Practical Assignment 2 - Dependency Parsing

User Manual

Marcelo Ferrer

Aymen Merchaoui

To start using the application, the files of the assignment should be copied to a google drive folder. Once they are in the drive, the file with the extension .ipynb should be open with Google Colab.

As seen in the figure1 ,the colab application is divided in 5 sections: Imports and Declarations, Data processing and Oracle Simulation, Model Training and Oracle Test, Results and Predictions and finally Conclusions.

It's important to emphasize that all cells should be run in a sequential mode (especially inside each section). Constants have been defined to run only part of the program in case of being necessary.

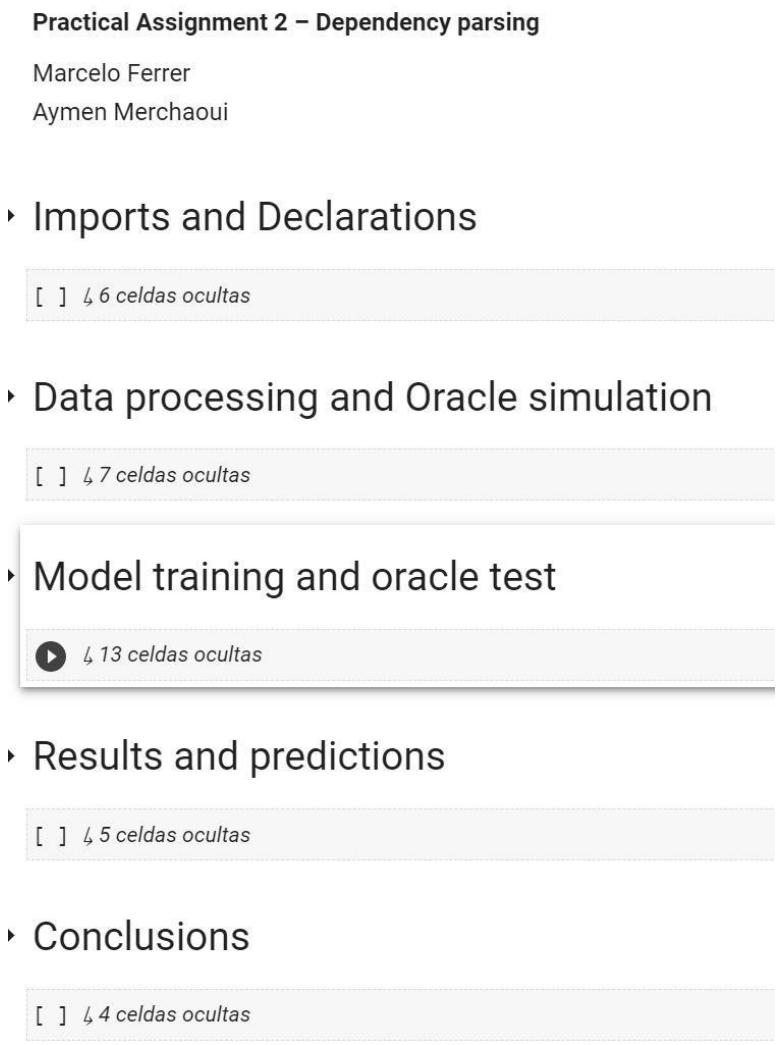


Figure 1-Sections

When we expand Imports and Declaration, we can see in Figure 2 that the first cell is for installing conllu, a python library that parses and allows the serialization of Universal Dependencies files. The execution of this cell could take a few minutes.

```
[ ]  # Installs conllu, a python library to parse Universal Dependencies https://pypi.org/project/conllu/
     !pip install conllu

     Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
     Collecting conllu
       Downloading conllu-4.5.2-py2.py3-none-any.whl (16 kB)
     Installing collected packages: conllu
     Successfully installed conllu-4.5.2
```

Figure 2-Install CONLLU

Passing to the second cell, we will find various constants to define depending on the work we want to realize. Here is an explanation of each configuration:

For our problem we have to set the maximum length for the words inside the stack and the buffer, so we assign that number to MAX_LEN. (Figure3)

As in the previous assignment we should set the maximum length of the word level and the maximum length of the characters level for training the upos prediction model, they are represented respectively by MAX_LEN_PRE and CHAR_LEN_PRE.

```
[ ]  # List of constants

     ######### MODEL #########
     # Lenght of each component of the sentences
     MAX_LEN=7
     # Maximun length of the sequences
     MAX_LEN_PRE = 128
     # Maximun length of the char sequences
     CHAR_LEN_PRE = 18
```

Figure 3-Model Constants

For training our model we could change the number of epochs through EPOCHS, we can also define the optimizer that we want to use with OPTIMIZER and the the size of batching in BATCH_SIZE. (Figure4)

```
######### TRAIN #########
# Amount of epochs to train
EPOCHS = 20
# Optimizer to use in the compilation of the model
OPTIMIZER = "Adam" #RMSProp, Nadam
# Size of the batch to use in the training
BATCH_SIZE = 64
```

Figure 4-Train Constants

With setting SHOW_SAMPLES to True we will have a sentence example for each of the preprocessing steps and with SHOW_NON_PROJECTIVE we will have a look of the non projective sentences that we removed.(Figure5)

```
######### DEBUG #########
# Print samples of the data and model
SHOW_SAMPLES = False
# Print non projective sentences
SHOW_NON_PROJECTIVE = False
```

Figure 5-Debug Constants

We can retrain the models from the start or we can use the previous saved models that we trained by setting TRAIN_MODELS to True or False. With ORACLE_TEST we can define if we want to generate the oracle result file again to test against the gold file. (Figure6)

```
########## EXECUTION ##########
# Train the models again
TRAIN_MODELS = True
# Execute oracle test
ORACLE_TEST = True
```

Figure 6-Execution Constants

Here we have all the constants to specify the folders path. The LOCAL_DRIVE_PATH is used to define the path of the drive content folder of the user where all the files should be located. Then we will pass the dataset files for the training, validation and testing. Next we should specify where our new generated datasets will be saved. Finally we have the path of the models already trained and ready to use.

```
######### PATHS #########
# Google drive folder path where the files are located
LOCAL_DRIVE_PATH = '/content/drive/MyDrive/P2/'
# Urls of the dataset
URL_TRAIN = LOCAL_DRIVE_PATH + 'Dataset/en_partut-ud-train.conllu'
URL_TEST = LOCAL_DRIVE_PATH + 'Dataset/en_partut-ud-test.conllu'
URL_VAL = LOCAL_DRIVE_PATH + 'Dataset/en_partut-ud-dev.conllu'
# Urls of the new generated datasets
URL_TEST_CLEAN = LOCAL_DRIVE_PATH + 'Dataset/en_partut-ud-test_clean.conllu'
URL_TEST_BASIC = LOCAL_DRIVE_PATH + 'Dataset/en_partut-ud-basic.conllu'
URL_TEST_ADVANCED = LOCAL_DRIVE_PATH + 'Dataset/en_partut-ud-advanced.conllu'
URL_TEST_ADVANCED_P1 = LOCAL_DRIVE_PATH + 'Dataset/en_partut-ud-advanced-p1.conllu'
# Model locations
URL_BASIC_MODEL = LOCAL_DRIVE_PATH + 'Models/basic'
URL_ADVANCED_MODEL = LOCAL_DRIVE_PATH + 'Models/upos'
```

