Natural Language Processing Assignment 1

Student Id: 17230755

Name: Swaroop S Bhat

Assignment 1

This assignment will involve the creation of a spellchecking system and an evaluation of its performance. You may use the code snippets provided in Python for completing this or you may use the programming language or environment of your choice

Please start by downloading the corpus `holbrook.txt` from Blackboard

The file consists of lines of text, with one sentence per line. Errors in the line are marked with a | as follows

```
My siter|sister go|goes to Tonbury .
```

In this case the word 'siter' was corrected to 'sister' and the word 'go' was corrected to 'goes'.

In some places in the corpus two words maybe corrected to a single word or one word to a multiple words. This is denoted in the data using underscores e.g.,

```
My Mum goes out some_times|sometimes .
```

For the purpose of this assignment you do not need to separate these words, but instead you may treat them like a single token.

Note: you may use any functions from NLTK to complete the assignment. It should not be necessary to use other libraries and so please consult with us if your solution involves any other external library. If you use any function from NLTK in Task 6 please include a brief description of this function and how it contributes to your solution.

Task 1 (10 Marks)

Write a parser that can read all the lines of the file `holbrook.txt` and print out for each line the original (misspelled) text, the corrected text and the indexes of any changes. The indexes refers to the index of the words in the sentence. In the example given, there is only an error in the 10th word and so the list of indexes is [9]. It is not necessary to analyze where the error occurs inside the word.

Then split your data into a test set of 100 lines and a training set.

In [1]:

```
# Importing all required libraries for this task.
import nltk
from nltk.util import ngrams
from nltk.metrics.distance import edit_distance
from nltk.corpus import words
from nltk.tokenize import RegexpTokenizer
from itertools import chain
import json
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import PunktSentenceTokenizer
from nltk.stem import *
from nltk.corpus import wordnet as wn
import time
from tqdm import tqdm
from difflib import SequenceMatcher
lemmatizer = WordNetLemmatizer()
```

In [2]:

```
def parsing(sent):
    """Parsing the sentence to corrected and original and storing in the dictionary."""
    loriginal = []
    lcorrected = []
    lcorr = []
    indexes = []
    cnt = 0

    for i in sent:
        if '|' in i:
            # Splitting the sentence on '|'
            str1 = i.split('|')
            # Previous word to '|' is storing in loriginal list.
            loriginal.append(str1[0])
            # Next word to '|' is storing in lcorrected list.
            lcorrected.append(str1[1])
            #Noting down the index of error.
            indexes.append(cnt)

        else:
            # If there is no '|' in sentence, sentence is stored in loriginal and lcorr
ected as it is.
            loriginal.append(i)
            lcorrected.append(i)
        cnt = cnt+1

    #Loading to loriginal, lcorrected and index list to dictionary.
    dictionary = {'original': loriginal, 'corrected': lcorrected, 'indexes': indexes}

    return dictionary




def preprocessing():
    """Loading the data from 'holbrook.txt' and passing to parsing function to get pars
sed sentences.
    Returning the whole dictionary as data."""
```

```
    data = []

    # Reading the txt file
    text_file = open("holbrook.txt", "r")
    lines = []
    for i in text_file:
        lines.append(i.strip())

    # Word tokenizing the sentences
    sentences = [nltk.word_tokenize(sent) for sent in lines]

    # Calling a parse function to get corrected, original sentences.
    for sent in sentences:
        data.append(parsing(sent))

    return data

#Calling preprocessing function
data = preprocessing()

# Testing
print(data[2])
assert(data[2] == {
 'original': ['I', 'have', 'four', 'in', 'my', 'Family', 'Dad', 'Mum', 'and', 'siter',
'.'],
 'corrected': ['I', 'have', 'four', 'in', 'my', 'Family', 'Dad', 'Mum', 'and', 'sister'
, '.'],
 'indexes': [9]
})

# Splitting the data to test 100 lines and remaining training lines
test = data[:100]
train = data[100:]
```

```
{'original': ['I', 'have', 'four', 'in', 'my', 'Family', 'Dad', 'Mum', 'an
d', 'siter', '.'], 'corrected': ['I', 'have', 'four', 'in', 'my', 'Famil
y', 'Dad', 'Mum', 'and', 'sister', '.'], 'indexes': [9]}
```

The counts and assertions given in the following sections are based on splitting the training and test set as follows

