Transition-based Dependency Parsing

Programming Assignment #4

Due: Friday, Dec 4th at 11:59pm

IMPORTANT NOTE #1 : You may complete this assignment individually or in pairs. *We strongly encourage collaboration*. Your submission must include a statement describing the contributions of each collaborator. See the collaboration policy on the course website: `http://web.stanford.edu/class/cs224n/grading.shtml`.

IMPORTANT NOTE #2 : Please read through this whole document before starting on the assignment. We are eager to answer your questions, but please make sure that your question about section 2.3 isn't answered in section 2.4. You should also read the comments and Javadocs in the starter code files.

IMPORTANT NOTE #3 : If you have never logged into Farmshare, then please make sure you can well before the assignment deadline: `http://farmshare.stanford.edu/`.

1 Introduction

The task of dependency parsing is to determine the grammatical structure of a sentence, establishing relationships between words and their dependents. In this assignment, you are going to implement arc-eager dependency parsing, which gives near state-of-the-art results in linear time. This parser, however, needs a classifier to select the action in each configuration. For this we will use a MaxEnt classifier and you will be implementing the objective function and gradient. In the last part of the assignment you'll be designing features to maximize the classification accuracy and the attachment scores.

1.1 How to Parse this Document Quickly

This document is long. Here's how to quickly parse it and get to work: Section 2 gives the basic setup, and Sections 3, 4 and 5 describe the three parts of the assignment, which are *MaxEnt Classifier*, *Arc-Eager Parsing* and *Feature Engineering*. Within each part are three subsections:

1. The first part is the theory behind what we are doing.
2. The second will walk you through the implementation.
3. Finally, the third part is a checkpoint to ensure that the implementation meets the requirement before you move on to the next part.

Then, there are several sections marked **Your Job** that tell you what to do. Complete them, write a report, and submit your code on afs and your writeup on Gradescope. This assignment involves a substantial amount of thinking and coding. *Don't wait until the last day to start this assignment*.

1.2 **Important Setup for Farmshare**

We encourage you to complete this assignment on Farmshare (`http://farmshare.stanford.edu`). This assignment is written for the bash shell, which may not be the default on Farmshare. If you aren't running bash, then you can switch to it by typing `bash`.[1]

Farmshare was upgraded over the summer. The new Ubuntu installation doesn't set a default file encoding. This leads to unpredictable encoding problems for our data files, which contain Unicode. Make sure to set this environment variable each time you login, either via your `.bash_profile` or manually:

```
export LANG=en_US.utf8
```

[1] To set bash as your default, see: `http://answers.stanford.edu/solution/can-i-change-default-shell-farmshare-host`.

2 Setup

The starter code can be found in `/afs/ir/class/cs224n/cs224n-pa/pa4-2/`. Copy over the starter code to your local directory and make sure you can compile it without errors

```
cd
mkdir -p cs224n/pa4-2
cd cs224n/pa4-2
cp -r /afs/ir/class/cs224n/cs224n-pa/pa4-2/java .
cd java
ant
```

The data for this assignment is located in `/afs/ir/class/cs224n/data/pa4-2/`. If you are not using farmshare, you can copy this data over to your local directory and modify the `trainPath` and `testPath` arguments.

The most important class for this assignment is `cs224n.assignments.DependencyParserTester`. The `main()` method of this class takes several command line options. You can try running it with

```
    java -Xmx500m -cp "classes/" cs224n.assignments.DependencyParserTester \
    -trainPath /afs/ir.stanford.edu/class/cs224n/data/pa4-2/ud-treebanks-v1.1/UD_English/en-ud-train.conllu \
    -testPath /afs/ir.stanford.edu/class/cs224n/data/pa4-2/ud-treebanks-v1.1/UD_English/en-ud-dev.conllu \
    -parser cs224n.assignments.ArcEagerParser
```

Currently, `ArcEagerParser` is not implemented yet, so you will see `UAS = 0.0` after running this command. `DependencyParserTester` takes several command line parameters. These include

- `-trainPath`: path to the training dataset.
- `-testPath`: path to evaluation dataset.
- `-parser` (default is `cs224n.assignments.ArcEagerParser`): the parser class.
- `-typed` (`true` or `false`, default is `false`): if present with the value true, uses typed dependencies for training and also evaluates Labeled Accuracy Scores (LAS).
- `-serializationPath`: if serialized weights exist at the location, skips training and evaluates on test set using those weights. Otherwise, trains on the train set, saves weights at the location for future use and tests on the test set.
- `-trainSentences` $[1, \text{MaxNumberOfSentences}]$: uses fewer training examples for debugging purpose with this argument. If it is not set, it uses the full training set.

