COMP-8730_Assignment-1 Spell Correction Using MED

Student Information

- Name: Jiajie Yang
- UWin Acc: yang4q
- Student ID: 110115897

```
!python --version
!pip install "pytrec-eval-terrier"
```

```
Python 3.8.10
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: pytrec-eval-terrier in /usr/local/lib/python3.8/dist-packages (0.5.5)
```

```
import sys
import nltk
nltk.download('omw-1.4')
#assert(nltk.download('wordnet'))
nltk.download('wordnet')
from nltk.corpus import wordnet as wn
import numpy
import heapq
```

```
[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
[nltk_data] Downloading package wordnet to /root/nltk_data...
```

Levenshtein Distance Pseudocode

Algorithm Min-Edit-Distance;
Input: Strings of source and target
Output: Minimum distance between source and target

```
n <- Len(source)
m <- Len(target)
Create a distance matrix, distance[n+1, m+1]

Initialization: the first row and column is the distance between the empty string and itself
D[0,0] = 0
for i from 1 -> n do:
    D[i,0] <- D[i-1,0] + del-cost(source[i])
for j from 1 -> m do:
    D[0,j] <- D[0,j-1] + ins-cost(target[j])

Recurrence relation:
for i from 1 -> n do:
    for j from 1 -> m do:
        D[i,j] <- Min(D[i-1,j] + del-cost(source[i]),
                      D[i-1,j-1] + sub-cost(source[i],target[j]),
                      D[i,j-1] + ins-cost(target[j])

return D[n,m]
```

```
def min_edit_distance(src, tar):
    n = len(src)
    m = len(tar)
    distances = numpy.zeros((n + 1, m + 1))

    for i in range(n + 1):
        distances[i][0] = i

    for j in range(m + 1):
        distances[0][j] = j
```

```
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if (src[i-1] == tar[j-1]):
                distances[i][j] = distances[i - 1][j - 1]
            else:
                distances[i][j] = 1 + min(distances[i - 1][j],     # delete
                                          distances[i - 1][j - 1], # substitute
                                          distances[i][j - 1])     # insert

    return distances[n][m]

# Tests:
assert min_edit_distance("c", "cu") == 1
assert min_edit_distance("cut", "ct") == 1
assert min_edit_distance("arccot", "arccos") == 1
```

Given Incorrect Spell to Find Top-K Similar Words

Goal: We want to find the k smallest minimum edit distance from the given incorrect spell among all words in the dictionary D by using a max heap

Algorithm Find-Top-K-Similar;