Figure 7-Path constants

After we finish with the constant configuration, the third cell mounts the Google drive. (Figure8)

```
[3]  # Mount google drive to use the .py classes
     from google.colab import drive
     drive.mount('/content/drive')

     Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

Figure 8-Google mount

When the execution of this cell is done, a popup asking about credentials to access may appear.(Figure9)

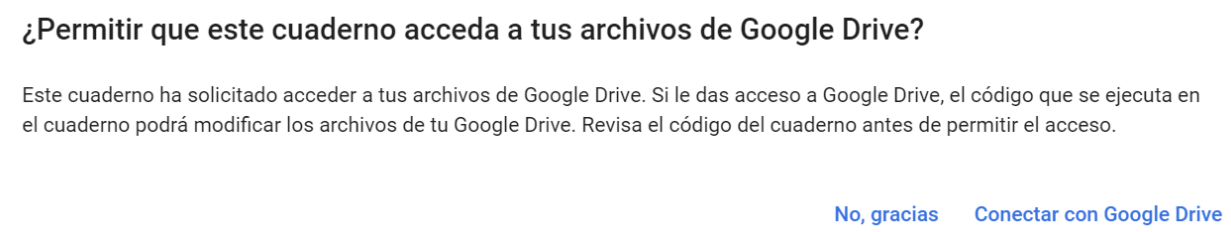


Figure 9-Google mount popup

The execution of this cell will grant access to the folder where the rest of the files are located which was defined in previous steps. (Figure10)

```
[4]  # Import the path in the drive where the py files are alocated
     import sys

     sys.path.append(LOCAL_DRIVE_PATH)
```

Figure 10-Import sys

Once the drive is mounted and the local path loaded, we can access the rest of the files from the menu at the left, pressing the files icon. A files tree will be deployed where in addition to the ipynb file, other files with the extension .py could be seen. (Figure11)

Figure 11-Menu Drive Files

In case that the code in these files wants to be visualized or even changed, a simple double click on the file will open it in a new window on the right showing the selected file in Figure 12.

DataHandler.py

```
# Imports all the necesary spacenames
from keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from pathlib import Path
import tensorflow as tf
import conllu
import numpy as np
from Oracle_Simulator_Handler import My_Oracle_Simulator_Handler
from Oracle_Test_Handler import My_Oracle_Test_Handler
from Oracle_Predict_Handler import My_Oracle_Predict_Handler

# Class in charge of manipulating the data
class MyDataHandler:
  def __init__(self, max_len):
    # Prepare dictionaries for columns FORM, UPOS and DEPREL
    # To get ready to convert from text/label to a numeric value and viceversa
    # Use NA as the oov token, clear filters (because we want puntuaction) and lower the wo
    self.form_tokenizer = Tokenizer(oov_token='<NA>', filters='', lower=True)
    self.upos_tokenizer = Tokenizer()
    self.deprel_tokenizer = Tokenizer()
    self.action_tokenizer = Tokenizer()

    self.max_len = max_len

  # Reads the file in universal dependecies format and returns a list with inputs
  def get_UD_Values(self, path, rewrite):
```

Figure 12-Python File

In case a change has to be made to the python classes, it would be necessary to restart the environment so the main program can reload these differences. For that, in the top menu the option Execution Environment has a restart option.(Figure13)

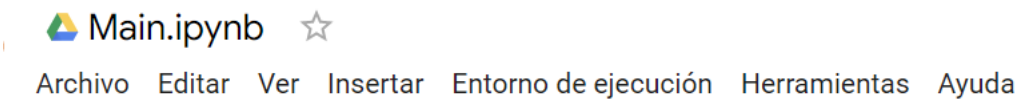


Figure 13-Menu Drive

Another thing to take in account is that the training of the models can be a little slow, so having the GPU acceleration option of the environment activated could be a good idea.

For this, in the same bar of the Execution Environment there is a menu for changing the type of the execution environment. When selecting this option, a pop up with a combo box will be displayed, inside that, the GPU option is available.

Note that this option is not limitless, so in case that more GPU power is needed, an option is to access with a premium account or opening the project with a different google user. (Figure14)

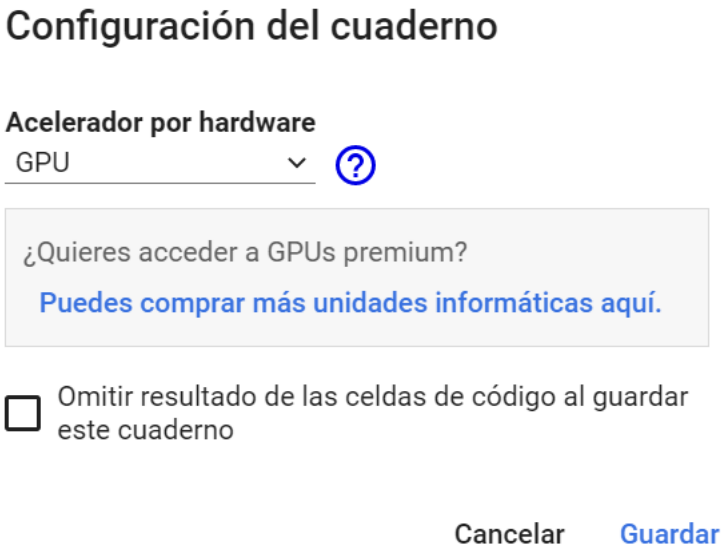


Figure 14-GPU Configuration

Regarding the code of the program and its execution, to start using the functions of the DataHandler class we create an instance while passing the maximum length already assigned.(Figure15)

Data processing and Oracle simulation

