A Report On

Cross Lingual Document Translator

Submitted

by **Group10**

Shantanu Vichare	**(2016A3PS0156P)**
Ritik Kandoria	**(2017A7PS0009P)**
Rajababu Sajkia	**(2017A7PS0007P)**
Advait Deshmukh	**(2017A7PS0155P)**
Rohit Sunil Bohra	**(2017A7PS0225P)**
Ashish Jayant Prabhune	**(2017A7PS0231P)**

For the Course

CS F469 Information Retrieval

Date - 6th November, 2019

<u>Task</u>

The task of the assignment is to create a Cross Lingual Document Translator, using Statistical Machine Translation model. The model can translate a Dutch document to an English document and vice versa. It also calculates the Cosine similarity and Jaccard Coefficient between two documents.

<u>Assumptions</u>

1. Dictionaries used to store probabilities while training are of the form
   Key= (string, string): Value=Float and will fit in memory.

2. Python's standard libraries have been used for processing steps. The pickle library is used to save and load probabilities from files and will only be used if the pre-trained model is required to be loaded from disk.

3. The model won't be used for prediction if not even trained for a single Expectation-Maximization (EM) iteration.

4. The log-likelihood function values can be observed to decide the number of EM iteration the model requires to converge.

5. The pre-trained model uses the IBM model 1 to train the conditional probabilities for 5 iterations and IBM model 2 uses these probabilities as its starting point and trains the probabilities for another 5 iterations.

6. Since the model inherently lacks realignment capabilities, it can only perform lexical translation without any reordering.

7. If the testing dataset has words that haven't been trained for, the model will simply skip these words.

8. Digits are not mapped by this model instead they are taken as given in the input document.

Implementation

Note: The model is explained assuming translation is from Dutch to English, although the model can be trained for English to Dutch as well.

Utility module:

The function *process_string* calls the function *preprocess_string* passing a string as an argument and removes digits from it.

The function *preprocess_string* takes a string and strips the leading and ending white spaces and punctuations.

The function *get_data* takes the data and calls *process_string* for each line of the file.

The function *run* splits the data into training and validation data and calls the *run_iter* function on the training data which calls the EM iterations over the data.

IBM model 1:

The function __*init*__ initializes the dictionary ‘t’.

The dictionary ‘t’ is in the form of key-value pair of tuple (English,Dutch) to float, which stores the conditional probability of p(English|Dutch).

The function *run_iter* executes *em_iter* function *NumIter* times.

Since the corpus size is tremendous, the data is sliced into batches and then Expectation is calculated batch-wise followed by Maximization at last.

The function *em_iter* runs an iteration of expectation maximization process:

- C1 is a dictionary which stores the count of (English|Dutch). – c(e|d)
- C2 is a dictionary which stores the count of total counts of a given Dutch word – total(d)
- *den* is a variable which stores the sum of probabilities of all English-Dutch pairs for a given Dutch sentence – s-total(e)

- This function updates the counts and the conditional probabilities of t[English|Dutch].
- The log-likelihood calculation predicts convergence values for each iteration.

The function *update_maxprob_words* derives the final mapping of Dutch-English pair which has the maximum probability for each pair. The mapping is stored in *wordmap* using the maximum of probabilities in ‘t’.

The function *get_translation* translates the given English sentence into Dutch and vice versa. The function takes the sentence, eliminates all the punctuations and extra white spaces and then gives the word by word output using ‘*wordmap’* mapping. Since the numbers remain the same in both languages, they are not translated and inserted directly in the translation in corresponding position.

The function *translate* calls the function *get_translation* for each sentence in the data.

IBM model 2:

The function __*init*__ initializes the term probabilities, word dictionary and mapping.

The function *run_iter* calls the *em_iter* function *NumIter* number of times. It also splits the data into batches of ‘step’ due to the large size of dataset.

The function *em_iter* executes the expectation maximization procedure.

