UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

Practical Assignment 1 - Part-of-speech (PoS) Tagging

Final Report

Marcelo Ferrer

Aymen Merchaoui

Program structure: the program has been divided into three files. The python class DataHandler, that is in charge of the processing of the datasets, the python class ModelHandler, that is in charge of the model creation, training and execution, and finally, the Colab notebook Main, who has the constant definitions, classes instantiation, the three languages main processing and final conclusions.

Data processing: for the procurement of the dataset from the UD repository, we decided to use conllu, a python library that parses the text and with a loop we distributed it to our wanted format, removing all the "_" words in the process. A previous version reading, looping and parsing manually all the UD information in code was used, but conllu has a better resolution time.

For handling the words, characters and upos information, we create separated tokenizers that allow us the creation of the dictionary, sequences and one hot encoding in an easy way. The words tokenizer also has the oov argument active (if this symbols weren't taken into account, the validation accuracy drops), filters (with empty value, to override the default function that remove punctuation) and lower (true in this case as the default value, because testing with false demonstrate that it not improves the accuracy).

For the character processing, we create a char-level tokenizer who treats each character as a token. To do that we use the parameter "char_level=True" and pass the parameter "filters='!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n'" in order to also consider the special characters.

Regarding the word oov values, we decided to assign them to the upos "other", since it gives the best accuracy and the prediction of the upos of oov values still predicts correctly and not the "other" value. We tested leaving the upos as it was, but accuracy was lower. Another way to do this could have been just remove the oov and its upos from the dataset, but that would alter the meaning of the phrases. As for the padding, the suggested pad_sequences was used, with a default value of zero and padding / truncating of the size constant.

The class DataHandler has been created to group all the necessary functions for the data processing. An instance of this class is created to process each language model. This class receives the sequence size as a parameter and the max word size as an optional.

The train, test and development dataset are used in the "traditional way" (the first to train the model, development to validate the training and test to evaluate the results). Other implementations like unifying the dataset and taking random values or using cross validation were taken in account and discarded.

Basic Architecture:

The first architecture consists in implementing a basic model that gets as inputs the word level embeddings. Then it will be passed to the word level LSTM layer, which should pass his outputs to a Dense layer that will finally output the label for each token by applying the softmax as an activation function.

Different models types and layers were tested to determine which one gives a better accuracy. As can be seen in the class ModelHandler, the three main basic models were a sequential (to start testing), a functional (to start using the functional api) and a functional bi (to use a bidirectional layer). Regarding the layers, an optional extra dense layer has been added which gives a ≈1% extra accuracy to the English model. In the case of the Spanish and German model, this extra layer actually makes the model over fit quickly so it was removed.

Advanced Architecture:

The second architecture adopted in our problem is the high level architecture with two inputs, adding the character level embeddings as second input with the word level embedding, then for each word we have the word-level representation concatenated to the representation of each characters.

To feed the char-level inputs to the character-level LSTM we tokenize the characters and pad them so that each word will have the same number of characters.

We pass the char-level embeddings to the LSTM to get the vectors of each word and then we concatenate with the word-level embeddings. We proceed here like in the first architecture but with an LSTM fed with that concatenation followed by a TimeDistributed Dense layer to output finally the labels of each token.

Final layers and parameters:

Input layer: the input layer for the words has defined just one parameter , the shape, where the sentence length is used (128). The char input, has also an extra dimension for the maximum number of characters for each word (18).

Text Vectorization layer: this type of layer was considerate in the initialization of the development, but a Tokenizer was used instead because this one was easier to manipulate.

Embedding layer: this layer has as parameters the input dimension (size of the corresponding dictionary), output dimension (128 or 18, size of the vector of each represented word or character) and mask zero (True for skipping the zeros added to the padding).

Dense layer: for the optional dense layer, 128 neurons are being used (other values were tested with similar results). For the Time Distributed dense layer, the number of neurons is the same as the number of possible outcomes, as is, 18 possible UPOS and the activation is softmax (for multiple classifications).

LSTM layer: as recommended, the hyper parameter return_sequences=True for the LSTM word instance. For the characters, this option is deactivated. For the number of units, 128 and 18 were configured (other values were tested with similar results).

Bidirectional layer: this layer wraps the LSTM layer with no additional parameters.

Time distributed layer: this layer wraps the dense layer defined before to apply the word sequences.

The class ModelHandler has been created to group all the necessary functions for the creation, training, predictions and evaluations of the model. This class receives the model type and the hyper parameters necessary to instance all the layers.

Different other model structures were tacked in consideration, like extra dense layers or a double bidirectional, but the accuracy did not improve. A RNN also was tested instead of the LSTM, with similar results. Older models and not used options have not been removed from the class to retest in case of being necessary and to demonstrate the different applications that have been taken in account in the decision of the model structure.

Compile and Training: the train dataset has been divided in batches of 64 to accelerate the time of training. For the optimizer four of them give good values (Adam, Rmsprop, Adamax, Nadam), all from the same family. Finally it was used Adam for the assignment which shows the less fluctuation values.

For the loss, categorical crossentropy with binary values and sparse categorical crossentropy with numeric values had been considered, both with similar accuracy values, choosing categorical crossentropy (generally speaking, the only difference between the both is the type of output).

The number of epochs is defined as 10, but we can see that after some epochs the accuracy is decreasing and the loss value is increasing, so our model could be overfitted. To prevent this we adopt the technique of early stopping (ending the training after the epoch when our model starts to overfit) using callbacks with keras in the fit() function, via the “callbacks” argument.

The EarlyStopping receives a parameter to monitor the val_loss, making the model stop training when we get a val_loss higher than the minimum val_loss achieved. We added patience=3 so we can wait 3 more epochs to see if the val_loss could get lower than the minimum one.

As for the metrics, just the accuracy has been solicited (others like True negatives and positives or the weighted mean are available but will not add extra value to the analysis).

Performance across the datasets: the performance of the final model across the 3 languages datasets is similar (between 89 and 93%), being Spanish the best result by a ≈2%.

A manual accuracy function has been implemented (evaluate_predict) because in early stages of the development, the accuracy of the model was high, but the actual predictions weren’t correct. This function compares upo expected against upo predicted one by one.

In the conclusions sections, both accuracy and manual accuracy statistics are displayed to see in a graphic way how the behavior of the models is. The final performance of the models is this one:

Basic Model	English	Spanish	German
Training	97.96%	98.76%	98.66%
Validation	90.55%	93.03%	89.84%
Testing	90.58%	93.38%	89.48%

Advanced Model	English	Spanish	German
Training	98.32%	98.30%	98.33%
Validation	92.11%	94.13%	91.33%
Testing	92.59%	94.25%	90.66%