CMPE 590
SP.TP. MACHINE TRANSLATION
2017 SPRING

ONUR MUSAOĞLU
2016700114

IBM MODELS 1-2-3
(PROGRAMMING PROJECT)

SUBMITTED TO TUNGA GÜNGÖR

DATE: 09.05.2017

1. INTRODUCTION

Machine Translation (MT) is a subfield of computational linguistics which converts a text to desired language. In MT system, for a given source language sentence, the system tries to convert the sentence as a target language sentence. At the very beginning of Machine Translation history, rule-based machine translation systems were popular. In rule based systems the problem is solved using rules. In 1988 a statistical machine translation system is introduced by Brown et. al. After these Brown introduced IBM models in detail in 1993. In statistical machine translation, we are trying to find most probable translation of a given sentence. There are some kinds of statistical machine translation like word-based or phrase-based.

In IBM models, a sentence aligned bilingual corpora was used. However, there is no alignment for words. In IBM Model 1, the only consideration is lexical translations of words which means they are trying to find word translation probabilities. By using an expectation-maximization algorithm for solving two-sided problem which if we know alignment of words we can investigate word translation probabilities and if we know word translation probabilities we can investigate alignment probabilities.

Because IBM Model 1 does not consider the alignment of the words and takes all alignment probabilities equally, a new model introduced to IBM which is IBM Model 2. In IBM Model 2, in addition to the lexical translation model, alignment probabilities were introduced for probability calculation. The result of IBM Model 2 outputs alignment probabilities and word translation probabilities.

After IBM Model 2, IBM introduced Model 3. IBM Model 2 does not consider fertility of words, but IBM Model 3 consider this property. Fertility of a word is how many words it is translated into. For instance, Turkish word ‘yapmam’ is translated as ‘I do not do’ in English, so it is fertility is 4 for this specific problem. So, in IBM Model 3 we have lexical translation model, distortion model (like alignment) and a fertility model. Since calculating such an exponential calculation for all alignment possibilities for two sentence is cost ineffective, they use a sampling method to decrease the number of alignments for calculating.

In this Programming Project Report, the IBM Model 1-2-3 implementations are described carefully. You can find regarding codes in appendices.

2. PROGRAM INTERFACE

There is no special user interface designed for the program. Program should be run in any environment which can run Python 3.5.1. User should call the “*python3 Main.py*” command in linux terminal or any other environment for program execution. In any of these environments, to stop the words, the regarding program execution stop command can be used like CTRL+C in linux. Additionally, by choosing terminate option by pressing 9, user can stop the program.

3. PROGRAM EXECUTION

To execute the code use "*python3 Main.py*" command in a python 3.5.1 enabled environment. After execution, user will be asked to choose what she wants to do. The screenshot for the question is as follows

```
Please choose what you want to do:
    1: training
    2: testing
    3: give a sentence to translate
    9:for exit
3
```

Here there are four options, if user choses 9 then program terminates. If user choses training, program starts to retrain itself, however, this operation can take hours, so avoid this option as much as possible.

3.1. Option 2: testing

If user choses 2 then testing phase starts and following options comes;

```
Choose Model to test:
    1: IBM Model 1
    2: IBM Model 2
    3: IBM Model 3
2
```

Here user choses which IBM model to test, in this example she chooses IBM model 2.

Then program starts to ask questions for each test sentence in test dataset. Here is an example for one test sentence. It firstly asks how many possible results user want to supply.

```
How many possible results you want to supply for sentece 'The verses continue:':
2
Type possible sentence number 1 : şeytan devam eder
Type possible sentence number 2 : şeytanlık devam eder
probability for sentence 'şeytan devam eder' is 0.0
word 'şeytanlık' is not found in target language dictionary
probability for sentence 'şeytanlık devam eder' is 8.84291296728e-30
tranlation result is 'şeytanlık devam eder'  with probability : 8.84291296728e-30
```

User want to enter two possibilities. Program one by one ask each possibility. User firstly enter "şeytan devam eder", then enter "şeytanlık devam eder". The program starts to calculate the probabilities of each sentences and outputs both sentences' possibilities and chooses the most probable one as translation.

