Automated Proof Pattern Recognition: the Manual

Ekaterina Komendantskaya and Rafig Almaghairbe

1 Introduction

This Documents is a Manual supporting the project **Machine-learning coalgebraic automated proofs**. Several experiments on pattern-recognition of proof-patterns are given here.

We provide a method to convert automatically produced proof-trees into feature vectors that contain essential proof characteristics. We sampled hundreds of examples of such feature vectors and used them as inputs to neural networks and SVMs with kernel functions. We tested whether they allow to detect essential proof patterns that determine meta-properties of proofs given by Problems 1-5, and Figures 13, 55, 59 below.

Statistical machine learning has a variety of methods to ensure the quality of the obtained experimental results, including analysis of sufficiently big and representative training sets, testing same data sets using various tools of pattern-recognition (neural networks, SVMs), and repeating the tests with varying parameters.

We followed all such good practices as follows: we used data sets of various sizes - from 120 to 400 examples of coinductive trees for various experiments; we sampled trees produced for several distinct logic programs – such as `ListNat` and `Stream` above. These programs were chosen to challenge the pattern classification tools, as they define dependent data types, and thus derivations involved structures of various kinds, and moreover, their intricate mixtures. We believe these experiments are important for proving the concepts for the future implementation of the techniques in dependently-typed interactive provers. Finally, we repeated all experiments using three-layer neural networks of various sizes, and compared the results with those given by the SVMs with kernel functions.

Note that we deliberately did not tune the learning functions used in pattern recognition [4, 10] to our symbolic data; and this makes our approach different from e.g. [7, 30, 39, 40]. The advantage is that we can see the power of the feature selection method, rather than the statistical learning function. Our proof representation method is generic and will allow for any future extensions, e.g. by employing new learning functions [39, 40]. In fact, the reader can repeat all our experiments using the data sets we share in [27]; using any arbitrary pattern-recognition tool.

All our experiments involving neural networks were made in MATLAB Neural Network Toolbox (*pattern-recognition package*), with a standard three-layer feed-forward network, with sigmoid hidden and output neurons. The network was trained with *scaled conjugate gradient back-propagation*. Such networks can classify vectors arbitrarily well, given enough neurons in the hidden layer, we tested their performance on 40, 50, 60, 70, 90 hidden neurons for all experiments. All experiments involving SVMs were performed in MATLAB Bioinformatics toolbox, *SVM package with Gaussian Radial Basis kernel*.

In the next following sections, we refresh definitions of coinductive trees and coinductive derivations; then we define the feature-vector encoding for the coinductive proof trees. Next, we give some background information on pattern-recognition in Neural networks and SVMs. We conclude by listing all the proofs that contributed to the feature-vector library [27].

2 Motivation and related work

Can automated proofs yield statistical pattern-recognition? — is the general question we raise in this project. This question subsumes several research questions:

1. What can we consider to be a *pattern* in an automated proof (Section 3)?
2. To what extent common practices of statistical pattern-recognition can be extrapolated to proof-pattern recognition (Section 4)?
3. Which properties of proofs can we recognise by statistical analysis of proof patterns (Section 6)?
4. How can such applications be applied within automated reasoning field (Section 7).

Statistical pattern recognition is a vibrant area within machine learning [4, 10]; it studies methods that automatically compute *classifiers* (or classifying functions), on the basis of already given examples. The most powerful methods comprise the family of non-linear classifiers, such as Neural networks, Support Vector machines (SVMs) and Kernels. We will experiment with these.

Common applications of pattern-recognition are detection of patterns in (CCTV) images (computer vision), sorting objects into classes, detecting anomalies in cells or populations, and similar tasks. The data are represented by the means of numeric vectors, each element represents a chosen feature of the objects the tool classifies; characteristic combinations of certain features are then called patterns; they determine the class of every given example in the data set.

Formal proofs are developed in a formal language and within well-defined logical theory. Following the rules of inference guarantees the correctness of such proofs. A formal proof has a precise nature, that leaves no place for statistical or probabilistic components. However, the *process* of searching for a correct proof may have statistical nature, e.g., involving trial-and-error "method", reasoning by analogy, or intuition [5, 21, 22, 23, 38, 39].

Interactive theorem provers (ITPs), such as HOL or Coq, require a programmer to tackle thousands of lemmas of variable sizes and complexities. The process of proof construction in such languages can be guided by combining a finite number of tactics. Some proofs may be composed of the same patterns of tactics, and can be fully automated, and others may require programmer's intervention. In this case, manually found proof for one problematic lemma may serve as a template for several other lemmas needing a manual proof. Discovery of such proof tactics may be one area of application of statistical pattern recognition; [32, 11].

Automated discovery of common proof-patterns using tools of statistical machine learning such as neural networks could potentially provide the much-sought automatisation for statistically similar proof-steps; as was argued e.g. in [8, 9, 11, 13, 32, 39, 40].

As was classified in [8], applications of machine-learning assistants to mechanised proofs can be divided into *symbolic* (akin e.g. Inductive logic programming), *numeric* (akin neural networks or Kernels), and *hybrid.* In this paper, we focus on neural networks. The advantages of the numeric methods over symbolic is tolerance to noise and uncertainty, as well as availability of powerful learning functions. For example, the standard multi-layer perceptrons with error back-propagation algorithm are capable of approximating any function from finite-dimensional vector space with arbitrary precision. In this case, it is not the power of the learning paradigm, but the feature selection and representation method that sets the limits.

In this paper, we are making a first step and investigate statistical machine-learning methods for first-order proofs in logic programs. There are several conceptual obstacles standing on the way of any such interdisciplinary study, all arising from the choice of the features of automated proofs analysed by the machine-learning tool. The features can be given by the truth values (boolean [18] or fuzzy [44]); one can choose to encode first-order syntax directly [25, 32, 39, 40], or enumerate the tactics which are used in ITPs and statistically analyse properties of their combinations [11]. We will briefly illustrate the effects of these three choices on proof pattern recognition. Our running example is as follows.

Example 1 *Let `ListNat` denote the logic program consisting of clauses*
```
1. nat(0) ←
2. nat(s(x)) ← nat(x)
3. list(nil) ←
4. list(cons x y) ← nat(x), list(y)
```

Neural-symbolic integration [7] is one field that merges logic (programming) and neural networks. Normally, this involves working with the Herbrand Base — the set of all the ground instances of all the atoms appearing in the program; and Herbrand models. The Herbrand base can be represented by vectors of the atoms' truth values that serve as network's inputs [7, 18, 44]. This method allows to avoid working with the first-order syntax directly. In cases like the program `ListNat`, however, the set of all ground instances would be infinite, and more elaborate methods are needed to approximate infinite models by means of finite ones [18, 30]. These methods do not adapt well to problems arising in proof-search; e.g. cannot account for such algorithms as unification or resolution; see [24]. For more on model-theoretic approach see [40].

For similar reasons, propositionalisation is often required when machine-learning various logical structures, as e.g. in [7, 36, 43].

Another solution would be to enumerate the first-order syntax and implement proof-search in neural networks directly, [26, 40]. However, in such applications, statistical nature of learning often conflicts with logical soundness; [26]. The related work on using analysis of formula occurrences in big libraries of proofs [39] traces dependencies of different lemma statements in the library, but does not analyse the patterns arising in proof search.

Another related work [11] on data-mining proofs takes combinations of tactics in ITPs as a basis for statistical analysis in Markov Networks. However, analysis of tactics on their own may not be sufficient for accurate results in proof recognition:

Example 2 *For `ListNat` and a goal G_0 = `list(cons(x, cons(y, z)))`, SLD-resolution produces a sequence of subgoals: G_1 = `nat(x),list(cons(y, z))`, G_2 = `list(cons(y,z))`, G_3 = `nat(y),list(z)`, G_4 = `list(z)`, $G_5 = \Box$. If we consider applications of each of the clauses 1-4 as analogous to the tactics used in ITPs, and also add "tactic" 5 for $\Box$, then the proof steps above could be represented as a sequence of tactics 4,1,4,1,5. However, we cannot statistically generalise this to future examples, as with some frequency, the same sequence of "tactics" will fail, e.g. take the goal G_0 = `list(cons(x,cons(y,x)))`.*

The method we present here is designed to steer away from the problems surveyed above. The solution comes in the guise of the coalgebraic methods for proof representation [29, 28]; see also position paper [25]. It allows for more subtle representation of proof patterns, that suits for statistical proof-pattern classification. Coalgebraic methods occur in different areas of computer science, and range from categorical semantics of programming languages [37, 29] and models of concurrent systems [33] to programming with infinite data-structures in Type Theory [6, 3], or in Logic Programming [15, 28]. Most coalgebraic methods pay attention to the issues of *concurrency* and *infiniteness* of computations, which requires particular attention to repeating patterns.

Apart from the technical contribution, the paper's novelty is in switching the emphasis from the properties of learning functions [32, 39, 40] to the formal analysis of what can be considered a proof pattern, irrespective of the learning function one applies on the statistical level. In Section 2, we explain our approach to "proof patterns" in proofs; we show how this general idea is realised by means of employing coinductive proof trees [29, 28]. In section 4, we represent proofs in terms of feature vectors. In Section 6, we apply and test these techniques on a range of proof-recognition tasks. Finally, Section 7 proposes several implementation scenarios for the method.

3 Coinductive Trees

Definition 3 *A* signature Σ *consists of a set of* function symbols $f, g, \ldots$ *each equipped with a fixed* arity*. The arity of a function symbol is a natural number indicating the number of its arguments. Nullary (0-ary) function symbols are allowed: these are called* constants*. Terms and substitution are defined in a standard way [31].*

Given a countably infinite set Var of variables, as follows.

Definition 4 *the set $Ter(\Sigma)$ of* terms *over Σ is defined inductively:*

- *$x \in Ter(\Sigma)$ for every $x \in Var$.*
- *If f is an n-ary function symbol ($n \geq 0$) and $t_1, \ldots, t_n \in Ter(\Sigma)$, then $f(t_1, \ldots, t_n) \in Ter(\Sigma)$.*

Variables will be denoted x, y, z, sometimes with indices $x_1, x_2, x_3, \ldots$. Substitution is the operation of filling in terms for variables.

Definition 5 *A* substitution *is a map $\theta : Ter(\Sigma) \rightarrow Ter(\Sigma)$ which satisfies $\theta(f(t_1, \ldots, t_n)) \equiv f(\theta(t_1), \ldots, \theta(t_n))$ for every n-ary function symbol f.*

We define an *alphabet* to consist of a signature Σ, the set Var, and a set of *predicate symbols* $P, P_1, P_2, \ldots$, each assigned an arity. Let P be a predicate symbol of arity n and $t_1, \ldots, t_n$ be terms. Then $P(t_1, \ldots, t_n)$ is a *formula* (also called an atomic formula or an *atom*). Complex formulae can be defined using connectives and quantifiers, but we will work only with atoms here. The *first-order language* $\mathcal{L}$ given by an alphabet consists of the set of all formulae constructed from the symbols of the alphabet.

Given a substitution θ as in Definition 5, and an atom A, we write $A\theta$ for the atom given by applying the substitution θ to the variables appearing in A. Moreover, given a substitution θ and a list of atoms $(A_1, ..., A_k)$, we write $(A_1, ..., A_k)\theta$ for the simultaneous substitution of θ in each A_m.

Definition 6 *Given a first-order language $\mathcal{L}$, a* logic program *consists of a finite set of clauses of the form $A \leftarrow A_1, \ldots, A_n$, where $A, A_1, \ldots, A_n$($n \geq 0$) are atoms. The atom A is called the* head *of a clause, and $A_1, \ldots, A_n$ is called its* body*. Clauses with empty bodies are called* unit clauses*.*

A *goal* is given by $\leftarrow B_1, \ldots B_n$, where $B_1, \ldots B_n$($n \geq 0$) are atoms.

Example 7 *Let* `ListNat` *denote the logic program consisting of clauses*

$$\begin{array}{rcl}
1.\mathtt{nat(0)} & \leftarrow & \\
2.\mathtt{nat(s(x))} & \leftarrow & \mathtt{nat(x)} \\
3.\mathtt{list(nil)} & \leftarrow & \\
4.\mathtt{list(cons\ x\ y)} & \leftarrow & \mathtt{nat(x),\ list(y)}
\end{array}$$

The program involves variables `x` *and* `y`*, function symbols* `0`*,* `s`*,* `nil` *and* `cons`*, and predicate symbols* `nat` *and* `list`*, with the choice of notation designed to make the intended meaning of the program clear.*

The algorithm of SLD-resolution [31] is a sequential proof-search algorithm. It takes a goal G, typically written as $\leftarrow B_1, \ldots, B_n$, where the list of B_i's is again understood to mean a conjunction of atomic formulae, typically containing free variables, and constructs a proof for an instantiation of G from substitution instances of the clauses in P [31]. The algorithm uses Horn-clause logic, with variable substitution determined universally to make a selected atom in G agree with the head of a clause in P, then proceeding inductively.

In what follows, we propose coinductive trees [28] as a suitable formalism for proof-pattern recognition. In their general structure, coinductive trees resemble the and-or trees used in concurrent logic programming [14]; thus, one proof tree exhibits all the possible options in the proof search for the given goal.

We will assume familiarity with the first-order logic programming [31]. The definitions of the signature Σ, the alphabet A, the first-order language $\mathcal{L}$, and a first-order logic program P are standard, see also Appendix A.

Definition 8 *Let P be a logic program and $G = \leftarrow A$ be an atomic goal. The* coinductive tree *for A is a tree T satisfying the following properties.*

- *A is the root of T.*
- *Each node in T is either an and-node or an or-node: Each or-node is given by •. Each and-node is an atom.*
- *For every and-node A' occurring in T, there exist exactly $m > 0$ distinct clauses $C_1, \ldots, C_m$ in P (a clause C_i has the form $B_i \leftarrow B^i_1, \ldots, B^i_{n_i}$, for some n_i), such that $A' = B_1\theta_1 = \ldots = B_m\theta_m$, for some substitutions $\theta_1, \ldots, \theta_m$, then A' has exactly m children given by or-nodes, such that, for every $i \in m$, the ith or-node has n children given by and-nodes $B^i_1\theta_i, \ldots, B^i_{n_i}\theta_i$.*

Compared to SLD-trees or and-or trees, the definition of the coinductive tree restricts *unification* to *term matching*, i.e., the unifying substitution θ is applied only to one atom, e.g. $A_1 = A_2\theta$, whereas traditionally mgus satisfy $A_1\theta = A_2\theta$. The term-matching algorithm is parallelisable, in contrast to the

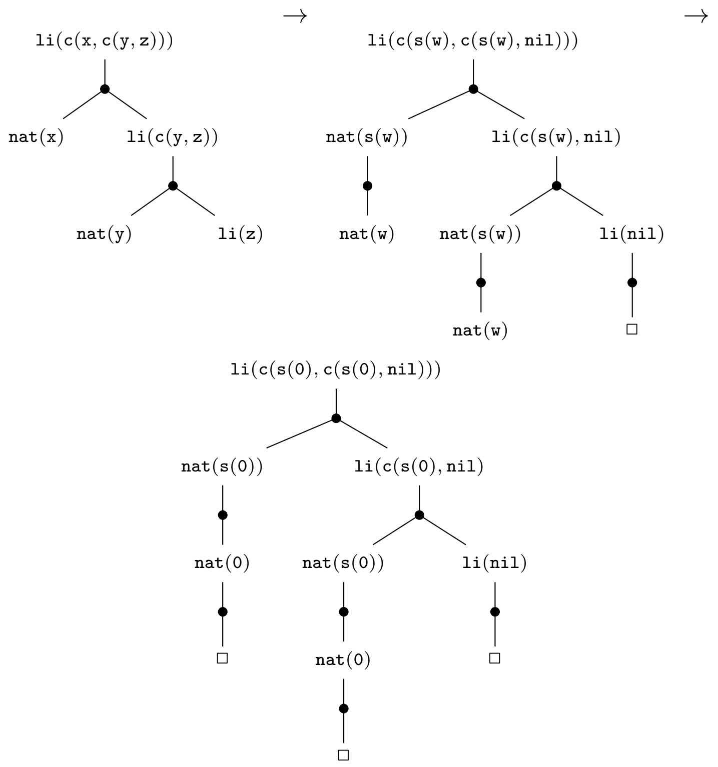


Figure 1: Two derivation steps by coinductive derivation trees, for the program `ListNat`. We abbreviate `cons` by `c` and `list` by `li` in this figure. The symbol □ signifies "success". The last tree is a *success tree* (Def. 9) which implies that the whole sequence of derivation steps above is successful.

unification algorithm, which is inherently sequential [12]. Concurrency is generally important for implementation in vector-based statistical tools. In the setting of proof pattern-recognition, the main effect of using term matching is that it delays variable identification, which, in its turn, allows to "preserve" certain proof structures during the proof search. This "laziness" of coinductive proof trees makes them convenient for dealing with programs defining infinite data structures: See also [28] for more details.

Definition 8 introduced coinductive trees, below are formal explanations of how they can be used to build derivations.

The notion of a successful proof is captured by the definition of *success subtrees* [28], they correspond to refutations in SLD-resolution.

Definition 9 *Let P be a logic program, A be a goal, and T be the coinductive derivation tree determined by P and A. A subtree T' of T is called a* success subtree *of T if it satisfies the following conditions:*

- *the root of T' is the root of T;*

- *if an and-node belongs to T', and the node has k children in T given by or-nodes, only one of these or-nodes belongs to T'.*
- *if an or-node belongs to T', then all its children given by and-nodes in T belong to T'.*
- *all the leaves of T' are and-nodes represented by $\Box$.*

Definition 10 *Let P be a logic program and $G = \leftarrow A$ be an atomic goal. The* coinductive forest *F for A is a set of all coinductive derivation trees for A. We say that the forest has* depth *n if the deepest tree in F has length n. A coinductive forest F has* breadth *k if at most k distinct variables appear in all and-nodes of all of its trees together.*

We define a goal to be a pair $< A, T >$, where A is an atom, and T is the coinduction tree determined by A, as in Definition 8, in which we restrict the choice of substitutions $\theta_1, \dots \theta_m$ to the most general unifiers only, in which case T is uniquely determined by A.

We can go further and introduce a new derivation algorithm that allows proof search using coinduction trees. We modify the definition of a goal by taking it to be a pair $< A, T >$, where A is an atom, and T is the coinduction tree determined by A. , as in Definition 8, in which we restrict the choice of substitutions $\theta_1, \dots \theta_m$ to the most general unifiers only, in which case T is uniquely determined by A.

Definition 11 *Let G be a goal given by an atom $\leftarrow A$ and the coinductive tree T induced by A, and let C be a clause $H \leftarrow B_1, \dots, B_n$. Then goal G' is* coinductively derived *from G and C using mgu θ if the following conditions hold:*

- *A' is a leaf atom, called the* selected *atom, in T.*
- *θ is an* mgu *of A' and H.*
- *G' is given by the atom $\leftarrow A\theta$ and the coinduction tree T' determined by $A\theta$.*

A *coinductive refutation* of $P \cup \{G\}$ is a finite coinductive derivation of $P \cup \{G\}$ such that its last goal contains a success subtree.

Definition 12 *A* coinductive derivation *of $P \cup \{G\}$ consists of a sequence of goals $G = G_0, G_1, \dots$ called* coinductive resolvents *and a sequence $\theta_1, \theta_2, \dots$ of mgus such that each G_{i+1} is derived from G_i using θ_{i+1}. A* coinductive refutation *of $P \cup \{G\}$ is a finite coinductive derivation of $P \cup \{G\}$ such that its last goal contains a success subtree. If G_n contains a success subtree, we say that the refutation has length n.*

Programs like `Stream` or `ListNat` always give rise to *finite* coinductive trees. This applies equally to any potentially infinite data defined using *constructors*, such as `scons` in `Stream` or `cons` and `nil` in `ListNat`. If a logic program P defines data in a guarded manner (cf. [3]), we call it a *well-founded Logic Program.* So one may view infinite coinductive trees as indicating "bad" cases, in which (co)recursion is *not guarded by constructors.*

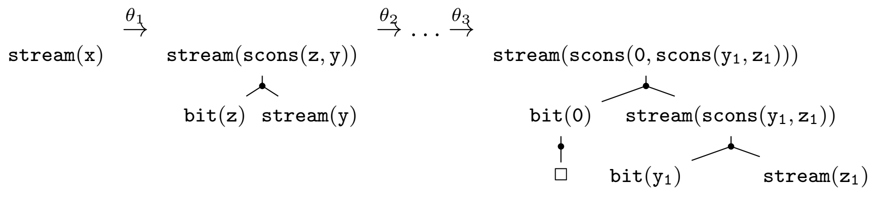


Figure 2: Coinductive derivation of length 3 for the goal $G =$ `stream(x)` and the program `Stream`, with $\theta_1 = x/scons(z, y)$ and $\theta_2 = z/0, \theta_3 = y/scons(y_1, z_1)$.

Example 13 *The following program* ***Stream*** *defines the infinite stream of binary bits. The program will induce infinite derivations if the SLD-resolution algorithms is applied, but only finite coinductive proof trees, see Figure 2.*

$$\begin{array}{rcl} bit(0) & \leftarrow & \\ bit(1) & \leftarrow & \\ stream(scons\ (x,y)) & \leftarrow & bit(x), stream(y) \end{array}$$

Compare this coinductive program with the inductive program LN from Example 7.

Coinductive derivations resemble *tree rewriting.* For every coinductive tree, there can be several transitions to a new goal, and these transitions can be made concurrently. See Figures 1 and 2 for examples of coinductive derivations; and [28] or Appendix B for formal definitions. Coinductive derivations are proven to be *sound and complete* relative to the coalgebraic semantics of logic programming, [29]. In this paper, we will not focus on their theoretical or computational properties, but we will instead use them as a convenient representation for proof-pattern recognition purposes.

In both sequential and coinductive examples of derivations, one notices apparent regularities in the structures of the proofs. In the next section, we will develop a method to machine-learn them.

For all the running examples we use in this paper, there will be only one coinductive tree for every goal. However, this will not be the case for programs containing clauses in which not all the variables appearing in the body appear in the head.

Example 14 *Figure 2 shows a coinductive derivation of length* 3 *for the goal* $G =$ ***stream(x)*** *and the program* ***Stream*** *from Example 13.*

Derivation steps by coinductive trees preserve some basic proof patterns – such as e.g. branching depending on the predicate in the goal and term structure. We now want to base our pattern recognition method on the structure of coinductive trees.

In particular, the new method must capture the tree structure and patterns arising in these trees, e.g. dependencies between the structure of terms, types of predicates, and influence of such relations on the structure of the trees. It involves more intricate data in the process of pattern recognition. yet we achieve the same goal of proof analysis as was attempted in [11]. How should we define a *proof-pattern* in the general setting? – is the question for the next section. We deliberately avoid giving a formal definition of a proof pattern. In line with the state-of-the art machine-learning applications, we wish the statistical tools to *learn* what they are; and this makes this method different from e.g. *inductive logic programming* perspective on learning.

4 Proof Feature Selection

In this section, we approach proof-pattern recognition from the statistical machine-learning perspective. Problem specification in pattern-recognition comprises defining the set of *features* of objects and a set of *classes* into which the examples are classified. Every example is represented as a numeric vector of features. A set of such vectors forms a training set for the chosen machine-learning tool. After training, combinations of features will determine the class of every new example. E.g., given a training set comprising a 100 of images of cells, the features may be the size, colour, or shape of the cells, measured numerically. Two classes work well when one detects anomalies: one class stands for "normal" sample, and one for "abnormal". However, the number of classes may grow. For example, there can be five kinds of cells we classify all samples into.

Often, one feature alone does not distinguish the class, but a combination of certain features – does; and such combinations of features are known as *patterns*. This observation is particularly true for analysis of proof patterns. It is usually a combination of factors that determines every proof step: there is a connection between such proof features as: rules of inference, unifiers, structures of predicates and terms in the current goal. Note that "feature" is a conceptually weaker notion than "pattern": features are some apparent properties of objects, that may not determine object's class. Patterns, on the contrary, are possibly hidden but conceptual properties that allow robust classification. Usually, the regular pattern that allows the classification is not known prior to learning.

We represent coinductive trees as feature vectors. Several ways of representing graphs as matrices are known from the literature: e.g., *incidence matrix* and *adjacency matrix*. However, these traditional methods obscure some patterns found in the coinductive trees. However, see [39] for adjacency matrix encoding of logic formulae represented as trees. For us, it is important to choose those features that capture dependencies between the predicates, the structures if terms, and the regularities in the tree structures. We propose a new method as follows.

For a given logic program P, a goal G, and the coinductive tree T built for G, we convert coinductive trees into vectors following the four steps below.

1. Numerical encoding of the signature Σ. Define a one-to-one function $[\![\ .\ ]\!]$ that assigns a numerical value to each function symbol in G, including nullary functions. Assign -1 to any variable occurring in T. Gödel numbering is one of the classical ways to show that first-order language can be enumerated. But in our case, the signature of each given program is restricted, and we use a simplified version of enumeration for convenience. In the method we present, the choice of the function $[\![\ .\ ]\!]$ is not crucial for proof classification, and may depend on implementation strategies, see also Section 7.

Example 15 *For program ListNat, one encoding we used was* $[\![\texttt{O}]\!] = 6$, $[\![\texttt{S}]\!] = 5$, $[\![\texttt{cons}]\!] = 2$, $[\![\texttt{nil}]\!] = 3$, $[\![\texttt{x}]\!] = [\![\texttt{y}]\!] = [\![\texttt{z}]\!] = -1$.

2. Numerical encoding of terms. Complex terms are encoded by simple concatenation of the values of the function symbols and variables. If a term $t = (f(x_1, \dots x_n))$ contains variables $x_1, \dots x_n$, the numeric values $[\![x_1]\!] \dots [\![x_n]\!]$ are negative. In this case, the positive values $|[\![x_1]\!]| \dots |[\![x_n]\!]|$ are concatenated, but the value of the whole term is assigned a negative value.