```
[6]  # Create an instance of the data handler
     dh = MyDataHandler(MAX_LEN)
```

Figure 15-Data Handler Instance Creation

Next we can start passing our training,validation and testing file for pre-processing in order to get the data that we need and receive it ready to implement. (Figure 16)

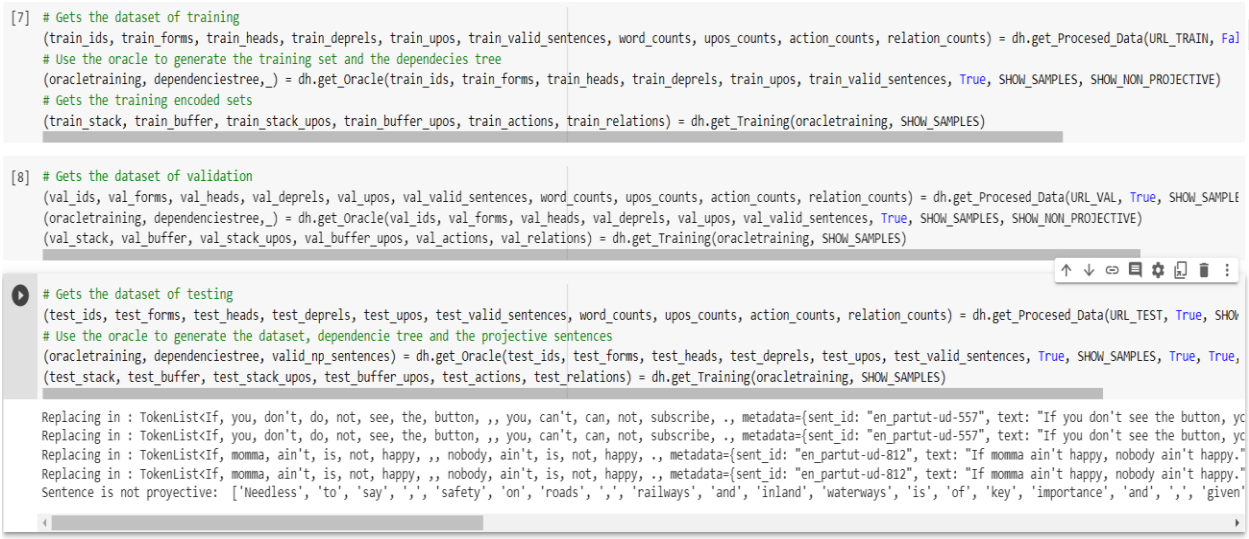


Figure 16-Data Preprocessing

Here we pass our data for pre-processing to get the training,validation and testing data for the upos prediction model. (Figure17)

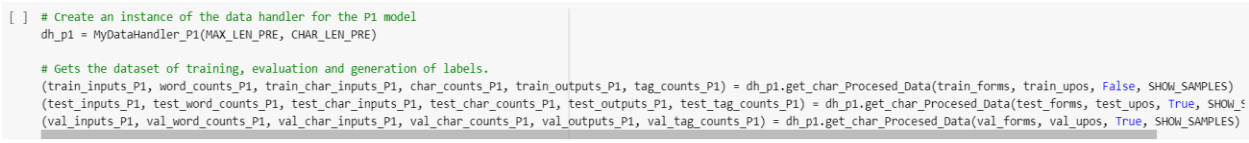


Figure 17-Data preprocessing for UPOS prediciton

Other libraries such as Keras, Numpy, Matplotlib or Path, used in this assignment, should already be installed in the environment. We define all the importations for our further use. (Figure18)

