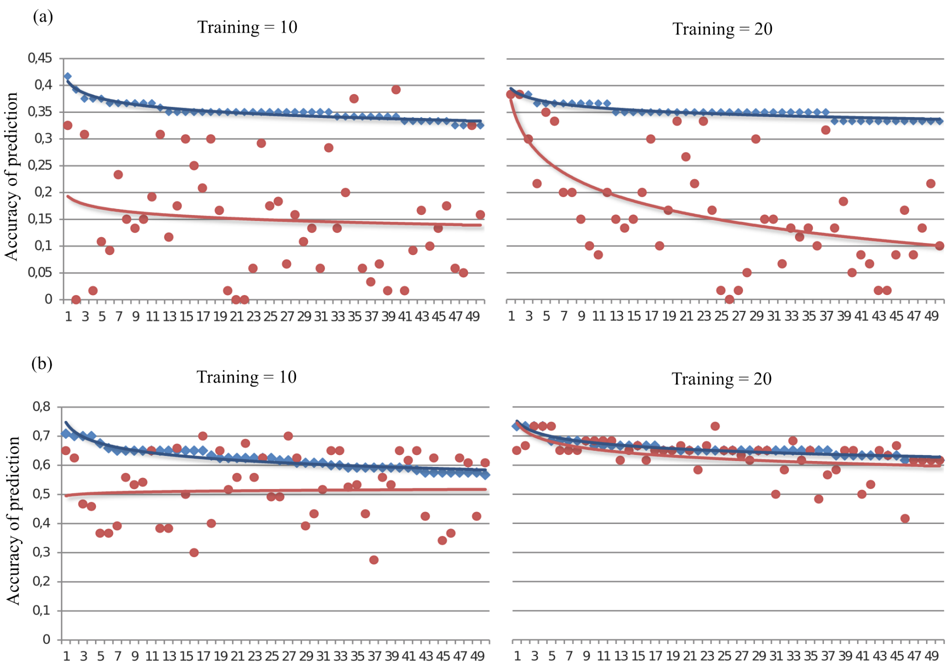

Figure 3.1: (a) The plots show a set of top 50 accurate expressions based on their measure accuracy percentage on a test set (also here referred to as *accuracy set*), sorted by the order of accuracy. Red dots are accuracy percentages of a corresponding set of 50 expressions with highest posterior probabilities (referred to as *posterior set*). The more the two trendlines match, the better posterior probabilities are measured. The plot on the left is the results with 10 training data samples of the first dataset in a 3-variable domain. The right plot is the same measures with 20 training data samples. (b) Similar to the upper row, here the second dataset in 3-variable domain is tested. All results are cross-validated in 3 folds.

In both figures and also for all other experiments, a set 40 data samples are used in testing procedure. Figures 3.2.a and 3.2.b show same juxtaposition for the case of 4 variables. Here also we have generated two datasets one

[2]In later experiments, we will also use *label* outputs instead, to study further effects.
[3]The expression resembles the Newtonian law of gravity.

based arithmetic expressions $x_1^3c_1 - x_3^3/x_4^{c_2}$ (Figure 3.2.a) where $c_1 = \sqrt{3}/2$ and $c_2 = 0.5$, and one based on the expressions $x_1x_4c_1/x_3 - c_2x_2^2x_3^2$ (Figure 3.2.b) where $c_1 = \sqrt{5}$ and $c_2 = 2$.

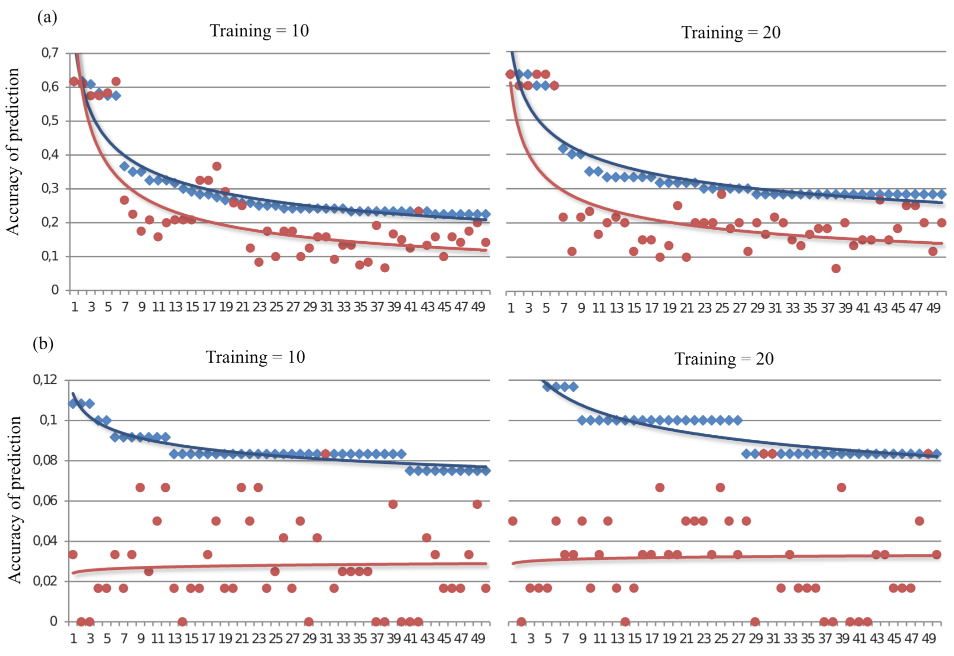

Figure 3.2: Description is similar to Figure 3.1 except for the number of grammar variables which is 4 variables here. All results are cross-validated in 3 folds.