3.2. Option 3: give a sentence to translate

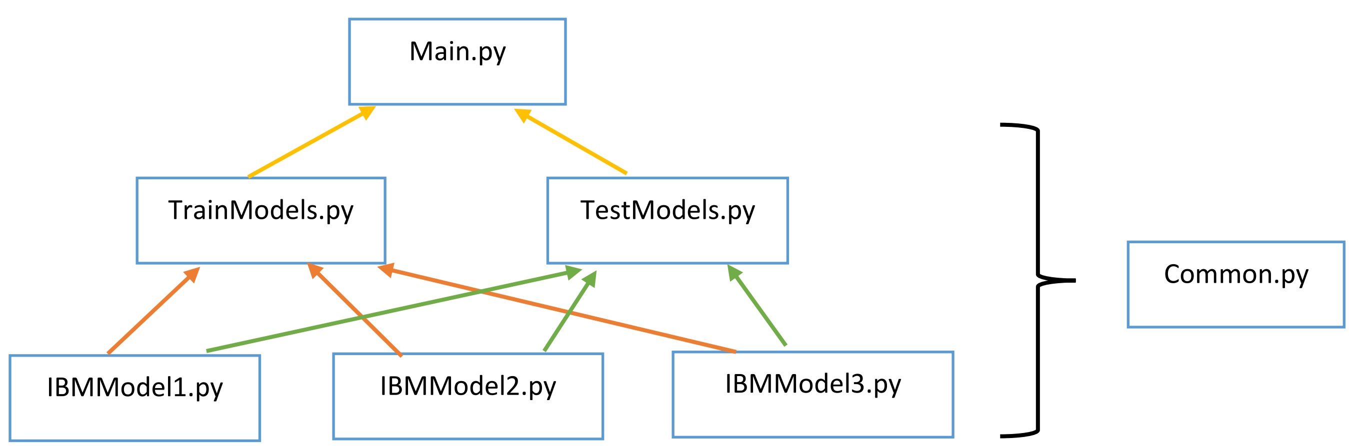
This option makes same operations with option 2 however, here user wanted to enter a sentence for translation rather than giving sentences from test dataset.

4. INPUT and OUTPUT

As stated, since the program runs in console the inputs and outputs are sentences or option numbers. Please after entering your input press enter. For output texts, system has multi-color output. The green texts are inputs supplied by users. The purple text shows the possibilities of sentences supplied by user. The Orange texts show not found target language words. The blue texts show not found source language words. Finally, the red texts show the translation results.

5. PROGRAM STRUCTURE

For the program, Python programming language is used. The relation between the python files is as in the following figure.



Here, *Main* program is the start of the program. By demanding input from user, it calls *TrainModels* or *TestModels* files. These both files call *IBMModel1*, *IBMModel2* and *IBMModel3* according to the operation done. These files use *Common* file which has common functions and variables to all programs.

Before going into details of these files, let firstly investigate the data structures used.

5.1. Data Structures

Because we are using the system for machine translation it should be fast enough.

5.1.1. Dictionaries: At the first place the problem is keeping all the words of both languages in the memory and searching for these words can be problematic because string comparison cost much. For that reason, the words are represented as numbers (word ids) and for searching the words corresponding to word ids, a dictionary is used. In Python, a dictionary is a Hash Map. In the dictionary, the key values are the string words, and values are the word ids. For both target and source languages, there are two *word-wordId* dictionaries.

5.1.2. Lists: Another important issue is keeping all the sentences of both target and source languages. For that reason, a list type of Python is used for both languages. List object can be with no size in the initialization so that we use this data structure to append sentences as we desired. It is like arrays.

5.1.3. NumPy Arrays: NumPy is a Python library. It has array objects and many functions supplying matrix operations. It is just one line of code to create a NumPy array and fill it with some value. These NumPy arrays are used for keeping all matrix required problems. For example, the translation probabilities are for two words, so keeping them in a 2-D array can solve the problem. So, we create a NumPy array with the size of dictionary size of the languages.

5.2. Program Files

After data structures introduced let's talk about Python files introduced above. In each file, algorithms used will be explained. You can fide the codes included in these file in appendices.

5.2.1. Main.py: This is the core of the program. It asks users what operation they want to do. According to user's answer it calls test sub programs or train subprograms.

5.2.2. TrainModels.py: In this python file, we have two main functions. The first function is reading the files and tokenizing sentences. Also in this function, a word dictionary is created for given dataset. Second function is for main training algorithm management. Firstly, we limit the number of words in a sentence to 10, then call each IBM Model one by one. Additionally, for the usage of tests it saves the IBM model results as NumPy files.

5.2.3. TestModels.py: There are basically two functions in this python file. According to user input, system decides which IBM model to test and it gets the regarding model data from saved models. That is, if user selects IBM Model 2 for testing, then system load IBM Model 2's translation probability matrix and alignment probability matrix. These matrixes are under *models* folder with the same directory level with TestModel.py file. These matrixes are created in training phase of the program, so with new trainings they can change.

5.2.4. IBMModel1.py: This file contains expectation maximization algorithm for IBM Model 1 and an evaluation function which returns the probability of the sentence according to transition probabilities. The former, namely expectation maximization algorithm, is implemented using pseudocode supplied to us. This expectation maximization algorithm returns the translation probabilities for words.

5.2.5. IBMModel2.py: IBM Model 2 is implemented in this file. There are IBM Model 2 expectation maximization algorithm and a test evaluation function. The former returns translation probabilities and alignment probabilities. The later uses the saved translation probabilities and alignment probabilities, and by looking at whether supplied word is included or not, it calculates the probability. There are two components when calculating this probability namely translation model and alignment model.

5.2.6. IBMModel3.py: This file contains all functions related to IBM Model 3. As we know, IBM Model 3 makes sampling for alignments. These sampling process is a hill climbing and there can be some local maxima problems. To overcome this problem, we use pegging and by fixing two words, we look at the neighbors of the most probable alignment coming from IBM Model 2. Thus, in this file there is a sampling function, a probability calculation function a neighboring function and a hill climbing function. In addition to them, there is a function for calculating the probability of a test sentence.