So, $[\![f(x_1, \dots, x_n)]\!] = -[\![f]\!] :: |[\![x_1]\!]| :: \dots :: |[\![x_n]\!]|$. And $[\![f(t_1, \dots, t_n)]\!] = [\![f]\!] :: [\![t_1]\!] :: \dots :: [\![t_n]\!]$, if $t_1, \dots t_n$ do not contain variables.

Example 16 $[\![\texttt{cons(x,cons(y,x))}]\!] = -21211$.

3. Matrix representation of the coinductive trees. For a given coinductive tree T and goal G, we build a matrix M for T as follows. The number of columns of M is equal to $n+2$, where n is the number of distinct predicates appearing in the program P. The number of rows is equal to the number m of distinct terms appearing in the nodes of T. We order all such terms according to their relative positions in G. We agree on some arbitrary ordering of all distinct predicates appearing in T, including additionally $\bullet$ and $\square$ to the tail of such list. As coinductive trees we work with are well-founded, the two orderings can be maintained e.g. as finite lists.

The entries of M are computed as follows. For the ith predicate R, and the jth term t, the ijth matrix entry is $[\![t]\!]$ if $R(t)$ is a node of T, and 0 otherwise. For the $n+1$ column and the jth term t, if every node containing $Q(t)$ for some $Q \in P$ has exactly k children given by or-nodes, then the $(n+1)j$th entry in M is equal to k; if the parameter k for $Q(t)$ differs in different branches of the tree, then the $(n+1)j$th entry is -1; and it is 0 otherwise. For the $n+2$ column and the jth term t, if all children of the node $Q(t)$, for some $Q \in P$ are given by or-nodes, such that all these or-nodes have children nodes $\square$, then $(n+2)j$th entry is 1; if some but not all such nodes are $\square$, then the then $(n+2)j$th entry is -1; and it is 0 otherwise. See Figure 3 and Algorithm 1.

4. Vector representation. The matrix M is then flattened into a vector, so that the columns of the given matrix are concatenated into a single vector.

Example 17 *The matrix* M_1 *above will be given by* $V_1 = [-21211, -211, 0, 0, -1, 0, 0, -1, -1, 0, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0]$.

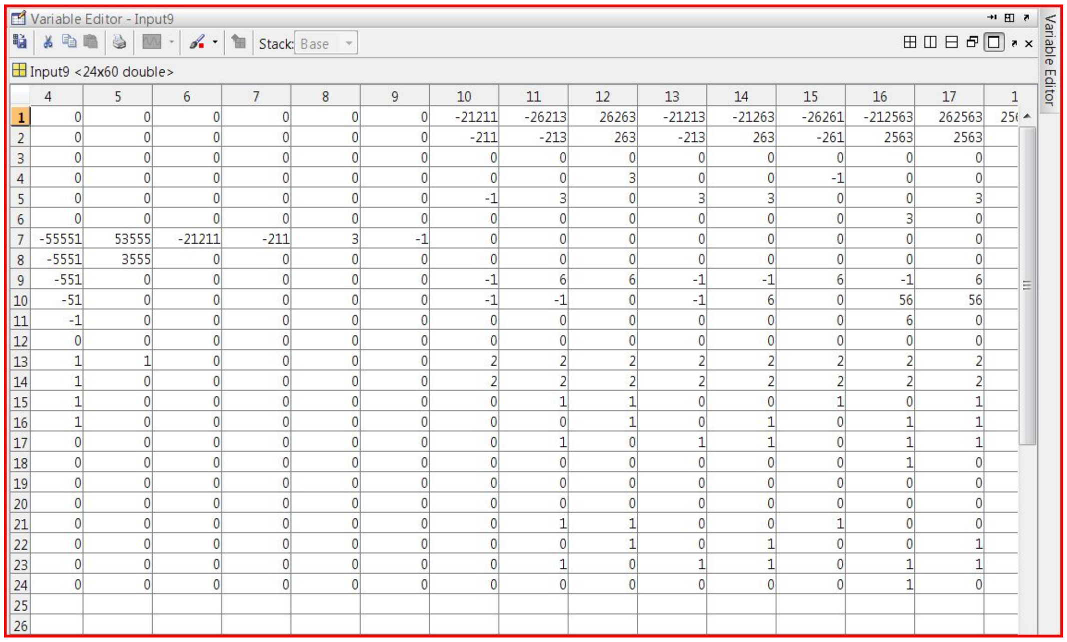
If a hundred of various coinductive proof trees are taken as examples for some proof-pattern recognition task, then there will be a hundred of vectors akin the one shown above stored in a matrix; and this matrix forms an input to

Algorithm 1 Feature extraction: proof trees to matrices

```
Require: $T$ – finite coinductive tree.
  $n$ = number of distinct predicates in $T$.
  $m$ = number of distinct terms appearing in the nodes of $T$.
  Construct a $(n+2) \times m$ matrix $M$, as follows:
  for $i = 1, \ldots n$ do
    for $j = 1, \ldots, m$ do
      if $P_i(t', \ldots, t_j, \ldots, t'')$ is a node of $T$ then
        $M_{ij} = [\![t_j]\!]$
      else $M_{ij} = 0$
      end if
    end for
  end for
  for $i = n+1$ do
    for $j = 1, \ldots, m$ do
      if every node containing $t_j$ has branching factor $k$ then
        $M_{ij} = k$
      else if nodes containing $t_j$ have different branching factors then
        $M_{ij} = -1$
      else $M_{ij} = 0$
      end if
    end for
  end for
  for $i = n+2$ do
    for $j = 1, \ldots, m$ do
      if all nodes containing $t_j$ have children given by $\Box$ then
        $M_{ij} = 1$
      else if some nodes containing $t_j$ have children given by $\Box$, and some
  - containing other formulas then
        $M_{ij} = -1$
      else $M_{ij} = 0$
      end if
    end for
  end for
    return $M$.
```

Matrix M_1	`list`	`nat`	$\bullet$	$\square$	Matrix M_2	`list`	`nat`	$\bullet$	$\square$
$\mathtt{cons}(\mathtt{x}, \mathtt{cons}(\mathtt{y}, \mathtt{z}))$	- 21211	0	2	0	$\mathtt{cons}(\mathtt{s0}, \mathtt{cons}(\mathtt{s0}, \mathtt{nil}))$	256263	0	2	0
$\mathtt{cons}(\mathtt{y}, \mathtt{z}))$	- 211	0	2	0	$\mathtt{cons}(\mathtt{s0}, \mathtt{nil}))$	2563	0	2	0
x	0	-1	0	0	`s0`	0	56	1	0
y	0	-1	0	0	`0`	0	6	1	1
z	-1	0	0	0	`nil`	3	0	1	1

Figure 3: Matrices M_1 and M_2 encode the left-most and right-most trees in Figure 1.



Input9 <24x60 double>

	4	5	6	7	8	9	10	11	12	13	14	15	16	17	1
1	0	0	0	0	0	0	-21211	-26213	26263	-21213	-21263	-26261	-212563	262563	25
2	0	0	0	0	0	0	-211	-213	263	-213	263	-261	2563	2563	
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	3	0	0	-1	0	0	
5	0	0	0	0	0	0	-1	3	0	3	3	0	0	3	
6	0	0	0	0	0	0	0	0	0	0	0	0	3	0	
7	-55551	53555	-21211	-211	3	-1	0	0	0	0	0	0	0	0	
8	-5551	3555	0	0	0	0	0	0	0	0	0	0	0	0	
9	-551	0	0	0	0	0	-1	6	6	-1	-1	6	-1	6	
10	-51	0	0	0	0	0	-1	-1	0	-1	6	0	56	56	
11	-1	0	0	0	0	0	0	0	0	0	0	0	6	0	
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
13	1	1	0	0	0	0	2	2	2	2	2	2	2	2	
14	1	0	0	0	0	0	2	2	2	2	2	2	2	2	
15	1	0	0	0	0	0	0	1	1	0	0	1	0	1	
16	1	0	0	0	0	0	0	0	1	0	1	0	1	1	
17	0	0	0	0	0	0	0	1	0	1	1	0	1	1	
18	0	0	0	0	0	0	0	0	0	0	0	0	1	0	
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
21	0	0	0	0	0	0	0	1	1	0	0	1	0	0	
22	0	0	0	0	0	0	0	0	1	0	1	0	0	1	
23	0	0	0	0	0	0	0	1	0	1	1	0	1	1	
24	0	0	0	0	0	0	0	0	0	0	0	0	1	0	
25															
26															

Figure 4: Input editing in pattern-recognition tool in MATLAB.

the chosen pattern-recognition tool (neural networks or SVM), see Figure 4 for a screenshot of one of the data bases used in the paper and available at [27].

Every entry in the matrix is a feature in terms of pattern-recognition; correlation of various features is detected by the pattern-classifier. Every feature vector v of length k defines a point in a k-dimensional space; and given a set of such vectors, the pattern-recognition task is akin fitting a function given a number of such points.

Proposition 18 (Properties of the vector encoding) *For every coinductive tree T built for a goal G and a logic program P, the following statements hold:*

1. *T can be encoded as a feature vector V with the following properties:*

   If n distinct predicates and m distinct terms are contained in nodes of T, then the matrix M will have the size $(n + 2) \times m$.

2. *For a given matrix M, there may exist more than one corresponding coinductive tree T; i.e., the mapping from the set $\mathcal{T}$ of coinductive trees to the set $\mathcal{M}$ of the corresponding matrices is not bijective.*

3. *If there exist two distinct formulae F_1 and F_2 whose coinductive trees are encoded by matrices M_1 and M_2 such that $M_1 \equiv M_2$, then F_1 and F_2 differ only in variables, however, F_1 and F_2 are not necessarily α-equivalent.*

Proof.

1. By construction above.

2. Consider the conductive trees for the two α-equivalent goals `list(cons(x,cons(y,z)))` and `list(cons(y,cons(z,x)))`; they will be represented by same matrices, see M_1 in Table 3.

3. Proof trees that are not α-equivalent can be represented by same matrices, e.g. `stream(scons (x, scons(y, scons (x, z))))` and `stream(scons(x, scons(x, scons (y, z))))`. However, the matrices will reflect the more substantial differences in such terms, e.g. those involving typing issues, e.g, the matrix for `stream(scons(x, scons(y, scons (z, x))))` (the ill-typed goal) will be different from the two matrices above, although the numerical encoding of the terms is the same.

   The matrices for `list(cons(s0,cons(s0,nil)))` (Table 3) and `list(cons(s0,cons(0,nil)))` will be very similar, showing the difference only in term encoding, but not in other patterns.

We sacrificed bijectivity of the matrix encoding for a reason: it actually allows to capture some important proof patterns better: e.g., it helps to identify any proof patterns related to variables, convenient for handling $\alpha-$equivalent terms and terms that are not α-equivalent but computationally similar. The feature vectors enable to trace inter-dependencies of terms and predicates with proof branching, typing, and ultimate success. The formulae and derivations that are identical up to variable substitution will receive identical matrix encoding, due to the uniform representation of variables. Identified proof trees as above are indeed closely related in terms of proof patterns involved, as we better explain in the next section. At the same time, the matrices are powerful enough to capture the patterns that identify ill-typed terms. E.g., for a goal `list(cons(x,cons(y,x)))`, the matrix will have non-zero values in both "`nat`" and "`list`" columns, which will be similar to all ill-typed goals.

5 Classification and Pattern-Recognition Problems

Linear regression is the problem of fitting a linear function to a set of input-output pairs given by a set of training examples, in which the input and output features are numeric.

Suppose the input features are $X_1, \ldots, X_n$. A linear function of these features is a function of the form:

$$f(X_1, \ldots, X_n) = w_0, +w_1 \times X_1 + \ldots + w_n \times X_n,$$

where $w_0, \ldots w_n$ are weights.

Suppose a set E of examples exists, where each example $e \in E$ has values $val(e, X_i)$ for feature X_i and has an observed value $val(e, Y)$. The predicted value is

$$pval(e, Y) = w_0, +w_1 \times val(e, X_1) + \ldots + w_n \times val(e, X_n)$$

Computing an error: Sum of squared errors (”sum-of-squares”):

$$Error(\bar{w}) = \sum_{e \in E} (val(e, Y) - pval(e, Y))^2$$

The next algorithm is one of the most commonly used for classification:

```
    Gradient Descent.
1:  Procedure LinearLearner(X,Y,E,$\eta$)
2:  Inputs:
3:  X: set of input features, $X = \{X - 1, \ldots, X_n\}$
4:  Y: target feature
5:  E: set of examples from which to learn
6:  $\eta$:  - learning rate.
7:  Output:  parameters $w_0, \ldots, w_n$.
8:  Local $w_0, \ldots, w_n$ - real numbers
9:  $pval(e, Y) = w_0, +w_1 \times val(e, X1) + \ldots + w_n \times val(e, X_n)$
10:  initialise $w_0, \ldots, w_n$ randomly
11:  repeat
12:  for each example e in E do $\delta := val(e, Y) - pval(e, Y)$
13:  for each $i \in [0, n]$ do
14:  $w_i = w_i + \eta \times \delta \times val(e, X_i)$
15:  until termination
16:  return $w_0, \ldots, w_n$.
```

Classification In classification tasks, there are normally two values — 0 and 1, so linear function is not well suited.

For classification, one uses **squashed linear function** of the form

$$f(X_1, \ldots, X_n) = G(w_0, +w_1 X_1 + \ldots + w_n X_n)$$

where G is **an activation function** from reals numbers to $[0, 1]$.

Example - a step function

$$S(x) = 1 \text{ if } x \geq 0 \text{ and } S(x) = 0 \text{ if } x \leq 0$$

- ... was used in Perceptron [Rosenblatt, 1958] - one of the first methods for learning.

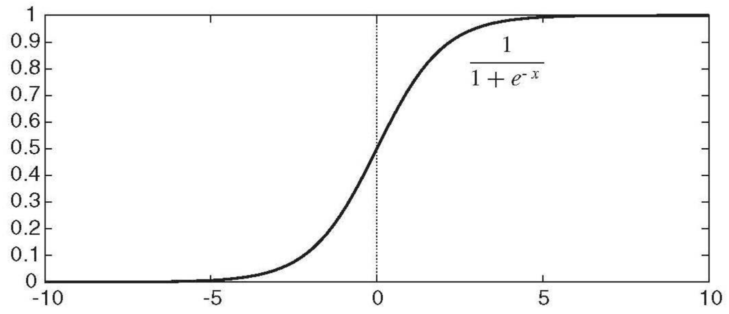


Figure 5: Sigmoid activation function.

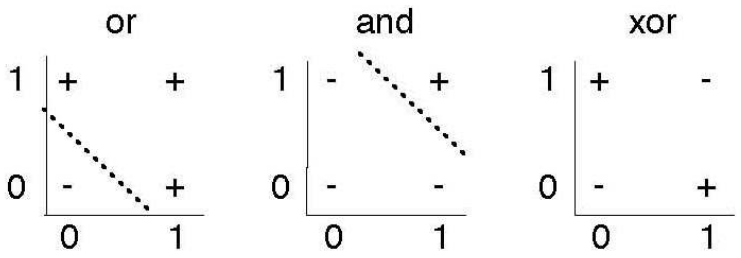


Figure 6: XOR is not linearly separable.

- Disadvantage: not differentiable.

If the function is differentiable, we can use gradient descent to update the weights:

For the sigmoid function, the derivative is $f'(x) = f(x) \times (1 - f(x))$.

We use this to change line 14 in the algorithm `LinearLearner` to

$$w_i := w_i + \eta \times \delta \times pval^{\overline{\omega}}(e, Y) \times [1 - pval^{\overline{\omega}}(e, Y)] \times val(e, X_i).$$

Problems with linear classifiers: Not all functions yield linear classification; the classical example is that Boolean function XOR could not be learned using this method; see Figure 6. In such cases, we say the function/problem is not *linearly separable.*

Neural networks can solve this!

- Multi-layered networks are like cascaded squashed linear functions.
- Each of the hidden neurons is a squashed linear function of its inputs.
- Output neurons can be linear (for regression) or sigmoid (for classification) functions.
- Learning by neural networks — is adjustment of the weights such that the prediction error is minimized.

Apart from Neural Networks, another example of Non-linear classifiers are Support Vector machines (SVMs), with various Kernel functions, such as e.g. Gaussian.

5.1 Neural Networks

In this section, we give formal definitions of neural networks.

We follow the definitions of a connectionist neural network given in [20], see also [7] and [18]. This basic definition is no different from the analogous definition of a neural network accepted in Neurocomputing [17, 16].

A *connectionist network* is a directed graph. A *unit* k in this graph is characterised, at time t, by its *input vector* $(v_{i_1}(t), \dots v_{i_n}(t))$, its potential $p_k(t)$, its *threshold* Θ_k, and its *value* $v_k(t)$. Note that in general, all v_i, p_i and Θ_i, as well as all other parameters of a neural network can be performed by different types of data, the most common of which are real numbers, rational numbers [20], fuzzy (real) numbers [35], complex numbers, numbers with floating point, and some others, see [17] for more details.

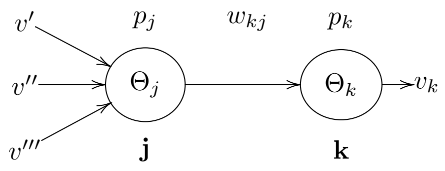
Units are connected via a set of directed and weighted connections. If there is a connection from unit j to unit k, then w_{kj} denotes the *weight* associated with this connection, and $i_k(t) = w_{kj}v_j(t)$ is the *input* received by k from j at time t. In each update, the potential and value of a unit are computed with respect to an *activation* and an *output function* respectively. Most units considered in this thesis compute their potential as the weighted sum of their inputs minus their threshold:

$$p_k(t) = \left(\sum_{j=1}^{n_k} w_{kj} v_j(t) \right) - \Theta_k.$$

The units are updated synchronously, time becomes $t+\Delta t$, and the output value for k, $v_k(t+\Delta t)$, is calculated from $p_k(t)$ by means of a given *output function* F, that is, $v_k(t+\Delta t) = F(p_k(t))$. For example, the output function used in [19] is the binary threshold function H, that is, $v_k(t+\Delta t) = H(p_k(t))$, where $H(p_k(t)) = 1$ if $p_k(t) > 0$ and 0 otherwise. Units of this type are called *binary threshold units.*

A unit is said to be a *linear unit* if its output function is the identity and its threshold Θ is 0. A unit is said to be a *sigmoidal* or *squashing unit* if its output function ϕ is non-decreasing and is such that $\lim_{t\to\infty}(\phi(p_k(t))) = 1$ and $\lim_{t\to-\infty}(\phi(p_k(t))) = 0$. Such functions are called *squashing functions.*

Example 19 *Consider two units, j and k, having thresholds Θ_j, Θ_k, potentials p_j, p_k and values v_j, v_k. The weight of the connection between units j and k is denoted w_{kj}. Then the following graph shows a simple neural network consisting of j and k. The neural network receives input signals v', v'', v''' and sends an output signal v_k.*



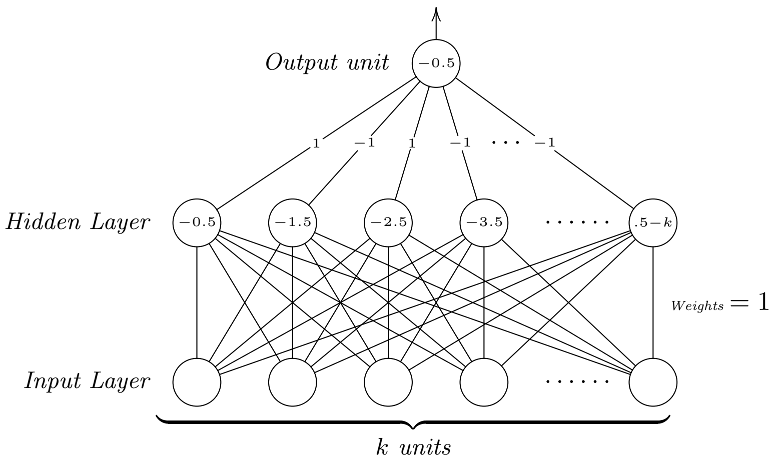
We will mainly consider connectionist networks where the units can be organised in layers. A *layer* is a vector of units. An *n-layer network* $\mathcal{F}$ consists of the *input* layer, $n-2$ *hidden* layers, and the *output* layer, where $n \geq 2$. Each unit occurring in the i-th layer is connected to each unit occurring in the $(i+1)$-st layer, $1 \leq i < n$. Neural networks consisting of layers are sometimes called *associative neural networks* [17].

The primary classification of associative neural networks is into *feedforward* and *recurrent* classes. In feedforward neural network connections between units do not form a directed cycle. In recurrent neural networks, on the contrary, connections between units form a directed cycle.

In many neural networks input units can receive only single inputs that arrive from the outside world. They typically have no function other than to distribute the signals to other layers of the neural networks. Such units are called *fanout units*.

There are interesting and useful results concerning the optimal number of layers needed for computations, and we will briefly outline them in the next subsection.

Example 20 *A typical three-layer neural network can have the following architecture:*



It is common to distribute various learning laws and techniques of neurocomputing into three major groups: Supervised Learning, Unsupervised Learning and Reinforcement Learning. Supervised error-correction leraning is the form of learning commonly used in Pattern Recognition.

5.1.1 Error-Correction Learning

We will use the algorithm of error-correction learning to simulate the process of unification described in the previous section.

Error-correction learning is one of the algorithms among the paradigms which advocate *supervised learning*. Supervised learning is the most popular type of learning implemented in artificial neural networks, and we give a brief sketch of error-correction algorithm in this subsection; see, for example, [16] for further details.

Let $d_k(t)$ denote some *desired response* for unit k at time t. Let the corresponding value of the *actual response* be denoted by $v_k(t)$. The response $v_k(t)$ is produced by a *stimulus* (vector) $v_j(t)$ applied to the input of the network in which the unit k is embedded. The input vector $v_k(t)$ and desired response $d_k(t)$ for unit k constitute a particular *example* presented to the network at time t. It is assumed that this example and all other examples presented to the network are generated by an environment. We define an *error signal* as the difference between the desired response $d_k(t)$ and the actual response $v_k(t)$ by $e_k(t) = d_k(t) - v_k(t)$.

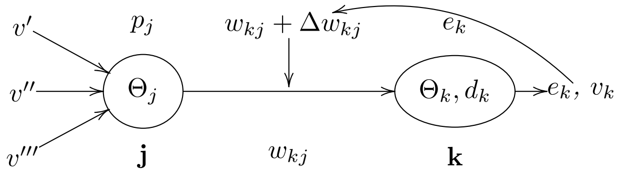
The *error-correction learning rule* is the adjustment $\Delta w_{kj}(t)$ made to the weight w_{kj} at time n and is given by

$$\Delta w_{kj}(t) = \eta e_k(t) v_j(t),$$

where η is a positive constant that determines the rate of learning.

Finally, the formula $w_{kj}(t+1) = w_{kj}(t) + \Delta w_{kj}(t)$ is used to compute the updated value $w_{kj}(t+1)$ of the weight w_{kj}. We use formulae defining v_k and p_k as in Section 5.1.

Example 21 *The neural network from Example 19 can be transformed into an error-correction learning neural network as follows. We introduce the* desired response *value d_k into the unit k, and the error signal e_k computed using d_k must be sent to the connection between j and k to adjust w_{kj}.*



5.1.2 Pattern-recognition in Neural Networks

Statistical pattern recognition is a vibrant area withing machine learning [4, 10]; it studies methods that automatically compute *classifiers* (or classifying functions), on the basis of already given examples. Once the classifier is "learned" from examples, it can be used to classify new examples. The most powerful methods comprise the family of non-linear classifiers, such as Neural networks, Support Vector machines (SVMs) and Kernels. We will experiment with these.

Common applications of pattern-recognition are detection of patterns in (CCTV) images (computer vision), sorting objects into classes, detecting anomalies in cells or populations, and similar tasks. The data are represented by the means of numeric vectors, each element represents a chosen feature of the objects the tool classifies; characteristic combinations of certain features are then called patterns; they can be statistically detected and they determine the class of every given example in the data set. E.g., if (some hundreds of) samples of flowers are classified into several sorts, one can form feature vectors where one element holds information about the height of the stem, another element – about the colour of the bud, etc.

For pattern-recognition purposes, the error-correction learning rule is taken, and namely, the backpropagation learning.

Given:

- values for parameters
- values for inputs
- set of examples

The Neural network needs to:

- predict a value for each target feature
- that is, adjust parameters (=weights)

Back-propagation learning is gradient descent search through the parameter space to minimize the sum-of-squares error, see Figure 7.

Gradient descent search:

- uses back-propagation
- repeats evaluation
- minimises the error - by iterating through all of the examples.

