

Επεξεργασία Φωνής & Φυσικής Γλώσσας

8ο Εξάμηνο 2022 – 2023

Lab Assignment 1: Solutions

Αστρεινίδης Ζαφείριος – el19053 Email: el19053@mail.ntua.gr

Ζαρίφης Στυλιανός – el20435 Email: el20435@mail.ntua.gr

Contents

ΜΕΡΟΣ 0: Preprocessing.....	2
Step 1: Corpora Construction	2
Step 2: Lexicon Construction	5
Step 3: Creating Input/Output symbols	7
Step 4: Creating Edit Distance Transducer.....	9
Step 5: Creating a Dictionary Acceptor.....	14
Step 6: Creating a Spell Checker	20
Step 7: Spell Checker Test.....	21
ΜΕΡΟΣ 1: Spell Checker Construction.....	23
Step 8: Edit Cost Calculation	23
Step 9: Introducing Word Occurrence Frequency (Unigram word model).....	28
Step 10: Spell Checkers Evaluation	32
Step 11: Spell Checker Improvements	34
ΜΕΡΟΣ 2: Familiarizing with W2V.....	37
Step 12: Word2Vec Representations.....	37
Step 13: Visualization of Word Embeddings	44
Step 14.....	52

ΜΕΡΟΣ 0: Preprocessing

Step 1: Corpora Construction

1.a Download Corpus

To script `fetch_gutenberg.py` κατεβάζει το Gutenberg corpus από την NLTK library και κάνει ένα preprocessing στο κείμενο. Τα βήματα αυτής της προεπεξεργασίας είναι:

1. Μετατροπή όλου του κειμένου σε lowercase
2. Αφαίρεση των σημείων στίξης
3. Αφαίρεση των αριθμών

* Ένα τέταρτο βήμα είναι η αφαίρεση των stopwords. Οι stopwords είναι λέξεις όπως `and`, `the` που παρατηρούνται με ιδιαίτερα μεγαλύτερη συχνότητα στα κείμενα και **δεν** προσδίδουν σημαντική πληροφορία για τις γειτονικές λέξεις. Πρακτικά, αυτές οι λέξεις θεωρούνται θόρυβος. Δε θα υλοποιήσουμε αυτό το βήμα, όμως, γιατί χρειάζονται για να φτιάξουμε τους ορθογράφους.

Ο σκοπός αυτής της προεπεξεργασίας είναι να μείνουμε με ένα πιο καθαρό κείμενο αφαιρώντας τον θόρυβο που επηρεάζει το language modeling.

Σημειώνουμε πως σε κάποιες περιπτώσεις τα σημεία στίξης πρέπει να παραμείνουν, για παράδειγμα σε ανάλυση ποιημάτων ή διαλόγων.

1.b Extending Corpus

Επιλέγουμε να επεκτείνουμε το corpus με περισσότερα κείμενα από τα corpora `genesis`, `shakespeare`, `movie_reviews`, τα οποία θεωρούμε ότι περιλαμβάνουν φυσιολογικές προτάσεις, όμοιες με εκείνες του corpus `gutenberg`.

Το πλεονέκτημα αυτής της επέκτασης του corpus είναι ότι πετυχαίνουμε να έχουμε περισσότερα και δείγματα λόγου και με μεγαλύτερη ποικιλομοφία. Αυτό θα βελτιώσει την ακρίβεια και την ικανότητα γενίκευσης των μοντέλων που θα φτιάξουμε. Ακόμα, μεγαλύτερο μέγεθος corpus σημαίνει περισσότερα training data και συνεπώς ποιοτικότερα μοντέλα.

Η σημασία των βημάτων της προεπεξεργασίας είναι η εξής:

1. **Κάθε λέξη με πεζά γράμματα:** Με την πεζογράμμιση όλων των λέξεων, εξασφαλίζουμε ότι όλες οι παραλλαγές των ίδιων λέξεων (π.χ. "η" και "The"), που έχουν την ίδια σημασία, συγκεντρώνονται σε ένα μόνο token για να μειωθεί το μέγεθος του λεξιλογίου.
2. **Αφαίρεση των σημείων στίξης:** Τα σημεία στίξης δεν θεωρούνται λέξεις με νόημα, επομένως (μπορεί να) εμποδίζουν τους αλγορίθμους να εντοπίζουν μοτίβα στα tokens. Η αφαίρεση των σημείων στίξης απλοποιεί τα δεδομένα και μειώνει το μέγεθος του λεξιλογίου.

3. **Αφαίρεση των stopwords:** Οι stopwords είναι πολύ κοινές λέξεις (π.χ. "and", "a" και "the") που δεν έχουν μεγάλη σημασία σε κάποιο context. Επιπρόσθετα, κάνουν πιο δδύσκολο τον εντοπισμό των λέξεων που είναι σημαντικές για την κατανόηση του κειμένου. η αφαίρεση των stop words μειώνει το μέγεθος του λεξιλογίου και βελτιώνει την ακρίβεια της ανάλυσης.

Αλλάξαμε τη συνάρτηση download_corpus για να κατεβάζουμε περισσότερα corpora, πέρα από το gutenberg.

```
import re
import sys

import contractions
import nltk

nltk.download('gutenberg')
nltk.download('genesis')
nltk.download('shakespeare')
nltk.download('movie_reviews')

def download_corpus(corpora= ["gutenberg", "genesis", "shakespeare",
"movie_reviews"],
                    funcs = [nltk.corpus.gutenberg.raw(),
                             nltk.corpus.genesis.raw(),
                             nltk.corpus.shakespeare.raw(),
                             nltk.corpus.movie_reviews.raw()]):

    rawTexts = []
    for i in range(len(corpora)):
        nltk.download(corpora[i])
        rawTexts.append(funcs[i])

    return rawTexts

def identity_preprocess(s):
    return s

def clean_text(s):
    s = s.strip() # strip Leading / trailing spaces
    s = s.lower() # convert to lowercase
    s = contractions.fix(s) # e.g. don't -> do not, you're -> you are
    s = re.sub("\s+", " ", s) # strip multiple whitespace
    s = re.sub(r"[^a-z\s]", " ", s) # keep only lowercase letters and spaces

    return s
```

```

def tokenize(s):
    tokenized = [w for w in s.split(" ") if len(w) > 0] # Ignore empty string
    return tokenized

def preprocess(s):
    return tokenize(clean_text(s))

def process_file(corpus, preprocess=identity_preprocess):
    lines = [preprocess(ln) for ln in corpus.split("\n")]
    lines = [ln for ln in lines if len(ln) > 0] # Ignore empty lines
    return lines

```

Όπως αναφέραμε πιο πάνω, η προεπεξεργασία των δεδομένων που μόλις κατεβάσαμε είναι πολύ σημαντική. Αποθηκεύουμε σε μια λίστα κάθε corpus μετά την προεπεξεργασία του.

```

rawCorpora = download_corpus()
preprocessedList = [process_file(rawCorpora[i], preprocess=preprocess) for i in range(len(rawCorpora))]

```

Ενώνουμε τις ανωτέρω λίστες σε 1 επειδή χρειάζεται σε μεταγενέστερο βήμα.

```

preprocessed = []
for innerList in preprocessedList:
    for elem in innerList:
        preprocessed.append(elem)

```

Αποθηκεύουμε ότι κάναμε σε ένα αρχείο για γρήγορη ανάκτηση.

```

with open('manyCorpora.txt', 'w') as file:
    for corpus in preprocessedList:
        for line in corpus:
            file.write(" ".join(line) + "\n")

```

Step 2: Lexicon Construction

Συνεχίζοντας από το προηγούμενο βήμα, μια διαδικασία εξίσου σημαντική κατά την προεπεξεργασία είναι η εξής:

Filtering out rare tokens: Λέξεις που εμφανίζονται λίγες φορές στο corpus τείνουν να είναι λιγότερο σημαντικές και είναι πιθανώς θόρυβος (όπως για παράδειγμα τυπογραφικά λάθη). Η απόρριψή των μειώνει το μέγεθος του λεξικού και είναι πιθανό να βελτιώσει το accuracy της ανάλυσης.

2.a Create a Histogram Dictionary

Σε αυτό το βήμα δημιουργούμε ένα dictionary object με keys τα μοναδικά tokens του προεπεξεργασμένου corpus και values τον αριθμό των εμφανίσεών τους.

```
from nltk.corpus import stopwords

dictionary = {}
final_dictionary = {}
noStopwordsDict = {}

# download the stopwords if necessary
nltk.download('stopwords')

with open('manyCorpora.txt', 'r') as file:
    # reading each line
    for line in file:
        # reading each word
        for word in line.split():
            # displaying the words
            # print(word)
            if(word in final_dictionary):
                final_dictionary[word] += 1
                if word not in stopwords.words('english'):
                    noStopwordsDict[word] += 1
            else:
                final_dictionary[word] = 1
                if word not in stopwords.words('english'):
                    noStopwordsDict[word] = 1
```

2.b Filtering out Rare Tokens

Απορρίπτουμε τα σπάνια tokens, όπως αναφέραμε προηγουμένως.

```
alphabet = []
# filter out words that occur less than 5 times
for k, v in final_dictionary.items():
    for char in k:
        if char not in alphabet:
            alphabet.append(char)
```

```

if v >= 5:
    final_dictionary[k] = v

alphabet = sorted(alphabet)

```

2.c Export Dictionary to the "Tab Separated Values" file vocab/words.vocab.txt

Αποθηκεύουμε το λεξικό σε ένα αρχείο για περεταίρω χρήση.

```

import os

# create the 'vocab' directory if it doesn't exist
if not os.path.exists('vocab'):
    os.mkdir('vocab')

# write to the 'words.vocab.txt' file
with open('vocab/words.vocab.txt', 'w') as f:
    for (k,v) in final_dictionary.items():
        f.write(k + '\t' + str(v) + '\n')

```

Ένα δείγμα από το λεξικό.

```

# print the 10 most frequent words
print("The 10 most frequent words in our corpora")
for i, (k, v) in enumerate(sorted(final_dictionary.items(), key=lambda item:
item[1], reverse=True)):
    print(f"{i+1}. {k}: \tfound {v} times")
    if i+1 == 10:
        break

```

Σημείωση: στις 10 πιο συχνές λέξεις φαίνεται το πρόβλημα που αναφέραμε πιο πάνω.

Κυριαρχούν οι stopwords που ελάχιστη αξία δίνουν στο context.

Και ένα shortcut για να φορτώνουμε το dictionary που φτιάξαμε κατευθείαν από το αρχείο vocab/words.vocab.txt.

```

final_dictionary={}
with open("vocab/words.vocab.txt", "r") as file:
    for line in file:
        key, value = line.strip().split("\t")
        final_dictionary[key] = int(value)

```

Step 3: Creating Input/Output symbols

Φτιάχνουμε τα αρχεία `chars.syms` και `words.syms` που αντιστοιχίζουν τα I/O symbols σε μοναδικούς integers. Με τη βοήθεια αυτών των αρχείων θα κατασκευάσουμε τους Finite State Transducers (FSTs).

Το `chars.syms` είναι η αντιστοίχιση των πεζών χαρακτήρων του αγγλικού αλφαβήτου σε αύξοντα ακέραιο. Το πρώτο σύμβολο είναι το (epsilon) με index 0.

Το `words.syms` είναι η αντιστοίχιση κάθε token από το vocabulary του βήματος 2 σε μοναδικό integer. Με αυτόν τον τρόπο γίνεται η αναπαράσταση ολόκληρων λέξεων με 1 σύμβολο (για κάθε λέξη) σε ένα FST.

3.a Creating the I/O symbols file

Φτιάχνουμε το I/O symbols file, όπως φαίνεται εδώ:

<http://www.openfst.org/twiki/pub/FST/FstExamples/ascii.syms>.

```
def createCharSymbolMap(filename):
    with open(filename, "w", newline='') as f:
        f.write("<epsilon> 0\n")
        for ascii_code in range(97, 123): # Loop over lower case letters a-z
            letter = chr(ascii_code)
            symbol_id = ascii_code - 96 # compute symbol ID starting at 1
            f.write(f"{letter} {symbol_id}\n")

createCharSymbolMap('vocab/chars.syms')
```

3.b Creating the word symbols file

Φτιάχνουμε το word symbols file, στο οποίο κάθε λέξη του vocabulary αντιστοιχίζεται σε έναν μοναδικό ακέραιο.

```
def createWordSymbolMap():
    # Read the vocab/words.vocab.txt file with UTF-8 encoding
    with open("vocab/words.vocab.txt", "r", encoding="utf-8") as f:
        # Create a dictionary from the file contents
        vocab_dict = {}
        for line in f:
            key, value = line.strip().split("\t")
            vocab_dict[key] = value

    # Write the words.syms file with UTF-8 encoding
    with open("vocab/words.syms", "w", newline='', encoding="utf-8") as f:
        # Write the epsilon symbol
        f.write("<epsilon>\t0\n")
        counter = 0
        # Write the rest of the words with their index
        for key in vocab_dict.keys():
            counter += 1
            f.write(key + "\t" + str(counter) + "\n")
```

```
# Write the words.syms file with UTF-8 encoding
with open("vocab/words2.syms", "w+", newline='') as f:
    # Write the epsilon symbol
    f.write("<epsilon> 0\n")
    counter = 0
    # Write the rest of the words with their index
    for key, value in vocab_dict.items():
        counter += 1
        f.write(f"{key} {str(counter)}\n")

createWordSymbolMap()
```

Step 4: Creating Edit Distance Transducer

4.a Construction of a transducer

O transducer \mathbf{L} έχει ένα state και υπολογίζει τη Levenshtein distance ταιριάζοντας:

1. Κάθε χαρακτήρα με τον εαυτό του με βάρος 0 (no edit)
2. Κάθε χαρακτήρα με το ϵ με βάρος 1 (deletion)
3. Το ϵ με κάθε χαρακτήρα με βάρος 1 (insertion)
4. Κάθε χαρακτήρα σε κάθε άλλο χαρακτήρα με βάρος 1

O transducer αυτός υπολογίζει ένα edit distance βάσει της Levenshtein distance. To shortest path σε αυτόν, δίνει το ελάχιστο πλήθος των απαιτούμενων edits που χρειάζονται για να μετατρέψουν ένα string σε ένα άλλο. Μίνι εξήγηση: Από τη στιγμή που κάθε edit operation κοστίζει 1 και no operation κοστίζει 0, μπορούμε να σχεδιάσουμε έναν βεβαρυμένο κατευθυνόμενο γράφο για τον οποίο κάθε κόμβος αναπαριστά ένα πιθανό state του edited string (edited με τις έως τότε αλλαγές) και κάθε ακμή αναπαριστά ένα πιθανό edit με βάρος το κόστος του. To shortest path από το αρχικό string στο target string είναι εκείνο με το ελάχιστο συνολικό κόστος.

