ReFr: An Open-Source Reranker Framework

Daniel M. Bikel, Keith B. Hall

Google Research, New York, NY

{dbikel,kbhall}@google.com

Abstract

ReFr (`http://refr.googlecode.com`) is a software architecture for specifying, training and using reranking models, which take the n-best output of some existing system and produce new scores for each of the n hypotheses that potentially induce a different ranking, ideally yielding better results than the original system. The Reranker Framework has some special support for building discriminative language models, but can be applied to any reranking problem. The framework is designed with parallelism and scalability in mind, being able to run on any Hadoop cluster out of the box. While extremely efficient, ReFr is also quite flexible, allowing researchers to explore a wide variety of features and learning methods. ReFr has been used for building state-of-the-art discriminative LM's for both speech recognition and machine translation systems.

Index Terms: language modeling, discriminative language modeling, reranking, structured prediction

1. Introduction

Creating effective software tools for research is a tricky business. The classic tension between flexibility and efficiency arises with greater urgency. We want researchers to be able to try out many different ideas easily, but we also want them to be able to have a quick code-test-evaluate cycle.

ReFr grew out of the 2011 Johns Hopkins Summer Workshop, from the team using automatically generated confusions to synthesize training data for discriminative language models for speech and machine translation, led by Prof. Brian Roark of OHSU. That approach required tools that would scale up to training data sizes orders of magnitude larger than had previously been used to build discriminative language models, so we not only needed our training and inference to be inherently fast, but we needed to design tools with distributed computing in mind from the outset.

This paper describes the tools we have developed to solve not only the immediate research problem of exploring confusions for discriminative language modeling, but also the more general problem of reranking approaches to speech and language processing, including structured prediction. We designed ReFr to have the following properites:

- "library quality" code
- industrial strength
- academic flexibility
- easy exploration of different types of features, different update methods (*e.g.*, MIRA-style, direct loss minimization, loss-sensitive) and different learning methods (*e.g.*, perceptron-style, log-linear, kernel methods)
- modern, object-oriented design, complete with dynamic factories and dynamic composition for flexibility
- parallelizable, especially for distributed-computing environments

2. Data Format for I/O

There are two main choices when building discriminative reranking models for speech or machine translation: (a) rescore a lattice or hypergraph or (b) simply use a strict reranking approach applied to n-best lists. For ReFr, early on we decided to use (b) reranking n-best lists. The primary reasons were the flexibility this would allow us in designing features and tools. N-best lists readily allow for sentence-level features in a way that, say, lattices do not. Additionally, it is far easier to define generic schemes of passing around n-best lists than it is for designing schemes to take speech lattices as well as machine translation hypergraphs or other, problem-specific data types.

ReFr is meant to be flexible enough to allow for a variety of data sources. In order to avoid the need for overly complex data formats, we have chosen to adopt a formalism which allows one to augment the input format, allowing for flexible feature extraction and data manipulation/analysis. We opted to use a data format which mirrors the data-structures that are used internally for training. The Google protocol buffers[1] provide a programming-language independent specification framework to define data formats. The protocol buffers specification language is used by the protocol buffer tools to generate source-code for serializing and deserializing the data stored in the format. Code is generated to allow for native programming-language encapsulation of the data. For example, in C++ each item of data is stored in an object based on a object oriented data specification (a C++ *class*) allowing for access to the data.[1]

3. Core learning framework

Consider Algorithm 1, which describes the training procedure for a generic online-learning algorithm. Each training example e_i comprises a set of *candidate hypotheses*, each of which is projected via some function Φ into a *feature space*, $\mathbb{R}^F$. We typically think of Φ as being a suite of *feature functions*, one per dimension. The model itself is defined as a weight vector in this space, w. Decoding, or inference, is carried out simply by taking the dot product of the model and a test instance. More generally, any kernel function $\mathcal{K}$ may be used. The training procedure iterates over the training data T—each iteration is called an *epoch*—until the NEEDTOKEEPTRAINING() predicate returns `false`. Often, such a predicate is based on the average loss of the current model on some held-out development data D, which is the purpose of the EVALUATE(D) line in the TRAIN(T) procedure.

[1] For the 2011 Johns Hopkins Workshop, we were targeting multiple tasks (ASR and MT), and so our toolkit provides a means to convert from two types of text-based n-best formats, one the output of an ASR system, the other the output of an MT system. These conversion tools are not only useful in their own right, but serve as example implementations for any developer converting from their own, proprietary format to the Google Protocol Buffer format used by ReFr.

Algorithm 1 Training algorithm for online-learning reranking models.

Let $e_i = \{c_1, \ldots, c_k\}$ be a *training example*, where each c_j is a *candidate hypothesis*.
Similarly, let $d_i = \{c_1, \ldots, c_k\}$ be a held-out development data example, also consisting of k candidate hypotheses.
Finally, let $\mathcal{K}$ be a *kernel function*.

procedure TRAIN($T = \{e_1, \ldots, e_n\}$, $D = \{d_1, \ldots, d_m\}$)
  while NEEDTOKEEPTRAINING() **do**
    TRAINONEEPOCH(T)
    EVALUATE(D)
  end while
end procedure

procedure TRAINONEEPOCH(T)
  foreach training example e_i **do**
    SCORECANDIDATES(e_i)
    if NEEDTOUPDATE() **then**
      UPDATE()
    end if
  end for
end procedure

procedure SCORECANDIDATES(e_i)
  foreach candidate hypothesis $c_j \in e_i$ **do**
    c_j.score $\leftarrow K(w_t, c_j)$
  end for
end procedure

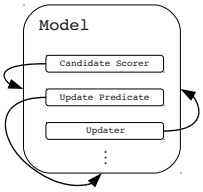

Figure 1: A pictorial view of how a `Model` wraps instances of other interfaces that specify the predicates and functions needed to carry out model training.