For pattern-recognition purposes, it is customary to use neural networks of the following structure: the neural network should have the input layer, of the size that corresponds to the size of the feature vectors to be given as inputs. It should have the hidden layer that may contain an arbitrary number of neurons, determined empirically. Finally, it has an output layer, with the number of neurons corresponding to the number of classes into which the examples are classified. See Figure 8, for the graphical structure of such networks; and Figure 9 for how such network is used in MATLAB pattern-recognition tool.

```
procedure $BackPropagationLearner(X, Y, E, n_h, \eta)$
    Inputs
        $X$: set of input features, $X = \{X_1, \dots, X_n\}$
        $Y$: set of target features, $Y = \{Y_1, \dots, Y_k\}$
        $E$: set of examples from which to learn
        $n_h$: number of hidden units
        $\eta$: learning rate
    Output
        hidden unit weights $hw[0:n, 1:n_h]$
        output unit weights $ow[0:n_h, 1:k]$
    Local
        $hw[0:n, 1:n_h]$ weights for hidden units
        $ow[0:n_h, 1:k]$ weights for output units
        $hid[0:n_h]$ value for each hidden units
        $hErr[1:n_h]$ error term for each hidden units
        $out[1:k]$ predicted value for each output unit
        $oErr[1:k]$ error term for each output unit
    initialize $hw$ and $ow$ randomly
    $hid[0] := 1$
    repeat
        for each example $e$ in $E$ do
            for each $h \in \{1, \dots, n_h\}$ do
                $hid[h] := \sum_{i=0}^{n} hw[i,h] \times val(e, X_i)$
            for each $o \in \{1, \dots, k\}$ do
                $out[o] := \sum_{h=0}^{n} hw[i,h] \times hid[h]$
                $oErr[o] := out[o] \times (1 - out[o]) \times (val(e, Y_o) - out[o])$
            for each $h \in \{0, \dots, n_h\}$ do
                $hErr[h] := hid[h] \times (1 - hid[h]) \times \sum_{o=0}^{k} ow[h,o] \times oErr[o]$
                for each $i \in \{0, \dots, n\}$ do
                    $hw[i,h] := hw[i,h] + \eta \times hErr[h] \times val(e, X_i)$
                for each $o \in \{1, \dots, k\}$ do
                    $ow[h,o] := ow[h,o] + \eta \times oErr[o] \times hid[h]$
    until termination
    return $w_0, \dots, w_n$
```

Figure 7: Back-propagation algorithm

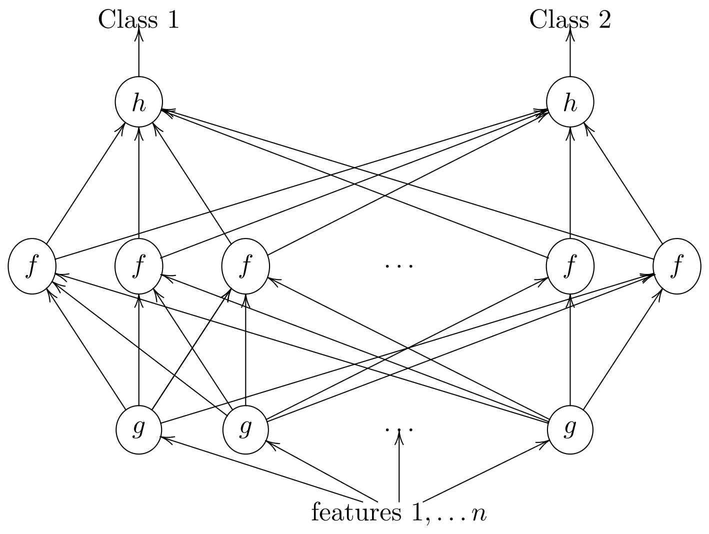
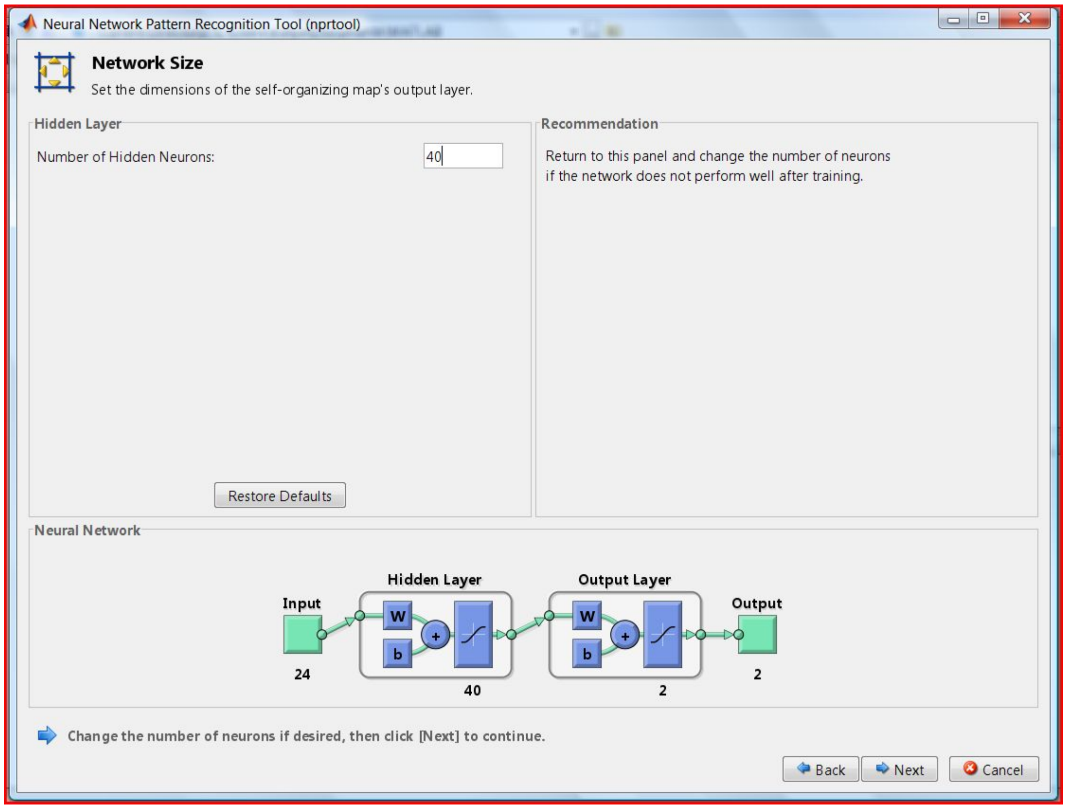


Figure 8: The abstractly represented standard neural network used for classification of objects into two classes. The size of the lower – input – level corresponds to the size of the feature space, and the two neurons in the output layer represent the two classes. The size of the intermediate (or middle) layer may vary, and can influence the precision in classification. Functions f, g, h are any activation functions, such as *threshold* functions, or as we used for classification, *sigmoid* function.



Neural Network Pattern Recognition Tool (nprtool)

Network Size

Set the dimensions of the self-organizing map's output layer.

Hidden Layer

Number of Hidden Neurons: 40

Recommendation

Return to this panel and change the number of neurons if the network does not perform well after training.

Change the number of neurons if desired, then click [Next] to continue.

Figure 9: Neural network akin Figure 8 in MATLAB pattern-recognition tool.

5.2 Support Vector Machine (SVM)

Support Vector Machine (SVM) is one of the major tools which used to perform the experiments in this thesis. In this chapter we give a brief summary of the Support Vector Machine theory and its application in the area of pattern recognition and machine learning. We will briefly outline the mathematical foundation of support vector machines for binary classification, then we will present an overview of the different approaches used for multi-class problems.

The Support Vector Machine (SVM) is a great machine learning tool based on firm statistical and mathematical foundations relating to generalization and optimization theory. It bids a powerful technique for many aspects of data mining including classification, regression, and novelty detection. Vapnik was the first researcher who suggested SVM in the early 1970's but it began to gain popularity in the mid-1990's. SVM is based on Vapnik's statistical learning theory and decreases at the intersection of kernel methods and maximum margin classifiers [41, 1, 2]. Support vector machines have been positively used to solve many real world problems such as face detection, intrusion detection, hand writing recognition, information extraction, and others.

Support Vector Machine is an attractive technique because of its high generalization capability and its ability to handle high-dimensional input data. The main advantages of SVM are that does not suffer from the local minima problem compared to neural network or decision trees, and also it has fewer learning parameters to select which produces stable and reproducible results. That's mean; if we train two SVMs on the same data with the same learning parameters then they produce the same results independent of the optimization algorithm the use. However, SVM suffers from slow training particularly with non-linear kernel and with large input data size. Support Vector Machine is mainly binary classifiers but it has the ability to solve multi-class problems by merging several binary machines in order to produce the final classification results. However, training one SVM to classify all classes require much more complex optimization algorithms and more time for training phase which is much slower compare to binary classifiers.

The following sections will explain the SVM mathematical foundation for the binary classification case and the different approaches which applied for multi-classification.

5.2.1 Binary Support Vector Classifications

Binary classification is the process of classifying the members of a given set of objects into two groups on the basis of whether they have some property or not. Many applications used binary classification tasks used which the answer to some question is either yes or no for instance product quality control, automated medical diagnosis, face detection, intrusion detection, or finding matches to specific class of objects.

The mathematical background of Support Vector Machines and the fundamental Vapnik-Chervonenkis dimension (VC Dimension) is explained in more details in the literature covering the statistical learning theory [41, 1, 2, 42, 34] and many other sources. This section introduces the mathematical foundation of SVMs in the linearly separable and non-linearly separable cases. One of the attractive features of support vector machines is the geometric intuition of its principles where one may the mathematical interpretation to simpler geometric analogies.

5.2.2 Linearly Separable Case

In the linearly separable case, there exists one or more hyperplanes that may separate the two classes represented by the training data with 100 % accuracy. Figure 3(a) shows many separating hyperplanes (in the case of a two-dimensional input the hyperplanes is simply a line). The key question is how to find the optimal hyperplane that would maximize the accuracy on the test data. The native solution is to maximize the gap or margin separating the positive and the negative examples in the training data. That's mean; the optimal hyperplane is then the one that consistently splits the margin between the two classes as shown in Figure 3(b).

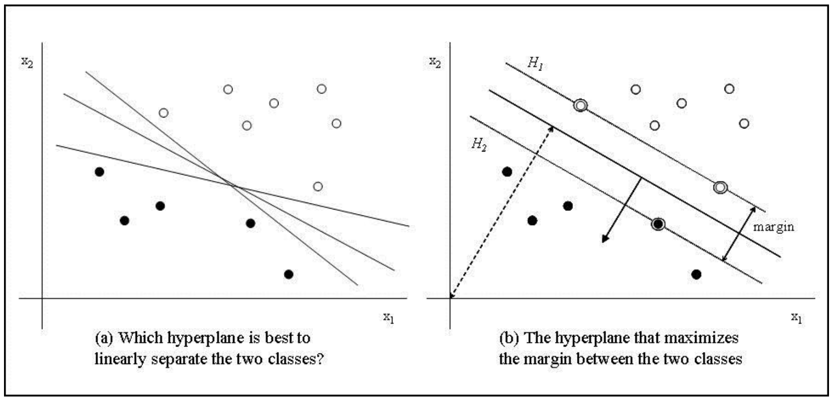

x2
x1
(a) Which hyperplane is best to linearly separate the two classes?
x2
H1
H2
margin
x1
(b) The hyperplane that maximizes the margin between the two classes

Figure 10: SVM Linearly Separable Case, Source:(Shigeo Abe 2005).

In Figure 3(b), the data points which are closest to the separating hyperplane are named support vectors. In mathematical expressions, the problem is to find $f(x) = (w^T x_i + b)$ with maximal margin, such as:

$$w^T x_i + b = 1 \text{ for data points that are support vectors}$$

$$w^T x_i + b > 1 \text{ for other data points}$$

Supposing a linearly separable dataset, the task of learning coefficients α and b of support vector machine $f(x) = (w^T x_i + b)$ decreases to solving the following constrained optimization problem:

$$\text{find } w \text{ and } b \text{ that minimize: } 1/2||w||^2$$

$$\text{subject to: } y_i(w^T x_i + b) \geq 1, \forall_i$$

Note that minimizing the inverse of the weights vector is equivalent to maximizing $f(x)$.

This optimization problem can be solved by using the Lagrangian function defined as:

$$L(w, b, \alpha) = \tfrac{1}{2}(w^T w) - \textstyle\sum_{i=1}^{N} \alpha_i[y_i(w^T x_i + b) - 1], \text{ such that } \alpha_i \geq 0, \forall_i$$

Where $\alpha_1, \alpha_2, ..., \alpha_N$ are Lagrange multipliers and $\alpha = [\alpha_1, \alpha_2, ..., \alpha_N]^T$.

The support vectors are those data points x_i with $\alpha_i > 0$, i.e., the data points within each class that are the closest to the separation margin.

Solving for the necessary optimization conditions results in:

$$w = \textstyle\sum_{i=1}^{N} \alpha_i y_i x_i$$

$$\text{where } w = \textstyle\sum_{i=1}^{N} \alpha_i y_i = 0$$

By replacing $w = \sum_{i=1}^{N} \alpha_i y_i x_i$ into the Lagrangian function and by using $\sum_{i=1}^{N} \alpha_i y_i = 0$ as a new constraint, the original optimization problem can be rewritten as its equivalent dual problem as follows:

$$\text{Find } \alpha \text{ that maximizes } \textstyle\sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j x_i^T x_j$$

$$\text{subject to } \textstyle\sum_{i=1}^{N} \alpha_i y_i = 0, \ \alpha_i \geq 0, \forall_i$$

The optimization problem is therefore a convex quadratic programming problem which has global minimum. This characteristic is a major advantage of support vector machines as compared to neural networks or decision trees. The optimization problem can be solved in $O(N^3)$ time, where N is the number of input data points.

5.2.3 Non-Linearly Separable case

In the non-linearly separable case, it is not potential to discover a linear hyperplane that separates all positive and negative examples. The solution for this case is the margin maximization approach which may be relaxed by allowing some data points to fall on the wrong side of the margin, i.e., to allow a degree of error in the separation. Slack Variables ξ_i are introduced to represent the error degree for each input data point. Figure (4) shows the non-linearly separable case where data points may fall into one of three options:

1. Points falling outside the margin that are correctly classified,with $\xi_i = 0$.
2. Points falling inside the margin that are correctly classified,with $0 < \xi_i < 1$.

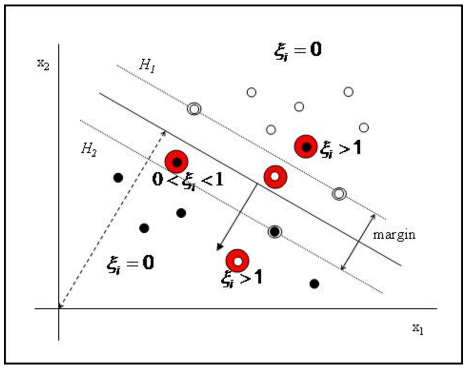


Figure 11: SVM Non-Linearly Separable Case, Source:(Shigeo Abe 2005.)

3. Points falling outside the margin and are incorrectly classified,with $\xi_i = 1$.

If all slack variables have a value of zero, the data is linearly separable. For the non-linearly separable case, some slack variables have nonzero values. In this case, the optimization should be used to maximize the margin while and at the same time minimizing the points with $\xi_i \neq 0.$, i.e., to minimize the margin error.

In other word, the optimization goal in mathematical terms becomes:

$$\text{find } w \text{ and } b \text{ that minimize: } 1/2||w||^2 + C\sum_i \xi_i^2$$

$$\text{subject to: } y_i(w^T x_i + b) \geq 1 - \xi_i,\ \xi_i \geq 0,\ \forall_i$$

Where C is an user defined parameters to enforce that all slack variables are close to zero as much as possible. Finding the most appropriate choice for C will depend on the input data set in use.

As in the linearly separable problem, this optimization problem can be converted to its dual problem:

$$\text{find } w \text{ that maximizes: } \sum_i \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j x_i^T x_j$$

$$\text{subject to:} \sum_{i=1}^{N} \alpha_i y_i = 0,\ 0 \leq \alpha_i \leq C,\ \forall_i$$

In order to solve the non-linearly separable case, SVM presents the use of mapping function $\phi : R^M \longrightarrow F$ to translate the non-linear input space into a higher dimension feature space where the data is linearly separable. Figure(5) shows an example of the effect of mapping the nonlinear input space into higher dimension linear feature space.

The dual problem is solved in feature space where its aim becomes to:

$$\text{find } \alpha \text{ that maximize } \sum_i \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j \phi(x_i)^T \phi(x_j)$$

$$\text{subject to } \sum_{i=1}^{N} \alpha_i y_i = 0,\ 0 \leq \alpha_i \leq C,\ \forall_i$$

The resulting SVM is of the form: $f(x) = w^T \phi(x_i) + b = \sum_{i=1}^{N} \alpha_i y_i \phi(x_i)^T \phi(x) + b$.

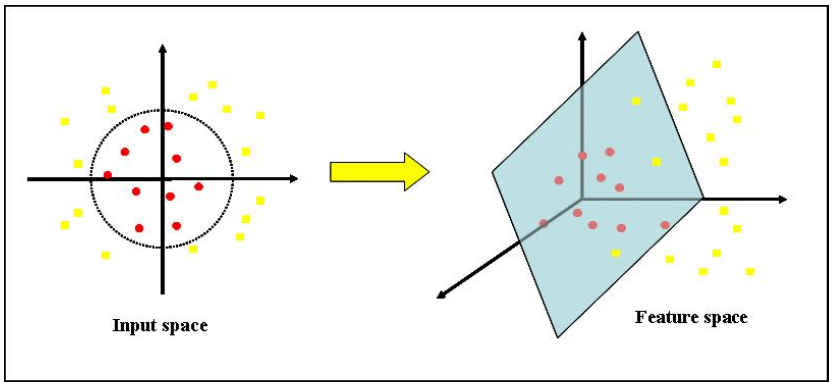


Figure 12: SVM Mapping to Higher Dimension Feature Space, Source:(Shigeo Abe 2005.)

5.2.4 The Kernel Trick

Mapping the input space into a higher dimension feature space transforms the nonlinear classification problem into a linear one that is more likely to be solved. However, the problem is more likely to face the curse of dimensionality. The kernel trick allows the computation of the vector product $\phi(x_i)^T\phi(x_j)$ in the lower dimension input space. From Mercer's theorem, there is a class of mappings ϕ such that $\phi(x)^T\phi(y) = K(x, y)$, where K is a corresponding kernel function. Being able to compute the vector products in the lower dimension input space while solving the classification problem in the linearly separable feature space is a major advantage of SVMs using a kernel function. The dual problem then becomes to:

$$\text{find } \alpha \text{ that maximize } \textstyle\sum_i \alpha_i - \frac{1}{2}\sum_i\sum_j \alpha_i\alpha_j y_i y_j K(x_i, x_j)$$

$$\text{subject to } \textstyle\sum_{i=1}^N \alpha_i y_i = 0,\ 0 \le \alpha_i \le C,\ \forall_i$$

The resulting SVM takes the form: $f(x) = w^T\phi(x_i) + b = \sum_{i=1}^N \alpha_i y_i K(x_i, x) + b$

The most common kernel function can be classified as follow:

- Linear kernel function (identity Kernel): $K(x, y) = (x^T y)$
- Polynomial kernel function with degree d: $K(x, y) = (\gamma x^T y + r)^d$, $\gamma > 0$
- Radial basis kernel function (RBF): $K(x, y) = \exp(-\gamma||x - y||^2)$,$\gamma > 0$
- Sigmoid kernel function: $K(x, y) = tanh(\gamma x^T y + r)$

Here, γ, r, and d are kernel parameters.

5.2.5 Experiments on Proof-Pattern Recognition by Support Vector Machines

This section explains the main experiments procedures which performed using SVM. Many users of SVM apply the basic procedures now which are transforming the data to the format of an SVM package, randomly try a few kernels and parameters, and test. In our experiments, we have used the following procedures:

- We transform the data format of an SVM package.
- We considered the RBF kernel function $K(x, y) = \exp(-\gamma||x - y||^2)$.
- We used the cross validation to find the best parameters C and γ.
- We used the the best parameter C and γ to train the data set.
- We tested the data.

All experiments including SVM were performed in Matlab by using Bioinformatics toolbox. We have used SVMCLASSIFY function to train an SVM model over training examples and SVMTRAIN function to test an SVM model over testing examples. In addition, we also have used CROSSVALIND function to set up cross validation function and CLASSPERF to evaluate the classifier performance. There is documentation available online in MathWorks website which explains how you can use these functions in more details.

We used data set with size of 400 examples of coinductive derivation tree for various experiments. All examples represent four different problems; we used these examples for train, test and validate the select SVM model. In addition, we applied two different cross validation methods which are K-Fold Cross Validation and Leave-One-Out Cross Validation. The main reason to apply two different approaches to validate the model is that most of problems have more positive examples than negative examples. As a result, we found out that Leave-One-Out Cross Validation method is more appropriate to validate SVM models for these problems than K-Fold Cross Validation. We will briefly present the main idea of two Cross Validation methods in following subsections:

5.3 K-Fold Cross Validation method

In this method, the original data set is randomly divided into k sub data sets. Single sub data set is taken out of the k sub data set as validation data for testing the selected model, and the remaining $(k-1)$ sub data sets are used for training the model. The cross validation procedure is then repeated k times, with each of the k sub data sets used exactly once as validation data. The k result from the folds then can be averaged or combined to produce a single estimation. The advantage of the method over repeated random sub data set is that all observations are used for both training and validation, and also each observation is used for validation exactly once. We used the following code to train, test and validate data set for Problem 1.

```
cvFolds = crossvalind('Kfold', 'Target_P5', 10);  %# get indices of 10-fold CV
cp = classperf(Target_P5);                          %# init performance tracker

for i = 1:10                                       %# for each fold
    testIdx = (cvFolds == i);                      %# get indices of test instances
    trainIdx = ~testIdx;                           %# get indices training instances

    %# train an SVM model over training instances
    svmModel = svmtrain(Input_P5(trainIdx,:), Target_P5(trainIdx), ...
                'Showplot',false, 'Method','QP', ...
                'Kernel_Function','rbf', 'RBF_Sigma',1);

    %# test using test instances
    pred = svmclassify(svmModel, Input_P5(testIdx,:), 'Showplot',false);

    %# evaluate and update performance object
    cp = classperf(cp, pred, testIdx);
end

%# get accuracy
cp.CorrectRate

ans =
      0.79333
```

6 Machine-learning proof patterns

Here, we test generality, accuracy, and robustness of the method of proof-tree feature representation on a range of classification tasks and implementation scenarios.

For the experiments of this section, we used data sets of various sizes - from 120 to 400 examples of coinductive trees for various experiments; we sampled trees produced for several distinct logic programs – such as `ListNat` and `Stream` above, see [27]. Finally, we repeated all experiments using three-layer neural networks of various sizes, and compared the results with those given by the SVMs with kernel functions.

Note that we deliberately did not tune the learning functions in neural networks to fit our symbolic data; but see e.g. [8, 39, 40].

Problem 1 (Classification of well-formed and ill-formed proofs)
Given a set of examples of well-formed and ill-formed coinductive trees, classify any new example of a coinductive tree in either of the two classes.

Figures 1, 2, 38, 44 show well-formed trees. Trees that do not conform to Definition 8 are ill-formed, see the numerous figures below for examples.

	Neural Net - `ListNat`	Neural net - `Stream`	SVM - `Stream`
Problem 1	76.4%	84.3 %	89 %
Problem 2.1	92.3%	99.1 %	88 %
Problem 2 Extended	96.3%		
Problem 2.2	n/a	90.6 %	88%
Problem 3	99 %	n/a	n/a
Problem 3 Extended	86 %	n/a	n/a
Problem 4	n/a	85.7 %	90%
Problem 5	82.4 %	82.4%	n/a

Figure 13: Summary of the best-average results of classification of coinductive proof trees for the classification problems of Section 6, performed in neural networks and SVMs.

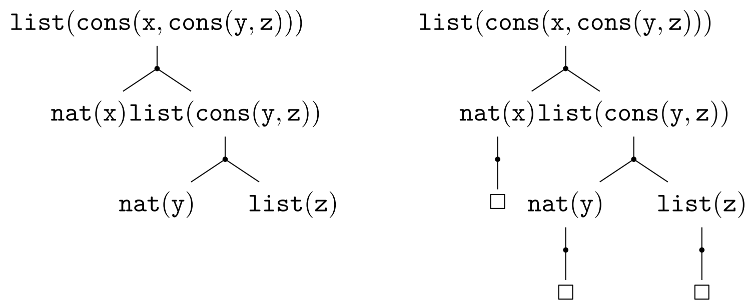


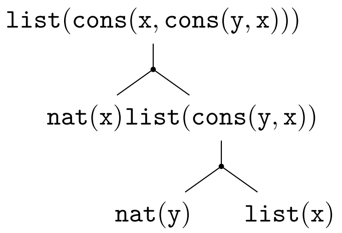
Figure 14: Well-formed and well-typed (left) and ill-formed (right) coinductive trees, well-formed trees are generated by the algorithm of Definition 8.

This task is one of the most difficult for pattern-recognition, due to a wide range of possible erroneous proofs compared to the correct ones. For experiments on this problem, we used a set of 117 examples of well-formed and ill-formed trees for `ListNat`, and 400 examples for `Stream`, the results are summarised in Figure 13. The samples of goals for forming the coinductive trees are given in Figures 26 and 27 - 28; and the sample trees are given in Figures 14 - 21. Some of these examples are quite challenging, and we deliberately chose them to possess a variety of structures and pattern, see Figures with example trees provided in this section.

The accuracy of classification for `ListNat` reached as high as 88.2% of accurate classifications, and on average it was 76.4% of accuracy for the best choice of the size of the hidden layer (50 neurons). The accuracy of classification for `Stream` was slightly higher, due to a bigger data set provided, but also due to a more regular structure of derivations. The best average was 84.3 % for ten re-training cycles.

We have also tried to perform similar test for `Stream` in SVM, which gave an average accuracy result of 89%. but the best test climbed as high as 100% accuracy.

A more interesting task in practical terms is recognition of various proof-families among well-formed proofs, as this is something that may help to opti-



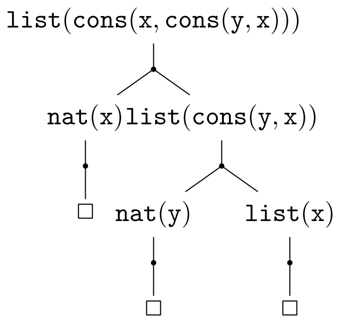
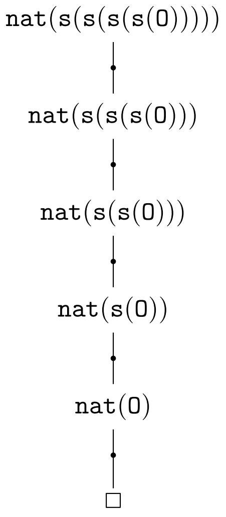


Figure 15: Well-formed and ill-typed (left) and ill-formed (right) coinductive trees; well-formed trees are generated by the algorithm of Definition 8.



