Embedded Machine Learning
Papers from the AAAI 2015 Fall Symposium

Saul: Towards Declarative Learning Based Programming

Parisa Kordjamshidi, Dan Roth, Hao Wu
University of Illinois at Urbana-Champaign
{kordjam,danr,haowu4}@illinois.edu

Abstract

We present *Saul*, a new probabilistic programming language designed to address some of the shortcomings of programming languages that aim at advancing and simplifying the development of AI systems. Such languages need to interact with messy, naturally occurring data, to allow a programmer to specify what needs to be done at an appropriate level of abstraction rather than at the data level, to be developed on a solid theory that supports moving to and reasoning at this level of abstraction and, finally, to support flexible integration of these learning and inference models within an application program. *Saul* is an object-functional programming language written in Scala that facilitates these by (1) allowing a programmer to learn, name and manipulate named abstractions over relational data; (2) supporting seamless incorporation of trainable (probabilistic or discriminative) components into the program, and (3) providing a level of inference over trainable models to support composition and make decisions that respect domain and application constraints. *Saul* is developed over a declaratively defined relational data model, can use piecewise learned factor graphs with declaratively specified learning and inference objectives, and it supports inference over probabilistic models augmented with declarative knowledge-based constraints. We describe the key constructs of *Saul* and exemplify its use in developing applications that require relational feature engineering and structured output prediction.

1 Introduction

Developing intelligent problem-solving systems for real world applications requires addressing a range of scientific and engineering challenges.

First, there is a need to interact with messy, naturally occurring data: text, speech, images, video and biological sequences. Second, there is a need to specify what needs to be done at an appropriate level of abstraction, rather than at the data level. Consequently, there is a need to facilitate moving to this level of abstraction, and reasoning at this level. These abstractions are essential even though, in most cases, it is not possible to declaratively define them (e.g., identify the topic of a text snippet, or whether there is a running woman in the image), necessitating the use of a variety of machine learning and inference models of different kinds and composed in various ways. And, finally, learning and inference models form only a small part of what needs to be done in an application, and they need to be integrated into an application program that supports using them flexibly.

Consider an application programmer in a large law office, attempting to write a program that identifies people, organizations and locations mentioned in email correspondence to and from the office, and identify the relations between these entities (e.g., person A works for organization B at location C). Interpreting natural language statements and supporting the desired level of understanding is a challenging task that requires abstracting over variability in expressing meaning, resolving context-sensitive ambiguities, supporting knowledge-based inferences, and, doing it in the context of a program that works on real world data. Similarly, if the application involves interpreting a video stream for the task of analyzing a surveillance tape, the program needs to reason with respect to concepts such as indoor and outdoor scenes, the recognition of humans and gender in the image, identifying known people, movements, etc., all concepts that cannot be defined explicitly as a function of raw data and have to rely on learning and inference methods.

Several research efforts have addressed some aspects of the above mentioned issues. Early efforts within the *logic programming* framework took a classical logical problem solving approach. Logic based formalisms naturally provide relational data representations, can incorporate background knowledge thus represented and support deductive inference over data and knowledge (Huang, Green, and Loo 2011). Of course, dealing with real-world data requires some level of abstraction over the observed data, and the formalism has been extended to support relational learning (Frasconi et al. 2012) but is still short of dealing with expressive and joint models, taking into account interacting variables, and more. In order to address some of these issues and better deal with uncertainty, most of the recent work in this direction has focused on *probabilistic programming* frameworks, some of which attempt to combine probabilistic representations with logic. Earlier attempts go back to BUGS (Gilks, Thomas, and Spiegelhalter 1994), AutoBayes (Fischer and Schumann 2003) and PRISM (Sato and Kameya 1997), and several new efforts attempt to develop languages to better support defining a probability distribution over the domain and reasoning with this representation (Raedt and Kersting 2004;

Goodman et al. 2008; Pfeffer 2009; Mansinghka, Selsam, and Perov 2014). Some of these languages are equipped with relational languages to facilitate the design of complex probabilistic models declaratively (Domingos and Richardson 2004) or imperatively (McCallum, Schultz, and Singh 2009). However, in all cases, these efforts focus on flexibly defining one joint probability distribution. They do not intend to support the development of applications that require combining, nesting and pipelining multiple models and running inference over these, nor to support the need to flexibly mix deterministic and probabilistic models (Mateescu and Dechter 2008). Moreover, current probabilistic programming languages do not address the desiderata outlined earlier and do not support the natural work flow of developing applications from the interaction with real world data, through learning models and combining them in different ways, to making coherent, global, decisions over a large number of variables with a range of interaction patterns.