H Levenshtein distance μεταξύ δύο strings a, b (μήκους $|a|, |b|$ αντίστοιχα) δίνεται από τη σχέση:

$$\text{lev}(a, b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \text{lev}(\text{tail}(a), \text{tail}(b)) & \text{if } a[0] = b[0], \\ 1 + \min \begin{cases} \text{lev}(\text{tail}(a), b) \\ \text{lev}(a, \text{tail}(b)) \end{cases} & \text{otherwise} \end{cases}$$

Source: https://en.wikipedia.org/wiki/Levenshtein_distance

Σε αυτό το βήμα κατασκευάζουμε ένα txt file που αναπαριστά τον Levenshtein Transducer. Χρησιμοποιούμε τη συνάρτηση getLevFST που δέχεται ορίσματα το target file, το αλφάβητο του Αυτομάτου και το κόστος που θα έχει κάθε transition εφόσον δεν είναι από έναν χαρακτήρα στον εαυτό του (σε αυτήν την περίπτωση το κόστος είναι 0). H συνάρτηση λειτουργεί ως εξής:

Διατρέχει όλα τα σύμβολα του αλφάβητου και για καθένα από αυτά εκτελεί δύο διαδικασίες:

1. Γράφει στο αρχείο τις μεταβάσεις $\{c \rightarrow c, \epsilon \rightarrow c, c \rightarrow \epsilon\}$
2. Διατρέχει όλα τα υπόλοιπα σύμβολα του αλφάβητου και γράφει στο αρχείο τη μετάβαση $c \rightarrow c'$

Importing the util functions (αν και δεν τις χρησιμοποιούμε για την παραγωγή των FST αρχείων)

```

import numpy as np

EPS = "<epsilon>" # Define once. Use the same EPS everywhere

CHARS = list("abcdefghijklmnopqrstuvwxyz")

INFINITY = 1000000000

def calculate_arc_weight(frequency):
    """Function to calculate the weight of an arc based on a frequency count
    Args:
        frequency (float): Frequency count
    Returns:
        (float) negative Log of frequency
    """
    return -np.log(frequency)

def format_arc(src, dst, ilabel, olabel, weight=0):
    """Create an Arc, i.e. a line of an openfst text format file
    Args:
        src (int): source state
        dst (int): sestination state
        ilabel (str): input Label
        olabel (str): output Label
        weight (float): arc weight
    Returns:
        (str) The formatted line as a string
    http://www.openfst.org/twiki/bin/view/FST/FstQuickTour#CreatingShellFsts
    """
    return "{} {} {} {} {}\n".format(src, dst, ilabel, olabel, weight)

```

4.b Create and save the fst file

```

def getLevFST(file, alphabet, weight):
    f = open(file,"w+", newline='')
    for symbol in alphabet:
        # f.write("0 0 " + str(symbol) + " " + str(symbol) + " 0\n")
        # f.write("0 0 <epsilon> " + str(symbol) + " " + str(weight) + "\n")
        # f.write("0 0 " + str(symbol) + " <epsilon> " + str(weight) + "\n")

        f.write(format_arc(src=0, dst=0, ilabel=<epsilon>, olabel=symbol,
weight = weight))
        f.write(format_arc(src=0, dst=0, ilabel=symbol, olabel=symbol, weight
= 0))
        f.write(format_arc(src=0, dst=0, ilabel=symbol, olabel=<epsilon>,
weight = weight))
        for symbol2 in alphabet:
            if(symbol2 != symbol):
                # f.write("0 0 " + str(symbol) + " " + str(symbol2) + " "
+

```

```

str(weight) + "\n")
f.write(format_arc(src = 0, dst = 0, ilabel=symbol,
olabel=symbol2, weight = weight))
f.write('0\n')
f.close()

```

Δημιουργούμε 3 Levenshtein FSTs:

1. Για αλφάβητο του επεκτεταμένου Corpus
2. Για 3 χαρακτήρες
3. Για 1 χαρακτήρα

```

getLevFST('fstst/L.fst', alphabet, 1)
getLevFST('fstst/smallLevFST.fst', alphabet[0:3], 1)
getLevFST('fstst/oneLetterLevFST.fst', alphabet[0], 1)

```

4.c Compiling L FST

Παραθέτουμε το bash script που παράγει Levenshtein FST `createLevFST.sh`. Λαμβάνει ως όρισμα το όνομα που θέλουμε να δώσουμε στο FST. Έχει, όμως, τον περιορισμό ότι προσδιορίζει και τα ονόματα των αρχείων που θα χρησιμοποιήσει για την κατασκευή του. Δηλαδή, για το Βήμα 4, θέλουμε να κατασκευάσουμε το αυτόματο `L.fst`. Το script, για να κάνει `compile` το `fst`, θα ψάξει τα `isymbols` στη θέση `vocab/chars.syms`, τα `osymbols` πάλι στη `vocab/chars.syms` και την περιγραφή του `FST` στη `fstst/L.fst`. Υπτερα θα εκτελέσει την `fstprint` με εισόδους ίδια `symbols` και το αρχείο `fstst/L.fst` και έξοδο το `printFST/"$name".txt`. Προσπερνάμε το βήμα που τυπώνουμε το Levenshtein FST γιατί απαιτεί πολύ χρόνο (το τυπώνουμε όμως επειδή το τρέξαμε μία φορά, όπως και τυπώνουμε το ίδιο `FST` με λιγότερα σύμβολα).

```

#!/bin/bash

name=$1
fstcompile -isymbols=vocab/chars.syms -osymbols=vocab/chars.syms
fstst/"$name".fst fstst/"$name".binfst

fstprint -isymbols=vocab/chars.syms -osymbols=vocab/chars.syms
fstst/"$name".binfst > printFST/"$name".txt

# fstdraw -isymbols=vocab/chars.syms -osymbols=vocab/chars.syms -portrait
fstst/"$name".binfst | dot -Tpng > imageFST/"$name".png

```

4.d Possible Edits to Include

Μπορούμε να συμπεριλάβουμε edits που δεν είναι της μορφής χαρακτήρας/`epsilon` -> χαρακτήρας/`epsilon`. Ένα παράδειγμα, είναι η αντιστροφή δύο γειτονικών χαρακτήρων.

Αν αυτή η αντιστροφή έχει βάρος ένα, τότε μπορούμε να μετατρέψουμε τη συμβολοσειρά `savlation` στη σωστά ορθογραφημένη λέξη `salvation`.

4.e Weight Improvement

1. Αν μια μετατροπή εμφανίζεται σπάνια, τότε μπορούμε να της βάλουμε μεγαλύτερο βάρος.

Π.χ. με είσοδο τη συμβολοσειρά yellow και η μετατροπή a->e έχει μικρότερο βάρος από τη μετατροπή a->q τότε ο αλγόριθμος συντομότερων μονοπατιών θα αποφύγει τη διερεύνηση των συμβολοσειρών με πρόθημα για που είναι μάλλον αδύνατον να οδηγούν σε σωστή λέξη

2. Μπορούμε να έχουμε μικρότερο βάρος σε μετατροπές μεταξύ γραμμάτων που αντιστοιχούν σε γειτονικά πλήκτρα στο πληκτρολόγιο.

Π.χ. μετατροπές m -> n και a -> s συμβαίνουν πολύ συχνότερα από τις q -> p και z -> m

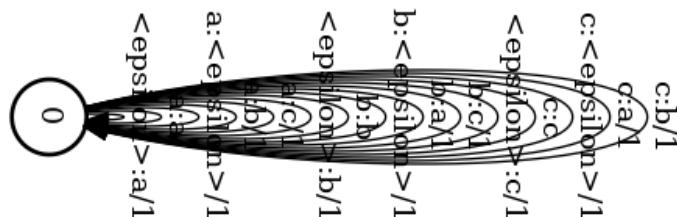
4.f Drawing the FSTs

```
from PIL import Image
from IPython.display import display

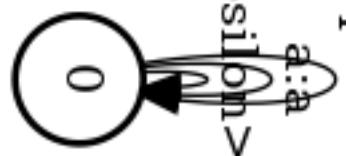
images = ['imageFST/L.png', 'imageFST/smallLevFST.png',
'imageFST/oneLetterLevFST.png']
sizes = ["26", "3", "1"]
for i in range(len(images)):
    with Image.open(images[i]) as img:
        print("Projecting Levenshtein FST with " + sizes[i] + "-letter
alphabet")
        img = img.rotate(270, expand=True)
        display(img)
```

Projecting Levenshtein FST with 26-letter alphabet

Projecting Levenshtein FST with 3-letter alphabet



Projecting Levenshtein FST with 1-letter alphabet



a:<epsilon>/1
a:a
<epsilon>:a/1

Step 5: Creating a Dictionary Acceptor

Στο βήμα αυτό θα κατασκευάσουμε ένα FST που αποδέχεται κάθε λέξη του λεξικού.

```
# STEP 5
final_dictionary={}
with open("./vocab/words.vocab.txt","r") as file:
    for line in file:
        key, value = line.strip().split("\t")
        final_dictionary[key] = value
```

5.a. Δημιουργούμε ένα FST με την ακόλουθη ιδέα:

1. Αρχικό state το 0. Εκεί ξεκινά η ανάγνωση της λέξης.
2. Δεσμεύουμε τα states 1 και 2 (και όχι τους τελευταίους αριθμούς που θα χρησιμοποιήσουμε) ως accepting και rejecting states. Κάθε λέξη που θα αποδέχεται το FST θα καταλήγει στο state 1, ενώ αν η λέξη που δίνεται ως input δεν υπάρχει, θα καταλήγουμε στο state 2. Η επιλογή αυτών των αριθμών γίνεται επειδή δε γνωρίζουμε το μέγεθος του Corpus, αφού μπορεί να αλλάζει ανάλογα με την εφαρμογή και τους πόρους.
3.
 - Για κάθε λέξη του λεξικού μας δημιουργούμε και από ένα state. Το transition από το 0 στο πρώτο state της λέξης γίνεται με το πρώτο γράμμα της τελευταίας. Το βάρος του transition είναι μηδενικό διότι προς το παρόν θέλουμε απλώς αποδοχή της λέξης. Το output του πρώτου transition είναι η ίδια η λέξη που θέλουμε να αποδεχτούμε.
 - Για κάθε επόμενο γράμμα που έρχεται από το input έχουμε και ένα transition με το ίδιο το γράμμα, μηδενικό βάρος και έξοδο το σύμβολο <epsilon>, αφού αν τελικά γίνει αποδοχή της τρέχουσας λέξης, την έχουμε επιστρέψει ως έξοδο του πρώτου transition.
4. Για κάθε λέξη, λοιπόν, δημιουργείται μια αλυσίδα από transitions με αρχή το state 0 και πέρας το state 1. Αν έρθουν οι χαρακτήρες της λέξης, το FST καλώς καταλήγει στο state 1 το οποίο έχει οριστεί ως accepting. Αν έρθουν χαρακτήρες που δεν ταιριάζουν σε κάποια λέξη, αυτό που συμβαίνει είναι να σταματήσει κάποια στιγμή η πορεία σε κάποια (ή κάποιες, αν υπάρχουν λέξεις με το ίδιο πρόθεμα) αλυσίδα, οπότε το FST καλώς δε θα καταλήξει ποτέ στο accepting state.
5. Η περίπτωση στην οποία δεν αναφερθήκαμε στα προηγούμενα points είναι να έρθει input με πρόθεμα μία υπάρχουσα λέξη και ένα επίθεμα που το καθιστά invalid. Σε αυτήν την περίπτωση χρησιμοποιούμε το state 2 ως rejecting state. Με ένα transition από το accepting state 1 με οποιονδήποτε χαρακτήρα, καταλήγουμε στο 2, οπότε πάλι καλώς απορρίπτουμε τη λέξη.

```
alphabet=['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q',
'r','s','t','u','v','w','x','y','z']
stateNo = 3
with open("./fsts/V.fst","w+", newline='') as f:
    test_dictionary = {"cat": 1, "cats": 55, "dog" : 5135000}
    for word in final_dictionary.keys():
```

```

        f.write("0 " + str(stateNo) + " " + word[0] + " " + word + " 0")
        f.write("\n")
        stateNo += 1
        for char in word[1:-1]:
            f.write(str(stateNo - 1) + " " + str(stateNo) + " " + char + " "
<epsilon> 0")
            f.write("\n")
            stateNo += 1
            f.write(str(stateNo - 1) + " 1 " + word[-1] + " <epsilon> 0")
            f.write("\n")
        for char in alphabet:
            f.write("1 2 " + char + " " + char + " 0")
            f.write("\n")
    f.write("1")

def createCharsWords(vocab_dict):
    alphabet = []
    for key in vocab_dict.keys():
        for char in key:
            if char not in alphabet:
                alphabet.append(char)
    with open("vocab/chars.sysms", "w", newline='') as f:
        symbol_id = 0
        f.write("<epsilon> 0\n")
        #print("<epsilon> 0")
        for letter in alphabet:      # Loop over Lower case Letters a-z
            symbol_id += 1           # compute symbol ID starting at 1
            f.write(f"{letter} {symbol_id}\n")
            #print(f"{letter} {symbol_id}")
    # Write the words.sysms file with UTF-8 encoding
    with open("vocab/words.sysms", "w+", newline='') as f:
        # Write the epsilon symbol
        f.write("<epsilon> 0\n")
        #print("<epsilon> 0")
        symbol_id = 0
        for key in vocab_dict.keys():
            symbol_id += 1
            f.write(f"{key} {str(symbol_id)}\n")
            #print(f"{key} {str(symbol_id)}")
        for letter in alphabet:
            symbol_id += 1
            f.write(f"{letter} {str(symbol_id)}\n")
            #print(f"{letter} {str(symbol_id)})")

def createVFST(filename, dictionary):
    alphabet = []
    for key in dictionary.keys():
        for char in key:
            if char not in alphabet:
                alphabet.append(char)