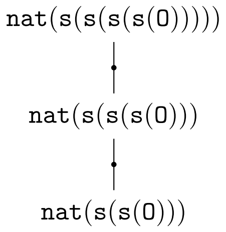


Figure 16: Well-formed and well-typed (left) and ill-formed (right) coinductive trees; well-formed trees are generated by the algorithm of Definition 8.

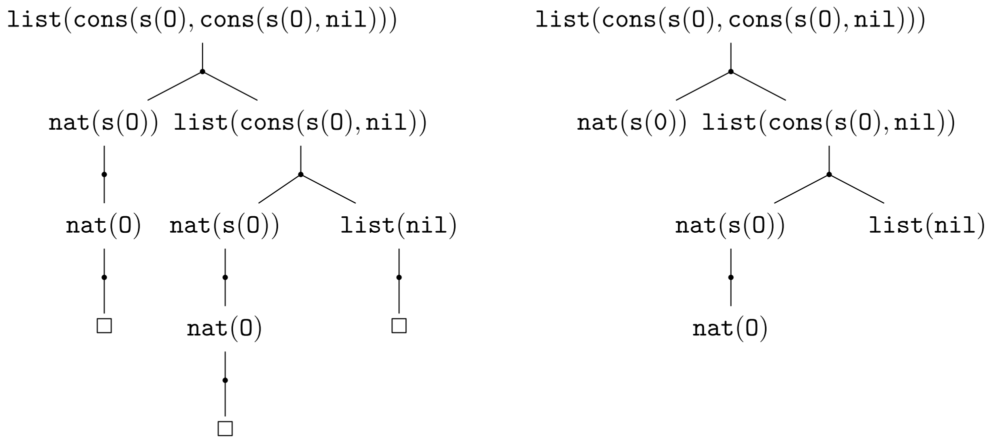


Figure 17: Well-formed and well-typed (left) and ill-formed (right) coinductive trees; well-formed trees are generated by the algorithm of Definition 8.

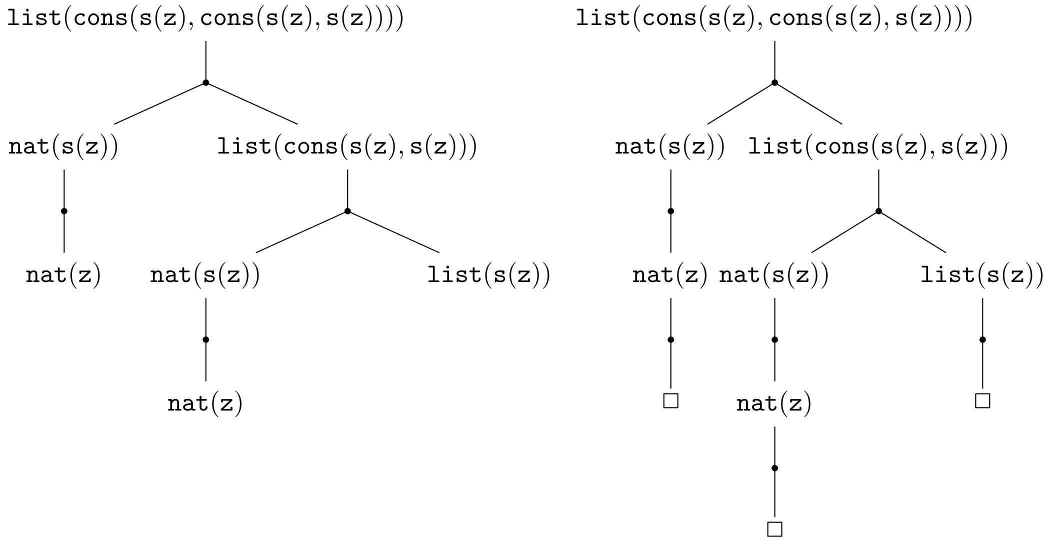


Figure 18: Well-formed and ill-typed (left) and ill-formed (right) coinductive trees; well-formed trees are generated by the algorithm of Definition 8.

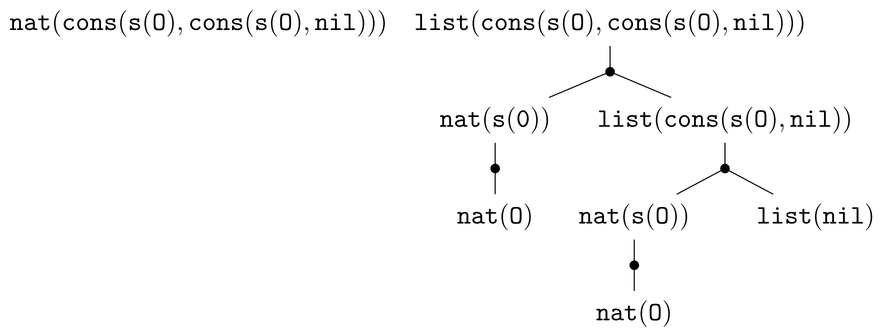

nat(cons(s(0), cons(s(0), nil)))
list(cons(s(0), cons(s(0), nil)))
nat(s(0))
list(cons(s(0), nil))
nat(0)
nat(s(0))
list(nil)
nat(0)

Figure 19: Well-formed and ill-typed (left) and ill-formed (right) coinductive trees; well-formed trees are generated by the algorithm of Definition 8.

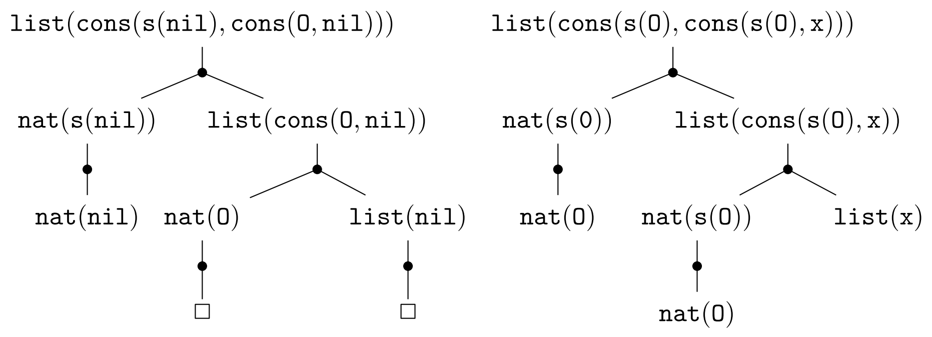


Figure 20: Well-formed and ill-typed (left) and ill-formed (right) coinductive trees; well-formed trees are generated by the algorithm of Definition 8.

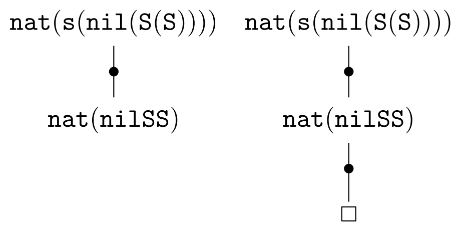


Figure 21: Well-formed and ill-typed (left) and ill-formed (right) coinductive trees; well-formed trees are generated by the algorithm of Definition 8.

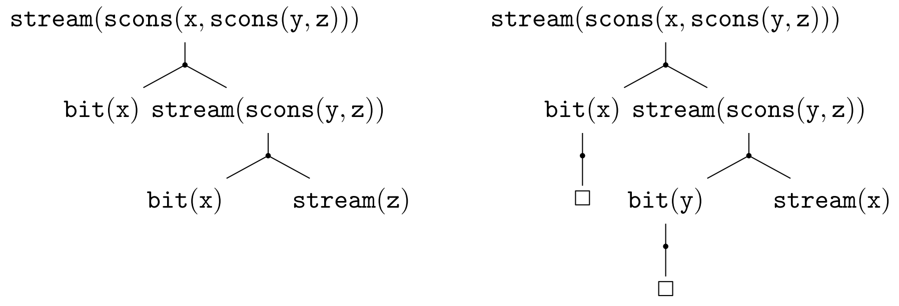

stream(scons(x, scons(y, z)))
bit(x) stream(scons(y, z))
bit(x) stream(z)
stream(scons(x, scons(y, z)))
bit(x) stream(scons(y, z))
□ bit(y) stream(x)
□

Figure 22: Well-formed and well-typed (left) and ill-formed (right) coinductive trees. Note that as opposed to proof-patterns in `ListNat` well-formed and well-typed proof-trees for `Stream` will always have the right-most branch incomplete.

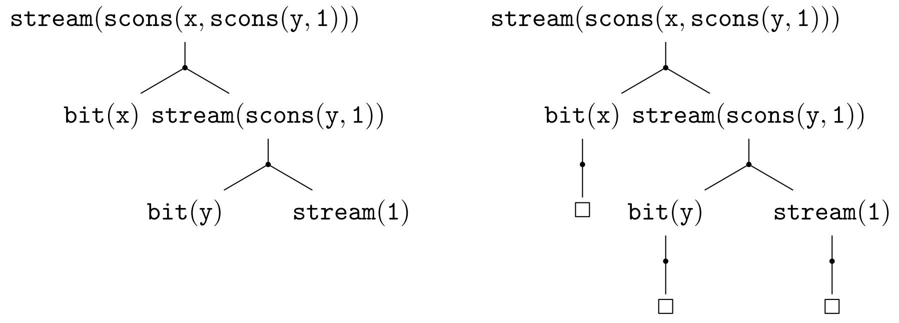

stream(scons(x, scons(y, 1)))
bit(x) stream(scons(y, 1))
bit(y) stream(1)
stream(scons(x, scons(y, 1)))
bit(x) stream(scons(y, 1))
□ bit(y) stream(1)
□ □

Figure 23: Well-formed and ill-typed (left) and ill-formed (right) coinductive trees.

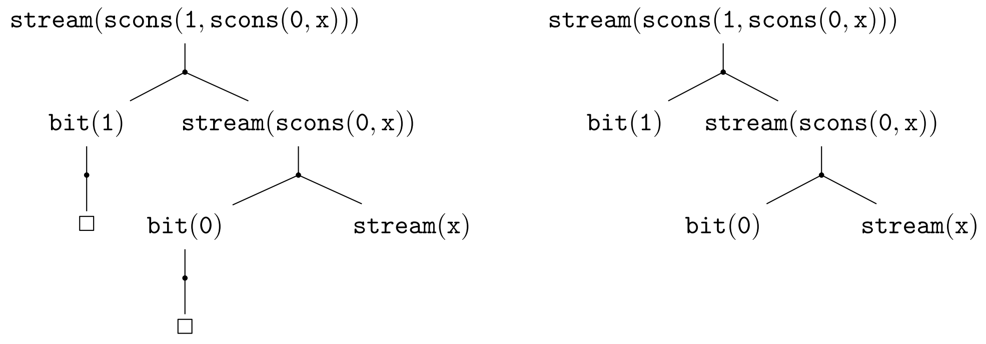

stream(scons(1, scons(0, x)))
bit(1) stream(scons(0, x))
□ bit(0) stream(x)
□
stream(scons(1, scons(0, x)))
bit(1) stream(scons(0, x))
bit(0) stream(x)

Figure 24: Well-formed and well-typed (left) and ill-formed (right) coinductive trees.

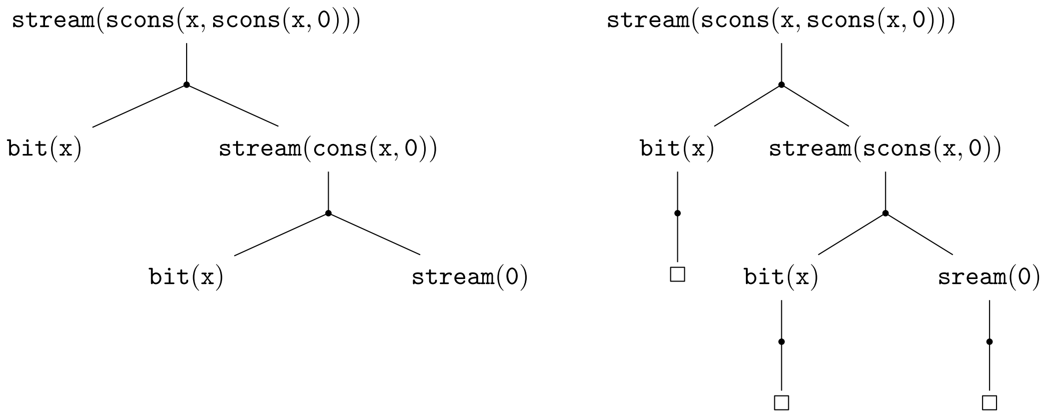


Figure 25: Well-formed and ill-typed (left) and ill-formed (right) coinductive trees.

Derivation goals	derivation goals
list(cons(x,y))	list(cons(Sx, cons(nil, nil)))
list(cons(sO, cons (SO, nil)))	list(cons(Sx, cons(y,nil)))
list(cons(x, cons(y,z))	list(cons(x, cons(S0,S0))
list(cons(nil,SO))	nat(SSSSO)
list(cons(S(nil), cons(O, nil)))	nat(SSSSnil)
list(cons(SO,cons(O,nil)))	nat(SSO)
nat(SSSSx)	nat(S(nil(SS)))

Figure 26: The table contains some of the goals which were used to generate coinductive proof trees (cf. Definition 8) for the Problem 1; program `ListNat`. The examples of well-formed and ill-formed coinductive trees for such goals are given in this section. There are 137 examples like the above used for the experiments in Figure 13; the data set is also available at [27].

mize proof-search.

Problem 2 (Discovery of proof families) *Given a set of positive and negative examples of well-formed coinductive trees belonging to a proof family, classify any new example of a coinductive tree, whether it belongs to the given family.*

Definition 22 *Given a logic program P, and an atomic formula A, we say that a tree T belongs to the* family of coinductive trees determined by *A, if*

- *T is a coinductive tree with root A' and*
- *there is a substitution θ such that $A\theta = A'$.*

Example 23 *The three trees in Figure 1 belong to the family of proofs determined by `list(cons(x,cons(y,z)))`.*

Determining whether a given tree belongs to a certain proof family has practical applications. For Figure 1, knowing that the right-hand tree belongs to the

Goals/Positive Examples	Goals/Positive Examples
1.stream(x)	51.stream(scons(z,scons(z,z)))
2.stream(0)	52.stream(scons(x,scons(z,z)))
3.stream(1)	53.stream(scons(0,scons(z,z)))
4.stream(scons(x,y))	54.stream(scons(x,scons(x,0)))
5.stream(scons(1,x))	55.stream(scons(0,scons(y,z)))
6.stream(scons(0,x))	56.stream(scons(x,scons(z,0)))
7.stream(scons(x,1))	57.stream(scons(x,scons(1,z)))
8.stream(scons(x,0))	58.stream(scons(1,scons(z,z)))
9.stream(scons(0,0))	59.stream(scons(1,scons(y,z)))
10.stream(scons(1,1))	60.stream(scons(1,scons(1,z)))
11.stream(scons(0,1))	61.stream(scons(1,scons(y,y)))
12.stream(scons(1,0))	62.stream(scons(x,scons(1,x)))
13.stream(scons(x,scons(y,z)))	63.stream(scons(x,scons(x,1)))
14.stream(scons(0,scons(z,y)))	64.bit(scons(0,scons(1,x)))
15.stream(scons(x,scons(0,y)))	65.bit(scons(1,scons(1,x)))
16.stream(scons(x,scons(y,0)))	66.bit(scons(0,scons(0,x)))
17.stream(scons(1,scons(y,z)))	67.bit(scons(1,scons(0,x)))
18.stream(scons(x,scons(1,z)))	68.bit(scons(x,scons(x,x)))
19.stream(scons(x,scons(y,1)))	69.bit(scons(x,scons(0,x)))
20.stream(scons(1,scons(1,1)))	70.bit(scons(0,scons(0,x)))
21.stream(scons(0,scons(0,0)))	71.bit(scons(x,scons(y,z)))
22.stream(scons(x,scons(1,1)))	72.bit(scons(x,scons(x,y)))
23.stream(scons(x,scons(0,0)))	73.bit(scons(y,scons(y,y)))
24.stream(scons(x,scons(1,0)))	74.bit(scons(z,scons(z,z)))
25.stream(scons(x,scons(0,1)))	75.bit(scons(x,scons(x,0)))
26.stream(scons(1,scons(1,scons(x,1)))	76.bit(scons(0,scons(z,z)))
27.stream(y)	77.bit(scons(x,scons(0,z)))
28.stream(z)	78.bit(scons(x,scons(y,0)))
29.stream(scons(x,scons(x,x)))	79.bit(scons(x,scons(1,z)))
30.stream(scons(y,scons(y,y)))	80.bit(scons(1,scons(y,z)))
31.stream(scons(z,scons(z,z)))	81.bit(scons(1,scons(1,z)))
32.stream(scons(x,scons(y,0)))	82.bit(scons(1,scons(z,z)))
33.stream(scons(x,scons(y,1)))	83.bit(scons(x,scons(1,x)))
34.stream(scons(1,scons(x,1)))	84.bit(scons(x,scons(y,1)))
35.stream(scons(1,scons(0,1)))	85.stream(0)
36.stream(scons(0,scons(1,1)))	86.stream(scons(0,scons(0,0)))
37.stream(scons(1,scons(0,0)))	87.stream(scons(1,scons(1,1)))
38.stream(scons(x,scons(x,scons(y,y)))	88.stream(scons(0,scons(1,0)))
39.stream(scons(x,scons(x,scons(x,0)))	89.stream(scons(0,scons(0,1)))
40.stream(scons(x,scons(x,scons(0,y)))	90.stream(scons(1,scons(0,0)))
41.stream(scons(1,scons(1,scons(0,y)))	91.stream(scons(1,scons(1,0)))
42.stream(scons(0,scons(0,scons(1,1)))	92.stream(scons(1,scons(0,1)))
43.stream(scons(0,scons(1,scons(y,y)))	93.stream(scons(0,scons(1,1)))
44.stream(scons(0,scons(1,y)))	94.bit(scons(0,scons(0,0)))
45.stream(scons(1,scons(1,y)))	95.bit(scons(1,scons(1,1)))
46.stream(scons(0,scons(0,y)))	96.bit(scons(0,scons(1,0)))
47.stream(scons(1,scons(0,y)))	97.bit(scons(1,scons(0,0)))
48.stream(scons(x,scons(x,x)))	98.bit(scons(1,scons(1,0)))
49.stream(scons(x,scons(0,x)))	99.bit(scons(1,scons(0,1)))
50.stream(scons(y,scons(y,y)))	100.bit(scons(0,scons(1,1)))

Figure 27: The table contains the goals which were used to generate coinductive proof trees (cf. Definition 8) for the Problem 1; program `Stream`. The examples of well-formed and ill-formed coinductive trees for such goals are given in this section. The data set is also available at [27].

Goals/Positive Examples	Goals/Positive Examples
101.stream(1)	151.stream(scons(0,scons(1,scons(1,1)))
102.stream(scons(0,z))	152.stream(scons(1,scons(0,scons(1,1)))
103.stream(scons(z,z))	153.stream(scons(1,scons(1,scons(0,1)))
104.stream(scons(y,1))	154.stream(scons(1,scons(1,scons(1,0)))
105.stream(scons(1,x))	155.stream(scons(1,scons(0,scons(0,0)))
106.stream(scons(0,y))	156.stream(scons(0,scons(1,scons(0,0)))
107.stream(scons(1,1))	157.stream(scons(0,scons(0,scons(1,0)))
108.stream(scons(0,0))	158.stream(scons(0,scons(0,scons(0,1)))
109.stream(scons(1,0))	159.stream(scons(0,scons(0,scons(1,1)))
110.stream(scons(0,1))	160.stream(scons(1,scons(1,scons(0,0)))
111.stream(scons(0,1))	161.stream(scons(1,scons(0,scons(0,1)))
112.bit(scons(0,x))	162.stream(scons(0,scons(1,scons(1,0)))
113.bit(scons(1,x))	163.stream(scons(x,scons(1,0)))
114.bit(scons(x,x))	164.stream(scons(x,scons(0,1)))
115.bit(scons(y,y))	165.stream(scons(1,scons(1,scons(y,1)))
116.bit(scons(z,z))	166.stream(y)
117.bit(scons(y,1))	167.stream(z)
118.bit(scons(z,0))	168.stream(scons(x,scons(x,x)))
119.bit(scons(1,1))	169.stream(scons(y,scons(y,y)))
120.bit(scons(0,0))	170.stream(scons(z,scons(z,z)))
121.bit(scons(1,0))	171.stream(scons(x,scons(y,0)))
122.bit(scons(0,1))	172.stream(scons(x,scons(y,1)))
123.stream(y)	173.stream(scons(1,scons(y,1)))
124.stream(scons(0,scons(0,scons(0,z)))	174.stream(scons(1,scons(0,1)))
125.stream(scons(1,scons(1,scons(1,z)))	175.stream(scons(0,scons(1,1)))
126.stream(scons(1,scons(1,scons(0,z)))	176.stream(scons(1,scons(0,0)))
127.stream(scons(1,scons(0,scons(1,z)))	177.stream(scons(x,scons(x,scons(x,x)))
128.stream(scons(0,scons(0,scons(1,z)))	178.stream(scons(y,scons(y,scons(y,0)))
129.stream(scons(1,scons(0,scons(0,z)))	179.stream(scons(x,scons(x,scons(0,z)))
130.stream(scons(0,scons(1,scons(0,z)))	180.stream(scons(1,scons(1,scons(0,z)))
131.stream(scons(0,scons(0,scons(1,z)))	181.stream(scons(0,scons(0,scons(1,1)))
132.stream(scons(x,scons(0,scons(0,z)))	182.stream(scons(0,scons(1,scons(z,z)))
133.stream(scons(x,scons(1,scons(1,0)))	183.stream(scons(0,scons(0,scons(0,z)))
134.stream(scons(x,scons(1,scons(1,0)))	184.stream(scons(1,scons(1,scons(1,z)))
135.stream(scons(x,scons(0,scons(1,0)))	185.stream(scons(1,scons(1,scons(0,z)))
136.stream(scons(x,scons(1,scons(1,1)))	186.stream(scons(1,scons(0,scons(1,z)))
137.stream(scons(x,scons(0,scons(0,0)))	187.stream(scons(0,scons(1,scons(1,z)))
138.stream(scons(x,scons(1,scons(0,0)))	188.stream(scons(1,scons(0,scons(0,z)))
139.stream(scons(x,scons(0,scons(1,1)))	189.stream(scons(0,scons(1,scons(0,z)))
140.stream(scons(0,scons(y,scons(0,1)))	190.stream(scons(0,scons(0,scons(1,z)))
141.stream(scons(1,scons(y,scons(1,0)))	191.stream(scons(x,scons(0,scons(0,1)))
142.stream(scons(1,scons(y,scons(0,1)))	192.stream(scons(x,scons(1,scons(1,0)))
143.stream(scons(0,scons(y,scons(1,0)))	193.stream(scons(x,scons(1,scons(0,1)))
144.stream(scons(1,scons(y,scons(1,1)))	194.stream(scons(x,scons(0,scons(1,0)))
145.stream(scons(0,scons(y,scons(0,0)))	195.stream(scons(x,scons(1,scons(0,1)))
146.stream(scons(1,scons(y,scons(0,0)))	196.stream(scons(x,scons(1,scons(1,1)))
147.stream(scons(0,scons(y,scons(1,0)))	197.stream(scons(x,scons(1,scons(0,0)))
148.stream(scons(0,scons(y,scons(1,1)))	198.stream(scons(x,scons(0,scons(1,1)))
149.stream(scons(1,scons(1,scons(1,1)))	199.stream(scons(0,scons(1,scons(0,1)))
150.stream(scons(0,scons(0,scons(0,0)))	200.stream(scons(1,scons(y,scons(1,0)))

Figure 28: The table contains the goals which were used to generate coinductive proof trees (cf. Definition 8) for the Problem 1; program `Stream`. The examples of well-formed and ill-formed coinductive trees for such goals are given in this section. The data set is also available at [27].

same family as the left-hand-side tree would save the intermediate derivation step. Note that unlike [39], it's not the shape of a formula, but proof patterns it induces that influences the classification. Moreover, according to Definition 11, determining such intermediate trees requires unification algorithm, which does not yield parallelisation [12].

For the program `ListNat`, we used neural networks of different sizes, and 60 examples (**the original set**) and 255 examples (**the extended set**) of trees classified as we used positive and negative examples of proofs belonging the proof family of `list(cons(x,cons(y,z)))`, cf. Figures 1 and 38, for the glimpse of how the example trees may look, and Figures 29 – 29 ; and [27] for the full list of the goals used to generate such trees. The accuracy of the pattern recognition for the smaller set showed the best average of 92.3%; and 96.3% for the extended set. For the program `Stream`, we tested two families of proofs, induced by `stream(scons(x,scons(y,z)))` and `stream(scons(x,scons(x,x)))` (cf. Problems 2.1 and 2.2 in Figures 36 – 37 and 33 – 35). Some of the coinductive trees from the former family also belong to the latter family, but not conversely, which made them interesting candidates for the experiments. We significantly increased the number of training examples to 193 and 143 for the two experiments. The results in Neural networks and SVMs were very robust, see Table 13.

Problem 3 (Discovery of potentially successful proofs) *Given a set of positive and negative examples of well-formed coinductive trees belonging to a success family, classify any new example of a coinductive tree accordingly.*

Definition 22 conveys the idea of coinductive derivations, but it may not exactly capture the common intuition of what a proof is. E.g., a coinductive tree for `list(cons(x,cons(y,x))))` will belong to the same proof-family as trees of Figure 1, however, the formula will never be proven, even if the coinductive tree induces several further derivations steps, such as shown in Figure 38.

We used 60 (**the original set**) and 255 (**the extended set**) examples of various trees, belonging or not belonging to the success family of `list(cons(x,cons(y,z))))` ; see Figures 40 – 43. The average accuracy was 99% and 86 % for the small and extended sets of examples respectively.