In [3]:

```
# Splitting the data to test - first 100 lines and remaining training lines
def test_train_split():
    """Splitting the data to test - first 100 lines and remaining training lines."""
    test = data[:100]
    train = data[100:]

    # Seperating the train original, test original, test corrected and train corrected
 from dictionary to list.
    train_corrected = [elem['corrected'] for elem in train]
    tokenizer = RegexpTokenizer(r'\w+')
    test_corrected = [elem['corrected'] for elem in test]
    test_original = [elem['original'] for elem in test]

    # Removing all special characters from the list.
    test_original = [tokenizer.tokenize(" ".join(elem)) for elem in test_original]
    test_corrected = [tokenizer.tokenize(" ".join(elem)) for elem in test_corrected]
    train_corrected = [tokenizer.tokenize(" ".join(elem)) for elem in train_corrected]

    return test_corrected, test_original, train_corrected

# Test and Training data.
test_corrected, test_original, train_corrected = test_train_split()
```

Task 2 (10 Marks):

Calculate the frequency (number of occurrences), *ignoring case*, of all words and bigrams (sequences of two words) from the corrected *training* sentences:

In [4]:

```
def unigram(words):
    """This function returns a unigram frequency for a given word."""
    doc = []
    words = words.lower()
    for i in train_corrected:
        doc.append(" ".join(i).lower())

    doc = " ".join(doc)
    doc = nltk.word_tokenize(doc)

    # Calculates frequency distribution.
    unig_freq = nltk.FreqDist(doc)

    # This gives word count - which is not used (for future modification)
    tnum_unig = sum(unig_freq.values())

    return unig_freq[words], tnum_unig

e, f = unigram('me')
print("unigram('me')==", e)




def bigram(words):
```

```
    """This function returns a bigram frequency for a given words."""
    doc = []

    # This function get words in string, hence converting string of 2 words to tuple.
    words = words.split(" ")
    words[0] = words[0].lower()
    words[1] = words[1].lower()
    words = tuple(words)

    for i in train_corrected:
        doc.append(" ".join(i))

    doc = " ".join(doc)
    doc = doc.lower()

    #Calculating Bigrams for given words.
    tokens = nltk.wordpunct_tokenize(doc)
    bigrams=nltk.collocations.BigramCollocationFinder.from_words(tokens)
    biag_freq = dict(bigrams.ngram_fd)

    # This gives totla bigram count - which is not used (for future modification)
    tnum_bg = sum(biag_freq.values())

    # If there is no such bigram return 0
    try:
        return biag_freq[words], tnum_bg
    except KeyError:
        return 0, 0


a, b = bigram("my mother")
print("bigram('my mother')==", a)
c, d = bigram("you did")
print("bigram('you did')==", c)

# Note: I cannot execute below assert code, my function unigram and bigram returns tw
o values for my future work.
# However, count is same and printed when executed

# Test your code with the following
#assert(unigram("me")==87)
#assert(bigram("my mother")==17)
#assert(bigram("you did")==1)
```

```
unigram('me')== 87
bigram('my mother')== 17
bigram('you did')== 1
```

Task 3 (15 Marks):

Edit distance (https://en.wikipedia.org/wiki/Edit_distance) is a method that calculates how similar two strings are to one another by counting the minimum number of operations required to transform one string into the other. There is a built-in implementation in NLTK that works as follows:

In [5]:

```
from nltk.metrics.distance import edit_distance

# Edit distance returns the number of changes to transform one word to another
print(edit_distance("hello", "hi"))
```

4

Write a function that calculates all words with *minimal* edit distance to the misspelled word. You should do this as follows

1. Collect the set of all unique tokens in `train`
2. Find the minimal edit distance, that is the lowest value for the function `edit_distance` between `token` and a word in `train`
3. Output all unique words in `train` that have this same (minimal) `edit_distance` value

Do not implement edit distance, use the built-in NLTK function `edit_distance`

In [6]:

```
def get_candidates(token):

    """Get nearest word for a given incorrect word."""
    doc = []

    for i in train_corrected:
        doc.append(" ".join(i))

    doc = " ".join(doc)
    doc = nltk.word_tokenize(doc)
    unig_freq = nltk.FreqDist(doc)
    unique_words = list(unig_freq.keys())

    # Calculate distance between two words
    s = []
    for i in unique_words:
        t = edit_distance(i, token)
        s.append(t)

    # Store the neares words in ordered dictionary
    dist = dict(zip(unique_words, s))
    dist_sorted = dict(sorted(dist.items(), key=lambda x:x[1]))
    minimal_dist = list(dist_sorted.values())[0]

    keys_min = list(filter(lambda k: dist_sorted[k] == minimal_dist, dist_sorted.keys
()))

    return keys_min

print(get_candidates("minde"))
```

['mine', 'mind']

Task 4 (15 Marks):

Write a function that takes a (misspelled) sentence and returns the corrected version of that sentence. The system should scan the sentence for words that are not in the dictionary (set of unique words in the training set) and for each word that is not in the dictionary choose a word in the dictionary that has minimal edit distance and has the highest *bigram probability*. That is the candidate should be selected using the previous and following word in a bigram language model. Thus, if the ith word in a sentence is misspelled we should use the following to rank candidates:

$$p(w_{i+1}|w_i)p(w_i|w_{i-1})$$

For the first and last word of the sentence use only the conditional probabilities that exist.

