Natural Language Processing (CS60075)

Assignment 2: POS Tagging and Dependency Parsing

Deadline: 19 March, 2024 23:59 hrs

Guidelines:

- ➔ This assignment is to be done individually by each student. You should not copy any code from one another, or from any web source. We will use standard plagiarism detection tools on the submissions. Plagiarized codes will be penalized heavily.
- ➔ This assignment is meant to use Python only. You are allowed to use libraries like numpy which are needed for efficient implementation of matrix operations.
- ➔ Solutions should be uploaded via the CSE Moodle website (see course website for details). Submit one .zip file containing a folder with your codes and the pdf report. Name the compressed file the same as your roll number. Example: "*24CS60R00.zip*""
- ➔ Try writing code that is organized into different functions and add explanatory comments. Make sure your code can be easily read. Codes that cannot be understood will be penalized.

Problem:

You are to train a POS Tagger *(Task1)* and a dependency parser *(Task2)* over the UD-English-GUM dataset.

1. Given the POS tags in the train dataset, you need to calculate the transition and emission probabilities of each word and tag, and create the Viterbi matrix from scratch using these values. You will have to show classification reports on both the train and test data.
2. Given the gold-standard dependency graphs and the Oracle (set of heuristics/rules) below, you need to train a simple linear classifier to generate dependency graphs for unknown sentences. You will also have to report the UAS (Unlabelled Attachment Score) metric over the test set.

Dataset Description:

- **The data folder 'NLP2' contains two *.txt* files. You can download the dataset from https://drive.google.com/file/d/1Pc8FU-VdDP8tZIR1V0PxCmiUYFAE1-d5/view?usp=sharing.**
- This folder contains a train and a test dataset, each consisting of multiple sentences.
- The Train dataset contains 2000 sentences, while the test dataset contains 100 sentences.
- Each sentence starts with a unique # sent_id and a # text describing the text of the sentence.
- Each sentence is then already tokenized into words, and their corresponding POS tags. Each word is also annotated with its head word in the dependency graph.

Format:

sent_id = GUM_academic_librarians-2

text = Leading Dutch Librarians into DH

1	Leading	lead	VBG	0	root
2	Dutch	Dutch	JJ	3	amod
3	Librarians	librarian	NNS	1	obj
4	into	into	IN	5	case
5	DH	DH	NNP	1	obl

sent_id = …

Task 1: *(40 marks)*

The primary objective of *Task1* is to implement and understand the Viterbi Algorithm for Part-of-Speech (POS) tagging in NLP. POS tagging involves assigning a grammatical category (such as noun, verb, adjective, etc.) to each word in a given text.

Process to follow:

- First apply this algorithm only on the train dataset given in the link.
- Compute the transition probabilities, emission probabilities in 2 different functions so that they can be used multiple times while calculating the values of each cell in the Viterbi matrix.
- Calculate Viterbi algorithm on Train data.
- Now depending on the previously calculated transition and emission probability matrix of the train dataset, apply the Viterbi algorithm on the test dataset. Do not calculate these probabilities on the new test data, use the old values only.
- If you see any new word (if present) in the test data, use *Add-One Smoothing* to overcome it.
- ***Add-One Smoothing*: For emission probabilities of a sentence *S*,**
  $P(w_n|s_n) = (count(w_n|s_n) + 1)/(count(s_n) + |V|) \quad \forall w_n \in S$
  Here, *V* is the vocabulary of tokens, created from *all* un-normalized tokens in the *train set only*. Naturally, some tokens in the test set may not have been seen during training, and thus this smoothing technique gets rid of the zero probability issue. Smoothing must be done for all tokens and not just the unseen tokens.
- **For transition probabilities, it is not required, as every tag will already be seen in the train set, so no chance of new tags coming.**
- Use dynamic programming to implement the algorithm.