Before continuing to our measures of similarity between the two sets depicted in each plot, here are some observations from the scatter plots. Firstly note that as the average accuracy of prediction goes higher for the accuracy set (the blue dots in the graphs), the posterior set (the red dots) tend to become more similar to the former set. First note that with increasing average accuracy, the posterior set tends to converge to the accuracy set. This can be observed by comparing the two rows (each a dataset) in both Figures 3.1 and 3.2 where accuracy measure over one data set is clearly higher than the other one. For instance in Figure 3.1, the plots on the right column show two different datasets for which the measurements are calculated identically. However, since more expressions in the lower dataset have captured the behavior of the data as compared to the upper plot, the overall rate of accuracy among the accuracy set is higher. This has led to a more accurate (and matching) posterior set for the case of the second dataset. As the number of accurate expressions falls down in the first dataset[4] the posterior

[4]This can be a result of several things. For example it can be caused by the fact that

value calculations also falls off the preciseness. In all other pairs of upper and lower scatter plots in Figures 3.1 and 3.2 the same dependency can be detected.

We should mention here that we distinguish between tests that are performed over datasets of different number of input variables. This is because we generally expect lower measures of performance as the number of input variables increase. The reason is that our only way of exploring the space of possible mathematical expressions is by generating a space of expressions using a context-free grammar, we are unable to efficiently look for all expressions. The more the number of input variables, the larger (and exponentially larger) is the number of possible expressions, and the poorer is our search. Higher number of variables results a more complex grammar and thus more time and space is required to traverse the parse tree of such grammar to reach to a certain depth (i.e. complexity of expressions). Having a fixed number of generated expressions therefore due to a time or space criteria, causes the resulting set of all expressions to be less rich of potential expressive expressions. Later in Chapter 4 we suggest some approaches to avoid this problem. As a result, we may get a biased conclusion if we compare results of experiments that have different number of input variables.

Table 3.1 shows the results of the performance measure introduced in the previous section for all cases plotted in Figures 3.1 and 3.2.

The first column shows the 4 datasets: two for each number of variables in the grammar. The second column is the size of training set over which the coefficients in expressions and the posterior probability have been calculated. Third and fourth columns show the scaled matching measurement and the average value of the top accurate list.

By comparing the four datasets we can settle that in general, the more is the number of expressions that accurately describe the data in a space of expressions, the better the method performs assigning posterior probability to that space. Let us emphasize again that we set the tolerance value to a fixed amount, meaning that the accuracy values (as well as posterior) may not be a true estimate of how successful each expression has been in prediction of the data. In other words, we can choose one or a number of expressions from the whole hypothesis space as those which describe the data the best, however with limitations. We cannot assign a measure to an

the target expression is more complex. Also as mentioned earlier, the tolerance value that have been used for both training and accuracy testing of the expressions, was selected to be a fixed value regardless of the characteristics of the dataset. Note that actual value of accuracy is not of interest here. Different accuracy for different datasets has helped us here understand where the results of posterior calculations should be trusted.

Table 3.1: Measurement of degree to which the posterior set matches the accuracy set. A fixed tolerance value has been used for evaluation for both training and accuracy testing of the expressions regardless of the characteristics of the dataset. The results in the table suggest that as the expressions get more accurate on a dataset in average, the posterior calculation picks finer expressions. All results are cross-validated in 3 folds.

Dataset	Training	$M/M_{\max}$	Mean Acc.
3-variable dataset 1	10	0.0086	0.3505
	20	0.2881	0.3706
3-variable dataset 2	10	0.1376	0.6215
	20	0.4369	0.6566
4-variable dataset 1	10	0.4295	0.3001
	20	0.3639	0.3446
4-variable dataset 2	10	0.0077	0.0850
	20	0.0131	0.0963

individual expression that expresses with certainty how well the expressions describes the data. The likelihood probability that we calculate for a hypothesis is meaningful relative to the same probability measure of the other hypotheses. Despite these limitation though, our approach serves our goal well, that is to select the most fitting expression. The other task which is to explore the arithmetic expression domain well enough to assure that such fit expressions exist, has been defined as a separate task as domain exploration, in our approach to equation discovery.

