srlearn: A Python Library for Gradient-Boosted Statistical Relational Models

Alexander L. Hayes
ProHealth Lab
Indiana University Bloomington
hayesall@iu.edu

Abstract

We present srlearn, a Python library for boosted statistical relational models. We adapt the scikit-learn interface to this setting and provide examples for how this can be used to express learning and inference problems.

Introduction

Traditional machine learning systems have generally been built as command line applications or as graphical user interfaces (Hall et al. 2009). Both have advantages, but offer fewer solutions when data acquisition, preprocessing, and model development must occur together. Systems such as scikit-learn, TensorFlow, Pyro, and PyTorch solve this problem by embedding data cleaning and model development as steps within general-purpose languages (Pedregosa et al. 2011; Abadi et al. 2016; Bingham et al. 2019; Paszke et al. 2017). This has also made open source implementations available to both experts and non-experts, providing each the tools to develop models.

Statistical Relational Learning (SRL) models have unique concerns, often inherited from underlying logical systems. This requires a data representation beyond fixed-length feature vectors, and a language bias to constrain the hypothesis space. By embedding both operations in a manner that machine learning researchers and practitioners may already be familiar with, we hope to speed up development time for SRL practitioners, and provide a more user-friendly experience for data scientists and the wider machine learning community—many of whom are not experts in SRL.

API Design in Machine Learning

The scikit-learn package (Pedregosa et al. 2011) has been influential for its consistent application programming interface (API) across a variety of machine learning models. In scikit-learn, an algorithm type (e.g. linear support vector classification) is implemented as a class. An estimator is an instance of an algorithm type whose hyperparameters have been set upon object construction. A predictor is an estimator that has been fit (i.e. trained) to a dataset, and is ready to predict (e.g. classify) new data instances. Although the estimation and prediction functions are logically distinguished in two separate protocols, it is generally a single class that implements both a learning algorithm and the model for applying the parameters to new data.

The aforementioned standard approach thus comprises configuring model hyperparameters, fitting training data, and predicting test data. However, scikit-learn has since developed into a full-fledged ecosystem that also services related functions in the modeling workflow: model selection, hyperparameter tuning, and model validation. Furthermore, multiple offshoots of scikit-learn have emerged to tackle more specialized challenges, including imbalanced datasets (Lemaitre, Nogueira, and Aridas 2017), generalized linear models (Blondel and Pedregosa 2016), and metric learning (de Vazelhes et al. 2019).

These offshoots still fit within the framework of only requiring inputs, outputs, and hyperparameters. But while this has been influential while designing APIs for classic statistical learning methods, it could also be a limitation when extending the API to incorporate the specific needs of models from other learning paradigms (Buitinck et al. 2013). Learning within frameworks designed for graphical models, active learning, or reinforcement learning typically requires the user to specify something outside of inputs and outputs. Graphical models require statistical independence assumptions to either be set by hand or inferred via structure learning. Active learning requires human intervention. Reinforcement learning needs a simulator.

Extending the API to handle new paradigms should ideally meet two goals: (1) expressiveness to describe what the user wants to achieve, and (2) complimentarity to what users are already familiar with.

srlearn

We propose a simple extension to the scikit-learn API for representing statistical relational models while staying close to our two goals. Specifically we incorporate a Background object and a Database object.

The Background object incorporates knowledge about relationships to constrain model search space, currently expressed in the language of “modes” (Srinivasan 2000). This is then provided to the statistical relational estimator.

The Database object generalizes inputs as being composed of positive examples, negative examples, and facts about the world—each expressed as Prolog predicates.
from srlearn.rdn import BoostedRDN
from srlearn import Background
from srlearn import example_data

bk = Background(
    modes=[
        "friends(+person,-person).",
        "friends(-person,+person).",
        "cancer(+person).",
        "smokes(+person).",
    ],
    use_std_logic_variables=True,
)

clf = BoostedRDN(
    background=bk,
    target="cancer",
)

clf.fit(example_data.train)
clf.predict_proba(example_data.test)

def learn_and_inference_on_senses(databases, domain):
    for database in databases:
        print(f"Learning and inference on {database}.")

    # Example code for learning and inference

Figure 1: Learning and inference on toy databases for a
smokes-friends-cancer domain. example_data.train
and example_data.test are Database objects.

