Edlin: an easy to read linear learning framework

Kuzman Ganchev *
University of Pennsylvania
3330 Walnut St, Philaldelphia PA
kuzman@seas.upenn.edu

Georgi Georgiev
Ontotext AD
135 Tsarigradsko Ch., Sofia , Bulgaria
georgi.georgiev@ontotext.com

Abstract

The Edlin toolkit provides a machine learning framework for linear models, designed to be easy to read and understand. The main goal is to provide easy to edit working examples of implementations for popular learning algorithms. The toolkit consists of 27 Java classes with a total of about 1400 lines of code, of which about 25% are I/O and driver classes for examples. A version of Edlin has been integrated as a processing resource for the GATE architecture, and has been used for gene tagging, gene name normalization, named entity recognition in Bulgarian and biomedical relation extraction.

Keywords

Information Extraction, Classification, Software Tools

1 Introduction

The Edlin toolkit provides a machine learning framework for linear models, designed to be easy to read and understand. The main goal is to provide easy to edit working examples of implementations for popular learning algorithms. To minimize programmer overhead, Edlin depends only on GNU Trove[1] for fast data structures and JUnit[2] for unit tests. A version of Edlin has been integrated as a processing resource for the GATE [7] architecture, and has been used in-house for gene tagging, gene name normalization, named entity recognition in Bulgarian and biomedical relation extraction. For researchers we expect the main advantage of Edlin is that its code is easy to read, understand and modify, meaning that variations are easy to experiment with. For industrial users, the simplicity of the code as well as relatively few dependencies means that it is easier to integrate into existing codebases.

Edlin implements learning algorithms for linear models. Currently implemented are: Naive Bayes, maximum entropy models, the Perceptron and one-best MIRA (optionally with averaging), AdaBoost, structured Perceptron and structured one-best MIRA (optionally with averaging) and conditional random fields. Because of the focus on clarity and conciseness, some optimizations that would make the code harder to read have not been made. This makes the framework slightly slower than it could be, but implementations are asymptotically fast and suitable for use on medium to large datasets.

The rest of this paper is organized as follows: §2 describes the code organization; §3-§4 describes an integration with the GATE framework and an example application; §5 describes related software; and §6 discusses future work and concludes the paper.

2 Overview of the code

The goal of machine learning is to choose from a (possibly infinite) set of functions mapping from some input space to some output space. Let $x \in X$ be a variable denoting an input example and $y \in Y$ range over possible labels for x. A linear model will choose a label according to

$$h(x) = \arg\max_y f(x, y) \cdot w \qquad (1)$$

where $f(x, y)$ is a feature function and w is a parameter vector. We take the inner product of the feature vector with the model parameters w and select the output y that has high-

*Supported by ARO MURI SUBTLE W911NF-07-1-0216 and by the European projects AsIsKnown (FP6-028044) and LTfLL (FP7-212578)

[1] http://trove4j.sourceforge.net/

[2] http://www.junit.org

est such score. The feature function $f(x, y)$ is specified by the user, while the parameter vector w is learned using training data.

Even though the learning and inference algorithms are generic, and can be used for different applications, Edlin is implemented with an natural language tasks in mind. The classes related to classification, are implemented in the `classification` package, while those related to sequence tasks are implemented in the `sequence` package. The code to perform gradient ascent and conjugate gradient is in an `algo` package. There are three helper packages. Two (`experiments` and `io`) are code for reading input and for driver classes for the examples. The final package, called `types` contains infrastructure code such as an implementation of sparse vectors, elementary arithmetic operations such as the inner product, and other widely used operations whose implementation is not interesting from the point of view of understanding the learning algorithms. This code organization, as well as the data structures we employ are similar to other learning packages such as StructLearn [12] and MALLET [11]. One attribute that distinguishes Edlin from both of these packages is the decomposition of the feature function into

$$f(x, y) = f_2(f_1(x), y) \quad (2)$$

where f_1 maps the input into a sparse vector and f_2 combines it with a possible output in order to generate the final sparse vector used to assess the compatibility of the label for this input. By contrast, many other learning frameworks only allow the user to specify f_1 and hard-code an implementation of f_2 as conjoining the input predicates (f_1 in the notation above) with the label. By allowing the user to specify f_2, we allow them to tie parameters and add domain information about how different outputs are related. See the illustration below for an example.