We present *Saul*, a new probabilistic programming language designed to address all the desiderata of a *Learning based program* (Roth 2005). *Saul* is an object-functional programming language written in Scala (Odersky, Spoon, and Venners 2008), that supports declarative definitions of trainable probabilistic or discriminative models and embedding them in programs, along with declarative definitions of inference patterns and objective functions over trainable models. This way it allows the development of programs that interact with real world data, allowing a programmer to abstract away most of the lower level details and reason at the right level of abstraction.

Saul can be viewed as a significant extension of an existing instance of the LBP framework, LBJava (Rizzolo and Roth 2010), where learned classifiers, discriminative or probabilistic, are first class objects, and their outcome can be exploited in performing global inference and decision making at the application level. However, in addition to these features, *Saul* supports joint learning and inference with structured outputs. From a theoretical perspective *Saul*'s underlying models are general factor graphs (Kschischang, Frey, and Loeliger 2001) extended to relational representations (Taskar, Abbeel, and Koller 2002) and expressing constrained factors (Cunningham, Paluri, and Dellaert 2010; Martins 2012) with directional semantics to represent various learning and inference architectures. This expressive power provides the possibility of mapping a *Saul* program to underlying *mixed networks* (Mateescu and Dechter 2008) that can consider both deterministic and probabilistic information. The inferences supported by *Saul* are all (extensions of) maximum a posteriori (MAP) inferences formulated, without loss of generality, as integer linear programs (Roth and Yih 2007; Sontag 2010), even though some of the inference engines within *Saul* may not perform exact ILP.

Learning and inference in *Saul* follows the Constrained Conditional Models (CCMs) (Chang, Ratinov, and Roth 2012) paradigm, allowing MAP inference over jointly or piecewise learned networks (Heckerman et al. 2000; Roth and Yih 2005). The CCM framework augments learning and inference with conditional (probabilistic or discriminative) models, with declarative constraints. Constraints are used as a way to incorporate expressive prior knowledge into the models and bias the assignments made by the learned models. When joint learning is done for a piece of the network, learning follows the *constraint-driven learning* (CODL) (Chang, Ratinov, and Roth 2007) or *posterior regularization* (Ganchev et al. 2010; Samdani, Chang, and Roth 2012) paradigms.

This programming paradigm allows one to think at the problem formulation level and incorporate domain and application specific insights in an easy way. The corresponding programming framework thus supports integrated learning and reasoning via (soft) constrained optimization; this allows for making decisions in an expressive output space while maintaining modularity and tractability of training and inference. This ability of *Saul* is augmented with its *data modeling* language: a declarative language via which the programmer can declare the relational schema of the data. This is used to support flexible structured model learning and decision making with respect to a customizable objective function.

Saul is intended to have some elements of automatic programming, by supporting programming by default at several steps of the computation. Once the programmer defines the data model by specifying what can be seen in the given data, and defines a set of variables of interest that will be assigned values, a default set of models can be learned and tuned automatically from data, and global inference done via a default objective function supports making coherent decisions that agree with the declared interdependencies among output variables. The programmer can then choose to further refine the data model, the structured model and the training and inference paradigms via declarative constructs, providing high level guidance for relational feature engineering, decomposing training and inference, or combining models. This way, *Saul* separates the layer of designing models from that of the underlying code for learning and inference and supports embedding sophisticated learning and inference models within application code in a seamless way.

In the rest of this paper we provide a motivating example exhibiting the advantages of *Saul* programming, sketch some of the key abstractions and relational representations that underlie it, and discuss the underlying probabilistic paradigm supporting learning and inference of structured models, along with composition, nesting and pipelining. We conclude with a discussion of the prospects of the *Saul* language.

2 *Saul*: The Language Components

In this section we provide an overview of the main constructs of the language. The *Saul* language has been implemented as a domain specific language (DSL) within Scala to easily provide Turing-complete functionality, as well as the possibility of exploiting the most recent advancements in programming languages and integrated development environments (IDEs).

This paper exemplifies *Saul* components using the *Entity-mention-Relation*[1] extraction task which aims at identifying

[1]Note that the terms *entity* and *relation* are overloaded and are used both to refer to *named entities* and semantic relations between

named entities of types *person*, *location*, *organization* and relations of types *lives-in*, *works-for*, in a given text. For this application, *Saul* is given as input free form text, and the programmer then writes a program that makes use of the identified named entities and relations. Within *Saul*, some components of the code are devoted to *training* models and instructing *Saul* on how to make *decisions (inferences)* – how to decide what entities and relations are present in a given text. For the sake of training, *Saul* is given as input *annotated data*, where text is augmented with the identified named entities and the identified relations of the aforementioned types. The rest of this article is devoted mostly to describing how *Saul* deals with annotated data, but it is important to realize that, in the end, *Saul* allows the programmer to develop an application in which these learned models and decision paradigms are first class objects.