The dependency of the likelihood to noise threshold has helped us seeing a wider range of outcomes in our experiments, without having to actually search for datasets that the method performs poorly and other datasets that it performs well on them. There is however an unjustified weakness in our method. When the average accuracy of the examined expressions is low (e.g. the case with the second 4-variable dataset, see Table 3.1) the performance of the method drops severely. Now since those accuracies are relative (and not definite), this means that the performance of the method is dependent to the noise threshold. When the threshold is set in a way that accuracies are spread in a low range, the posterior (which is still a relative measure) is not a good measure of the actual accuracy of the expressions any more. The reason to this is clear. We calculate the posterior using a fixed threshold of noise over the training data as well. This has the same effect on the accuracy over the training set (expressed by the likelihood) as it has on the test set accuracy measurements, resulting the posterior to be a poor measure when calculating accuracies are poor on a certain dataset. We shall suggest some possible approaches to overcome this problem as future direction of improvement. Here we only continue with the same technique.

3.2.2 Does more training data usually mean more precision?

To answer this question, different training data sizes of various (Gaussian) noise are tested. It is expected that the more training data used for evaluation of the likelihoods (and the evaluation of constant values) per expression, the higher goes the accuracy measures. However because of the noisy data this maybe violated especially for noisier training sets. The goal is to track the effect of number of training data as well as noise in the data together and to see how they interact. As in the previous experiment we will use the similarity measure between the posterior set and the accuracy set. Also, we measure accuracy across a fixed Gaussian noise level. Since we take a fixed tolerance value to test the accuracies, it would be worthless to measure accuracy of the expression for different levels of noise for same number of training data.

In our experiment we use training sizes of 2, 5, 10, and 20. Since we use artificially generated data, we have then added Gaussian noise to each of these sets with sigma values of 2, 5, and 10, generating a number of 12 ($= 3 \times 4$) datasets. We have also repeated each experiment (all training and testing steps) 3 times. The data sets are generated using the second 3-variable expression from the previous section: $x_1 - x_2^2 x_3 / c_1$. All expressions were trained on each set, and tested on 40 data samples of the same data and noise parameters. We have used the second dataset of the previous experiment because we wanted a rich set of expressions. This helps our judgments to be not based only on few accurate expressions, which is the case with other datasets of the first experiment. But rather, where having several expressions that can (to some extent) describe the data, we are able to see more consistent and input-insensitive outcomes and thus more valid conclusions.

Figure 3.3 shows the performance measure $M/M_{\max}$ for each of these test settings. Recall that this measure is our only way to evaluate performance of our method independent of any parameters. What it essentially presents, is the degree that the algorithms are successful in elevating by the means posterior value, those expressions that actually have high accuracy.

The chart on the left shows that in general, as the number of training samples grows, the performance also improves. Having more observations, it should be easier to find the relation between the input and output variables. This increase of performance is also noticeable in the right chart of Figure

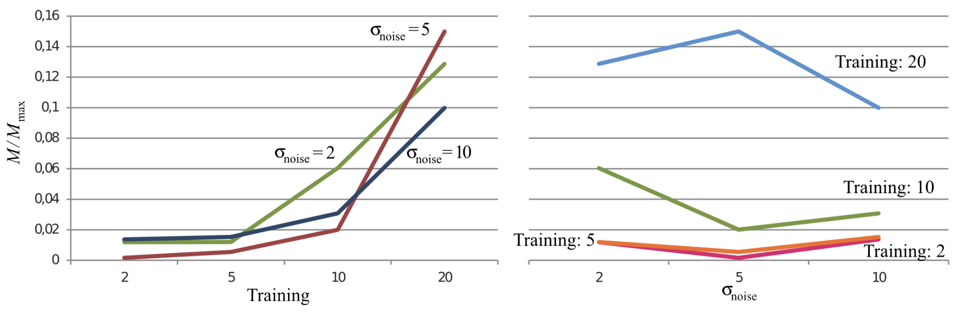

Figure 3.3: Figure illustrates a comparison of performance, per 4 training data sizes and 3 noise levels. Both charts are showing the same data. Left has training set size and Right has noise levels as their horizontal axes. A line connects measurements of each group together. All results are cross-validated in 3 folds.

3.3 as the line segments connecting different noise levels for a fixed training set size are lifted upwards as that size increases.

In the right chart another issue is evident and worth to note. All for connecting lines have an extremum point in the middle (at noise variance equal to 5) and then lower or higher values for noise sigma 2 and 10. In processes of training and posterior calculation, there are two phases that noisy data can affect the performance. The first is when all-symbolic expressions are processed and the coefficient symbols are determined and replaced by actual numeric values. The second is the accuracy evaluation of the expressions on the training set, of which the results are used in calculation of the likelihood. For these two processes the noise parameter has a dual effect. If it is too small compared to the actual (unknown) noise of the dataset, training of the expressions can result in overfitting to the data, especially for large training sets. Therefore a noise with a higher variance can inadvertently avoid this. On the other hand a higher noise means less accuracy. So from one side noisy data prevents overfitting of the coefficients[5] inside the expressions causing the accuracy to increase, and from the other side, higher noise induces a lower accuracy and thus a lower performance (see previous section) in test outcomes. In the chart this dual effect is sensible in all the 4 lines. For the lower three noise levels this has caused the graphs to have a minimum point, for the upper one it is the opposite.

[5]Note that there is no explicit procedure in the method that would work to prevent overfitting to the training data. The only prevention occurs when the priori filters expressions with high complexity.

