ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

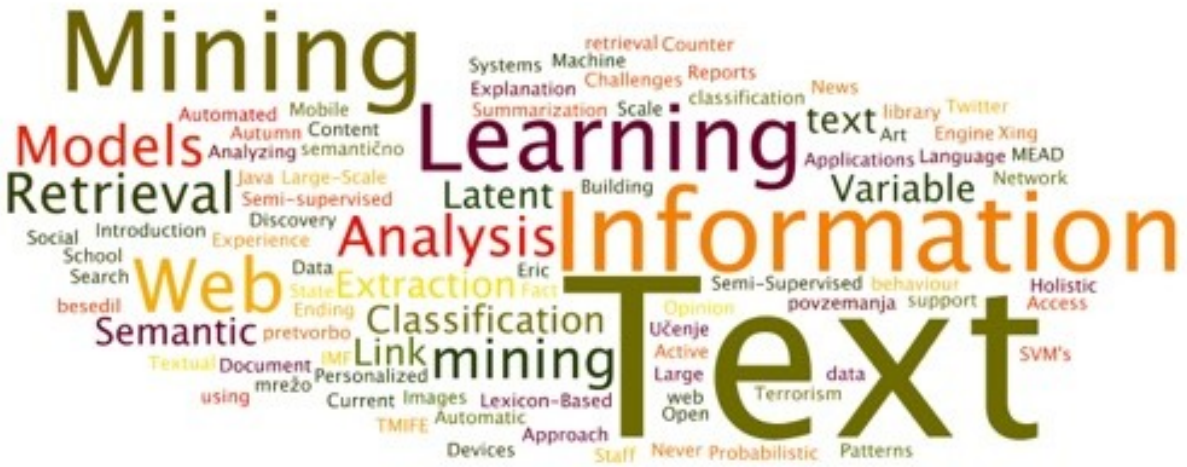
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Text Analytics
Exercise 1 Report

Papastamos Konstantinos

Kanellis Georgios

Tsolas Leonidas



Exercise

The exercise given is described below:

1. Implement (in any programming language) a bigram and a trigram language model for word sequences (e.g., sentences), using Laplace smoothing (slide 8) or optionally (much better in practice) Kneser-Ney smoothing (slide 38). Train your models on a training subset of a corpus (e.g., from the English part of Europarl). Include in the vocabulary only words that occur, e.g., at least 10 times in the training subset; use the same vocabulary in the bigram and trigram models. Replace all out-of-vocabular (OOV) words (in the training, development, test subsets) by a special token *UNK*. Assume that each sentence starts with the pseudo-token *start* (or the pseudo-tokens *start1*, *start2* for the trigram model) and ends with *end*.

2. Check the log-probabilities that your trained models return when given (correct) sentences from the test subset vs. (incorrect) sentences of the same length (in words) consisting of randomly selected vocabulary words.

3. Estimate the language cross-entropy and perplexity of your models on the test subset of the corpus, treating the entire test subset as a single sequence, with *start* (or *start1*, *start2*) at the beginning of each sentence, and *end* at the end of each sentence. Do not include probabilities of the form P(*start*|...) (or P(*start1*|...) or P(*start2*|...)) in the computation of perplexity, but include probabilities of the form P(*end*|...).

4. Optionally combine your two models using linear interpolation (slide 10) and check if the combined model performs better.

For our implementation we used python and the nltk package. Below all steps taken will be described in detail along with the necessary result screenshots.

The full code used for the exercise along with the output text files can be found **Here**.

1. Data Loading and Preprocessing

First the source release file of the europarl corpus, as found here, was downloaded. Only the english folder ("en") was kept. After the files were ready, the script iterated over the files, loaded the raw text and did the following preprocessing tasks for each individual one:

1. Cleaned the file text using the Clean_Text() function. The function removes all line breaks, removes unnecessary text such as the html-style tags present in the corpus, reduces any remaining whitespace down to one and turns everything to lowercase.
2. After cleaning the text body the file was tokenized and split to sentences.
3. For each sentence, the punctuation was removed and the trailing and leading whitespace was striped.
4. The resulting sentences were also tokenized to words resulting to a list (the full text) containing many lists (the sentences) of strings (the words).

```
regex = re.compile('[%s]' % re.escape(string.punctuation))
sentences = [word_tokenize(regex.sub(' ', sent).strip()) for sent in sent_tokenize(clean_text(file_text))]


corpus += sentences
```

