C2_W3_lecture_nb_03_oov

August 2, 2022

# Out of vocabulary words (OOV) ### Vocabulary In the video about the out of vocabulary words, you saw that the first step in dealing with the unknown words is to decide which words belong to the vocabulary.

In the code assignment, you will try the method based on minimum frequency - all words appearing in the training set with frequency >= minimum frequency are added to the vocabulary.

Here is a code for the other method, where the target size of the vocabulary is known in advance and the vocabulary is filled with words based on their frequency in the training set.

[1]:
```python
# build the vocabulary from M most frequent words
# use Counter object from the collections library to find M most common words
from collections import Counter

# the target size of the vocabulary
M = 3

# pre-calculated word counts
# Counter could be used to build this dictionary from the source corpus
word_counts = {'happy': 5, 'because': 3, 'i': 2, 'am': 2, 'learning': 3, '.': 1}

vocabulary = Counter(word_counts).most_common(M)

# remove the frequencies and leave just the words
vocabulary = [w[0] for w in vocabulary]

print(f"the new vocabulary containing {M} most frequent words: {vocabulary}\n")
```

```
the new vocabulary containing 3 most frequent words: ['happy', 'because',
'learning']
```

Now that the vocabulary is ready, you can use it to replace the OOV words with $< UNK >$ as you saw in the lecture.

[2]:
```python
# test if words in the input sentences are in the vocabulary, if OOV, print␣
↪<UNK>
sentence = ['am', 'i', 'learning']
output_sentence = []
```

```
print(f"input sentence: {sentence}")

for w in sentence:
    # test if word w is in vocabulary
    if w in vocabulary:
        output_sentence.append(w)
    else:
        output_sentence.append('<UNK>')

print(f"output sentence: {output_sentence}")
```

```
input sentence: ['am', 'i', 'learning']
output sentence: ['<UNK>', '<UNK>', 'learning']
```

When building the vocabulary in the code assignment, you will need to know how to iterate through the word counts dictionary.

Here is an example of a similar task showing how to go through all the word counts and print out only the words with the frequency equal to `f`.

[3]:
```
# iterate through all word counts and print words with given frequency f
f = 3

word_counts = {'happy': 5, 'because': 3, 'i': 2, 'am': 2, 'learning':3, '.': 1}

for word, freq in word_counts.items():
    if freq == f:
        print(word)
```

```
because
learning
```

As mentioned in the videos, if there are many $< UNK >$ replacements in your train and test set, you may get a very low perplexity even though the model itself wouldn't be very helpful.

Here is a sample code showing this unwanted effect.

[4]:
```
# many <unk> low perplexity
training_set = ['i', 'am', 'happy', 'because','i', 'am', 'learning', '.']
training_set_unk = ['i', 'am', '<UNK>', '<UNK>','i', 'am', '<UNK>', '<UNK>']

test_set = ['i', 'am', 'learning']
test_set_unk = ['i', 'am', '<UNK>']

M = len(test_set)
probability = 1
probability_unk = 1

# pre-calculated probabilities
```

```
bigram_probabilities = {('i', 'am'): 1.0, ('am', 'happy'): 0.5, ('happy',␣
↪'because'): 1.0, ('because', 'i'): 1.0, ('am', 'learning'): 0.5,␣
↪('learning', '.'): 1.0}
bigram_probabilities_unk = {('i', 'am'): 1.0, ('am', '<UNK>'): 1.0, ('<UNK>',␣
↪'<UNK>'): 0.5, ('<UNK>', 'i'): 0.25}

# got through the test set and calculate its bigram probability
for i in range(len(test_set) - 2 + 1):
    bigram = tuple(test_set[i: i + 2])
    probability = probability * bigram_probabilities[bigram]

    bigram_unk = tuple(test_set_unk[i: i + 2])
    probability_unk = probability_unk * bigram_probabilities_unk[bigram_unk]

# calculate perplexity for both original test set and test set with <UNK>
perplexity = probability ** (-1 / M)
perplexity_unk = probability_unk ** (-1 / M)

print(f"perplexity for the training set: {perplexity}")
print(f"perplexity for the training set with <UNK>: {perplexity_unk}")
```

```
perplexity for the training set: 1.2599210498948732
perplexity for the training set with <UNK>: 1.0
```

### Smoothing

Add-k smoothing was described as a method for smoothing of the probabilities for previously unseen n-grams.

Here is an example code that shows how to implement add-k smoothing but also highlights a disadvantage of this method. The downside is that n-grams not previously seen in the training dataset get too high probability.

In the code output bellow you'll see that a phrase that is in the training set gets the same probability as an unknown phrase.