5.2.7 Common.py: This file contains the common functions and variables to all files. For instance, the convergence control function is in this file. Also, some variables like number of iterations, number of training and test samples are included in this file.

6. EXAMPLES

We can exemplify the translation process as follows. Let say we want to convert the English sentence ‘the god can protect people from ignorant’ to Turkish. We should give some candidate translations to system for evaluation and finding the best one. I give ‘tanrı insanları cahilden koruyabilir’ and ‘allah cahilleri engeller’ as two candidate sentences. The system outputs that ‘tanrı insanları cahilden koruyabilir’ is more probable than the other candidate.

7. IMPROVEMENTS AND EXTENSIONS

The dataset used for training has a lot of aligned sentences, however, since my computer power is not enough to use all of it, I use a subset of it which has 4000 sentences. Additionally, for computation time problems, I use sentences with length at most 10. So, these choices decrease the size of the dictionaries and this causes to not found some words in the dictionary. If the size of the corpus increases the success of the translator will increase.

8. DIFFICULTIES ENCOUNTERED

There are some difficulties encountered when developing the system. Especially for Model 3 since the pseudo code in the book and in the course notes is not same it causes a confusion and a problem. The code in the book has a lot of errors. Also, some points in the pseudocodes is not clear like calculating *null* value. In addition to these difficulties there are also some computational problems. For instance, the sentences longer than 10 words cause IBM Model 3 to work nearly twenty minutes on them, even sometimes it keeps forty minutes. So, I limit the sentence length with 10 words. Moreover, since iterations keep much time I limit the iteration number as 15. However, my IBM Model 3 is not working healthy, there are always some problems. At least it puts non-number values to probability matrices.

9. CONCLUSION

This project has been teaching many things for statistical machine translation. I learned the general logic of statistical machine translation and have a better idea about IBM Models. The logic derived and its causes also teach me how to think about machine translation problems. It excites me to produce my own machine translation system for my mother language Zazaki and Turkish.

10. APPENDENCIES

The source code of the python files is as follow.

Main.py

```
import TrainModels
import TestModels

while True:
    try:
        mode = int(input('\n\nPlease choose what you want to do: \n\t1: training \n\t2: testing
\n\t3: give a sentence to translate\n\t9:for exit\n'))
    except ValueError:
        print ("Not a number")

    if mode == 1:
        TrainModels.train_models()
    elif mode == 2:
        try:
            test_model = int(input('Choose Model to test: \n\t1: IBM Model 1 \n\t2: IBM Model 2
\n\t3: IBM Model 3\n'))
        except ValueError:
            print ("Not a number")

        if test_model > 3 or test_model < 1:
            print("invalid number")
            exit()

        TestModels.test(test_model,False,'')
    elif mode == 3:
        sentence_to_translate = input("Plese provide sentence to translate: ")

        try:
```

```
            test_model = int(input('Choose Model to test: \n\t1: IBM Model 1 \n\t2: IBM Model 2
\n\t3: IBM Model 3\n'))
        except ValueError:
            print ("Not a number")

        TestModels.test(test_model,True,sentence_to_translate)
    elif mode == 9:
        break
    else:
        print("invalid mode")

print("goodbye!")
```

TestModels.py

```
import numpy as np
from nltk.tokenize import word_tokenize
import string
import IBMModel1
import IBMModel2
import IBMModel3
import Common


def get_tokens_of_sentence(sentence):
    translate_table = dict((ord(char), None) for char in string.punctuation)
    sentence = sentence.translate(translate_table)
    tokens = word_tokenize(sentence.lower())
    return tokens


def sentence_tester(sentence):
    try:
        num_of_sentences = int(input("\nHow many possible results you want to supply for sentece
'"+ sentence.strip() +"': \n"))
    except ValueError:
        print ("Not a number")

    possible_sentences = list()
    for i in range(num_of_sentences):
        input_sentence = input("Type possible sentence number " + str((i+1)) + " : ")
        possible_sentences.append(input_sentence)

    f_sentence = get_tokens_of_sentence(sentence)

    max_score = -1
    max_sentence = ""
    for poss_sentence in possible_sentences:
        e_sentence = get_tokens_of_sentence(poss_sentence)

        if model_number == 1: #IBM Model 1
            prob =
IBMModel1.get_translation_prob(e_sentence,f_sentence,t_e_f,e_word_dict,f_word_dict)
            print(Common.P + "probability for sentence '" + poss_sentence + "' is " + str(prob) +
Common.BL)
        elif model_number == 2: #IBM Model 2
            prob =
IBMModel2.get_translation_prob(e_sentence,f_sentence,t_e_f,a_i_j,e_word_dict,f_word_dict)
            print(Common.P + "probability for sentence '" + poss_sentence + "' is " + str(prob) +
Common.BL)
        elif model_number == 3: #IBM Model 3
            prob =
IBMModel3.get_translation_prob(e_sentence,f_sentence,t_e_f,a_i_j,e_word_dict,f_word_dict)
            print(Common.P + "probability for sentence '" + poss_sentence + "' is " + str(prob) +
Common.BL)

        if prob > max_score:
            max_score = prob
            max_sentence = poss_sentence
```

```
    print(Common.R + "tranlation result is '" + max_sentence +"'  with probability : " +
str(max_score) + Common.BL)

def test(arg_model_number, is_sentence_translate,sentence_to_translate):
    global t_e_f, a_i_j, n_fi_f, e_word_dict,f_word_dict,content_f,model_number

    model_number = arg_model_number

    if model_number == 1: #IBM Model 1
        t_e_f = np.load('models/t_e_f_mat_model1.npy')
    elif model_number == 2: #IBM Model 2
        t_e_f = np.load('models/t_e_f_mat_model2.npy')
        a_i_j = np.load('models/a_i_le_lf_mat_model2.npy')
    elif model_number == 3: #IBM Model 3
        t_e_f = np.load('models/t_e_f_mat_model3.npy')
        a_i_j = np.load('models/d_i_j_le_lf_mat_model3.npy')
        n_fi_f = np.load('models/n_fi_f_mat_model3.npy')

    e_word_dict = np.load("models/e_word_dict.npy").item()
    f_word_dict = np.load("models/f_word_dict.npy").item()

    if is_sentence_translate:
        sentence_tester(sentence_to_translate)
    else:
        with open("Dictionary_files\BU_en.txt", encoding="utf8") as f:
                content_f = f.readlines()

        for sentence in content_f [Common.num_of_train_sample:(Common.num_of_train_sample +
Common.num_of_test_sample)]:
            sentence_tester(sentence)
```