A statistical relational estimator may then be described
in the same language as a standard scikit-learn estimator
that also incorporates background knowledge to constrain
the hypothesis space, and learn on a database of predicates
rather than vectors. Currently we have focused on incorpo-
rating methods from BOOSTSRL, a Java tool for learning
relational dependency networks and Markov logic networks
via gradient boosting (Natarajan et al. 2018). Figure 1 shows
how modules from srlearn can be put together to learn on
a built-in data set, then make predictions on a test database.

Development

srlearn is developed as an open source project on
GitHub and is distributed under the terms of the GNU Gen-
eral Public License v3.0 (GPL-3.0). Within the code, we
have taken several measures to aid its maintenance. This
includes formatting conventions (black, pycodestyle),
linting (pylint), and running the main branch and all pull
requests through static analysis (igtm).

We also maintain a test suite to compare each build
against previous versions. Tests run on Linux and Windows
machines each time the code is pushed to GitHub; metrics
track (1) that all tests pass, and (2) that a sufficient code
coverage is maintained. At the time of writing, all tests pass
(results meet expectations), and code coverage is at 100%
(every line of code is visited during testing). Perfect cover-
age often grows unrealistic as projects grow, but we aim to
keep it above 90% while passing all tests.

Finally, we maintain documentation to help acclimate
users to the code base; this includes user guides with nar-
avative documentation and examples motivating specific tasks.

Experiments

We expect a small overhead due to the Python interpreter and
data structures at runtime; but since the core algorithms bor-
row heavily BOOSTSRL’s Java implementations, we expect
this overhead to be negligible compared to the time spent
during learning. To evaluate this, we compare runtime in
seconds on standard benchmark data using the BOOSTSRL
command line interface and the srlearn API. We hold
the modes and hyperparameters fixed, then record the time
taken while learning a boosted RDN with the srlearn and
BOOSTSRL systems on three benchmark data sets. Table 1
shows the time averaged over ten runs of each, which we use
to conclude that the time differences are indeed negligible.\(^3\)

<table>
<thead>
<tr>
<th>Database</th>
<th>srlearn (Python/Java)</th>
<th>BOOSTSRL (Java/Shell)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebKB</td>
<td>4.2 (0.5)</td>
<td>4.9 (0.3)</td>
</tr>
<tr>
<td>IMDB</td>
<td>10.2 (1.1)</td>
<td>13.0 (1.3)</td>
</tr>
<tr>
<td>UWCE</td>
<td>17.5 (1.4)</td>
<td>18.3 (1.7)</td>
</tr>
</tbody>
</table>

Table 1: Seconds elapsed while learning a Boosted RDN
on three benchmark data sets. Mean (and standard devia-
tion) are calculated over ten runs. Small differences in times
may also be influenced by small differences in measurement:
epoch time (Bash) and perf.time (Python).

Conclusion

It is possible that the imperative programming style here
is not ideal for SRL models—the underlying logic formal-
ism is often better expressed through declarative approaches,
which have further been suggested as ways to unify software
development with learning systems (Kordjamshidi, Roth,
and Kersting 2018).

Nonetheless, many learning frameworks have been built
around the Python ecosystem. Programming abstractions
such as the one presented here may therefore be an im-
portant step toward bridging the gap between SRL and neural
approaches by providing developers the tools to more easily
work with both in a common environment.

In the future, we intend on extending the modeling lan-
guage with more methods that have been successful within
SRL—such as learning with advice (Odom and Natarajan
2018), incorporating a relational database for learning and
inference (Malec et al. 2017), and incorporating SRL meth-
ods such as Probabilistic Soft Logic (Bach et al. 2015) or
Conditional Random Fields (Sutton and McCallum 2007).

Acknowledgements

ALH is supported through Indiana University’s “Precision
Health Initiative” (PHI) Grand Challenge. ALH would like
to thank Sriraam Natarajan, Travis LaGrone, and members
of the StARLinG Lab at the University of Texas at Dallas.

\(^1\)https://github.com/hayesall/srlearn/
\(^2\)https://srlearn.readthedocs.io
\(^3\)Scripts for reproducing this table is available on GitHub:
https://github.com/hayesall/srlearn-StarAI-2020-workshop
References


