rfgb Documentation

Release 0.3.2.dev

Alexander L. Hayes (@hayesall)

Aug 14, 2019

Overview:

CHAPTER 1

Getting Started

Installation and a few motivating examples.

1.1 Development Environment Setup

The first step is to get Python running on your machine (skip to the next step if you've already done this).

1.1.1 Linux (yum/dnf)

```
$ sudo yum update
$ sudo yum install python
```

1.1.2 Linux (apt-get)

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install python
```

1.1.3 Windows

Download Python from python.org or anaconda.com.

A fairly in-depth guide is available as part of the Conda documentation.

1.2 Getting Started

1.2.1 Installation

`rfgb` can be installed via the following methods:

1. Stable builds on PyPi

```
pip install rfgb
```

2. Development builds on GitHub

```
pip install git+git://github.com/hayesall/rfgb.git
```

3. Bleeding-edge development builds on the GitHub Development Branch

```
pip install git+git://github.com/hayesall/rfgb.git@development
```

1.2.2 Background

The main function of `rfgb` is to learn relational dependency networks[1] via gradient tree boosting, based on Natarajan et al. "Boosting Relational Dependency Networks"[2].

This algorithm is implemented as the `__main__` method for the `rfgb` package.

```
# rfgb.__main__

from .boosting import updateGradients
from .tree import node
from .utils import Utils

# ... class Arguments:

parameters = Arguments().args

for target in parameters.target:

    # Read the training data
    trainData = Utils.readTrainingData(target,
                                       path=parameters.train,
                                       regression=parameters.reg,
                                       advice=parameters.expAdvice)

    # Initialize an empty list for the trees.
    trees = []

    # Learn each tree and update the gradients.
```

(continues on next page)

[1] Jennifer Neville and David Jensen, "Relational Dependency Networks." *Journal of Machine Learning Research (JMLR)*, 2007.

[2] Siraam Natarajan, Tushar Khot, Kristian Kersting, Bernd Gutmann, and Jude Shavlik, "Boosting Relational Dependency Networks. *International Conference on Inductive Logic Programming (ILP)*, 2010.

(continued from previous page)

```
    for i in range(parameters.trees):

        node.setMaxDepth(2)
        node.learnTree(trainData)
        trees.append(node.learnedDecisionTree)
        updateGradients(trainData, trees)
```

1.2.3 File Structure

File structure follows the structure used by BoostSRL.

Training directories and testing directories are currently used and flat files are read from, converted to a relational internal representation, and then the relationships may be reasoned about.

1.2.4 References

1.3 Running rfgb

1.3.1 Reasoning about the World

Object and their relationships are a natural way to think about the world. In this example, we have some facts about the world which we want to learn from. More specifically, we have a table of people and their relationships.

Name	Gender	Child	Sibling
James	Male	[Harry]	•
Lily	Female	[Harry]	Petunia
Harry	Male	•	•
Arthur	Male	[Ron, Fred]	•
Molly	Female	[Ron, Fred]	•
Ron	Male	•	[Fred]
Fred	Male	•	[Ron]

Assume that the goal is to learn `father(Y,X)`. We want to learn logical rules representing that domain object X is the father of Y (both of which are people in this case), given that you know information about their gender, children, and siblings.

1.3.2 From Tables to First-order Predicate Logic

Once we have a high-level idea of what these relationships look like, the next step is to convert this into predicate logic format. This format is standard for most Prolog-based systems.

A few assumptions we will make about our data:

1. 'Name' is an identifier.
2. 'Gender' is male or female in this case, so we can make it a true/false value.
3. 'Child' and 'Sibling' are binary relationships encoding a relationship between two people (e.g. `childof(lily, harry)` denotes that 'harry' is the childof 'lily').

The *target* we want to learn is `father(x,y)`. To learn this rule, `rfgb` learns a decision tree that most effectively splits the positive and negative examples. This example is fairly small so a small number of trees should suffice, but for more complicated problem more may be needed to learn a robust model.

Positive Examples:

```
father(harrypotter,jamespotter).
father(ginnyweasley,arthurweasley).
father(ronweasley,arthurweasley).
father(fredweasley,arthurweasley).
...
```

Negative Examples:

```
father(harrypotter,mollyweasley).
father(georgeweasley,jamespotter).
father(harrypotter,arthurweasley).
father(harrypotter,lilypotter).
father(ginnyweasley,harrypotter).
father(mollyweasley,arthurweasley).
father(fredweasley,georgeweasley).
father(georgeweasley,fredweasley).
father(harrypotter,ronweasley).
father(georgeweasley,harrypotter).
father(mollyweasley,lilypotter).
...
```

Facts:

