RST Project

057931354

Location of the project:

~levyrafi/courses/NLP/rst_proj

Running environment:

- Directory structure:

  There are couple subdirectories under the root of the project:

  code – the sourse files

  work – here we are running the code. The are few subdirectories:

  TRAINING

  DEV

  glove – the embedding vectors files

  doc – the location of this document

- Running command:

  ..\code\main.py –m <linear|neural> -baseline

  By default model is 'neural' and baseline flag is off

  In main.py there are several variables which determine the location of the files. These variables are already set. They can naturally be redefined, if we are not following the default directory names.

- Perl version and other libraries:

  Perl vecrion is 3.6.4

  PyTorch – 0.4.0

  nltk 3.2.5

  sklearn 0.19.1

- OS: Windows 10

- Out files:

  linear_out – running the linear model (SGDClassifier)

nearal_out – running the neural network model

baseline_out – evaluating the DEV dataset against a baseline (the most frequent relation was use).

Program flow and central units:

- Initially we are doing preprocessing to the TRAINING dataset. There are several sub-stages. The main part is the binarization of the trees – ensuring each intermediate node has exactly two children. A collection of trees is generated and is held in memory. These binarized trees will be passed to the next stages. Additionaly, we generate a table of sentences (by reading the .out files and merging lines into sentences), and splitting each EDU to a list of words and pos-tags (by calling the nltk functions word_tokenizier and pos_tags).
- Second, we generate the vocabulary. Built the vocabulary based on TRAINING data-set. Unknown words in DEV/TEST set will be mapped to the empty string. Embedding vectors for each word in the vocabulary are taken from the glove data. We use vectors of dimension 50 (glove.6B.50d.txt file).
- Third, here we start with the training part – we generate samples from the training trees. Each sample is composed of an input part and of an output part (the label): the input is defined by the **configuration** of the queue and the stack (top element in queue and the two top elements in stack), and the output part is the **action** which is reresented as a triple of fields <shift|reduce><nuc><relation>. These samples are passed to the learning model stage
- Forth, we have the learning model part. We implemented two (simplified) models:
    - Neural network model – the model suggested by authors of the article Cross-lingual RST Discourse Parsing (in short, "cross lingual"). It has three layers (the input layer, the hidden layer and the ouyput layer) and two ReLu activation functions. We sampled 200 mini batch of size 5000 each.

- Linear model. Here we used the GDClassifier with the same mini batch parameters (size and number of iterations) as in the neural network.

- Fifth, the feature extraction function – it is called by the mini batch model, and it generates vectorial features from a subset of samples.
  The features we have generated are a based on the articles "cross-lingual" and the article of Ji and Eisenstien. Specifically the following features are implemented:
  - Three first words/pos-tags and last word/pos-tag in each element (recall, there are three elements: one at the top of the queue and two other at the top of the stack).
  - Distance between the element in the top of the queue to first and the last EDU in document. The same for the top element in queue.
  - Distance between top element in queue and top element in stack (if the element in stack is compound, we take the most left nucleus EDU leaf).
  - Boolean which indicates whether the two elements in queue and and in stack belong to the same sentence.
  - Length of the elements in number of words.

  All-together number of features is 619.
- Sixth, evaluation part –
  - Preprocessing the DEV trees. This time the preprocessing, in addition to binarization, generates a **gold** directory with the serial trees files inside (to be used later by the evaluate function)
  - Parsing phase – we start with a queue loaded with all the EDUS of the document. Stack is empty. By operations of shift and reduce we eventually ended at the final configuration in which queue is empty and stack has a single element - the root of the generated tree. The aim to generate a tree similar to its respective gold tree.
    On each state, we generate a sample (the configuration of the queue and the stack; there is no output/label this time), we call

'extract features' for that sample only, and finally we call the model prediction to find the **action** (the label) with best score.

- There is an option to use a baseline, namly, not to call the model prediction, but to return the action which is most frequent.

Reults

- Neural model:

  Span F1: 0.5994
  Nuclearity F1: 0.3109
  Relation F1: 0.1799

- Linear model :

  Span F1: 0.573
  Nuclearity F1: 0.2791
  Relation F1: 0.1114

- Baseline:

  Span F1: 0.6041
  Nuclearity F1: 0.3188
  Relation F1: 0.1965

As can be seen, results are disappointing. In fact the baseline gives us the best result. It can be explained by the features quality in which they are not capturing well the relations between the EDUs, and by the models which are too simplistic. Taking pre-trained vectors which were not generated by the specific dataset can be a major drawback.

Fixing the dataset

I have added the ".out" suffix to tree file names of the form file<n>
The modified datasets is part of the submitted code and data files.

Possible crashes…

Since the pos-tags are computed dynamically during the training phase, having a new tag in the test-data, a tag which does not exist in the training data, will cause an out-of-range error. If this happen, and if possible to identify the tree which causes that problem, it would be best to add that tree to the training-data (I

should have added an unknow tag entry to which unrecognized tags would have been mapped to).