2.1 Example Application

Perhaps the best way to describe the minimal amount to make reading the code easy is to trace how information is propagated and transformed in an example application. Take a POS tagging task as an example. Suppose we are given a collection of sentences that have been manually annotated and these have been split for us into a training set and a testing set. The sentences are read from disk and converted to a sparse vector representing $f_1(x)$ by a class in the `io` package. For example we might extract suffixes of length 2 to 5 from each word in a sentence. We look these up in an alphabet that maps them to a unique dimension, and store the counts of the words in a sparse vector for each word. The alphabet and sparse vector are implemented in `Alphabet` and `SparseVector` respectively. The array of sparse vectors for a sentence (recall there is one for each word) and alphabet are bundled together in an `SequenceInstance` object along with the true label. Next we want to train a linear sequence model using the perceptron algorithm on the training portion of our data. We construct a `sequence.Perceptron` object and call its `batchTrain` method. Figure 1 reproduces the implementation. The method takes the training data as a Java `ArrayList` of `SequenceInstance` objects, and the `Perceptron` class has parameters for whether averaging is turned on and the number of passes to make over the data. It also contains a `SequenceFeatureFunction` object (`fxy` in Figure 1) that implements f_2 from above. For part of speech tagging, it is typical to let $f_t(x, y^{(t-1)}, y^{(t)})$ conjoin $f_1(x)$ with $y^{(t)}$ and also conjoin $y^{(t)}$ with $y^{(t-1)}$, but not to have any features that look at x, $y^{(t)}$ and $y^{(t-1)}$ all at the same time. By contrast for named entities it is typical to have features that look at all three. The linear sequence model is created in the first line of the `batchTrain` method as a `LinearTagger` object, which has access to the alphabet used in the initial construction of the sparse vectors, the label alphabet (`yAlphabet` in Figure 1) and f_2 (`fxy` in Figure 1). It computes the prediction which is represented as an `int` array, with the interpretation `yhat[t]=j` as word t has the label at position j in the label alphabet (accessible via `yAlphabet.lookupIndex(j)`). The `batchTrain` method returns the linear sequence model.

3 GATE integration

GATE [8, 7] is a framework for engineering NLP applications along with a graphical development environment for developing components. GATE divides language processing resources into language resources, processing re-

```
public LinearTagger batchTrain(
          ArrayList<SequenceInstance> trainingData) {
  LinearTagger w = new LinearTagger(xAlphabet,
                                   yAlphabet, fxy);
  LinearTagger theta = null;
  if (performAveraging)
    theta = new LinearTagger(xAlphabet, yAlphabet,
                                                fxy);
  for (int iter = 0; iter < numIterations; iter++) {
    for (SequenceInstance inst : trainingData) {
      int[] yhat = w.label(inst.x);
      // if y = yhat then this update won't change w.
      StaticUtils.plusEquals(w.w, fxy.apply(inst.x,
                                          inst.y));
      StaticUtils.plusEquals(w.w, fxy.apply(inst.x,
                                       yhat), -1);
      if (performAveraging)
        StaticUtils.plusEquals(theta.w, w.w, 1);
    }
  }
  if (performAveraging) return theta;
  return w;
}
```

Fig. 1: *Edlin's perceptron implementation, reproduced verbatim to show code organization.*

sources, and graphical interfaces. We have integrated a version of Edlin into the GATE framework as a set of processing resources, by defining interfaces in Edlin for training, classification, and sequence tagging. These interfaces are used to communicate between Edlin's machine learning implementations and the concrete implementations of tagger and classifier processors in GATE. The integration allows Edlin to be used for robust, complex text processing applications, relying on GATE processors such as tokenizers, sentence splitters and parsers, to preprocess the data. The integration also makes it easy to pipeline Edlin-trained linear models using the GATE infrastructure for processing pipelines. Since Edlin has very readable code, this makes it easy for a researcher or engineer to try a modified learning algorithm if they already use the GATE framework.

4 Biomedical Relation Extraction

In this section we show an example text processing application within the Edlin and GATE architectures, focusing on the text processing components organization. Our problem domain is the BioNLP 2009 shared task [17], a biomedical relation extraction task. The goal is to identify relations between genes/gene products. We chose this task as an example because it is relatively complex and uses both Edlin and several GATE processing components. The results are described in [9].

Following BioNLP terminology, we use the term proteins to refer to both genes and gene products. Both trigger chunks and proteins are called *participants*. For example, the text "... phosphorylation of TRAF2 ..." would be a relation of type Phosphorylation with a theme of TRAF2. The relation is called an *event*, while the string "phosphorylation" is called a trigger. Gene boundary annotations are provided by the task organizers. In general, there are events with multiple participants in addition to the trigger. The event instances are organized into the structure of the Gene Ontology [5].