```
male(jamespotter).
male(harrypotter).
male(arthurweasley).
male(ronweasley).
male(fredweasley).
male(georgeweasley).
siblingof(ronweasley,fredweasley).
siblingof(ronweasley,georgeweasley).
siblingof(ronweasley,ginnyweasley).
siblingof(fredweasley,ronweasley).
siblingof(fredweasley,georgeweasley).
siblingof(fredweasley,ginnyweasley).
siblingof(georgeweasley,ronweasley).
siblingof(georgeweasley,fredweasley).
siblingof(georgeweasley,ginnyweasley).
siblingof(ginnyweasley,ronweasley).
```

(continues on next page)

(continued from previous page)

```
siblingof(ginnyweasley,fredweasley).
siblingof(ginnyweasley,georgeweasley).
childof(jamespotter,harrypotter).
childof(lilypotter,harrypotter).
childof(arthurweasley,ronweasley).
childof(mollyweasley,ronweasley).
childof(arthurweasley,fredweasley).
childof(mollyweasley,fredweasley).
childof(arthurweasley,georgeweasley).
childof(mollyweasley,georgeweasley).
childof(arthurweasley,ginnyweasley).
childof(mollyweasley,ginnyweasley).
...
```

1.3.3 Training a Model

There is one more piece we still need: background knowledge about the world.

```
// Parameters
setParam: maxTreeDepth=3.
setParam: nodeSize=1.
setParam: numOfClauses=8.

// Modes
mode: male(+name).
mode: childof(+name,+name).
mode: siblingof(+name,-name).
mode: father(+name,+name).
```

Begin training:

```
python -m rfgb --help
```

CHAPTER 2

Development

Comments on developing `rfgb` further.

2.1 Contributing

From the BoostSRL Contributing Guidelines:

"Our goal is to push the boundaries of machine learning and statistical relational learning through open development and explainable approaches to decision making in both learning and inference. We believe that these are some of the best ways to create trustworthy systems that people can learn from and interract with in their daily lives."

The goal in this project is to match and eventually extend beyond BoostSRL (the Java version of the codebase), contributions which further this are welcome.

2.1.1 Code of Conduct

We adopt the Contributor Covenant Code of Conduct

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at alexander.hayes@utdallas.edu. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate for the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of the incident.

2.1.2 Development Cheat-Sheet

1. **Fork and clone the source from GitHub**

```
git clone https://github.com/hayesall/rfgb.git
```

2. **Building local copy of documentation**

   We use Sphinx autodoc with a combination of inline docstrings and reStructuredText for documenting this project. Pull requests and further updates should include appropriate documentation.

   A local copy of the documentation may be built from the Makefile:

```
cd docs
make html
xdg-open build/html/index.html
```

3. **Running the unit tests**

   `rfgb/tests/` contains a suite of unit tests, these can be ran via the following:

```
python rfgb/tests/tests.py
```

Note: As of 0.2.0, these should be ran from the base of the repository due to their import structure.

2.2 Unit Tests

The main testing module for `rfgb` must be ran from the base of the project repository.

For example:

```
python rfgb/tests/tests.py
```

Verbosity may be explicitly set by passing an integer with the `-v` flag. The value will be passed into unittest.TextTestRunner, so integers higher than 1 will lead to more verbose outputs.

```
python rfgb/tests/tests.py -v 2
```

2.2.1 Testing Individual Modules

Individual modules may be tested with unittest via the command line.

```
python -m unittest rfgb/tests/rfgbtests/test_Utils.py
.......
----------------------------------------------------
Ran 7 tests in 0.005s

OK
```

CHAPTER 3

Commandline

3.1 Workflow

CLI interface for performing learning and inference with different types of statistical relational learning methods, and managing these learned models for particular data sets.

Some of the ideas built here are shamelessly inspired by Git, so the workflows for using the commandline interface to `rfgb` should hopefully feel somewhat familiar to those familiar with version control.

```
$ pip install rfgb
$ rfgb --help
usage: rfgb [-h] [-V] {init,learn,infer} ...

rfgb: Relational Functional Gradient Boosting is a gradient-boosting
approach to learning statistical relational models.

optional arguments:
  -h, --help          show this help message and exit
  -V, --version       show version number and exit

rfgb Subcommands:
  Commands and subcommands for rfgb.