A *Saul* program includes two main types of declarations to support problem solving: (i) data and knowledge representation, and (ii) learning and inference model declarations. In the standard machine learning setting, models are trained given a set of labeled examples. In *Saul*, an example consists of two sets of instantiated variables; a set of observed variables, which will be present in future observation at runtime (when the trained models are *used* on "real" data), and a *set* of variables of interest, to which we want to assign values, and which are instantiated only in the training data. A *Saul* program interacts with the original datasets which we refer to as *Collections of data*. These collections give rise to examples, represented as feature vectors, which are generated automatically given declarations provided by the programmer. Flexible relational *Feature declarations* constitute an important part of the *Saul* language.

In addition to the information that comes from the *Collections of data*, the programmer can express higher level *Background knowledge* that could relate assignments to variables *Saul* observes or assigns values to. This is represented in a first-order-like logical language that *Saul* makes use of automatically, either in the feature extraction stage or at the learning and inference step, as a way to bias a learned model toward or away from some variable assignments. In addition to declare how raw data collections are transformed into the internal *Saul* representation, the programmer makes use of *Learning declarations* and *Inference declarations*. The former allow the programmer to define which variable(s) will be represented as learned functions of other variables (observed or not). The programmer can resort to strong defaults once the basic definitions are set, but *Saul* provides the flexibility of designing various training models and composing them to train local models, global models or pipelines, as will be clear in Sec 4. In particular, the learning process may follow different strategies during training and at runtime evaluation, as declared by the programmer. The *Inference declarations* determine how decisions (predictions) are made, and what interdependencies among variables are taken into account when making decisions.

Learning and Inference in *Saul* are facilitated by an underlying data model that accommodates all the data and knowledge in a unified relational framework and provides efficient and flexible access to all pieces of information about the problem. Hence, a part of the underlying infrastructure of the *Saul* language depends on determining and declaring (and evolving) the relational schema of the data.

them in our example, and to elements in *Saul*'s data model.

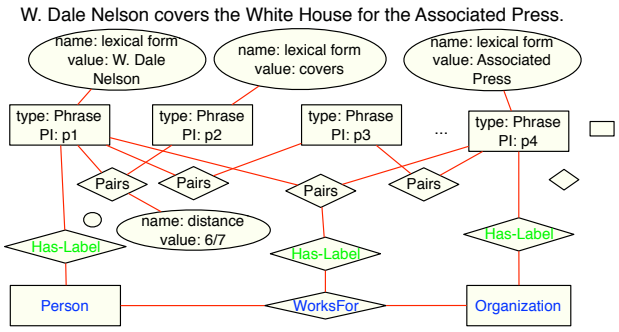

Figure 1: A part of the *data graph*; this is a propositionalized graph containing the complete instances as observed by *Saul*; rectangles show entities, diamonds show relationships and ovals show properties.

3 Data and Knowledge Representation

The *Saul* language provides the programmer easy communication with heterogeneous, real world data. To facilitate the use of various types of data and exploit its structure, we use a graphical representation that serves as a unified model for all the data used by a *Saul* program. We represent the data using two different but related graphs: *data graph* and *model graph* (Ghrab et al. 2013).

The *data graph* encodes the propositional information given in the individual (annotated) instances provided to *Saul* – it encodes what the program can "see" in the given data. The representation consists of three kinds of nodes: *entity nodes*, *relationship nodes* and *property nodes*, and edges that connect various nodes. An example of the *data graph* for the *Entity-mention-Relation* task is shown in Figure 1.

The *model graph*, on the other hand, is a first order graph that represents the relational schema of the data. The nodes in this graph indicate the *type*s (for both entities and relationships) and the edges that connect the various *type*s of nodes to each other. Edges have a degree that indicates a property of the relation (i.e., 1:1, 1:n or n:n).

The *model graph* in Figure 2, shows entities of type *Word* and *Phrase* and the *Contains* relationship between them. The words can have a *nextTo* relationship with each other. Each phrase contains a number of words. The phrases are related to each other by a *Pairs* relationship. This graph also represents the nodes that constitute the target variables, those variables we want to eventually assign values to in the *Entity-mention-Relation* task, and their assumed dependencies on input nodes. This connection is represented by relationship nodes of type *Has-label*, that indicate the range of each variable; *Has-label* is also being used to indicate the range of relationship nodes such as *Pairs*. E.g., it indicates that a pair of entity nodes could take the type *WorkFor*.