```

```

createCharsWords(dictionary)
stateNo = 3
with open(filename, "w+", newline='') as f:
    for word in dictionary.keys():
        f.write("0 " + str(stateNo) + " " + word[0] + " " + word + " 0")
        f.write("\n")
        stateNo += 1
        for char in word[1:-1]:
            f.write(str(stateNo - 1) + " " + str(stateNo) + " " + char +
" <epsilon> 0")
            f.write("\n")
            stateNo += 1
        f.write(str(stateNo - 1) + " 1 " + word[-1] + " <epsilon> 0")
        f.write("\n")
    for char in alphabet:
        f.write("1 2 " + char + " " + char + " 0")
        f.write("\n")
f.write("1")

createVFST("./fst/V.fst", final_dictionary)

test_dictionary = {
    "lion" : 1,
    "tiger" : 1,
    "leopard" : 1,
    "jaguar" : 1,
    "cheetah" : 1,
    "lynx" : 1,
    "bobcat" : 1,
    "ocelot" : 1
}
createVFST("./fst/V.fst", test_dictionary)
createVFST("./fst/V.fst", final_dictionary)
# createCharsWords(vocab_dict = test_dictionary)

```

5.b. FST optimization

Παραθέτουμε και το bash script `modifyVFST.sh` που υλοποιεί το modification που είδαμε παραπάνω. Αυτό περιλαμβάνει τα εξής:

1. **`fstrmepison`**: Αφαίρεση των epsilon - transitions
2. **`fstdeterminize`**: Μετατροπή του FST από non deterministic σε deterministic
3. **`fstminimize`**: Ελαχιστοποίηση του FST

```
#!/bin/bash
```

```

name=$1
# Remove epsilons from the input FST
fstrmepison "$name".fst "$name"NoEps.fst

# Determinize the output FST of fstrmepison

```

```

fstdeterminize "$name"NoEps.fst "$name"NoEpsDeter.fst

# Minimize the output FST of fstdeterminize
fstminimize "$name"NoEpsDeter.fst "$name"NoEpsDeterMin.fst

# Print the modified FST
fstprint -isymbols=vocab/chars.sym -osymbols=vocab/words.sym
fst/"$name".binfst > printFST/"$name".txt

# Draw the modified FST
fstdraw -isymbols=vocab/chars.sym -osymbols=vocab/words.sym -portrait
fst/"$name".binfst | dot -Tpng > imageFST/"$name".png

```

Το όφελος είναι ότι μειώνεται η πολυπλοκότητα διέλευσης του FST και άρα το τελευταίο γίνεται πιο αποδοτικό.

Σε ένα αυτόματο, η πολυπλοκότητα σχετίζεται με τον αριθμό βημάτων της διάσχισης ενός μονοπατιού από την αρχική ως την τελική κατάσταση.

Σε ένα μη ντετερμινιστικό αυτόματο, μπορεί να υπάρχουν πολλαπλές transitions από ένα state για κάποιο σύμβολο εισόδου, το οποίο αυξάνει την πολυπλοκότητα διάσχισης. Αυτό συμβαίνει επειδή το αυτόματο μπορεί να επιλέξει όλες τις πιθανές transitions και πιθανώς να οδηγήσει σε combinatorial explosion [1] του αριθμού των μονοπατιών που πρέπει να εξερευνηθούν. Επιπλέον, τα μη ντετερμινιστικά αυτόματα έχουν epsilon - transitions, δηλαδή μεταβάσεις χωρίς την αξιοποίηση συμβόλου εισόδου, οι οποίες προσθέτουν πρόσθετα μονοπάτια.

Μετατρέποντας το αυτόματο σε ντετερμινιστικό, έχουμε μόνο μία μετάβαση για δεδομένο σύμβολο εισόδου και κατάσταση, άρα μειώνεται η πολυπλοκότητα.

Συμπερασματικά, προτιμάμε τα ντετερμινιστικά αυτόματα, αφού επεξεργάζονται τις εισόδους αποδοτικότερα και ταχύτερα.

[\[1\]: \[https://en.wikipedia.org/wiki/Combinatorial_explosion\]\(https://en.wikipedia.org/wiki/Combinatorial_explosion\)](https://en.wikipedia.org/wiki/Combinatorial_explosion)

5.c. Save the modified V.fst

Έγινε στο προηγούμενο κομμάτι με το bash script

5.d. Compile the modified FST to fst/V.binfst

Χρησιμοποιούμε ξανά το bash script για το compilation του FST.

```
#!/bin/bash
```

```

name=$1
fstcompile -isymbols=vocab/chars.sym -osymbols=vocab/chars.sym
textFST/"$name".txt fst/"$name".fst

fstprint -isymbols=vocab/chars.sym -osymbols=vocab/chars.sym

```

```

fststs/"$name".fst > printFST/"$name".txt

fstdraw -isymbols=vocab/chars.syms -osymbols=vocab/chars.syms -portrait
fststs/"$name".fst | dot -Tpng > imageFST/"$name".png

```

5.e. Οπτικοποίηση του FST V

Σχεδιάζουμε τη μορφή του FST V για ένα λεξικό που περιλαμβάνει 8 λέξεις:

1. lion
2. tiger
3. leopard
4. jaguar
5. cheetah
6. lynx
7. bobcat
8. ocelot

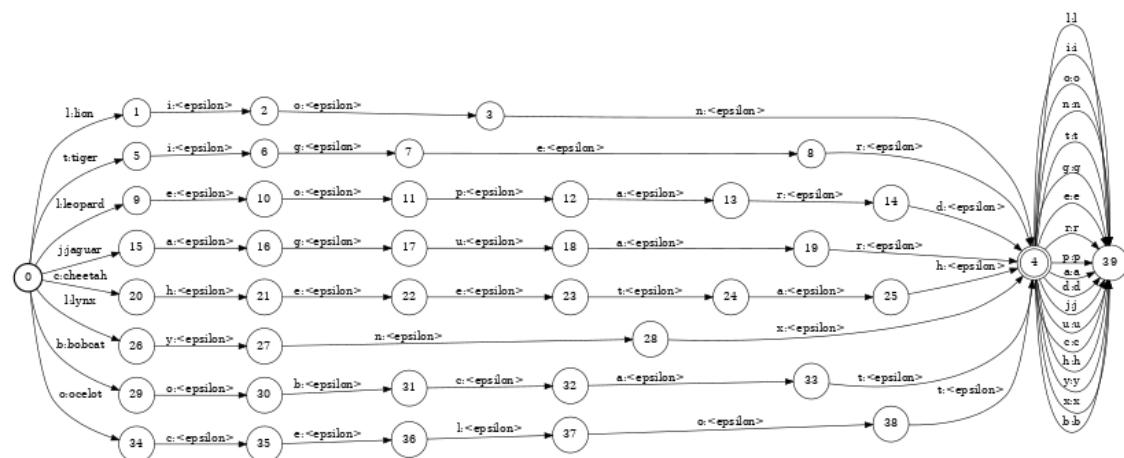
```

from PIL import Image
from IPython.display import display

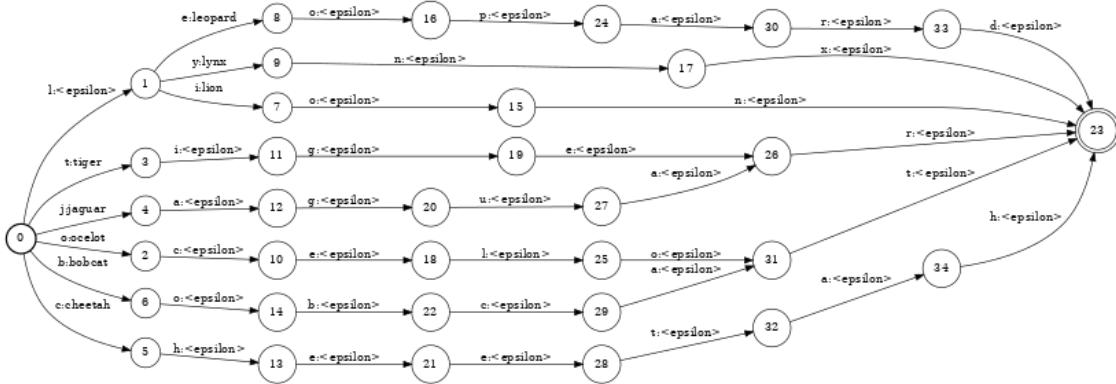
images = ['imageFST/VsmallDictionary.png',
'imageFST/VNoEpsDeterMinSmallDictionary.png']
labels = ["initial", "modified"]
for i in range(len(images)):
    with Image.open(images[i]) as img:
        print("Projecting " + labels[i] + " V FST")
        display(img)

```

Projecting initial V FST



Projecting modified V FST



Σημείωση: Παρατηρούμε ότι το rejection state που φτιάξαμε για το αρχικό FST δε
χρειάζεται. Αυτό διότι αν με είσοδο μια λέξη βρισκόμαστε σε ένα συγκεκριμένο μονοπάτι
και έρθει ένας χαρακτήρας για τον οποίο δεν υπάρχει transition, το FST θα απορρίψει (το
υπόλοιπο μονοπάτι, διότι αν 2 λέξεις έχουν ίδιο πρόθεμα, θα επιβιώσει το άλλο μονοπάτι -
σε μη ντετερμινιστικό FST)

Step 6: Creating a Spell Checker

6.a Composition of L & V Transducers

```
# compose(arg1, arg2) = arg3
first=$1
second=$2
outName=$3

# Sort the olabels of the first FST and the ilabels of the second FST
fstarcsort --sort_type=olabel fststs/"$first".binfst fststs/"$first"Sorted.binfst
fstarcsort --sort_type=ilabel fststs/"$second".binfst
fststs/"$second"Sorted.binfst

# Compose the FSTs to create the min edit distance spell checker
fstcompose fststs/"$first"Sorted.binfst fststs/"$second"Sorted.binfst
fststs/"$outName".binfst
```

Κάνουμε λοιπόν το composition από το Terminal ως:

```
./composeFST.sh L V S
```

Και ελέγχουμε το αποτέλεσμα με το script predict.sh ως εξής:

```
./predict.sh ../fststs/S.binfst abandoned
abandoned
```

6.b Possible Predictions for cit, cwt

```
./predict.sh ../S.binfst cit
it
```

```
./predict.sh ../S.binfst cwt
cut
```

Step 7: Spell Checker Test

7.a Downloading Evaluation data set

```
wget https://raw.githubusercontent.com/slp-ntua/slp-labs/master/lab1/data/spell_test.txt
```

7.b Spell Checker Testing

Χρησιμοποιούμε τις πρώτες 20 λέξεις για να δοκιμάσουμε τον ορθογράφο μας:

```
./predict.sh ../fstst/S.binfst contenpted  
contented  
./predict.sh ../fstst/S.binfst contende  
contend  
./predict.sh ../fstst/S.binfst contended  
contended  
./predict.sh ../fstst/S.binfst contentid  
contented  
  
./predict.sh ../fstst/S.binfst begining  
beginning  
  
./predict.sh ../fstst/S.binfst problam  
problem  
./predict.sh ../fstst/S.binfst proble  
problem  
./predict.sh ../fstst/S.binfst promblem  
problem  
./predict.sh ../fstst/S.binfst proplen  
problem  
  
./predict.sh ../fstst/S.binfst dirven  
direz  
  
./predict.sh ../fstst/S.binfst exstacy  
ecstasy  
./predict.sh ../fstst/S.binfst ecstacy  
ecstasy  
  
./predict.sh ../fstst/S.binfst guic  
guil  
./predict.sh ../fstst/S.binfst juce  
jude  
./predict.sh ../fstst/S.binfst jucie  
julie  
./predict.sh ../fstst/S.binfst juise  
guise  
./predict.sh ../fstst/S.binfst juse  
just
```

```
./predict.sh ../fstst/S.binfst localy  
locals  
  
./predict.sh ../fstst/S.binfst compair  
compter  
  
./predict.sh ../fstst/S.binfst pronounciation  
provocation
```

Με μια πρώτη ματιά, ο ορθογράφος φαίνεται να δουλεύει όπως περιμέναμε: Αν και δεν επιστρέφει την αναμενόμενη διορθωμένη λέξη, επιστρέφει την κοντινότερη (κατά Levenshtein) υπαρκτή στο lexicon λέξη.

