WOLFE: An NLP-friendly Declarative Machine Learning Stack

Sameer Singh[†] **Tim Rocktäschel**[*] **Luke Hewitt**[*] **Jason Naradowsky**[*] **Sebastian Riedel**[*]
[†]University of Washington
Seattle WA
sameer@cs.washington.edu
[*]University College London
London UK
{t.rocktaschel,luke.hewitt.10,j.narad,s.riedel}@cs.ucl.ac.uk

Abstract

Developing machine learning algorithms for natural language processing (NLP) applications is inherently an iterative process, involving a continuous refinement of the choice of model, engineering of features, selection of inference algorithms, search for the right hyper-parameters, and error analysis. Existing probabilistic program languages (PPLs) only provide partial solutions; most of them do not support commonly used models such as matrix factorization or neural networks, and do not facilitate interactive and iterative programming that is crucial for rapid development of these models.

In this demo we introduce WOLFE, a stack designed to facilitate the development of NLP applications: (1) the WOLFE *language* allows the user to concisely define complex models, enabling easy modification and extension, (2) the WOLFE *interpreter* transforms declarative machine learning code into automatically differentiable terms or, where applicable, into factor graphs that allow for complex models to be applied to real-world applications, and (3) the WOLFE *IDE* provides a number of different visual and interactive elements, allowing intuitive exploration and editing of the data representations, the underlying graphical models, and the execution of the inference algorithms.

1 Introduction

Machine learning has become a critical component of practical NLP systems, however designing and training an appropriate, accurate model is an iterative and time-consuming process for a number of reasons. First, initial intuitions that inform model design (such as which features to use) are often inaccurate, requiring incremental model tweaking based on performance. Even if the model is accurate, the final performance depends quite critically on the choice of the algorithms and their hyper-parameters. Further, bugs that are introduced by the user may not even be reflected directly in the performance (such as a feature computation bug may not degrade performance). All these concerns are further compounded due to the variety of approaches commonly used in NLP, such as conditional random fields (Sutton and McCallum, 2007), Markov random networks (Poon and Domingos, 2007), Bayesian networks (Haghighi and Klein, 2010), matrix factorization (Riedel et al., 2013), and Deep learning (Socher et al., 2013).

Probabilistic programming languages (PPLs), by closing the gap between traditional programming and probabilistic modeling, go a long way in aiding quick design and modification of expressive models[1]. However, creating accurate machine learning models using these languages remains challenging. Of the probabilistic programming languages that exist today, no language can easily express the variety of models used in NLP, focusing instead on a restricted set of modeling paradigms, for example, Markov logic networks can be models by Alchemy (Richardson and Domingos, 2006), Bayesian generative networks by Church (Goodman et al., 2008), undirected graphical models by Factorie (McCallum et al., 2009), and so on. Further, these toolkits are only restricted to model design and inference execution, and do not provide the appropriate debugging and interactive

[1]For a comprehensive list of PPLs, see `http://probabilistic-programming.org/`.

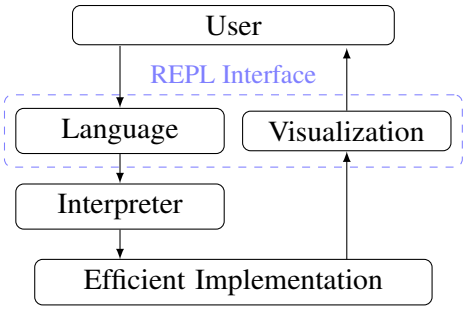


Figure 1: Overview of the WOLFE Stack.

```
//per cell scores
def F(t: Thetas.Term)(cell: Cells.Term) =
  t.rels(cell.rel) dot t.pairs(cell.pair)
def E(t: Thetas.Term)(cell: Cells.Term) =
  t.slots1(cell.relSlot1) dot t.arg1s(cell.arg1)) +
  t.slots2(cell.relSlot2) dot t.arg2s(cell.arg2))
def FE(t: Thetas.Term)(cell: Cells.Term) =
  F(t)(cell) + E(t)(cell)
def Tucker2(t: Thetas.Term)(cell: Cells.Term) =
  (t.relMatrices(cell.rel) * t.arg1s(cell.arg1)) dot t.arg2s(cell.arg2)
def TransE(t: Thetas.Term)(cell: Cells.Term) =
  l2(t.arg1s(cell.arg1) + t.rels(cell.rel), t.arg2s(cell.arg2))
//objectives
def bpr(pos: DoubleTerm, neg: DoubleTerm) =
  log(sigm(pos - neg))
def ll(pos: DoubleTerm, neg: DoubleTerm) =
  log(sigm(pos)) - log(sigm(neg))
//Bayesian Personalized Ranking with FE model
def bprFEObjective(t: Thetas.Term)(pos: Cells.Term, neg: Cells.Term) =
  bpr(FE(t)(pos), FE(t)(neg))
```

Figure 2: Implementation of various matrix and tensor factorization models in WOLFE.

visualization tools required for developing such models in practice. Due to these concerns, probabilistic programming has not found significant adoption in natural language processing, and application of machine learning to NLP still consists either of arduously designing, debugging, and iterating over a variety of models, or more commonly, giving up and using the first model that is "good enough".