2. Set Splitting

The next step was to randomly split the corpus into training, validation, test1 and test2 sets. The splitting was done using sklearn's train_test_split function as seen below:

```
# 60% for train
training_idx, tuning_idx = train_test_split(sets, train_size=.6, random_state=2019)

# 20%,10%,10% for validation(development), test1 and test2 datasets.
validation_idx, test_idx = train_test_split(tuning_idx, train_size=.5, random_state=2019)
test1_idx, test2_idx = train_test_split(test_idx, train_size=.5, random_state=2019)

training_set_init = [corpus[i] for i in training_idx]
validation_set_init = [corpus[i] for i in validation_idx]
test1_set_init = [corpus[i] for i in test1_idx]
test2_set_init = [corpus[i] for i in test2_idx]
```

A 60 – 20 – 10 – 10 split was used to form the training, validation, test1 and test2 datasets.

3. Vocabulary and Rare Words

After forming the training set, the script iterated over all of its sentences and added all words to a list. A dictionary of word counts is then created by using the ***Counter()*** function from the ***Collections*** library. Finally the vocabulary was created by keeping all keys from the wordcount dictionary that had a count value greater than 10 as seen below:

```
AllWords = []

for sentence in training_set_init:
    AllWords += sentence


WordCounts = Counter(AllWords)

vocabulary = [k for k, v in WordCounts.items() if v > 10]
```

Then the script iterated over all sets and replaced words that did not exist in the vocabulary with the special “UNK” token.

```
valid_WordCounts = {k:v for k, v in WordCounts.items() if v>10}

for i in range(0,train_size):
    for j in range(0,len(training_set_init[i])):
        if training_set_init[i][j] not in valid_WordCounts:
            training_set[i][j] = 'UNK'
```

At this point the data are ready for n-gram modelling.

4. N-Gram Models

In order to form the language models, first all unigrams, bigrams and trigrams in the training set were counted and stored in a counter dictionary. Then 3 functions were created that given a n-gram, the vocabulary size and an alpha value as arguments, they calculated using the training set, a probability for the given n-gram. Using these functions the second task of the exercise was tackled as seen below:

A random seed was chosen and 10 different tests were executed. For each test a different random sentence was chosen from the test1 set. At the same time an invalid sentence of the same length was formed by choosing random words from the vocabulary. All models estimated a log probability for both the valid and the invalid sentences at each individual experiment. The code implementing these trials can be seen below:

```
## Test 10 random sentences in the test1 set against 10 sentences randomly created from the vocabulary

np.random.seed(666)

for i in range(0,10):

    # Get random normal sentence
    print('-------------------------------------------------------------------------------')
    print('Test ',str(i))
    print('Normal sentence results:')
    sentence_idx = np.random.randint(low = 0, high = len(test1_set))

    valid_sentence = test1_set[sentence_idx]
    print(' '.join(valid_sentence))

    print_sentence_unigram_probs(valid_sentence,V,C)
    print_sentence_bigram_probs(valid_sentence,V)
    print_sentence_trigram_probs(valid_sentence,V)

## VS a randomly created non-sense sentence
    print()
    print('Invalid sentence results:')

    random_sent_idx = np.random.randint(low = 0, high = len(vocabulary), size = len(valid_sentence))

    invalid_sentence = [vocabulary[idx] for idx in random_sent_idx]

    print(' '.join(invalid_sentence))
    print_sentence_unigram_probs(invalid_sentence,V,C)
    print_sentence_bigram_probs(invalid_sentence,V)
    print_sentence_trigram_probs(invalid_sentence,V)
    print('-------------------------------------------------------------------------------')
```

The results can be found on the “Experimental Results” file on this link.

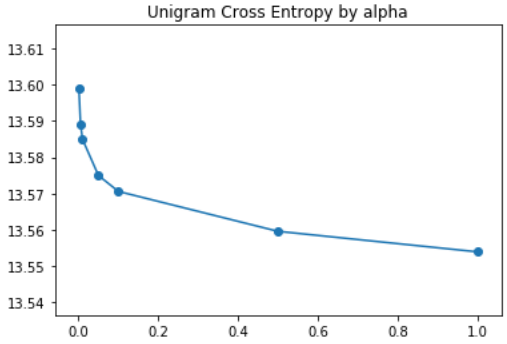
It’s clear that the log probabilities for the invalid sentences are lower than the ones for the valid sentences, indicating that the models work as expected.