Definition 24 *We say that a proof-family F is a* success family *if, for all $T \in F$, T generates a proof-family that contains a success subtree (cf. Definition 9).*

Proposition 25 *Given a coinductive tree T, there exists a success family F such that $T \in F$ if and only if T has a successful derivation.*

This problem has solutions only for inductive definitions and corresponding coinductive trees. Therefore, we tested it only for `ListNat`. We used neural-network pattern-recognition tool to recognise trees from the success family of

Goals/positive examples	Goals/negative examples
10. list(cons(x, cons (y,z))	1. nat(ssss0)
11. list(cons(0, cons (y,nil))	2. nat(ssssnil)
12. list(cons(0, cons (0,nil))	3. nat(s s (cons(0,nil)))
13. list(cons(x, cons (y,nil))	4. nat(ssssx)
14. list(cons(x, cons (0,nil))	5. nat(s nil sss)
15. list(cons(x, cons (0,y))	6. nat(cons(x, cons(y,z)))
16. list(cons(0, cons (y,z))	7. nat(cons(x,y))
17. list(cons(0, cons (0,y))	8. nat(nil)
18. list(cons(x, cons (s0,nil))	9. nat(x)
19. list(cons(0, cons (s0,nil))	50 list(cons(0,nil))
20. list(cons(s0, cons (x,nil))	51. list(cons(x,nil))
21. list(cons(s0, cons (0,nil))	52. list(cons(nil,x))
22. list(cons(s0, cons (s0,nil))	53. list(cons(x,0))
23. list(cons(x, cons (s0,y))	54. list(cons(nil,0))
24. list(cons(0, cons (s0,x))	55. list(x)
25. list(cons(s0, cons (x,y))	56. list(nil)
26. list(cons(s0, cons (0,y))	57. list(0)
27. list(cons(s0, cons (s0,nil))	61. nat(s(s(s(s(s(0))))))
28. list(cons(x,cons(nil,y)))	62. nat(s(s(s(s(s(x))))))
29. list(cons(x,cons(nil,nil)))	63. nat(s(s(s(0))))
30. list(cons(nil,cons(x,nil)))	64. nat(s(s(0)))
31. list(cons(nil,cons(nil,nil)))	65. nat(s(0))
32. list(cons(x,cons(y,0)))	66. nat(0)
33. list(cons(0,cons(y,0)))	
34. list(cons(0,cons(0,0)))	
35. list(cons(x,cons(0,0)))	
36. list(cons(x,cons(nil,0)))	
37. list(cons(0,cons(nil,x)))	
38. list(cons(0,cons(nil,0)))	
39. list(cons(0,cons(nil,nil)))	
40. list(cons(nil,cons(0,nil)))	
41. list(cons(nil,cons(x,y)))	
42. list(cons(nil,cons(0,x)))	
43. list(cons(nil,cons(x,0)))	
44. list(cons(nil,cons(0,0)))	
45. list(cons(nil,cons(nil,x)))	
46. list(cons(nil,cons(x,nil)))	
47. list(cons(nil,cons(nil,0)))	
58. list(cons(x,cons(y,x)))	
59. list(cons(x,cons(x,x)))	
60. list(cons(x,cons(y,y)))	

Figure 29: The table contains the goals which were used to generate examples coinductive proof trees (cf. Definition 8) for the pattern-recognition Problem 2; program `ListNat`; and proof0family of `list(x,cons(y,z))`. This is the **original** (smaller) data set; it is also available at [27].

Goals/positive examples	Goals/negative examples
90. list(cons(x,cons(x,cons(x,y))))	61. nat(s(s(s(s(s(0))))))
91. list(cons(x,cons(y,cons(y,z))))	62. nat(s(s(s(s(s(x))))))
92. list(cons(x,cons(x,cons(x, cons(x,y)))))	63. nat(s(s(s(0))))
93. list(cons(x,cons(x,cons(y,z))))	64. nat(s(s(0)))
94. list(cons(x,cons(x,cons(x,x))))	65. nat(s(0))
95. list(cons(x,cons(x,cons(x, cons(x,x)))))	66. nat(0)
96. list(cons(x,cons(x,cons(x,cons(x, cons(x,x))))))	67. nat(x(s(s(s(s(0))))))
118. list(cons(s(x)),cons(x,y))	68. nat(s(s(s(s(s(nil))))))
98. list(cons(x,cons(x,cons(y,cons(x,y)))))	69. nat(s(s(s(nil))))
99. list(cons(x,cons(y,cons(x,cons(x,y)))))	70. nat(s(s(nil)))
100. list(cons(y,cons(x,cons(x,z))))	71. nat(s(nil))
101. list(cons(0,cons(x,cons(x,y))))	72. nat(s(s(s(s(s(x))))))
102. list(cons(x,cons(0,cons(x,y))))	73. nat(s(s(s(0))))
103. list(cons(0,cons(0,cons(x,y))))	74. nat(s(s(x)))
104. list(cons(0,cons(0,cons(0,y))))	75. nat(s(x))
105. list(cons(0,cons(0,cons(0,nil))))	76. nat(s(x(s(s(s(x))))))
106. list(cons(s(0),cons(x,z)))	77. nat(s(x(s(s(s(0))))))
107. list(cons(x,cons(s(0),y)))	78. nat(s(s(x(s(s(x))))))
108. list(cons(s(0),cons(x,nil)))	79. nat(s(s(x(s(s(0))))))
109. list(cons(x,cons(s(0),nil)))	80. nat(s(0(s(s(s(0))))))
110. list(cons(x,cons(x,cons(x,nil))))	81. nat(s(s(0(s(s(0))))))
111. list(cons(x,cons(x,cons(x,cons(x,nil)))))	82. nat(0(s(s(s(s(0))))))
112. list(cons(x,cons(y,cons(x,nil))))	83. nat(s(s(s(0(s(0))))))
113. list(cons(x,cons(0,cons(x,nil))))	84. nat(s(s(s(s(0(0))))))
114. list(cons(0,cons(0,cons(x,nil))))	85. nat(s(s(s(s))))
115. list(cons(0,cons(0,cons(0,nil))))	86. nat(s(s(s(0(0(0))))))
119. list(cons(s(x)),cons(x,nil))	87. nat(s(s(0(0(0(0))))))
120. list(cons(x),cons(s(x),y))	88. nat(s(s(0(s(s(x))))))
121. list(cons(s(x)),cons(s(x),y))	89. nat(s(s(0(s(x)))))
122. list(cons(x),cons(s(x),nil))	97. list(cons(x,x))
123. list(cons(s(x),cons(s(x),nil)))	116. list(cons(s(s(0)),nil))
124. list(cons(s(x),cons(s(x),cons(s(x),y))))	117. list(cons(s(s(x)),nil))
125. list(cons(s(x),cons(s(x),cons(s(x),nil))))	97. list(cons(x,x))
126. list(cons(s(x),cons(x,cons(x,y))))	181. list(s(s(0)),nil)
127. list(cons(s(x),cons(x,cons(x,nil))))	182. list(s(s(x)),nil)
128. list(cons(x,cons(s(x),cons(x,y))))	228. list(s(s(s(0))),nil)
129. list(cons(x,cons(s(x),cons(x,nil))))	229. list(s(s(s(x))),nil)
130. list(cons(x,cons(x,cons(s(x),y))))	230. list(s(s(s(0))),x)
131. list(cons(x,cons(x,cons(s(x),nil))))	231. list(s(s(s(x))),y)
132. list(cons(s(0),cons(x,y)))	232. list(s(s(s(x))),x)
133. list(cons(s(0),cons(0,y))))	233. list(s(s(x)),x)
134. list(cons(s(0),cons(x,nil)))	234. list(s(s(s(s(x)))),s(x))
135. list(cons(s(0),cons(0,nil))))	235. list(s(s(s(s(x)))),s(s(x)))
136. list(cons(s(0),cons(s(x),y))))	236. list(s(s(s(s(x)))),s(s(s(x))))
137. list(cons(0,cons(s(0),y))))	237. list(s(s(s(s(x)))),s(s(s(s(x)))))
138. list(cons(0,cons(s(x),nil))))	238. list(x,s(s(s(s(x)))))
139. list(cons(0,cons(s(0),nil))))	253. list(s(s(x)),s(s(s(s(x)))))
140. list(cons(s(0),cons(s(y),y))))	254. list(s(s(s(x))),s(s(s(s(x)))))

Figure 30: The table contains the goals which were used to generate the **extended set of** examples of coinductive proof trees (cf. Definition 8) for the pattern-recognition Problem 2; program `ListNat`; and proof-family of `list(x,cons(y,z))`. The the data set is also available at [27].

Goals/positive examples	Goals/negative examples
141. list(cons(s(0),cons(s(0),y))))	
142. list(cons(s(0),cons(x,nil))))	
143. list(cons(s(0),cons(s(0),nil))))	
144. list(cons(s(0),cons(s(0),cons(s(0),y))))	
145. list(cons(s(0),cons(s(0),cons(s(0),nil))))	
146. list(cons(s(0),cons(0,cons(0,y))))	
147. list(cons(s(0),cons(0,cons(0,nil))))	
148. list(cons(0,cons(s(0),cons(0,y))))	
149. list(cons(0,cons(s(0),cons(0,nil))))	
150. list(cons(0,cons(0,cons(s(0),y))))	
151. list(cons(0,cons(0,cons(s(0),nil))))	
152. list(cons(y,cons(x,cons(x,nil))))	
153. list(cons(nil,cons(y,cons(x,nil))))	
154. list(cons(nil,cons(nil,cons(x,nil))))	
155. list(cons(nil,cons(nil,cons(nil,nil))))	
156. list(cons(nil,cons(nil,cons(nil,cons(nil,nil)))))	
157. list(cons(nil,cons(nil,cons(nil,cons(nil,cons(nil,nil))))))	
158. list(cons(nil,cons(x,cons(x,y))))	
159. list(cons(nil,cons(x,cons(y,x))))	
160. list(cons(x,cons(nil,cons(y,x))))	
161. list(cons(x,cons(nil,cons(x,y))))	
162. list(cons(x,cons(nil,cons(x,cons(x,x)))))	
163. list(cons(x,cons(nil,cons(x,x))))	
164. list(cons(s(0),cons(s(0),cons(s(0),s(0)))))	
165. list(cons(s(0),cons(0,cons(0,s(0)))))	
166. list(cons(nil,cons(0,cons(0,s(0)))))	
167. list(cons(nil,cons(s(0),cons(0,0))))	
168. list(cons(nil,cons(s(0),cons(s(0),nil))))	
169. list(cons(s(0),cons(nil,cons(s(0),nil))))	
170. list(cons(s(0),cons(s(0),cons(nil,nil))))	
171. list(cons(s(s(0)),cons(s(0),nil)))	
172. list(cons(s(s(0)),cons(0,nil)))	
173. list(cons(0,cons(s(s(0)),nil)))	
174. list(cons(s(s(0)),cons(s(s(0)),nil)))	
175. list(cons(s(0),cons(s(s(0)),nil)))	
176. list(cons(s(s(0)),cons(s(0),x)))	
177. list(cons(s(s(0)),cons(0,x)))	
178. list(cons(0,cons(s(s(0)),x)))	
179. list(cons(s(s(0)),cons(s(s(0)),x)))	
180. list(cons(s(0),cons(s(s(0)),x)))	
183. list(cons(s(s(x))),cons(s(s(x)),nil)))	
184. list(cons(s(s(0)),cons(x,x)))	
185. list(cons(x,cons(s(s(0)),x)))	
186. list(cons(s(s(x)),cons(s(x),nil)))	
187. list(cons(s(s(x)),cons(x,nil)))	
188. list(cons(x,cons(s(s(x)),nil)))	
189. list(cons(s(s(x)),cons(s(s(x)),nil)))	
190. list(cons(s(x),cons(s(s(x)),nil)))	
191. list(cons(s(s(x)),cons(s(x),y)))	
192. list(cons(s(s(x)),cons(x,y)))	
193. list(cons(x,cons(s(s(x)),y)))	
195. list(cons(s(x),cons(s(s(x)),y)))	

Figure 31: The table contains the goals which were used to generate the **extended set of** examples of coinductive proof trees (cf. Definition 8) for the pattern-recognition Problem 2; program `ListNat`; and proof-family of `list(x,cons(y,z))`. The the data set is also available at [27].

Goals/positive examples	Goals/negative examples
196. list(cons(s(s(x)),cons(s(x),x)))	
197. list(cons(s(s(x)),cons(x,x)))	
198. list(cons(x,cons(s(s(x)),x)))	
199. list(cons(s(s(x)),cons(s(s(x)),x)))	
200. list(cons(s(x),cons(s(s(x)),x)))	
201. list(cons(s(s(s(x))),cons(s(s(s(x))),x)))	
202. list(cons(s(s(x)),cons(s(x),x)))	
203. list(cons(s(s(x)),cons(x,s(x))))	
204. list(cons(x,cons(s(s(x)),s(x))))	
205. list(cons(s(s(x)),cons(s(s(x)),s(x))))	
206. list(cons(s(x),cons(s(s(x)),s(x))))	
207. list(cons(s(s(x)),cons(s(x),s(s(x)))))	
208. list(cons(s(s(x)),cons(x,s(s(x)))))	
209. list(cons(x,cons(s(s(x)),s(s(x)))))	
210. list(cons(s(s(x)),cons(s(s(x)),s(s(x)))))	
211. list(cons(s(x),cons(s(s(x)),s(s(x)))))	
212. list(cons(s(s(s(x))),cons(s(s(s(x))),s(s(s(x))))))	
213. list(cons(s(s(s(x))),cons(s(s(x)),s(s(s(x))))))	
214. list(cons(s(s(s(x))),cons(s(s(s(x))),s(s(x)))))	
215. list(cons(s(s(x)),cons(s(s(s(x))),s(s(s(x))))))	
216. list(cons(s(s(s(x))),cons(s(s(s(x))),x)))	
217. list(cons(x,cons(s(s(s(x))),s(s(s(x))))))	
218. list(cons(s(s(s(x))),cons(x,s(s(s(x))))))	
219. list(cons(s(s(s(x))),cons(s(s(s(x))),s(x))))	
220. list(cons(s(s(s(x))),cons(s(x),s(s(s(x))))))	
221. list(cons(s(x),cons(s(s(s(x))),s(s(s(x))))))	
222. list(cons(s(s(s(x))),cons(s(s(s(x))),s(x))))	
223. list(cons(s(s(s(x))),cons(s(x),x)))	
224. list(cons(s(s(s(x))),cons(s(x),s(x))))	
225. list(cons(s(x),cons(s(s(s(x))),x)))	
226. list(cons(s(s(s(x))),cons(s(s(x)),x)))	
227. list(cons(s(s(x)),cons(s(s(s(x))),x)))	
239. list(cons(0,cons(0, cons(0,0))))	
240. list(cons(0,cons(0, cons(0, cons(0,0)))))	
241. list(cons(0,cons(0, cons(0, cons(0,cons(0,0))))))	
242. list(cons(0,cons(x, cons(x,x))))	
243. list(cons(0,cons(x, cons(y,x))))	
244. list(cons(0,cons(x, cons(y,y))))	
245. list(cons(x,cons(0, cons(y,y))))	
246. list(cons(y,cons(0, cons(x,y))))	
247. list(cons(x,cons(0, cons(x,x))))	
248. list(cons(0,cons(x, cons(x,0))))	
249. list(cons(0,cons(x, cons(y,0))))	
250. list(cons(x,cons(x, cons(y,0))))	
251. list(cons(x,cons(y, cons(0,0))))	
252. list(cons(x,cons(0, cons(0,0))))	
255. list(cons(0,cons(x, cons(x,cons(x,x)))))	

Figure 32: The table contains the goals which were used to generate the **extended set of** examples of coinductive proof trees (cf. Definition 8) for the pattern-recognition Problem 2; program `ListNat`; and proof-family of `list(x,cons(y,z))`. The the data set is also available at [27].

Goals/Positive Examples	Goals/Negative Examples
13.stream(scons(x,scons(y,z)))	1.stream(x)
14.stream(scons(0,scons(y,z)))	2.stream(0)
15.stream(scons(x,scons(0,z)))	3.stream(1)
16.stream(scons(x,scons(y,0)))	4.stream(scons(x,y))
17.stream(scons(1,scons(y,z)))	5.stream(scons(1,y)
18.stream(scons(x,scons(1,z)))	6.stream(scons(0,y))
19.stream(scons(x,scons(y,1)))	7.stream(scons(x,1))
20.stream(scons(1,scons(1,1)))	8.stream(scons(x,0))
21.stream(scons(0,scons(0,0)))	9.stream(scons(0,0)))
22.stream(scons(x,scons(1,1)))	10.stream(scons(1,1)))
23.stream(scons(x,scons(0,0)))	11.stream(scons(0,1))
24.stream(scons(x,scons(1,0)))	12.stream(scons(1,0))
25.stream(scons(x,scons(0,1)))	27.stream(y)
26.stream(scons(1,(scons(1,scons(z,1)))	28.stream(scons(x,x))
30.stream(scons(x,(scons(x,x)))	29.stream(scons(y,y))
31.stream(scons(y,(scons(y,y)))	136.stream(y)
32.stream(scons(z,(scons(z,z)))	137.bit(scons(0,1))
33.stream(scons(x,(scons(x,0)))	138.stream(z)
34.stream(scons(x,(scons(x,1)))	139.bit(scons(y,1)).
35.stream(scons(1,(scons(x,1)))	69.stream(1)
36.stream(scons(1,(scons(0,1)))	70.stream(scons(x,x))
37.stream(scons(0,(scons(0,1)))	71.stream(scons(x,1))
38.stream(scons(0,(scons(1,1)))	72.stream(scons(x,0))
39.stream(scons(1,(scons(0,0)))	73.stream(scons(1,1))
40.stream(scons(x,scons(x,scons(y,z))))	74.stream(scons(0,0))
41.stream(scons(x,scons(x,scons(y,0))))	75.stream(scons(1,0))
42.stream(scons(x,scons(x,scons(0,x))))	76.stream(scons(0,1))
43.stream(scons(1,scons(1,scons(0,x))))	77.bit(scons(0,x)
44.stream(scons(0,scons(0,scons(1,1))))	78.bit(scons(1,x))
45.stream(scons(0,scons(1,scons(y,y))))	79.bit(scons(x,y))
46.stream(scons(0,scons(1,y)))	80.bit(scons(x,x))
47.stream(scons(1,scons(1,y)))	81.bit(scons(y,x))
48.stream(scons(0,scons(0,y)))	82.bit(scons(y,1))
48.stream(scons(1,scons(0,y)))	83.bit(scons(0,scons(1,x)))
49.stream(scons(1,scons(1,y)))	84.bit(scons(1,scons(1,x)))
50.stream(scons(x,scons(0,x)))	85.bit(scons(0,scons(0,x)))
51.stream(scons(x,scons(z,z)))	86.bit(scons(1,scons(0,x)))
52.stream(scons(0,scons(z,z))	87.bit(scons(x,scons(x,x)))
53.stream(scons(x,scons(x,0)))	88.bit(scons(x,scons(0,x)))
54.stream(scons(0,scons(y,z)))	89.bit(scons(0,scons(0,x)))
55.stream(scons(x,scons(z,0)))	90.bit(scons(x,scons(y,z)))
56.stream(scons(x,scons(z,0)))	91.bit(scons(x,scons(x,y)))
57.stream(scons(x,scons(1,z)))	92.bit(scons(y,scons(y,y)))
58.stream(scons(1,scons(y,z)))	93.bit(scons(z,scons(z,z)))
59.stream(scons(1,scons(1,z)))	94.bit(scons(x,scons(x,0)))
95.stream(scons(0,scons(0,scons(0,0))))	114.bit(scons(0,scons(0,0)))
96.stream(scons(0,scons(1,scons(1,1))))	115.bit(scons(1,scons(1,1)))
97.stream(scons(1,scons(0,scons(1,1))))	116.bit(scons(0,scons(1,0)))
98.stream(scons(1,scons(1,scons(0,1))))	117.bit(scons(1,scons(0,0)))

Figure 33: The table contains some of the goals which were used to generate examples of coinductive proof trees (cf. Definition 8) for the pattern-recognition Problem 2.1; for program `Stream`; and proof-family induced by `stream(scons(x,scons(y,z)))` . The the data set is also available at [27].

Goals/Positive Examples	Goals/Negative Examples
99.stream(scons(1,scons(1,scons(1,0))))	118.bit(scons(1,scons(1,0)))
100.stream(scons(1,scons(0,scons(0,0))))	119.bit(scons(1,scons(0,1)))
101.stream(scons(0,scons(1,scons(0,0))))	120.bit(scons(0,scons(1,1)))
102.stream(scons(0,scons(0,scons(1,0))))	127.stream(scons(0,z))
103.stream(scons(0,scons(0,scons(0,1))))	128.stream(scons(z,z))
104.stream(scons(0,scons(0,scons(1,1))))	129.bit(scons(0,x))
105.stream(scons(1,scons(1,scons(0,0))))	130.bit(scons(1,x))
106.stream(scons(1,scons(0,scons(0,1))))	131.stream(scons(0,y))
107.stream(scons(0,scons(1,scons(1,1))))	132.bit(scons(x,x))
109.stream(scons(1,scons(x,scons(1,1))))	133.bit(scons(y,y))
110.stream(scons(0,scons(x,scons(0,0))))	134.bit(scons(z,z))
111.stream(scons(1,scons(x,scons(0,0))))	135.bit(scons(z,z))
112.stream(scons(0,scons(x,scons(1,1))))	
113.stream(scons(1,scons(1,scons(1,1))))	
121.stream(scons(1,scons(1,y)))	
122.stream(scons(0,scons(0,y)))	
123.stream(scons(1,scons(0,y)))	
124.stream(scons(x,scons(x,x)))	
125.stream(scons(x,scons(0,x)))	
126.stream(scons(y,scons(y,y)))	
140.stream(scons(0,scons(0,scons(0,z)))	
141.stream(scons(1,scons(1,scons(1,z)))	
142.stream(scons(1,scons(1,scons(0,z)))	
143.stream(scons(1,scons(0,scons(1,z)))	
144.stream(scons(0,scons(1,scons(1,z)))	
145.stream(scons(1,scons(0,scons(0,z)))	
146.stream(scons(0,scons(1,scons(0,z)))	
147.stream(scons(0,scons(0,scons(1,z)))	
148.stream(scons(x,scons(0,scons(0,z)))	
149.stream(scons(x,scons(1,scons(1,0)))	
150.stream(scons(x,scons(1,scons(1,0)))	
151.stream(scons(x,scons(0,scons(1,0)))	
152.stream(scons(x,scons(1,scons(1,1)))	
153.stream(scons(x,scons(0,scons(0,0)))	
154.stream(scons(x,scons(1,scons(0,0)))	
155.stream(scons(x,scons(0,scons(1,1)))	
156.stream(scons(0,scons(y,scons(0,1)))	
157.stream(scons(1,scons(y,scons(1,0)))	
158.stream(scons(1,scons(y,scons(0,1)))	
159.stream(scons(0,scons(y,scons(1,0)))	
160.stream(scons(1,scons(y,scons(1,1)))	
161.stream(scons(0,scons(y,scons(0,0)))	
162.stream(scons(1,scons(y,scons(0,0)))	
163.stream(scons(0,scons(y,scons(1,0)))	
164.stream(scons(0,scons(y,scons(1,1)))	
165.stream(scons(1,scons(1,scons(1,1)))	
166.stream(scons(0,scons(0,scons(0,0)))	
167.stream(scons(1,scons(0,1)))	
168.stream(scons(0,scons(1,1)))	

Figure 34: The table contains some of the goals which were used to generate examples of coinductive proof trees (cf. Definition 8) for the pattern-recognition Problem 2.1; for program `Stream`; and proof-family induced by `stream(scons(x,scons(y,z)))`. The the data set is also available at [27].

Goals/Positive Examples	Goals/Negative Examples
169.stream(scons(1,scons(0,0)))	
170.stream(scons(x,scons(x,scons(x,x)))	
171.stream(scons(y,scons(y,scons(y,0)))	
172.stream(scons(x,scons(x,scons(0,z)))	
173.stream(scons(1,scons(1,scons(0,z)))	
174.stream(scons(0,scons(0,scons(1,1)))	
175.stream(scons(0,scons(1,scons(z,z)))	
176.stream(scons(0,scons(0,scons(0,z)))	
177.stream(scons(1,scons(1,scons(1,z)))	
178.stream(scons(1,scons(1,scons(0,z)))	
179.stream(scons(1,scons(0,scons(1,z)))	
180.stream(scons(0,scons(1,scons(1,z)))	
181.stream(scons(1,scons(0,scons(0,z)))	
182.stream(scons(0,scons(1,scons(0,z)))	
183.stream(scons(0,scons(0,scons(1,z)))	
184.stream(scons(x,scons(0,scons(0,1)))	
185.stream(scons(x,scons(1,scons(1,0)))	
186.stream(scons(x,scons(1,scons(0,1)))	
187.stream(scons(x,scons(0,scons(1,0)))	
188.stream(scons(x,scons(1,scons(0,1)))	
189.stream(scons(x,scons(1,scons(1,1)))	
190.stream(scons(x,scons(1,scons(0,0)))	
191.stream(scons(x,scons(0,scons(1,1)))	
192.stream(scons(0,scons(1,scons(0,1)))	
193.stream(scons(1,scons(y,scons(1,0)))	

Figure 35: The table contains some of the goals which were used to generate examples of coinductive proof trees (cf. Definition 8) for the pattern-recognition Problem 2.1; for program `Stream`; and proof-family induced by `stream(scons(x,scons(y,z)))`. The the data set is also available at [27].