In this demo, we introduce our probabilistic programming toolkit WOLFE (Riedel et al., 2014) that aids in the iterative design of machine learning models for NLP applications. The underlying probabilistic programming language can be used to concisely express a wide range of models, including undirected graphical models, matrix factorization, Bayesian networks, neural networks, and further, its modular nature allows combinations of these modeling paradigms. We additionally present an easy-to-use IDE for the interactive designing of NLP models, consisting of an interactive and visual presentation of structured data, graphical models, and inference execution. Using the WOLFE language and IDE can thus enable the users to quickly create, debug, and iterate on complex models and inference.

2 Overview of the WOLFE Stack

The overall goal of the demo will be to guide users in creating complex graphical models using an easy-to-use mathematical language for defining models, and in performing learning and inference for the created model using an IDE. Figure 1 summarizes the overview of the WOLFE stack, consisting of the language and the visualization that form the user-facing interface, with the interpreter and efficient learning and inference engine as the back-end.

2.1 Declarative Modeling Language

Existing PPLs primarily focus on a single representation for the probabilistic models, and either do not support, or provide only inefficient implementations for other kinds of machine learning models. Thus a practitioner either has to write her own customized implementation of the models she is trying to explore, or decide apriori on the family of models she will be restricted to; both undesirable options. Instead, we introduce a probabilistic programming language that is universal in its expression of models, yet allows for efficient implementations of these models.

The design of the WOLFE language is inspired by the observation that most machine learning algorithms can be formulated in terms of scalar functions (such as distributions and objectives/losses), search spaces (such as the universe of possible labels) and a small set of mathematical operations such as maximization, summation and expectations that operate on these functions and spaces. Using this insight, a program in WOLFE consists of a declarative description of the machine learning algorithm in terms of implementations of these scalar functions, definitions of the search spaces, and the use of appropriate operators on these. For example, named-entity recognition tagging using conditional random fields consists of a scalar function that defines the model *score* using a dot product between the parameters and the *sum* of node and edge features, while inference using this model involves finding the label sequence that has the *maximum* model score over all label sequences.

The focus on scalar functions as building blocks allows for rapid prototyping of a large range of ma-

chine learning models. For instance, there exist a variety of matrix and tensor factorization methods for knowledge base population that have a succinct, unified mathematical formulation (Nickel et al., 2015). In WOLFE these models can be easily implemented with a few lines of code. See Figure 2 for examples of a Tucker2 decomposition, TransE (Bordes et al., 2013), and Riedel et al. (2013)'s feature model (F), entity model (E), and combination of the two (FE), either based on a log likelihood or Bayesian Personalized Ranking (Rendle et al., 2009) objective.

2.2 Interpreter, and Efficient Implementations

In WOLFE users write models using a domain-specific-language that supports a wide range of mathematical expressions. The WOLFE *interpreter* then evaluates these expressions. This is non-trivial as expressions usually contain operators such as the *argmax* functions which are, in general, intractable to compute. For efficient evaluation of WOLFE programs our interpreter *compiles* WOLFE expressions into representations that enable efficient computation in many cases. For example, for terms that involve maximization over continuous search spaces WOLFE generates a computation tree that supports efficient forward and back-propagation for automatic differentiation. Likewise, when maximizing over discrete search spaces, WOLFE constructs *factor graphs* that support efficient message passing algorithm such as Max-Product or Dual Decomposition. Crucially, due to the compositional nature of WOLFE, discrete and continuous optimization problems can be nested to support a rich class of *structured prediction* objectives. In such cases the interpreter constructs nested computational structures, such as a factor graph within a back-propagation graph.

2.3 Visual and Interactive IDE

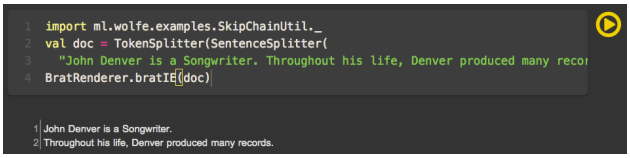
In this demonstration, we present an integrated developing, debugging and visualization toolkit for machine learning for NLP. The IDE is based on the read-eval-print loop (REPL) to allow quick iterations of writing and debugging, and consists of the following elements: (1) Editor (read): Users define the model and inference in the declarative, math-like language described in Section 2.1 using a syntax highlighted code editor. (2) Build Automation (eval): The use of the interpreter as described in the previous section to provide efficient code that is executed. (3) Debugging/Visualization (print): Our tool presents the underlying factor graph as an interactive UI element that supports clicking, drag and drop, hover, etc. to explore the structure and the factors of the model. We visualize the results of inference in a graphical manner that adapts to the type of the result (bar charts for simple distributions, shaded maps for matrix-like objects, circles/arrows for NLP data types, etc.). For further fine-grained debugging, we can also surface intermediate results from inference, for example, visualizing the messages in belief propagation for each edge in the factor graph.

