Learning Based Programming

Dan Roth
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave. Urbana 61801
danr@cs.uiuc.edu

Abstract
Learning Based Programming is a programming paradigm that extends conventional programming to support writing programs in which some of the definitions are generated in a data driven way and some are learned from observations the program encounters. The paper introduces the paradigm as well as the design and implementation of a specific learning based programming language (LBP) within it.

LBP allows the programming of complex systems whose behaviors depend on naturally occurring data and that require reasoning about data and concepts in ways that are hard, if not impossible, to write explicitly. In LBP the programmer can reason using high level concepts without the need to explicitly define all the variables they might depend on, or the functional dependencies among them. Instead, LBP supports reasoning in terms of the information sources that might contribute to decisions and allows the specific to be determined in a data-driven way.

We provide the formalisms for the main constructs of the language, including knowledge representations and learning constructs as well as the notions of an interaction and sensors through which an LBP program interacts with its environment. Examples and experimental evidence are given from natural language and visual processing applications.

Introduction
...these machines are intended to carry out any operations which could be done by a human computer ... supposed to be following fixed rules. ... We may suppose that these rules are supplied in a book which is altered whenever he is put on to a new job....

A. M. Turing, Computing Machinery and Intelligence, sec. 4, 1950 (Turing 1950)

The fundamental question that led Turing and others to the development of the theory of computation and of conventional programming languages is the attempt to model and understand the limitation of mechanical computation. Programming languages were then developed to simplify the process of setting the explicit rules of computation that digital machines can follow.

Learning Theory attempts to study how concepts can be defined and represented without the need for an explicit programming. Quoting (Valiant 1984), it attempts to study models of “what can be learned just as computability theory does on what can be computed”.

This work develops a programming paradigm that extends conventional programming languages by allowing a programmer to write programs in which some of the definitions are not made explicit but instead, are learned from experience. In the same way that conventional programming languages allow the design and implementation of large scale software systems and rely on a programmer to explicitly define all the concepts and relations involved, the goal here is to develop a programming model that supports building large scale systems in which some components cannot be explicitly defined by a programmer; which accepts that some of the variables, concepts and relations may not be known at programming time, may be defined only in a data driven way, or may not be defined unambiguously without relying on other concepts acquired this way.

These situations are abound when trying to build systems that interact with naturally occurring data (text, speech and images, or even stream of digital data originating from a multi-media or financial application or biological sequences) and reason with respect to concepts that are complex in terms of the raw data observed (but may be simpler in terms of some other concepts induced from it). Consider, for example, a human-machine interaction situation in which a program interacts with user’s queries in order to extract information from a knowledge base. The interaction might incur context-sensitive ambiguities at several levels of interpreting the user’s queries. A program handling it needs to tolerate a query presented in the context of a previous one, user’s input errors (e.g. typing our instead of out, now instead of know; spurious pronunciations), domain specific vocabulary and context specific usage of it, and adapt to the styles and idiosyncrasies of different users. As a second example consider an automatic aid that analyses a surveillance tape; it may need to recognize indoor and outdoor scenes, focus on humans in the image, identify gender and style of clothing, identify gestures etc. Inferences made by this analyzer depend on
a large number of hierarchical decisions with respect to the observed. Given that no two inputs observed by this program are ever the same, it seems impossible to explicitly write down the dependencies among all factors that may potentially contribute to a decision, or a Turing-like (see citation above) list of definitions and rules the program can follow in its processing.

This paper describes a programming paradigm as well as the design and implementation of a specific learning based programming language (LBP) within it.

LBP allows the programming of complex systems whose behaviors depend on naturally occurring data and that require reasoning about data and concepts in ways that are hard, if not impossible, to write explicitly. In particular, LBP allows reasoning about systems that depend, in principle, on a very large number of predicates and perhaps hierarchies of those. The programmer can reason using high level concepts without the need to explicitly define all the variables they might depend on, or the functional dependencies among them. And, it supports reasoning in terms of the information sources that might contribute to decisions, while the specifics of which variables are extracted from these information sources and how to combine them is to be determined in a data-driven way.