Positive Examples	Negative Examples
1.stream(scons(scons(x,x),scons(scons(x,x),scons(x,x))))	37.stream(scons(0,scons(1,z)))
2.stream(scons(scons(0,0),scons(scons(0,0),scons(0,0))))	38.stream(scons(1,scons(1,z)))
3.stream(scons(scons(1,1),scons(scons(1,1),scons(1,1))))	39.stream(scons(0,scons(0,z)))
4.stream(scons(scons(y,y),scons(scons(y,y),scons(y,y))))	40.stream(scons(1,scons(0,z)))
5.stream(scons(scons(z,z),scons(scons(z,z),scons(z,z))))	42.stream(scons(0,scons(1,z)))
6.stream(scons(scons(x,scons(x,x))),scons(scons(x,scons(x,x))),scons(x,scons(x,x))))	43.stream(scons(x,scons(y,z)))
7.stream(scons(scons(1,0),scons(scons(1,0),scons(1,0))))	44.stream(scons(x,scons(x,z)))
8.stream(scons(scons(0,1),scons(scons(0,1),scons(0,1))))	45.stream(scons(x,scons(y,z)))
9.stream(scons(scons(1,x),scons(scons(1,x),scons(1,x))))	46.stream(scons(x,scons(z,z)))
10.stream(scons(scons(x,1),scons(scons(x,1),scons(x,1))))	47.stream(scons(0,scons(y,z)))
11.stream(scons(scons(0,x),scons(scons(0,x),scons(0,x))))	48.stream(scons(x,scons(y,z)))
12.stream(scons(scons(x,0),scons(scons(x,0),scons(x,0))))	49.stream(scons(x,scons(0,z)))
13.stream(scons(scons(0,scons(0,0))),scons(scons(0,scons(0,0))),scons(0,scons(0,0))))	50.stream(scons(x,scons(y,0)))
14.stream(scons(scons(0,scons(x,x))),scons(scons(0,scons(x,x))),scons(0,scons(x,x))))	51.stream(scons(x,scons(1,z)))
15.stream(scons(scons(1,scons(x,x))),scons(scons(1,scons(x,x))),scons(1,scons(x,x))))	52.stream(scons(1,scons(y,z)))
16.stream(scons(scons(x,scons(0,x))),scons(scons(x,scons(0,x))),scons(x,scons(0,x))))	53.stream(scons(1,scons(1,z)))
17.stream(scons(scons(x,scons(1,x))),scons(scons(x,scons(1,x))),scons(x,scons(1,x))))	54.stream(scons(1,scons(z,z)))
18.stream(scons(scons(x,scons(0,x))),scons(scons(x,scons(0,x))),scons(x,scons(0,x))))	55.stream(scons(x,scons(1,x)))
19.stream(scons(scons(x,scons(x,1))),scons(scons(x,scons(x,1))),scons(x,scons(x,1))))	56.stream(scons(x,scons(x,1)))
20.stream(scons(scons(0,scons(0,x))),scons(scons(0,scons(0,x))),scons(0,scons(0,x))))	57.bit(scons(0,scons(1,z)))
21.stream(scons(scons(1,scons(1,x))),scons(scons(1,scons(1,x))),scons(1,scons(1,x))))	58.bit(scons(1,scons(1,z)))
22.stream(scons(scons(0,scons(x,0))),scons(scons(0,scons(x,0))),scons(0,scons(x,0))))	59.bit(scons(0,scons(0,z)))
23.stream(scons(scons(1,scons(x,1))),scons(scons(1,scons(x,1))),scons(1,scons(x,1))))	60.bit(scons(1,scons(0,z)))
24.stream(scons(scons(x,scons(1,1))),scons(scons(x,scons(1,1))),scons(x,scons(1,1))))	61.bit(scons(x,scons(y,z)))
25.stream(scons(scons(x,scons(0,0))),scons(scons(x,scons(0,0))),scons(x,scons(0,0))))	62.bit(scons(x,scons(0,y)))
26.stream(scons(scons(0,scons(1,1))),scons(scons(0,scons(1,1))),scons(0,scons(1,1))))	63.bit(scons(x,scons(x,z)))
27.stream(scons(scons(1,scons(0,0))),scons(scons(1,scons(0,0))),scons(1,scons(0,0))))	64.bit(scons(x,scons(z,z)))
28.stream(scons(scons(0,scons(0,1))),scons(scons(0,scons(0,1))),scons(0,scons(0,1))))	65.bit(scons(x,scons(y,y)))
29.stream(scons(scons(0,scons(1,0))),scons(scons(0,scons(1,0))),scons(0,scons(1,0))))	66.bit(scons(x,scons(y,x)))
30.stream(scons(scons(1,scons(0,1))),scons(scons(1,scons(0,1))),scons(1,scons(0,1))))	67.bit(scons(x,scons(x,0)))
31.stream(scons(scons(0,scons(1,0))),scons(scons(0,scons(1,0))),scons(0,scons(1,0))))	68.bit(scons(0,scons(z,z)))
32.stream(scons(scons(1,scons(1,0))),scons(scons(1,scons(1,0))),scons(1,scons(1,0))))	69.bit(scons(0,scons(1,z)))
33.stream(scons(x,scons(x,x)))	70.bit(scons(x,scons(y,0)))
34.stream(scons(y,scons(y,y)))	71.bit(scons(x,scons(1,z)))
35.stream(scons(1,scons(1,1)))	72.bit(scons(1,scons(x,z)))
36.stream(scons(0,scons(0,0)))	73.bit(scons(1,scons(1,x)))
41.stream(scons(z,scons(z,z)))	74.bit(scons(1,scons(y,z)))
77.stream(scons(1,1))	75.bit(scons(x,scons(1,x)))
78.stream(scons(0,0))	76.bit(scons(x,scons(y,1)))
	79.stream(0)
	80.stream((scons(0,scons(1,0))))
	81.stream((scons(0,scons(0,1))))
	82.stream((scons(1,scons(0,0))))
	83.stream((scons(1,scons(1,0))))
	84.stream((scons(1,scons(0,1))))
	85.stream((scons(0,scons(1,1))))
	86.bit(scons(0,scons(0,0))
	87.bit(scons(1,scons(1,1)))
	88.bit(scons(0,scons(1,0)))
	89.bit(scons(0,scons(0,1)))
	90.bit(scons(1,scons(0,0))))
	91.bit(scons(1,scons(1,0)))

Figure 36: The table contains some of the goals which were used to generate examples of coinductive proof trees (cf. Definition 8) for the pattern-recognition Problem 2.2; for program `Stream`; and proof-family induced by `stream(scons(x,scons(x,x)))`. The the data set is also available at [27].

Positive Examples	Negative Examples
	92.bit(scons(0,scons(0,1)))
	93.bit(scons(0,scons(1,1)))
	94.stream(1)
	95.stream(scons(0,x))
	96.stream(scons(1,x))
	97.stream(scons(x,y))
	98.stream(scons(x,x))
	99.stream(scons(x,1))
	100.stream(scons(x,0))
	101.stream(scons(1,0))
	102.stream(scons(0,1)
	105.bit(scons(0,z))
	106.bit(scons(1,z))
	107.bit(scons(z,z)).
	108.bit(scons(x,x))
	109.bit(scons(y,y))
	110.bit(scons(x,1))
	111.bit(scons(x,0))
	112.bit(scons(1,1))
	113.bit(scons(0,0))
	114.bit(scons(1,0))
	115.bit(scons(0,1))
	116.stream(z)
	117.stream(scons(0,scons(0,scons(0,x)))
	118.stream(scons(1,scons(1,scons(1,x)))
	119.stream(scons(1,scons(1,scons(0,x)))
	120.stream(scons(1,scons(0,scons(1,x)))
	121.stream(scons(0,scons(0,scons(0,x)))
	122.stream(scons(0,scons(1,scons(1,x)))
	123.stream(scons(1,scons(0,scons(0,x)))
	124.stream(scons(0,scons(0,scons(0,x)))
	125.stream(scons(x,scons(0,scons(0,1)))
	126.stream(scons(x,scons(1,scons(0,1)))
	127.stream(scons(x,scons(0,scons(1,0)))
	128.stream(scons(x,scons(0,scons(1,0)))
	129.stream(scons(x,scons(1,scons(1,1)))
	130.stream(scons(x,scons(0,scons(0,0)))
	140.stream(scons(scons(x,scons(x,x)),scons(scons(x,x),scons(x,x))))
	141.stream(scons(scons(x,x)),scons(scons(1,1),scons(1,1))))
	142.stream(scons(scons(1,1)),scons(scons(x,(x,x)),scons(x,(x,x))))
	143.stream(scons(scons(0,0)),scons(scons(0,0),scons(1,1))))

Figure 37: The table contains some of the goals which were used to generate examples of coinductive proof trees (cf. Definition 8) for the pattern-recognition Problem 2.2; for program `Stream`; and proof-family induced by `stream(scons(x,scons(x,x)))`. The the data set is also available at [27].

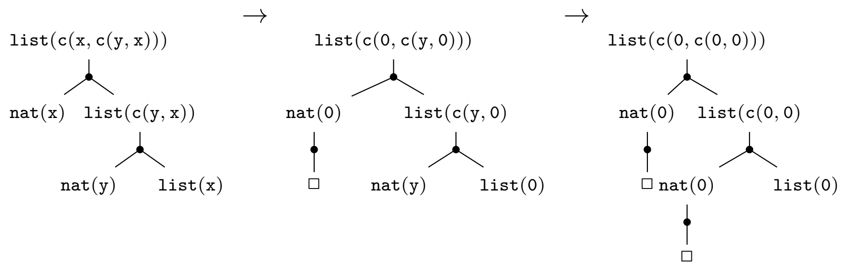


Figure 38: The unsuccessful derivation and ill-typed proof family for the program ListNat and the goal `list(cons(x,cons(x,x)))`. We abbreviate `cons` by `c`.

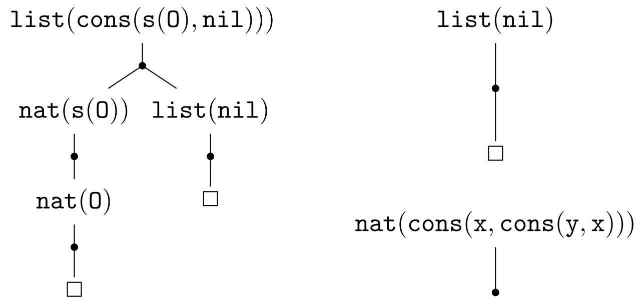


Figure 39: Well-formed trees (right) that do not belong to the family on the left.

`list(cons(x,cons(y,z)))`, that is, trees like the ones given in Figure 38 were negative examples for the training purposes, and trees akin Figure 1 were given as positive examples. Bearing in mind subtlety of the notion of a success family, the accuracy of classification was astonishing, cf. Figure 13.

Problem 4 (Discovery of ill-typed proofs in a proof family) *Given a set of positive and negative examples of ill-typed coinductive trees belonging to a given proof family, classify any new example, whether it is ill-typed or well-typed.*

When it comes to coinductive logic programs like `Stream`, detection of success families is impossible, see Figures 2 and 44. In such cases, detection of well-typed and ill-typed proofs within a proof family will be an alternative to determining success families. Figures 38 and 44 show ill-typed members of the proof families induced by the trees from Figure 1 and 2. Definition 26 below emulates the notion of ill-typing suitable for the (co-)inductive logic programs used in this paper. It may have to be re-adjusted for different kinds of programs in the future, but we found the current definition fit for our purposes.

Definition 26 *Given a goal G, a program P, and a coinductive tree t for G and P, we say that T (or G) is ill-typed if there is a term $t \in G$, and there are*

Goals/positive examples	Goals/negative examples
10. list(cons(x, cons (y,z))	1. nat(ssss0)
11. list(cons(0, cons (y,nil))	2. nat(ssssnil)
12. list(cons(0, cons (0,nil))	3. nat(s s (cons(0,nil)))
13. list(cons(x, cons (y,nil))	4. nat(ssssx)
14. list(cons(x, cons (0,nil))	5. nat(s nil sss)
15. list(cons(x, cons (0,y))	6. nat(cons(x, cons(y,z)))
16. list(cons(0, cons (y,z))	7. nat(cons(x,y))
17. list(cons(0, cons (0,y))	8. nat(nil)
18. list(cons(x, cons (s0,nil))	9. nat(x)
19. list(cons(0, cons (s0,nil))	50 list(cons(0,nil))
20. list(cons(s0, cons (x,nil))	51. list(cons(x,nil))
21. list(cons(s0, cons (0,nil))	52. list(cons(nil,x))
22. list(cons(s0, cons (s0,nil))	53. list(cons(x,0))
23. list(cons(x, cons (s0,y))	54. list(cons(nil,0))
24. list(cons(0, cons (s0,x))	55. list(x)
25. list(cons(s0, cons (x,y))	56. list(nil)
26. list(cons(s0, cons (0,y))	57. list(0)
	27. list(cons(s0, cons (s0,nil))
	33. list(cons(0,cons(y,0)))
	29. list(cons(x,cons(nil,nil)))
	34. list(cons(0,cons(0,0)))
	35. list(cons(x,cons(0,0)))
	30. list(cons(nil,cons(x,nil)))
	36. list(cons(x,cons(nil,0)))
	37. list(cons(0,cons(nil,x)))
	31. list(cons(nil,cons(nil,nil)))
	38. list(cons(0,cons(nil,0)))
	39. list(cons(0,cons(nil,nil)))
	28. list(cons(x,cons(nil,y)))
	40. list(cons(nil,cons(0,nil)))
	41. list(cons(nil,cons(x,y)))
	32. list(cons(x,cons(y,0)))
	42. list(cons(nil,cons(0,x)))
	43. list(cons(nil,cons(x,0)))
	44. list(cons(nil,cons(0,0)))
	45. list(cons(nil,cons(nil,x)))
	46. list(cons(nil,cons(x,nil)))
	47. list(cons(nil,cons(nil,0)))
	58. list(cons(x,cons(y,x)))
	59. list(cons(x,cons(x,x)))
	60. list(cons(x,cons(y,y)))

Figure 40: The table contains the goals which were used to generate examples coinductive proof trees (cf. Definition 8) for the pattern-recognition Problem 3; program `ListNat`; and **success proof-family** of `list(x,cons(y,z))`. This is the **original** (smaller) data set; it is also available at [27].

Goals/positive examples	Goals/negative examples
90. list(cons(x,cons(x,cons(x,y))))	61. nat(s(s(s(s(s(0))))))
91. list(cons(x,cons(y,cons(y,z))))	62. nat(s(s(s(s(s(x))))))
92. list(cons(x,cons(x,cons(x, cons(x,y)))))	63. nat(s(s(s(0))))
93. list(cons(x,cons(x,cons(y,z))))	64. nat(s(s(0)))
134. list(cons(s(0),cons(x,nil)))	65. nat(s(0))
135. list(cons(s(0),cons(0,nil))))	66. nat(0)
136. list(cons(s(0),cons(s(x),y))))	67. nat(x(s(s(s(s(0))))))
118. list(cons(s(x)),cons(x,y))	68. nat(s(s(s(s(s(nil))))))
137. list(cons(0,cons(s(0),y))))	69. nat(s(s(s(nil))))
138. list(cons(0,cons(s(x),nil))))	70. nat(s(s(nil)))
100. list(cons(y,cons(x,cons(x,z))))	71. nat(s(nil))
101. list(cons(0,cons(x,cons(x,y))))	72. nat(s(s(s(s(s(x))))))
102. list(cons(x,cons(0,cons(x,y))))	73. nat(s(s(s(0))))
103. list(cons(0,cons(0,cons(x,y))))	74. nat(s(s(x)))
104. list(cons(0,cons(0,cons(0,y))))	75. nat(s(x))
105. list(cons(0,cons(0,cons(0,nil))))	76. nat(s(x(s(s(s(x))))))
106. list(cons(s(0),cons(x,z)))	77. nat(s(x(s(s(s(0))))))
107. list(cons(x,cons(s(0),y)))	78. nat(s(s(x(s(s(x))))))
108. list(cons(s(0),cons(x,nil)))	79. nat(s(s(x(s(s(0))))))
109. list(cons(x,cons(s(0),nil)))	80. nat(s(0(s(s(s(0))))))
110. list(cons(x,cons(x,cons(x,nil))))	81. nat(s(s(0(s(s(0))))))
111. list(cons(x,cons(x,cons(x,cons(x,nil)))))	82. nat(0(s(s(s(s(0))))))
112. list(cons(x,cons(y,cons(x,nil))))	83. nat(s(s(s(0(s(0))))))
113. list(cons(x,cons(0,cons(x,nil))))	84. nat(s(s(s(s(0(0))))))
114. list(cons(0,cons(0,cons(x,nil))))	85. nat(s(s(s(s))))
115. list(cons(0,cons(0,cons(0,nil))))	86. nat(s(s(s(0(0(0))))))
119. list(cons(s(x)),cons(x,nil))	87. nat(s(s(0(0(0(0))))))
120. list(cons(x),cons(s(x),y))	88. nat(s(s(0(s(s(x))))))
121. list(cons(s(x)),cons(s(x),y))	89. nat(s(s(0(s(x)))))
122. list(cons(x),cons(s(x),nil))	97. list(cons(x,x))
123. list(cons(s(x),cons(s(x),nil)))	116. list(cons(s(s(0)),nil))
124. list(cons(s(x),cons(s(x),cons(s(x),y))))	117. list(cons(s(s(x)),nil))
125. list(cons(s(x),cons(s(x),cons(s(x),nil))))	97. list(cons(x,x))
126. list(cons(s(x),cons(x,cons(x,y))))	181. list(s(s(0)),nil)
127. list(cons(s(x),cons(x,cons(x,nil))))	182. list(s(s(x)),nil)
128. list(cons(x,cons(s(x),cons(x,y))))	94. list(cons(x,cons(x,cons(x,x))))
129. list(cons(x,cons(s(x),cons(x,nil))))	95. list(cons(x,cons(x,cons(x, cons(x,x)))))
130. list(cons(x,cons(x,cons(s(x),y))))	96. list(cons(x,cons(x,cons(x,cons(x, cons(x,x))))))
131. list(cons(x,cons(x,cons(s(x),nil))))	98. list(cons(x,cons(x,cons(y,cons(x,y)))))
132. list(cons(s(0),cons(x,y)))	99. list(cons(x,cons(y,cons(x,cons(x,y)))))
133. list(cons(s(0),cons(0,y))))	140. list(cons(s(0),cons(s(y),y))))
139. list(cons(0,cons(s(0),nil))))	228. list(s(s(s(0))),nil)
141. list(cons(s(0),cons(s(0),y))))	229. list(s(s(s(x))),nil)
142. list(cons(s(0),cons(x,nil))))	230. list(s(s(s(0))),x)
143. list(cons(s(0),cons(s(0),nil))))	231. list(s(s(s(x))),y)
144. list(cons(s(0),cons(s(0),cons(s(0),y))))	232. list(s(s(s(x))),x)
152. list(cons(y,cons(x,cons(x,nil))))	233. list(s(s(x)),x)
145. list(cons(s(0),cons(s(0),cons(s(0),nil))))	234. list(s(s(s(s(x)))),s(x))
146. list(cons(s(0),cons(0,cons(0,y))))	235. list(s(s(s(s(x)))),s(s(x)))

Figure 41: The table contains the goals which were used to generate the **extended set of** examples of coinductive proof trees (cf. Definition 8) for the pattern-recognition Problem 3; program `ListNat`; and **success proof-family** of `list(x,cons(y,z))`. The the data set is also available at [27].

Goals/positive examples	Goals/negative examples
147. list(cons(s(0),cons(0,cons(0,nil))))	236. list(s(s(s(s(x)))),s(s(s(x))))
148. list(cons(0,cons(s(0),cons(0,y))))	237. list(s(s(s(s(x)))),s(s(s(s(x)))))
149. list(cons(0,cons(s(0),cons(0,nil))))	238. list(x,s(s(s(s(x)))))
150. list(cons(0,cons(0,cons(s(0),y))))	253. list(s(s(x)),s(s(s(s(x)))))
151. list(cons(0,cons(0,cons(s(0),nil))))	254. list(s(s(s(x))),s(s(s(s(x)))))
171. list(cons(s(s(0)),cons(s(0),nil)))	153. list(cons(nil,cons(y,cons(x,nil))))
172. list(cons(s(s(0)),cons(0,nil)))	154. list(cons(nil,cons(nil,cons(x,nil))))
173. list(cons(0,cons(s(s(0)),nil)))	155. list(cons(nil,cons(nil,cons(nil,nil))))
174. list(cons(s(s(0)),cons(s(s(0)),nil)))	156. list(cons(nil,cons(nil,cons(nil,cons(nil,nil)))))
175. list(cons(s(0),cons(s(s(0)),nil)))	157. list(cons(nil,cons(nil,cons(nil,cons(nil,cons(nil,nil))))))
176. list(cons(s(s(0)),cons(s(0),x)))	158. list(cons(nil,cons(x,cons(x,y))))
177. list(cons(s(s(0)),cons(0,x)))	159. list(cons(nil,cons(x,cons(y,x))))
178. list(cons(0,cons(s(s(0)),x)))	160. list(cons(x,cons(nil,cons(y,x))))
179. list(cons(s(s(0)),cons(s(s(0)),x)))	161. list(cons(x,cons(nil,cons(x,y))))
180. list(cons(s(0),cons(s(s(0)),x)))	162. list(cons(x,cons(nil,cons(x,cons(x,x)))))
186. list(cons(s(s(x)),cons(s(x),nil)))	163. list(cons(x,cons(nil,cons(x,x))))
187. list(cons(s(s(x)),cons(x,nil)))	164. list(cons(s(0),cons(s(0),cons(s(0),s(0)))))
188. list(cons(x,cons(s(s(x)),nil)))	165. list(cons(s(0),cons(0,cons(0,s(0)))))
189. list(cons(s(s(x)),cons(s(s(x)),nil)))	166. list(cons(nil,cons(0,cons(0,s(0)))))
190. list(cons(s(x),cons(s(s(x)),nil)))	167. list(cons(nil,cons(s(0),cons(0,0))))
191. list(cons(s(s(x)),cons(s(x),y)))	168. list(cons(nil,cons(s(0),cons(s(0),nil))))
192. list(cons(s(s(x)),cons(x,y)))	169. list(cons(s(0),cons(nil,cons(s(0),nil))))
193. list(cons(x,cons(s(s(x)),y)))	170. list(cons(s(0),cons(s(0),cons(nil,nil))))
195. list(cons(s(x),cons(s(s(x)),y)))	184. list(cons(s(s(0)),cons(x,x)))
183. list(cons(s(s(x))),cons(s(s(x)),nil)))	185. list(cons(x,cons(s(s(0)),x)))
250. list(cons(x,cons(x, cons(y,0))))	196. list(cons(s(s(x)),cons(s(x),x)))
	197. list(cons(s(s(x)),cons(x,x)))
	198. list(cons(x,cons(s(s(x)),x)))
	199. list(cons(s(s(x)),cons(s(s(x)),x)))
	200. list(cons(s(x),cons(s(s(x)),x)))
	201. list(cons(s(s(s(x))),cons(s(s(s(x))),x)))
	202. list(cons(s(s(x)),cons(s(x),x)))
	203. list(cons(s(s(x)),cons(x,s(x))))
	204. list(cons(x,cons(s(s(x)),s(x))))
	205. list(cons(s(s(x)),cons(s(s(x)),s(x))))
	206. list(cons(s(x),cons(s(s(x)),s(x))))
	207. list(cons(s(s(x)),cons(s(x),s(s(x)))))
	208. list(cons(s(s(x)),cons(x,s(s(x)))))
	209. list(cons(x,cons(s(s(x)),s(s(x)))))
	210. list(cons(s(s(x)),cons(s(s(x)),s(s(x)))))
	211. list(cons(s(x),cons(s(s(x)),s(s(x)))))
	212. list(cons(s(s(s(x))),cons(s(s(s(x))),s(s(s(x))))))
	213. list(cons(s(s(s(x))),cons(s(s(x)),s(s(s(x))))))
	214. list(cons(s(s(s(x))),cons(s(s(s(x))),s(s(x)))))
	215. list(cons(s(s(x)),cons(s(s(s(x))),s(s(s(x))))))
	216. list(cons(s(s(s(x))),cons(s(s(s(x))),x)))
	217. list(cons(x,cons(s(s(s(x))),s(s(s(x))))))
	218. list(cons(s(s(s(x))),cons(x,s(s(s(x))))))
	219. list(cons(s(s(s(x))),cons(s(s(s(x))),s(x))))

Figure 42: The table contains the goals which were used to generate the **extended set of** examples of coinductive proof trees (cf. Definition 8) for the pattern-recognition Problem 3; program `ListNat`; and **success proof-family** of `list(x,cons(y,z))`. The the data set is also available at [27].

Goals/positive examples	Goals/negative examples
	220. list(cons(s(s(s(x))),cons(s(x),s(s(s(x))))))
	221. list(cons(s(x),cons(s(s(s(x))),s(s(s(x))))))
	222. list(cons(s(s(s(x))),cons(s(s(s(x))),s(x))))
	223. list(cons(s(s(s(x))),cons(s(x),x)))
	224. list(cons(s(s(s(x))),cons(s(x),s(x))))
	225. list(cons(s(x),cons(s(s(s(x))),x)))
	226. list(cons(s(s(s(x))),cons(s(s(x)),x)))
	227. list(cons(s(s(x)),cons(s(s(s(x))),x)))
	239. list(cons(0,cons(0, cons(0,0))))
	240. list(cons(0,cons(0, cons(0, cons(0,0)))))
	241. list(cons(0,cons(0, cons(0, cons(0,cons(0,0))))))
	242. list(cons(0,cons(x, cons(x,x))))
	243. list(cons(0,cons(x, cons(y,x))))
	244. list(cons(0,cons(x, cons(y,y))))
	245. list(cons(x,cons(0, cons(y,y))))
	246. list(cons(y,cons(0, cons(x,y))))
	247. list(cons(x,cons(0, cons(x,x))))
	248. list(cons(0,cons(x, cons(x,0))))
	249. list(cons(0,cons(x, cons(y,0))))
	251. list(cons(x,cons(y, cons(0,0))))
	252. list(cons(x,cons(0, cons(0,0))))
	255. list(cons(0,cons(x, cons(x,cons(x,x)))))

Figure 43: The table contains the goals which were used to generate the **extended set of** examples of coinductive proof trees (cf. Definition 8) for the pattern-recognition Problem 3; program `ListNat`; and **success proof-family** of `list(x,cons(y,z))`. The the data set is also available at [27].