[5]:
```
def add_k_smooting_probability(k, vocabulary_size, n_gram_count,␣
↪n_gram_prefix_count):
    numerator = n_gram_count + k
    denominator = n_gram_prefix_count + k * vocabulary_size
    return numerator / denominator

trigram_probabilities = {('i', 'am', 'happy') : 2}
bigram_probabilities = {( 'am', 'happy') : 10}
vocabulary_size = 5
k = 1

probability_known_trigram = add_k_smooting_probability(k, vocabulary_size,␣
↪trigram_probabilities[('i', 'am', 'happy')],
                           bigram_probabilities[( 'am', 'happy')])

probability_unknown_trigram = add_k_smooting_probability(k, vocabulary_size, 0,␣
 ↪0)

print(f"probability_known_trigram: {probability_known_trigram}")
print(f"probability_unknown_trigram: {probability_unknown_trigram}")
```

```
probability_known_trigram: 0.2
probability_unknown_trigram: 0.2
```

### Back-off Back-off is a model generalization method that leverages information from lower order n-grams in case information about the high order n-grams is missing. For example, if the probability of an trigram is missing, use bigram information and so on.

Here you can see an example of a simple back-off technique.

[6]:
```
# pre-calculated probabilities of all types of n-grams
trigram_probabilities = {('i', 'am', 'happy'): 0}
bigram_probabilities = {( 'am', 'happy'): 0.3}
unigram_probabilities = {'happy': 0.4}

# this is the input trigram we need to estimate
trigram = ('are', 'you', 'happy')

# find the last bigram and unigram of the input
bigram = trigram[1: 3]
unigram = trigram[2]
print(f"besides the trigram {trigram} we also use bigram {bigram} and unigram␣
 ↪({unigram})\n")

# 0.4 is used as an example, experimentally found for web-scale corpuses when␣
 ↪using the "stupid" back-off
lambda_factor = 0.4
probability_hat_trigram = 0

# search for first non-zero probability starting with trigram
# to generalize this for any order of n-gram hierarchy,
# you could loop through the probability dictionaries instead of if/else cascade
if trigram not in trigram_probabilities or trigram_probabilities[trigram] == 0:
    print(f"probability for trigram {trigram} not found")

    if bigram not in bigram_probabilities or bigram_probabilities[bigram] == 0:
        print(f"probability for bigram {bigram} not found")

        if unigram in unigram_probabilities:
            print(f"probability for unigram {unigram} found\n")
            probability_hat_trigram = lambda_factor * lambda_factor *␣
↪unigram_probabilities[unigram]
        else:
            probability_hat_trigram = 0
    else:
        probability_hat_trigram = lambda_factor * bigram_probabilities[bigram]
else:
    probability_hat_trigram = trigram_probabilities[trigram]

print(f"probability for trigram {trigram} estimated as␣
↪{probability_hat_trigram}")
```

```
besides the trigram ('are', 'you', 'happy') we also use bigram ('you', 'happy')
and unigram (happy)

probability for trigram ('are', 'you', 'happy') not found
probability for bigram ('you', 'happy') not found
probability for unigram happy found

probability for trigram ('are', 'you', 'happy') estimated as 0.06400000000000002
```

### Interpolation The other method for using probabilities of lower order n-grams is the interpolation. In this case, you use weighted probabilities of n-grams of all orders every time, not just when high order information is missing.

For example, you always combine trigram, bigram and unigram probability. You can see how this in the following code snippet.

[7]:
```
# pre-calculated probabilities of all types of n-grams
trigram_probabilities = {('i', 'am', 'happy'): 0.15}
bigram_probabilities = {( 'am', 'happy'): 0.3}
unigram_probabilities = {'happy': 0.4}

# the weights come from optimization on a validation set
lambda_1 = 0.8
lambda_2 = 0.15
lambda_3 = 0.05

# this is the input trigram we need to estimate
trigram = ('i', 'am', 'happy')

# find the last bigram and unigram of the input
bigram = trigram[1: 3]
unigram = trigram[2]
print(f"besides the trigram {trigram} we also use bigram {bigram} and unigram␣
↪({unigram})\n")
```

```
# in the production code, you would need to check if the probability n-gram␣
↪dictionary contains the n-gram
probability_hat_trigram = lambda_1 * trigram_probabilities[trigram]
+ lambda_2 * bigram_probabilities[bigram]
+ lambda_3 * unigram_probabilities[unigram]

print(f"estimated probability of the input trigram {trigram} is␣
↪{probability_hat_trigram}")
```

```
besides the trigram ('i', 'am', 'happy') we also use bigram ('am', 'happy') and
unigram (happy)

estimated probability of the input trigram ('i', 'am', 'happy') is 0.12
```

That’s it for week 3, you should be ready now for the code assignment.