NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Speech and Language Processing

Lab 1: Introduction to language representations

LAB PROCEDURE

Explain your work comprehensively and adequately. Code without comments will not be graded. Collaboration within groups of 2 people is allowed if they belong to the same programme (either undergraduate or postgraduate groups). Each group of 2 people submits a joint report that represents only the personal work of its members. If you use a source other than the textbooks and course materials, you should mention it. The delivery of the report and the code of the work will be done electronically at helios. **Please note that it is forbidden to post the solutions of the laboratory exercises on github, or on other websites.**

HELPER CODE / MATERIAL

In the following link, you can find the examples we saw during the preparatory lab presentation. On the following page you can find helper code for the labs. On this page you can submit questions and queries to the teaching assistants in the form of issues. Questions regarding the laboratory that will be asked by mail will not be answered.

LAB DESCRIPTION

The purpose of the 1st lab exercise is to get acquainted with the use of classical and widely used linguistic representations for natural language processing.

The exercise is divided into two parts.

In the first part we will use simple language models and transformations to create a spell corrector based on finite state machines. The openfst library will be used.

You can install Openfst using the following script:
https://github.com/slp-ntua/slp-labs/blob/master/lab1/install_openfst.sh

Openfst Documentation: http://www.openfst.org/
Openfst Examples: http://www.openfst.org/twiki/bin/view/FST/FstExamples.

We will use Python and Bash scripting.
For familiarization with Python and Bash you can refer to the preparatory labs: https://github.com/slp-ntua/prep-lab

ATTENTION: Use OpenFST version 1.6.1. Next versions make breaking changes for fst composition and other operations.

LAB EXECUTION

PART 1: SPELL CORRECTOR

Step 1: Corpus construction

In this step we will create a small corpus using open resources available on the internet.
The Gutenberg project is a collection of books in the public domain and a good source for fast data collection for the construction of language models.

a) You can use the following script https://github.com/slp-ntua/slp-labs/blob/master/lab1/scripts/fetch_gutenberg.py.
This script downloads the gutenberg corpus subset available in the NLTK library.
This corpus will be used to extract statistics during the construction of language models.
We also perform preprocessing to clean the corpus. Read and run the script. Briefly comment on the pre-processing steps. Comment on cases where we would like to keep punctuation and avoid this aggressive preprocessing.
b) You can also expand this corpus with more books or texts from other sources to build a larger corpus. What are the advantages of this technique (apart from the size of the data). Mention at least 2.

Step 2: Lexicon construction

In this step we will build a dictionary that will be used to build the spelling corrector.
a) Create a dictionary that contains as keys all the unique tokens in the corpus that we created in Step 1. The values should be the number of occurrences of the corresponding token in the text.
b) Filter all tokens that appear less than 5 times. Why is this step needed?
c) Write the dictionary in the vocab/words.vocab.txt file in two tab separated columns, where the first column contains the tokens and the second column the corresponding number of occurrences.

Step 3: Input/Output symbol construction

a) For the FST construction, we need files that map the input (or output) symbols to unique indices. Write a Python function that assigns each lowercase character of the English language to an integer index (serial). The first symbol with index 0 is ε (<eps>). The result will be written to the vocab/chars.syms file in this format
http://www.openfst.org/twiki/pub/FST/FstExamples/ascii.syms
b) Similarly, create the words.syms file that maps each word (token) from the vocabulary you created in Step 2 to a unique index. The result must be written in vocab/words.syms file.

Step 4: Edit distance transducer construction

To create the spell corrector we will use a transducer **L** based on the Levenshtein distance (https://en.wikipedia.org/wiki/Levenshtein_distance).

We will model 3 types of edits: character insertions, character deletions and character substitutions. Each of these edits is associated with a cost. At this stage we will assume that all edits have a cost of 1.

a) Construct a transducer **L** with a single state that implements the Levenshtein distance by assigning:
1) each character to itself with weight 0 (no edit)
2) each character to ε with weight 1 (deletion)
3) ε to each character with weight 1 (insertion)
4) each character to any other character with weight 1 (substitution).

