11-711: Algorithms for NLP
Assignment 3: Discriminative Reranking

Due Friday, November 10 at 11:59pm

Collaboration Policy

You are allowed to discuss the assignment with other students and collaborate on developing algorithms at a high level. However, your writeup and all of the code you submit must be entirely your own.

Setup

As usual you will need:

1. `assign_rerank.tar.gz`
2. `data_rerank.tar.gz`

For features, you will find it useful to consult Charniak and Johnson 2005 "Coarse-to-fine n-best parsing and MaxEnt discriminative reranking," Johnson and Ural 2010 "Reranking the Berkeley and Brown Parsers", and/or Hall et al. 2014 "Less Grammar, More Features."

For learning, you might consult Shalev-Shwartz et al. 2007 "Pegasos: Primal Estimated sub-GrAdient SOlver for SVM" or Kummerfeld et al. 2015 "An Empirical Analysis of Optimization for Max-Margin NLP."

Preliminaries

In this assignment you will be implementing a parsing reranker trained with two different learning algorithms of your choice.

Data You are given 20-best lists extracted from a the Penn treebank (*.20best.gz) as well as train/dev/test trees (*.mrg); these are the same data splits and same trees as in Project 3 but now one per line in a single file each. The base parser is a simplified version of the Berkeley parser (only 4 latent categories for each nonterminal) that gets 84.12 on the development set (length 40). An oracle reranker (the best possible tree from each k-best list) would get 93.01 on the development set; note that the gold tree is not always in our k-best list. All trees in this project are unbinarized.

`-maxTrainLength` and `-maxTestLength` are set to 40 by default; in addition, you can use `-maxTrainTrees` and `-maxTestTrees` to explicitly cap the number of train and test trees loaded. You can also use `-kbestLen` to load shorter k-best lists.

Interface `ParsingRerankerFactory.trainParserReranker` takes an iterator over training examples, k-best lists paired with gold trees (which aren't necessarily in the k-best lists themselves!) and returns a `ParsingReranker`, which takes a k-best list and returns the best tree. Note that you are presented with an iterator over training examples so that you can choose to do feature extraction and any other necessary computations on the fly and discard the trees as you go (which will save a substantial amount of memory). However, we're also running this assignment with a higher memory cap of 10 GB (see the Submission information for exact commands).

You have two main tasks in this project:

1. Extract features from trees in k-best lists
2. Implement at least two training procedures to learn weights for features

Feature Extraction

Each input and output pair (k-best list and choice of a particular tree) must be converted into a feature vector in real space. `SimpleFeatureExtractor` implements a simple feature extractor that extracts an indicator of the position of the tree in the k-best list (useful for learning to prefer the first tree, since it's often the best in the absence of other information) and indicators over rules that fire (which might be non-binary; such features are non-local for binarized trees). This basic feature set actually does worse than the 1-best baseline: 83.54 on dev/40 with our implementation of the perceptron. These features are mostly meant to serve as an example of how feature extraction should look.

Each feature in the baseline extractor is specified by a String, which conjoins of properties of the input sentence and output parse tree, e.g. the string "`NP -> DT NN`" would be an indicator of the presence of a given rule in the tree. Other lightweight classes can be used for this purpose (though you'll want to remember to implement `equals` and `hashCode`). Whatever form your indicators take, you'll want to follow the baseline's lead and use an `Indexer` to index them. The baseline returns features as indicators in an integer array, with feature multiplicity handled by duplicates. If you want to use real-valued features, you'll need to extend the code to produce those.

Several helper functions are provided to assist with feature extraction; these are briefly described in `SimpleFeatureExtractor`. You can use `AnchoredTree.fromTree` to convert the recursive `Tree` structure to an immutable `AnchoredTree` where each node has an associated start and end index in the sentence. `AnchoredTree.toSubtreeList` will give you all anchored subtrees, which is useful for firing features over anchored rule productions (lots of interesting and diverse features can be fired in this way). If you just want to iterate over labeled spans, `Tree.toConstituentList` returns a list of type `Constituent`, where each element stores a label and the corresponding span of the sentence. Finally, `SurfaceHeadFinder` can take a sequence of preterminals and return its syntactic head, which is useful for firing some of the features specified in Charniak and Johnson.

