DEPENDENCY PARSER

STEPS to Solve Dependency Parser Task:

1. Both train and test datasets are in the form of a txt file with columns ['sent_id', 'text', 'word', 'normalized_word', 'POS_tag', 'head_word_index', 'dependency_relation'], So I have iterated over a txt file and convert it into a data frame for easy usage during the whole assignment.
2. I have to do the preprocessing in the training dataset which is removing all the rows where head_word_index is equal to '_' which is done so as to remove tokens which are not have any head-word relationships as we want should not take those words as they will ruin the results.
3. First, I have tried to gather all the information so as to create the feature vector for the given configuration of Stack, Buffer, and Arc set and the given transition.
4. For the features TOP and FIRST, I have created a new vocabulary as told in the assignment which is to take the most frequent 1000 normalized tokens that do not occur in more than 50% of sentences. This vocabulary is used for these 2 features only, for all other features I have used the whole train dataset.
5. I have created a new column having an index of the position of each word in the corresponding sentence. Also, using these index columns of the headword index which is given, I have extracted the headword for each word in the sentence for all sentences and stored it in the same data frame. This headword and dependent word are stored as arcs in a separate column in the same data frame as we will be using them to create a Gold- standard Dependency Graph.
6. I have defined token_mapping so that each token gets a unique no. then is used at the time of feature creation to get the index which is set to 1 in the feature vector corresponding to the token. This mapping is based on the 1000-word vocabulary that we have created.
7. After that, I defined a function to get POS tags from the given sent_id and token word, as well as created a POS mapping so that each POS tag is assigned a unique no. which will be used in the feature vector. I have here considered all the POS tags present in the train set.
8. Similar to the POS tags, I have defined a function to get the dependency relations of a word and the dependency mapping considering all the dependent relations present in the train set.
9. I have created 1 separate function to create a binary vector from given indices corresponding to each feature. This function will return a binary vector of size (2|V|+3|p| +4|R|) by setting 1 to the indices obtained from each given feature.

10. The feature vector is created as per the feature given in the problem statement. I have assumed corresponding indices to be none if any feature is not found for a particular configuration and transition.
    Assumption: One assumption that I have used is considering a dummy word **"root"** at the start of a sentence. This is used so as to capture the "root" dependency relationship given in the training data for some tokens. This "root" word will never be used for calculating any features as it is a dummy variable and not actually a part of the sentence. I have added it for my convenience to capture that "root" word relationship.
11. So, every time we have to do something related to the top of the stack, then I have added a check condition that the length of the Stack should be greater than 1 so that there is some other word in the stack apart from "root" and it will be used for feature calculation. Also, every time I check whether the stack and buffer are empty or not before calculating a feature so as to avoid any errors corresponding to an empty stack or buffer.
12. For the features like x.yDEP, I have first calculated all the relevant sets of arcs, and then among those arcs take that word which is relevant as per the features. I have also added comments at the time of creating a feature vector in the feature vector function. Once, I obtained all the indices corresponding to each feature, I created a binary vector from it and injected this binary vector into a feature vector of size 4*(2|V|+3|p| +4|R|) at the appropriate position based on the transition type and maintain the same for all the time so there is uniformity
13. I have also convert the transition type to a numbering so it will be easier to access them. This mapping is as follows:
    LEFR_ARC - 0
    RIGHT_ARC - 1
    SHIFT - 2
    REDUCE - 3
14. I have separately defined functions to perform the execution of each transition. This will take stack, buffer, and arc_set in input and depending on the transition perform certain operations and return the updated stack, buffer, and arc_set. This function only does direction execution, not check the condition. So, we will use this function when we are sure that we have to perform certain transitions.
15. Next, I define the Oracle function which defines a certain set of rules that are given in the problem statement which are as per the eager parsing algorithm, and based on these rules, I have defined which transition to perform. The corresponding function is called and in return, I get an updated stack, Buffer, and Arc set. The oracle is required to determine the next transition based on the heuristics.
16. The training function is defined to train a parser, where we define a certain no. of epochs to iterate on train data. The standard algorithm of Eager arc parsing is used where for a particular training example, we are starting with Buffer

containing all the words, Arc set is empty, Stack storing the dummy word “root” and then stopping when reaching the transition at which Buffer is empty.

17. Now, the model is trained, I have used epochs as 2 as training the model is taking much time for me and so I have to restrict myself to less No. of epochs. The weights of the trained model are stored in the form of a numpy array so that they can be reloaded easily and can be used at the time of testing.
18. At the time of testing, as we don't have access to gold-standard dependency relationships, I have created a majority classifier for features like x.yDEP as told in the problem statement and found which is the most frequent dependency tag for each POS tag pair.
19. The feature vector for the test is almost the same as to train one, with some minor modifications for certain features. Similarly, I have created a new function for performing transition based on certain new heuristics rules given in the problem statement for the test data.
20. Using the weights obtained at the time of train, I have run the depenedency parsing for test and obatiend a UAS score of **27.64 %.**

```
Average UAS Score over Test Data: 0.27649188475321057
```

21. A low Unlabeled Attachment Score (UAS) in eager arc parsing can result from various factors, including limited model complexity, insufficient or poor-quality training data, inadequate feature representation, and error propagation due to the sequential nature of decisions in parsing. Additionally, the inherent limitations of the eager arc parsing algorithm, especially in handling complex syntactic constructions, the need for optimal hyperparameter tuning, challenges in generalizing to unseen data, and dealing with linguistic diversity, can further contribute to reduced performance. To improve UAS, strategies might involve enhancing the training dataset, refining feature engineering, utilizing more sophisticated models, optimizing hyperparameters, and adopting techniques to reduce error propagation and accommodate linguistic complexities.

Difficulties I have faced:

1. The task is quite big and there are many places where I have to take care of may edge conditions.
2. Code of the assignment is quite big.
3. Also, the features that are defined are quite complex and it took me time to understand those features.
4. I am not able to run the model for large no. of epochs as it is taking more than 2 hours for a single epoch. So, weights are not train that much good causing poor result on test data.