ΜΕΡΟΣ 1: Spell Checker Construction

Step 8: Edit Cost Calculation

8.a Acceptor & Transducer for the words abandonned & abandoned

Αρχικα δημιουργούμε τον Acceptor για τη λέξη abandonned και τον Transducer για τη λέξη abandoned.

Αξιοποιούμε το δωσμένο script mkfstinput.py και δημιουργούμε τα FSTs:

```
python mkfstinput.py abandonned > ../fsts/M.fst
python mkfstinput.py abandoned > ../fsts/N.fst
```

8.b FSTs Composition

Κάνουμε compile τα FSTs που δημιουργήσαμε και τα συνθέτουμε: $MLN = (M \circ L) \circ N$

```
fstcompile --isymbols=vocab/chars_syms --osymbols=vocab/chars_syms fst/M.fst
fst/M.binfst
```

```
fstcompile --isymbols=vocab/chars_syms --osymbols=vocab/chars_syms fst/N.fst
fst/N.binfst
```

```
./composeFST.sh M L MoL
```

```
./composeFST.sh MoL N MLN
```

```
fstshortestpath fst/MLN.binfst | fstprint --isymbols=vocab/chars_syms --
osymbols=vocab/chars_syms --show_weight_one
```

Το αποτέλεσμα φαίνεται παρακάτω:

12	11	a	a	0
0	0			
1	0	<epsilon>	<epsilon>	0
2	1	<epsilon>	<epsilon>	0
3	2	d	d	0
4	3	e	e	0
5	4	n	n	0
6	5	n	<epsilon>	1
7	6	o	o	0
8	7	d	d	0
9	8	n	n	0
10	9	a	a	0
11	10	b	b	0

8.c Script Comments

Το script υπολογίζει την minimum edit distance ανάμεσα σε μια ανορθόγραφη λέξη και τη σωστά ορθογραφημένη χρησιμοποιώντας το Levenshtein FST που δημιουργήσαμε.

Αρχικά φτιάχνει ένα FST για την ανορθόγραφη λέξη και το συνθέτει με το L FST. Μέχρι στιγμής έχει υπολογίσει το minimum edit distance προς τη σωστή λέξη.

Υστερα συνθέτει το τελευταίο με το FST της ορθογραφημένης λέξης και τυπώνει τις edit operations που επέλεξε.

8.d Python Script to Automate 8.c

Αποθηκεύουμε στο αρχείο vocab/edits.txt το ζητούμενο.

```
import subprocess
import os
os.chdir('c:\\github\\nlpLabs\\NLP-LABS\\scripts')

counter = 0
with open("../vocab/wiki.txt", "r", newline='') as f:
    with open("../vocab/edits.txt", "wt", newline = '') as out_f:
        for line in f:
            counter += 1
            if counter % 430 == 0:
                print(str(counter//430) + "% completed...")
            # Split the line into misspelled and corrected words
            misspelled, corrected = line.strip().split('\\t')
            command = f"./word_edits.sh {misspelled} {corrected}"
            output = subprocess.run(['bash', '-c', command],
capture_output=True, text=True)

            # Split the output into individual edits
            output_str = output.stdout.strip()
            edits = output_str.split('\\n')

            # Save the edits to a file
            out_f.write('\\n'.join(edits) + '\\n')
```

Μερικά από τα edits που βρήκαμε με το τελευταίο script:

```
n      <epsilon>
<epsilon>  r
y      i
<epsilon>  i
<epsilon>  i
o      a
b      <epsilon>
<epsilon>  t
t      <epsilon>
c      <epsilon>
```

Για το ζεύγος abotu, about παρατηρούμε αλλαγή <epsilon> -> t και t -> <epsilon>. Θα μπορούσε να ήταν επίσης t -> u και u -> t, αφού έχουμε ορίσει τις αλλαγές ε σε κάποιον χαρακτήρα και χαρακτήρα σε κάποιον άλλο χαρακτήρα να έχουν το ίδιο βάρος.

8.e Edit Frequency

```
# Initialize an empty dictionary to store the edit counts
edit_counts = {}
with open("../vocab/edits.txt", "r", newline = '') as f:
    # Iterate over each line in the file
    for line in f:
        # Skip empty lines
        if not line.strip():
            continue
        # Strip the newline character and split the line into misspelled and
        # corrected words
        misspelled, corrected = line.strip().split('\t')

        # Concatenate the misspelled and corrected words into a tuple for use
        # as a dictionary key
        edit = (misspelled, corrected)

        # Increment the count for the edit in the dictionary
        edit_counts[edit] = edit_counts.get(edit, 0) + 1

# Print the dictionary of edit counts
print(edit_counts)
```

8.f Βελτίωση του L FST

Χρησιμοποιούμε για βάρη στον L τους αρνητικους λογαρίθμους των συχνοτήτων εμφανίσεων των λέξεων. Όποτε δεν υπάρχει η συγκεκριμένη λέξη στο λεξικό δίνουμε βάρος 100000 $\approx \infty$

```
import os
import numpy as np
os.chdir('c:\\github\\nlpLabs\\NLP-LABS')
sumChanges = sum(edit_counts.values())
with open("fsts/L.fst", 'r', newline = '') as f:
    with open("fsts/E.fst", 'w+', newline = '') as out_f:
        for line in f:
            if not line.strip():
                continue
            cols = line.strip().split()
            if len(cols) != 5:
                continue
            if cols[-3] == cols[-2]:
                out_f.write(f"{cols[0]} {cols[1]} {cols[2]} {cols[3]} {cols[4]}\n")
                print(f"{cols[0]} {cols[1]} {cols[2]} {cols[3]} {cols[4]}")
            elif (cols[-3], cols[-2]) in edit_counts.keys():
                out_f.write(f"{cols[0]} {cols[1]} {cols[2]} {cols[3]} {-np.log(edit_counts[(cols[-3], cols[-2])]/sumChanges)}\n")
                print(f"{cols[0]} {cols[1]} {cols[2]} {cols[3]} {-np.log(edit_counts[(cols[-3], cols[-2])]/sumChanges)})")
```

```

        else:
            out_f.write(f"{cols[0]} {cols[1]} {cols[2]} {cols[3]}
100000\n")
            print(f"{cols[0]} {cols[1]} {cols[2]} {cols[3]} 100000")
        out_f.write("0")
        print("0")

```

8.g Constructing and Testing the new Spell Checker

Ελέγχουμε

```
fstcompile -isymbols=vocab/chars.syms -osymbols=vocab/chars.syms fst/E.fst
fst/E.binfst
```

```
./composeFST.sh E V EV
```

```
./predict.sh ../fst/EV.binfst abandonned
```

Και το αποτέλεσμα είναι abandoned

Με το ακόλουθο bash script ελέγχουμε για τις πρώτες 20 λέξεις:

```
#!/usr/bin/env bash

# Define the list of words to spell check
WORDS=(
    "contenpted"
    "contende"
    "contended"
    "contentid"
    "begining"
    "problam"
    "proble"
    "promblem"
    "proplen"
    "dirven"
    "exstacy"
    "ecstacy"
    "guic"
    "juce"
    "jucie"
    "juise"
    "juse"
    "localy"
    "compair"
    "pronounciation"
)
# Loop through the words and execute the spell corrector for each one
for WORD in "${WORDS[@]}"
do
```

```
./predict.sh ../fst/S.binfst "${WORD}"
echo    # new line
done
```

Και τα αποτελέσματα:

```
./multiPredict.sh
contented
contend
contended
contented
beginning
problem
problem
problem
problem
problem
direz
ecstasy
ecstasy
guil
jude
julie
guise
just
locals
compter
provocation
```

Step 9: Introducing Word Occurrence Frequency (Unigram word model)

9.a Using dictionary created in step 2

```
print("Using final_dictionary:")
final_dictionary
```

9.b Constructing W FST

```
import numpy as np
import sys

sum=0
aux={}
with open("vocab/words.vocab.txt","r") as file:
    for line in file:
        key,value=line.strip().split("\t")
        sum=sum+int(value)
        aux[key]=value

f= open("fsts/W.fst", "w", newline='')
for (k,v) in aux.items():
    f.write(f"{str(0)}\t{str(0)}\t{k}\t{k}\t{-np.log(int(v)/sum)}\n")

for char in
['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s',
't','u','v','w','x','y','z']:
    f.write(f"{str(0)}\t{str(0)}\t{char}\t{char}\t{str(sys.maxsize)}\n")

f.write(str(0))
f.close()
```

9.c Composition of L, V, W FSTs

Κάνουμε compile και ελαχιστοποιούμε το FST που φτιάξαμε και ύστερα υπολογίζουμε τη σύνθεση $LVW = (L \circ V) \circ W$

```
fstcompile -isymbols=vocab/words.syms -osymbols=vocab/words.syms fst/fsts/W.fst
fst/fsts/W.binfst
```

```
./modifyVFST.sh W
```

```
./composeFST.sh L V LV
./composeFST.sh LV W LVW
```

9.d Composition of E, V, W FSTs

```
./composeFST.sh E V EV
./composeFST.sh EV W EVW
```

9.e Testing LVW FST

Χρησιμοποιώντας το bash script του βήματος 8 για τον LVW, έχουμε:

```

./multiPredict.sh
contented
contend
contended
contented
beginning
problem
problem
problem
people
given
ecstasy
ecstasy
guil
jude
julie
juice
just
local
company
provocation

```

9.f Comparing LV and LVW FSTs

Οι διαφορες των προβλέψεων στις πρώτες 20 λέξεις είναι οι εξής:

1. Με είσοδο "problem" το LV FST έδωσε "problem" ενώ το LVW FST έδωσε "people".
2. Με είσοδο "locals" το LV FST έδωσε "locals" ενώ το LVW FST έδωσε "local".
3. Με είσοδο "compter" το LV FST έδωσε "compter" ενώ το LVW FST έδωσε "company".

Οι υπόλοιπες προβλέψεις είναι ίδιες για τα 2 FSTs.

```

./predict.sh ../fstst/LVW.binfst cit
it