TrainModels.py

```
from nltk.tokenize import word_tokenize
import IBMModel1
import IBMModel2
import IBMModel3
import string
import numpy as np
import Common


def create_tokenized_sentences(content_list, max_index):
    return_sentence_list = list()
    word_dictionary = {} #this dictionary will keep both word and its order in its language
    lang_order = 0
    cnt = 0
    max_len_sentence = 0
    translate_table = dict((ord(char), None) for char in string.punctuation)
    for row in content_list[:max_index]:
        if cnt == 0 :
            row = row.replace(u'\ufeff', '')
            cnt += 1

        row = row.translate(translate_table)
        tokens = word_tokenize(row.lower())

        if len(tokens) > max_len_sentence :
            max_len_sentence = len(tokens)

        produced_sentence = ""
        for token in tokens:
            if token not in word_dictionary:
                word_dictionary[token] = lang_order
                lang_order += 1
            produced_sentence = produced_sentence + token + " "
```

```
        produced_sentence = produced_sentence[:(len(produced_sentence) - 1)]  # remove last empty
        return_sentence_list.append(produced_sentence)

    return_sentence_list[0] = return_sentence_list[0].replace(u'\ufeff', '')  # ufeff character
from document start
    return return_sentence_list, word_dictionary,max_len_sentence


def train_models():
    with open("Dictionary_files\BU_en.txt", encoding="utf8") as f:
        content_en = f.readlines()

    with open("Dictionary_files\BU_tr.txt", encoding="utf8") as f:
        content_tr = f.readlines()

    #just use sentences with length at most 10 words.
    new_content_en = list()
    new_content_tr = list()

    for sen_idx in range(len(content_en)):
        cur_en_sen = content_en[sen_idx].split()
        cur_tr_sen = content_tr[sen_idx].split()
        if len(cur_en_sen) < 11 and len(cur_tr_sen) < 11:
            new_content_en.append(content_en[sen_idx])
            new_content_tr.append(content_tr[sen_idx])

    content_en = new_content_en.copy()
    content_tr = new_content_tr.copy()

    max_num_of_translations = Common.num_of_train_sample

    # parse turksih sentences, tokenize the words
    turkish_sentences, turkish_word_dict, max_le = create_tokenized_sentences(content_tr,
max_num_of_translations)

    # parse english sentences, tokenize the words
    english_sentences, english_word_dict, max_lf = create_tokenized_sentences(content_en,
max_num_of_translations)

    np.save("models/e_word_dict",turkish_word_dict)
    np.save("models/f_word_dict",english_word_dict)

    t_e_f  =
IBMModel1.expectation_maximization(turkish_word_dict,english_word_dict,turkish_sentences,english_
sentences)
    np.save("models/t_e_f_mat_model1",t_e_f)

    t_e_f, a_i_le_lf_mat =
IBMModel2.train(t_e_f,turkish_word_dict,english_word_dict,turkish_sentences,english_sentences,max
_le,max_lf)
    np.save("models/t_e_f_mat_model2",t_e_f)
    np.save("models/a_i_le_lf_mat_model2",a_i_le_lf_mat)

    t_e_f_mat, d_i_j_le_lf_mat, n_fi_f,p0,p1 = IBMModel3.train(t_e_f,
a_i_le_lf_mat,turkish_word_dict,english_word_dict,turkish_sentences,english_sentences,max_le,max_
lf)
    np.save("models/t_e_f_mat_model3",t_e_f_mat)
    np.save("models/d_i_j_le_lf_mat_model3",d_i_j_le_lf_mat)
    np.save("models/n_fi_f_mat_model3",n_fi_f)
    np.save("models/p0",p0)
    np.save("models/p1",p1)
```