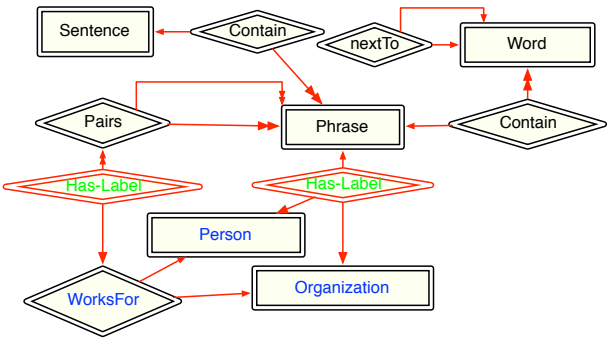

Figure 2: A part of the *model graph* of the *Entity-mention-Relation* task. We show the entities and relationships with double line indicating the classes of instances in contrast to the *data model* representation. The arrows show the degree of relationships in each direction.

Real world data supplied to *Saul* is assumed to be a dataset consisting of similar data items (e.g., a collection of text documents). *Saul* assumes that datasets come along with a *Reader*, a program that reads data into the aforementioned *data graph*. Note that the reader needs to read annotated data (and form a complete *data graph* that can be used by *Saul* to train models), but also runtime data that is not annotated and forms a partial *data graph*, in which some of the variables are not instantiated. Also note that the reader itself could be a *Saul* program, and use sophisticated models to generate the *data graph*. The description of *Saul* in the rest of this paper assumes the graph representations described above as input.

3.1 Graph Evolution

One important idea supported by *Saul* that, in turn, facilitates some of *Saul* properties, is that of building new layers of abstraction from the raw data and subsequently learning and inference with more expressive models in terms of these. *Saul* provides ways to generate new nodes and edges in the graphs, via *Saul*'s learned models, given the initial *data graph* provided by the *Readers*. We use the term *functor* to refer to a function $F : \{\tau\} \mapsto \{\tau'\}$, that maps one or more typed collection to one or more new typed collection. The *functor*s make changes to the *data graph*, that could also impact the *model graph*, by providing richer and more expressive vocabulary for the programmer to use. The graphs can evolve using two types of functors: *extractors* and *sensors*. Sensors are arbitrary functions that have external sources of information outside the *data graph* and the information that the Readers can provide directly. Their computational mechanism is a black box for the *Saul* language and they can be viewed as determining what *Saul* can "see" in the data (a "word", "prefix", etc.). Extractors are *Saul* programs that act on the information contained in the *data graph* and make use of the background knowledge representation, described in the next section, to generate new information by doing inference. *Saul* consists of a relational calculus that can be used over the sensors and extractors to generate other sensors and extractors, in a way similar to other Feature Extraction languages (Cumby and Roth 2003). The classic *Join*, *Project* and *Select* operations and the logical combinations of these operators thus support evolving the *data* and *model graph*s.

3.2 Relational Features and Graph Queries

In the classical learning setting, information about a problem is represented using feature vectors representing the examples. Each learning example in *Saul* is a *rooted subgraph* of the *data graph* and the features for each example are directly derived from properties of the entities and relationships in the *data graph* and the sensor applied to them. Namely, each feature is an outcome of a query that extracts information from the *data graph* or from sensors. This can be viewed as an extension of relational feature extraction languages such as (Cumby and Roth 2003). A *Saul* programmer can simply list the graph queries he/she wants to use as feature "types" and these will be instantiated in the *data graph*s. The code excerpt below exhibits this for features of the type Phrase.

```
val EntityFeatures = FeaturesOf[Phrase] {
  x:Phrase=>{ pos(x)::lexForm(x)::lemma(x)::
  lemma(x.next())::pos(x.window(2))::Nil}}
```

This one line of code declares a list of queries from the *data graph* for retrieving the properties of an object of type Phrase in the *model graph*. The list includes the linguistic features of posTags, lemma, lexical form, in addition to a contextual feature which is the lemma of the adjacent Phrase, using the function `next` and the posTag of the phrases in a window with length 2 around that phrase.

Any Scala expression built on the graph queries can be used as a feature, and a programmer can always define a complex feature, name it, and use it in further definitions. Moreover, *Saul* is rich in defaults, and allows the programmer to not define features, in which case all properties based on the *model graph* declarations are used as default features.

3.3 Representing Background Knowledge

The *model graph* represents the concepts, their properties and the relationships between them but, often, it is important to represent higher level information; for example, to express expectations over the output predictions (Chang, Ratinov, and Roth 2007; Ganchev et al. 2010). These are often hard to express using the graph only, wasteful to try to acquire directly from the data, and an expressive declarative language could be easier and intuitive to use. Of course, exploiting this knowledge requires particular attention in both representation and learning and *Saul* builds on rich literature that guides us how to do it.