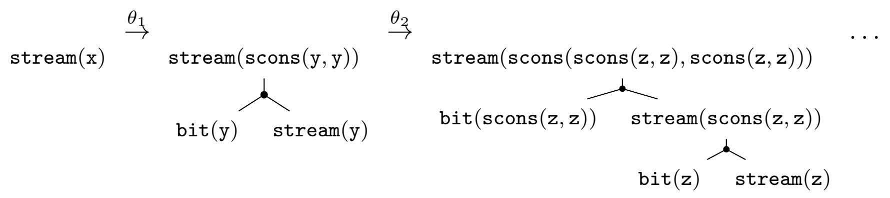


Figure 44: Ill-typed derivation for the goal $G =$ `stream(x)` and the program `Stream`.

two distinct predicates P and Q in T such that $P(t)$ and $Q(t)$ appear as nodes in T.

All the training and testing examples were trees from the proof family of `stream(scons(x,cons(y,z)))`, marked as well-typed (cf. Figure 2) or ill-typed. Figure 45 shows the data set that was used for the classification purposes. Again, the results in Neural networks and SVMs are both very robust ranging between 86% - 90 %, and are summarised in Table 13.

The proof-search with coinductive trees (cf. Definition 11) can become very expensive in presence of (co)recursion. This is why, using statistical *proof-hints* akin the ones given by Problems 3 and 4, would reduce the number of the redundant steps by up to 85 - 99 %. In derivations similar to Figures 1 and 2, knowing the proof family of the goal will allow to prune the search. In derivations similar to Figure 38 and 44, it can help to avoid ill-formed steps.

Problem 5 (Discovery of ill-typed proofs) *The problem is similar to Problem 4, however, the restriction that all examples of proofs belong to the same proof family is lifted, and so examples can represent a wider variety of proofs.*

We used 255 examples of trees for ListNat, see Figures 46 – 48; and 85 examples for Stream, see Figures 49 – 50. The success of classification for `ListNat` and `Stream` was 82.4%; with extremely robust results through the cycle. The task was more difficult for statistical pattern recognition: due to presence of the two data types in each program – such as `nat` and `list`, the well-typed proof trees had less regular shape, and sometimes the trees of similar shape contributed to the opposite classes, see also the Manual [27].

The important conclusion of this section is that the proposed method indeed captures the essential proof-patterns, allowing classification of proofs according to a wide variety of meta-properties. The percentage of inaccuracies correlates with the number of cases in which proofs with some regular structure do not fall into the same class. This explains why experiments with proof families show higher accuracy than Problems 1 and 5.

Goals/Positive Examples	Goals/Negative Examples
1.stream(scons(0,scons(1,scons(0,x))))	26.stream(scons(0,scons(0,1))
2.stream(scons(1,scons(0,scons(1,x))))	27.stream(scons(x,scons(y,scons(z,x))))
3.stream(scons(0,scons(0,scons(0,1))))	28.stream(scons(x,scons(y,scons(z,x))))
4.stream(scons(1,scons(1,scons(1,x))))	29.stream(scons(x,scons(x,scons(y,y))))
5.stream(scons(x,scons(1,scons(y,z))))	30.stream(scons(x,scons(y,scons(0,x))))
6.stream(scons(x,scons(0,scons(y,z))))	31.stream(scons(x,scons(y,scons(1,x))))
7.stream(scons(x,scons(1,scons(x,z))))	32.stream(scons(x,scons(y,scons(0,0))))
8.stream(scons(x,scons(0,scons(x,z))))	33.stream(scons(x,scons(y,scons(1,1))))
9.stream(scons(x,scons(1,scons(1,z))))	34.stream(scons(0,scons(y,scons(0,0))))
10.stream(scons(x,scons(0,scons(0,z))))	35.stream(scons(0,scons(0,scons(0,0))))
11.stream(scons(x,scons(y,z)))	36.stream(scons(1,scons(1,scons(1,1))))
12.stream(scons(0,scons(y,z)))	37.stream(scons(x,scons(1,scons(1,1))))
13.stream(scons(x,scons(0,z)))	38.stream(scons(x,scons(x,scons(y,y))))
14.stream(scons(x,scons(0,y)))	39.stream(scons(x,scons(y,scons(z,x))))
15.stream(scons(1,(scons(y,z)))	40.stream(scons(0,scons(y,scons(0,0))))
16.stream(scons(x,(scons(1,z)))	41.stream(scons(0,scons(x,scons(0,x))))
17.stream(scons(x,(scons(y,1)))	42.stream(scons(1,scons(y,scons(1,y))))
18.stream(scons(x,(scons(1,z)))	43.stream(scons(1,scons(0,scons(1,0))))
19.stream(scons(x,(scons(x,z)))	44.stream(scons(1,scons(x,scons(y,x))))
20.stream(scons(x,(scons(x,1)))	45.stream(scons(0,scons(x,scons(z,z))))
21.stream(scons(1,(scons(1,y)))	46.stream(scons(x,scons(1,scons(1,1))))
22.stream(scons(x,scons(y,scons(x,z))))	47.stream(scons(x,scons(z,scons(x,z))))
23.stream(scons(x,scons(0,scons(0,z))))	48.stream(scons(x,scons(y,scons(z,x))))
24.stream(scons(x,scons(x,scons(0,y))))	49.stream(scons(0,scons(x,scons(z,z))))
25.stream(scons(1,scons(1,scons(0,z))))	50.stream(scons(0,scons(x,scons(0,x))))
	51.stream(scons(1,scons(x,scons(1,x))))
	52.stream(scons(1,scons(0,scons(1,0))))
	53.stream(scons(1,scons(x,scons(z,z))))
	54.stream(scons(1,scons(x,scons(y,y))))
	55.stream(scons(0,scons(x,scons(z,x))))

Figure 45: The table contains the goals which were used to generate the examples of coinductive proof trees (cf. Definition 8) for the pattern-recognition Problem 4; program `Stream`; and **ill-typed proofs in a proof-family** of `stream(x,scons(y,z))`. The the data set is also available at [27].

Positive examples	Negative examples
1. nat(s(s(s(s(0)))))	2. nat(s(s(s(s(nil)))))
4. nat(s(s(s(s(x)))))	3. nat(s(s(cons(0,nil))))
9. nat(x)	5. nat(s(nil s(s(s))))
10. list(cons(x,cons(y,z)))	6. nat(cons(x, cons(y,z)))
11. list(cons(0,cons(y,nil)))	7. nat(cons(x,y))
12. list(cons(0,cons(0,nil)))	8. nat(nil)
13. list(cons(x,cons(y,nil)))	28. list(cons(x,cons(nil,y)))
14. list(cons(x,cons(0,nil)))	29. list(cons(x,cons(nil,nil)))
15. list(cons(x,cons(0,y)))	30. list(cons(nil,cons(x,nil)))
16. list(cons(0,cons(y,x)))	31. list(cons(nil,cons(nil,nil)))
17. list(cons(0,cons(0,x)))	32. list(cons(x,cons(y,0)))
18. list(cons(x,cons(s0,nil)))	33. list(cons(0,cons(y,0)))
19. list(cons(0,cons(s0,nil)))	34. list(cons(0,cons(0,0)))
20. list(cons(s0,cons(x,nil)))	35. list(cons(x,cons(0,0)))
21. list(cons(s0,cons(0,nil)))	36. list(cons(x,cons(nil,0)))
22. list(cons(s0,cons(s0,nil)))	37. list(cons(0,cons(nil,x)))
23. list(cons(x,cons(s0,xnil))	38. list(cons(0,cons(nil,0)))
24. list(cons(0,cons(s0,x)))	39. list(cons(0,cons(nil,nil)))
25. list(cons(s0,cons(y,x)))	40. list(cons(nil,cons(0,nil)))
26. list(cons(s0,cons(0,y)))	41. list(cons(nil,cons(x,y)))
27. list(cons(s0,cons(s0,nil)))	42. list(cons(nil,cons(0,x)))
48. list(cons(y,x))	43. list(cons(nil,cons(x,0)))
49. list(cons(0,x))	44. list(cons(nil,cons(0,0)))
50 list(cons(0,nil))	45. list(cons(nil,cons(nil,x)))
51. list(cons(x,nil))	46. list(cons(nil,cons(x,nil)))
55. list(x)	47. list(cons(nil,cons(nil,0)))
56. list(nil)	52. list(cons(nil,x))
61. nat(s(s(s(s(s(0))))))	53. list(cons(x,0))
62. nat(s(s(s(s(s(x))))))	54. list(cons(nil,0))
63. nat(s(s(s(0))))	57. list(0)
64. nat(s(s(0)))	58. list(cons(x,cons(y,x)))
65. nat(s(0))	59. list(cons(x,cons(x,x)))
66. nat(0)	60. list(cons(x,cons(y,y)))
72. nat(s(s(s(s(s(x))))))	76. nat(s(x(s(s(s(x))))))
73. nat(s(s(s(0))))	77. nat(s(x(s(s(s(0))))))
74. nat(s(s(x)))	78. nat(s(s(x(s(s(x))))))
75. nat(s(x))	79. nat(s(s(x(s(s(0))))))
90. list(cons(x,cons(x,cons(x,y))))	61. nat(s(s(s(s(s(0))))))
91. list(cons(x,cons(y,cons(y,z))))	62. nat(s(s(s(s(s(x))))))
92. list(cons(x,cons(x,cons(x, cons(x,y)))))	63. nat(s(s(s(0))))
93. list(cons(x,cons(x,cons(y,z))))	64. nat(s(s(0)))
134. list(cons(s(0),cons(x,nil)))	65. nat(s(0))
135. list(cons(s(0),cons(0,nil))))	66. nat(0)
136. list(cons(s(0),cons(s(x),y))))	67. nat(x(s(s(s(s(0))))))
118. list(cons(s(x)),cons(x,y))	68. nat(s(s(s(s(s(nil))))))
137. list(cons(0,cons(s(0),y))))	69. nat(s(s(s(nil))))
138. list(cons(0,cons(s(x),nil))))	70. nat(s(s(nil)))
100. list(cons(y,cons(x,cons(x,z))))	71. nat(s(nil))
101. list(cons(0,cons(x,cons(x,y))))	80. nat(s(0(s(s(s(0))))))
102. list(cons(x,cons(0,cons(x,y))))	81. nat(s(s(0(s(s(0))))))
103. list(cons(0,cons(0,cons(x,y))))	82. nat(0(s(s(s(s(0))))))
104. list(cons(0,cons(0,cons(0,y))))	83. nat(s(s(s(0(s(0))))))

Figure 46: The table contains the goals which were used to generate examples of coinductive proof trees (cf. Definition 8) for the pattern-recognition Problem 5; for program `List`. The the data set is also available at [27].

Goals/positive examples	Goals/negative examples
105. list(cons(0,cons(0,cons(0,nil))))	204. list(cons(x,cons(s(s(x)),s(x))))
106. list(cons(s(0),cons(x,z)))	203. list(cons(s(s(x)),cons(x,s(x))))
107. list(cons(x,cons(s(0),y)))	205. list(cons(s(s(x)),cons(s(s(x)),s(x))))
108. list(cons(s(0),cons(x,nil)))	206. list(cons(s(x),cons(s(s(x)),s(x))))
109. list(cons(x,cons(s(0),nil)))	207. list(cons(s(s(x)),cons(s(x),s(s(x)))))
110. list(cons(x,cons(x,cons(x,nil))))	208. list(cons(s(s(x)),cons(x,s(s(x)))))
111. list(cons(x,cons(x,cons(x,cons(x,nil)))))	209. list(cons(x,cons(s(s(x)),s(s(x)))))
112. list(cons(x,cons(y,cons(x,nil))))	210. list(cons(s(s(x)),cons(s(s(x)),s(s(x)))))
113. list(cons(x,cons(0,cons(x,nil))))	84. nat(s(s(s(s(0(0))))))
114. list(cons(0,cons(0,cons(x,nil))))	85. nat(s(s(s(s))))
115. list(cons(0,cons(0,cons(0,nil))))	86. nat(s(s(s(0(0(0))))))
116. list(cons(s(s(0)),nil))	211. list(cons(s(x),cons(s(s(x)),s(s(x)))))
117. list(cons(s(s(x)),nil))	212. list(cons(s(s(s(x))),cons(s(s(s(x))),s(s(x)))))
119. list(cons(s(x)),cons(x,nil))	87. nat(s(s(0(0(0(0))))))
120. list(cons(x),cons(s(x),y))	88. nat(s(s(0(s(s(x))))))
121. list(cons(s(x)),cons(s(x),y))	89. nat(s(s(0(s(x)))))
122. list(cons(x),cons(s(x),nil))	97. list(cons(x,x))
123. list(cons(s(x),cons(s(x),nil)))	213. list(cons(s(s(s(x))),cons(s(s(x)),s(s(s(x))))))
124. list(cons(s(x),cons(s(x),cons(s(x),y))))	214. list(cons(s(s(s(x))),cons(s(s(s(x))),s(s(x)))))
125. list(cons(s(x),cons(s(x),cons(s(x),nil))))	97. list(cons(x,x))
126. list(cons(s(x),cons(x,cons(x,y))))	215. list(cons(s(s(x)),cons(s(s(s(x))),s(s(x)))))
127. list(cons(s(x),cons(x,cons(x,nil))))	216. list(cons(s(s(s(x))),cons(s(s(s(x))),x)))
128. list(cons(x,cons(s(x),cons(x,y))))	94. list(cons(x,cons(x,cons(x,x))))
129. list(cons(x,cons(s(x),cons(x,nil))))	95. list(cons(x,cons(x,cons(x, cons(x,x)))))
130. list(cons(x,cons(x,cons(s(x),y))))	96. list(cons(x,cons(x,cons(x,cons(x, cons(x,x))))))
131. list(cons(x,cons(x,cons(s(x),nil))))	98. list(cons(x,cons(x,cons(y,cons(x,y)))))
132. list(cons(s(0),cons(x,y)))	99. list(cons(x,cons(y,cons(x,cons(x,y)))))
133. list(cons(s(0),cons(0,y))))	140. list(cons(s(0),cons(s(y),y))))
139. list(cons(0,cons(s(0),nil))))	217. list(cons(x,cons(s(s(s(x))),s(s(x)))))
141. list(cons(s(0),cons(s(0),y))))	218. list(cons(s(s(s(x))),cons(x,s(s(x)))))
142. list(cons(s(0),cons(x,nil))))	219. list(cons(s(s(s(x))),cons(s(s(s(x))),s(x))))
143. list(cons(s(0),cons(s(0),nil))))	
144. list(cons(s(0),cons(s(0),cons(s(0),y)))))	232. list(s(s(s(x))),x)
152. list(cons(y,cons(x,cons(x,nil))))	233. list(s(s(x)),x)
145. list(cons(s(0),cons(s(0),cons(s(0),nil))))	234. list(s(s(s(s(x)))),s(x))
146. list(cons(s(0),cons(0,cons(0,y))))	235. list(s(s(s(s(x)))),s(s(x)))
147. list(cons(s(0),cons(0,cons(0,nil))))	236. list(s(s(s(s(x)))),s(s(s(x))))
148. list(cons(0,cons(s(0),cons(0,y))))	237. list(s(s(s(s(x)))),s(s(s(s(x)))))
149. list(cons(0,cons(s(0),cons(0,nil))))	238. list(x,s(s(s(s(x)))))
150. list(cons(0,cons(0,cons(s(0),y))))	253. list(s(s(x)),s(s(s(s(x)))))
151. list(cons(0,cons(0,cons(s(0),nil))))	254. list(s(s(s(x))),s(s(s(s(x)))))
171. list(cons(s(s(0)),cons(s(0),nil)))	153. list(cons(nil,cons(y,cons(x,nil))))
172. list(cons(s(s(0)),cons(0,nil)))	154. list(cons(nil,cons(nil,cons(x,nil))))
173. list(cons(0,cons(s(s(0)),nil)))	155. list(cons(nil,cons(nil,cons(nil,nil))))
174. list(cons(s(s(0)),cons(s(s(0)),nil)))	156. list(cons(nil,cons(nil,cons(nil,cons(nil,nil)))))
175. list(cons(s(0),cons(s(s(0)),nil)))	157. list(cons(nil,cons(nil,cons(nil,cons(nil,cons(nil,nil))))))
176. list(cons(s(s(0)),cons(s(0),x)))	158. list(cons(nil,cons(x,cons(x,y))))
177. list(cons(s(s(0)),cons(0,x)))	159. list(cons(nil,cons(x,cons(y,x))))
178. list(cons(0,cons(s(s(0)),x)))	160. list(cons(x,cons(nil,cons(y,x))))
179. list(cons(s(s(0)),cons(s(s(0)),x)))	161. list(cons(x,cons(nil,cons(x,y))))
180. list(cons(s(0),cons(s(s(0)),x)))	162. list(cons(x,cons(nil,cons(x,cons(x,x)))))
186. list(cons(s(s(x)),cons(s(x),nil)))	163. list(cons(x,cons(nil,cons(x,x))))

Figure 47: The table contains the goals which were used to generate the examples of coinductive proof trees (cf. Definition 8) for the pattern-recognition Problem 5; program `ListNat`. The the data set is also available at [27].

Goals/positive examples	Goals/negative examples
187. list(cons(s(s(x)),cons(x,nil)))	164. list(cons(s(0),cons(s(0),cons(s(0),s(0)))))
188. list(cons(x,cons(s(s(x)),nil)))	165. list(cons(s(0),cons(0,cons(0,s(0)))))
189. list(cons(s(s(x)),cons(s(s(x)),nil)))	166. list(cons(nil,cons(0,cons(0,s(0)))))
190. list(cons(s(x),cons(s(s(x)),nil)))	167. list(cons(nil,cons(s(0),cons(0,0))))
191. list(cons(s(s(x)),cons(s(x),y)))	168. list(cons(nil,cons(s(0),cons(s(0),nil))))
192. list(cons(s(s(x)),cons(x,y)))	169. list(cons(s(0),cons(nil,cons(s(0),nil))))
193. list(cons(x,cons(s(s(x)),y)))	170. list(cons(s(0),cons(s(0),cons(nil,nil))))
195. list(cons(s(x),cons(s(s(x)),y)))	184. list(cons(s(s(0)),cons(x,x)))
183. list(cons(s(s(x))),cons(s(s(x)),nil)))	185. list(cons(x,cons(s(s(0)),x)))
250. list(cons(x,cons(x, cons(y,0))))	196. list(cons(s(s(x)),cons(s(x),x)))
181. list(s(s(0)),nil)	201. list(cons(s(s(s(x))),cons(s(s(s(x))),x)))
182. list(s(s(x)),nil)	202. list(cons(s(s(x)),cons(s(x),x)))
228. list(s(s(s(0))),nil)	197. list(cons(s(s(x)),cons(x,x)))
229. list(s(s(s(x))),nil)	198. list(cons(x,cons(s(s(x)),x)))
230. list(s(s(s(0))),x)	199. list(cons(s(s(x)),cons(s(s(x)),x)))
231. list(s(s(s(x))),y)	200. list(cons(s(x),cons(s(s(x)),x)))
	220. list(cons(s(s(s(x))),cons(s(x),s(s(s(x))))))
	221. list(cons(s(x),cons(s(s(s(x))),s(s(s(x))))))
	222. list(cons(s(s(s(x))),cons(s(s(s(x))),s(x))))
	223. list(cons(s(s(s(x))),cons(s(x),x)))
	224. list(cons(s(s(s(x))),cons(s(x),s(x))))
	225. list(cons(s(x),cons(s(s(s(x))),x)))
	226. list(cons(s(s(s(x))),cons(s(s(x)),x)))
	227. list(cons(s(s(x)),cons(s(s(s(x))),x)))
	239. list(cons(0,cons(0, cons(0,0))))
	240. list(cons(0,cons(0, cons(0, cons(0,0)))))
	241. list(cons(0,cons(0, cons(0, cons(0,cons(0,0))))))
	242. list(cons(0,cons(x, cons(x,x))))
	243. list(cons(0,cons(x, cons(y,x))))
	244. list(cons(0,cons(x, cons(y,y))))
	245. list(cons(x,cons(0, cons(y,y))))
	246. list(cons(y,cons(0, cons(x,y))))
	247. list(cons(x,cons(0, cons(x,x))))
	248. list(cons(0,cons(x, cons(x,0))))
	249. list(cons(0,cons(x, cons(y,0))))
	251. list(cons(x,cons(y, cons(0,0))))
	252. list(cons(x,cons(0, cons(0,0))))
	255. list(cons(0,cons(x, cons(x,cons(x,x)))))

Figure 48: The table contains the goals which were used to generate the examples of coinductive proof trees (cf. Definition 8) for the pattern-recognition Problem 5; program `ListNat`. The the data set is also available at [27].

Goals/Positive Examples	Goals/Negative Examples
1.stream(scons(0,scons(1,x)))	5.stream(scons(x,scons(x,x)))
2.stream(scons(1,scons(1,x)))	6.stream(scons(x,scons(0,x)))
3.stream(scons(0,scons(0,x)))	7.stream(scons(x,scons(y,x)))
4.stream(scons(1,scons(0,x)))	8.stream(scons(y,scons(x,x)))
12.stream(scons(0,scons(x,y)))	10.stream(scons(0,scons(x,x)))
13.stream(scons(x,scons(0,y)))	11.stream(scons(x,scons(x,0)))
17.stream(scons(1,scons(1,x)))	14.stream(scons(x,scons(y,0)))
18.stream(scons(1,scons(x,y)))	15.stream(scons(x,scons(1,x)))
19.stream(scons(x,scons(1,y)))	16.stream(scons(1,scons(x,x)))
31.stream(scons(0,x))	20.stream(scons(x,scons(y,1)))
32.stream(scons(1,x))	21.stream(0)
34.stream(scons(x,y))	22.stream(scons(0,scons(0,0)))
47.stream(x)	23.stream(scons(1,scons(1,1)))
48.stream(scons(0,(scons(0,scons(0,x)))	24.stream(scons(0,scons(1,0)))
49.stream(scons(1,(scons(1,scons(1,x)))	25.stream(scons(0,scons(0,1)))
50.stream(scons(1,(scons(1,scons(0,x)))	26.stream(scons(1,scons(0,0)))
51.stream(scons(1,(scons(0,scons(1,x)))	27.stream(scons(1,scons(1,0)))
52.stream(scons(0,(scons(1,scons(1,x)))	28..stream(scons(1,scons(0,1)))
53.stream(scons(1,(scons(0,scons(0,x)))	29.stream(scons(0,scons(1,1)))
54.stream(scons(0,(scons(1,scons(0,x)))	30.stream(1)
55.stream(scons(0,(scons(0,scons(1,x)))	33.stream(scons(x,x))
86.stream(y)	35.stream(scons(x,1))
87.stream(scons(x,y))	36.stream(scons(x,0))
88.stream(scons(1,y))	37.stream(scons(1,1))
89.stream(scons(0,y))	38.stream(scons(0,0))
90.stream(scons(x,scons(y,z)))	39.stream(scons(1,0))
91.stream(scons(0,scons(y,z)))	40.stream(scons(0,1))
92.stream(scons(x,scons(0,z)))	41.bit(scons(0,x)
93.stream(scons(1,scons(y,z)))	42.bit(scons(1,x))
94.stream(scons(x,scons(1,z)))	43.bit(scons(x,y))
95.stream(scons(x,scons(x,scons(y,z)))	44.bit(scons(x,x))
96.stream(scons(1,scons(1,scons(0,x)))	45.bit(scons(y,x))
9. stream(scons(x,scons(x,y)))	46.bit(scons(y,1))

Figure 49: The table contains some of the goals which were used to generate examples of coinductive proof trees (cf. Definition 8) for the pattern-recognition Problem 5; for program `Stream`; cf. [27].

Goals/Positive Examples	Goals/Negative Examples
	56.stream(scons(x,scons(0,scons(0,1))))
	57.stream(scons(x,scons(1,scons(1,0))))
	58.stream(scons(x,scons(1,scons(0,1))))
	59.stream(scons(x,scons(0,scons(1,0))))
	60.stream(scons(x,scons(1,scons(1,1))))
	61.stream(scons(x,scons(0,scons(0,0))))
	62.stream(scons(x,scons(1,scons(0,0))))
	63.stream(scons(x,scons(0,scons(1,1))))
	64.stream(scons(0,scons(x,scons(0,1))))
	65.stream(scons(1,scons(x,scons(1,0))))
	66.stream(scons(1,scons(x,scons(0,1))))
	67.stream(scons(0,scons(x,scons(1,0))))
	68.stream(scons(1,scons(x,scons(1,1))))
	69.stream(scons(0,scons(x,scons(0,0))))
	70.stream(scons(1,scons(x,scons(0,0))))
	71.stream(scons(0,scons(x,scons(1,1))))
	72.stream(scons(1,scons(1,scons(1,1))))
	73.stream(scons(0,scons(0,scons(0,0)))
	74.stream(scons(0,scons(1,scons(1,1)))
	75.stream(scons(1,scons(0,scons(1,1)))
	76.stream(scons(1,scons(1,scons(0,1)))
	77.stream(scons(1,scons(1,scons(1,0)))
	78.stream(scons(1,scons(0,scons(0,0)))
	79.stream(scons(0,scons(1,scons(0,0)))
	80.stream(scons(0,scons(0,scons(1,0)))
	81.stream(scons(0,scons(0,scons(0,1)))
	82.stream(scons(0,scons(0,scons(1,1)))
	83.stream(scons(1,scons(1,scons(0,0)))
	84.stream(scons(1,scons(0,scons(0,1)))
	85.stream(scons(0,scons(1,scons(1,10))))

Figure 50: The table contains some of the goals which were used to generate examples of coinductive proof trees (cf. Definition 8) for the pattern-recognition Problem 5; for program `Stream`; cf. [27].

