Report - CS272 HW4 Generating Function Descriptions

Junze Liu (junzel1)

March 2020

1 Implementation of the Informed Model

As the naive model is originally implemented, I directly got the BLEU score by running the given code. For the informed model, I adjusted the vector dimensions of embedders and encoders. The dimension of the embedded vector should be the concatenation of naive encoder and extra encoder. Then, I got the following results on Java source code as the baseline of the naive and informed model after 50 training epochs.

h-param	naive	informed
valid	0.0335	0.0760
test	0.0311	0.0742

Table 1: Different configurations for Java

Similarly, I got the baseline results on Python source code.

h-param	naive	informed
valid	0.0137	0.1050
test	0.0106	0.0915

Table 2: Different configurations for Python

2 Customizing all the Configurations

2.1 Naive Model

First, I explored more sophisticate embedders for the naive model. Adopting experiences from HW3, I tried pre-trained Glove embedder and Elmo. Since available pre-trained Glove embedders online are only compatible with 50-D embedding space. So, the token embedding dimension is raised to 50 from original 30, while the character token embedder remains the same. Because character token embedder is considered to process information of simpler structure. Furthermore, to study whether 50 is the optimal embedding dimension, a series of experiments with different embedding dimension settings are shown in the table below.

Lang	Emb-D	30	50	100	100*	1000
Java	valid	0.0335	0.0349	0.0331	0.0331	0.0349
	test	0.0311	0.0334	0.0308	0.0308	0.0343
Python	valid	0.0137	0.0064	0.0063	0.0064	0.0137
	test	0.0106	2.73e-6	2.73e-6	2.77e-6	0.0106

Table 3: Different embedding dimensions

It can be seen from the table above that 50-D token embedding provides a relatively good performance both in training and test process. Also, the dimension of pre-trained Elmo is 1024, which potentially affect the embedder's performance. So, I also tried using 1000-D embedding as a comparison. Though character

token embedder is considered dealing with information of simpler structure, I also increased the embedding dimension of character token embedder and decreased the embedding dimension of token embedder, which differentiate the embedding dimension distribution between tokens and character tokens. From the table above, different embedding dimension distribution between tokens and character tokens does not affect the performance.

As a conlustion, embedding dimensions of 30 and 1000 provide the best test performance on Python data, and embedding dimensions of 50 and 1000 provide the best test performance on Java data. Considering the efficiency and compatibility, I will choose 50 as the dimension of the tokens embedding.

h-param	baseline	Glove	Elmo
valid	0.0349	0.0331	0.1468
test	0.0334	0.0308	0.1387

Table 4: Different configurations for Java

From the results shown in the tables above, based on Java data, naive model using pre-trained Elmo embedder gives the best result. The BLEU score is increase by 3 times, but each training requires several hours on a PC using Intel-9700K (8-core, 8 threads).

h-param	baseline	Glove	Elmo
valid	0.0064	0.0435	0.0660
test	2.73e-6	0.0438	0.0469

Table 5: Different configurations for Python

Similarly, I got the experiment results on Python source code. The Elmo embedder again provides the highest BLEU score, while Glove embedder also gives good performance. Extremely low test BLEU score of baseline model here can be ignored, as Glove and Elmo embedder both providese BLEU score much higher than the original model's (0.0106). Different from results on Python, the improvement is fairly limited. (The test BLEU score is increased by 7%.) Considering the the computation power consumption and time efficiency, Glove is the best option for this task.

2.2 Informed Model

Following the experiment setup I used for naive models, the same embedder/encoder settings are applied to informed models. Different from naive models, informed models have two main embedders (instead of one), source embedder and extra embedder. In this case, I applied the different choice of embedders to both source embedder and extra embedder. The experiment results are shown in the following table.

h-param	baseline	Glove	Elmo
valid	0.0664	0.2045	0.2474
test	0.0689	0.1998	0.2438

Table 6: Different configurations for Java

From the results, we can see that using more complicated embedders significantly improved the BLEU score of informed models on Java data. The Elmo embedder provides the best BELU score but just slightly overpasses the pre-trained Glove.

Similarly, I got the results on Python source code.

h-param	baseline	Glove	Elmo
valid	0.0440	0.0881	0.0616
test	0.0358	0.0830	0.0462

Table 7: Different configurations for Python

Different from the results I observed on Java data, though more complicated embedders provided better BLEU score, results from Elmo embedder are actually worse than pre-trained Glove. The reason could be complicate Elmo embedder is not trained on Python source code, which produces some bias when facing Python data. Thus, less complicate Glove embedder is the best choice for Python data. However, my strange observation is that using 50-D embeddings, the results of informed model on Python are worse than the original model using 30-D embeddings. But being limited to the available pre-trained Glove embedders, I may not be able to explore Glove or Elmo on 30-D embeddings.