IBMModel1.py

```
import numpy as np
from datetime import datetime
import math
import Common
```

```python
def
expectation_maximization(turkish_word_dict,english_word_dict,turkish_sentences,english_sentences)
:
    num_of_tur_word = len(turkish_word_dict)
    num_of_eng_word = len(english_word_dict)
    # em algorithm
    t_e_f_mat = np.full((len(turkish_word_dict), len(english_word_dict)), 1 /
len(english_word_dict),dtype=float)
    t_e_f_mat_prev = np.full((len(turkish_word_dict), len(english_word_dict)), 1,dtype=float)

    cnt_iter = 0
    while not Common.is_converged(t_e_f_mat,t_e_f_mat_prev,cnt_iter) :
        print(cnt_iter)
        cnt_iter += 1
        t_e_f_mat_prev = t_e_f_mat.copy()
        count_e_f = np.full((len(turkish_word_dict), len(english_word_dict)), 0, dtype=float)
        total_f = np.full((len(english_word_dict)),0, dtype=float)
        print("sentece pair giris")
        for idx_tur, tur_sen in enumerate(turkish_sentences): #for all sentence pairs (e,f) do
            #compute normalization
            tur_sen_words = tur_sen.split(" ")
            s_total = np.full((len(tur_sen_words)),0,dtype=float)
            for idx_word in range(len(tur_sen_words)): #for all words e in e do
                tur_word = tur_sen_words[idx_word]
                s_total[idx_word] = 0
                eng_sen_words = english_sentences[idx_tur].split(" ")
                for eng_word in eng_sen_words: #for all words f in f do
                    idx_tur_in_dict =turkish_word_dict[tur_word]
                    idx_eng_in_dict = english_word_dict[eng_word]
                    s_total[idx_word] += t_e_f_mat[idx_tur_in_dict][idx_eng_in_dict]
                #end for
            #end for

            #collect counts
            tur_sen_words = tur_sen.split(" ")
            for idx_word in range(len(tur_sen_words)): #for all words e in e do
                tur_word = tur_sen_words[idx_word]
                eng_sen_words = english_sentences[idx_tur].split(" ")
                for eng_word in eng_sen_words: #for all words f in f do
                    idx_tur_in_dict =turkish_word_dict[tur_word]
                    idx_eng_in_dict = english_word_dict[eng_word]
                    count_e_f[idx_tur_in_dict][idx_eng_in_dict] +=
t_e_f_mat[idx_tur_in_dict][idx_eng_in_dict] / s_total[idx_word]
                    total_f[idx_eng_in_dict] += t_e_f_mat[idx_tur_in_dict][idx_eng_in_dict] /
s_total[idx_word]
                #end for
            #end for
        #end for

        print("ucuncu for loop giris ")
        print(str(datetime.now()))
        #estimate probabilities
        for eng_idx in  range(num_of_eng_word): #for all foreign words f do
            for tur_idx in range(num_of_tur_word): #for all English words e do
                if count_e_f[tur_idx][eng_idx] != 0 :
                    t_e_f_mat[tur_idx][eng_idx] = count_e_f[tur_idx][eng_idx] / total_f[eng_idx]
            #end for
        #end for

        print("finish ")
        print(str(datetime.now()))
    #end while

    print(t_e_f_mat)
    print(cnt_iter)

    return t_e_f_mat


def get_translation_prob(e,f,t,e_dict,f_dict):
```

```
    const = Common.const
    l_e = len(e)
    l_f = len(f)
    res = const / math.pow((l_f+1),l_e)
    for j in range(l_e):
        e_word = e[j]
        if e_word in e_dict:
            e_j = e_dict[e_word]
        else:
            print("word '"+ e_word +"' is not found in target language dictionary")
            continue
            #return 0

        sum = 0
        for i in range(l_f):
            f_word = f[i]

            if f_word in f_dict:
                f_i = f_dict[f_word]
                sum += t[e_j][f_i]
            else:
                print("word '" + f_word  +"' is not found in source language dictionary")

        res *= sum

    return res
```