We separated the task in two main sub-tasks (i) recognition of trigger chunks using an Edlin sequence tagger and, (ii) classification of triggers and proteins as either forming an event from one of 9 predefined types or not participating in an event together. At the end of the section we discuss the final pipeline of processors used in this relation extraction task.

4.1 Gene and Trigger Tagging

The trigger chunks are simply words and phrases that describe the events linking proteins. For instance *binds* is such a trigger word that would link two or more genes in a *Binding* event. We used the Edlin GATE integration described in Section 3 to create one GATE processing resource that trains an Edlin linear sequence model and another that uses that Edlin sequence model to tag trigger chunks.

Both processors work in a pipeline with GATE preprocessors including a tokenizer, sentence splitter, POS tagger and chunker. Because Edlin represents linear models trained using different algorithms in the same way it was easy for us to compare different learning algorithms for the task. For this application tagger recall is an upper bound on system performance, and used MIRA with a loss function designed to achieve high recall since that performed best.

4.2 Relation Extraction

We separate the process of relation extraction into two stages: in the first stage, we generate events corresponding to relations between a trigger word and one or more proteins (*simple* events), while in the second stage, we gener-

ate events that correspond to relations between trigger words, proteins and simple events (we call the new events *complex* events).

For the purpose of this task we designed and implemented four GATE processing resources for two for training and two for classification of genes and trigger chunks into the 9 predefined types of events. The training of an Edlin linear model and classification using that model are again done using the Edlin-GATE integration, and are integrated in a GATE pipeline that now also includes dependency and phrase-structure parsers.

As with finding trigger words in the previous section, the uniform representation of linear models allowed us to compare different learning methods. We compared max entropy, perceptron and one-best MIRA, and again chose MIRA with a loss function designed to increase recall, since getting high recall was the most challenging part of the task. Finally, this tunable loss function was appealing for us because it allows application-specific tuning. For example, a search might require high recall, but high precision might be more important for adding relations to a knowledge base.

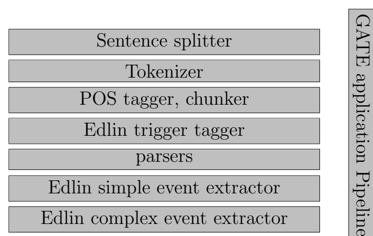

Fig. 2: *Graphical view of our relation extraction system pipeline.*

Figure 2 shows our event extraction pipeline, stringing together different GATE text processors. The first stages of the pipeline as well as the parsers are included in order to create features useful for later stages.

As described above, the trigger tagging stage uses an Edlin GATE processor trained using one-best MIRA. Furthermore, we employ a maximum entropy constituency parser [1] and a dependency parser [13]. These components are also represented as GATE processors. In the last stage of the pipeline we use two components, one for *simple* and one for *complex* events, based on the classification version of one-best MIRA algorithm implemented in Edlin and used as GATE processors.

5 Related Work

There are a number of machine learning tools available either as open source packages, or with source code for research purposes. To our knowledge Edlin is the only framework that represents linear models in a uniform fashion, and is also the only learning framework that prioritizes code readability. The NLTK [3, 4] (natural language toolkit) emphasizes code readability but focuses on natural language processing rather than learning.

MALLET [11] is a Java toolkit for machine learning. MALLET implements most of the learning algorithms available in Edlin in addition to many others. The exceptions are perceptron and MIRA, which are available as a separate MALLET-compatible package called StructLearn [12, 6]. For sequence data, one of MALLET's main strengths is a way to easily create predicate functions (f_1 in the notation of Section 2). Edlin does not have sophisticated code for feature engineering, and in our experiments we used GATE to generate features. MALLET also contains a very general implementation of CRFs that allows linear-chain models with varying order n Markov properties. These enhancements lead to a larger and hence harder to read code-base. For example the CRF model implementation in MALLET comprises 1513 lines of code compared to 56 for Edlin's simplistic implementation.[3] Note that the authors actively recommend MALLET in particular for CRFs, however it serves a different need than Edlin. While MALLET is very general and easy to use, Edlin is very simple and easy to understand.

LingPipe[2] is a Java toolkit for linguistic analysis of free text. The framework provides tools for classification, sequence tagging, clustering and a variety of problem-specific tasks such as spelling correction, word segmentation named entity normalization and parsing for biomedical text among others. Some trained models are provided, but it is possible to train a new models for new tasks and data. The software is available along with source code. We did not investigate source code complexity due to time constraints, but the full featured nature of the software and its marketing to enterprise customers suggests that its focus is on stability, and scalability rather than code simplicity and readability.