Considering relations between variables in the input boils down to representing input features and is relatively standard. However, encoding knowledge about interactions among output variables is more challenging. *Saul* can induce these interactions automatically in the learning process but, importantly, also supports expressing these declaratively, as constraints. These constraints are represented using first-order-like logical language that also supports quantifiers, allowing a programmer to easily express constraints, as in the example below, to indicate that the value predicted

for the relation between two argument variables constrains the values that could be assigned to arguments.

```
val WorkFor=constraintOf[Pairs] {
   x:Pairs => {
     ((WorkFor on x) is true) ==>
     ((PER on x.firstArg) is true) &&&
     ((WorkFor on x) is true) ==>
     ((ORG on x.secondArg) is true) } }
```

Here, for the input pair `x=(firstArg,secondArg)`, if `firstArg` and `secondArg` have the *WorkFor* relationship then `secondArg` should be an organization (ORG) and `firstArg` a person (PER). Via our defined `constraintOf` template, *Saul* compiles this declarative representation to a set of equivalent linear constraints automatically and efficiently (Rizzolo and Roth 2010)), and this is used as part of the objective function discussed later. It is important to note that while the constraints are represented declaratively, *Saul*'s learning and inference paradigms allow using them either as hard or soft constraints (Chang, Ratinov, and Roth 2012). As has been shown repeatedly in the literature over the last few years, this capability, which is simply encoded as a single construct in *Saul*, provides a unique flexibility for using background knowledge both in feature generation (e.g., as part of a pipeline process) and for constraining global inference.

4 Learning and Inference

The key innovations in *Saul* are in the learning and inference models – the representations used, the formulations facilitated and the flexibility within which an *Saul* programmer can declare, learn, and make decisions with a range of learning and inference paradigms.

The underlying representation of a *Saul* program is a general factor graph. This provides the expressivity of different kinds of probabilistic and discriminative models (Kschischang, Frey, and Loeliger 2001; Collins 2002; Roth and Yih 2005) and hence the possibility of using a variety of algorithms for learning and inference. The *Saul* factor graphs are significantly augmented to facilitate:

- Representing relational factors (Taskar, Abbeel, and Koller 2002),
- Representing constraint factors (Martins 2012; Cunningham, Paluri, and Dellaert 2010),
- Representing directional computation with *Saul* specific semantics.

The goal of this extension is to increase the expressivity of the standard factor graphs, support more expressive modeling architectures for learning and inference, and provide the most modularity and flexibility to the *Saul* programmer.

The training process in *Saul* can be formulated, without loss of generality, as follows: given a set of N input-output pairs of training examples with any arbitrary structure $E = \{(x^i, y^i) \in \mathcal{X} \times \mathcal{Y} : i = 1..N\}$, find a function $g(x, y; W)$ that assigns a score to a pair of input and output. Function g is the *objective function*, that is then used to make a prediction: given x, it predicts the best scoring y.

In the standard structured learning framework, this objective is assumed to be a linear discriminant function in terms of the combined feature representation of inputs and outputs f, that is, $g = \langle W, f(x, y)\rangle$, where W is a weigh vector representing the parameters of the linear model. By explicitly representing the relational, declarative knowledge that captures global characteristics of the model and (expectations of) constraints over output space assignments, objectives used in *Saul* are formulated, as a constrained conditional model (CCM) (Chang, Ratinov, and Roth 2012):

$$g = \langle W, f(x, y)\rangle - \langle \rho, c(x, y)\rangle, \tag{1}$$

where c is a constraint function and ρ is a vector of penalties for violating each constraint. A *Saul* factor graph thus corresponds to a general CCM objective that can be used in training probabilistic graphical models as well as generalized linear models (Roth and Yih 2005). When the constraints in the objective are treated as hard constraints it directly corresponds to an integer linear program, and thus can be used to encode any MAP problem. Specifically, the g function can be represented as the sum of local joint feature functions which are the counterparts of the probabilistic factors:

$$g(x, y; W) = \sum_{l_p \in \mathbf{l}} \sum_{x_k \in \{\tau\}} \langle W_p, f_p(x_k, l_{pk})\rangle + \sum_{m=1}^{|C|} \rho_m c_m(x, y) \tag{2}$$

where x and y are subgraphs of the *data graph*, and each f_p is a local joint feature function applied on a type τ (i.e. a *model graph* node) when it is relevant for the output node of type $l_p \in \mathbf{l}$; $\mathbf{l}$ is a set of output type nodes. In other words, each f_p is a *relational clique template* (Taskar, Abbeel, and Koller 2002) based on the types of input and output that it takes. However, g can be decomposed and learned in other ways to represent a more general scoring function than the one corresponding to the likelihood of an assignment.