IBMModel2.py

```
import numpy as np
from datetime import datetime
import Common


def train(t_e_f_mat, e_word_dict,f_word_dict,e_sentences,f_sentences,max_le,max_lf):
    print("IBMModel2 Starts " + str(datetime.now()))
    a_i_le_lf_mat = np.zeros((max_lf, max_le, max_lf,max_le), dtype=float)

    for lf in range(max_lf):
        a_i_le_lf_mat[:,:,lf,:] = 1/(lf+1)

    num_of_e_word = len(e_word_dict)
    num_of_f_word = len(f_word_dict)

    t_e_f_mat_prev = np.full((num_of_e_word, num_of_f_word), 1,dtype=float)
    cnt_iter = 0

    print("While starts " + str(datetime.now()))
    while not Common.is_converged(t_e_f_mat,t_e_f_mat_prev,cnt_iter) :
        print(cnt_iter)
        cnt_iter += 1
        t_e_f_mat_prev = t_e_f_mat.copy()
        count_e_f = np.full((num_of_e_word, num_of_f_word), 0, dtype=float)
        total_f = np.full((num_of_f_word),0, dtype=float)
        count_a_i_le_lf = np.zeros((max_lf, max_le, max_lf,max_le), dtype=float)
        total_a_j_le_lf = np.zeros((max_le,max_le,max_lf),dtype=float)

        print("Sentence pair loop starts " + str(datetime.now()))
        for idx_e, e_sen in enumerate(e_sentences): #for all sentence pairs (e,f) do
            #le = length(e), lf = length(f)
            e_sen_words = e_sen.split(" ")
            f_sen_words = f_sentences[idx_e].split(" ")
            l_e = len(e_sen_words)
            l_f = len(f_sen_words)

            #compute normalization
            s_total = np.full((l_e),0,dtype=float)
            for j in range(l_e): #for j = 1 .. le do // all word positions in e
                s_total[j] = 0 #s-total(ej) = 0
```

```
                e_word = e_sen_words[j]
                for i in range(l_f): #for i = 0 .. lf do // all word positions in f
                    f_word = f_sen_words[i]
                    e_j = e_word_dict[e_word]
                    f_i = f_word_dict[f_word]
                    s_total[j] += t_e_f_mat[e_j][f_i] * a_i_le_lf_mat[i][j][l_f-1][l_e-1] #s-total(ej) += t(ej|fi) * a(i|j,le,lf)
                #end for
            #end for

            #collect counts
            for j in range(l_e): #for j = 1 .. le do // all word positions in e
                e_word = e_sen_words[j]
                for i in range(l_f): #for i = 0 .. lf do // all word positions in f
                    f_word = f_sen_words[i]
                    e_j = e_word_dict[e_word]
                    f_i = f_word_dict[f_word]

                    c = t_e_f_mat[e_j][f_i] * a_i_le_lf_mat[i][j][l_f-1][l_e-1] / s_total[j] #c = t(ej|fi) * a(i|j,le,lf) / s-total(ej)
                    count_e_f[e_j][f_i] += c #count(ej|fi) += c
                    total_f[f_i] += c #total(fi) += c
                    count_a_i_le_lf[i][j][l_f-1][l_e-1] += c #counta(i|j,le,lf) += c
                    total_a_j_le_lf[j][l_e-1][l_f-1] += c #totala(j,le,lf) += c
                #end for
            #end for
        #end for

        print("Estimate Probabilities starts " + str(datetime.now()))
        #estimate probabilities
        t_e_f_mat = np.full((num_of_e_word, num_of_f_word), 0,dtype=float) #t(e|f) = 0 for all e,f
        a_i_le_lf_mat = np.zeros((max_lf, max_le, max_lf,max_le), dtype=float) #a(i|j,le,lf) = 0 for all i,j,le,lf
        for f_idx in  range(num_of_f_word): #for all foreign words f do
            for e_idx in range(num_of_e_word): #for all English words e do
                if count_e_f[e_idx][f_idx] != 0 :
                    t_e_f_mat[e_idx][f_idx] = count_e_f[e_idx][f_idx] / total_f[f_idx]
            #end for
        #end for

        print("Estimating alignments starts " + str(datetime.now()))
        for i in range(max_lf):
            for  j in range(max_le):
                for le in range(max_le):
                    for lf in range(max_lf):
                        if count_a_i_le_lf[i][j][lf][le] != 0 :
                            a_i_le_lf_mat[i][j][lf][le] = count_a_i_le_lf[i][j][lf][le] / total_a_j_le_lf[j][le][lf]

    print("While loop ends print starts  " + str(datetime.now()))

    print(t_e_f_mat)
    print("IBMModel2 Ends " + str(datetime.now()))
    return t_e_f_mat, a_i_le_lf_mat


def get_translation_prob(e,f,t,a,e_dict,f_dict):
    const = Common.const
    l_e = len(e)
    l_f = len(f)
    res = const
    for j in range(l_e):
        e_word = e[j]
        if e_word in e_dict:
            e_j = e_dict[e_word]
        else:
            print(Common.O + "word '"+ e_word +"' is not found in target language dictionary" + Common.BL)
            continue
            #return 0
```

```
        sum = 0
        for i in range(l_f):
            f_word = f[i]

            if f_word in f_dict:
                f_i = f_dict[f_word]
                sum += t[e_j][f_i]*a[i][j][l_f-1][l_e-1]
            else:
                print(Common.B + "word '" + f_word  +"' is not found in source language
dictionary"+ Common.BL)

        res *= sum

    if res == const:
        return 0
    return res
```