3 MaxEnt Classifier

3.1 Theory

A Maximum Entropy (MaxEnt) classifier is a feature-based linear classifier which chooses parameters λ to maximize the conditional likelihood of the data. The log conditional likelihood of i.i.d data (C, D) according to MaxEnt model is a function of the data and parameters λ.

$$\log P(C \mid D, \lambda) = \sum_{(c,d)\in(C,D)} \log \frac{\exp\left(\sum_i \lambda_i f_i(c,d)\right)}{\sum_{c'} \exp\left(\sum_i \lambda_i f_i(c,d)\right)}$$

For training, we use the L-BFGS optimization algorithm, which needs the first order derivative of the objective funtion. The class notes explain the inner workings and also the mathematical formulation which you will be implementing. Also, make sure to read up about the smoothing term from the class notes on smoothing (not expressed in the above formula) and implement it.

3.2 Your Job: Implementation

Once you have copied the starter code, you are ready to implement the core elements of MaxEnt Classifier. You will find an extensive code base (courtesy: Dan Klein) which efficiently implements a MaxEnt classifier tailored for NLP. Your task is to:

- Implement the MaxEnt objective and gradient in the `calculate` function inside the class `cs224n.assignments.MaxEntObjectiveFunction`. Note that the gradient should be an array of doubles, because the gradient is taken with respect to each parameter separately. Also, remember to flip signs of objective and gradient as we use a minimizer and we want to minimize the *negative* log likelihood.
- Implement the `getLogProbabilities` function in class `MaximumEntropyClassifier`.

To understand how to extract features counts and labels, we would recommend looking at the class `EncodedDatum` and the explanation in the comments. Since the optimizer code expects a one-dimensional input for parameters(λ), we linearize the two-dimensional feature-label space in the class `IndexLinearizer`. This provides methods to go from linear indexing to two-dimensional indexing and vice-versa.

3.3 Checkpoint

Once you have implemented the MaxEnt Classifier, you can test it using the provided `NumericalGradientChecker` class. Note that you should not change anything in this class as we will be using this class to test your implementation for correctness and your changes would thus be overwritten.

The idea behind numerical gradient checks is simple. We approximate the gradient at a point, using values of the objective at neighboring points along all dimensions and compare it with the analytical gradient obtained by your implementation.

We test gradient checks for both the regularized and unregularized case. Your implementation should pass both the gradient checks upon running the `NumericalGradientChecker` (You should see `Passed gradient checks` upon running the following line of code.). You can use the provided bash script `gradient_check.sh` or equivalently the following command.

```
java -cp classes cs224n.assignments.NumericalGradientChecker
```

Note - The gradient checker does not check the correctness of `getLogProbabilities` function nor does it check if the objective in `calculate` function is correct. It simply tests if the derivative corresponds to the objective.

4 Arc-Eager Dependency Parsing

In this section you will implement an Arc-Eager parser based on the lectures and as explained in the videos.

4.1 A word about data and evaluation

We shall be using the English portion of Universal Dependencies v1.1 to train and evaluate our parser. It is highly recommended to take a closer look at what the data represents as it will help in the next section to develop features. For this section, however, it is sufficient to know that each sentence in the data is broken down into tokens. Each token is labelled with its head and dependency relation. Each token also has a few linguistic features.

We provide code to read these files and load the data into objects. `readData` function in `CoNLLUReader` class takes in a path and returns the parsed data. For simplicity, each sentence is represented simply as a list of `TokenAnnotation`. A `TokenAnnotation` contains the `id` of the word along with `TokenFeatures` and `TokenLabel`. For training you'll be given `TokenAnnotations`. However for testing, you'll have to return a `TokenLabel` for each `TokenFeature`. Note that for the untyped task, the `deprel` is simply an empty string.

We will be using Unlabeled Attachment score (UAS) and Labeled attachment scores (LAS) as taught in class for evaluation. You are provided with train and dev sets for training and hyperparameter tuning (as used in the command in Section 2)

4.2 Reading and references

We expect you to implement "arc-eager" variant of dependency parsing as taught in class. Refer to lecture notes for complete understanding.

