PRATICAL ASSIGNMENT 2: DEPENDENCY PARSER

Master in Artificial Intelligence 2024-25

Authors: Adriano Miranda Seoane Sara Micol Ferraina **Date:** December 12, 2024

Introduction:

This project aims to implement a Transition-based dependency parser using Machine Learning.
In this methodology , the parsing process is developed as a sequence of transitions (e.g., SHIFT, LEFT-ARC, RIGHT-ARC) that build a dependency tree incrementally.
The parser predicts the most likely transition at each step based on the current state.

Implementation note:

This section highlights key implementation details for the dependency parsing system developed in this project. This project has been developed using Visual Studio Code and it is composed by **5 main files:**
1) algorithm.py: This code implements an Arc-Eager dependency parsing system with three main classes: **Transition:** Represents parsing transitions (SHIFT, REDUCE, LEFT-ARC, RIGHT-ARC) and stores action and dependency type.
Sample: Represents a training sample. Contains parsing state and corresponding transition. And it contains the function **state_to_feats** that extracts features from parsing states.
ArcEager: Implements the Arc-Eager parsing algorithm.
Namely, it creates initial parsing state,it generates gold standard transition sequence and at the end it applies the transitions to the state generating the dependency arcs.

2)data_utils.py: It contains some function for read the .conllu files and for the creation of the input data for the model:
read_file() : Reads a CoNLL-U formatted file (a standard format for dependency parsing) and returns a list of dependency trees.
create_inputs() : Processes a list of samples to extract features. It extracts: words, POS (Part of Speech) tags, transitions, dependencies. In particular, it calls extract_dependency() to get dependency labels
extract_dependency() : Parses transition strings to extract dependency labels.
create_dataset_testing() : Uses similar feature extraction as **create_inputs()** but it extracts words and POS tags but for each single sample.
update_trees(): To update the trees after the predictions
3)model_utils.py: It contains some function that are then used in model.py:

map_transitions_to_int():converts transition strings to integer representations.
is_transition_valid() : Checks if a given transition is valid for the current parser state ,using ArcEager parser methods.
select_valid_transition(): Check if the current prediction of the model is valid.
4)model.py: This file contains the neural network model and the necessary functions for training, testing, and evaluation. In particular,all the functions are implemented in the **class ParserMLP** and they are the following:

__init__()):Inizialization: The constructor sets up key hyperparameters like embedding dimensions, hidden layer size, and model parameters then calls build_model to construct the neural network architecture.

build_model():Model Building: The function that builds the Network, that is characterized as following:

Embedding Layer: Maps input tokens (sequence of word IDs) to dense word embeddings.

Flatten Layer: Converts embeddings to a single vector for dense layer input.

Hidden Layers: Two fully connected layers with ReLU activation and L2 regularization.

Dropout Layer: Prevents overfitting.

Output Layers: Two separate softmax output layers, one for transition actions and another for dependency labels.

preprocess_samples(): Data Preprocessing

This function processes the raw training and validation data:extracts features, transitions, and dependencies from input samples.Uses TensorFlow's TextVectorization layer for vectorizing text features.Maps transitions and dependencies to categorical IDs and creates one-hot encodings for model training.

train()(Training): Trains the model using processed features and labels and it validates it using validation data.

evaluate()(Evaluation):Performs the evaluation on the dev set.

run()Inference: Performs inference for syntactic parsing. It simulates parsing for input sentences using the ArcEager parser. It iteratively predicts transitions and dependencies until parsing completes and at the end it ensures valid transitions are selected using helper functions like select_valid_transition.

5)Main.pynb: This is a Python notebook containing the main code of the project. The code follows the following key steps:

1) Retrieve trees from the .conllu file and remove the non-projective trees
2) Build the network, train, and evaluate it. (The data preparation is inside of the train function).
3) Make predictions.
4) Use the predicted arcs in the finals state to obtain updated test_trees.
5)Convert the trees to .conllu.
6)Post-process the obtained .conllu file
7)Evaluate with the .conllu evaluation script.

How to run the code:

To run the code is sufficient to download all the files in the folder P2, open it in Visual Studio and run **Main.ipynb**.

However, in the Horizontal Prediction section, since it takes a long time to do the predictions for all the test_trees we advice to set: n = 50 ,sentences= test_trees[:n], but that is already implemented in the provided code.

Hyperparameter Tuning and Comparison :

Model with different hyperparameters:

Model	Word emb. dim.	Hidden dim.	Batch Size	Epochs
1	100	64	64	10
2	32	64	64	10
3	32	128	128	10
4	64	64	64	10
5	128	64	32	10

Model performance:

Model	Dependency Accuracy	Dependency loss	Transition Accuracy	Transition Loss
1	0.8430	0.6005	0.8446	0.5564
2	0.8335	0.6392	0.8247	0.6076
3	0.8278	0.6252	0.8278	0.6252
4	0.8346	0.6700	0.8259	0.6064
5	0.8284	0.6717	0.8198	0.6209

Discussion and results analysis:

This section provides an analysis of the results obtained from evaluating different model configurations with varying hyperparameters. The key metrics considered are dependency accuracy, dependency loss, transition accuracy, and transition loss, which reflect the model's performance in dependency parsing and transition prediction tasks.

Best Dependency Accuracy: Model 1 achieved the highest dependency accuracy of 0.8430, with the lowest dependency loss (0.6005). This suggests that a larger word embedding dimension (100) and a moderate hidden dimension (64) effectively captured the linguistic features required for dependency parsing.

Best Transition Accuracy: Model 1 also achieved the highest transition accuracy (0.8446) with the lowest transition loss (0.5564), making it the best-performing configuration overall for transition prediction. **Impact of Hyperparameters:**

Word Embedding Dimension: Larger embedding dimensions (e.g., 100 in Model 1) provided richer word representations and improved accuracy. However, excessively large embeddings (e.g., 128 in Model 5) may overfit the data, as seen in higher loss values.

Hidden Dimension: Increasing the hidden dimension (e.g., 128 in Model 3) slightly degraded performance compared to smaller hidden dimensions (e.g., 64 in Model 1). This suggests that moderate complexity is sufficient for the task.

Batch Size: Models with smaller batch sizes (e.g., 32 in Model 5) underperformed compared to those with batch sizes of 64 or 128, highlighting the benefit of processing more samples per update.

The results demonstrate the importance of balancing model complexity and data representation to achieve optimal performance in dependency parsing and transition prediction tasks.