IBMModel3.py

```
import numpy as np
import math
from datetime import datetime
import Common

def probability(a,e,f):
    fi_0 = len(e) - len(f) # buraya dikkat hata olabilir
    if fi_0 < 0:
        fi_0 = 0
    null_insert_prob = Common.nCr(len(e)-fi_0,fi_0) * (math.pow(p1, fi_0)) * (math.pow(p0,
(len(e)-2*fi_0)))
    fertility_prob = 1
    for i in range(len(f)):
        f_word = f[i]
        f_i = f_word_dict[f_word]

        fertility = 0
        for j in range(len(e)):
            if i == a[j] : fertility += 1
        fertility_prob *= Common.factorial(fertility) *n_fi_f[fertility][f_i]

    lexical_distortion_prob = 1
    for j in range(len(e)):
        e_word = e[j]
        f_word = f[a[j]]
        e_j = e_word_dict[e_word]
        f_i = f_word_dict[f_word]
        lexical_distortion_prob *= t_e_f_mat[e_j][f_i] * d_i_j_le_lf_mat[a[j]][j][len(f)-
1][len(e)-1]

    return null_insert_prob * fertility_prob * lexical_distortion_prob


def neighboring(a,j_pegged,e_words,f_words):
    N = []
    l_f = len(f_words)
    l_e = len(e_words)
    #N.append(a) # bu satırı ben ekledim.
    for neg_j in range(l_e):
        if neg_j != j_pegged:
            for neg_i in range(-1,l_f):
                # !!!!!!!!!!! buraya neg_i != a[j_pegged]= i kontrolü gelmeli mi !!!!!
                neg_a = a.copy()
                neg_a[neg_j] = neg_i
                N.append(neg_a.copy())

    for j_1 in range(l_e):
        if j_1 != j_pegged:
            for j_2 in range(l_e):
```

```
                if j_2 != j_pegged and j_2 != j_1 :
                    neg_a = a.copy()
                    temp = neg_a[j_1]
                    neg_a[j_1] = neg_a[j_2]
                    neg_a[j_2] = temp
                    N.append(neg_a.copy())
    return N


def hillclimb(a,j_pegged,e_words,f_words):
    a_old = []
    while a != a_old:
        a_old = a.copy()
        for a_neighbor in neighboring(a,j_pegged,e_words,f_words):
            if probability(a_neighbor,e_words,f_words) > probability(a,e_words,f_words):
                a = a_neighbor.copy()
    return a


def sample(e_words, f_words):
    A = []
    a = []
    l_f = len(f_words)
    l_e = len(e_words)
    for j in range(l_e):
        a.append(-1)

    for j in range(l_e):
        for i in range(-1,l_f):
            a[j] = i
            for neg_j in range(l_e):
                if neg_j != j:
                    #a(j') = argmaxi' t(ej' |fi' ) d(i'|j', length(e), length(f))
                    argmaxi = 0
                    for neg_i in range(-1,l_f):
                        #if neg_i != i:
                        f_word = f_words[neg_i]
                        e_word = e_words[neg_j]
                        e_j = e_word_dict[e_word]
                        f_i = f_word_dict[f_word]
                        temp = t_e_f_mat[e_j][f_i] * d_i_j_le_lf_mat[neg_i][neg_j][l_f-1][l_e-1]
                        if temp > argmaxi:
                            argmaxi = temp
                            a[neg_j] = neg_i
            #end for
            a = hillclimb(a,j,e_words,f_words)
            N = neighboring(a,j,e_words,f_words)
            #!!!!!!!!!!!!!!!!burada a da sample set e eklenmeli mi? !!!!!!!!!!!
            if N != []:
                for n in N:
                    #if not n in A:
                    A.append(n)
        #end for
    #end for
    return A


def train(arg_t_e_f_mat, arg_d_i_le_lf_mat,
arg_e_word_dict,arg_f_word_dict,e_sentences,f_sentences,max_le,max_lf):
    print("IBMModel3 Starts " + str(datetime.now()))
    global t_e_f_mat, d_i_j_le_lf_mat, n_fi_f, e_word_dict,f_word_dict, p0, p1

    t_e_f_mat = arg_t_e_f_mat
    d_i_j_le_lf_mat = arg_d_i_le_lf_mat
    e_word_dict = arg_e_word_dict
    f_word_dict = arg_f_word_dict
    p0 = 0.5
    p1 = 0.5

    num_of_e_word = len(e_word_dict)
    num_of_f_word = len(f_word_dict)
```

```
    max_fertility = 20

    n_fi_f =  np.full((max_fertility, num_of_f_word), 1/max_fertility, dtype=float) #φ(n|f) = 0
for all f,n

    t_e_f_mat_prev = np.full((num_of_e_word, num_of_f_word), 1,dtype=float)
    cnt_iter = 0

    print("While starts " + str(datetime.now()))
    while not Common.is_converged(t_e_f_mat,t_e_f_mat_prev,cnt_iter) :
        print(cnt_iter)
        cnt_iter += 1
        t_e_f_mat_prev = t_e_f_mat.copy()
        #set all count* and total* to 0
        count_t = np.full((num_of_e_word, num_of_f_word), 0, dtype=float)
        total_t = np.full((num_of_f_word),0, dtype=float)
        count_d = np.zeros((max_lf, max_le, max_lf,max_le), dtype=float)
        total_d = np.zeros((max_le,max_le,max_lf),dtype=float)
        count_f = np.full((max_fertility, num_of_f_word), 0, dtype=float)
        total_f = np.full((num_of_f_word),0, dtype=float)
        count_p0 = 0
        count_p1 = 0

        for idx_e, e_sen in enumerate(e_sentences): #for all sentence pairs (e,f) do
            print("we are in sentence " + str(idx_e) + "  " + str(datetime.now()))
            #le = length(e), lf = length(f)
            e_sen_words = e_sen.split(" ")
            f_sen_words = f_sentences[idx_e].split(" ")