Input: Incorrect spell string, inc_str; integer, k

Output: A list of most similar words of length k

```
# Implementation of Customized Max Heap
class MaxHeap_MED_Word:

    def __init__(self, maxsize):

        self.maxsize = maxsize
        self.size = 0
        self.Heap = [[]] * self.maxsize
        self.Root = 0

    # Function to return True if self is an empty heap; False, otherwise
    def is_empty(self):

        return self.size == 0

    # Function to return True if self is full; False, otherwise
    def is_full(self):

        return self.size == self.maxsize

    # Function to return the index of the last index
    def last(self):

        return self.size - 1

    # Function to return the position of parent for the node currently at pos
    def parent(self, pos):

        return (pos - 1) // 2

    # Function to return the position of the left child for the node currently at pos
    def leftChild(self, pos):

        return 2 * pos + 1

    # Function to return the position of the right child for the node currently at pos
    def rightChild(self, pos):

        return 2 * (pos + 1)

    # Function to return True if pos has left child; False, otherwise
    def has_leftChild(self, pos):

        return self.leftChild(pos) < self.size;

    # Function to return True if pos has right child; False, otherwise
    def has_rightChild(self, pos):

        return self.rightChild(pos) < self.size;
```

```
# Function that returns true if the passed node is a leaf node
def isLeaf(self, pos):

    return self.leftChild(pos) >= self.size and pos < self.size

# Function that returns true if the passed node is a leaf node
def isRoot(self, pos):

    return pos == 0

# Function to swap two nodes of the heap
def swap(self, pos_1, pos_2):

    tmp = self.Heap[pos_1]
    self.Heap[pos_1] = self.Heap[pos_2]
    self.Heap[pos_2] = tmp
    return

# Function to heapify the node at pos
def maxHeapify(self, pos):

    while not self.isLeaf(pos):
        large_idx = self.leftChild(pos)
        if self.has_rightChild(pos) and (self.Heap[self.rightChild(pos)][0] >
                                         self.Heap[large_idx][0]):
            large_idx = self.rightChild(pos)
        if self.Heap[pos][0] >= self.Heap[large_idx][0]:
            break
        self.swap(pos, large_idx)
        pos = large_idx
    return

# Function to insert a node into the heap
def insert(self, element):

    if self.size >= self.maxsize:
        return
    self.size += 1
    self.Heap[self.size - 1] = element

    current = self.size -1

    while (not self.isRoot(current) and
           self.Heap[current][0] > self.Heap[self.parent(current)][0]):
        self.swap(current, self.parent(current))
        current = self.parent(current)
    return

# Function to return the maximum element from the heap
def top(self):

    return self.Heap[self.Root]

# Function to return the maximum element's key from the heap
def topKey(self):

    return self.Heap[self.Root][0]

# Function to remove and return the maximum element from the heap
def extractMax(self):

    popped = self.Heap[self.Root]
    self.Heap[self.Root] = self.Heap[self.last()]
    self.size -= 1
    if not self.is_empty():
        self.maxHeapify(self.Root)

    return popped

# Function to print the contents of the heap
def Print(self):

    for i in range(0, self.size):
        sl = self.Heap[i]
        if self.has_rightChild(i):
            lc = self.Heap[self.leftChild(i)]
```

```
                rc = self.Heap[self.rightChild(i)]
                print("PARENT-" + str(i) + ": (" + str(sl[0]) + "," + sl[1] +
                          ") ->LEFT CHILD: (" + str(lc[0]) + "," + lc[1] +
                          ") ->RIGHT CHILD: (" + str(rc[0]) + ", " + rc[1] + ")")
            elif self.has_leftChild(i):
                lc = self.Heap[self.leftChild(i)]
                print("PARENT-" + str(i) + ": (" + str(sl[0]) + "," + sl[1] +
                          ") ->LEFT CHILD: (" + str(lc[0]) + "," + lc[1] + ")")
            elif self.isLeaf(i):
                print("PARENT(Leaf)-" + str(i) + ": (" + str(sl[0]) + "," + sl[1] + ")")

# Tests:
tuple1 = (99, "worldsmall")
tuple2 = (990, "worldbig")
tuple3 = (32, "Msmall")
tuple4 = (46, "Mbig")
tuple5 = (120, "BigSS")
maxHeap = MaxHeap_MED_Word(4)
maxHeap.insert(tuple1)
maxHeap.insert(tuple2)
maxHeap.insert(tuple3)
maxHeap.insert(tuple4)
#maxHeap.Print()
print("The Max val is (" + str(maxHeap.top()[0]) + ", " + maxHeap.top()[1] + ")")
maxHeap.extractMax()
maxHeap.insert(tuple5)
#maxHeap.Print()
print("The Max val is (" + str(maxHeap.top()[0]) + ", " + maxHeap.top()[1] + ")")
maxHeap.extractMax()
#maxHeap.Print()
print("The Max val is (" + str(maxHeap.top()[0]) + ", " + maxHeap.top()[1] + ")")
maxHeap.extractMax()
#maxHeap.Print()
print("The Max val is (" + str(maxHeap.top()[0]) + ", " + maxHeap.top()[1] + ")")
maxHeap.Print()

     The Max val is (990, worldbig)
     The Max val is (120, BigSS)
     The Max val is (99, worldsmall)
     The Max val is (46, Mbig)
     PARENT-0: (46,Mbig) ->LEFT CHILD: (32,Msmall)
     PARENT(Leaf)-1: (32,Msmall)


def find_top_k_similar(inc_str, k):
    result = MaxHeap_MED_Word(k) # result is a k-element max heap
    for synset in wn.words():
        word = synset
        cur_tuple = (min_edit_distance(word, inc_str), word)
        if not result.is_full():
            result.insert(cur_tuple)
        elif cur_tuple[0] < result.topKey():
            result.extractMax()
            result.insert(cur_tuple)
    return result

def find_top_k_similar_using_synset(inc_str, k):
    result = MaxHeap_MED_Word(k) # result is a k-element max heap
    for synset in list(wn.all_synsets(lang='eng')):
        word = synset.name().split(".")[0]
        cur_tuple = (min_edit_distance(word, inc_str), word)
        if not result.is_full():
            result.insert(cur_tuple)
        elif cur_tuple[0] < result.topKey():
            result.extractMax()
            result.insert(cur_tuple)
    return result

# Tests:
def test_find_top_k():
    r1 = find_top_k_similar_using_synset("saalt", 1)
    r2 = find_top_k_similar_using_synset("sag", 2)
    r3 = find_top_k_similar_using_synset("desinged", 10)
    i = 0
    r3.Print()
    while not r3.is_empty():
        popped = r3.extractMax()
        print("r3[" + str(i) + "]: (" + str(popped[0]) + "," + popped[1] + ")")
```

```
        i = i + 1

    r3_using_words = find_top_k_similar("saalt", 10)
    i = 0
    while not r3_using_words.is_empty():
        popped = r3_using_words.extractMax()
        print("r3_words[" + str(i) + "]: (" + str(popped[0]) + "," + popped[1] + ")")
        i = i + 1
```