Algorithm to implement:

```
Let T = # of part-of-speech tags
    W = # of words in the sentence
/* Initialization Step */
for t = 1 to T
    Score(t, 1) = $Pr(Word_1 | Tag_t) * Pr(Tag_t | \varphi)$
    BackPtr(t, 1) = 0;
/* Iteration Step */
for w = 2 to W
   for t = 1 to T
      Score(t, w) = $Pr(Word_w | Tag_t) * MAX_{j=1,T}(Score(j, w-1) * Pr(Tag_t | Tag_j))$
      BackPtr(t, w) = index of j that gave the max above
/* Sequence Identification */
Seq(W) = t that maximizes Score(t,W)
for w = W -1 to 1
   Seq(w) = BackPtr(Seq(w+1),w+1)
```

Evaluation:

- Compare the *accuracy, precision, recall, and F1-score* (metrics for evaluation) of your model with the gold-standard POS tags given in the dataset. Do it for both the train and test dataset.
- **For calculating the above metrics, put the gold-standard and predicted POS tags from all tokens across all sentences into two individual lists (not nested). This means that individual sentences no longer matter during evaluation, the metrics are calculated across all the tokens in the dataset.**
- Report for *how many words* you used *Add-One Smoothing* in the test dataset.
- **You can use the SKLearn library only for evaluation purposes. You can't use NLTK, spaCy or any other NLP based libraries in your code.**
- In the report, mention if you faced any challenges faced during the implementation, and how you resolved it.
- Report if you used any assumptions in the code.

Deliverables:

- A python file/ipython notebook **"posViterbi.py"/"posViterbi.ipynb"**
- Create a report **"viterbi_report.pdf"** to record all metrics and other information (mentioned above).
- The set of predictions on train data **"viterbi_predictions_train.tsv"** and test data **"viterbi_predictions_test.tsv"**
    - Each prediction file is a tab-separated file with the format
      sent_id <TAB> token number <TAB> word <TAB> predicted POS tag

Marks Division for Task1:
Training the model and showing the predictions -> 20
Testing the model and showing the predictions -> 10
Reporting the metrics and other information -> 10

Task 2: *(60 marks)*

In this task, you will have to (i) design a dependency parser that can be trained over sentences annotated with their dependency graphs, and then (ii) use it for creating the dependency graph for unknown sentences in inference mode.

Dataset

For this task, you will use the same UD-English-GUM dataset. This dataset consists of many sentences already tokenized into words. Each word is annotated with its POS tag and its head word in the dependency graph, along with the type of dependency relation. A fictional *root* word is added as the root of the dependency graph.

Preprocessing: You may notice that some tokens in the dataset do not have the head-word relationships. You will also notice that the token id of such tokens is usually a different format, like, 15-16 or 24.1, etc. Ignore these tokens. After ignoring these tokens, you should still get a valid sentence in most cases (all tokens have head-words assigned). In case you see that some head-word relationships are still missing, you should ignore the sentence altogether.

Basic Functioning

As mentioned in Lecture slide 14, any dependency parser works by maintaining a stack S (consisting of words that have been processed), a buffer B (words that are yet to be processed) and a set of arcs A (head-word dependency relations). The state of the parser, i.e., contents of S, B and A is called the configuration C.

The parser starts with the initial configuration C_0, which consists of empty S and A, and B containing all words of the sentence. Thereon, the Arc-Eager Parsing algorithm can be used to transition from C_0 to C_n (final state, empty B). The parser makes these transitions (sequence of actions) by utilizing a classifier, which is predict the transition t_i from a set of pre-defined transitions T to take the parser from C_i to C_{i+1}, where $i \in [0, n)$. A transition t can be either of the following:

1. **LEFT-ARC: Add $top(S) \leftarrow first(B)$ to A, Pop $top(S)$**
2. **RIGHT-ARC: Add $top(S) \rightarrow first(B)$ to A, Pop $first(B)$ and Push to S**
3. REDUCE: Pop $top(S)$
4. SHIFT: Pop $first(B)$ and Push to S