  {init,learn,infer}  $ rfgb --help
    init              Initialize a .rfgb directory.
    learn             Learn various SRL models.
    infer             Infer with various SRL models.
```

Assuming you start with a training and test set (we'll talk about those later), you can initialize a place where your models and meta-data will be stored.

```
$ rfgb init
```

This creates a `.rfgb` directory containing a `models` directory.

3.2 Data

As data scientists, a great deal of time is often spent getting data into a particular format. It is overly-ambitious to claim that we have *solved* this problem, but we try to reduce the time spent cleaning data.

The format we use is similar to *Prolog*, but with a clear distinction between *data* and *programs*.

Machine Learning is often described as learning a function over a vector x such that we can learn a target value y.

$$f(x) = P(y|x)$$

3.2.1 Defining Terms

The terms we invoke to describe these functions are **Positives**, **Negatives**, **Facts**, and **Background Knowledge**.

- **Positive** examples are true (or correct) examples that we want to learn from.
- **Negative** examples are false (or incorrect), examples that we do not want to do.
- **Facts** are features we use to learn. We make the assumption that some combination of the facts can be used to distinguish between positives and negatives.
- **Background Knowledge** comes in many forms, but is a way to introduce more information to learn more effectively. If a classifier is learning to distinguish handwritten digits, extra negative examples might be created by rotating digits. *Background Knowledge* about this domain might involve **not** rotating "6" and "9", since they are identical when rotated.

Background Knowledge is often described as the "black magic" or "expert knowledge" in machine learning. Many of our methods are designed to effectively incorporate this kind of knowledge, and solicit it in a variety of ways.

3.2.2 Format

Positives, negatives, and facts are contained in `pos.txt`, `neg.txt`, and `facts.txt`. Some examples are contained in the `testDomains` directory at the base of this repository.

For example: `testDomains/HeartAttack/train/`:

pos.txt	neg.txt	facts.txt
ha(p1) ha(p6)	ha(p2) ha(p3) ha(p4) ha(p5) ha(p7) ha(p8) ha(p9) ha(p10)	chol(p1,high) race(p1,r1) chol(p2,medium) race(p2,r1) chol(p3,medium) race(p3,r1) chol(p4,medium) race(p4,r1) chol(p5,low) race(p5,r1) chol(p6,high) race(p6,r2) chol(p7,medium) race(p7,r2) chol(p8,medium) race(p8,r2) chol(p9,medium) race(p9,r2) chol(p10,low) race(p10,r2)

```
ha(person)
chol(+person,[low;medium;high])
race(+person,[r1;r2])
```

Positve

The latter are inspired by the FOIL method and paper.

CHAPTER 4

rfgb Submodules

Submodules found in `rfgb`.

4.1 rfgb.boosting module

Core methods for performing learning and inference, such as computing gradients, updating gradients, and performing inference.

4.1.1 Documentation

`rfgb.boosting.`**`computeAdviceGradient`**(*example*)
Proves each clause (*`Prover.prove()`*) and computes the advice gradient as `NumberTrue - NumberFalse`.

Parameters **`example`** –

`rfgb.boosting.`**`computeSumOfGradients`**(*example*, *trees*, *data*)
Computes new gradients for an example.

Parameters

- **`example`** –
- **`trees`** –
- **`data`** –

`rfgb.boosting.`**`inferTreeValue`**(*clauses*, *query*, *data*)
Returns the probability of *query* given data and the clauses learned.

Parameters

- **`clauses`** –
- **`query`** –

- **data** –

rfgb.boosting.**performInference**(*testData*, *trees*)
Computes the probabilities for test examples.

Parameters

- **testData** (*utils.Data* object.) – Data for testing.
- **trees** (*list.*) – List of strings representing learned decision trees.

Example:

```
from rfgb.boosting import performInference
```

rfgb.boosting.**updateGradients**(*data*, *trees*, *loss='LS'*, *delta=None*)
Update gradients of the data.

Parameters

- **data** (*utils.Data* object.) – Training or testing data (with parameters).
- **trees** (*list.*) – List of strings representing trees.
- **loss** (*str.*) – Loss function for regression (currently implemented: 'LS', 'LAD', 'Huber').
- **delta** (*float*) – Delta value for Huber loss.

Example:

```
from rfgb.boosting import updateGradients
```

4.2 rfgb.logic module

(docstring)

class rfgb.logic.**Goal**(*rule*, *parent=None*, *env={}*)
Bases: object

class for each goal in rule during prolog search

class rfgb.logic.**Logic**
Bases: object

Class for logic operations.

static constantsPresentInLiteral(*literalTypeSpecification*)
Returns true if constants present in type specification.

static generateTests(*literalName*, *literalTypeSpecification*, *clause*)
Generates tests for literal according to modes and types.

static getVariables(*literal*)
Returns variables in the literal.

class rfgb.logic.**Prover**
Bases: object

class for prolog style proof of query

goalId = 100

static prove(*data*, *example*, *clause*)
Proves if example satisfies clause given the data. Returns True if it satisfies, else return False.

Prover.rules: contains all of the rules. Prover.trace: If this is 1, displays the proof tree. Prover.goalID: stores the goal ID.

rules = []

static search(*term*)
Method to perform prolog style query search.

trace = 0

static unify(*srcTerm*, *srcEnv*, *destTerm*, *destEnv*)
Unification method.

class rfgb.logic.**Rule**(*s*)
Bases: object

Class for logic rules in prolog proof.

class rfgb.logic.**Term**(*s*)
Bases: object

Class for term in prolog proof.

4.3 rfgb.tree module

Data structures and methods for learning decision trees.

class rfgb.tree.**node**(*test=None*, *examples=None*, *information=None*, *level=None*, *parent=None*, *pos=None*)
Bases: object

A node in a tree.

Parameters

- **expandQueue** – Breadth first search node expansion strategy
- **depth** – initial depth is 0 because no node present
- **maxDepth** – max depth set to 1 because we want to at least learn a tree of depth 1
- **learnedDecisionTree** – this will hold all the clauses learned
- **data** – stores all the facts, positive and negative examples

data = None

depth = 0

expandOnBestTest(*data=None*)
Expand the node based on the best test.

expandQueue = []

getTrueExamples(*clause*, *test*, *data*)
Returns all examples that satisfy the clause with conjoined test literal.

static initTree(*trainingData*)
Create the root node of the tree.

static learnTree(*data*)
Method to create and learn the decision tree.

learnedDecisionTree = []

maxDepth = 1

static setMaxDepth(*depth*)
Set the maximum depth of the tree.

4.4 rfgb.utils module

(docstring for utils)

class rfgb.utils.**Data**(*regression=False*, *advice=False*, *softm=False*, *alpha=0.0*, *beta=0.0*)
Bases: object

Object containing the relational data.

getExampleTrueValue(*example*)
Returns true regression value of an example for regression learning.

getFacts()
returns the facts in the data

getLiterals()
gets all the literals in the facts

getTarget()
Returns the target.

getValue(*example*)
Returns the regression value for an example.

Example:

```
from rfgb.utils import Utils
from rfgb.utils import Data

trainingData = Utils.readTrainingData('cancer',
                    path='testDomain/ToyCancer/train/')

x = trainingData.getValue('cancer(earl)')
# x == -0.5, since earl doesn't have cancer.

y = trainingData.getValue('cancer(alice)')
# y == 0.5, since alice does have cancer
```

setBackground(*bk*)
Obtains the literals and their type specifications. Types can be either variable or a list of constants.

setExamples(*examples*, *target*)
Set examples for regression.

setFacts(*facts*)
Mutate the facts in the data object.
Parameters **facts** (*list.*) – List of strings representing the facts.
Returns None

setNeg(*neg*, *target*)
Set negative examples based on the contents of a list.

setPos(*pos*, *target*)
Set positive examples based on the contents of a list.

setTarget(*bk*, *target*)
Sets self.target as a target string. Sets self.variableType
Parameters
- **bk** (*list.*) – List of strings representing modes.
- **target** (*str.*) – Target relation or attribute.

Returns None
Example:

```
from rfgb.utils import Data

data = Data(regression=False)
background = ['friends(+person,-person)',
              'friends(-person,+person)',
              'smokes(+person)',
              'cancer(-person)']
target = 'cancer'

data.setTarget(background, target)