            l_e = len(e_sen_words)
            l_f = len(f_sen_words)
            A = sample(e_sen_words,f_sen_words)
            total_null = 0
            # collect counts
            c_total = 0
            for a in A: #for all a ∈ A do ctotal += probability( a, e, f );
                c_total += probability(a,e_sen_words,f_sen_words)


            for a in A: #for all a ∈ A do
                c = probability(a,e_sen_words,f_sen_words) /c_total #c = probability( a, e, f ) /
ctotal
                total_null = 0
                for j in range(l_e): #for j = 0 .. length(f) do
                    e_word = e_sen_words[j]
                    f_word = f_sen_words[a[j]]
                    e_j = e_word_dict[e_word]
                    f_i = f_word_dict[f_word]
                    count_t[e_j][f_i] += c              # lexical translation
                    total_t[f_i] += c                   # lexical translation
                    count_d[a[j]][j][l_f-1][l_e-1] += c #distortion
                    total_d[a[j]][l_e-1][l_f-1] += c    #distortion
                    if a[j] == -1: total_null += 1       #if a(j) == 0 then null++; // null
insertion
                #end for

                #countp1 += null * c; countp0 += (length(e) - 2 * null) * c
                count_p1 += total_null * c
                count_p0 += (l_e-2*total_null)*c

                for i in range(l_f):
                    fertility = 0
                    for j in range(l_e):
                        if i == a[j]: fertility += 1
                    #end for
                    f_word = f_sen_words[i]
                    f_i = f_word_dict[f_word]
                    count_f[fertility][f_i] += c
                    total_f[f_i] += c
                #end for
            #end for
        #end for
```

```
        #estimate probability distribution
        t_e_f_mat = np.full((num_of_e_word, num_of_f_word), 0,dtype=float) #t(e|f) = 0 for all e,f
        d_i_j_le_lf_mat = np.zeros((max_lf, max_le, max_lf,max_le), dtype=float) #d(j|i,le,lf) = 0 for all i,j,le,lf
        n_fi_f =  np.full((max_fertility, num_of_f_word), 0, dtype=float) #φ(n|f) = 0 for all f,n

        #for all (e,f) in domain( countt ) do t(e|f) = countt(e|f) / totalt(f)
        for f_idx in  range(num_of_f_word): #for all foreign words f do
            for e_idx in range(num_of_e_word): #for all English words e do
                if count_t[e_idx][f_idx] != 0 :
                    t_e_f_mat[e_idx][f_idx] = count_t[e_idx][f_idx] / total_t[f_idx]
            #end for
        #end for

        #for all (i,j,le,lf) in domain( countd ) do
        for i in range(max_lf):
            for  j in range(max_le):
                for le in range(max_le):
                    for lf in range(max_lf):
                        if count_d[i][j][lf][le] != 0 :
                            d_i_j_le_lf_mat[i][j][lf][le] = count_d[i][j][lf][le] / total_d[j][le][lf]

        for fi in range(max_fertility):
            for j in range(num_of_f_word):
                if count_f[fi][j] != 0:
                    n_fi_f[fi][j] = count_f[fi][j] / total_f[j]

        p1 = count_p1 / (count_p1 + count_p0)
        p0 = 1-p1
        print(t_e_f_mat)

    print("While loop ends print starts  " + str(datetime.now()))


    print("IBMModel3 Ends " + str(datetime.now()))
    return t_e_f_mat, d_i_j_le_lf_mat,n_fi_f,p0,p1


def get_translation_prob(e,f,t,a,e_dict,f_dict):
    const = Common.const
    l_e = len(e)
    l_f = len(f)
    res = const
    for j in range(l_e):
        e_word = e[j]
        if e_word in e_dict:
            e_j = e_dict[e_word]
        else:
            print("word '"+ e_word +"' is not found in target language dictionary")
            continue
            #return 0

        sum = 0
        for i in range(l_f):
            f_word = f[i]

            if f_word in f_dict:
                f_i = f_dict[f_word]
                sum += t[e_j][f_i]*a[i][j][l_f-1][l_e-1]
            else:
                print("word '" + f_word  +"' is not found in source language dictionary")

        res *= sum

    return res
```

Common.py

```python
import math


def is_converged(new,old,num_of_iterations):
    epsilone = 0.00000001
    if num_of_iterations > max_num_of_iterations :
        return True

    for i in range(len(new)):
        for j in range(len(new[0])):
            if math.fabs(new[i][j]- old[i][j]) > epsilone:
                return False
    return True


def nCr(n,r):
    try:
        if n-r < 0 :
            return 1
        f = math.factorial
        return f(n) / f(r) / f(n-r)
    except:
        print( "value error " + str(n) + "  " + str(r))
        raise


def factorial(n):
    if n < 0 :
        return 1
    f = math.factorial
    return f(n)


const = 0.1
num_of_train_sample = 4000
num_of_test_sample = 100
max_num_of_iterations = 15

#colors for print W  = '\033[0m'  # white (normal)
BL = '\033[30m' # black
R = '\033[31m' # red
G = '\033[32m' # green
O = '\033[33m' # orange
B = '\033[34m' # blue
P = '\033[35m' # purple
```