Our main motivation in developing this direction is that of cognitive computations - computations that rely heavily and acquire the bulk of their knowledge from raw, real world data, and behave robustly when presented with new previously unseen situations. We believe, however, that the computational approach and programming environment developed here will contribute in domains such as human machine interaction, embedded systems, adaptive software and software engineering and thus benefit computer science in general.

This line of research can be viewed both as an end point of a line of research in machine learning - building on the maturity level the machine learning community has reached in understanding and in the design of classifiers - and as a beginning of a promising new programming discipline. We attempt to push research away from the study of single classifiers to the study of \textit{systems that learn}; research on the theory and practice of systems in which many classifiers abound, concepts are defined in terms of the outcome of induced concepts and one can reason not only about explicitly defined concepts but also at a higher level, with respect to concepts and definitions that may be very complex in terms of the raw observations.

In the rest of the paper we present the LBP programming model: discuss, abstractly, the three main technical issues introduced in developing LBP - the knowledge representation, the interaction constructs and the learning operators; mention some theoretical issues that arise and hint on implementation and experiments.

\section*{Programming Model}

A computer program can be viewed as a system of definitions. Programming languages are designed to express definitions of variables in terms of other variables, and allow the manipulations of them so that they represent some meaningful concepts, relations or actions in a domain. In traditional programming systems the programmer is responsible for setting up these definitions. Some may be very complicated and may require a hierarchy of functions to be defined.

In LBP variables can be set up in a data driven way, and variables’ definitions in terms of other variables can be generated in a data driven way, all by interactions with data sources. Specifically, an LBP program extends traditional programming in two ways:

- It allows the definition of \textit{types} of variables, that may give rise to (potentially infinitely) many variables, determined in a data-driven way.

- It allows some of the definitions to be learned.

LBP provides a formalism and an implementation that allows a programmer to set up definitions only by declaring the target concepts (what is being defined) and the information sources that, potentially, contribute to the definition. The specific variables participating in the definitions and the exact dependencies are then determined during a data-based compilation process. As in traditional programming, the programmer can set up a hierarchy of definitions; learning, however, is done one definition at a time. Here, too, the programmer is responsible to ensure that variables used at some stage in the program have already been defined.

An LBP program is data driven. That is, some of the variables are defined and initialized only when the program interacts with data. The notion of an \textit{interaction} is thus fundamental in LBP. An LBP program interacts with real world data and is actually \textit{determined} by the data it interacts with. This is supported by abstracting the interaction in terms of sensors. These are preprogrammed or learned functions whose role is to extract useful parts of the observations the program interacts with, so as to allow their manipulation and use it in defining concepts. This also allows LBP to give a uniform treatment to programmed knowledge, external knowledge sources and induced representations.

An LBP program is a set of possible programs. The specific element in this set of possible programs that will perform the computation is determined either in a \textit{data based compilation} stage or at run-time. Although LBP is formalized in the traditional manner, in the sense that we provide a syntax and operational semantics, we focus here on describing an abstract view of some of the constructs so that the general view of the language and programming with it is clear.

\section*{Knowledge Representations for LBP}

LBP extends the set of usual variables used in C/C++ and introduces a new set of \textit{relational variables}. These variables are sometimes called “features” in the machine learning terminology and we use the term “relational” in order to emphasize that they can actually stand for relational (quantified) entities. The relational
variables are all formulae in a relational language \( \mathcal{R} \) that we now define. We then describe how a programmer can initiate a data driven generation of these variables (rather than explicitly define all of them), how they can be manipulated and how they can be defined in terms of others. For most of the discussion that follows we treat the relational variable as Boolean variables, taking only values 0 or 1 (which we sometimes call “active” and “non-active”). It will later be clear that these variables can also take real-values (in the range \([0, 1]\)).

The relational language \( \mathcal{R} \) is a restricted function-free first order language defined in the standard way (Lloyd 1987) with the restriction that all formulae have only one single predicate in the scope of each variable. Therefore, a ground atomic formula is a proposition and a quantified atomic formula, a quantified proposition (Khadron, Roth, & Valiant 1999). We use the term relational variable for both. The informal semantics of the quantifiers and connectives is as usual. Note that for formulae in \( \mathcal{R} \), the scope of a quantifier is the unique predicate that occurs with it in the atomic formula.