3.2.3 How is the performance when using label outputs instead of quantitative output values?

As discussed before, one of our main goals is to have this method to work when outcome values are labels rather than quantities. In this case expressions that qualify are similar to classifiers of the target datasets rather than regressors. At first we focus on realizing how good the method performs when trained and tested on labeled data[6]. For all experiments with labeled datasets, we assume that there are two possible labels -1 and 1 which indicate whether or not the outcome value is negative or positive respectively. These negative and positive labels correspond to the actual quantitative outcomes that are in the data; so wherever the quantitative output value for a data sample is positive its label is 1, etc. Recall that the calculation of the likelihood in this case is based on exact match between the predicted and true labels. Same label values are regarded as likelihood probability equal to 1, and non-identical labels are treated as likelihood 0 [7]. We have mentioned in the previous chapter that because the overall likelihood is calculated as multiplication of these single likelihood probabilities, only one non-matching label would result 0 for the whole calculation and thus make the posterior value to go infinitely small. For noise-free data, such likelihood calculation can lead to selection of few (if any) expressions that are the same as the actual mathematical relation underlying the data. Such synchrony occurs rarely with noisy data and therefore the system performs very poorly on labeled datasets (See chapter 2.3.5). Table 3.2 shows the results for mean accuracy values as well as our performance measure for the labeled data with 3 input variables. We have used the same dataset as in the second experiment, only with labeled outputs.

Despite high average accuracy (Table 3.2), performance measures are almost in all the cases close or equal to zero. We explained earlier in pervious chapter how to overcome this problem by the means of assigning an outlying-ness probability e^{-b} to each of the observations. Using the formulation described in the previous chapter for calculation of likelihood for labeled data (2.3.5), we calculate the likelihood and posterior values for the same setting as the Table 3.2. Let us note here that the value for parameter b can be found empirically for a given dataset by doing repeated experiments and comparing performance measures. We present a related experiment later in this section. For now however, as tolerance parameter for the case of quantitative outcomes, for labels outcomes also, we take a fixed value for parameter b. Figure 3.4 depicts our performance measure $M/M_{\max}$ for three noise variances: 2, 5 and 10 and three training-set sizes 2, 10, and 20 when these

[6]There are also similar studies in the literature that deal with labels rather than quantities. Especially in the field of computational cognitive sciences, there are several example of such domains [22].

Table 3.2: Table compares mean accuracy of the top accurate expressions and the measure of performance for the case where simple likelihood calculations are used on labeled data. Despite high average accuracy performance measures are generally very low. All results are cross-validated in 3 folds.

σ_{noise}	Training	Mean Acc.	M/M_{max}
2	2	0.7932	0.0212
	5	0.7858	0
	10	0.7910	0
	20	0.7780	0
5	2	0.7557	0.0442
	5	0.7670	0.004
	10	0.7462	0
	20	0.7227	0
10	2	0.6288	0.0384
	5	0.6335	0.0116
	10	0.6352	0
	20	0.6333	0.0074

probabilities are applied.

Figure 3.4 (a) and (b) show the result of measurement for two different values for parameter b. As with the case of quantitative outputs where we had to fix the tolerance regardless of the level of noisy-ness of the dataset, here also we have to fix b. In practice, b must be found by running several experiments on a specific domain and adjusting e^{-b} to match the actual probability of observing an outlier.

As seen in both cases, this new way of calculation of the posterior has helped avoiding the problem of getting zero posteriors because of few outlying samples. Finding the proper b for a domain is a vital issue for ensuring performance of the system though. Figure 3.4 (b) shows measurement with outlying-ness probability less than half of the same probability for 3.4 (a) and has improved the performance twice as much. In both charts the method has performed worse in case of noise variance equal to 2, compared to the case that the noise variance is equal to 5. This suggests that the outlying-ness probability has compensated for the higher noise level from 2 to 5 and has led to better outcome. This holds also in case when noise level is increased from 5 to 10 as long as the training dataset size is not to large. Note that when 20 samples are used for training, a large noise variance (here 10) results a low performance compared to case the same data with smaller noise