For the basic perceptron, the model starts out at time step 0 as the zero vector; that is, $w_o = \vec{0}$. The update is

$$w_{t+1} = w_t + R_t \left[\Phi\left(y_{\text{oracle}}\left(e_i\right)\right) - \Phi\left(\hat{y}\left(e_i\right)\right) \right], \qquad (1)$$

where y_{oracle} is a function that picks out the hypothesis towards which we want to bias our model, $\hat{y}$ is a function that picks out the candidate hypothesis we want to bias our model against and R_t is a *learning rate* or *step size*. Most often, y_{oracle} is defined to pick the hypothesis with the lowest loss relative to some gold-standard truth, and $\hat{y}$ is defined to pick the candidate hypothesis that scores highest under the current model w_t.

Most of the variations of this basic learning method involve finding different ways of defining R_t, Φ, y_{oracle} and $\hat{y}$, along with the various procedures and predicates shown in Algorithm 1. Therefore, we would like our Reranker Framework to make it easy for the researcher to define these various functions, as well as to specify which ones to use at run-time.

ReFr defines a `Model` interface with *virtual methods* for all of the functions shown in Algorithm 1. To avoid the exponential blow-up of overriding different combinations of these methods, ReFr also employs *dynamic composition.* That is, we keep the idea of a `Model` interface, but additionally have each `Model` instance wrap a set of predicate/manipulator objects, each of which itself conforms to an interface. Figure 1 shows a pictorial representation of this scheme.

As we discussed above, we employ dynamic composition to avoid defining a new subclass of `Model` every time we wish to explore a new combination of learning method functions. To do this, ReFr includes a very lightweight and yet powerful interpreter for a language that allows for assignment statements for primitives, vectors of primitives, Factory-constructible objects and vectors of Factory-constructible objects. Figure 2 shows an example ReFr configuration file. The syntax is intentionally very similar to that of C++. This lightweight language provides a flexible mechanism by which to specify how feature extraction, training and inference shall occur.

```
model_file = "my_model_file"; // model output file
model =
  PerceptronModel(
    name("my model"),
    score_comparator(DirectLossScoreComparator()));
exec_feature_extractor =
  ExecutiveFeatureExtractorImpl(
    feature_extractors({NgramFeatureExtractor(n(2)),
                        RankFeatureExtractor()});
training_efe = exec_feature_extractor;
dev_efe = exec_feature_extractor;
training_files = {"training1.gz", "training2.gz"};
devtest_files = {"dev1.gz", "dev2.gz"};
```

Figure 2: An example ReFr configuration file, read by its `Interpreter` class.

4. Cluster-based distributed training

As Algorithm 1 shows, the basic perceptron algorithm involves "online" updating, and thus it is possible to read in each training example from file each time it is needed, only keeping the model's parameters persistently in memory. The Reranker Framework allows both the memory-intensive way of training as well as this "streaming mode" version of training, essential for distributed learning.

The structured perceptron [2] and it's variants have proven to be effective in supervised, discriminative language modeling work [3]. We have centered the development of our open-source discriminative learning toolkit around perceptron-style algorithms, which are, by definition, online learning algorithms. Identifying the optimal solution for a distributed online optimization algorithm is still an open research question. We borrow from our previous work on distributed perceptron training in [4, 5] and use the *Iterative Parameter Mixtures* algorithm for distributed computation. The Reranker Framework makes it easy to switch between single processor and distributed training, which uses the Hadoop implementation of MapReduce [6].

5. Demo Plan

Our demo will consist of a walk-through of all ReFr's features, followed by a hands-on demonstration of how easy it is to implement a new class of features for the reranker based on the rank of each candidate hypothesis. We will also show how easy it is to integrate that new class of features into training and inference. We will then demonstrate the ease with which one can use the API and the interpreted configuration language to alter the training algorithm. Finally, we will demonstrate the simple way that a user can switch from single processor training to large-scale distributed training.

6. Acknowledgements

The authors would like to thank Prof. Brian Roark of Oregon Health and Science University for leading a fantastic team at the 2011 Johns Hopkins Workshop, and we would also like to thank all of our teammates, especially Prof. Izhak Shafran of OHSU and Ph.D. candidate Maider Lehr, who are actively working with and helping us improve ReFr.

7. References

[1] Google, "Protocol buffers," http://code.google.com/apis/protocolbuffers/.

[2] M. Collins, "Discriminative training methods for hidden Markov models: Theory and experiments with perceptron algorithms," in *Proc. EMNLP*, 2002, pp. 1–8.

[3] B. Roark, M. Saraçlar, and M. Collins, "Discriminative n-gram language modeling," *Computer Speech and Language*, vol. 21, no. 2, pp. 373 – 392, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0885230806000271

[4] R. McDonald, K. Hall, and G. Mann, "Distributed training strategies for the structured perceptron," in *HLT-NAACL*, 2010.

[5] K. Hall, S. Gilpin, and G. Mann, "Mapreduce/bigtable for distributed optimization," in *NIPS Workshop on Leaning on Cores, Clusters, and Clouds*, 2010.

[6] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *CACM*, vol. 51:1, 2008.