print(data.target)
# 'cancer(C)'
```

variance(*examples*)
Calculates the variance of the regression values from a subset of the data.

class rfgb.utils.**Utils**
Bases: object

Class of utilities used by rfgb, such as reading files, removing mode symbols, calculating Cartesian Products, etc.

UniqueVariableCollection = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',

static addVariableTypes(*literal*)
As literals are encountered, update Utils.data.variableType with the type of the variables encountered.
Parameters **literal** (*str.*) – A literal of the form smokes(W) or friends(A,B)

static cartesianProduct(*itemSets*)
Returns the Cartesian Product of all sets contained in the item sets.

data = None

static getleafValue(*examples*)
returns average of regression values for examples

static load(*location*)
Loads json version of learnedDecisionTree from location.
Parameters **location** (*str.*) – Name of the file to load.
Returns None.

static readTestData(*target*, *path='test/'*, *regression=False*)
Reads the testing data from files.
Parameters
- **target** (*str.*) – The target predicate.

- **path** (*str.*) – Path to the training data.
- **regression** (*bool*) – Read from `examples.txt` instead of `pos.txt` and `neg.txt`.

Default path 'train/'
Default regression False
Returns A Data object representing the training data.
Return type *utils.Data*

static readTrainingData(*target*, *path='train/'*, *regression=False*, *advice=False*, *softm=False*, *alpha=0.0*, *beta=0.0*)

Reads the training data from files.

Parameters

- **target** (*str.*) – The target predicate.
- **path** (*str.*) – Path to the training data.
- **regression** (*bool*) – Read from `examples.txt` instead of `pos.txt` and `neg.txt`.
- **advice** (*bool*) – Read advice from an advice file, which should be contained in the same directory as the examples.

Default path 'train/'
Default regression False
Default advice False
Returns A Data object representing the training data.
Return type *utils.Data*

static removeModeSymbols(*inputString*)

Returns a string with the mode symbols (+,-,#) removed.

Example:

```
from rfgb.utils import Utils

removeModeSymbols('#city')
# == 'city'

i = ['+drinks', '-drink', '-city']
o = list(map(removeModeSymbols, i))
# o == ['drinks', 'drink', 'city']
```

static save(*location*, *saveItem*)

Dumps json version of learnedDecisionTree to location.

Parameters **location** (*str.*) – Name of the file to write.
Returns None.

static sigmoid(*x*)

Parameters **x** (*int or float*) – Number to apply sigmoid to.
Returns `exp(x)/float(1+exp(x))`
Return type float

CHAPTER 5

rfgb.rdn package

New in version 0.3.0.

Learn and infer with relational dependency networks.

```
# Example script for performing learning and inference.

from rfgb import rdn

# rdn.learn requires a list of targets as strings.
trees = rdn.learn(['cancer'], path='testDomains/ToyCancer/train/')

# rdn.learn returns a dictionary mapping targets to trees.
cancer_trees = trees['cancer']

# rdn.infer classification returns a tuple of pos and neg.
results = rdn.infer('cancer', cancer_trees, path='testDomains/ToyCancer/test/')

# ({'cancer(xena)': 0.34460796550872186,
#   'cancer(yoda)': 0.34460796550872186,
#   'cancer(zod)': 0.34460796550872186},
#  {'cancer(watson)': 0.34460796550872186,
#   'cancer(voldemort)': 0.34460796550872186})
```

5.1 rfgb.rdn.learn module

rfgb.rdn.learn.**learn**(*targets*, *numTrees=10*, *path=''*, *regression=False*, *advice=False*, *softm=False*, *alpha=0.0*, *beta=0.0*, *saveJson=True*)

New in version 0.3.0.

Learn a relational dependency network from facts and positive/negative examples via relational regression trees.

Note: This currently requires that training data is stored as files on disk.

Parameters

- **targets** (*list of str.*) – List of target predicates to learn models for.
- **numTrees** (*int.*) – Number of trees to learn.
- **path** (*str.*) – Path to the location training data is stored.
- **regression** (*bool.*) – Learn a regression model instead of classification.
- **advice** (*bool.*) – Read an advice file from the same directory as trainPath.

Default regression False

Default advice False

Returns Dictionary where the key is the target and the value is the set of trees returned for that target.

Return type dict.

5.2 rfgb.rdn.infer module

rfgb.rdn.infer.**infer**(*target*, *trees*, *path=''*, *regression=False*)

New in version 0.3.0.

Perform inference on data with a set of trees.

Note: This currently requires that test data is stored as files on disk.

Parameters

- **trees** (*list of str.*) – Trees to perform inference with.
- **path** (*str.*) – Path to the location test data is stored.
- **regression** (*bool.*) – Infer with a regression model instead of classification.

Default regression False

Returns Tuple of results. In classification these results will be a tuple of positive and negative examples. In regression this will be the examples.

Return type tuple

CHAPTER 6

Indices and tables

- genindex
- modindex

Python Module Index

r

Index

S

T

U

V