Relational variables in \( \mathcal{R} \) receive their “truth values” in a data driven way, with respect to an observation.

**Definition 1** An instance is an interpretation (Lloyd 1987) which lists a set of domain elements and the truth values of all instantiations of the predicates on them.

Given an instance \( x \), a formula \( F \) in \( \mathcal{R} \) is given a unique truth value, the value of \( F \) on \( x \), defined inductively using the truth values of the predicates in \( F \), and the semantics of the connectives.

**Relation Generation Functions**

We view a formula in \( \mathcal{R} \) as a relation \( F : x \to \{0, 1\} \), which maps the instance to its truth value on \( x \). A formula is active in \( x \) if it has truth value true in this instance. We denote by \( X \) the set of all instances, the instance space. A formula \( F \in \mathcal{R} \) is thus a relation over \( X \), and we call it a relational variable.

**Example 1** An instance \( x \) would be defined using a specific data structure later. At this point we could think about it, say, as an unordered collection of words: he, ball, the kick, would. If word and tag are predicates then some active relations on this instance are word(he), word(ball), and tag(DET) (assuming one knows how to compute these predicates given the instance). In this case we could say that the variable word(he) has value 1 in the instance. If object is another predicate, the variable object(z) represents an unbound formula that would evaluate to true if a word exists in the input instance which is an object in it.

We now define one of the main constructs in LBP. The notion of relation generating functions allow an LBP programmer to define a collection of relational variables without writing them explicitly. This is important, in particular, in the situations for which we envision LBP is most useful; that is, over very large (or infinite) domains or in on-line situations where the domain elements are not known in advance, and it is simply impossible to write down all possible variables one may want to define. However, it may be possible for the programmer to define all “types” of variables that might be of interest.

**Definition 2** Let \( \mathcal{X} \) be an enumerable collection of relations over the instance space \( X \). A relation generation function (RGF) is a mapping \( G : X \to 2^\mathcal{X} \) that maps an instance \( x \in X \) to a set of all elements in \( \mathcal{X} \) that satisfy \( \chi(x) = 1 \). If there is no \( x \in X \) for which \( \chi(x) = 1 \), \( G(x) = \phi \).

RGFs provide a way to define “kinds” of formulae (relational variables), or to parameterize over a large space of variables. Only when an instance \( x \) is presented, a concrete variable (or a collection of) is generated.

**Relational Calculus**

The family of relation generation functions for \( \mathcal{R} \) are RGFs whose output are variables (formulae) in \( \mathcal{R} \). Those are defined inductively, just like the definition of the language \( \mathcal{R} \). The relational calculus is a calculus of symbols that allows one to inductively compose relation generation functions. The alphabet for this calculus consists of (i) basic RGFs, called sensors and (ii) a set of connectives. While the connectives are the same for every alphabet the sensors vary from domain to domain. A sensor is a way to encode basic information one can extract from an instance.

**Definition 3** A sensor is a relation generation function that maps an instance \( x \) into a set of atomic formulae in \( \mathcal{R} \). When evaluated on \( x \), a sensor \( s \) outputs all atomic formulae in its range which are active.

Several mechanisms are used in the relational calculus to define the operations of RGFs. These include (1) a conditioning mechanism, that restricts the range of an RGF to formulae in a given set (or those which satisfy a given property), (2) a focus mechanism that restricts the domain of an RGF to a specified part of the instance and (3) a naming mechanism that allows an easy manipulation of RGFs and a simple way to define new RGFs in terms of existing ones. The operation of the RGFs is defined inductively starting with the definitions of the sensors, using these mechanisms and standard definitions of connectives.

**Structural Instance Space**

An important part of the relational calculus is the ability to exploit the structure of a domain of interest. This is abstracted using the notion of a structural domain. Instances in a structural domain are augmented with some structural information and, as a result, it is possible to define more expressive RGFs in terms of the sensors provided along with the domain.