```
# Imports all the necesary spacenames
import tensorflow as tf
import keras
import matplotlib.pyplot as plt
import numpy as np
from keras import layers
from keras.models import Sequential
from tensorflow.keras.layers import Dense, InputLayer, Embedding
from keras.callbacks import EarlyStopping
import matplotlib.pyplot as plt
```

Figure 18-Libraries

One last step before starting to train the models is converting all the data that the model could take to tensors for been use by the keras models. (Figure19)

```
[ ]  #convert to tensors so we could pass to the model
     train_actionst=tf.convert_to_tensor(train_actions)
     train_relationst=tf.convert_to_tensor(train_relations)
     train_stackt=tf.convert_to_tensor(train_stack)
     train_buffert=tf.convert_to_tensor(train_buffer)
     train_stack_upost=tf.convert_to_tensor(train_stack_upos)
     train_buffer_upost=tf.convert_to_tensor(train_buffer_upos)

     val_actionst=tf.convert_to_tensor(val_actions)
```

Figure 19-Conversion to Tensors

Once we have our data converted, its ready to be use. So, we first create an instance of the class ModelHandler while passing the constants used to build the models.(Figure 20)

Model training and oracle test

```
[ ]  # Create an instance of the data handler
     mh = MyModelHandler(MAX_LEN, MAX_LEN_PRE, CHAR_LEN_PRE, word_counts, upos_counts, action_counts, relation_counts, tag_counts_P1)
```

Figure 20-ModelHandler instance creation

In our problem we adopted different architectures while constructing our models and layers, so each approach has a cell (Figure21) for training its model and showing the metrics results after each epoch.

Basic model (use only words to predict)

```
[ ]  # Get the basic model (only words)
     if TRAIN_MODELS:
       model_basic = mh.get_model("Basic")
       es = EarlyStopping(monitor='val_loss', mode='min', verbose=1 , patience=3)
       model_basic.compile(loss="categorical_crossentropy", optimizer='adam', metrics=['accuracy'])
       history_basic = model_basic.fit(x=[train_stackt,train_buffert], y=[train_actionst, train_relationst], validation_data=([val_stackt, val_buffert]
       #Evaluate accuracy against the test data
       model_basic.evaluate([test_stackt,test_buffert], [test_actionst, test_relationst])
       model_basic.save(URL_BASIC_MODEL)
     else:
       model_basic = keras.models.load_model(URL_BASIC_MODEL)
```

```
Epoch 1/10
1269/1269 [==============================] - 17s 6ms/step - loss: 2.6615 - dense_loss: 0.8950 - dense_1_loss: 1.7665 - dense_accuracy: 0.6115 - der
Epoch 2/10
1269/1269 [==============================] - 7s 6ms/step - loss: 1.8396 - dense_loss: 0.5798 - dense_1_loss: 1.2597 - dense_accuracy: 0.7674 - dens
```

Figure 21-Model training

For better understanding of the model architecture, layers, inputs and outputs you can run a cell to plot the model graph and see how the model was constructed: (Figure22)