```

```

./predict.sh ../fstst/LVW.binfst cwt
cut

```

9.g Draw the FSTs for a smaller lexicon

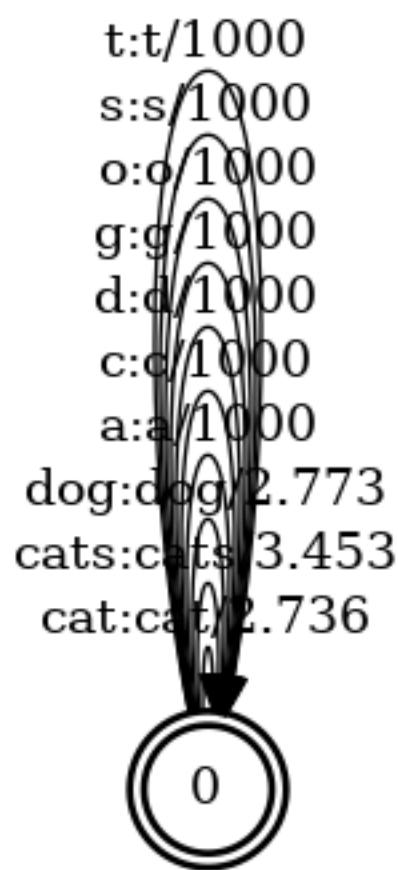
```

from PIL import Image
from IPython.display import display

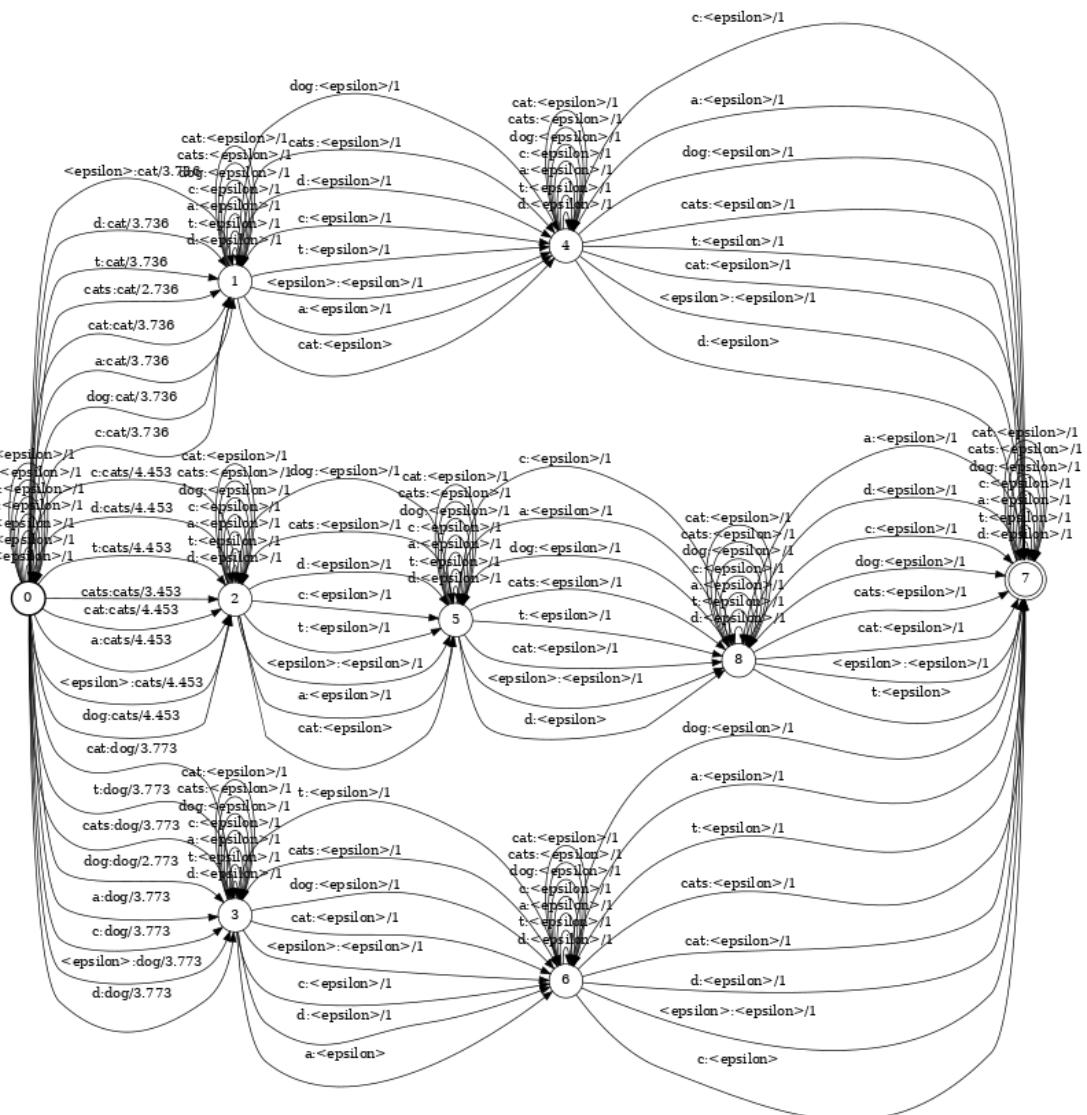
images = ['imageFST/smallWFST.png', 'imageFST/smallVWFST.png']
labels = ["initial", "modified"]
for i in range(len(images)):
    with Image.open(images[i]) as img:
        print("Projecting " + labels[i] + " V FST")
        #img = img.rotate(270, expand=True)
        display(img)

```

Projecting initial V FST



Projecting modified V FST



Step 10: Spell Checkers Evaluation

```
# Evaluate LV FST
python run_evaluation.py ../fsts/S.binfst

# Evaluate LVW FST
python run_evaluation.py ../fsts/LVW.binfst

# Evaluate EV FST
python run_evaluation.py ../fsts/EV.binfst

# Evaluate EVW FST
python run_evaluation.py ../fsts/EVW.binfst
```

Και έχουμε τα αποτελέσματα:

Για το LV FST:

```
1 contenpted -> contented: contented
2 contende -> contend: contented
3 contended -> contended: contented
4 contentid -> contended: contented
5 begining -> beginning: beginning
.....
269 supercede -> superhero: supersede
270 superceed -> superhero: supersede
271
100% |████████████████████████████████████████████████████████████████| 270/270 [56:22<00:00, 12.53s/it]
272 Accuracy: 0.5814814814814815
```

Για το LVW FST:

```
1 contenpted -> contented: contented
2 contende -> contend: contented
3 contended -> contended: contented
4 contentid -> contended: contented
5 begining -> beginning: beginning
.....
269 supercede -> superhero: supersede
270 superceed -> superhero: supersede
271
100% |████████████████████████████████████████████████████████████████| 270/270 [2:42:28<00:00, 36.11s/it]
272 Accuracy: 0.6407407407407407
```

Για το EV FST:

```
1 contenpted -> contented: contented
2 contende -> contender: contented
3 contended -> contended: contented
4 contentid -> contended: contented
```

```
5 begining -> beginning: beginning
.....
269 supercede -> suspected: supersede
270 superceed -> suspected: supersede
271
100% |██████████| 270/270 [3:52:54<00:00, 51.76s/it]
272 Accuracy: 0.7185185185185186
```

Για το EVW FST:

```
1 contened -> contented: contented
2 contende -> contender: contented
3 contended -> contended: contented
4 contentid -> contented: contented
5 begining -> beginning: beginning
.....
269 supercede -> suspected: supersede
270 superceed -> suspected: supersede
271
100% |██████████| 270/270 [3:52:54<00:00, 51.76s/it]
272 Accuracy: 0.7185185185185186
```

Παρατηρήσεις:

- Ο EV ειναι καλυτερος απο τον LV γιατί αντι για σταθερό κόστος σε κάθε edit, έχει για κόστη ελαττούμενα, συναρτήσει των πιθανοτήτων εμφάνισης του edit στο corpus. Έτσι τείνει να επιλέξει την πιο πιθανή αλλαγή και ότι τυχαία ανάμεσα σε εκείνες με ίσα κόστη!
- Ο LVW οδηγεί σε πιθανότερες λέξεις σε σχέση με τον LV από τη στιγμή που λαμβανει υπόψιν τις συχνότητες εμφάνισης των πιθανών συμβολοσειρών.
- Ομοίως, ο EVW οδηγεί σε πιθανότερες λέξεις σε σχέση με τον EV.

Step 11: Spell Checker Improvements

11.a Improving with Add1 - Smoothing

```
import os
import numpy as np
os.chdir('c:\\github\\nlpLabs\\NLP-LABS')
sumChanges = sum(edit_counts.values())
with open("fsts/L.fst", 'r', newline = '') as f:
    with open("fsts/Ladd1.fst", 'w+', newline = '') as out_f:
        for line in f:
            if not line.strip():
                continue
            cols = line.strip().split()
            if len(cols) != 5:
                continue
            if cols[-3] == cols[-2]:
                out_f.write(f"{cols[0]} {cols[1]} {cols[2]} {cols[3]} {cols[4]}\n")
            print(f"{cols[0]} {cols[1]} {cols[2]} {cols[3]} {cols[4]}")
            elif (cols[-3], cols[-2]) in edit_counts.keys():
                out_f.write(f"{cols[0]} {cols[1]} {cols[2]} {cols[3]} {-
np.log((edit_counts[(cols[-3], cols[-2])]) + 
1)/(sumChanges+len(edit_counts)))}\n")
                print(f"{cols[0]} {cols[1]} {cols[2]} {cols[3]} {-
np.log((edit_counts[(cols[-3], cols[-2])]) + 
1)/(sumChanges+len(edit_counts)))}")
            else:
                out_f.write(f"{cols[0]} {cols[1]} {cols[2]} {cols[3]} {-
np.log(1/(sumChanges+len(edit_counts)))}\n")
                print(f"{cols[0]} {cols[1]} {cols[2]} {cols[3]} {-
np.log(1/(sumChanges+len(edit_counts)))}")
                out_f.write(f"0")
                print(f"0")

./createLFST.sh Ladd1
./composeFST.sh Ladd1 V Sadd1
```

Και έχουμε τα αποτελέσματα για το LV FST με add1 - smoothing:

```
1 contenpted -> contented: contented
2 contende -> contend: contented
3 contended -> contended: contented
4 contentid -> contented: contented
5 begining -> beginning: beginning
.....
269 supercede -> superhero: supersede
270 superceed -> superhero: supersede
271
```

100% |██████████|

270/270 [1:51:14<00:00, 11.27s/it]
272 Accuracy: 0.5814814814814815

11.b Enhancing Lexicon with en_50k.txt

```

enhanced_dictionary = {}
with open("vocab/en_50k.txt","r") as file:
    for line in file:
        key, value = line.strip().split(" ")
        enhanced_dictionary[key] = int(value)

createVFST("./fsts/Venhanced.fst", enhanced_dictionary)

import numpy as np
import sys

sumVals=0
aux={}
with open("vocab/en_50k.txt","r") as file:
    for line in file:
        key,value=line.strip().split(" ")
        sumVals = sumVals + int(value)
        aux[key]=int(value)

f= open("fsts/Wenhanced.fst","w", newline=' ')
for (k,v) in aux.items():
    f.write(f"{str(0)}\t{str(0)}\t{k}\t{k}\t{-np.log(int(v)/sumVals)}\n")

for char in
['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s',
't','u','v','w','x','y','z']:
    f.write(f"{str(0)}\t{str(0)}\t{char}\t{char}\t{str(sys.maxsize)}\n")

f.write(str(0))
f.close()

fstcompile -isymbols=vocab/words_syms -osymbols=vocab/words_syms
fststs/Venhanced.fst fststs/Venhanced.binfst
fstcompile -isymbols=vocab/words_syms -osymbols=vocab/words_syms
fststs/Wenhanced.fst fststs/Wenhanced.binfst
./composeFST.sh L Venhanced LVenhanced
./composeFST.sh LVenhanced Wenhanced LVWenhanced

```

Και έχουμε τα αποτελέσματα για το LVW FST για το en_50k.txt:

```
1 contenpted -> contented: contented
2 contende -> contend: contented
3 contended -> contended: contented
4 contentid -> contented: contented
5 begining -> beginning: beginning
.....
269 supercede -> superhero: supersede
```

```
270 superceed -> superhero: supersede  
271  
100% |████████████████████████████████████████████████████████|  
270/270 [1:51:14<00:00, 11.27s/it]  
272 Accuracy: 0.5814814814814815
```

```
./composeFST.sh E VEnhanced EVenhanced  
./composeFST.sh EVenhanced Wenhanced EVWenhanced
```

Και έχουμε τα αποτελέσματα για το EVW FST για το en_50k.txt:!!!!!!!!!!!!!!

```
1 contenpted -> contented: contented  
2 contende -> contend: contented  
3 contended -> contended: contented  
4 contentid -> contented: contented  
5 begining -> beginning: beginning  
.....  
269 supercede -> superhero: supersede  
270 superceed -> superhero: supersede  
271  
100% |████████████████████████████████████████████████████████|  
270/270 [1:51:14<00:00, 11.27s/it]  
272 Accuracy: 0.5814814814814815
```

ΜΕΡΟΣ 2: Familiarizing with W2V

Step 12: Word2Vec Representations

12.a Creating list of Tokenized Sentences from Corpus preprocessed

12.b Training 100 - dimensional word2vec embeddings

```
import logging
import multiprocessing
import os

from gensim.models import Word2Vec
from gensim.models.callbacks import CallbackAny2Vec

# Enable gensim logging
logging.basicConfig(
    format="%(levelname)s - %(asctime)s: %(message)s",
    datefmt="%H:%M:%S",
    level=logging.INFO,
)

class W2VLossLogger(CallbackAny2Vec):
    """Callback to print Loss after each epoch
    use by passing model.train(..., callbacks=[W2VLossLogger()])
    """

    def __init__(self):
        self.epoch = 0

    def on_epoch_end(self, model):
        loss = model.get_latest_training_loss()

        if self.epoch == 0:
            print("Loss after epoch {}: {}".format(self.epoch, loss))
        else:
            print(
                "Loss after epoch {}: {}".format(
                    self.epoch, loss - self.loss_previous_step
                )
            )
        self.epoch += 1
        self.loss_previous_step = loss

    def train_w2v_model(
        sentences,
        output_file,
```

```

    window=5,
    embedding_dim=100,
    epochs=300,
    min_word_count=10,
):
    """Train a word2vec model based on given sentences.
Args:
    sentences list[list[str]]: List of sentences. Each element contains a
list with the words
        in the current sentence
    output_file (str): Path to save the trained w2v model
    window (int): w2v context size
    embedding_dim (int): w2v vector dimension
    epochs (int): How many epochs should the training run
    min_word_count (int): Ignore words that appear less than
min_word_count times
"""
# raise NotImplementedError("You should use gensim to train your w2v
model")
workers = multiprocessing.cpu_count()

# TODO: Instantiate gensim.models.Word2Vec class
model = Word2Vec(
    sentences,
    vector_size = embedding_dim,
    window = window,
    min_count = min_word_count,
    workers = workers,
)
# TODO: Build model vocabulary using sentences
# model.build_vocab(sentences)

# TODO: Train word2vec model
model.train(
    sentences,
    total_examples = len(sentences),
    epochs = epochs,
    callbacks = [W2VLossLogger()],
    compute_loss = True
)

# model.train(..., callbacks=[W2VLossLogger()])
# Save trained model
model.save(output_file)

return model

if __name__ == "__main__":

```

```

# read data/gutenberg.txt in the expected format
sentences = preprocessed
output_file = "w2vModels/corpus_w2v.100d.model"
window = 5
embedding_dim = 100
epochs = 100
min_word_count = 10

train_w2v_model(
    sentences,
    output_file,
    window=window,
    embedding_dim=embedding_dim,
    epochs=epochs,
    min_word_count=min_word_count,
)

```

Εκτελούμε τον παραπάνω κώδικα για 10, 100, 300, 1000 εποχές, με παράθυρο 5 λέξεων και για 100 εποχές με παράθυρα 3 και 10 λέξεων.