How does this converter transform an input word if we take the shortest path?
b) Save the fsts/L.fst file that contains the description of the transducer in Openfst text format.
c) Use fstcompile to compile L and save the result to fsts/L.binfst.
d) Give an example of other possible edits we could include.
e) Give examples of how we could improve the weights of the edits by introducing prior knowledge.
g) Use fstdraw to draw L. You can use a small subset of characters for easier visualization.

Hint: For d) and e) think often about spelling / typing errors.

Step 5: Build the dictionary acceptor

a) Construct an acceptor **V** with an initial state that accepts every word in the dictionary from Step 2. The weights of all edges are 0. This is an acceptor who simply accepts a word if it belongs in the dictionary.
b) Use fstrmepsilon, fstdeterminize and fstminimize to optimize the model. Describe each of these operations and their benefits. Comment on the traversal complexity and number of edges for a deterministic and a non-deterministic automaton.
c) Save the acceptor description file to fsts/V.fst in Openfst text format.
d) Use fstcompile to compile the acceptor and save the file in fsts/V.binfst
e) Use fstdraw to draw **V**. You can use a small subset of words for easier visualization. Specifically draw **V** before fstrmepsilon, fstdeterminize and fstminimize operations as well as after each of them. Compare the resulting FSTs.
Hint: Use the chars.syms file for input symbols and words.syms for output symbols

Step 6: Putting it all together. Spell corrector construction

a) Combine the Levenshtein transducer **L** with the acceptor **V** of step 5a producing the min edit distance spell checker **S**. This transducer corrects the words without taking into account any linguistic information, with the criterion of making the least amount of possible conversions in the input word.
Analyze the behavior of this inverter

1) for the case where the edits have equal weights,
2) for different weights of edits (eg cost(insertion) = cost(deletion) = 1, cost(substitution) = 1.5)

Hint: use fstcompose
b) What are the possible predictions of the min edit spell checker if the (misspelled) words *cit* and *cwt* are given as input?
(Hint: You may need to call fstarcsort for the transducer outputs and / or acceptor inputs)

Step 7: Testing the spell corrector

a) Download this evaluation dataset https://github.com/slp-ntua/slp-labs/blob/master/lab1/data/spell_test.txt
b) The optimal correction is predicted based on the shortest paths algorithm for the transducer of Step 6.
Use the transducer to correct some of the words in the test set. At this point select the first 20 words and comment on the result. You can use this code as a basis:
https://github.com/slp-ntua/slp-labs/blob/master/lab1/scripts/predict.sh
What does this script do? Comment on the operations performed on fsts, the intermediate outputs and how this sequence of operations produces our corrected word.

LAB PREPARATION END

MAIN LAB: PART 1

In this part of the exercise we will try to improve the spelling corrector we created in the lab preparation by introducing linguistic information.

Βήμα 8: Computing the edit weights

In this step we will assign different costs for each edit operation of the Levenshtein transducer. But to calculate these different costs we will need some additional information.
For this reason we provide the corpus https://github.com/slp-ntua/slp-labs/blob/master/lab1/data/wiki.txt which contains common spelling mistakes from Wikipedia articles. The corpus has the following format, where each line contains a misspelled word and its correction.

```
abandonned      abandoned
aberation       aberration
abilityes       abilities
abilties        abilities
abilty          ability
```

a) Use a single example (abandonned-> abandoned). Create an acceptor **M** for the misspelled word (abandonned) and a transducer **N** for the corrected word (abandoned).
b) Create the transducer **MLN** through fst composition, by composing **M** with transducer **L** and the result with **N**.
c) Run fstshortestpath followed by fstprint as shown below

```
fstshortestpath MLN.fst |
   fstprint --isymbols=chars.syms --osymbols=chars.syms  --show_weight_one
```

The result shows you the path (transformations) that are followed to turn the misspelled word into the correct one.

```
11     10     a      a      0
0      0
1      0      <eps>  <eps>  0
2      1      d      d      0
3      2      e      e      0
4      3      n      n      0
5      4      n      <eps>  3
6      5      o      o      0
7      6      d      d      0
8      7      n      n      0
9      8      a      a      0
10     9      b      b      0
```