In *Saul*, each relational factor f_p is declared as a classifier (Heckerman et al. 2000; Roth and Yih 2005; Sutton and McCallum 2005). In other words, a classifier is a *relational clique template*, with a set of features that are applied on a type of input and output nodes in the *model graph*. Each factor is a learning model defined declaratively as a Scala object using the following `Learnable` template (for example):

```
object ORG extends Learnable[Phrase] {
def label = entityType is "Organization"
def features=EntityFeatures}
```

This classifier corresponds to factor 1 in Figure 3-a and 3-b. Other factors can be defined in a similar way. The constraint factors are *Saul*'s first order logical expressions, shown as black boxes in Figure 3-b. The code of factor 5 in this figure was presented in Section 3.3. It bounds the labels of type ORG, PER and WorkFor to each other. Factor 4 can be declared as follows, using *Saul*'s `constraintOf` definition Scala template:

```
val SingleEntityLabel=constraintOf[Phrase]{
 x:Phrase=>{
 ((PER on x) is true)==>((ORG on x) is false
     ) &&&
```

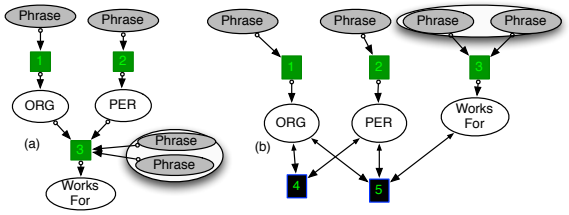

Figure 3: a) the FG of a pipeline model, b) the FG of a joint model in which the labels are related with constraint factors.

```
((ORG on x) is true)==>((PER on x) is false
    )}}
```

This constraint imposes the assignment of only one entity-mention type to each phrase. These declarations or a subset of them can be used in an *Saul* program to compose a spectrum of models from very local models up to more global (but decomposed) and fully joint models.

4.1 Model Composition

Now that we have discussed how each factor is expressed in a *Saul* program, we describe how these factors can be easily composed and chained in various configurations, resulting in different models. *Saul* allows a programmer to easily compose complex models using the relational factors and the constraint factors. The terminology used to describe the various models is taken from (Punyakanok et al. 2005), but our models are more general than those described there.

Local models (LO). Each *relational clique template* f_p can be defined as a local classifier such as the above `ORG` object using the *Saul*'s `Learnable` template, and trained independently[2]. In figure 3-b if we ignore the factors 4 and 5 we will have three independent local models. Training these models is simply done by adding the data, that comes form the reader in collections to the data graph and running a line such as, `ORG.learn()`, or test a classifier using `ORG.test()` for the classifier ORG, in this case. The collection of test data can be passed explicitly to the test function too. A trained model is a *named* object that returns a prediction for the relevant objects. of figure 3-b

Local learning and global prediction (L+I). In many real world situations, training a complex model can be decomposed into independently training a number of local models (Besag 1977; Heckerman et al. 2000; Roth and Yih 2005; Sutton and McCallum 2005; Samdani and Roth 2012), but it can gain an advantage from a joint, constrained, prediction using the trained local models. In *Saul* this requires the programmer to only declare how the outputs of decomposed models (target variables) are related to each other. For example, the three classifiers of the entities PER, ORG and WorkFor can be trained locally, by calling the $learn$ function for each. To predict globally, the programmer will write the following constraint conditional classifier:

[2]Setting specific algorithms' parameters can be done by the programmer, or automatically by *Saul*; these details are omitted.

```
object WorkForCCM extends
    ConstraintClassifier[Pairs]{
 def subjectTo= WorkFor }
```

This one line of code automatically recalls the factors/feature templates related to these target models in the WorkFor constraint, builds an appropriate objective function and optimizes it. Given this declaration, later in the program, the programmer can call `WorkForCCM(obj)` to make a joint prediction for an input object of type $Pair$ and its arguments.

Joint training. To use all the connections in the factor graph of figure 3-b, the same CCM object of the previous example can be invoked and used during training, in an inference-based training (IBT) setting for structured learning. Training joint models, as in the joint prediction above, only requires the programmer to define the way the outputs of the variables of interest are related to each other. This can be done, for example, via this learning model:

```
object WorkForJoint extends
    ConstraintLearner[Pairs]{
 def subjectTo= WorkFor }
```

Basically, the same objective that was created earlier for making predictions is created now during the training and the parameters of the all (simpler) learning models involved, (here, for PER, ORG, WorkFor) are updated jointly in an inference-based training fashion (Chang, Ratinov, and Roth 2007). The learning models that have been trained jointly, can used later by the programmer for making predictions in any arbitrary configuration.