12.c Cosine Similarity

Βρίσκουμε τις σημασιολογικά κοντινότερες λέξεις στις ζητούμενες για τα 4 μοντέλα που φτιάχαμε

```

def findSimilar(model):
    # voc = model.wv.index2word
    words = ["bible", "book", "bank", "water"]
    for word in words:
        sim = model.wv.most_similar(word, topn = 2)
        print(word, sim)

print("\n ===== Trained with 10 epochs - 5 word
window ===== ")
model10 = Word2Vec.load("w2vModels/corpus_w2v.10d.model")
findSimilar(model10)
print("\n ===== Trained with 100 epochs - 5 word
window ===== ")
model100 = Word2Vec.load("w2vModels/corpus_w2v.100d.model")
findSimilar(model100)
print("\n ===== Trained with 700 epochs - 5 word
window ===== ")
model700 = Word2Vec.load("w2vModels/corpus_w2v.700d.model")
findSimilar(model700)
print("\n ===== Trained with 1000 epochs - 5
word window ===== ")
model1000 = Word2Vec.load("w2vModels/corpus_w2v.1000d.model")
findSimilar(model1000)
print("\n ===== Trained with 100 epochs - 3 word
window ===== ")

```

```

model100_3 = Word2Vec.load("w2vModels/corpus_w2v.100d_window3.model")
findSimilar(model100)
print("\n ===== Trained with 100 epochs - 10 word window =====")
model100_10 = Word2Vec.load("w2vModels/corpus_w2v.100d_window10.model")
findSimilar(model100)

===== Trained with 10 epochs - 5 word window =====
bible [('pen', 0.5356961488723755), ('novel', 0.526168167591095)]
book [('novel', 0.6352971792221069), ('letter', 0.54446941614151)]
bank [('beach', 0.6666355133056641), ('floor', 0.6418591737747192)]
water [('smoke', 0.6200596690177917), ('furnace', 0.590398371219635)]

===== Trained with 100 epochs - 5 word window =====
bible [('book', 0.47234031558036804), ('extracts', 0.4441847801208496)]
book [('letter', 0.5826464295387268), ('novel', 0.5626602172851562)]
bank [('hill', 0.4878132939338684), ('landing', 0.4711122214794159)]
water [('waters', 0.6357617974281311), ('streams', 0.5898309946060181)]

===== Trained with 700 epochs - 5 word window =====
bible [('book', 0.4640274941921234), ('circumcision', 0.39845409989356995)]
book [('novel', 0.5958874225616455), ('letter', 0.5003246665000916)]
bank [('hill', 0.5105876326560974), ('floor', 0.4596291780471802)]
water [('waters', 0.6473720073699951), ('fire', 0.5830942988395691)]

===== Trained with 1000 epochs - 5 word window =====
bible [('book', 0.4431823790073395), ('riff', 0.38461700081825256)]
book [('novel', 0.5690536499023438), ('books', 0.5229795575141907)]
bank [('hill', 0.49244222044944763), ('floor', 0.458516389131546)]
water [('waters', 0.6185929179191589), ('blood', 0.5374496579170227)]

===== Trained with 100 epochs - 3 word window =====
bible [('book', 0.4431823790073395), ('riff', 0.38461700081825256)]
book [('novel', 0.5690536499023438), ('books', 0.5229795575141907)]
bank [('hill', 0.49244222044944763), ('floor', 0.458516389131546)]
water [('waters', 0.6185929179191589), ('blood', 0.5374496579170227)]

===== Trained with 100 epochs - 10 word window =====
bible [('book', 0.4431823790073395), ('riff', 0.38461700081825256)]
book [('novel', 0.5690536499023438), ('books', 0.5229795575141907)]
bank [('hill', 0.49244222044944763), ('floor', 0.458516389131546)]
water [('waters', 0.6185929179191589), ('blood', 0.5374496579170227)]

```

1 & 3. Είναι τα αποτελέσματα τόσο ποιοτικά όσο περιμένατε;

Όχι, ειδικά όταν το μοντέλο έχει εκπαιδευθεί με λίγες εποχές. Για παράδειγμα, η πλησιέστερη λέξη στην τράπεζα (bank) είναι η παραλία (!) (beach), όταν το μοντέλο έχει εκπαιδευθεί με 10 εποχές. Ακόμα, κι όταν έχει εκπαιδευθεί το μοντέλο με 1000 εποχές, η πλησιέστερη λέξη στην τράπεζα είναι η λέξη λόφος (!) (hill). Βέβαια, για άλλες λέξεις, ακόμα και με λίγες εποχές εκπαίδευσης εμφανίζονται καλά αποτελέσματα (water [('waters', 0.6357617974281311), ('streams', 0.5898309946060181)] με 100 μόλις εποχές)

2 & 3. Βελτιώνονται αν αλλάξετε το μέγεθος του παραθύρου context / τον αριθμό εποχών;

Σε γενικές γραμμές, ναι. Π.χ. water [('smoke', 0.6200596690177917), ('furnace', 0.590398371219635)] με 10 εποχές & water [('waters', 0.6185929179191589), ('blood', 0.5374496579170227)] με 1000 εποχές. Βέβαια, για την λέξη τράπεζα η πλησιέστερη εμφανίζομενη λέξη παραμένει παράδοξη (παραλία με 10 εποχές, λόφος με 1000 εποχές) Επίσης, οι πλησιέστερες λέξεις αν και εύλογες μπορεί να αλλάζουν λίγο όσο αυξάνεται ο αριθμός των εποχών.

4. Τι θα κάνατε για να βελτιώσετε τα αποτελέσματα;

Γενικά, μεγαλύτερο παράθυρο λέξεων δίνει υψηλότερου επιπέδου πληροφορία για μια λέξη και μικρότερο παράθυρο δίνει πιο εντοπισμένη πληροφορία και άρα υπάρχει ένα tradeoff.

12.d Algebraic Expression of Semantic Analogies for Concepts

```
import numpy as np
from gensim.models import KeyedVectors

# Load the pre-trained GoogleNews vectors
model = KeyedVectors.load('w2vModels/corpus_w2v.1000d.model')

# Define the word triplets
word_triplets = [("girls", "queen", "kings"), ("taller", "tall", "good"),
("france", "paris", "london")]

# Iterate over the word triplets and perform the algebraic operation
for triplet in word_triplets:
    a, b, c = triplet
    print(triplet)
    v = model.wv[b] - model.wv[a] + model.wv[c]
    sim = model.wv.most_similar(positive = [a, c], negative = [b], topn = 7)
    print(f"{b} is to {a} as {c} is to {sim[0][0]}")
```

Η ακρίβεια αυξάνεται αν διαθέτουμε ένα μεγάλο και καλό Corpus :)

```
('girls', 'queen', 'kings')
queen is to girls as kings is to men
('taller', 'tall', 'good')
tall is to taller as good is to better
('france', 'paris', 'london')
paris is to france as london is to country
```

```

12.e Loading GoogleNewsVectors
from gensim.models import KeyedVectors
path = 'googleNews/GoogleNews-vectors-negative300.bin'
model = KeyedVectors.load_word2vec_format(path, binary = True, limit = 10000)

12.f Cosine Similarity for GoogleNews
def findSimilarGoogle(model):
    # voc = model.wv.index2word
    words = ["bible", "book", "bank", "water"]
    for word in words:
        sim = model.most_similar(word, topn = 2)
        print(word, sim)

# Load GoogleNews vectors
NUM_W2V_TO_LOAD = 1000000
model = KeyedVectors.load_word2vec_format(path, binary=True,
limit=NUM_W2V_TO_LOAD)

# Call findSimilar() function with loaded model
findSimilarGoogle(model)

=====
===== GoogleNews Model =====
=====
bible [('Bible', 0.736778199672699), ('bibles', 0.6052598357200623)]
book [('tome', 0.7485830783843994), ('books', 0.7379177808761597)]
bank [('banks', 0.7440759539604187), ('banking', 0.690161406993866)]
water [('potable_water', 0.6799106001853943), ('Water', 0.6706871390342712)]

```

Παρατηρούμε πως το 1.5 GB Corpus από τα GoogleNews είναι αρκετό για να διορθώσει το πρόβλημα με τη λέξη bank που είχαμε με το δικό μας Corpus!

12.g Algebraic Expression of Semantic Analogies for Concepts

```

# Define the word triplets
word_triplets = [("girls", "queen", "kings"), ("taller", "tall", "good"),
("france", "paris", "london")]

# Iterate over the word triplets and perform the algebraic operation
for triplet in word_triplets:
    a, b, c = triplet
    print(triplet)
    v = model[b] - model[a] + model[c]
    sim = model.most_similar(positive = [a, c], negative = [b], topn = 7)
    print(f"{b} is to {a} as {c} is to {sim[0][0]}")

('girls', 'queen', 'kings')
queen is to girls as kings is to boys
('taller', 'tall', 'good')
tall is to taller as good is to better
('france', 'paris', 'london')
paris is to france as london is to england

```

Και βλέπουμε πως με ένα πραγματικά μεγάλο Corpus αυτές οι πράξεις έχουν τέλειο αποτέλεσμα!

Step 13: Visualization of Word Embeddings

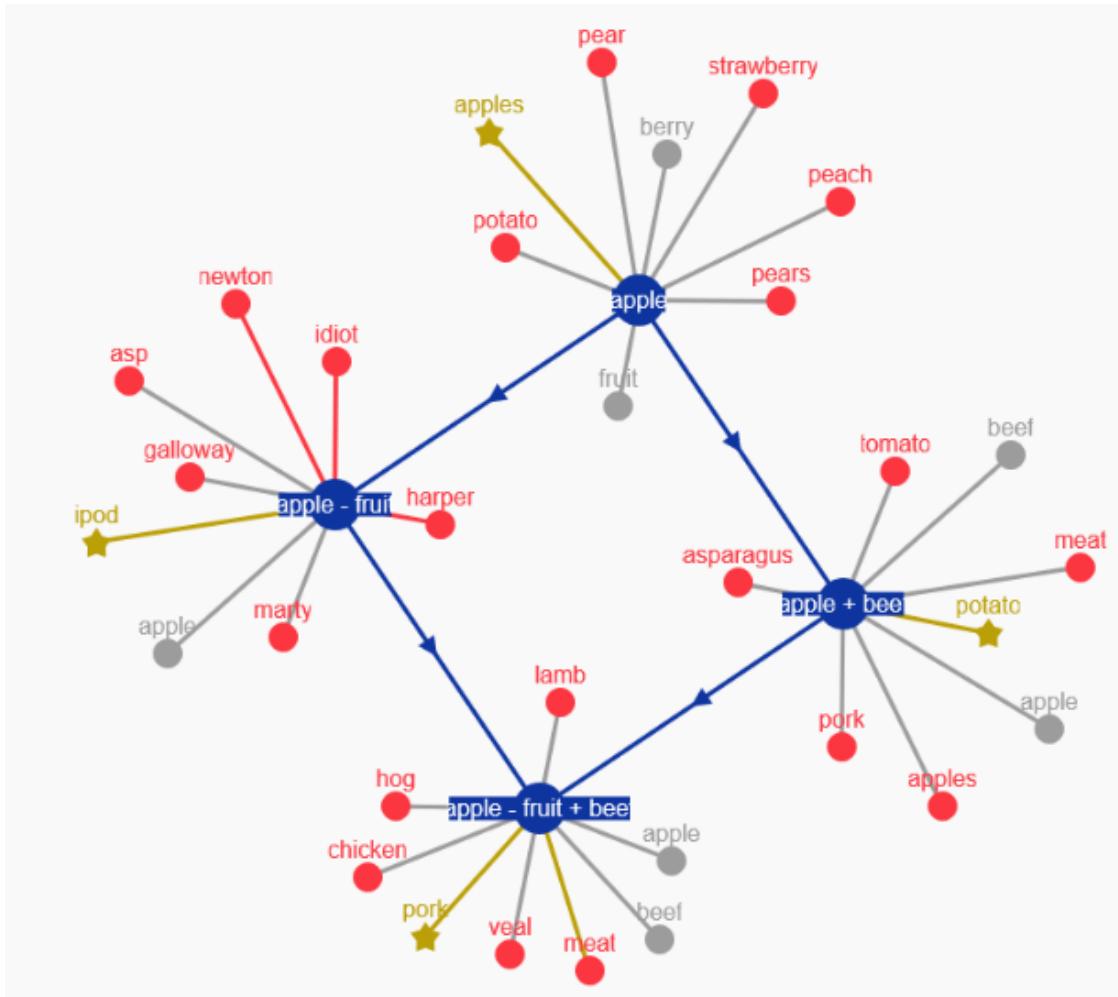
13.a Experimenting with DASH WORD EMBEDDINGS ARITHMETIC