The last column shows the cost of each edit. You can ignore the lines that have zero cost and save only the actual edits using the grep and cut commands.
You are given scripts/word_edits.sh which performs the above procedure for a pair of words.

```
❯ bash scripts/word_edits.sh abandonned abandoned
n      <eps>
# source target
```

Read and comment on the operation of this script and run it for some input pairs.
d) Write a python script that will perform the procedure of step 8c for each line of wiki.txt to produce all the edits. Save the edits to a file.
e) Extract the frequency of each edit and save it in a dictionary with the tuples (source, target) as keys and the corresponding frequencies as values.
f) Repeat Step 4, using the negative logarithm of the frequency of the respective edit as edge weights. If this edit has not appeared in the corpus you can put a very large number as a weight (infinite cost). This is transducer **E**.
g) Repeat Steps 6 and 7 using **E** instead of **L** to construct and test the **EV** spell corrector.

Step 9: Including word frequencies (Unigram word model)

In this step we will introduce the word occurrence frequencies in the model, so that the spell corrector suggests more likely words in its corrections.
a) Use the dictionary with the word frequencies you created in Step 2.
b) Construct the acceptor **W** which consists of a state and maps each word to itself using the negative logarithm of the word occurrence frequency as weight (unigram language model).
c) Build the **LVW** spell correcto by combining the Levenshtein transducer **L** with the vocabulary acceptor **V** and the language model **W**. Do not forget to use fstrmepsilon, fstdeterminize, fstminimize and fstarcsort where needed.
d) Construct the **EVW** spell corrector by performing the composition of the Levenshtein transducer **E** with the vocabulary acceptor **V** and the language model **W**.
e) Evaluate the **LVW** spell corrector using the procedure from Step 8.
f) Compare **LVW** results with **LV**. What is the correction suggested by each transducer if we enter the tokens "cwt" and "cit" in the input? Why;
g) Use fstdraw to draw **W** and **VW**. You can use a small subset of words for easy visualization.

Step 10: Spell corrector evaluation

You are given the following test set for evaluation
https://github.com/slp-ntua/slp-labs/blob/master/lab1/data/spell_test.txt, and the script
https://github.com/slp-ntua/slp-labs/blob/master/lab1/scripts/run_evaluation.py.
Use run_evaluation.py to compare the results of spell correctors **LV, LVW, EV, EVW** and compute the accuracy of each spell corrector on the test set. Comment on the results and compare the behavior of each spell corrector.

Step 11: (Bonus) Advanced Improvements

a) In step 8f we used infinite cost for edits that do not appear in the Wikipedia corpus. A possible improvement of the model is to perform smoothing on the probabilities of edits that do not appear. In this Step you are tasked to implement a simple Add-1 smoothing (Jurafsky 3.4.1). The excess probability mass can be divided for the edits with zero occurrence frequency.
Build and evaluate the **EV** spell corrector using additive smoothing for edits with zero occurrence frequency. Use the entire corpus for evaluation as in Step 10.
b) The size and quality of the dictionary, as well as the domain from which it results, play a big role in the performance of the spelling corrector. Try dictionaries of different sizes from different / larger corpora. You can use external sources to obtain these dictionaries such as the following, which is created from opensubtitles (https://github.com/hermitdave/FrequencyWords/blob/master/content/2016/en/) or lexica from wikipedia articles https://en.wiktionary.org/wiki/Wiktionary:Frequency_lists. Comment on the effect of the lexicon size, quality and domain.
c) Describe and implement other possible improvements.

MAIN LAB PART 2: Familiarization with word2vec

In the first part of the lab exercise we mainly consider syntactic models for the construction of a spelling corrector. Here we will deal with the use of semantic word representations and specifically with the Word2Vec model. We will first train our own model on the Gutenberg texts. Next we will compare it with a model trained by Google. The comparison will be made in the first phase intuitively based on your (human) perception of language.
Next we will build an emotion classifier. To build such a model we will use data from movie reviews on the IMDB website. The purpose is to categorize the reviews into positive and negative in terms of emotion. The models you will build will accept as input the linguistic representations of either your own model or the pre-trained one. This experiment is an empirical comparison of the two representations.