In this project you will be dealing with lots of features and long weight vectors; you might easily have a hundred thousand or a million unique features. Computing, storing, and handling these features must be carefully thought through in order for your code to be efficient. See Implementation Tips below for more about useful data structures and concepts along these axes.

Training

You have several possible options for how to learn your model. We are asking that you implement at least two learning algorithms. Implementing more sophisticated algorithms and deeper analysis of their behavior will both lead to higher grades.

Maximum Entropy One option for learning is a conditional likelihood model where you're optimizing $\sum_{i=1}^{n} \log p_\theta(y_i^*|x_i) - \lambda\|w\|_2$, where y_i^* is the best tree for sentence x_i. This objective is differentiable, so you can either optimize it with stochastic gradient (pay attention to your batch size!) or in a batch manner with LBFGS.[1] We have provided an implementation of LBFGS in `edu.berkeley.nlp.math.LBFGSMinimizer`. You provide a differentiable function, which should return objective values and gradients for the training set, and it will do the rest. Be sure to set up your objective so that you'll be minimizing; for historical reasons, most optimization problems are formulated as minimizations rather than maximizations. However you optimize your objective, you should see your weight vector converge to something close to the global optimum.

An additional improvement you might explore is softmax-margin following Gimpel and Smith 2010 "Softmax-Margin Training for Structured Log-Linear Models."

Perceptron You might also try the perceptron algorithm as described in lecture. If you have trouble debugging, try making a toy dataset with separable data and make sure perceptron converges in that case. Note that the actual parsing data may or may not be separable (depending on how many features you extract, and of what kind), and in any case running the perceptron until it actually separates the data may result in overfitting. Try varying numbers of iterations; this is regularization via early stopping. Perceptron can also be improved by not simply taking the final weight vector, but by averaging many weight vectors from various points throughout learning, so you might experiment with this as well.

[1] A quasi-Newton method for optimizing convex functions: http://en.wikipedia.org/wiki/Limited-memory_BFGS

Margin You have at least two choices for optimizing the SVM objective:

1. Primal SVM sub-gradient descent: optimize the primal SVM objective using a sub-gradient descent method like Adagrad (Duchi et al. 2011)
2. Dual SVM coordinate descent: optimize the dual SVM objective using a coordinate descent method like SMO (Platt 1998)

An implementation of primal SVM sub-gradient descent is provided for you: `PrimalSubgradientSVMLearner`. It uses adagrad, which is also provided. You are free to use this code, though you will have to implement the `LossAugmentedLinearModel` interface to hook up your specific re-ranking problem. You should be able to munge your code into this abstraction without too much trouble. `PrimalSubgradientSVMLearner` and `LossAugmentedLinearModel` are defined using Java generics so that you can implement the interface in terms of a generic datum of type `D` as you see fit.

Notes: (1) `D` will need to contain both the input k-best list and the gold tree since you'll need it to compute the loss-augmented decode. (2) The weights vector `IntCounter` that's handed to the `getLossAugmentedUpdateBundle` method (the one you'll have to implement) is not a normal `IntCounter`. It's a special `IntCounter` that lazily computes the results of the last step of adagrad. This means that many standard `IntCounter` methods are not implemented. See `AdagradMinimizer.LazyAdaGradResult` to check which ones you can use. If you implement things correctly, for each training instance (or training batch) adagrad will only have to touch weight indices corresponding to the non-zero features on that instance (or batch). This can make things a lot faster, though it's not strictly necessary to get things to work. (3) Finally, you don't have to use `PrimalSubgradientSVMLearner` if you don't want to!