Let σ be the stack, β be the buffer and A be the set of dependency relations. We start with $\sigma = [\texttt{ROOT}]$, the entire sentence on the buffer $\beta = [w_1, \ldots, w_n]$ and $A = \emptyset$. At a given step with configuration $C = (\sigma, \beta, A)$, we look at the parser configuration and decide on the next action to be performed (the classifier comes handy in here). Four types of actions are defined:

- **Left-Arc**$_r$: We pop the top item from the stack after ensuring that it not the root and it does not have a head ($\nexists r'(w_k, w_i) \in A, w_i \neq \texttt{ROOT}$), add it as a modifier to the item in front of the buffer:
$$(\sigma \mid w_i, w_j \mid \beta, A) \longrightarrow (\sigma, w_j \mid \beta, A \cup \{r(w_j, w_i)\})$$
- **Right-Arc**$_r$: We add the front of buffer as a modifier of the top of the stack, then we remove the front of the buffer and push it onto the stack:
$$(\sigma \mid w_i, w_j \mid \beta, A) \longrightarrow (\sigma \mid w_i \mid w_j, \beta, A \cup \{r(w_i, w_j)\})$$
- **Reduce**: After verifying that the top of the stack has a head ($\exists r'(w_k, w_i) \in A$) , we pop it off the stack:
$$(\sigma \mid w_i, \beta, A) \longrightarrow (\alpha, \beta, A)$$
- **Shift**: Remove the front of the buffer and push it onto the stack:
$$(\sigma, w_i \mid \beta, A) \rightarrow (\sigma \mid w_i, \beta, A)$$

We stop when the buffer gets empty ($\beta = \emptyset$). Note that in this scheme there can be cases where a token is not assigned a head and there are clever ways of handling it. However for the numbers reported in the handout we do not assign a head.

Recall that you need to generate training instances to train the classifier and do so you need to figure out the gold actions which were taken by the parser. This was not covered in class but is a fun problem in itself. To help you think about it here are a few clues:

- If a token has a left arc, the only time the arc can be formed is when the token is on top of buffer.
- Once a token is shifted to stack, it is never shifted to buffer.
- A token can form a right arc only when it is on top of stack.

4.3 Your Job: Implementation

`DependencyParser` is an interface which is implemented by `ArcEagerParser`. You need to implement two methods: `train` and `parse`. You are provided with scaffolding code as guide. You can choose to ignore it and start from scratch. In that case skip the section titled scaffolding code.

4.3.1 Essential functions

- The `train` method takes in a list of sentences, where each sentence is a list of `TokenAnnotation`. Your job is to first determine an action sequence given the dependency tree. Then, use this action sequence to generate a labeled set of training instances to train the classifier.
- The `parse` method takes in a sentence and generates the best parse based on the Arc-Eager scheme using the classifier trained in the `train` method. If the parser action entails creation of an arc, store it as a `TokenLabel`. Iterate until the buffer is empty and all arcs have been generated.

4.3.2 Scaffolding code

We need labeled data for training. As a guide, you are encouraged to implement the `extractLabeledDatum` function which takes in a sentence and returns a list of `LabeledDatum` with `string` features and labels. You would need to determine the action sequence given the gold dependencies. To do so you would need to think how the dependencies dictate the next action which needs to be taken given the current parser configuration.

To help you with that you can use the `ParserAction` and `ParserSnapshot` classes. The function `applyLabelledAction` in `ParserSnapshot` class applies a given action to the current configuration and is at the core of the Arc-Eager method.

In the given code we place the stack, buffer, and the dependencies in the snapshot. Feel free to add any other information. Also remember that you need to check for certain preconditions before an action can be applied. You are provided with function `canApplyAction`, which you can fill in and check before any transitions are taken.

However a sequence of parser configurations is not enough. You need to extract features based on the snapshots. You are given a few (really few) basic features in the `extractFeatures` function in `ParserSnapshot` class.

We recommend that you first implement the `ParserSnapshot` class completely. For now it is sufficient to leave `extractFeatures` untouched for debugging. In the next section you will implement non-trivial features to achieve desired performance. Then implement the `extractLabeledDatum` function and use the provided `checkLabelExtraction` function which prints out the labeled training data generated using the functions you wrote. This pipeline is complete in itself and you should be able to debug your implementation from end-to-end.

4.4 Checkpoint

Using the given baseline features, you should be able to finish the untyped task (no `typed` flag) fairly quickly and get a UAS of around 60 in 100 iterations with a `sigma` of 100. These scores are provided just for reference. Use the command given in section 2 to run the parser. You are also given a bash script `run.sh` to help you with it. Once you achieve this score, move on to the next section.

5 Feature Engineering

In this section you will implement the `extractFeatures` method (or its equivalent if you chose to ignore the scaffolding code).

5.1 Reading and References

This is the creative part of this assignment where you get to design features which help you classify the next action to be taken and the type of dependency (if an arc is created). As has been the running theme throughout this class, along with thinking about which features would help you characterize a specific action, you should also think of features which help you reject an action.

A good place to start off would be the `TokenFeatures` class. This contains the token specific features and is provided to you for free as part of the dataset. However simply looking at the individual tokens and their features will only get you so far.