3 Demo Outline

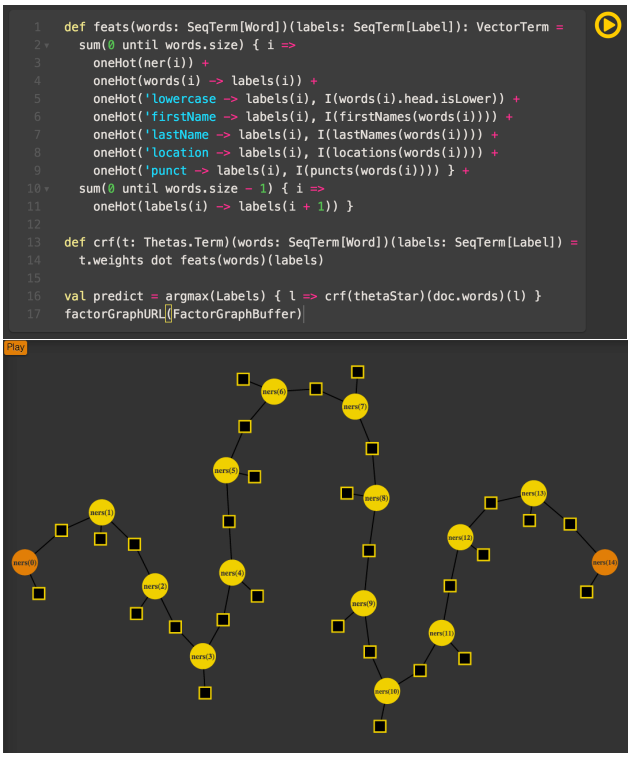
The overall objective of the demo is for users to design, debug, and modify a machine learning model for an NLP application, starting from scratch. The demo takes the user through all the steps of loading data, creating an initial model, observing the output errors, modifying the model accordingly, and rerunning to see the errors fixed: the complete set of steps often involved in real-life application of ML for NLP. We provide pre-built functions for the menial tasks, such as data loading and feature computation functions, leaving the more interesting aspects of model design to the user. Further, we include an "open-ended" option for interested users to develop arbitrary models. Based on their interest or area of expertise, the user has an option of investigating any (or all) of the following applications: (1) sequence tagging using CRFs, (2) relational learning using MLNs, (3) matrix factorization for relation extraction, and (4) dependency parsing (for advanced users). Each of these are similar in the overall "script", differing in the data, models, and inference algorithms used; we describe the steps of the demo using the CRF example. All of the demo applications are available online at `http://wolfe.ml/demos/nlp`.

1. The first step of the demo allows the user to read as input a standard dataset of the task, and visualize instances in an easy-to-read manner. In Figure 3a for example, we show two sentences read for the purpose of sequence tagging.

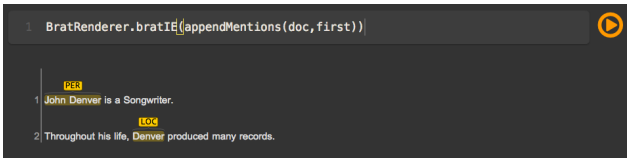
2. The user then defines an initial model for the task, which is visualized as a factor graph for



(a) Data Loading



(b) Initial Model

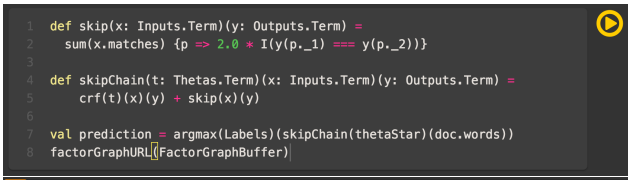


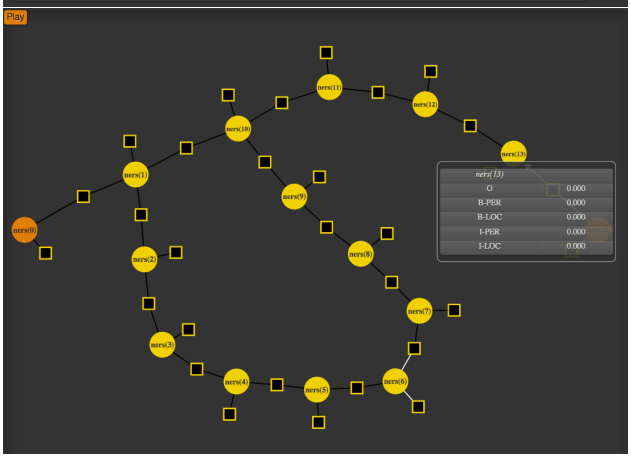
(c) Error in Prediction

Figure 3: **Model Creation and Evaluation:** An example instance of the demo showing the *creation* steps, including the loading and visualization of the sentences, designing and presentation of a linear chain CRF, and Viterbi decoding for the sentences.