- C1 is a dictionary which stores the count of (English|Dutch). – c(e|d)
- C2 is a dictionary which stores the count of total counts of a given dutch word – total(d)
- C3 is a dictionary stores the count of alignments – $c(j|i,l_e,l_d)$.
- C4 is a dictionary stores the count of alignments – $c(i,l_e,l_d)$.
- ‘q’ stores the alignment probability i.e. $a(j|i,l_e,l_d)$.
- Here, *j* is the position of a word in Dutch sentence and *i* is the position of a word in English sentence while l_e and l_d are the lengths of the sentences.
- The log-likelihood calculation predicts convergence values for each iteration.

The function *update_maxprob_words* derives the final mapping of Dutch-English pair which has the maximum probability for each pair. The mapping is stored in *wordmap* using the maximum of probabilities in ‘t’.

The function *get_translation* can translate the given Dutch sentence into English and vice versa. The function takes the sentence, eliminates all the punctuations and extra white spaces and then gives the word by word output using ‘*wordmap’* mapping. Since the numbers remain the same in both languages, they are not translated and inserted directly in the translation in corresponding position.

The function *translate* calls the *get_translation* for each sentence in the data.

<u>IBM model 1 pseudocode</u>:

```
initialize t(e|f) uniformly
 do until convergence
   set count(e|f) to 0 for all e,f
   set total(f) to 0 for all f
   for all sentence pairs (e_s,f_s)
     set total_s(e) = 0 for all e
     for all words e in e_s
       for all words f in f_s
         total_s(e) += t(e|f)
     for all words e in e_s
       for all words f in f_s
         count(e|f) += t(e|f) / total_s(e)
         total(f)   += t(e|f) / total_s(e)
   for all f words
     for all e words
       t(e|f) = count(e|f) / total(f)
```

Innovation –

1. IBM model 2 is implemented which takes into consideration the distortion probability. The alignment probability is calculated and updated for each Dutch-English sentence pair. IBM model 2 gives better results in comparison to model 1 as the information of the alignment is incorporated by the distortion probabilities.

2. Since the entire data doesn't fit into the memory in the required format, we break the data into slices (Batch of sentences) which allows us to process the entire dataset given the constraints of our system.

Limitations –

1. The IBM model 1 doesn't consider the alignment of the words in the sentence and their translations. It translates a sentence by finding the word by word translation considering the best mappings based on training dataset. The alignment probabilities are not calculated.
2. A given sentence can be translated into another language in multiple ways. For example, a sentence of 5 words maybe translated into an English sentence of 6 or 8 or 10 words. The model will not be able to choose and eliminate the redundant words accurately, hence may give inaccurate results.
3. The limitation of IBM model 1 is its inability to give good results when one word is translated to multiple words of different language. For example, let's say a Dutch word 'Visserijbeleid' is translated into 'Fishing policy 'and 'Fishery policy' and 'Fisheries policy'. The dictionary will have the maximum 'policy' with higher probability (say 0.39) and 'fishing' and 'fishery' with lower probabilities (say 0.33). So, the mapping will choose only 'policy' rather than choosing 'fishing policy'. It will concatenate only when the maximum probabilities are same for multiple words. Hence due to the inaccuracy of our model, it will translate 'Visserijbeleid' into 'policy'.
4. This model doesn't take semantics of a sentence into consideration while translating unlike modern translators.

Code snippets –

# IBM Model 1 EM function:

```
def em_iter(self, sliced_bitext, iterNum):
"""Runs a single iteration of the EM algorithm on the model"""
likelihood = 0.0
c1 = defaultdict(float)
c2 = defaultdict(float)

for sentNum,(lang1_slice,lang2_slice) in enumerate(sliced_bitext):

      # Final Processing of data
      lang1_slice = [sent.split() for sent in lang1_slice]
      lang2_slice = [sent.split() for sent in lang2_slice]
      corpus_slice = zip(lang1_slice, lang2_slice)

      # The E-Step
      for k,(e, d) in enumerate(corpus_slice):
            d = [None] + d
            l = len(d)   # l+1
            m = len(e)   # m
            q = 1.0      # alignment probability is equal for all unlike IBM 2

            for i in range(0, m):
                  num = [ q * self.t[(e[i], d[j])] for j in range(0,l) ]
                  den = float(sum(num))
                  likelihood += math.log(den)
```

```
                    for j in range(0, l):
                          delta = num[j] / den
                          c1[(e[i], d[j])] += delta
                          c2[(d[j],)]      += delta
            if(k%1000==999):
            print('\rIter:%d  Sentences:%d    '%(iterNum,  sentNum*10000+k+1),
            end='')

      # The M-Step
      self.t = defaultdict(float, { k: (v/c2[k[1:]]) for k, v in c1.items() if
      v>0.0})
      print('\tLog-likelihood: %.5f'%(likelihood))
return
```

