Homework 3 Instructions

NLP with Representation Learning, Fall 2022
Instructor: Tal Linzen
TAs: Rui Chen, Abhinav Gupta, Will Merrill, Ying Wang

General Instructions

1. *Fill in the missing code in the provided code files, and answer any written questions in README.md.*
2. *Cite any other open-source code that helps you to complete this assignment.*
3. *As always, you should check CampusWire for any clarifications or questions.*
4. *Good luck! The assignment is due* ***December 2, 11:59 am (before the start of class)****.*

Part 0: Required Packages

Part 1 requires:

```
pip install conllu
pip install datasets
pip install spacy
python -m spacy download en_core_web_sm
```

Part 2 requires `torch`*, as is probably familiar by now. It also requires:*

```
pip install transformers
```

Part 3 requires `scipy`.

Part 1: Dependency Parsing Data, Metrics, and Baseline (20 Points)

In this section, you will load labeled dependency parses from the `universal_dependencies` dataset. This dataset contains labeled parses from many different languages and domains; we will use the `en_gum` subset of the dataset, which contains labeled dependency parses in English.

1. **Data:** We provide a `DependencyParse` class in dependency_parse.py that can be used to represent dependency parses. In eval_parser.py, implement `get_parses`, which should return a list of `DependencyParse`s from the `universal_dependences` dataset. This dataset contains sentences and labeled dependency parses in multiple languages. `get_parses(subset)` should return validation parses in the language specified by the argument `subset`. It may be helpful to refer to the documentation for from Huggingface datasets. To create a `DependencyParse` object, you can use `DependencyParse.from_huggingface_dict()`.

2. **Metrics:** In src/metrics.py, implement the `get_metrics()` function, which takes a predicted `DependencyParse` and a gold-standard label `DependencyParse` and returns two standard metrics for evaluating the quality of dependency parses:
    a. *Unlabeled attachment score* (UAS)*:* the proportion of tokens in the prediction that have the correct head (according to the gold-standard parse). Returns a value in [0, 1].
    b. *Labeled attachment score* (LAS)*:* the percentage of tokens in the predicted parse with the correct head and correct dependency label (according to the gold-standard parse). Returns a float in [0, 1].

    These metrics will be averaged by the main code across all examples in a dataset.
3. **Baseline:** In src/parsers/spacy_parser.py, implement a baseline parser that uses a spaCy model to extract dependencies from a sentence and return a DependencyPa object. To apply your method with the model `en_core_web_sm` to the English (`en_gum`) universal dependencies validation set, you can run:

    ```
    python eval_parser.py spacy
    ```

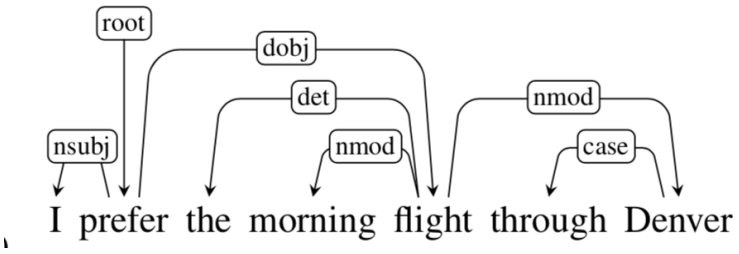
    You can specify different model and dataset names in the command line arguments to run your method on other languages. Look at the possible flags for more information. Record the four metrics for the baseline spaCy parsing method.

    Hand in: In README.md, report the UAS and LAS for your SpaCy parser on `en_gum`.

Part 2: Dependency Parsing By Finetuning BERT (40 Points)

We will now implement a method for dependency parsing by finetuning a pretrained transformer, developed in the paper *Parsing as Tagging* (Vacareanu et al., 2020). This section (Part 2) will involve data pre-processing, finetuning the model, and hyperparameter search. The next section (Part 3) will explore several ways to decode dependency parses from the model predictions.

Vacareanu et al. (2020) frame parsing as a tagging problem (i.e., predict labels at each token position). For each token, we predict two labels: the *relative position* of the token's head, as well as the *label* of the associated dependency arc. For example, in the tree below[1], so the head of "flight" (position 4) is "prefer" (position 1), so the labeled relative position of the head at position 4 is -3. The associated dependency arc label is `dobj`. In the case where the head is the root ("prefer" below), the relative position is 0, and the dependency arc label is `root`.



[1] Image credit

Your task will be to implement the following in finetune_bert.py. You may structure your code however you like, but we have some hints and suggestions in the header docstring in that file.

1. **Data Preprocessing:** Load the training set for `en_gum` and construct appropriate labels (relative positions and dependency arc labels). The relative positions should be a list of integers, and the dependency arc labels should be a list of strings. It may be useful to import or copy your code for `get_parses` from eval_parser.py.