Compute Average S@K in Birkbeck Corpus

Method 1: Each iteration, we find the correct word with a prefix of '$' and calculate the s@k of the following incorrect words. Then, we record s@k's in a list of three elements for each k = {1, 5, 10} and calculate the averages finally

```
def avg_sak(lst_k, path):
    file_ = open(path, 'r')
    lines = file_.readlines()

    # Init total_sak_lst to be list of list of zero's to represent
    #     list of [# of successes, # of failures]
    total_sak_lst = []
    for elem in lst_k:
        total_sak_lst.append([0,0])

    largest_k = lst_k[-1] # the largest k in the list of all k values
    top_largest_k_lst = [""] * largest_k

    word_cor = ""
    word_inc_lst = []
    for l in lines:
        if '$' in l:
            if not word_cor == "" and not len(word_inc_lst) == 0:
                for inc_word in word_inc_lst:
                    top_largest_k_heap = find_top_k_similar(inc_word, largest_k)
                    for i in range(1, largest_k + 1):
                        top_largest_k_lst[largest_k - i] = top_largest_k_heap.extractMax()[1]
                    for i in range(0, len(lst_k)):
                        if word_cor in top_largest_k_lst[:(lst_k[i])]:
                            total_sak_lst[i][0] += 1
                        else:
                            total_sak_lst[i][1] += 1
                word_cor = ""
                word_inc_lst = []
            word_cor = l.split("$")[1].split("\n")[0].lower()
        else:
            word_inc_lst.append(l.split("\n")[0].lower())
    return total_sak_lst

# Test:
def test_avg_sak():
    print(avg_sak([1,5,10], '/content/birkbeck.dat'))
```

Method 2: Use the API, pytrec-eval-terrier

```
import pytrec_eval
import json

def eval_success(path):
    file_ = open(path, 'r')
    lines = file_.readlines()

    avg_success = [0,0,0]
    qrel = {}
    run = {}

    lst_k = [1, 5, 10]
    largest_k = lst_k[-1]
    word_cor = ""
    for l in lines:
        if '$' in l:
```

```
            word_cor = l.split("$")[1].split("\n")[0].lower()
        else:
            word_inc = l.split("\n")[0].lower()
            qrel[word_inc] = {word_cor : 1}
            run[word_inc] = {}
            # Find the k-closest wrods
            top_largest_k_heap = find_top_k_similar(word_inc, largest_k)
            for i in range(1, len(lst_k) + 1):
                cut_off = lst_k[len(lst_k) - i]
                key_val = 1 / cut_off
                interval = cut_off
                if not i == len(lst_k):
                    interval -= lst_k[len(lst_k) - i - 1]
                for j in range(interval):
                    predict = top_largest_k_heap.extractMax()[1]
                    run[word_inc][predict] = key_val
    # Apply evaluation measures
    evaluator = pytrec_eval.RelevanceEvaluator(qrel, {'success'})
    print(json.dumps(evaluator.evaluate(run), indent=2))
    res = evaluator.evaluate(run)
    print('Averages:')
    idx = 0
    for measure in list(res[list(res.keys())[0]].keys()):
        q = [query_measures[measure] for query_measures in res.values()]
        val = pytrec_eval.compute_aggregated_measure(measure, q)
        avg_success[idx] = val
        idx += 1
        print("  ", measure, val)
    return avg_success

eval_success('/content/birkbeck.dat')
```


```
      "success_10": 0.0
    },
    "ameraca": {
      "success_1": 1.0,
      "success_5": 1.0,
      "success_10": 1.0
    },
    "amercia": {
      "success_1": 0.0,
      "success_5": 1.0,
      "success_10": 1.0
    },
    "ameracan": {
      "success_1": 1.0,
      "success_5": 1.0,
      "success_10": 1.0
    },
    "apirl": {
      "success_1": 0.0,
      "success_5": 0.0,
      "success_10": 0.0
    },
    "austrain": {
      "success_1": 0.0,
      "success_5": 0.0,
      "success_10": 1.0
    },
    "badcock": {
      "success_1": 0.0,
      "success_5": 0.0,
      "success_10": 0.0
    },
    "bechuarnia_land": {
      "success_1": 0.0,
      "success_5": 0.0,
      "success_10": 0.0
    },
    "botuania": {
      "success_1": 0.0,
      "success_5": 0.0,
      "success_10": 1.0
    },
    "cambrige": {
```

```
  "canda": {
    "success_1": 0.0,
    "success_5": 0.0,
    "success_10": 1.0
  }
}
Averages:
   success_1 0.2727272727272727
   success_5 0.36363636363636365
   success_10 0.6363636363636364
[0.2727272727272727, 0.36363636363636365, 0.6363636363636364]
```

Parallel Execution

Idea: Since the spell correction is context-free, we have that the probabilty of successfully correcting a spelling error is independent from other errors. We can divide the large spelling error corpus C into $C_1, C_2, \ldots, C_n$, then apply the same algorithm, eval_success() on $C_i \forall i$ on different cores and run simultaneously. If we evenly divide C such that $|C_1| = |C_2| = \ldots = |C_n|$, ideally we can reduce the time complexity by a factor of n.