Towards README-EVAL : Interpreting README File Instructions

James Paul White
Department of Linguistics
University of Washington
Seattle WA 98195-4340
`jimwhite@uw.edu`

Abstract

This abstract describes README-EVAL, a novel measure for semantic parsing evaluation of interpreters for instructions in computer program README files. That is enabled by leveraging the tens of thousands of Open Source Software programs that have been annotated by package maintainers of GNU/Linux operating systems. We plan to make available a public shared implementation of this evaluation.

1 Introduction

That natural language is learned by humans in rich grounded perceptual contexts has been recognized by many researchers for quite some time (Regier, 1996) (Silberer and Lapata, 2012). But most efforts at machine learning of natural language continue to address tasks which are entirely divorced from any grounding and/or have perceptual requirements for which machines are ill-suited. Computers are machines and their natural perceptual context is that of the computing machine world. Therefore, to apply the model of grounded language learning most effectively, we should choose tasks in which the relevant percepts are of those in the computing world (e.g., bits, bytes, characters, files, memory, operations, programs, events, processes, services, devices, processors, drivers, operating systems, and networks).

This abstract describes proposed work aimed at the goal of deep semantic parsing of the web, which for us includes the ability to interpret documents that give instructions for acting on computer systems in human natural language. To facilitate research in that direction, we plan to evaluate systems that build software packages by following the README[1] file instructions contained in GNU/Linux distributions like Centos and Debian. Key to this plan is the novel README-EVAL score which we propose as an extrinsic (i.e. goal-oriented) performance measure for parsing, mapping/planning, and related linguistics tasks. The planned baseline system is a pipeline using a document classifier and instruction sequence extractor trained on hand-labeled data followed by a reinforcement learner for mapping the instructions to a build script (plan of actions) for that software package (context).

2 Background

A significant challenge for semantic parsing research is finding a method to measure a system's performance that will indicate its effectiveness in the domain of interest. Traditionally the approach has been to gather and have human annotators make judgements that are of the same kind the system is intended to perform. That process is relatively costly and may result in a corpus which is actually too small considering the amount of variation that occurs when humans perform an activity. Relevant prior work in the computing domain produced the Linux and Monroe plan corpora (Blaylock and Allen, 2005). The Linux Plan Corpus consists of 457 interactive shell script sessions, with an average of 6.1 actions each, captured from human experimental subjects attempting to satisfy one of 19 different goals stated as an English sentence. Although it has been used successfully by those and other researchers, the natural variation in human behavior means that a corpus of such relatively small size appears to be very noisy. As a result they have had to rely on artificially generated data such as the Monroe Plan Corpus in order to get results that are more easily compared across system evaluations.

[1]We use the term README file in a broad sense meaning a document that contains instructions to be read by a human that concern performing actions on a computer (whether at the keyboard or some other input device). For this task we confine ourselves to instructions given for the purpose of building a software package.

More promising therefore is the way some researchers have discovered ways to repurpose data and/or judgements created for other purposes and turn them into training data and/or evaluations of NLP systems. We employ that paradigm here by repurposing the efforts of Open Source Software (OSS) package maintainers who have created annotations (aka metadata) including dependency relations and scripts that build computer programs.

3 GNU/Linux Software Package Data

The advent of the Internet resulted in explosive growth for OSS, the premier example of which is the GNU/Linux operating system family. Current distributions contain packages built from over 15,000 program source bundles.[2] The production of OSS packages for such systems typically involves two different types of programmers working independently. The authors of the source computer program usually do not produce packaging metadata for their work and instead tend to write README files and related documentation explaining how to build and use the software. The package maintainers then work out the specific requirements and scripts necessary to build the program as some package(s) using the particular package manager and format of the OS distribution (aka ”distro”) that they are supporting. Software package metadata contained in bundles such as Debian `.deb` and Fedora RPM `.spec` files are rich in annotations.[3,4]

See Figure 1 for excerpts of text describing the Bean Scripting Framework (BSF) from its Source RPM Package Manager (SRPM) package in the Fedora Core 17 distribution.[5] The two kinds of data shown are file contents (1a, 1c, 1e), which usually originate with the “upstream” program author(s), and sections from the RPM Spec file (1b, 1d, 1f), which are annotations (aka metadata) curated by the package maintainers. There are other sections and fields used in RPM Spec files, but those tend to more distro-specific and these suffice for this discussion.

Figure 1a shows some BSF package description text from the source README.txt file and Figure 1b shows the version appearing the RPM Spec. That close textual similarity is a common occurrence in the data and can be used to identify some likely README files. Those are only a starting point though, because the natural language program build instructions are often in other files, as in this case. For many packages those instructions are in a file named INSTALL. There is an INSTALL.txt file with some instructions for BSF here (Figure 1e), but they are for a binary installation. The instructions for building from source that we will primarily concerned with here are in the file BUILDING.txt (Figure 1c).