If you go the `PrimalSubgradientSVMLearner` route, you should make sure you play around with and understand the hyperparameters and the flow of the algorithm. Your analysis should include some discussion of how it behaves during learning.

You are also free to just use the provided adagrad code to optimize an objective of your choice. Note, however, that adagrad is a special sub-gradient optimizer that has L2 regularization baked into the underlying algorithm. Whatever objective you point adagrad at will implicity have an L2 regularization term added to it! Be careful not to regularize twice!

Implementation Tips

Efficiency Feature computation tends to be slow because of the heavyweight operations required to compute and create features. You can speed up your learning substantially by caching the features that you compute, so that they only need to be computed on the first iteration through the training data. If you make n passes through the training data, you might easily see an almost n-fold speedup as a result.

You'll also want to think about space efficiency of your features. Do you want your features to have associated values, or are you just going to use indicators, which implicitly take the value of 1?

(Remember, indicator features can be used for real-valued quantities by bucketing them.) Either way, you will want a sparse representation of the features for each example, since of your million total features only a few hundred at most will fire on a particular tree. You will want to use arrays of integers and doubles primarily, and avoid any additional object overhead from instantiating an object for each feature token.

Features Charniak and Johnson (2005), Johnson and Ural (2010), and Hall et al. (2014) are good sources of features for reranking and the latter paper provides some qualitative analysis of which ones might be useful. You certainly do not need to implement every feature from those papers to get some nice gains, since you're reranking a much weaker baseline parser. Their feature templates are quite underspecified, so part of your job will be to interpret and implement them. Think about the effects they're capturing and try to come up with something that works well for you.

By no means limit yourself to what is in those papers: we want to see what you come up with on your own as well! Although the Berkeley parser is not quite the PCFG that you implemented in Project 2, it is likely to capture many of the same linguistic effects (with latent categories capturing structural information about a symbol's context). A good place to start is by trying to address deficiencies of that parsing model.

Debugging Debugging implementations of learning algorithms can be challenging. Try printing the values of objectives (either regularized log likelihood or the SVM objective) and see that they are being correctly optimized; if not, you have a bug. Perceptron is not optimizing an objective, so it may be more difficult to debug than the other learning algorithms (in addition to being more fragile). Running your learning algorithm on synthetic toy data can also be informative.

Submission and Grading

Write-ups You should turn in a 2-3 page write-up as usual. You should clearly describe what implementation choices you made, especially what feature templates you used (be precise yet succinct; think about what diagrams and notation you might want to use), what learning algorithms you implemented, and how they compare. As always, report your performance on the various relevant metrics and provide error analysis.

We're giving you more freedom on this project, so use your judgment and carve out a project that you deem sufficient and which you think gives you interesting things to say in the writeup.

Submission You will submit `assign_rerank-submit.jar` and a PDF of your writeup to an online system. We will sanity-check with the following command:

```
java -cp assign_rerank.jar:assign_rerank-submit.jar -server -mx300m
  edu.berkeley.nlp.assignments.rerank.ParsingRerankerTester
  -path path/to/data -rerankerType AWESOME -sanityCheck
```

We will grade with the following command:

```
java -cp assign_rerank.jar:assign_rerank-submit.jar -server -mx10000m
  edu.berkeley.nlp.assignments.rerank.ParsingRerankerTester
  -path path/to/data -rerankerType AWESOME -test
```

Grading Training and testing on the development set with length 40, our reference implementation gets around 86.2 F_1 using perceptron and 86.8 F_1 using LOGO. Strong submissions should be able to exceed this. You should at the very least aim to outperform the 1-best baseline by 1.5 F_1 with your AWESOME reranker (that's the only one we care about). Your system should run in an hour or less so that we can run all submissions in a timely fashion, but for this project, accuracy is generally more important than speed. As with past projects, implementing additional interesting techniques and doing extra analysis will result in a higher score.