Πειραματιστήκαμε με το εργαλείο και τυπώνουμε μερικά επιτυχημένα παραδείγματα

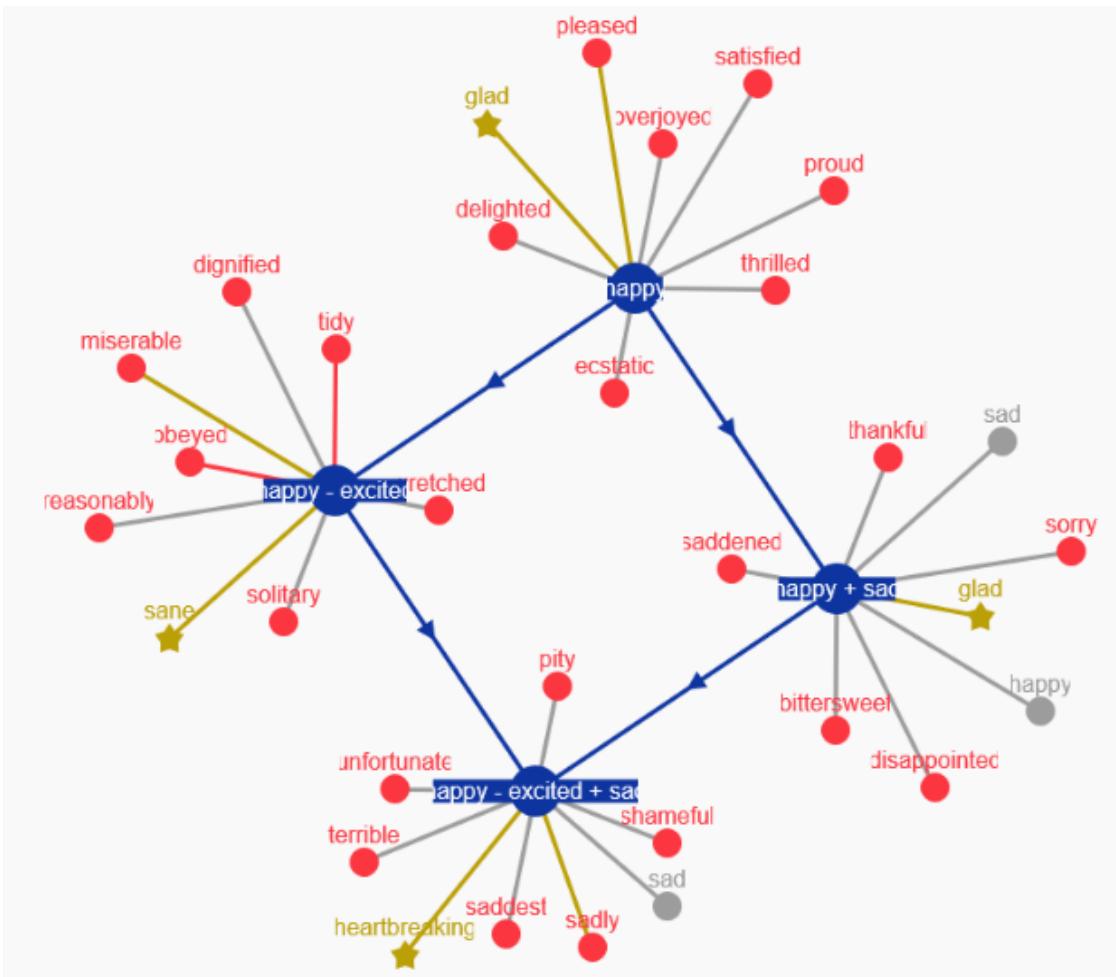
```
from PIL import Image
from IPython.display import display

images = ['imageFST/apple_fruit_beef.png', 'imageFST/happy_excited_sad.png',
'imageFST/queen_woman_man.png', 'imageFST/sad_happy_excited.png']
labels = ["embeddings for apple - fruit + beef", "embeddings for happy -",
excited + sad",
"embeddings for queen - woman + man" , "embeddings for sad - happy
+ excited"]
for i in range(len(images)):
    with Image.open(images[i]) as img:
        print("Projecting " + labels[i])
        display(img)
```

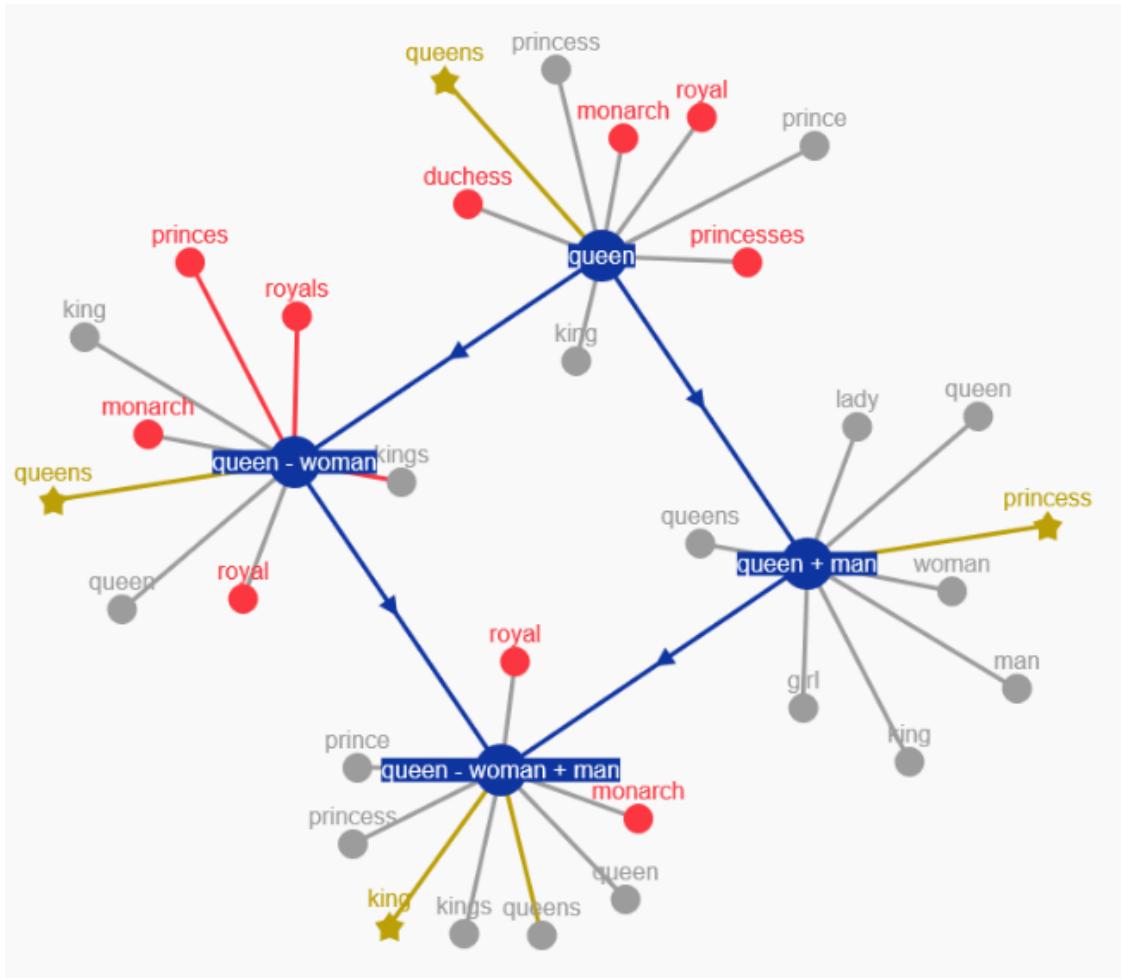
Projecting embeddings for apple - fruit + beef



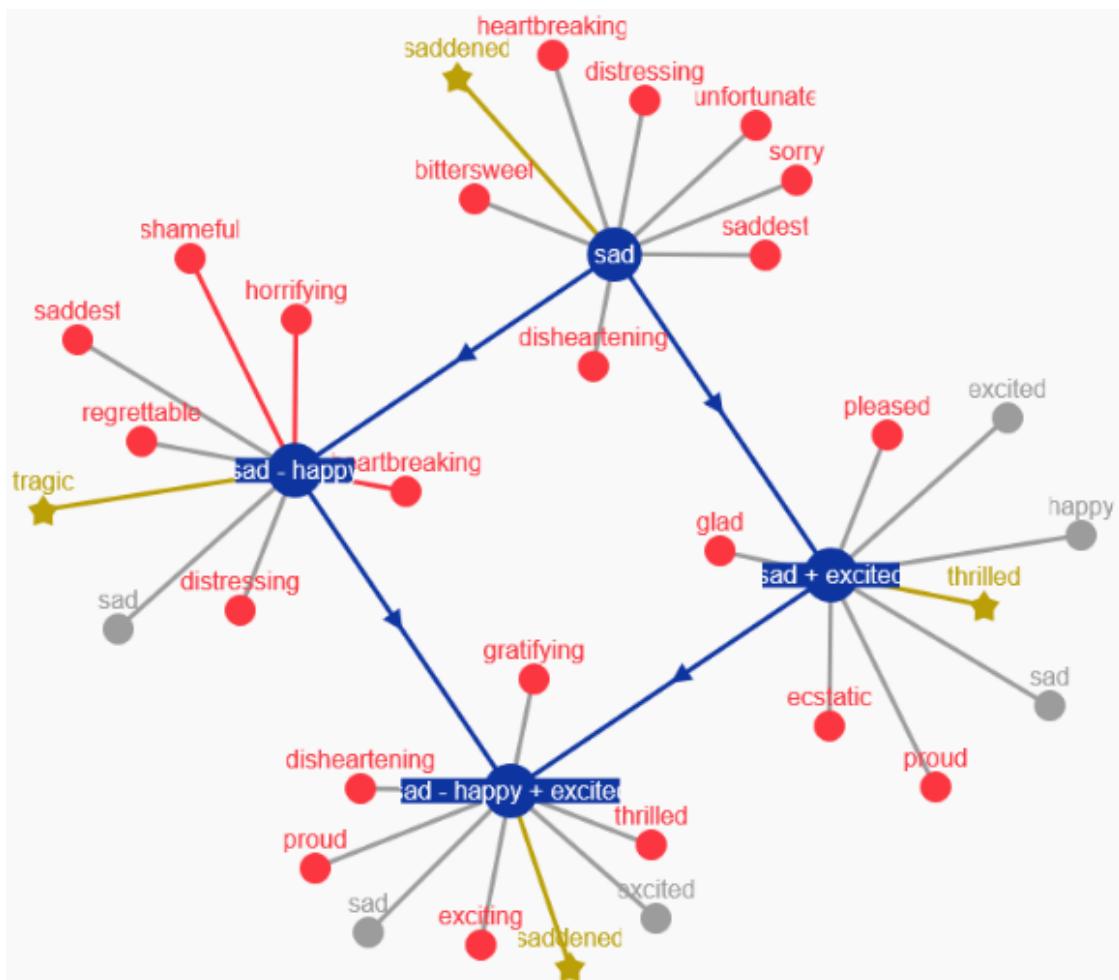
Projecting embeddings for happy - excited + sad



Projecting embeddings for queen - woman + man



Projecting embeddings for sad - happy + excited



Παρατηρήσαμε πολύ καλά αποτελέσματα:

1. apple - fruit + beef = pork (1), meat (2)
2. happy - excited + sad = heartbreaking
3. queen - woman + man = king
4. sad - happy + excited = saddened

Για την πρώτη περίπτωση η δεύτερη κοντινότερη λέξη (που δεν απείχε πολύ από την πρώτη), είναι αυτή που περιμέναμε σημασιολογικά.

13.b Saving Word Embeddings in the TSV file

```
import os
from gensim.models import Word2Vec

directory = "data"

if not os.path.exists(directory):
    os.makedirs(directory)
```

```

model = Word2Vec.load("w2vModels/corpus_w2v.100d.model")

with open("data/embeddings.tsv", "w", newline='') as f:
    # Get the list of words in the vocabulary
    words = model.wv.index_to_key
    # print(words)
    for word in words:
        vector = list(model.wv.get_vector(word))
        # print(vector)
        for vector_element in vector:
            # print(vector_element)
            print(vector_element, file=f, end="\t")
    print(file=f)

with open("data/metadata.tsv", "w") as f:
    words = list(model.wv.key_to_index.keys())
    for word in words:
        print(word, file=f)

print("Embeddings and metadata files saved to 'data' directory.")

```

13.c Embeddings Projection

PCA: Principal Component Analysis

Η μέθοδος χρησιμοποιείται για τη μείωση διαστατικότητας δεδομένων. Μετατρέπει τα αρχικά, υψηλής διάστασης δεδομένα σε ένα χώρο μικρότερης διάστασης διατηρώντας όση περισσότερη πληροφορία γίνεται.