A potential use for this data that we haven’t explored yet is its use in summarization tasks. In addition to the text which is usually in the README file and RPM Spec DESCRIPTION section, there is the “Summary” field of the PACKAGE section. Although in Figure 1d the value for the summary field appears as just the package’s full name, this is typically a full sentence that is a good one-line summary of the multiple line description section.

It is worthwhile to notice that thousands of programs have been packaged multiple times for different systems (e.g. Debian, Fedora, Cygwin, NixOS, Homebrew, and others) and many packages have also been internationalized.[6] Both of those aspects point to opportunities for learning from parallel data.

For the present discussion we focus on two particular elements of package metadata: dependencies and build scripts.[7] The packages in a distribution have dependency relationships which designate which packages must be built and installed for other packages to be built, installed, and/or executed. These relationships form a directed acyclic graph (DAG) in which the nodes are packages and the edges are dependency relationships.

[2]Debian Wheezy has over 37,000 packages from about 17,500 source packages `https://www.debian.org/News/2013/20130504` and Fedora 20 has more than 15,000 packages `https://admin.fedoraproject.org/pkgdb/collections/`.

[3]`https://www.debian.org/doc/manuals/maint-guide/dreq.en.html`

[4]`http://www.rpm.org/max-rpm/ch-rpm-inside.html`

[5]For more examples, we refer the interested reader the author’s web page which includes access to a web linked data explorer for the entire corpus. `http://students.washington.edu/jimwhite/sp14.html`

[6]Debian for example currently lists more than 800k sentences in the localization database and about 75 human languages have translations for at least 100k of them with the top ten languages having over 500k each `https://www.debian.org/international/l10n/po/rank`.

[7]Packaging systems usually support at least three types of scripts: build, install, and remove. The build script usually has more in common with the README instructions than the install and remove scripts which are more distro specific. Some packages also have a check script to validate the state of a build prior to performing the install operation.

(a) README.txt file

Bean Scripting Framework (BSF) is a set of Java classes which provides an easy to use scripting language support within Java applications. It also provides access to Java objects and methods from supported scripting languages. ...

(b) RPM Spec DESCRIPTION section

Bean Scripting Framework (BSF) is a set of Java classes which provides scripting language support within Java applications, and access to Java objects and methods from scripting languages.
...

(c) BUILDING.txt file

```
From the ant "build.xml" file:
  Master Build file for BSF
Notes:
  This is the build file for use with
  the Jakarta Ant build tool.
Optional additions:
 BeanShell -> http://www.beanshell.org/
 Jython -> http://www.jython.org/
 JRuby -> http://www.jruby.org/ (3rd ...)
 Xalan -> http://xml.apache.org/xalan-j
 ...
Build Instructions:
 To build, run
  java org.apache.tools.ant.Main <target>
 on the directory where this file is
 located with the target you want.
Most useful targets:
- all -> creates the binary and src
 distributions, and builds the site
- compile -> creates the "bsf.jar"
 package in "./build/lib" (default target)
- samples -> creates/compiles the samples
...
```

(d) RPM Spec PACKAGE section (metadata)

```
Name:           bsf
Version:        2.4.0
Release:        12.fc17
Summary:        Bean Scripting Framework
License:        ASL 2.0
URL:    http://commons.apache.org/bsf/
Group:          Development/Libraries
BuildRequires: jpackage-utils >= 1.6
BuildRequires: ant, xalan-j2, jython
BuildRequires: rhino
BuildRequires: apache-commons-logging
Requires:       xalan-j2
Requires:       apache-commons-logging
Requires:       jpackage-utils
BuildArch:      noarch
...
```

(e) INSTALL.txt file

```
Installing BSF consists of copying
bsf.jar and .jars for any languages
intended to be supported to a directory
in the execution CLASSPATH of your
application, or simply adding them
to your CLASSPATH.
BSF can be used either as a standalone
system, as a class library, or as part
of an application server. In order to be
used as a class library or as a standalone
system, one must simply download the
bsf.jar file from the BSF web site
(http://jakarta.apache.org/bsf/index.html)
and include it in their CLASSPATH, along
with any required classes or jar files
implementing the desired languages.
...
```

(f) RPM Spec BUILD section (shell script)

```
[ -z "$JAVA_HOME" ] && export JAVA_HOME=/usr/lib/jvm/java
export CLASSPATH=$(build-classpath apache-commons-logging jython xalan-j2 rhino)
ant jar
/usr/bin/rm -rf bsf/src/org/apache/bsf/engines/java
ant javadocs
```

Figure 1: Bean Scripting Framework (BSF) excerpts from Fedora Core 17 RPMS.

