In-game Toxicity Detection with Slot Tagging

Ananda Aziz Hardjanto

[1] University of Sydney, NSW 2006, Australia
ahar3481@sydney.edu.au

1 Data preprocessing

The provided data for this slot filling framework, is extracted from the CONDA dataset. There are three files given, train, validation and test data files, which are subsets from the CONDA dataset. The data is in from of lists of sentences, with many punctuations and other grammatical characteristics. For each word in the sentence, there is a token provided to label the word, basing on the word type to see what slot token it belongs to. There are 7 tokens, T (toxicity), C (Character), D (Dota-specific), S (Game Slang), P (Pronoun), O (Other) and a special token 'SEPA' that refers to utterance in the game chat. To make the slot filling model work, some data preprocessing is needed to help feed the data to the model. Three data preprocessing techniques was used: lower-case, tokenization and word-tag index. Firstly, the dataset is loaded to the Jupyter notebook. The data is loaded using the panda package to read the csv file that it is on. They are then separated to train sentence, train labels, test sentence, validation sentence and validation labels.

1.1 Lower-Case

The first pre-processing method done, it to make every letter in the sentence to lower case. The purpose of this preprocessing is to create consistency in the dataset. In a in-game chat environment, players usually would chat without any regards to the proper uses of capital letters. Players even in some cases would mix capital letters in-between words, as these chats are typed in the heat of the moment, with slang words being most of what is used. By making sure every letter in every word is lower, in the following preprocessing of indexing each word token, it will not create separate indexes for words that differ for only its capital letters. This gives words with the same index more samples to train from, which may increase its training accuracy. When letters have different capital letters, they will have different indexes even when they have the same meaning. To lower the letters in every word, the python function *.lower* is used while iterating in all the train, test and validation sentence.

1.2 Tokenization

Following the lowering of every letter, the data is then tokenized. Tokenization is used on all the dataset that was loaded. To tokenize the data, the package from *nltk.tokenize* is used. The package *RegexpTokenizer* is used to tokenize the letters, while also removing any unwanted punctuation. The sentence data is tokenized by word, while the labels are tokenized on each label. The purpose of this is so that it is easier to index the data, and feed to the model. Since the purpose of this experiment is for slot tagging on the word level, by tokenizing it word by word instead of by sentence, it makes it easier for the model to train based on the word's index and find its label, instead of referencing it to the entire sentence. By doing this, it should increase the accuracy of the training by being able to predict each word's labels more clearly.

1.3 Word and Tag Index

Lastly, the tokenized words and labels/tags are indexed. Indexing is done by iterating through each token, and giving each word an index. When the same word is iterated through, no new index will be produced. The indexing function is also defined prior to this to simplify the indexing process. Indexing is a necessary process in machine learning as when the data is feed through the model, it was take numbers of the index as its inputs to learn the pattern.

2 Input Embedding

2.1 Aspect 1: PoS Tagging

The first aspect used is the Part of Speech tagging. This syntactic textual feature embedding. The purpose of this Part of Speech tagging here, is to further do a text classification of each word. Since the words are already tokenized, the Part of Speech tagging here is only interested on a word-to-word basis, without having any reference to the sentence. In the Part of Speech tagging, each word will be given their grammatical characteristics, whether a word is noun, plural, and many more [4].

In this report, the package *RegexpTagger* from nltk is used on a predefined pattern. The set of patterns here, are simple patters with 8 different patterns that would define the grammatical characteristic of the tokenized words. The PoS Tagging is done based on the rule-based tagging, in which the rule is defined by the pattern above. The function will then iterate through the train, validation and test tokenized sentence to pair each word with their grammatical characteristic. The approach to the Part of Speech tagging here is by using the Morphological criteria by seeing the form of the word, and its affixes.

The purpose of this PoS tagging is to add more information on each token of the word embedding. By using the PoS tagging, it gives each word their belonged grammatical parts of speech. The generated grammatical parts of speech paired with each word

will then be used as an extra feature for the Semantic word embedding which will be discussed further bellow.

2.2 Aspect 2: Word2Vec with Dataset

In the Semantic Textual Feature embedding, the genism model Word2Vec is used to create the embedding feature. In this section, the CBOW is used for the prediction-based word representation. In CBOW, it predicts the center word from a bag of context word. By running the data sentences through the Word2Vec, it creates an embedding matrix of a one-hot vector that can be used as a vector representation of each word in the model [5].

One of the embedding input methods that is tested on the baseline model, is by using Word2Vec that is trained on the dataset itself. The model is trained on the train, validation and testing tokenized words. The function Word2Vec is used that is imported from Genism to simplify the Word2Vec model.