Finally, the set of arcs A can be used to determine the head-word relationships (**$w_j \rightarrow w_k$ means that w_j is the head-word of w_k, whereas $w_j \leftarrow w_k$ means that w_k is the head-word of w_j**), which is the final output of the parser. *For this assignment, we are interested in only predicting only the head word for each word (the dependency relation with the head word does not need to be predicted for inference).*

Classifier and Features

In this assignment, you will consider a linear classifier L, which can be defined as a function $L: 2^C \rightarrow T$. The classifier works by taking as input a set of features (numerical values that represent the configuration). The feature vector can be created using the feature function $f(C_i, t), where\ t \in T$ (features are transition-dependent). In this case, the following conditions determine f:

1. TOP: $top(S)$ token
2. TOP.POS: POS-Tag of $top(S)$
3. **TOP.DEP: DEP-Tag for $top(S) \leftarrow head(top(S))\ or\ head(top(S)) \rightarrow top(S) \ \in A$ (i.e., only if the head of top(S) has been discovered so far)**
4. **TOP.LDEP: DEP-Tag for the left-most $w \leftarrow top(S) \ \in A$ (i.e., only if such a relationship has been discovered so far)**
5. **TOP.RDEP: DEP-Tag for the right-most $top(S) \rightarrow w \ \in A$ (i.e., only if such a relationship has been discovered so far)**
6. FIRST: $first(B)$ token
7. FIRST.POS: POS-Tag of $first(B)$
8. **FIRST.LDEP: DEP-Tag for the left-most $w \leftarrow first(B) \ \in A$ (i.e., only if such a relationship has been discovered so far)**
9. LOOK.POS: POS-Tag of $first(B - \{first(B)\})$

Each condition can be represented as a Bag-of-Words (BOW) vector. For TOP and FIRST, these vectors have the size $|V|$, where V is the vocabulary. V is created by considering the most frequent 1000 normalized tokens in the training set, which do not occur in more than 50% of training sentences.

For x.POS, the vectors have dimensionality $|P|$, where P is the set of all POS Tags. x.yDEP features can be represented by vectors of size $|R|$, where R is the set of all Dependency Relationships. The final size of the feature vector $f(C_i, t)$ is $4 \times (2|V| + 3|P| + 4|R|)$, corresponding to the 4 possible transitions $t \in T$.

Remember that this final feature vector is a binary vector.
The classifier L consists of a vector W having the same size as features described above.

Example
Suppose that following the conditions above, at a configuration C you get a binary vector [1,1,0] (This is just an illustration, in your case the vector will be of size $2|V| + 3|P| + 4|R|$.)

Then, your final feature vectors will look like:
$f(C, LA) = [1, 1, 0,\ 0, 0, 0,\ 0, 0, 0,\ 0, 0, 0]$
$f(C, RA) = [0, 0, 0,\ 1, 1, 0, 0, 0, 0,\ 0, 0, 0]$
$f(C, SH) = [0, 0, 0,\ 0, 0, 0,\ 1, 1, 0,\ 0, 0, 0]$
$f(C, RE) = [0, 0, 0,\ 0, 0, 0,\ 0, 0, 0,\ 1, 1, 0]$

Essentially, the order does not matter. But this is how you will get 4 different feature vectors for 4 different transition operations LA, RA, SH, RE. This is also how the final feature vectors are of length $4 * (2|V| + 3|P| + 4|R|)$. Each has to be multiplied with the weight vector to get a score for each transition.

Training
To train the classifier, the gold-standard dependency graph D is used. An oracle (set of heuristics/rules/conditions) is used to determine what should be the transition taken by an ideal parser given any configuration C_i. The oracle is given below:

1. **If $top(S) \leftarrow first(B) \in D$ and $top(S) \leftarrow *\ or\ * \rightarrow top(S) \notin A$ (means that head of $top(S)$ has not been discovered yet),**
   Then LEFT-ARC
2. **Else if $top(S) \rightarrow first(B) \in D$**
   Then RIGHT-ARC
3. **Else if $top(S) \leftarrow *\ or\ * \rightarrow top(S) \in A$ and $\exists w \in S,\ w \neq top(S),\ w \leftarrow first(B) \in D$ or $w \rightarrow first(B) \in D$ (means that if head of $top(S)$ has been discovered and there is a token $w \in S,\ w \neq top(S)$ which is related to $first(B)$)**
   Then REDUCE
4. **Else SHIFT**