Source: https://en.wikipedia.org/wiki/Principal_component_analysis

T-SNE: t-distributed stochastic neighbor embedding

Η στατιστική αυτή μέθοδος βοηθά στην οπτικοποίηση δεδομένων υψηλής διάστασης αναθέτοντας σε κάθε datapoint ένα vector στον 2-D ή 3-D χώρο:

1. Βάσει της κατανομής πιθανότητας ζευγών από Embeddings (του αρχικού χώρου):

$$p_{j|i} = \frac{\exp(-\| \mathbf{x}_i - \mathbf{x}_j \|^2 / 2\sigma_i^2)}{\sum_{k \neq i} \exp(-\| \mathbf{x}_i - \mathbf{x}_k \|^2 / 2\sigma_i^2)}$$

2. Επιδιώκοντας την ελαχιστοποίηση της Kullback–Leibler απόκλισης μεταξύ των κατανομών πιθανότητας:

$$\text{KL}(P \parallel Q) = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}, \text{ óπου } q_{ij} = \frac{(1 + \| \mathbf{y}_i - \mathbf{y}_j \|^2)^{-1}}{\sum_k \sum_{l \neq k} (1 + \| \mathbf{y}_k - \mathbf{y}_l \|^2)^{-1}}$$

Τα αποτελέσματα της μεθόδου αυτής οδηγούν κάποιες φορές σε virtual clusters οπότε χρειάζεται να θυμόμαστε ότι το αποτέλεσμα της **δεν** είναι clustering!

Τα βήματα του αλγορίθμου (πολύ high - level) είναι τα εξής:

1. Δημιουργείται η κατανομή πιθανότητας για όλα τα ζεύγη embeddings στον αρχικό χώρο ώστε σημασιολογικά κοντινές λέξεις να έχουν μεγάλη πιθανότητα και σημασιολογικά μακρινές λέξεις να έχουν μικρή πιθανότητα.
2. Ορίζεται μια παρόμοια κατανομή πιθανότητας για τα σημεία του μικρότερου (νέου) χώρου και σκοπός είναι να ελαχιστοποηθεί η απόκλιση μεταξύ κατανομών σχετικά με τα vectors. Έτσι, ο αλγόριθμος μπορεί να αντιστοιχίζει υψηλής διάστασης δεδομένα σε έναν (πολύ) μικρότερο χώρο χωρίς να χάνεται η ομοιότητα των αρχικών διανυσμάτων, όσο είναι εφικτό.

Source: https://en.wikipedia.org/wiki/T-distributed_stochastic_neighbor_embedding

Παρατηρήσεις από τα αποτελέσματα της PCA: Βρήκαμε τα εξής πολύ κοντινά, μετασχηματισμένα vectors:

balaam, pharisee, judas, elkanah, enos

Όλα είναι λέξεις από την Αγία Γραφή, άρα είναι λογικό να γειτονεύουν στο χώρο μικρότερης διάστασης.

Παρατηρήσεις από τα αποτελέσματα του T - SNE: Βρήκαμε τα εξής μετασχηματισμένα vectors πολύ κοντά μεταξύ των:

anathoth, samlah, shechem, baal, succoth, haran, hazael, benjamin, cyrus

Όλα είναι επίσης λέξεις από την Αγία Γραφή, προερχόμενες από το Corpus Genesis, προφανώς.

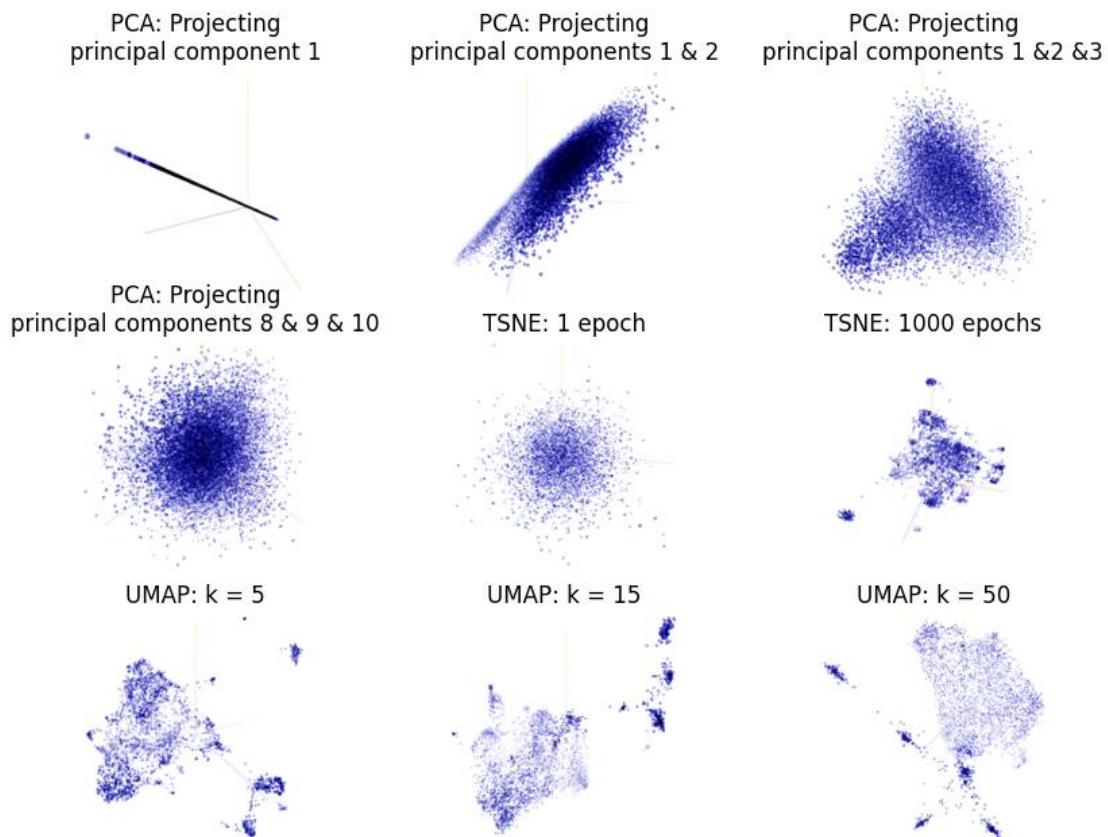
Παρόμοιες παρατηρήσεις είχαμε και για το UMAP (k Nearest Neighbours)

```
import os
from PIL import Image
import matplotlib.pyplot as plt

directory = "data"
files = [f for f in os.listdir(directory) if f.endswith(".png")]
titles = ['PCA: Projecting\nprincipal component 1', 'PCA: Projecting\nprincipal components 1 & 2', 'PCA: Projecting\nprincipal components 1 &2 &3',
          'PCA: Projecting\nprincipal components 8 & 9 & 10', 'TSNE: 1 epoch', 'TSNE: 1000 epochs', 'UMAP: k = 5', 'UMAP: k = 15', 'UMAP: k = 50']
fig = plt.figure(figsize=(10, 10))

for i, file in enumerate(files):
    img = Image.open(os.path.join(directory, file))
    img = img.resize((200, 200)) # resize to 200x200
    ax = fig.add_subplot(int(len(files)/3+1), 3, i+1)
```

```
ax.set_title(titles[i]) # set title to filename  
plt.imshow(img)  
plt.axis('off')  
  
plt.show()
```



Step 14

14.a Downloading Stanford Data

Αποθηκεύουμε τα δεδομένα στον φάκελο stanfordData

14.b Preprocessing Data

```
import glob
import os
import re
from gensim.models import KeyedVectors
from gensim.models import Word2Vec

import numpy as np
import sklearn
from sklearn.linear_model import LogisticRegression

# SCRIPT_DIRECTORY = os.path.realpath(__file__)

data_dir      = "stanfordData\\aclImdb"
train_dir     = data_dir + "\\train"
test_dir      = data_dir + "\\test"
pos_train_dir = train_dir + "\\pos"
neg_train_dir = train_dir + "\\neg"
pos_test_dir  = test_dir + "\\pos"
neg_test_dir  = test_dir + "\\neg"

# For memory Limitations. These parameters fit in 8GB of RAM.
# If you have 16G of RAM you can experiment with the full dataset / W2V
MAX_NUM_SAMPLES = 5000
# Load first 1M word embeddings. This works because GoogleNews are roughly
# sorted from most frequent to least frequent.
# It may yield much worse results for other embeddings corpora
NUM_W2V_TO_LOAD = 1000000

SEED = 42

# Fix numpy random seed for reproducibility
np.random.seed(SEED)

def strip_punctuation(s):
    return re.sub(r"[^a-zA-Z\s]", " ", s)

def preprocess(s):
    return re.sub("\s+", " ", strip_punctuation(s).lower())
```

```

def tokenize(s):
    return s.split(" ")

def preproc_tok(s):
    return tokenize(preprocess(s))

def read_samples(folder, preprocess=lambda x: x):

    samples = glob.iglob(os.path.join(folder, "*.txt"))
    data = []

    for i, sample in enumerate(samples):
        if MAX_NUM_SAMPLES > 0 and i == MAX_NUM_SAMPLES:
            break
        with open(sample, "r", encoding="utf8") as fd:
            x = [preprocess(l) for l in fd][0]
            data.append(x)

    return data

def create_corpus(pos, neg):
    corpus = np.array(pos + neg)
    y = np.array([1 for _ in pos] + [0 for _ in neg])
    indices = np.arange(y.shape[0])
    np.random.shuffle(indices)

    return list(corpus[indices]), list(y[indices])

def extract_nbownbow(corpus, size, model, myModel = True):
    nbownbow = []
    for corp in corpus:
        counts = 0
        init_vec = np.zeros(size)
        if myModel:
            index2word_set = set(model.wv.index_to_key)
        else:
            index2word_set = set(model.index_to_key)
        for word in corp:
            counts = counts + 1
            if word in index2word_set:
                if myModel:
                    init_vec = np.add(init_vec, model.wv.get_vector(word))
                else:
                    init_vec = np.add(init_vec, model.get_vector(word))
            nbownbow.append(init_vec)
    return nbownbow

```

```

        init_vec = np.divide(init_vec, counts)
        nbow.append(init_vec)

    return nbow

def train_sentiment_analysis(train_corpus, train_labels):
    print("training of LR started")
    regression_model = LogisticRegression()
    regression_model.fit(train_corpus, train_labels)
    return regression_model

def evaluate_sentiment_analysis(classifier, test_corpus, test_labels):
    accuracy = sklearn.metrics.accuracy_score(test_labels,
    classifier.predict(test_corpus))
    print("The accuracy of the model is ", accuracy)

    return accuracy

if __name__ == "__main__":
    #Load models
    print("loaded models")
    model_1 = Word2Vec.load('w2vModels/corpus_w2v.100d.model')
    path = 'googleNews/GoogleNews-vectors-negative300.bin'
    model_2 = KeyedVectors.load_word2vec_format(path, binary=True,
limit=1000000)
    # googleModel.save("w2vModels/google_w2v.model")
    # model_2 = KeyedVectors.load("w2vModels/google_w2v.model")

    # model_2 = KeyedVectors.load('models/word2vec_google.model')
    # read data and create corpuses
    print("read stuff")
    print(pos_train_dir)
    pos_train = read_samples(pos_train_dir, preproc_tok)
    neg_train = read_samples(neg_train_dir, preproc_tok)
    pos_test = read_samples(pos_test_dir, preproc_tok)
    neg_test = read_samples(neg_test_dir, preproc_tok)
    train_corpus, train_labels = create_corpus(pos_train, neg_train)
    test_corpus, test_labels = create_corpus(pos_test, neg_test)

    #create NBOWs and train with simple model
    print("creating nbows for our model")
    nbow_train_corpus = extract_nbew(train_corpus, 100, model_1, True)
    nbow_test_corpus = extract_nbew(test_corpus, 100, model_1, True)
    log_reg = train_sentiment_analysis(nbow_train_corpus, train_labels)
    print("Stats for our model:")
    acc_log_reg = evaluate_sentiment_analysis(log_reg, nbow_test_corpus,
test_labels)

```

```
#create NBOWs and train with google model
print("create nbows for Google News model")
nbow_train_corpus_g = extract_nbownbow(train_corpus, 300, model_2, False)
nbow_test_corpus_g = extract_nbownbow(test_corpus, 300, model_2, False)
log_reg_g = train_sentiment_analysis(nbow_train_corpus_g, train_labels)
print("Stats for Google News model:")
acc_log_reg = evaluate_sentiment_analysis(log_reg_g, nbow_test_corpus_g,
test_labels)
```

Stats for our model:

The accuracy of the model is 0.8002

Stats for Google News model:

The accuracy of the model is 0.8347

14.c Logistic Regression using Embeddings of Neural Bag of Words from Corpus

Εκπαιδεύσαμε το Logistic Regression μοντέλο για αναγνώριση θετικών και αρνητικών συναισθημάτων, χρησιμοποιώντας τα Embedddings που δημιουργήσαμε στο βήμα 12. Πετύχαμε accuracy 80.02%. Το σχετικά υψηλό score αποδίδουμε στη χρήση μεγάλου Corpus. Συγκεκριμένα, συνδυάσαμε 4 corpora που περιέχουν φυσικό λόγο και πολλά συναισθήματα:

1. Gutenberg
2. Genesis
3. Shakespear
4. Movie-Reviews

14.d Logistic Regression using Embeddings of Neural Bag of Words from Google News

Εκπαιδεύσαμε ξανά το Logistic Regression μοντέλο, χρησιμοποιώντας τα Google News Embedddings. Πετύχαμε accuracy 83.47%. Το υψηλό score τώρα συνέβη χάρη στη χρήση του **τεράστιου** Corpus από τα Google News.