4 From Dependencies to Validation

The idea that turns the package dependency DAG into training, test, and evaluation data is to choose dependency targets for test (i.e. the system build script outputs will be used for them in test) and dependency sources (the dependent packages) for validation (their package maintainer written build scripts are used as is to observe whether the dependencies are likely to be good). Validation subsets can be arranged for both internal validation (tuning) and external validation (evaluation).

Two kinds of dependency relationships are of special interest here: `Requires` and `BuildRequires`. The former typically means the target package (its name appears to the right of a `Requires` or `BuildRequires` in Figure 1d) is required at both build time and execution time by the source package (identified by the `Name` field of Figure 1d) while the latter means it is only required at build time. That distinction can be used to guide the selection of which packages to choose for the validation and test subsets. Packages that are the target of a `BuildRequires` relationship are more likely to cause their dependents' build scripts to fail when they (the targets) are built incorrectly than targets of a `Requires` relationship.

Analysis of the 2,121 packages in Release 17 of the Fedora Core SRPM distribution shows 1,673 package nodes that have a build script and some declared dependency relationship. Those build scripts average 6.9 non-blank lines each. Of those nodes, 1,009 are leaves and the 664 internal nodes are the target of an average of 7 dependencies each. There are 218 internal nodes that are the direct target of at least one leaf node via a `BuildRequires` relationship and they average 12.4 such dependent leaves each. We expect to have a larger corpus prepared from a full GNU/Linux distribution (at least 15,000 source packages) at the time of the workshop.

5 Task Description

The top-level README-EVAL task would be to generate complete packaging metadata given the source files for a program thus automating the work of a package maintainer. Since that task is somewhat complicated, it is useful to break it down into multiple subtasks which can be addressed and evaluated separately before proceeding to combine them. For the discussion here we will consider a partial solution using a four stage pipeline: README document classification, instruction extraction, dependency relation extraction, and build script generation.

The corpus' package metadata can be used to directly evaluate the results of the last two stages of that pipeline. The first two stages, README document classification and instruction extraction, are well understood tasks for which a moderate amount of manually labelled data can suffice to train and test effective classifiers.

The dependency relation extraction subtask can be treated as a conventional information extraction task concerned with named entity recognition for packages and relation extraction for dependencies. We may regard the dependencies in the corpus as effectively canonical because the package maintainers strive to keep those annotations to a reasonable minimum. Therefore computing precision and recall scores of the dependency DAG edges and labels of this stage's output versus the corpus' metadata will be a meaningful metric.

Work on instruction and direction following is applicable to the build script generation subtask. Such systems tend to be somewhat more complex than shallow extraction systems and may incorporate further subcomponents including goal detectors and/or planners that interact with a semantic parser (Branavan et al., 2012). It is possible to evaluate the final stage output by comparing it to the build script in the package's metadata, but that would suffer from the same sort of evaluation problems that other language generation tasks have when we are concerned with semantics rather than syntax. This is where the superiority of an NLP task where the target language is understood by computers comes in, because we can also evaluate it using execution. Which isn't to say we can solve the program equivalence problem in general, but README-EVAL does a pragmatic determination of how good a substitute it is based on its usage by the package's dependency sources.

6 README-EVAL Scoring

The README-EVAL score is a measure of how effective the system under test (SUT) is at generating software package metadata. For the components of the SUT this score can serve as an extrinsic indication of their effectiveness.

Let N be a set of tuples (x, y) representing the corpus in which x is the package data and relevant metadata subset minus the labels to be generated and y is a known good label for x. To prepare the corpus for the task, two disjoint subsets C and T are selected from the set of all package nodes N. C is for the common packages which are available to the SUT for training, and T is for the test packages that the SUT's interpretation function will be tested on. A third set V which is disjoint from T is selected from N for the validation packages.

Many partitioning schemes are possible. A simple method is to choose the leaf nodes (packages that are sources but not targets of dependency relationships) for V. The members of T can then be chosen as the set of packages which are the direct targets of the dependency relationships from V. The members of V are expected to be likely to fail to build correctly if there are errors in the system outputs for T. Note that for the SUT to do tuning it will need some leaf node packages in C. Therefore if V is made disjoint from C then it should not actually select all of those leaves.

The README-EVAL score R is computed using a suitable loss function L for the SUT's label predictor function $\hat{Y}$. $\hat{Y}$ is presumed to have been trained on C and it yields a set of $(x, \hat{y})$ tuples given a set of x values. The loss function $L((x, y), D)$ yields a real number in the range 0 to 1 inclusive that indicates what fraction of the components in package (x, y) are incorrect given the context $D \subset N$. It is required for all $v \in V$ that $L(v, (C \cup T \cup V) \setminus \{v\}) = 0$.