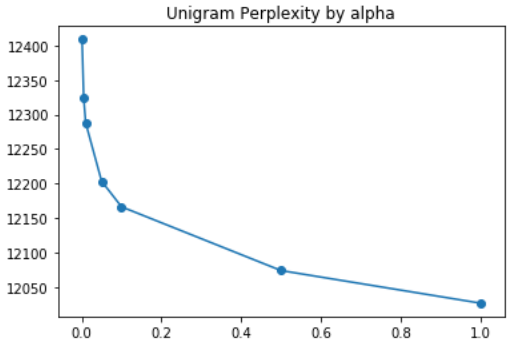
5. Cross Entropy and Perplexity Estimation

In order to choose the best language model, we tuned the smoothing parameter alpha by choosing different values and evaluating our models on the resulting Cross Entropy and Perplexity. The results are seen below:

For **Unigrams:**

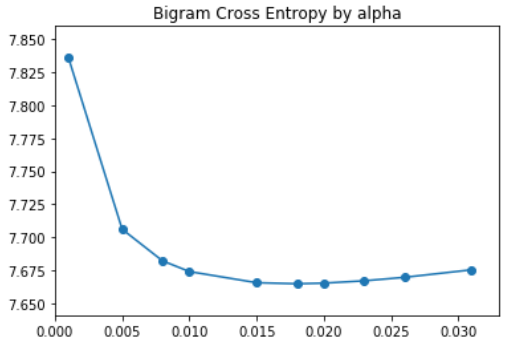
The Unigram test results can be found on the Unigrams.txt file on this link.

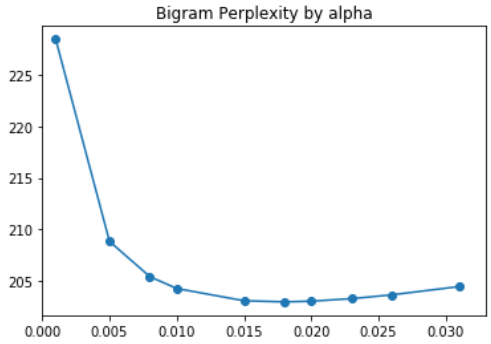




For **Bigrams**:

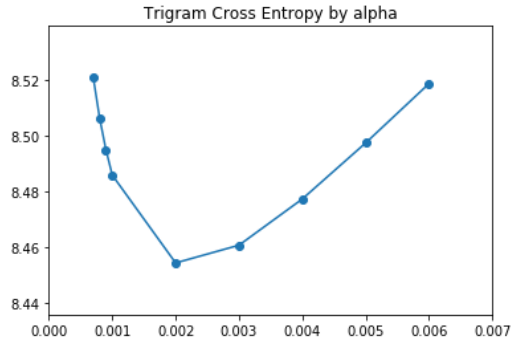
The Bigram test results can be found on the Bigrams.txt file file on this link.

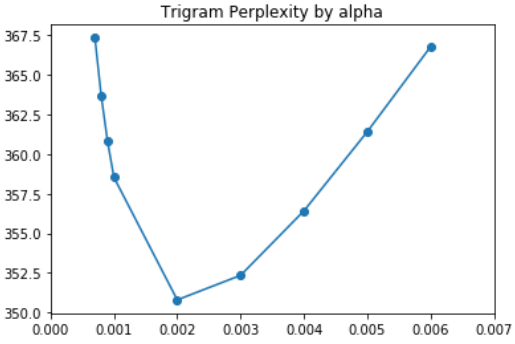




For **Trigrams**:

The Trigram test results can be found on the Trigrams.txt file on this link.





So after tuning alpha on the validation set we concluded on choosing the value that minimized Cross Entropy & Perplexity for each respective model as seen below:

- alpha = 1 for the Unigram model yielding a Cross Entropy of 13,554 and a perplexity of 12028,308 on the test1 set
- alpha = 0,018 for the Bigram model yielding a Cross Entropy of 7.663 and a perplexity of 202.640 on the test1 set
- alpha = 0.002 for the Trigram model yielding a Cross Entropy of 8.453 and a perplexity of 350.455 on the test1 set

6. Linear Interpolation

Finally the bigram and trigram models were combined using linear interpolation and tuning the lambda parameters on the validation set. However after using different values for the parameters, we concluded that the model performs best when only the bigram model is used (lambda = 0 for the trigram model as seen below:

The interpolation test results can be found on the Interpolation Results.txt file on this link.

So the final model used to evaluate the test2 set was the bigram model, scoring a total of 7.662 **Cross Entropy** and 202,51 **Perplexity**.