```
tf.keras.utils.plot_model(model_basic, to_file='model_basic.jpg', show_shapes=True)
```

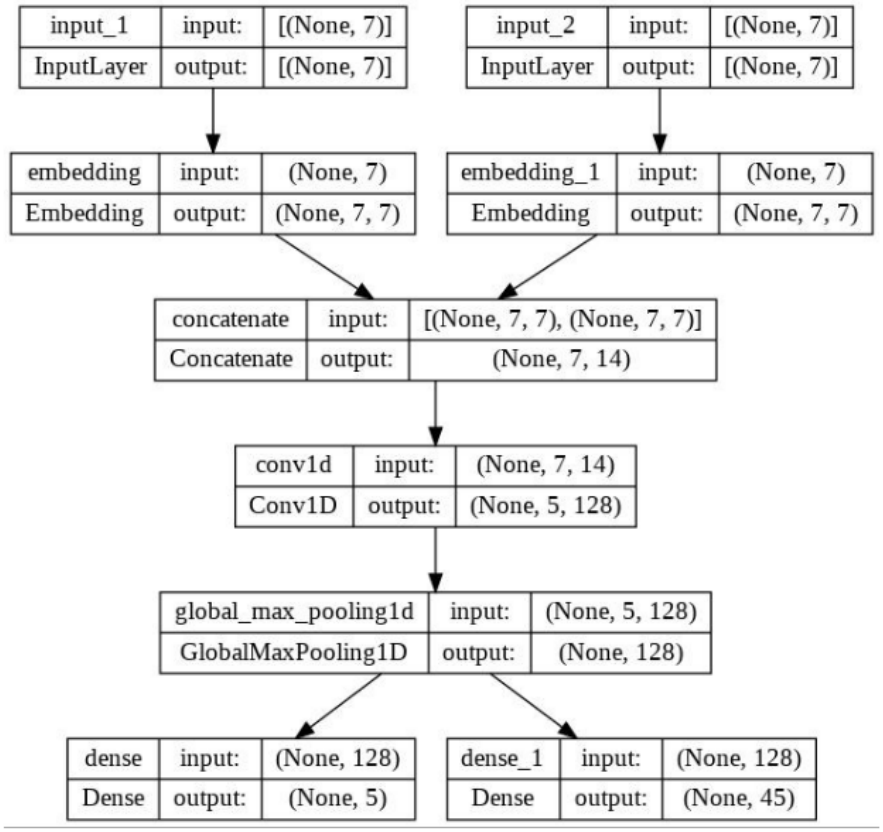


Figure 22-Model Graph

As we said in the report the real evaluation of the model will be by comparing the conllu testing data file with his conllu prediction data file.

The next step is the test inputs prediction considering the pre-conditions of the Oracle that should be satisfied. We call the function get_Real_Oracle as shown in Figure 23 to process the test data and create the conllu file containing the dependency relations predictions and updating the head column.

```
[ ]  if ORACLE_TEST:
       new_dependencies_basic = dh.get_Real_Oracle(valid_np_sentences, SHOW_SAMPLES, model_basic,URL_TEST_BASIC)
```

```
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 21ms/step
```

Figure 23-Real Oracle prediciton

We now pass to the results and predictions section. The metrics that we should use to evaluate a dependency parser model are UAS and LAS. In order to get those metrics we pass the gold test file (the clean file that doesn't contain multiwords and non-projective sentences) for comparison with the predicted test file. We have been provided a file for this process, so as output we have a table that shows how well different metrics are matching. We are interested in the values corresponding to UAS and LAS. So we read the metrics corresponding as shown in Figure 24, an example of the metrics result of the first basic model

▾ Results and predictions

```
[ ]  print("Results for the basic word model:")
     !python3 /content/drive/MyDrive/P2/conll18_ud_eval.py /content/drive/MyDrive/P2/Dataset/en_partut-ud-test_clean.conllu /content/drive/MyDrive/P2/Dataset/en_partut-ud-basic.conllu
```