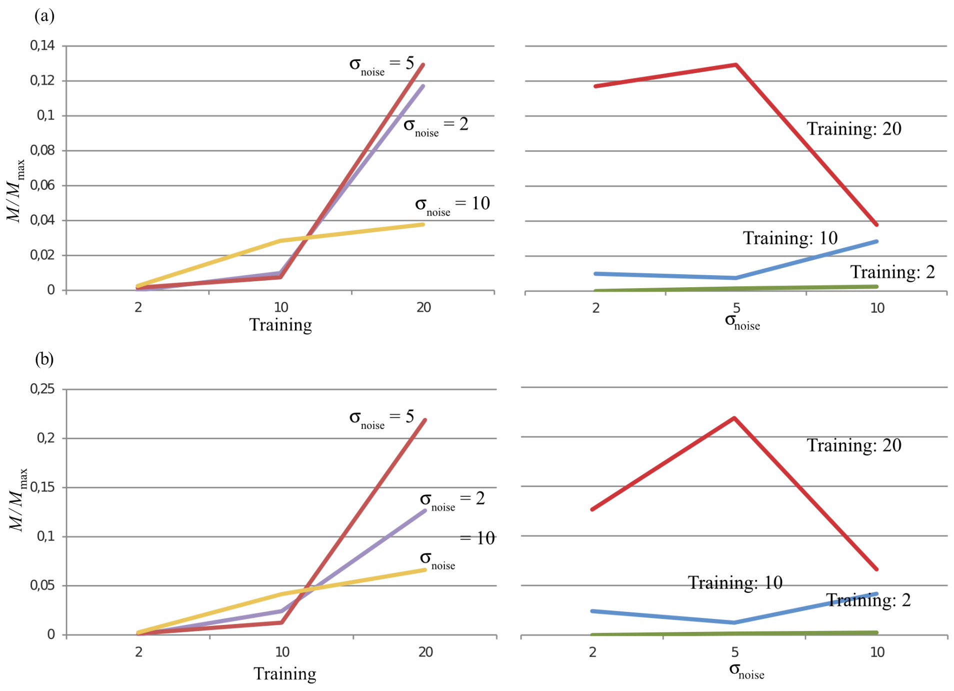

Figure 3.4: Figure shows a comparison of performance measure for two different values of parameter b: (a) 2 and (b) 3. Left charts show the three levels of noise in colors, while the horizontal axis is the size of the training set. The charts on the right show same measure replacing the horizontal axis to be the noise level and the colored lines to be connecting same noise variances. All results are cross-validated in 3 folds.

is used[7]. The left and right charts represent the same data over different noise levels and different training sizes, respectively.

Choosing the right value for b is hard in the same way that a proper noise threshold is problematic to be choosen. There is however another important effect that b has on the performance of the system. In section 2.3.5 we discussed how the outlying-ness probability acts as a filter that allows some expressions to pass and disposes the rest. The correct value for b thus, has a very major role in performance of the system when dealing with the non-quantitative labeled datasets. A small b (i.e. large outlying probability) lets many expressions to pass the filter simply because it makes it easy to score a rather high posterior while not describing most of data. On the other hand a large value can filter out many fine expressions[8]. Figure 3.5 shows

[7]Since our datasets are generated artificially and we are therefore able to adjust the noise on the same data. Also recall that we cross-validate all results in 3 folds.

[8]The extreme case happens when b is set to a very large value ($+\infty$). This causes the probability e^{-b} fall to zero. The effect is the same as when simple calculation of likelihood

how an optimal value for b varies as a function of the size of the training-set and the variance of the noise of the dataset.

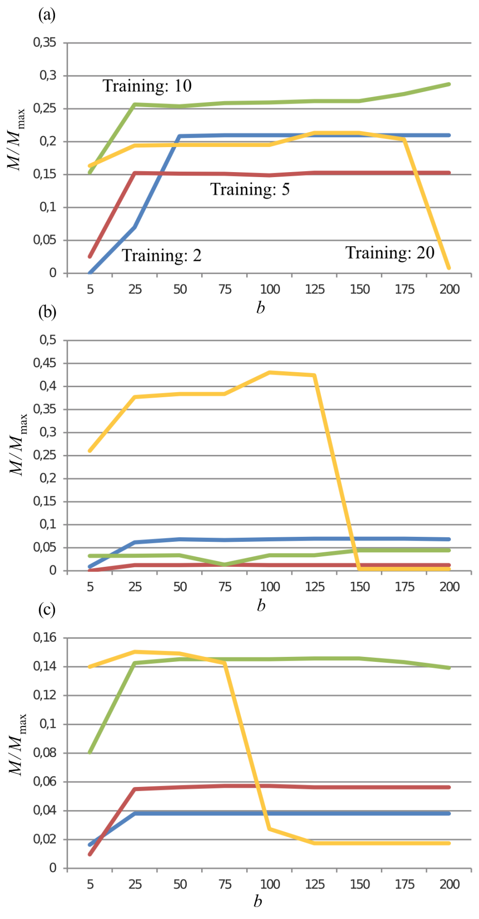

Figure 3.5: The three charts display a comparison of performance measurements for several values of parameter b. A line connects measurements of same training size. The measurement are done for 3 levels of dataset noise variance 2 (a), 5 (b), and 10 (c). All results are cross-validated in 3 folds.

Figure 3.5 suggests that in most cases the maximizing b value is somewhere in the range of 50 to 150 and it does not have sharp changes in that interval. The performance measure usually stand the same in areas around the maximum point. When b is as high as for example 100, the probability e^{-b} is as low as 3.7×10^{-44}. While this not being always the case, a question that arises here is that why such small probability even matters. This is where the filtering function we discussed comes in. Those expressions that predict a significant percentage of the training data are able to

is used.

pass the filter while the rest are filtered away simply because their posteriors become very close to zero. This happens due to those expression that have too many small values multiplied in their calculation of the likelihood (as the result of many mispredictions) and therefore resulting a zero the log of which, constitutes the posterior. Note in Figure 3.5 that for the case of training with 20 data samples, a sudden drop of performance occurs after the optimal point. That is where the filter gets too strict and filters out most of accurate expressions. The more is the number of training-set the faster one should expect this sudden drop to happen, because more infinitesimal likelihood values are multiplied together in the final likelihood product.