**Definition 4** Let \( D \) be the set of elements in the domain. A structured instance \( x \) is a tuple \((V, E_1, E_2, \ldots, E_k)\) where \( V \subseteq D \) is a set of elements in the domain, and \( E_i \) is a set of edges on \( V \). The graph
$G_i = (V, E_i)$, is called the $i$th structure of the instance $x$ and is restricted to be an acyclic graph on $V$.

**Example 2 (NLP)** Let $D$ be the set of all words in English. A structured instance can correspond to a sentence, with $V$, the set of words in the sentence and $E = E_1$ describes the linear structure of the sentence. That is, $(v_i, v_j) \in E$ if the word $v_i$ occurs immediately before $v_j$ in the sentence.

**Example 3 (VP)** Let $D$ be the set of all positions in a 100 x 100 gray level image. A structured instance can correspond to a sub-image, such that $V$ is the set of pixels in the sub-image and $E = E_1$ describes the top-down and left-right adjacencies in it. That is, $(v_j, v_k) \in E$ if the pixel $v_j$ is either immediately to the left or immediately above $v_k$.

The relational calculus is augmented by adding structural operations. These operations exploit the structural properties of the domain as expressed in the graphs $G_1$, $G_2$ in order to define RGFs, and thereby generate non-atomic formulae that may have special meaning in the domain. Basically, structural operations allow to construct RGFs that conjunct existing RGFs evaluated at various nodes of the structural instances. The restriction we impose, a crucial one for computational reasons, is that these each operation considers only nodes that are on a chain in one of the graphs of the structured instance. Detailed information and computational properties appear in (Author, KR’2000).

**Example 4 (NLP)** Say we are interested in Subject-Verb relations in sentences, and assume that the structured instance consists of, in addition to the linear structure of the sentence ($G_1$), a graph $G_2$ encoding functional relationships among the words. And RGF can be written that would extract Subject-Verb relations.

**Example 5 (VP)** To find an edge relation in an image, we might define an RGF using a structural operation building on a sensor’s producing active relations for pixels with intensity value above 50.

**Interaction**

An LBP program interacts with its environment via a data structure called a structured instance along with a set $S$ of sensors that act on this data structure. A structured instance is an implementation of Def. 4. This is a list of records (place holders) each of which consists of a list of properties. The place holders represent the nodes in the definition and the properties (e.g., word, tag, intensity) can be viewed as predicates that hold in this instance. The structure of the instance is described by edges, labeled with the graph name, that point to adjacent records in the graph.

**Example 6** Along with observations which represent a sentence or an image, the program has a set of sensors which can operate on the observation. Sensors may extract information directly from the input, such as word, tag, or intensity, but can also utilize outside knowledge in processing the input; for example, a vowel sensor (which output an active variable if its focus word starts with a vowel), needs to use some information to determine its output. An ISA sensor may need to access an external data structure such as wordnet in order to return an active value (e.g., when focused on “desk” it might return furniture among the active variables).

An LBP programmer needs first to map the real world data available (text, images, biological sequences) into a format that can be parsed into structured instances and define basic sensors accordingly. From that point, programming proceeds as usual, and may consist of manipulating conventional variables, defining new RGFs as functions of sensors, evaluating them on observations or defining new RGFs without supplying explicit definition for how they are computed, which we discuss next.

**Learning Operators in LBP**

The knowledge representations discussed before provide a way to generate and evaluate efficiently intermediate representations given an observation.

**Definition 5** Let $X$ be the instance space, $C = \{c_1, \ldots, c_k\}$ a discrete set of labels. A set-function $G : X \to C$ maps an instance $x \in C$ to a set $c \subseteq C$ of labels. $G$ is a multi-valued function if $|c| = 1$.

An RGF maps an observation into a collection of Boolean values, a set of variables, and is thus a set function. When defined using the operator L below, an RGF would be used mostly as a multi-valued function.