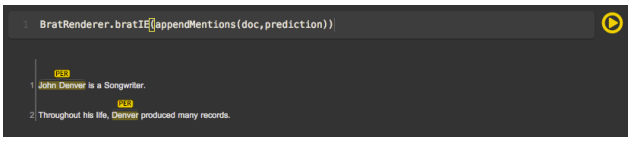
the purpose of debugging the model definition. The initial model for sequence tagging is a simple linear chain, defined and visualized for a sentence in Figure 3b.

3. The user writes the declarative definition of inference, and makes predictions of the input data. The predictions are appropriately visualized, allowing the user to detect mistakes (for example, the incorrect NER tag of location to "Denver" in Figure 3c).





(a) Modify the Model (add skip edge)



(b) Fixed Prediction

Figure 4: **Debugging Loop:** The remaining steps of the iterative development, consisting of modification of the model to fix the error from Figure 3c by adding a skip-factor to the original model, and confirming the inference in the skip-chain model results in the correct prediction.

4. The user then modifies the model (adding a skip-factor in Figure 4a) that will likely correct the mistake. The modified model is then visualized to confirm it is correct. (Optionally, the user can, at any point, visualize the execution of the inference to confirm the modifications as well, for example Figure 4a shows the state of messages in belief propagation.)

5. On the execution of the model, the user confirms that the original error has been fixed, for example the skip factor allows the correct tag of person for "Denver" in Figure 4b.

4 Conclusions

This demo describes WOLFE, a language, interpreter, and an IDE for easy, iterative development of complex machine learning models for NLP applications. The language allows concise definition of the models and inference by using universal, mathematical syntax. The interpreter performs program analysis on the user code to automatically generate efficient low-level code. The easy-to-use IDE allows the user to iteratively write and execute such programs, but most importantly supports intuitive visualizations of structured data, models, and inference to enable users to understand and debug their code. The demo thus allows a user to design, debug, evaluate, and modify complex machine learning models for a variety of NLP applications.

Acknowledgments

We would like to thank Larysa Visengeriyeva, Jan Noessner, and Vivek Srikumar for contributions to early versions of WOLFE. This work was supported in part by Microsoft Research through its PhD Scholarship Programme, an Allen Distinguished Investigator Award, a Marie Curie Career Integration Grant, and in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. In *Advances in Neural Information Processing Systems*, pages 2787–2795.

Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: a language for generative models. In *Uncertainty in Artificial Intelligence (UAI)*.

Aria Haghighi and Dan Klein. 2010. Coreference resolution in a modular, entity-centered model. In *North American Chapter of the Association for Computational Linguistics - Human Language Technologies (NAACL HLT)*, pages 385–393.

Andrew McCallum, Karl Schultz, and Sameer Singh. 2009. FACTORIE: Probabilistic programming via imperatively defined factor graphs. In *Neural Information Processing Systems (NIPS)*.

Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. 2015. A review of relational machine learning for knowledge graphs: From multi-relational link prediction to automated knowledge graph construction. *arXiv preprint arXiv:1503.00759*.

Hoifung Poon and Pedro Domingos. 2007. Joint inference in information extraction. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI '07)*, pages 913–918.

Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. 2009. BPR: Bayesian personalized ranking from implicit feedback. In *Uncertainty in Artificial Intelligence (UAI)*.

Matthew Richardson and Pedro Domingos. 2006. Markov logic networks. *Machine Learning*, 62(1-2):107–136.

Sebastian Riedel, Limin Yao, Benjamin M. Marlin, and Andrew McCallum. 2013. Relation extraction with matrix factorization and universal schemas. In *Joint Human Language Technology Conference/Annual Meeting of the North American Chapter of the Association for Computational Linguistics (HLT-NAACL '13)*, June.

Sebastian Riedel, Sameer Singh, Vivek Srikumar, Tim Rocktaschel, Larysa Visengeriyeva, and Jan Noessner. 2014. Wolfe: Strength reduction and approximate programming for probabilistic programming. In *International Workshop on Statistical Relational AI (StarAI)*.

Richard Socher, Alex Perelygin, Jean Y Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Charles Sutton and Andrew McCallum. 2007. An introduction to conditional random fields for relational learning. In *Introduction to Statistical Relational Learning*.