   The number of possible relative positions is infinite, and there could potentially be relative positions that occur during validation/test time but not during training. You should handle this by defining the label vocabulary of relative positions to be those that occur during training, plus a special `unk` symbol. Any relative position unseen during training should be converted to `unk`.

   Hand in: Print the first 10 preprocessed examples to the file en_gum_10.tsv in the format `text\trel_pos\tdep_label`, where `\t` represents a tab, text is the input text, and `rel_pos` and `dep_label` are the two different label lists, serialized by calling `str()` on the Python lists.

2. **Finetuning BERT:** You should implement a model class (extending `torch.nn.Module`) that encodes the input with pretrained DistilBERT and then predicts `rel_pos` and `dep_label` using two different linear output heads. Although the number of possible relative positions is infinity in theory, we need to constrain it to a finite number because the task is framed as a tagging problem. You can use the number of unique relative positions seen in the training data.

   The DistilBERT encoder should be unfrozen, so that its weights can be updated during finetuning. The network is then trained with the following loss function.

   Let L_{rel} be the loss for relative position prediction, and L_{dep} be the loss for dependency label prediction. Let λ be a hyperparameter, and define the overall loss L:

   $$L = \lambda L_{\text{rel}} + (1 - \lambda) L_{\text{dep}}$$

   You will finetune three models with λ varying over {.25, .5, .75}. For each model, use HuggingFace to load the model and tokenizer for `distilbert-base-uncased`. Finetune on the dataset you have constructed for 3 total epochs with a batch size of 32 and learning rate of 1e-4. Clip the gradient norm to maximum norm 1.0. L_{rel} and L_{dep} should each get a cross entropy loss head, and the resulting losses should be combined according to λ.

   It will be easiest to finetune the models on a GPU, either by using HPC resources or Colab. If you finetune the models on Colab, you can either add a command-line cell running finetune_bert.py, or you can instead submit a notebook file finetune_bert.ipynb.

   Hand in: Make sure to **save final checkpoints** for each of the three models using `torch.save()`, with the filename bert-parser-λ.pt, where λ is replaced by one of {.25, .5, .75}.

Also report the validation accuracy for predicting both relative head position and the dependency arc label, for each model (that is, 2 accuracy numbers for 3 different models).

Part 3: Decoding Dependency Parses from BERT (40 Points)

In Part 2, we trained a neural net to predict relative head positions and dependency labels. Our ultimate goal, though, was to use a neural network as a dependency parser, which requires *decoding* a dependency tree from the neural network predictions. There are a variety of ways to do this, but we will focus on two:

1. **Argmax decoding:** We find the head of each token by selecting the relative head position with the highest probability. Similarly, we select the dependency label for that edge by taking the highest probability one.
2. **Maximum spanning tree (MST) decoding:** In this approach, we find the head of each token by viewing the predicted relative head positions as weighted edges in a graph over tokens, and find the maximum spanning tree over this graph.

   Hint: you can follow this post to compute maximum spanning trees as minimum spanning trees, using an adjacency matrix representation of the original input graph.

Implement both methods in `BertParser` src/parsers/bert_parser.py. If `BertParser` is constructed with `mst=True`, then use MST decoding. Otherwise, use argmax decoding.

One thing to be careful about is that DistilBERT tokenizes differently (subword tokenization) than the tokenization in the labeled parse. This makes your decoding pipeline a bit complicated. Your DistilBERT model will return a sequence of predicted head positions/labels for each subword token. You should collapse these into a sequence of predicted head positions/labels for each gold-standard token by taking the first sub-word prediction of each gold-standard token as the prediction for that token. For example, if `hello` gets split as `he ##llo`, then the predictions for `he` should be used for `hello`.

Apply each decoding method to all three finetuned model checkpoints, and record the validation UAS and LAS for each method/value of λ. You can run your method on the validation set using:

```
python eval_parser.py bert
```

Pick the best value of λ for each decoding method and evaluate on the test set with that value of λ, reporting test UAS and LAS. You can run your method on the test set using:

```
python eval_parser.py bert --test
```

Hand in: UAS/LAS on validation set (x6 total), UAS/LAS on test set (x2 total)

Extra Credit: Multilingual Dependency Parsing (10 Points)

Run the full homework pipeline (and obtain analogous output to Part 3) for dependency parsing in a language of your choice. You should select a language that has annotated dependency parses in the `universal_dependencies` dataset, and change the path to load these instead of the English parses. If you have implemented the previous parts correctly, this part should not involve significant coding, but rather, using the code you have already written with different flags, and running more experiments.

Hand in: Describe the language you pick and report UAS/LAS on validation set (x6 total), UAS/LAS on test set (x2 total), discussion of results in comparison to English