3 Analysis of Predictions

Based on the scale of improvement observed in the previous result tables, I choose Java as the language to do the prediction analysis. Followings are 10 randomly picked test samples. According to the instruction, I will explain several interesting ones out of these 10.

index	source	rate	sentence
1	*source code*		*util timezone object clone*
	gold description		Creates a copy of this TimeZone
	naive prediction	1	returns a string representation of this map entry
	informed prediction	2	creates and executes a scheduledfuture that becomes enabled after the given delay

Table 8: Prediction Analysis for Java

In this case, the naive model and informed both failed making the correct prediction. But the informed model is slightly better because it captured some information out of the 'hint', "this TimeZone", and predicted the "creates ... scheduledfuture ...".

index	source	rate	sentence
2	*source code*		*text decimalformat roundingmode get rounding mode*
	gold description		Gets the link java
	naive prediction	1	return the next pseudorandom number
	informed prediction	3	gets the datetime formatter with the given formatting styles for the given locale

Table 9: Prediction Analysis for Java

In this example, naive model completely failed. It is probably because two rare phrases appear in the source code. But given the 'hint', the informed model would know it is relative to the 'formatting' or 'formatter'.

index	source	rate	sentence
3	*source code*		*util concurrent synchronousqueue boolean remove all collection c*
	gold description		Always returns false
	naive prediction	1	removes from this set all of its elements that are contained in the specified collection
	informed prediction	1	always returns true if class descriptor supports a of a method of that it can be read without blocking

Table 10: Prediction Analysis for Java

Naive model and informed model both failed in this case.

index	source	rate	sentence
4	*source code*		*util vector int hash code*
	gold description		Returns the hash code value for this Vector
	naive prediction	4	returns the hash code value for this map
	informed prediction	4	returns a hash code for this code long

Table 11: Prediction Analysis for Java

Both models perform pretty good in this example, except missing a key information of 'Vector'.

index	source	rate	sentence
5	*source code*		*net inetsocketaddress boolean is unresolved*
	gold description		Checks whether the address has been resolved or not
	naive prediction	1	returns true if this set contains no elements
	informed prediction	1	returns true if this class is serializable but not externalizable and defines a conformant readobjectnodata method

Table 12: Prediction Analysis for Java

In this example, both models failed. But it is clear that the 'hint', "or not", gave a strong suggestion to both models.

index	source	rate	sentence
6	*source code*		*io printstream void print boolean b*
	gold description		Prints a boolean value
	naive prediction	3	prints a character
	informed prediction	4	writes a bit int

Table 13: Prediction Analysis for Java

It can be seen that informed model performed better. It is because the 'hint', "boolean value", helped predict "bit".

index	source	rate	sentence
7	*source code*		*io filterwriter void write string str int off int len*
	gold description		Prints a boolean value
	naive prediction	3	writes a bit float
	informed prediction	4	writes a bit int

Table 14: Prediction Analysis for Java

index	source	rate	sentence
8	*source code*		*io objectoutput void flush*
	gold description		Flushes the stream
	naive prediction	5	flushes the stream
	informed prediction	5	flushes the stream

Table 15: Prediction Analysis for Java

Both models performed perfectly here. The potential explanation may be this line of code is common in the Java dataset.

index	source	rate	sentence
9	*source code*		*util arrays void swap short x int a int b*
	gold description		Swaps xa with xb
	naive prediction	5	swaps xa with xb
	informed prediction	5	swaps xa with xb

Table 16: Prediction Analysis for Java

index	source	rate	sentence
10	*source code*		*util concurrent linkedblockingqueue boolean offer object e long timeout timeunit unit*
	gold description		Inserts the specified element at the tail of this queue waiting if necessary up to the specified wait time for space to become available
	naive prediction	1	nullpointerexception inheritdoc
	informed prediction	4	inserts the specified element into this delay queue

Table 17: Prediction Analysis for Java

In our last example, informed model performed nearly perfect but naive model completely failed. Considering the source code is difficult to describe in text, the 'hint' could gave the right information for the model.

To dig deeper in how the 'hint' effects the model's performance, I tried giving the true 'hint', related 'hint' and unrelated 'hint'. I took the example of *"io filterwriter void write string str int off int len"* (example # 7). The results of informed model with different 'hint's are shown in the following table.

hint	rate	prediction
"boolean value" (real)	4	writes a bit int
"float value"	2	writes a portion of an array of characters
"a a"	1	writes block data header

Table 18: Prediction Analysis for Java

According to the results, real 'hint' provides the closest prediction. When using "float value" as 'hint', the informed model is misled. When using "a a", which is a completely unrelated 'hint', the informed model cannot capture the key information at all.

In conclusion, informed models are generally better than naive models. They may be both good at shorter source code. But, when facing longer source code or rare phrases, informed model does fetch information from the 'hint' and makes better prediction. In addition, the prediction of informed models does not always end with the 'hint'.

4 Statement of Collaboration

I completed this homework by myself. No collaboration to be reported.