```
Results for the basic word model:
Metric     | Precision |    Recall |  F1 Score | AligndAcc
-----------+-----------+-----------+-----------+-----------
Tokens     |    100.00 |    100.00 |    100.00 |
Sentences  |    100.00 |    100.00 |    100.00 |
Words      |    100.00 |    100.00 |    100.00 |
UPOS       |    100.00 |    100.00 |    100.00 |    100.00
XPOS       |    100.00 |    100.00 |    100.00 |    100.00
UFeats     |    100.00 |    100.00 |    100.00 |    100.00
AllTags    |    100.00 |    100.00 |    100.00 |    100.00
Lemmas     |    100.00 |    100.00 |    100.00 |    100.00
UAS        |     54.46 |     54.46 |     54.46 |     54.46
LAS        |     32.20 |     32.20 |     32.20 |     32.20
CLAS       |     26.91 |     19.51 |     22.62 |     19.51
MLAS       |     23.13 |     16.77 |     19.44 |     16.77
BLEX       |     26.91 |     19.51 |     22.62 |     19.51
```

Figure 24-Real evaluation metrics

Last part is dedicated to the dependency trees of the sentences that the user could use. For this, we provide a function that receive a list of string sentences, make the corresponding oracle prediction and get the dependency tree(Figure 25). Next we show those results.

```
[ ]  sentences=["Reach your target customer.","Google is a nice search engine.", "Mary has a cat", "Spiderman exists", "I am real"]

     new_dependencies = dh.get_Predictions(sentences, SHOW_SAMPLES, model_advanced, model_advanced_pre, dh_p1)

     for x in new_dependencies:
       print(x)

     # Results can be check against an online dependency parser
     # like https://corenlp.run/
```

Figure 25-Predict dependecy parsing

We pass the sentences in the list and we run the cell. The model would always be the advanced one with the P1 ups prediction, because we don't have the upos of the sentences, so we have to look for them before making the dependency relations prediction.

As outputs we will have the dependency trees predicted for each sentence. Figure 26 is an exemple of the predicition results of some sentences passed