An important set of features can be derived from pairwise combinations of indicator features. An example of such a feature would be $[postag(head(\texttt{stack})) = \mathrm{DET} \wedge postag(head(\texttt{buffer})) = \mathrm{NOUN}]$. You are not limited to postags. However, remember that we are dealing with feature templates and their number can blow up very fast for pairwise features, requiring large amounts of memory and slowing down the training time.

Another class of features is derived from looking at the established dependencies. The distance between the top of stack and buffer is also very indicative. For a detailed explanation and more features you can refer to Transition-based Dependency Parsing with Rich Non-local Features by Zhang and Nivre.

5.2 Your Job: Implementation

As is the norm in most of NLP, we will be working with feature templates and indicator features. We would encourage you to take this paradigm a step forward and write a simple parser which takes in a feature template as a string and generates a feature. The scheme in which features are represented in the above mentioned paper is a good starting point. However, it would not be graded and is for you to be able to iterate faster through different features.

Your job is to implement `extractFeatures` method and return a list of strings as the feature set. Note that you should only return the features which are fired and not all possible features.

A word on memory usage: We expect you to hit train for the typed task as well and you can add features to that effect. However, note that, now, we are preditcing for a lot more labels and the memory usage is proportional to it. You are allowed to use up to 10 Gigabytes of memory for training.

5.3 Checkpoint

With the improved set of features, we expect your UAS on English language to exceed 80.0 for the untyped task and LAS to exceed 75.0 for the typed task.

6 Administrative Information

6.1 Grading Criteria

Please write a report describing the following

1. Objective and gradient function for MaxEnt
2. Additions to arc-eager, handling of edge cases, and algorithm to generate labeled instances
3. Your features and the intuition behind them

Careful error analysis is essential for both the workings of the parsing algorithm and the features. There is a **hard limit of 4 pages + 1 page for extra credit**. *We may deduct 10% for each page that exceeds this limit.* For the code that you submit, please make sure that

- The code is clear and works as prescribed
- You have an efficient implementation of the MaxEnt objective and the parser
- You can produce reasonable results

6.2 Extra Credit

You may earn up to 20% extra credit on this assignment by completing one or more of the following (note that the rubric for extra credit is very flexible to encourage creativity and that each of the following are suggestions and do not correspond to the rubric):

- We only looked at English, however Universal Dependencies contains data for a vast number of languages. See if the same features work well across languages and if there are any significant differences due to the linguistic structure of different languages. A thorough error analysis is expected.
- We selected the greedy choice. However, in practice, the best algorithms perform beam search. Your job would be to implement beam search and observe an improvement in the results.
- You can consider to explore other parsing systems, e.g., `Arc-Standard`, or systems with `SWAP` operations, [2] and compare them with your implemented Arc-Eager system.
- The current implementation assumes that there is only one deterministic gold action for each configuration, which is not always the case. A clever idea is to introduce dynamic oracle, allowing the classifier to explore more transition paths during training. [3]

We will assess extra credit based on the quality of your work.

6.3 Submission Instructions

You will submit your writeup on Gradescope and your program code on afs using a script that we've prepared. To submit your program, first put your files in one directory on Farmshare. This should include all source code files, but should not include compiled class files or serialized weights files. **Put your report PDF in the root of your submission directory**. Then submit:

```
cd /path/to/my/submission/directory
/afs/ir/class/cs224n/bin/submit
```

[2]Reference: http://www.aclweb.org/anthology/P09-1040.pdf

[3]Reference: http://www.aclweb.org/anthology/Q/Q13/Q13-1033.pdf

This script will (recursively) copy everything in *your* submission directory into the official submission directory for the class. If you need to re-submit your assignment, you can run:

```
/afs/ir/class/cs224n/bin/submit -replace
```

Please ensure you retain the proper structure of your directory as copied from the starter code so that it is easy for us to run your code. Ensure your submission directory on afs is of the following pattern `dir/java/src` and not `dir/pa4-2/java/src` or `dir/src`.

6.3.1 Points to note

- We will be replacing `DependencyParserTester.java` and `NumericalGradientChecker.java` with our versions. Please do not change these.
- We will compile and run your program on Farmshare using `ant` and the `build.xml` that you provide. This way you can submit extra credit without changing the basic submission.
- If you did not complete the assignment on Farmshare, then please verify that your code compiles and runs on Farmshare before submission. *The submissions which do not compile shall be penalized*.
- If you have implemented a different parser for extra credit, make sure that the basic submission (Arc-eager parser) runs independently of this parser. Also include the command(s) as ec1.sh (,ec2.sh and so on) to run the extra credit parser using the DependencyParserTester class.
- Apart from this, if there's anything special we need to know about compiling or running your program, please include a `README` file with your submission.