Pipeline models. The most commonly used models in many applications including natural language processing tasks, are pipeline models. *Saul*'s local learning models can be used in a pipeline system in a straight forward way to support complex decisions based on local decisions made in earlier layers of the pipeline. A named model can be called by its name to assign a label to an input object. Thus, variables that were learned in the *Saul* program can be naturally used as features of other learning models in the same program. For example, we can call `PER.learn()` and `ORG.learn()` to train two independent models and then define the features for the WorkFor learner as follows:

```
val RelationFeatures = FeaturesOf[Pair]{
x:Pair=>{ORG(x.firstArg)::PER(x.secondArg)::
    lexForm(x.firstArg)::Nil}}
```

This feature declaration defines a list of features, two of which, ORG and PER, are *Saul* models themselves:

```
Object WorkForPipe extends Learnable[Pair]{
def label = retype is "Work-For"
def features=RelationFeatures}
```

This model is represented in the factor graph in Figure 3-b. This example also clarifies the semantics of the directions on the arrows of the *Saul* factor graphs. These arrows show the direction of the computations; having only one direction means that the model is decomposed and components are then pipelined. Defining complex pipeline models does not need any additional programming effort in *Saul*.

Approach	PER	ORG	LOC	WorkFor	LiveIn
LO	77.71	72.56	83.51	60.99	54.88
L+I	78.57	72.46	83.69	57.97	60.75
Pipeline	77.71	72.56	83.51	52.06	62.57
Pipeline+I	77.94	72.32	83.67	50.59	65.60
Joint (IBT)	78.88	71.15	81.60	69.00	69.48

Table 1: The F1 of various models architectures including local, global, pipeline models with different training and prediction configurations which are designed in *Saul*, 5-fold cross validation, CoNLL-04 dataset

4.2 Experimental Results

In order to exemplify the ease within which *Saul* allows one to develop expressive models, we designed a set of experiments, expanding on (Roth and Yih 2007). Table 1 shows the experimental results of running the learning and inference models described in Section 4.1 on the CoNLL-04 data for the *Entity-mention-Relation* (see Section 3). The applied features are designed in *Saul* based on the previous literature and inference includes the `WorkFor` constraint described in Section 3.3, a similar constraint for the `LiveIn` relation, the `SingleEntityLabel` constraint, defined in Section 4, which allows only one label per entity and a similar one for relations.

The goal of this experiment is not to show improved results, but rather exhibit the fact that a preliminary implementation of *Saul* is already functional and, most importantly, that minor variation to the *Saul* code, as shown above, already yield powerful results.

In addition to the models of Section 4.1, we experimented with another variation, Pipeline+I, that trains models in a pipeline fashion but makes a prediction using global inference. The results indicate that in this problem, particularly for predicting the relations, the joint training setting with global predictions is the best setting. The results show that for the entities, all settings are fairly similar, though joint training slightly decreases F1 of the entities due to a drop in recall. The joint training (IBT) shows its advantage in the prediction of the relationships, WorkFor and LiveIn, for which F1 has sharply increased compared to all other models.

5 Conclusion

We presented *Saul*[3], a new probabilistic programming language within the *Learning based programming* family. The language was designed to support, advance and simplify the development of AI systems. We described the key components of this object-functional programming language and argued that it allows a programmer to represent data and knowledge, to learn, name and manipulate named abstractions over relational data, support the incorporation of first order logical constraints and trainable components into the program, and provide a level of inference over trainable models to support a rich set of model chaining and composition. We showed some experimental results achieved by minimal coding in *Saul*, that could have taken significantly more effort in conventional programming languages. *Saul* is designed over an expressive data and modeling representation, as a high level and very easy to use language, thus promising to advance our thinking on how to better model and develop learning based programs in AI.

[3]http://cogcomp.cs.illinois.edu/page/software_view/Saul

6 Acknowledgments

This research was supported by grant 1U54GM114838 awarded by NIGMS through funds provided by the trans-NIH Big Data to Knowledge (BD2K) initiative (www.bd2k.nih.gov) and by DARPA under agreement number FA8750-13-2-0008. The content, views and conclusions contained herein is solely the responsibility of the authors and does not necessarily represent the official views of the NIH, DARPA nor the U.S. Government.

References

Besag, J. E. 1977. Efficiency of pseudolikelihood estimation for simple Gaussian fields. *Biometrika* 64(3):616–618.

Chang, M.; Ratinov, L.; and Roth, D. 2007. Guiding semi-supervision with constraint-driven learning. In *Proc. of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 280–287. Prague, Czech Republic: Association for Computational Linguistics.

Chang, M.; Ratinov, L.; and Roth, D. 2012. Structured learning with constrained conditional models. *Machine Learning* 88(3):399–431.

Collins, M. 2002. Discriminative training methods for hidden Markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the Conference on Empirical Methods for Natural Language Processing (EMNLP)*.

Cumby, C., and Roth, D. 2003. On kernel methods for relational learning. In *Proc. of the International Conference on Machine Learning (ICML)*, 107–114.

Cunningham, A.; Paluri, M.; and Dellaert, F. 2010. Ddfsam: Fully distributed slam using constrained factor graphs. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, 3025–3030.