7 Implementation scenarios

In this Section, we offer several implementation scenarios for the results of the previous section. We invite the reader to imagine an automated theorem prover (ATP) used for proof development of industrial scale, which involves thousands of lemmas, many of them having similar structure. We assume that the prover routinely solves similar cases by using the same kind of proof steps or tactics; and in some cases, when a customary path in proof fails, the user discards the unsuccessful attempt and tries another route. This is the ideal setting for which we design the statistical proof-pattern recognition tool (SPPT). To start with, SPPT registers all – successful and unsuccessful – attempts of proofs, and stores them in a form of feature vectors. They form the basis for statistical proof-pattern recognition. Further, it generates a set of neural networks which use the feature vectors for training and classification. With each new example, the tool re-trains itself, and thus becomes more accurate in its future classification. If several people work in a group, this data can be shared. For every newly started attempt of a proof, we would like SPPT to do the following:
– warn the user if (after some number of proof steps) the proof structurally resembles the previous failed attempts of other proofs; that is, SPPT may serve for early diagnosis of proof failure.
– inform the user if the proof belongs to an earlier detected proof family. This would allow to re-use the previously discovered proof-patterns; or to optimise proof-search.

So, in practice, SPPT would play the role of a *statistical hint generator*. There will be cases in automated theorem proving that will be so conceptually demanding that they will not yield mere statistical proof-pattern analysis. For such cases, we assume that SPPT is just switched-off or ignored.

From this general picture, we now descend to implementational level, and discuss several scenarios how such application can be designed in practice. Note that the tool can be used to classify proofs as described in either of the Problems 1-5 considered in Section 6. All experiments in this section use neural networks described in Section 6.

It is likely that the automated proof development will involve working with a variety of data structures, in our case represented by separate logic programs. In this case, two scenarios are likely.

Scenario 1 *The ATP may be used to work with one data structure (logic program) at a time, but such programs regularly change.*

The obvious objection to such approach is that creating a new neural network for every new fragment of a big program development may be cumbersome; it will not capture possible common patterns across different programs and fragments, but also, it will handle badly the cases where some apparently disconnected programs are bound by a newly added definition. The next two implementation scenarios address these problems.

Scenario 2 *or else, the ATP is used to work with several data structures. , so examples are mixed from the start.*

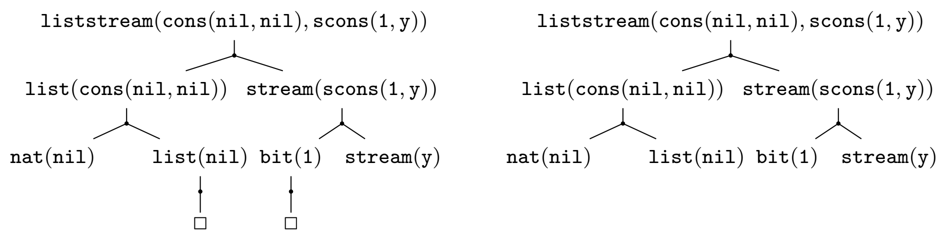


Figure 51: Well-formed and ill-typed (left) and ill-formed (right) coinductive trees (Problems 1 and 5) for program `Listream`.

Example 27 *Using our running examples, the ATP in question first worked with the proofs over the program* `ListNat` *and then switched to* `Stream`*, or vice versa.*

Depending on this, the proof feature vectors can be stored and used for training in several **Implementation Scenarios (IS)**:

IS 3 *The SPPT can create a new neural network for every new logic program.*

Example 28 *Using our running examples, a separate sets of feature vectors for* `nat`*,* `ListNat`*, and* `Stream`*, can be used to train three separate neural networks, see also Figure 3 for experiments supporting this approach.*

This implementation strategy would be particularly convenient if the ATP uses varied data structures, but they do not mix in the same proofs. But it may not always be practical for other cases. Consider the example below.

Example 29 *Suppose the ATP was used to work with proofs constructed for two programs –* `ListNat`*, and* `Stream`*; and maintains two corresponding neural networks. However, a new clause is added by the user:*
`listream(x,y)` $\leftarrow$ `list(x), stream(y)`.

The new, extended program including all our running examples and the clause above will be called `Listream`*; see Figures 51 and 52. Neither of the two neural nets will accept the changed feature vectors for new proofs, as additional new predicate will infer the change in size of the vectors.*

If the proofs for varied logic programs normally mix, we offer two other implementation choices below.

IS 4 *SPPT contains only one neural network, and re-trains it irrespective of the changes in program clauses, new predicates or proof structures.*

Clearly, if e.g. `ListNat` and `Stream` have different proof-patterns, some misclassification when switching between programs is inevitable; and so is the loss of accuracy with respect to the old proof-patterns, as the neural network re-trains itself in favour of the new examples. In this case, legitimate questions are: How

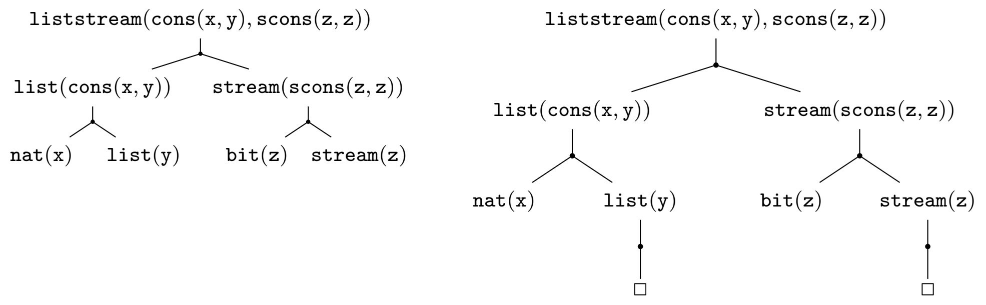


Figure 52: Well-formed and ill-typed (left) and ill-formed (right) coinductive trees (Problems 1 and 5) for program `Listream`.

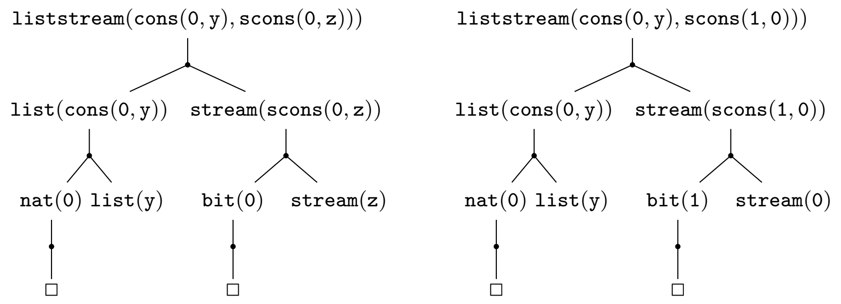


Figure 53: Well-formed well-typed (left) and ill-typed (right) coinductive trees, used for Problem 5 experiments for program `Listream`.

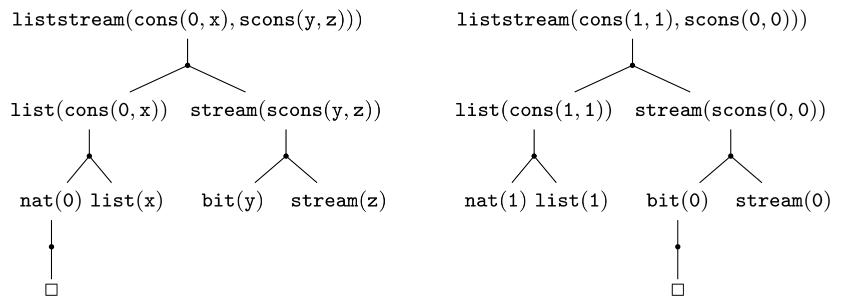


Figure 54: Well-formed well-typed (left) and ill-typed (right) coinductive trees used for Problem 5 experiments for program `Listream`.

X-Y — Problem	Initial accuracy for X	Test 1 on Y	Test 2	Test 3	X-Y Mixed data
List-Stream — P1	76.4%	44.2%	51.9%	63.9%	67.1%
Stream-List — P1	84.3%	36.7%	44%	67%	67.1%
List-Stream — P5	82.4%	65.6%	80.0%	100%	80.1%
Stream-List — P5	79.0%	43.5%	63.5%	85.9%	80.1%

Figure 55: **Gradual adaptation to newly discovered proof patterns.** Letters X and Y above stand for our two running examples of logic programs we used interchangeably for various experiments; P1 and P5 stand for Problems 1 and 5 from Section 6. First a type X is taken, and it's accuracy is tested on several neural networks of different sizes; the average is shown in the first column. Then these trained networks were used to classify examples of the proofs of new type Y. The accuracy drops at the start, see the "Test 1" column. Further columns show how the neural network adapts when more and more of new examples are used for its training. Note that transitions from `ListNat` proofs to `Stream` are generally more smooth than the other way around; Problem 5 is generally easier than Problem 1.

well will neural network trained on one set of proof patterns will recognise new proof-patterns? and How soon will it adapt, as it gets more of new examples to learn from? To answer these questions, we designed a series of experiments, as shown in Figure 55. We used the data base of examples we give in [27] and Figures 57 and 58 below. The experiments convince us that the neural networks trained one one type of proof patterns do not immediately show good results when tested on proofs of new type. However, they re-gain accuracy quite well, when given a chance to use this new data for re-training.

We concentrate now only on Problem 5 re-training implementation, as the proofs arising there are more regular. Figure 56 shows how subtle the retraining can be. The setting for the experiments was as follows. First an arbitrary neural network was trained on the `Listnat` data set of 255 examples. Then the data set of 96 `Stream` examples was divided into three equal parts. First part was fed as a data to the initial neural network trained on `ListNat` as a test, the result was 65.6%. Then a proportion of this new small set of examples was used to retrain this old neural network, and smaller proportion – to test it again – the accuracy was 80% for these new reserved examples; see the "Test2" row of Figure 56. Then the same re-training and testing experiment was repeated with the second part of the fresh data set `Stream`. Thus, this time we retrained second neural network using the second data set on `Stream`, and the accuracy increased to 100%, and it did not fall after training on the new, third part, of the `Stream` data set. Finally, the last column shows the results of training on a data set in which examples for `ListNat` and `Stream` are mixed. The second half of Table 56 shows a similar example but with training done on `Stream` eaxmples, and re-training on `ListNat`.

From these experiments, it is clear that networks trained with proofs of more complex structure (like `ListNat`) adapt better to proofs of relatively simpler structure (like `Stream`) than the other way around, see Figure 56. Also, the story of gradual adaptation is not always as smooth as appears in Figure 55, it will vary greately depending on whether new examples of proofs bear similar features to the old examples, irrespective of the data type. In this respect, if we

X-Y	Initial accuracy for X	Test 1 on Y	Test 2	Test 3	Test 4	Mixed data
List-Stream-1	82.4%	65.6%	80.0%	100%	100%	80.1 %
List-Stream-2	82.4%	65.6%	100%	68.6%	80%	80.1%
List-Stream-3	82.4%	65.6%	60%	60%	100%	80.1%
Stream-List-1	79.0%	67.1%	43.5%	63.5%	85.9%	80.1%
Stream-List-2	79.0%	67.1%	53.8%	53.8%	61.5%	80.1%
Stream-List-3	79.0%	67.1%	30.8%	76.9%	61.5%	80.1%

Figure 56: **Gradual adaptation to newly discovered proof patterns, for Problem 5.** Letters X and Y above stand for our two running examples of logic programs we used interchangeably for various experiments. First a type X is taken, and it's accuracy is tested on several neural networks of different sizes; the average is shown in the first column. Then these trained networks were used to classify examples of the proofs of new type Y. The accuracy drops at the start, see the "Test 1" column. Further columns show how the neural network adapts when more and more of new examples are used for its training. Note that transitions from `ListNat` proofs to `Stream` are generally more smooth than the other way around. The table show how different the results of training can be when we change the order in which training and testing examples arrive as inputs to the neural network.

look at examples of adaptation of a neural network trained on a bigger data set for `ListNat` to a smaller dataset for `Stream`, we will see that despite the sizes of the new data set, adaptation is very successful, and reaches 100% for some examples; being clearly more accurate than training on a mixed data set. The same Figure shows that adaptation in the other direction does not go nearly so well, even despite the fact that the new data set is twice bigger than the old data set. Only for some examples accuracy exceeds the accuracy of the mixed set results.

In many respects, this is good news for the method we test. It means that its accuracy is more sensitive to the structural properties of proofs than to the data structure the given logic program defines. It also shows that structure matters more than the size of the data set, which is again a positive feature: in real-life applications, new examples of proofs may not be coming in very large batches, and patterns may need to be recognised on a basis of just a few examples.

The only possible reservation concerning this implementation is that to some extent, the neural network "forgets" about old patterns it "knew" in favour of the new patterns it adapts to, cf. the leftmost and rightmost columns of Figure 55. This is why, we suggest the third implementation scenario, as follows.

IS 5 *The feature vectors are re-defined and extended to fit all available programs.*

In this case, statistically the proof features originating from different programs are all considered as features of one meta-proof. Example of an extended feature matrix for `Listream` is given in Figure 59.

We performed an experiment of classifying merged matrices for proofs in `ListNat` and `Stream`, see Figure 59. The merged matrices did not loose accuracy comparing to the feature vectors taken for these programs separately, which is quite exceptional from machine-learning perspective, where the growth of vector size normally causes a drop in accuracy. This can happen only if the

Goals/Positive Examples	Goals/Positive Examples
1.listream(cons(x,x),cons(y,z))	2.listream(cons(1,1),cons(y,z))
3.listream(cons(0,0),cons(y,z))	4.listream(cons(x,y),cons(0,0))
5.listream(cons(x,y),cons(z,z))	6.listream(cons(x,y),cons(1,1))
7.listream(cons(x,y),scons(z,z))	8.listream(cons(x,y),scons(0,0))
9.listream(cons(x,y),scons(1,1))	10.listream(scons(x,y),cons(z,z))
11.listream(cons(nil,nil),scons(1,0))	12.listream(cons(nil,s(0)),scons(0,0))
13.listream(cons(s(0),nil),scons(0,0))	14.listream(cons(s(0),s(0)),scons(1,0))
15.listream(cons(0,0),scons(1,1))	16.listream(cons(1,1),scons(0,0))
17.listream(cons(1,1),scons(1,1))	18.listream(cons(0,0),scons(0,0))
19.listream(x,y)	20.listream(nil,x)
21.listream(cons(0,y),scons(1,z))	22.listream(cons(s(0),x),scons(0,z))
23.listream(cons(s(0),nil),scons(0,x))	24.listream(cons(s(0),x),scons(y,z))
25.listream(cons(0),nil),scons(1,x))	26.listream(cons(0,x),scons(0,x))

Figure 57: The goals that we used to generate the set of examples of coinductive proof trees for the program `Listream`, Problem 1. The trees were generated according to the algorithm of Definition 8 and then converted into vectors, the full data base is given in [27]. Each goal above was represented by both a well-formed and an ill-formed tree.

patterns are strongly defined. In another experiment, we tested examples of proofs for `Listream`. Finally, we tested how merged matrices would adapt to new proofs for `Listream`, cf Figure 59. As was expected, the accuracy drops, but as analysed in Figure 55, it can be re-gained with training.

It is encouraging that for Problem 5, training on merged-matrix features over-performed simple mixing of data sets (as in Figure 55). When working with extended feature vectors, `Listream` over-performed the simpler merged-matrix data training. This shows that the feature-selection method we present allows extensions that capture significant and increasingly intricate proof-patterns.

8 Conclusions and Future work

We have developed a method for representing proofs as feature vectors; and employed the coinductive trees for this purpose. We tested the method on a range of classification problems (Problems 1-5), and on a range of implementation scenarios. We detected and analysed some strong and weak points of the method, but overall, the method's accuracy and adaptivity is encouraging. Our key contribution compared to e.g. a similar approach [39] is that we do not base pattern recognition on the structure of the formula alone, but on the proof-search patterns that implicitly involve resolution and unification algorithms. Also, the main attention was given to accuracy of proof-pattern representation, rather than the learning functions. Employing the various kernel functions studied in [39] to our data sets is one of our future plans.

More work will be done in the future, mostly on further automatisation of the method, and devising its extensions to other kinds of proofs in logic programs

Goals/Positive Examples	Goals/Negative Examples
1.listream(x,y)	21.listream(cons(1,1),cons(y,z))
2.listream(nil,x)	22.listream(cons(x,x),cons(y,z))
3.listream(cons(s(0),nil),scons(0,x))	23.listream(cons(x,y),cons(0,0))
4.listream(cons(0,y),scons(1,z))	24.listream(cons(0,0),cons(y,z))
5.listream(cons(s(0),x),scons(0,z))	25.listream(cons(x,y),cons(z,z))
6.listream(cons(s(0),x),scons(y,z))	26.listream(cons(x,y),cons(1,1))
7.listream(conss(S(0),nil),scons(1,x))	27.listream(cons(x,y),scons(z,z))
8.listream(cons(0,x),scons(0,x))	28.listream(cons(x,y),scons(0,0))
9.listream(cons(0,y),scons(0,z))	29.listream(scons(x,y),cons(z,z))
10.listream(cons(0,nil),scons(1,x))	30.listream(cons(x,y),scons(1,1))
11.listream(cons(x,0),scons(x,z))	31.listream(cons(nil,nil),scons(1,0))
12.listream(cons(S(0),nil),scons(1,x))	32.listream(cons(nil,s(0)),scons(0,0))
13.listream(cons(x,y),scons(x,y))	33.listream(cons(s(0),nil),scons(0,0))
14.listream(cons(0,x),scons(0,scons(0,y)))	34.listream(cons(s(0),s(0)),scons(1,0))
15.listream(cons(nil),scons(1,scons(1,y)))	35.listream(cons(0,0),scons(1,1))
16.listream(cons(s(0),y),scons(0,z))	36.listream(cons(1,1),scons(0,0))
17.listream(cons(0,y),scons(1,z))	37.listream(cons(1,1),scons(1,1))
18.listream(cons(s(0),nil),scons(0,x))	38.listream(cons(0,0),scons(0,0))
19.listream(cons(s(0),nil),scons(y,z))	39.listream(cons(nil,nil),scons(1,y))
20.listream(cons(0,x),scons(0,x))	40.listream(cons(x,x),scons(0,x))
45.listream(cons(x,nil),scons(0,y))	41.listream(cons(x,x),scons(0,x)
46.listream(cons(x,nil),scons(1,y))	42.listream(cons(x,0),x)
47.listream(cons(s(0),nil),scons(y,x))	43.listream(cons(nil,x),scons(1,y)
48.listream(cons(s(0),nil),scons(1,x))	44.listream(cons(x,0),scons(0,scons(0,y)))

Figure 58: The goals that we used to generate the set of examples of coinductive proof trees for the program `Listream`, Problem 5. The trees were generated according to the algorithm of Definition 8 and then converted into vectors, the full data base is given in [27]. Positive examples are given in the lett column, and negative – in the right column.

Matrix M_3	`listream`	`stream`	`bit`	`list`	`nat`	●	□
`cons(x, x)`	- 211411	0	0	-211	0	2	0
`scons(y, z))`	- 211411	-411	0	0	0	2	0
x	0	0	0	-1	-1	0	0
y	0	0	-1	0	0	0	0
z	0	-1	0	0	0	0	0

	Prob 1	Prob 5
Merged matrices	84.3%	82%
`Listream`	76.3%	88.6%
Merged-`Listream`	51.2%	64.9%

Figure 59: **Top:** Feature matrix for the coinductive tree for the goal `listream(cons(x,x), cons(y,z))`. **Bottom:** Accuracy of proof-pattern recognition for matrices featuring proofs with predicates from both `ListNat` and `Stream` ("Merged matrix" row); proofs for extended program `Listream` ("`Listream`" row); and experiment of training on "Merged Matrix", and testing the trained neural network on `Listream` (last row).

and in ITPs.

References

[1] S. Abe. *Support Vector Machines for Pattern Classification*. Springer-Verlag.

[2] E. Alpaydin. *Introduction to machine learning*. MIT Press.

[3] Y. Bertot and E. Komendantskaya. Inductive and coinductive components of corecursive functions in coq. *Electr. Notes Theor. Comput. Sci.*, 203(5):25–47, 2008.

[4] C. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[5] S. Colton. *Automated Theory Formation in Pure Mathematics*. Springer, 2002.

[6] T. Coquand. Infinite objects in type theory. In *Types for Proofs and Programs, Int. Workshop TYPES'93*, volume 806 of *LNCS*, pages 62–78. Springer-Verlag, 1994.

[7] A. d'Avila Garcez, K. B. Broda, and D. M. Gabbay. *Neural-Symbolic Learning Systems: Foundations and Applications*. Springer-Verlag, 2002.

[8] J. Denzinger, M. Fuchs, C. Goller, and S. Schulz. Learning from previous proof experience: A survey. Technical report, Technische Universitat Munchen, 1999.

[9] J. Denzinger and S. Schulz. Automatic acquisition of search control knowledge from multiple proof attempts. *Inf. Comput.*, 162(1-2):59–79, 2000.

[10] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. John Wiley, 2001.

[11] H. Duncan. *The use of Data-Mining for the Automatic Formation of Tactics*. PhD thesis, University of Edinburgh, 2002.

[12] C. Dwork, P. Kanellakis, and J. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1:35–50, 1984.

[13] G.Grov, E.Komendantskaya, and A.Bundy. A statistical relational learning challenge - extracting proof strategies from exemplar proofs. In *ICML'12 worshop on Statistical Relational Learning, Edinburgh, 30 July 2012*, 2012.

[14] G. Gupta and V. Costa. Optimal implementation of and-or parallel prolog. In *Conference proceedings on PARLE'92*, pages 71–92, NY, 1994. Elsevier.

[15] G. Gupta and et al. Coinductive logic programming and its applications. In *ICLP 2007*, volume 4670 of *LNCS*, pages 27–44. Springer, 2007.

[16] S. Haykin. *Neural Networks. A Comprehensive Foundation*. Macmillan College Publishing Company, 1994.

[17] R. Hecht-Nielsen. *Neurocomputing*. Addison-Wesley, 1990.

[18] P. Hitzler, S. Hölldobler, and A. K. Seda. Logic programs and connectionist networks. *Journal of Applied Logic*, 2(3):245–272, 2004.

[19] S. Hölldobler and Y. Kalinke. Towards a massively parallel computational model for logic programming. In *Proceedings of the ECAI94 Workshop on Combining Symbolic and Connectionist Processing*, pages 68–77. ECCAI, 1994.

[20] S. Hölldobler, Y. Kalinke, and H. P. Storr. Approximating the semantics of logic programs by recurrent neural networks. *Applied Intelligence*, 11:45–58, 1999.

[21] A. Ireland, G. Grov, and M. Butler. Reasoned modelling critics: Turning failed proofs into modelling guidance. In *ASM*, pages 189–202, 2010.

[22] M. Johansson, L. Dixon, and A. Bundy. Case-analysis for rippling and inductive proof. In *ITP*, volume 6172 of *LNCS*, pages 291–306. Springer, 2010.

[23] K. Kersting, L. D. Raedt, and T. Raiko. Logical hidden markov models. *J. Artif. Intell. Res. (JAIR)*, 25:425–456, 2006.

[24] E. Komendantskaya. Neurons or symbols: why does or remain exclusive? Position paper. In *Proceedings of ICNC'09, Madeira, 3-7 October*. INSTICC, 2009.

[25] E. Komendantskaya. Machine-learning coalgebraic proofs. In *Short Post-proceedings of ITP'11*, 2011.

[26] E. Komendantskaya. Unification neural networks: unification by error-correction learning. *Logic Journal of the IGPL*, 19(6):821–847, 2011.

[27] E. Komendantskaya. ML-CAP home page, 2012. http://www.computing.dundee.ac.uk/staff/katya/MLCAP-man/.

[28] E. Komendantskaya and J. Power. Coalgebraic derivations in logic programming. In *CSL'11*, 2011.

[29] E. Komendantskaya and J. Power. Coalgebraic semantics for derivations in logic programming. In *CALCO'11*, 2011.

[30] E. Komendantskaya and A. Seda. On approximation of the semantic operators determined by bilattice-based logic programs. In *FTP'05*, pages 112–130, 2005.

[31] J. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

[32] J. Lloyd. *Logic for Learning: Learning Comprehensible Theories from Structured Data.* Springer, Cognitive Technologies Series, 2003.

[33] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[34] K.-R. Muller, S. Mika, G. Ratsch, K. Tsuda, and B. Scholkopf. An introduction to kernel-based learning algorithms. *Neural Networks, IEEE Transactions on.*, 12(2):181, 2001.

[35] D. Nauck, F. Klawonn, R. Kruse, and F.Klawonn. *Foundations of Neuro-Fuzzy Systems.* John Wiley and Sons Inc., NY, 1997.

[36] A. Passerini, P. Frasconi, and L. D. Raedt. Kernels on prolog proof trees: Statistical learning in the ilp setting. *Journal of Machine Learning Research*, 7:307–342, 2006.

[37] J. Rutten. Universal coalgebra: a theory of systems. *TCS*, 2000.

[38] V. Sorge, A. Meier, R. L. McCasland, and S. Colton. Automatic construction and verification of isotopy invariants. *J. Autom. Reasoning*, 40(2-3):221–243, 2008.

[39] E. Tsivtsivadze, J. Urban, H. Geuvers, and T. Heskes. Semantic graph kernels for automated reasoning. In *SDM'11*, pages 795–803. SIAM / Omnipress, 2011.

[40] J. Urban, G. Sutcliffe, P. Pudlák, and J. Vyskocil. Malarea sg1- machine learner for automated reasoning with semantic guidance. In *IJCAR*, LNCS, pages 441–456. Springer, 2008.

[41] V. Vapnik. *Statistical Learning Theory.* John Wiley & Sons.

[42] L. Vlacic. Learning and soft computing, support vector machines, neural networks, and fuzzy logic models, vojislav kecman. *Neurocomputing*, 47(1-4):305 – 307, 2002.

[43] J. Wang and P. Domingos. Hybrid markov logic networks. In *AAAI*, pages 1106–1111, 2008.

[44] L. Zadeh. Interpolative reasoning in fuzzy logic and neural network theory. *Fuzzy Systems*, pages 1–20, 1992.