The reason on training the Word2Vec on the dataset, instead of using a pretrained word embedding is due to the nature of the words that is in the dataset. The data consists of unique words that are informal and mostly slang words, which would not be present in the usual corpus in which most Word2Vec models are trained on. Having so, using a pretrained Word2Vec such as Glove-25-Twitter, would not be a great representation of the dataset, as it does not share any similar domain. By using the dataset itself to train the embedding, it helps create a more accurate embedding matrix in which would be used in training the model.

For the first embedding input is to use the embedding matrix that is generated from the word2vec as the weights for the embedding in the model. This embedding matrix weight is then inserted into the model in the *self.word_embeds.weight.data.copy_* section of the model. By not specifying whether *freeze* is true or not in the parameters, the default is set to True, in which the weights will not be learnable upon training.

2.2.1 Concatenation of PoS Tagging with Word2Vec with Dataset Embedding

The second embedding input method is a concatenation of the Word2Vec Semantic embedding with the Part of Speech tag syntactic embedding. In this embedding, the Word2Vec is trained by taking the list of words that already have their Part of Speech tag generated from section 2.1. The Word2Vec model when trained using the dataset from the sentence has a limitation of not being able to analyze any morphological similarity. It will represent every word as the independent vector. However, by using the result from the PoS tagged words, it will help the Word2Vec to train on words which has its grammatical part of speech for each word. This will act as an extra feature for the Word2Vec to train on.

2.3 Aspect 3: Word2Vec with Dota Heroes

In this domain feature word embedding, Word2Vec is also used to create the embedding matrix for the model. However, instead of the Word2Vec being trained on the dataset, it is trained using a corpus of the names of Dota Heroes. The corpus is downloaded from an API of open dota [2]. The corpus has the details on every heroes that is in the game, with their names being one of the columns.

The API of the open Dota is loaded into the Jupyter notebook using pandas, as the file is in the form of .json, and loaded into a data frame. From here, the column '*localized_name'* is extracted into a list. The reason of taking only this column is that the localized name is what usually players would use when referring to a hero in chat. Most of the time, when playing an online video game, people would chat and refer to one another by the name of the heroes that they are playing, instead of referring to the gamer tag of a player. This is due to simplicity to both side and easier to understand.

2.3.1 Concatenation of Word2Vec with Dota Heroes and Dataset

The third word input embedding is a concatenated Word2Vec with both the dataset and the Dota Heroes data. By using both the data, it will put an emphasis on any Dota heroes names that are mentioned in the chat. This will change the weights of the embedding matrix that is pass through to the model. By doing this, the model should be able to detect more effectively on any tokens that is mentioning the Dota heroes, and label is as C (character).

3 Slot Filling/Tagging model

3.1 Baseline Model (Bi-LSTM + CRF)

For this report, the baseline model that is used to train is a bidirectional Long Short-Term Memory (Bi-LSTM) with the addition of Conditional Random Field (CRF). Firstly, Bi-LSTM is an important base of the neural network as the LSTM nature allows the neural network to reduce the problem of vanishing gradient upon training due to learning on large data. This would help on getting the model to train on the features and getting the predicted labels. In addition, the Bi-LSTM works similar to the standard LSTM model, however it works twice by feeding the algorithm the features forward once, from the beginning of the data towards the end. With a second time running back from the end to the beginning of the data. By using the Bi-LSTM, it can take advantage of the features from the forward state and the backward state.

The addition of the CRF model, it would help to increase the tagging accuracy **[3]**. This is done through tag sequencing, using the tag neighbors the get the pattern of the tags. By combining the sequence of the tags, with respect the tag itself of the current word being feed onto the model, it is able to perform a higher accuracy on tagging.

On this baseline model, the model is essentially a CRF model, in which the features are feed into it through Bi-LSTM **[1]**.

The model is design by firstly creating a class of the Bi-LSTM + CRF. In the model, in the *init* function of the class, the value of the Bi-LSTM is then defined, with the embedding_dim and hidden_dim being defined. In here, the input embedding and its weights will be used as well. As it is a bidirectional model, upon applying values to the lstm model, the bidirectional is set to *True.*

Following this, a few more functions inside the class is defined with *forward_alg, get_lstm_features, score_sentence, Viterbi_decode, neg_log_likelihood and forward.* Most of the function defined above will be used for the CRF part of the model. In the forward algorithm function, it initialize the CRF model in which it takes the features given. The get_lstm_features extract the features of the data that is send into the model, both forward and backward. The score sentence function would use the best score to each label and its transition score, to get the best label for the input. The Viterbi decode then would then take into the transition and current input to get the score of the current tag by using the current input and also the next and previous one. The neg_log_likelihood is the loss function that is used upon training the model. With the last function, forward is the forward function of the Bi-LSTM model.