From any given configuration C_i, the transition predicted by the classifier $t_i^* = argmax_t\, W.\ f(C_i, t)$ is compared with the gold-standard transition (given by oracle) t_i^o and the classifier weights W are updated accordingly.

$$W \leftarrow W + f(C_i, t_i^o) - f(C_i, t_i^*)$$

Thus, each individual training example is actually $< C_i, t_i^o >$. You can collect these instances by applying the oracle on *all* the sentences in the training set and their respective dependency graphs in the training set and concatenating them together.

Prediction and Evaluation
During test time, for predicting the set of transitions for a given sentence, we start with the initial configuration C_0 and apply the transitions t_i^* predicted using the *learned* weights, until we reach the terminal configuration C_n. Collecting the relationships in A give us the head-words of every word in the sentence.

Invalid Transitions

You need to remember certain points when predicting. It is possible that the classifier predicts an operation like LEFT-ARC, but your stack might be empty, making such a prediction invalid. Hence, you should use the following heuristics when predicting or testing (Training should be done normally without these heuristics).

1. Model predicts SHIFT: Can be done at all times

2. Model predicts RIGHT-ARC: Can be done at all times except when S is empty.
IF *S* is empty, then SHIFT.
ELSE RIGHT-ARC

3. Model predicts LEFT-ARC: Can be done only if S is non-empty and head(top(S)) has not been assigned.
IF S is empty, then SHIFT.
ELSE IF head(top(S)) has been assigned already, then REDUCE.
ELSE LEFT-ARC (all conditions satisfied)

4. Model predicts REDUCE: Can be done only if S is non-empty and head(top(S)) has been assigned.
IF S is empty, then SHIFT.
ELSE IF head(top(S)) has not been assigned, then LEFT-ARC.
ELSE REDUCE (all conditions satisfied)

At the end of the prediction step, for every token in the sentence, a head word is predicted. The Unlabeled Attachment Score (UAS) metric measures the fraction of tokens in a sentence for which the head word has been identified correctly. Calculate the measure for every sentence and then average to obtain the final UAS metric. **Remember that despite the heuristics above, it is possible that the head word is not predicted for some token *t*. In that case, during evaluation, you must assume that the prediction for that token *t* is WRONG.**

Implementation Summary

- Define the feature vector function, which generates the binary feature vector $f(C_i, t)$ given a configuration C_i and a transition t.
- Use the Oracle given above over the training set of UD-English-GUM to generate training instances of the form $< C_i, t_i^o >$.
- Train a classifier with an online learning setting, i.e., a single training example and prediction is used at a time, and the weights are updated. The next example is used with the updated weights, and so on. Assume that the weights are randomly initialized.
- Use the classifier over the test set (in inference mode) of UD-English-GUM to obtain the transitions (and consequently, head-word relationships) predictions.
- Evaluate the predictions using the UAS metric.
- You are allowed to use only the Numpy library for this part. No other library will be allowed.

Deliverables

- A python file/ipython notebook **"dependency.py"/"dependency.ipynb"**
- Create a report **"dependency_report.pdf"** to record all metrics and other information.

- A single model **“dependency_model_on.npy”** which is actually the weight vector.
- Similarly, a single set of predictions **“dependency_predictions_on.tsv”**, which is a tab-separated file with the format

  sent_id <TAB> token number <TAB> word <TAB> predicted root token number
- Report the UAS metrics in the report.
- If you face any challenges during implementation, mention that in the report as well.

Marks Division for Task2:

Feature vector creation -> 20
Using oracle to obtain gold-standard transitions -> 20
Weight updates, inference -> 10
Metric calculation, preprocessing and other functionalities -> 10