The $E$ operator is used to evaluate an RGF (or a collection of RGFs). Given as input an RGF(s) and an instance it generates a list of all relations specified by the RGFs that are active in the instance. $E$ has several parameters that we do not describe here. These allow the programmer, for example, to select only a subset of the set of active variables. Specifically, when an RGF associate a real values (in $[0,1]$) with the variables in its range, $E$ can output a list sorted by these values along with the values themselves, or choose the top variable.

**Definition 6** Let $Y$ be the collection of all possible variables specified by the programmer or generated by the RGFs $r_1, \ldots, r_k$, and $x$ an instance. An example $e$ is an element in $2^Y$; that is, it is a list of variables names defined by $e \subseteq E(r_1, r_2, \ldots, r_k, x)$.

Notice that this example representation is reminiscent of the infinite attribute model (Blum 1992). It is useful especially when the number of variables is very large, but relatively few are active in each observation. Notice also that an explicit notion of a label does not exist here. Any one of the variables (features) in the example can be treated as a label. If needed, the programmer can define an RGF to focus on part of the observation and use it as a label or can generate an RGF specifically to label an observation using external information.

We now use the formalisms developed so far to learn representations of RGFs from observations. The L operators learns a definition of a multi-valued RGF in
terms of other variables. The L operators receives as input a collection \( r_1, \ldots, r_k \) of RGFs along with a collection of instances and a set of values (names of variables) \( T = \{ t_1, \ldots, t_k \} \) that determines its range. Elements in the set \( T \) are called the target variables. Notice that the new RGF is defined to be a function of the variables produced by RGFs in its domain, but the variables themselves need not be specified: only “types” of variables are specified, by listing the RGFs. %beginequation
\[
T = L(T, r_1, r_2, \ldots, r_k, \text{instances}).
\]%endequation
This expression defines a new RGF that has the variables names in \( T \) as its possible values. Which of these values will be active when \( T \) is evaluated on a future instance is not specified directly by the programmer, and will be learned by the operator L.

L is implemented via an on-line multi-class learner. When computing L, the RGFs \( r_1, \ldots, r_k \) are first evaluated on each instance to generate an example; the example is then fed into L to update its representation for all the elements in \( T \) as a function of the variables that are currently active in the example. The specific implementation of the operator L is not important at this level. There are only a few functional requirements that any implementation of L needs to satisfy. The learner needs to be a supervised multi-class learner, work in an on-line fashion and in the infinite attribute model.

Examples are represented simply as a list of variables. The operator L treats each of the targets as an autonomous learning problem. All the examples that have \( t \) active in them are treated as positive for \( t \), and all the others are treated as negative for it. The occurrence of one (or several) target variables in the example is a function of the RGFs specified by the programmer and the observation. It is the responsibility of the programmer to make sure that the learning problem set up by defining \( T \) as above is well defined.

Once so defined, \( T \) is a well defined RGF that can be evaluated by the operator \( E \). Moreover, each of the variables in the target set \( T \) is (trivially) an RGF in itself, only that it has only a single variable in its range.

The Learning Algorithm

The operator L has several modes which we do not describe here. Moreover, we purposefully abstract away the learning algorithm itself. We view the L operator essentially as the multiplication operation in conventional programming languages in that the specific details of the algorithm used can be are hidden from the programmer. We note that from the point of view of the operator L, learning in LBP is always supervised. However, this need not be the case at the application level. It is quite possible that the supervision is “manufactured” by the programmer. Examples may include learning from text (e.g., context sensitive spelling (Golding & Roth 1999)) where at the application level the problem is unsupervised (the program assumes that the text given to it is correct), but an element of the data (in this case, one of the words in the text) is supplied by the programmer as supervision. Labels can also be supplied from external sources, using an RGF written specifically for that, or may be generated as part of the program, as done in bootstrapping applications (Collins & Singer 1999).

Theoretical Aspects

We have addressed several theoretical issues related to LBP that we mention briefly.

Knowledge Representation: The structural operations allow for RGFs that output formulae that share variables across predicates, but manages to avoid incurring the cost usually associated with enlarging the scope of free variables (the problem of subsumption). This is done by enlarging the scope only as required by the structure of the domain, modeled by the \( G_i \)s. (In Author, KR’00) this problem is discussed and it is shown that evaluating an RGF can be done efficiently.