4 Evaluation

4.1 Evaluation setup

To setup the evaluation of this experiment, every model here is run in a single jupyter notebook. The first section of the notebook would be the data loading and data pre-processing as this will be kept constant throughout all the models to create consistency on the inputs given to the model to train and for the embedding as well. Following this section, there is the embedding section in which it is divided to mainly three sections, aspect 1, 2 and 3 as per described in the assignment documentation. However, inside aspect 2, there is two code chunks in which one will run Word2Vec with only the dataset, and the other would train Word2Vec with the PoS tagging and the dataset. On the last aspect, number 3, it will run Word2Vec with the dota heroes embedding and the dataset. As for the embedding dimension for every embedding, it will be kept constant in which it is the vector size of the input towards the model, which is 100. The last section of the notebook would be the models itself, in which every code chunk is marked with a markdown, to help identify which model it is, and which training is being done.

As for the learning rate, epoch, weight decay, hidden dimension and optimizer, it is all kept constant. This is to ensure that any changes in the accuracy of each model is based on the change of what it is being observed, which is changes in the model or embedding technique, and not due to the hyperparameters being adjusted. The learning rate is kept constant at 0.001. This value was chosen based on trials. When the learning rate was set at 0.01, the model tends to overfit, in which it was the training

was doing too well on the data. When the value was changed to 0.001, the baseline model was able to learn a bit better without overfitting. The weight decay is at 1e-4, and the optimizer used is Stochastic Gradient Descent. The hidden dimension is also kept constant at the value of 100, with epoch at 2.

4.2 Evaluation result

4.2.1 Performance Comparison

4.2.2 Ablation Study – Different Input Embedding

Combinations	F1 Mean Score
Word2Vec Dataset	**0.8123**
Word2Vec + PoS Tag	**0.8105**
Word2Vec Dataset + Dota Heroes	**0.7066**

Table 1: Embedding Input

As it can be seen from table 1, the three input embedding combination results in different F1 mean score. The embedding input that performs the best is the Word2Vec that uses the dataset only with a F1 mean score of 0.8123. The Word2Vec concatenated with PoS tag being a close second, with a F1 mean score of 0.8105. As for the Word2Vec that is trained on the dataset and the Dota Herores API, it performed poorly as it only got a F1 Mean score of 0.706 which is significantly lower than the other two comparison.

A possible reason for this trend is due to the population of the features in the embedding input. As the embedding features is increase, as such in the Word2Vec with PoS tagging, it gives the model more vector of features. However, PoS tagging is helpful when the model is trying to tag and label word tokens with respect to the sentence sequencing. In this case, the tagging is on a word to word base, without any consideration of the sequence. Therefore, the PoS tag might have added more features to the embedding population that are not helpful for the model to learn.

The same case can be said for the Word2Vec with an additional domain of Dota hero names. In this case, as mentioned previously, a lot of the in-game chat would use slang and shorten words, including the names of the heroes. In which would create different embeddings for different spellings. In addition, this embedding would put an emphasis on the tag of the heroes only, with no regards to the other tags. Hence, it may have increase the performance of the tags on detecting tokens of the hero names, however, it reduce the weight of the embedding for other tokens, reducing the

accuracy of tagging other ones. Since the character token is not a significant amount of the support in the data, this may have reduced the overall accuracy.

References

1. Pytorch.org. 2022. *Advanced: Making Dynamic Decisions and the Bi-LSTM CRF — PyTorch Tutorials 1.11.0+cu102 documentation*. [online] Available at: <https://pytorch.org/tutorials/beginner/nlp/advanced_tutorial.html> [Accessed 3 June 2022].
2. Api.opendota.com. 2022. [online] Available at: <https://api.opendota.com/api/heroes> [Accessed 3 June 2022].
3. Huang, Z., Xu, W. and Yu, K., 2015. *Bidirectional LSTM-CRF Models for Sequence Tagging*. [online] Arxiv.org. Available at: <https://arxiv.org/pdf/1508.01991.pdf> [Accessed 3 June 2022].
4. Pykes, K., 2020. *Part Of Speech Tagging for Beginners*. [online] Medium. Available at: <https://towardsdatascience.com/part-of-speech-tagging-for-beginners-3a0754b2ebba#:~:text=Part%2Dof%2Dspeech%20(POS,the%20word%20and%20its%20context.> [Accessed 3 June 2022].
5. Vaidya, K., 2020. *Sentiment Analysis using LSTM and GloVe Embeddings*. [online] Medium. Available at: <https://towardsdatascience.com/sentiment-analysis-using-lstm-and-glove-embeddings-99223a87fe8e> [Accessed 3 June 2022].