```
[('Root', 'root', 'Reach'), ('customer', 'nmod', 'target'), ('customer', 'nmod', 'your'), ('Reach', 'obl', 'customer'), ('Reach', 'punct', '.')]
[('search', 'nmod', 'nice'), ('engine', 'nmod', 'search'), ('engine', 'det', 'a'), ('engine', 'cop', 'is'), ('Google', 'conj', 'engine'), ('Googl
[('Root', 'root', 'Mary'), ('Mary', 'nsubj', 'has'), ('cat', 'det', 'a'), ('has', 'obj', 'cat')]
[('Root', 'none', 'Spiderman'), ('Spiderman', 'conj', 'exists')]
[('real', 'cop', 'am'), ('real', 'expl', 'I'), ('Root', 'root', 'real')]
```

Figure 26-Dependency parsing results

The last section is the conclusion:

Conclusions

In this part we discussed the metrics results that we get from our models and their predictions. So to a better understanding we have some plots showing the comparison between the models results. Those plots are explained in the final part of the colab file under **Final Conclusions**

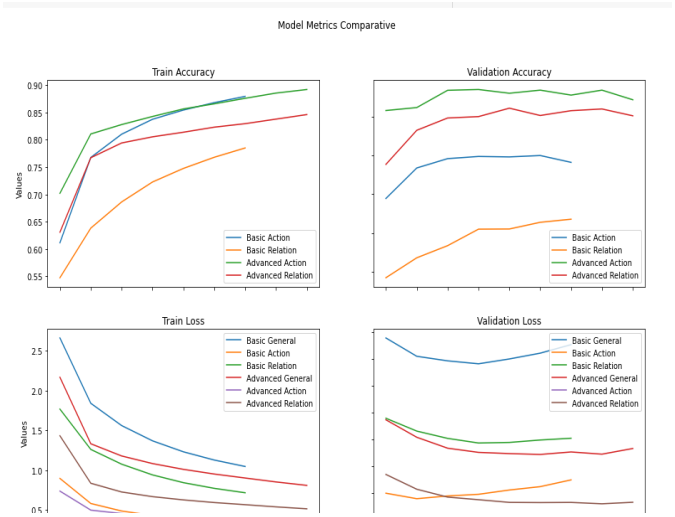
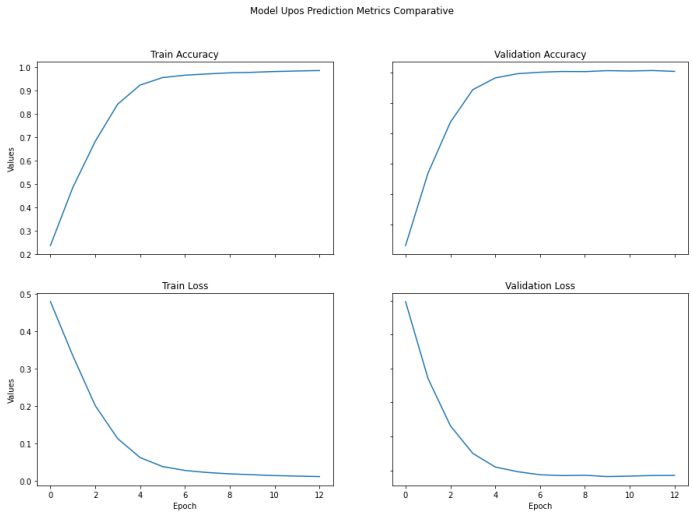


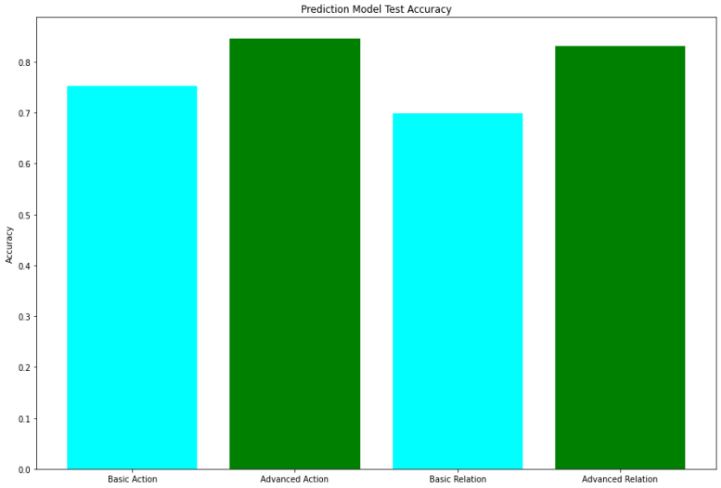
Figure 27-Model metrics comparative

The figure 27 shows for example how the accuracy and loss evolved after each epoch for the models



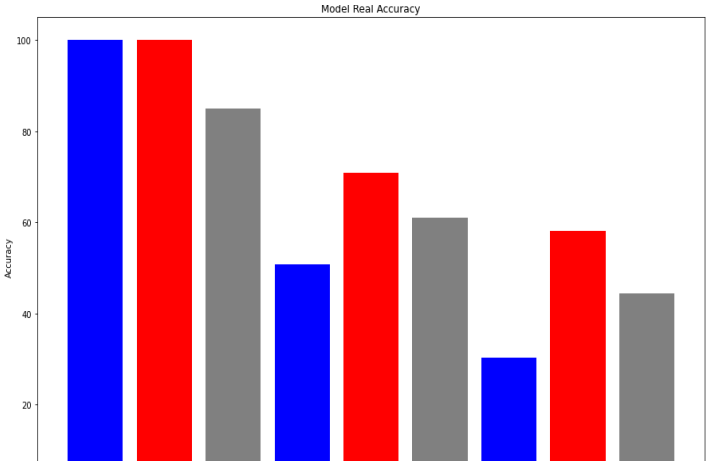
The figure 28 gives us information about the train and validation accuracy of the upos predicition model

Figure 28-Models upos prediction metrics comparative



The figure 29 shows the comparison of model accuracies for each of the output: Actions and Relations

Figure 29-Prediciton model test accuracy



The figure 30 shows the comparison of the UAS, LAS and the UPOS results of the three approaches adopted.

Figure 30-Real Oracle accuracies