Domingos, P., and Richardson, M. 2004. Markov logic: A unifying framework for statistical relational learning. In *ICML'04 Workshop on Statistical Relational Learning and its Connections to Other Fields*, 49–54.

Fischer, B., and Schumann, J. 2003. Autobayes: a system for generating data analysis programs from statistical models. *Journal of Functional Programming* 13(3):483–508.

Frasconi, P.; Costa, F.; De Raedt, L.; and De Grave, K. 2012. kLog: a language for logical and relational learning with kernels. *CoRR* abs/1205.3981.

Ganchev, K.; Graça, J.; Gillenwater, J.; and Taskar, B. 2010. Posterior regularization for structured latent variable models. *Journal of Machine Learning Research*.

Ghrab, A.; Skhiri, S.; Jouili, S.; and Zimnyi, E. 2013. An analytics-aware conceptual model for evolving graphs. In *Data Warehousing and Knowledge Discovery*, volume 8057 of *LNCS*. Springer Berlin Heidelberg. 1–12.

Gilks, W. R.; Thomas, A.; and Spiegelhalter, D. J. 1994. A language and program for complex bayesian modelling. *The Statistician* 43(1):169–177.

Goodman, N. D.; Mansinghka, V. K.; Roy, D. M.; Bonawitz, K.; and Tenenbaum, J. B. 2008. Church: A language for generative models. In *UAI*, 220–229.

Heckerman, D.; Chickering, D. M.; Meek, C.; Rounthwaite, R.; and Kadie, C. M. 2000. Dependency networks for inference, collaborative filtering, and data visualization. *Journal of Machine Learning Research* 1:49–75.

Huang, S. S.; Green, T. J.; and Loo, B. T. 2011. Datalog and emerging applications: An interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, 1213–1216. ACM.

Kschischang, F.; Frey, B.; and Loeliger, H. 2001. Factor graphs and the sum-product algorithm. *Information Theory, IEEE Transactions on* 47(2):498–519.

Mansinghka, V. K.; Selsam, D.; and Perov, Y. N. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR* abs/1404.0099.

Martins, A. F. T. 2012. *The geometry of constrained structured prediction: applications to inference and learning of natural language syntax*. Ph.D. Dissertation, Universidade Tecnica de Lisboa (Instituto Superior Tecnico) and Carnegie Mellon University.

Mateescu, R., and Dechter, R. 2008. Mixed deterministic and probabilistic networks. *Annals of Mathematics and Artificial Intelligence* 54(1-3):3–51.

McCallum, A.; Schultz, K.; and Singh, S. 2009. FACTORIE: Probabilistic Programming via Imperatively Defined Factor Graphs. In *The Conference on Advances in Neural Information Processing Systems (NIPS)*.

Odersky, M.; Spoon, L.; and Venners, B. 2008. *Programming in Scala: A Comprehensive Step-by-step Guide*. USA: Artima Incorporation, 1st edition.

Pfeffer, A. 2009. Figaro: An object-oriented probabilistic programming language. Technical report, Charles River Analytics.

Punyakanok, V.; Roth, D.; Yih, W.; and Zimak, D. 2005. Learning and inference over constrained output. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1124–1129.

Raedt, L. D., and Kersting, K. 2004. Probabilistic logic learning.

Rizzolo, N., and Roth, D. 2010. Learning based java for rapid development of nlp systems. In *Proc. of the International Conference on Language Resources and Evaluation (LREC)*.

Roth, D., and Yih, W. 2005. Integer linear programming inference for conditional random fields. In *Proc. of the International Conference on Machine Learning (ICML)*, 737–744.

Roth, D., and Yih, W. T. 2007. Global inference for entity and relation identification via a linear programming formulation. In Getoor, L., and Taskar, B., eds., *Introduction to Statistical Relational Learning*. MIT Press.

Roth, D. 2005. Learning based programming. *Innovations in Machine Learning: Theory and Applications*.

Samdani, R., and Roth, D. 2012. Efficient decomposed learning for structured prediction. In *Proc. of the International Conference on Machine Learning (ICML)*.

Samdani, R.; Chang, M.; and Roth, D. 2012. Unified expectation maximization. In *Proc. of the Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

Sato, T., and Kameya, Y. 1997. Prism: A language for symbolic-statistical modeling. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1330–1339.

Sontag, D. 2010. *Approximate Inference in Graphical Models using LP Relaxations*. Ph.D. Dissertation, Massachusetts Institute of Technology.

Sutton, C. A., and McCallum, A. 2005. Piecewise training for undirected models. In *UAI*, 568–575. AUAI Press.

Taskar, B.; Abbeel, P.; and Koller, D. 2002. Discriminative probabilistic models for relational data. In *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, 485–492.