One thing to mention here is that as notable from the graphs in Figure 3.5 the performance does not reach to high percentages. This is the result of the comparison of 50 top posterior with top 50 accuracy lists. Since we have taken 50 as a fixed number of expressions that we compare from the head of both lists and since there are not always that much matching expressions, it is normal to have such difference. From all expressions in our hypothesis space that are trained and tested, only few may be a fine explainer of a given dataset. The goal is not to achieve a high value for the introduced measure of performance, but rather, to discover under what setting it is maximized.

Chapter 4

Discussion, Conclusion and Future Directions

4.1 Concluding remarks

We presented a framework for learning arithmetic expressions from a set of observations. Our intention has been to introduce a method for rule-based equation discovery. This method is based on evaluating a degree of belief (posterior probability) for a set of hypotheses to find those which best explain the observed data. The hypotheses are expressions that each may explain some of the observations to some degree. Together they shape the *hypothesis space* of the problem, which we search in for the fittests. We discussed that our method takes the hypothesis space of expressions as granted. We distinguish two tasks in the process of equation discovery, namely: the task of exploring the space of arithmetic expressions and that of evaluating the degree that an expression describes the data. Separating these two, allows them to be applied independently.

4.1.1 Our view toward this research work

The idea of equation discovery has one inherent assumption in itself that is, there is an actual *true* function that describes the behavior of the nature that we are observing fully and flawlessly. In many fields of empirical sciences though, scientists have learned only through gradual growth of scientific thinking, that the actual true might not be what we should be after. In fact, many models of the mathematical nature of our surrounding events that had been powerful and much explaining every observation, were overthrown by new models of new eras. The Newtonian model of force and gravity was an excellent description of the motion of bodies at normal speeds and predicted every interaction between them accurately. Einstein's theory of general relativity however, could introduce and explain cases for which the Newtonian model failed. Yet new development in quantum mechanics tend to disprove Einstein's theories, for more expressive theories that may unify the forces and motion in a broader perspective.

Our initial view of our work was to look for a certain expression which we know we have generated that data from and to measure our performance by how successful we are in finding that certain expression. As we developed the work though, this view changed. We altered our goal from finding a certain model that we *know* is the correct one, toward finding a model that *best describes* our observations so far, having in mind that in an actual setting of a real-world experiment, we would never know what certain rule guides our universe.

4.1.2 Discussion of the theory

As described in details in chapter 2, we generate the hypothesis space by using a context-free grammar . This hypothesis space contains a large number of hypotheses (each an arithmetic expression) that should be tested against the data. We also evaluate complexity of the expressions using this grammar in the form of prior probability. We explained that this is done by multiplying the probabilities of all derivations that form an arithmetic expressions. As we have mentioned earlier, our algorithm for exploration of the hypothesis space is trivial and naive and it lacks efficiency. It is not however, our primary focus in this work.

Here we have mainly focused on the posterior evaluation using Bayes formula. Our Method tests a hypothesized expression against a set of provided samples that have quantitative features. We look for an expression that best describes this training set. We calculate a likelihood probability and multiply it by the prior acquired from the parsing of expressions using the grammar to achieve a posterior value per expression. This posterior probability is a measure of qualification for each expression and it is the one by which we choose the final fit expression.

Finally we test the performance of our method over a number of hand-seeded datasets. We use a part of each dataset for the training and a part for testing. The training consists of both evaluation of constant and calculation of the posterior probability. For testing we predict results for all test data using each expression in the hypothesis space. Then we compare the rate of correct predictions per expression with posterior probabilities.

4.1.3 Experiments

Our tests (as presented in detail chapter 3) are mainly on the performance of Bayesian posterior evaluation over several situations and parameterizations. We introduce a measure for comparing the two above-mentioned factors and we use it to understand outcomes of all expressions.

We define a measure of performance and we use it to explain the results. The measure expresses the degree to which our method has been able to assign highest posterior values to those expressions that are in fact the most accurate. In our experiments we have tried to answer three main questions. Initially we test dependency of the method on the overall accuracy of the expressions in the hypothesis space. We assert that unsurprisingly, performance is high where there are expressions in the hypothesis space that accurately predict the test data. The average accuracy measure of the expressions, directly relates to the performance. Thus for a dataset that the average prediction accuracy is relatively low, the algorithm fails to distinguish much between a fine expression and a loose one. On the other hand, when there are expressions in the hypothesis space that accurately predict the data, they are signified with a high posterior probability and hence the performance is high.

We also test the effect of data noise and the training size on the performance. The main trend in the result suggests that with higher number of training samples and lower noise, performance improves. There are however some other effects between the two parameters. For example on a dataset with relatively high noise variance, bigger training sets does not always help. In such case, a large number of training data results selections of those expressions that are over-fitted to the training set and thus will not predict the test set accurately. A method that can provide us with an estimation of noise, can help us avoid this problem. A suggestion on such method is given in the next section.