Learning and Correctness LBP makes certain requirements on the type of output and robustness expected from the learning mechanisms. The issues that need to be discussed here include the semantics of the RGFs, correctness and learnability. Briefly, the semantics is PAC semantics (Valiant 1984; 1990) which guarantees that RGFs learned are probably approximately correct with respect to the same fixed (but unknown) distribution of instances on which they were trained. The crucial issue of the correctness of chaining RGFs is addressed in (Valiant 1999) where it is shown that in our context, when RGFs are learned one at a time, there is no accumulation of error. Finally, for learnability, we mention that the functional form of all our RGF is that of a linear threshold function in terms of the variables generated by the RGFs. RGFs are programmed by the programmer so as to enrich the feature space and ensure that linear representation are expressive enough. It is an experimental issue whether this might impose on the learning mechanisms that implements L more constraints than we have mentioned above (e.g., the necessity to use feature efficient algorithm (Littlestone 1988)).

Implementation

LBP is devised as an extension of C/C++. Several additional constructions, data structures and operators are developed which allow the programmer to write an LBP program. The current implementation is as a collection of libraries and an API (Application Programming Interface). This allows the programmer to program a C++ program and use all the extensions described above. The program is then compiled using a C++ compiler and then runs through a data-driven compilation process (a training process) or in an on-line mode; both are transparent to the programmer.

The knowledge representations constructs and the interaction model have been implemented in a KR language that is described elsewhere (Author, KR’00)
along with its theoretical properties. The learning operators are implemented using the SNoW learning architecture (Roth 1998; Carleson et al. 1999). SNoW can be viewed as an on-line multi-class learner that learns in the infinite attribute model and is thus appropriate for an implementation of the operator L. Moreover, SNoW learns a representation for each of the target variables and returns an activation value for each of them. The operator E can then produce the appropriate outcome, based on the programmer requirements.

Programming examples

LBP has already been used to develop programs that have significant learning components in the natural language and visual processing domain. We briefly describe a Shallow Parsing application (e.g., (Ramshaw & Marcus 1995; Cardie & Pierce 1998; Munoz et al. 1999)). This program uses several sensors, including “word” “default-pos” (most common pos for a word). It sets up explicit RGFs that represent complex features in terms of the sensors and uses them to learn and RGF that produces the correct pos of a word in its context. As a function of these it sets up RGFS that learn to recognize beginning of important phrases (such as noun phrases and subject-verb phrases). These, in turn, along with the previous ones are used to (learn to) represent end of phrases. Since the program may identify several potential beginnings and ends, a constraint satisfaction algorithm is then used on top of the (learned) RGFS to pair them and select a coherent set of phrases.

Not only the this application has been very successful in terms of performance, we found that programming with LBP allows the programmer to reason at a level above the classification. LBP has been used also in the visual domain, where it has been used to learn definitions for faces and then to search for human faces in images, as well as to recognize objects and then search for them in images (references suppressed).

Conclusion

We have described a new programming paradigm that extends conventional programming and support writing programs in which some of the definitions are generated in a data driven way and some are learned from observations the program encounters. We believe that LBP is required if we are to write large scale programs that interact with and reason about real world data.

LBP is different from some more restricted efforts at developing programming languages for constructing of robot control software (Thrun 1998; Levesque et al. 1997) or stochastic programs (McAllester, Koller, & Pfeffer 1997). It is aimed at developing the ability to interact with real data sources across modalities, and reason about concepts and definitions that are not apparent in the raw data, using definitions that cannot be written explicitly in terms of the data. As such, we had to address a variety of issues in knowledge representation and learning in an effort to abstract away the learning component and at the same time provide an expressive, domain independent language. The notions of interaction and sensors were introduced to supply the link to the domain and have already shown to be effective in allowing the use of LBP in several domains.

Further work on LBP would include both theoretical work – developing representations and interaction modes and studying the learning operator as well as experimental work with the programming paradigm.

References