# IBM Model 2 EM function:

```
def em_iter(self, sliced_bitext, iterNum):
"""Runs a single iteration of the EM algorithm on the model"""
likelihood = 0.0
c1 = defaultdict(float)
c2 = defaultdict(float)
c3 = defaultdict(float)
c4 = defaultdict(float)

for sentNum,(lang1_slice,lang2_slice) in enumerate(sliced_bitext):

      # Final Processing of data
      lang1_slice = [sent.split() for sent in lang1_slice]
      lang2_slice = [sent.split() for sent in lang2_slice]
      corpus_slice = zip(lang1_slice, lang2_slice)
```

```
    # The E-Step
    for k,(e, d) in enumerate(corpus_slice):
        d = [None] + d
        l = len(d)   # l+1
        m = len(e)   # m

        for i in range(0, m):
            num = [ 10000 * self.q[(j, i, l, m)] * self.t[(e[i], d[j])]
            for j in range(0,l) ]
            den = float(sum(num))
            likelihood += math.log(den)

            for j in range(0, l):
                delta = num[j] / den
                c1[(e[i], d[j])] += delta
                c2[(d[j],)]      += delta
                c3[(j, i, l, m)] += delta
                c4[(i, l, m)]    += delta
        if(k%1000==999):
        print('\rIter:%d  Sentences:%d    '%(iterNum,  sentNum*10000+k+1),
        end='')

    # The M-Step
    self.t = defaultdict(float, { k: (v/c2[k[1:]]) for k, v in c1.items() if
    v>0.0})
    self.q = defaultdict(float, { k: (v/c4[k[1:]]) for k, v in c3.items() if
    v>0.0})
    print('\tLog-likelihood: %.5f'%(likelihood))
return
```

# Auxiliary function for getting tf vectors:

```
def get_logvectors(ls1,ls2):
    '''Returns tf vector representations of documents'''
    def get_vector(d):
        vec = np.array([v for v in d.values()], dtype=float)
        for i,v in enumerate(vec):
            if(v>0):
                vec[i] = 1+np.log10(v)
            else:
                vec[i] = 0

        return vec

    d1 = defaultdict(int)
    d2 = defaultdict(int)

    for s1 in ls1:
        s1 = utils.preprocess_string(s1).split()
        for w in s1:
            d1[w] += 1
            d2[w] = d2[w]
    for s2 in ls2:
        s2 = utils.preprocess_string(s2).split()
        for w in s2:
            d2[w] += 1
            d1[w] = d1[w]

    return get_vector(d1),get_vector(d2)
```

```
# Cosine similarity function:
def cosine_similarity(ls1,ls2):
    '''Computes cosine similarity of two documents'''
    v1,v2 = get_logvectors(ls1,ls2)
    v1 = v1/np.linalg.norm(v1)
    v2 = v2/np.linalg.norm(v2)
    return np.dot(v1,v2)

# Jaccard Coefficient function:
def jaccard_coefficient(ls1,ls2):
    '''Computes jaccard coefficient of two documents'''
    set1 = set()
    set2 = set()
    for s1 in ls1:
        s1 = utils.preprocess_string(s1).split()
        for w in s1:
            set1.add(w)
    for s2 in ls2:
        s2 = utils.preprocess_string(s2).split()
        for w in s2:
            set2.add(w)

    intersection = len(set1.intersection(set2))
    union = len(set1.union(set2))
    return float(intersection) / union
```