Step 12: Extracting word2vec representations

Word embeddings can be used to represent words and sentences in machine learning pipelines. Pre-trained embeddings are used for this purpose. Word embeddings are dense representations that encode semantic features of a word based on the assumption that words with similar meanings appear in similar contexts.
In this step we will focus on word2vec embeddings. These embeddings result from a neural network with one layer which is used to predict a word based on its context (3-5 word window around it). This is called the CBOW model. Alternatively the network is called to predict the context based on the word (skip-gram model).
You can read the paper and the following article.

You can use this code to train your word2vec model
https://github.com/slp-ntua/slp-labs/blob/master/lab1/scripts/w2v_train.py

a) Read the corpus you made in Step 1 in a list of tokenized sentences.
b) Use gensim's Word2Vec class to train 100 dimensional word2vec embeddings based on the sentences in step 12a. Use window = 5 and 1000 epochs.
(examples: https://radimrehurek.com/gensim/models/word2vec.html)
c) Select the words "bible", "book", "bank", "water" from the dictionary and find the nearest words in the semantic space (using cosine similarity).

1) Are the results as good as you expected?
2) Do they improve if you resize the context window / number of epochs?
3) Why?
4) What would you do to improve the results?

d) Another property of word2vec is the ability to perform semantic analogies for concepts using algebraic operations. By semantic analogy we mean the following: if we have two pairs of concepts (a, b) and (c, d) then there is a semantic analogy if the sentence "a is for b what c is for d" is satisfied, e.g. "Athens is for Greece what Paris is for France."
In word2vec embeddings we can express this as an algebraic vector operation
v = w2v("Greece") - w2v("Athens") + w2v("Paris")
With this operation we expect the nearest word for the vector v to correspond to the word Paris.
In this step you are asked to find the resulting word using the word triplets ("girl", "queen", "king"), ("good", "taller", "tall") and ("france", "paris ","london ") using the word2vec embeddings you trained.
e) Download and read the pretrained GoogleNews vectors.
https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit
Hint:

```
model = KeyedVectors.load_word2vec_format('./GoogleNewsvectorsnegative300.bin', binary=True,
limit=NUM_W2V_TO_LOAD)
```

Use the limit argument to avoid out of memory errors.
στ) Repeat step 12γ using GoogleNews vectors. Compare the results.
ζ) Repeat step 12δ using GoogleNews vectors. Compare the results.

Step 13: Word embeddings visualization

Word2vec representations encode the semantic features of words in a multidimensional vector space. Visualizing these spaces is often a useful tool and provides intuition for how concepts are encoded. For visualization it is necessary to reduce the dimension of the semantic space to two or three dimensions using a dimensionality reduction algorithm. An algorithm that works well in practice is T-SNE (more here).
a) Experiment with the following tool. Give some examples and comment on the visualizations.
b) Save the word embeddings from the model you trained in Step 12 in a tab separated file named embeddings.tsv and the corresponding words in a file named metadata.tsv. The correct format is shown in the following example:

```
# embeddings.tsv
0       0       1
0       1       0
1       0       0
```

```
# metadata.tsv
word1
word2
word3
```

c) Load the .tsv files of step 13b) to the Embedding Projector and experiment with

1) dimensionality reduction methods
2) examples that are intuitively correct / incorrect.

Present the relevant results.

Step 14: Sentiment analysis using word2vec embeddings

A sentence can be represented as the average of the w2v vectors of each word it contains (Neural Bag of Words). In this Step we will use these representations for sentiment analysis in movie reviews.

a) Download the data here http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
b) Read and preprocess the data. We provide the helper code here:
https://github.com/slp-ntua/slp-labs/blob/master/lab1/scripts/w2v_sentiment_analysis.py
c) Use the embeddings you trained to construct NBOW representations for each sentence in the corpus. Train a Logistic Regression model for classification using the NBOW features. Comment on the results. Describe how you could improve model performance.
Hint: Use the zero vector to represent Out Of Vocabulary (OOV) words
δ) Repeat step 14c using Google News vectors. Compare the results.

DELIVERABLES

Create **a .zip file** which will contain 1) the folder lab1 from github with your implementations included 2) A .pdf file with your report. Do not include fst binary files in the .zip file. If you want us to check a binary file, share using a google drive link. The .zip file should be submitted in helios before the submission deadline.