There is another side to interaction of noise and training size. Because we use the training set for both tasks of constants estimation and likelihood calculation, a dual effect appears in the performance. When fitting the constant values using the Simplex optimization, higher noise keeps the expressions from being too fit to the training data, because the optimization function simply does not converge. As a result, for expressions that match to actual target expression only partially, this lack of convergence avoids over-fitting to data. On the other hand, when the noise is higher the likelihood measures less and the selection of fine expressions based on their posteriors, becomes less precise. This dual effect of noise variance causes the performance to be non-monotonic over different values of noise parameter.

Since we test datasets with both quantitative and labeled outputs, we also test the performance of our method on labeled data. Similar to the case of quantitative outputs, performance with labeled outputs also increases as a function of average accuracy of the expressions. The tradeoff between the noise level and the training size, as well as the dual effect mentioned above appear also in case of the labeled data. Since posterior evaluation of the labeled data involves a parameter b that adjusts what we call the probability of a sample to be an outlier, we also investigate the behavior of or model to and examine the relation between the value for this parameter and the performance measure. The experiment suggests that as the noise increases, a higher value of outlying-ness probability result to better performance. Clearly higher noise variance increase that probability of outliers in a dataset. It also demonstrates how this parameter can act as a filter that selects best-fitting expressions.

4.1.4 Strengths and weaknesses

Our proposed Method has several weak and strong points that are worth to mention and to consider as possible areas of improvements. Its weak points include firstly the fact that it uses an inefficient algorithm to explore the hypothesis space of arithmetic expressions. Without such exploration, the practicality of our approach to equation discovery is very much compromised. The reason is that size of the domain of all arithmetic expressions, even with strong simplicity limitations, is exponentially large. Searching this domain by merely considering every hypothesis would either need too much time or lose precision. In our experiments we have hand-seeded the testing data with selected simple target expressions. However, in a real world application, this can clearly not be the case. Therefore our naive exploration method can very possibly miss the target.

Another problem is that the posterior value that we calculate for an expression is only interpretable relative to the posterior value of another one. That is it can only be used to choose one expression over others and it does not express much on how fit is an individual hypothesized expression. As we mentioned earlier, this is caused by the fact that we take a constant as the variance of noise and we do not adapt it to actually match the noise variance of the dataset. It is not however easy to estimate this noise from the set of observed samples. We will discuss a suggestion to overcome this problem in the next section.

Despite its shortcomings, our approach has a number of advantages. In general, it integrates the Bayesian-based work by Goodman et. al. [7] (on the domain of logical expressions) to the domain of arithmetic expressions and equation discovery and it is novel in that regard. The flexibility of approach allows using easily on limited or totally different domains. Because or posterior evaluation is independent of the type of hypothesis space, it is easy to apply it on any grammar based approaches (e.g. [26]) to equation discovery.

Bayesian posterior evaluation has also the advantage of being adjustable. For example, one can favor the likelihood more than the prior probability if one does not want to avoid complex expressions where precision is of more importance. Besides, it might be helpful to adjust the calculation of priori and likelihood for some specific domains where certain types of arithmetic expressions are targeted. Our method has the advantage that it is flexible and easy to integrate with other techniques for domain exploration, grammar definition, and the way the probabilities are calculated. The better and the more efficient these subproblems are solved then, the more reliable outcomes are obtained.

4.2 Directions of future research and improvements

4.2.1 Exploration of the hypothesis space

We mentioned already that the way we explore the problem space is neither efficient nor comprehensive. What we do here is merely producing a number of hypotheses and testing them. Therefore a room for improvement is immediately clear in this regard. The nature of the domain of the arithmetic expressions can be of a great help for example. The expressions are produced using a grammar that not only parses each expression to a tree structure (i.e. parse tree) but also is a compact representation of the whole problem space. Thus the whole problem space can be represented as an infinitely large tree structure that contains every derivation possible from the starting symbol of the grammar. Every path from the root (the initial symbol) to any leaf (an expression) would represent a number of derivations that have led to the expression. A middle node will represent a subset of expressions that are similar, dependending on the depth of that node.

As a result, there are imaginable ways to exploit the tree structure of the domain and take advantages over the naive approach of simply producing expressions at first place. For example the training data can be used to apply a branch and bound technique and to guide the learner through most likely paths from root to the leaves and to cut off the unlikely ones. Exploring the tree, one can bound the search at a certain node based on a measure that expresses the probability of finding a solution under that node. This probability measure should account for all possible derivations that is possible to reach while moving downwards from the node. But because the tree can be infinitely expanded, it is not possible to sum over all possibilities. A solution to this problem might be integrating over all (infinite) possibilities using a Monte Carlo sampling technique. Such method for exploration of the hypothesis space not only would result a save in posterior evaluation process time but also would search the space more thoroughly and reliably.