For this exposition, assume y is a build script and L yields 0 if it succeeds and 1 if it fails. Linux processes typically indicate success by returning a zero exit code. Therefore a simple realization of

L is to return 0 if the process executing the build script y of (x, y) given D returns zero and 1 otherwise.

The computation iterates over each member $D \in partition(T)$ and obtains measures of correctness by evaluating $B(\hat{Y}(X(D)) \cup C \cup T \setminus D)$ where X is a function that yields the set of x values for a given set of (x, y) tuples. To keep the task as easy as possible, the members of $partition(T)$ may be singletons.

$$B(D) = |V| - \textstyle\sum_{v \in V} L(v, (D \cup V) \setminus \{v\})$$

Those values are normalized by a scale factor for each D determined by the value of B given D minus B given $Z(D)$. $Z(D)$ is the set of tuples (x, λ) for a given set D where λ is the null label. A null label for a build script is one which has no actions and executes successfully.

$$R(D) = \frac{B(\hat{Y}(X(D)) \cup C \cup T \setminus D)}{B(C \cup T) - B(Z(D) \cup C \cup T \setminus D)}$$

The final README-EVAL measure R is the average score over those partitions:

$$R = \frac{\sum_{D \in partition(T)} R(D)}{|partition(T)|}$$

6.1 Loss Function Variations

There are other useful implementation variations for the loss function L. In a system where the number of components can be determined independently from whether they are correct or not, a possibly superior alternative is to return the number of incorrect components divided by the total number of components. To determine loss for a build script for example, the value may be determined by counting the number of actions that execute successfully and dividing by the total number of steps.

A further consideration in semantic evaluation is parsimony, which is the general expectation that the shortest adequate solution is to be preferred (Gagne et al., 2006). To incorporate parsimony in the evaluation we can add a measure(s) of the solution's cost(s), such as the size of the label y and/or execution resources consumed, to L.

7 Conclusion

A common objection to tackling this task is that it seems too hard given the state of our knowledge about human language, computer programming (as performed by humans), and especially the capabilities of current NLP systems. We consider that to be a feature rather than a bug. It may be some time before a state-of-the-art implementation of a README interpreter is sufficiently capable to be considered comparable to an expert human GNU/Linux package maintainer performance, but that is perfectly fine because we would like to have an evaluation that is robust, long-lived, and applicable to many NLP subtasks. We also have the more pragmatic response given here which shows that that difficult task can be decomposed into smaller subtasks like others that have been addressed in the NLP and computational linguistics communities.

To conclude, this proposal recommends README-EVAL as an extrinsic (goal-oriented) evaluation system for semantic parsing that could provide a meaningful indication of performance for a variety of NLP components.

Because the evaluation platform may be somewhat complicated to set up and run, we would like to make a publicly available shared evaluation platform on which it would be a simple matter to submit new systems or components for evaluation. The MLcomp.org system developed by Percy Liang and Jacob Abernethy, a free website for objectively comparing machine learning programs, is an especially relevant precedent (Gollub et al., 2012). But we notice that the NLP tasks on MLcomp receive little activity (the last new run was more than a year ago at this writing) which is in stark contrast to the other ML tasks which are very active (as they are on sites like Kaggle). With the README-EVAL task available in such an easy-to-use manner could draw significant participation because of its interesting and challenging domain, especially from ML and other CS students and researchers.

Finally we look forward to discussing this proposal with the workshop attendees, particularly in working out the details for manual annotation of the README files for the instruction extractor (including whether it is needed), and discussing ideas for a baseline implementation.

8 Acknowledgements

Thank you to my University of Washington colleagues who reviewed earlier drafts of this abstract and the workshop's blind reviewers for their helpful comments.

References

Nate Blaylock and James Allen. 2005. Recognizing Instantiated Goals using Statistical Methods. In *IJCAI Workshop on Modeling Others from Observations (MOO-2005)*, page 79.

S. R. K. Branavan, Nate Kushman, Tao Lei, and Regina Barzilay. 2012. Learning High-Level Planning from Text. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers-Volume 1*, page 126. Association for Computational Linguistics.

Christian Gagne, Marc Schoenauer, Marc Parizeau, and Marco Tomassini. 2006. Genetic Programming, Validation Sets, and Parsimony Pressure. In *Genetic Programming*, page 109. Springer.

Tim Gollub, Benno Stein, and Steven Burrows. 2012. Ousting Ivory Tower Research: Towards a Web Framework for Providing Experiments as a Service. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*, page 1125. ACM.

Terry Regier. 1996. *The Human Semantic Potential: Spatial Language and Constrained Connectionism*. MIT Press.

Carina Silberer and Mirella Lapata. 2012. Grounded Models of Semantic Representation. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, page 1423. Association for Computational Linguistics.