[3] Counted with *cloc* `http://cloc.sourceforge.net/`

Weka [18] is a widely used framework developed at the University of Waikato in New Zealand and comprises a collection of learning algorithms for classification, clustering, feature selection, and visualizations. Weka includes a very friendly graphical user interface, and is targeted largely at researchers in the sciences or social sciences who would like to experiment with different algorithms to analyze their data. Weka does not contain code for structured learning and is more suitable for use as a versatile black box than for reading and modifying source code. For example Weka's perceptron algorithm is implemented in 600 lines of code compared to 38 for Edlin. By contrast Weka has a very good graphical user interface and allows visualization not implemented in Edlin. GATE integration allows some visualizations and evaluation for Edlin, but specialized only for text.

ABNER [16] is a tool for processing natural language text aimed at the biomedical domain. ABNER is widely used for annotation of biomedical named entities such as genes and gene products. It contains a CRF implementation and a graphical user interface for visualization and modification of annotations, in addition to domain specific tokenizers and sentence segmenters. BioTagger [14] is a different tool for named entity recognition in biomedical text also using linear-chain CRFs. It has been applied to genes/gene products [14], malignancy mentions [10] and genomic variations in the oncology domain [15].

6 Conclusions and Future Work

We have presented a linear modeling toolkit of implementations written specifically for readability. We described the toolkit's layout, learning algorithms and an application we have found it useful for. The main goal of Edlin is to be easy to read and modify and we have used the toolkit in teaching a Master's level class. While we have not performed a scientific evaluation, initial feedback from students has been positive and at least one fellow researcher commented that he liked the organization and simplicity of the code.

Future work includes the implementation of maximal margin learning (i.e. support vector machines) and further improvements to the integration between Edlin and GATE. Finally, we intend to improve the implementation of the optimization algorithms to improve training run-time for maximum entropy models and CRFs.

References

[1] OpenNLP. `http://opennlp.sourceforge.net`, 2009.

[2] Alias-i. LingPipe. `http://alias-i.com/lingpipe`, 2008. (accessed 2008-04-20).

[3] S. Bird and E. Loper. Nltk: The natural language toolkit. In *Proceedings ACL*. ACL, 2004.

[4] S. Bird and E. Loper. Natural language toolkit. `http://www.nltk.org/`, 2008.

[5] T. G. O. Consortium. Gene ontology: tool for the unification of biology. *Nature Genetics*, 25(1):25–29, 2000.

[6] K. Crammer, R. McDonald, and F. Pereira. Scalable large-margin online learning for structured classification. Department of Computer and Information Science, University of Pennsylvania, 2005.

[7] H. Cunningham. GATE – general architecture for text engineering. `http://gate.ac.uk/`.

[8] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics*, 2002.

[9] G. Georgiev, K. Ganchev, V. Momchev, D. Peychev, P. Nakov, and A. Roberts. Tunable domain-independent event extraction in the mira framework. In *Proceedings of BioNLP*. ACL, June 2009.

[10] Y. Jin, R. McDonald, K. Lerman, M. Mandel, M. Liberman, F. Pereira, R. Winters, and P. White. Identifying and extracting malignancy types in cancer literature. In *BioLink*, 2005.

[11] A. McCallum. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu, 2002.

[12] R. McDonald, K. Crammer, K. Ganchev, S. P. Bachoti, and M. Dredze. Penn StructLearn. `http://www.seas.upenn.edu/~strctlrn/StructLearn/StructLearn.html`.

[13] R. McDonald, K. Crammer, and F. Pereira. Online large-margin training of dependency parsers. In *Proceedings of ACL*. ACL, 2005.

[14] R. McDonald and F. Pereira. Identifying gene and protein mentions in text using conditional random fields. *BMC Bioinformatics*, (Suppl 1):S6(6), 2005.

[15] R. McDonald, R. Winters, M. Mandel, Y. Jin, P. White, and F. Pereira. An entity tagger for recognizing acquired genomic variations in cancer literature. *Journal of Bioinformatics*, 2004.

[16] B. Settles. ABNER: An open source tool for automatically tagging genes, proteins, and other entity names in text. *Bioinformatics*, 21(14):3191–3192, 2005.

[17] J. Tsujii. BioNLP'09 Shared Task on Event Extraction. `http://www-tsujii.is.s.u-tokyo.ac.jp/GENIA/SharedTask/index.html`, 2009.

[18] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2005.