4.2.2 Estimation of noise in data

In our framework we measure the posterior probability of an expressions relative to the other expressions in the hypothesis space. This is due to assuming a constant noise across all datasets that the learner is exposed to them. This, although fulfils our purpose to compare expressions, has a disadvantage. The problem is that it is not possible to know how fit an individual expressions is regarding the observed data by calculating its posterior probability. Therefore when the set of all hypothesized expressions is not exhaustive enough, the judgment on fitness of expressions regarding the data is compromised.

Unfortunately though, there is no straightforward way for calculating this noise. There are statistical methods (e.g. regression) that can be applied to estimate the noise. The learner can first preprocess the training data in order to find an approximation of this noise per dataset and then continue with an approximated noise variance parameter. Due to possible complexity of datasets however, it might not be easy to have a good estimation. This seems to make comparison-based posterior evaluation a necessity. Nevertheless, it is worth investigating that how does a pre-estimation of the noise help a better evaluation of the posterior probability.

Another approach is to use a Bayesian model comparison technique [3] by defining a prior to represent the distribution of noise variance and a likelihood function that signifies the probability of observing the data samples given a certain parameter value. By integrating the probability of observing the dataset over all values of the noise parameter, one will attain a posterior probability distribution of data over all parameter values. It is then possible to evaluate the maximizing value (MAP) for the noise variance parameter.

Bibliography

[1] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* Dover, New York, ninth dover printing, tenth gpo printing edition, 1964.

[2] Dana Angluin. Queries and concept learning. *Machine Learning*, 2:319–342, 1988. 10.1007/BF00116828.

[3] C.M. Bishop. *Pattern Recognition And Machine Learning.* Information Science and Statistics. Springer, 2006.

[4] M. Botta, A. Giordana, and L. Saitta. Learning fuzzy concept definitions. In *Fuzzy Systems, 1993., Second IEEE International Conference on*, pages 18 –22 vol.1, 1993.

[5] Christopher J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2:121–167, 1998.

[6] Michael T. Flanagan. Michael thomas flanagan's java scientific and numerical library.

[7] Noah D. Goodman, Joshua B. Tenenbaum, Jacob Feldman, and Thomas L. Griffiths. A rational analysis of rule-based concept learning. *Cognitive Science*, 32(1):108–154, 2008.

[8] T. L. Griffiths and J. B. Tenenbaum. Theory-based causal induction. *Psychological Review*, 116(4):661–716, 2009.

[9] Thomas Hofmann, Bernhard Schölkopf, and Alexander J. Smola. Kernel methods in machine learning. July 2008.

[10] Anil K. Jain, Robert P.W. Duin, and Jianchang Mao. Statistical pattern recognition: A review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22:4–37, 2000.

[11] E. T. Jaynes. *Probability Theory: The Logic of Science.* Cambridge University Press, April 2003.

[12] Charles Kemp and Joshua B. Tenenbaum. Theory-based induction. In *CogSci 2003*, 2003.

[13] Charles Kemp and Joshua B. Tenenbaum. The discovery of structural form. *Proceedings of the National Academy of Sciences*, 105(31):10687–10692, August 2008.

[14] P. Langley, H. A. Simon, and G. L. Bradshaw. Computational models of learning. chapter Heuristics for empirical discovery, pages 21–54. Springer-Verlag, London, UK, UK, 1987.

[15] B.D. McCullough and David A. Heiser. On the accuracy of statistical procedures in microsoft excel 2007. *Computational Statistics and Data Analysis*, 52(10):4570 – 4578, 2008.

[16] S Ryszard Michalski, G Jaime Carbonell, and M Tom Mitchell, editors. *Machine learning an artificial intelligence approach volume II*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1986.

[17] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965.

[18] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965.

[19] Robert M. Nosofsky, Thomas J. Palmeri, and Stephen C. McKinley. Rule-Plus-Exception Model of Classification Learning. *Psychological Review*, 101(1):53–79, 1994.

[20] Luc De Raedt. Logical settings for concept-learning. *Artificial Intelligence*, 95(1):187 – 201, 1997.

[21] Carl E. Rasmussen and Christopher Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

[22] Thomas R. Shultz, Denis Mareschal, and William C. Schmidt. Modeling cognitive development on balance scale phenomena. *Mach. Learn.*, 16(1-2):57–86, July 1994.

[23] E. Smith and D. Medin. *Categories and Concepts*. Harvard University Press, Cambridge, MA, 1981.

[24] J. Tenenbaum, T. Griffiths, and C. Kemp. Theory-based Bayesian models of inductive learning and reasoning. *Trends in Cognitive Sciences*, 10(7):309–318, July 2006.

[25] Joshua B Tenenbaum and Thomas L Griffiths. Generalization, similarity, and bayesian inference. *Behavioral and Brain Sciences*, 24(4):629–40; discussion 652–791, 2001.

[26] Ljupco Todorovski and Saso Dzeroski. Declarative bias in equation discovery. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pages 376–384. Morgan Kaufmann, 1997.

[27] L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27:1134–1142, November 1984.

[28] A.R. Webb. *Statistical pattern recognition.* Wiley, 2002.

[29] Emine Yilmaz, Javed A. Aslam, and Stephen Robertson. A new rank correlation coefficient for information retrieval. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '08, pages 587–594, New York, NY, USA, 2008. ACM.