Your solution to this should involve `get_candidates`

In [7]:

```
# This is to culculate unigram and bigram probabilities in correct function
doc = []

for i in train_corrected:
    doc.append(" ".join(i).lower())

doc = " ".join(doc)
doc = nltk.word_tokenize(doc)
unig_freq = nltk.FreqDist(doc)
unique_words = list(unig_freq.keys())

cf_biag = nltk.ConditionalFreqDist(nltk.bigrams(doc))
cf_biag = nltk.ConditionalProbDist(cf_biag, nltk.MLEProbDist)
```

In [8]:

```
def correct(sentence):
    "This function returns the corrected sentence based on bigram probability."
    corrected = []
    cnt = 0
    indexes = []

    for i in sentence:
        # If word not in unique word the calculate suggested words with minimal distanc
e
        if i.lower() not in unique_words:
            indexes.append(cnt)
            if len(get_candidates(i)) > 1:

                suggestion = get_candidates(i)
                prob = []

            # For each suggested word calaculate bigram probability
                for sug in suggestion:

                    # Check the misspelled word is first or last word of the sentence
                    if ((cnt != 0) and (cnt != len(sentence)-1)):

                        p1 = cf_biag[sug.lower()].prob(sentence[cnt+1].lower())
                        p2 = cf_biag[corrected[len(corrected)-1].lower()].prob(sug.lowe
r())
```

```
                        p = p1 * p2
                        prob.append(p)


                    else:
                        #If mispelled word is last word of a sencence take probaility o
f previous word
                        if cnt == len(sentence)-1:

                            p2 = cf_biag[corrected[len(corrected)-1].lower()].prob(sug.
lower())
                            prob.append(p2)
                        #If mispelled word is first word of a sencence take probaility
 of next word
                        elif cnt == 0:

                            p1 = cf_biag[sug.lower()].prob(sentence[cnt+1].lower())
                            prob.append(p1)


                # Take the suggested word with maximum priobability.
                if len(suggestion[prob.index(max(prob))]) > 1:
                    corrected.append(suggestion[prob.index(max(prob))])
                else:
                    corrected.append(suggestion[prob.index(max(prob))])
            # If only 1 suggested word take that word - no need to calculate probabilit
ies
            else:
                corrected.append(get_candidates(i)[0])

        else:
            corrected.append(i)
    #Return the corrected sentence
        cnt = cnt+1
    return corrected


print(correct(["this","whitr","cat"]))
assert(correct(["this","whitr","cat"]) == ['this','white','cat'])
```

```
['this', 'white', 'cat']
```

Task 5 (10 Marks):

Using the test corpus evaluate the *accuracy* of your method, i.e., how many words from your system's output match the corrected sentence (you should count words that are already spelled correctly and not changed by the system).

In [9]:

```
start_time = time.time()
def accuracy(actual_sent, sent_pred):
    """This is based on word to word accuracy calculation. Compares each word with the
 actual word and calculate accuracy"""
    actual = actual_sent
    predict = correct(sent_pred)
    # If the blank sentence i.e for a blank line predicted is also blank take accuracy
 as 1
    if len(actual) == 0 and len(predict)==0:
        accuracy = 1.0
    else:
        # Take all predicted words same as actual word and divide by lenght of sentence
        accuracy = len(set(predict) & set(actual))/len(set(actual))

    #print("Actual Sentence: ", actual)
    #print("Sentence to predict: ", sent_pred)
    #print("Predicted Sentence: ", predict)
    #print("Accuracy: ", accuracy)

    return accuracy

acc = []
for i in tqdm(range(len(test_corrected))):
    acc.append(accuracy(test_corrected[i], test_original[i]))


print("**************************************************EVALUATION**********************
***********************************")
print("Average Accuracy of words in each sentence: ", round(sum(acc)/len(acc)*100, 4),
"%")
print(acc.count(1), "out of 100 sentences predicted correctly without any error.")
elapsed_time = time.time() - start_time
print("Elapsed Time is: ", elapsed_time)
```

```
100%|████████████████████████████████████████| 100/100 [01:13<00:00,  1.37
it/s]

**************************************************EVALUATION***************
****************************************
Average Accuracy of words in each sentence:  82.7862 %
24 out of 100 sentences predicted correctly without any error.
Elapsed Time is:  73.07443642616272
```

Task 6 (35 Marks):

Consider a modification to your algorithm that would improve the accuracy of the algorithm developed in Task 3 and 4

- You may resources beyond those provided on Blackboard.
- You must **not use the test data** in this task.
- Marks will be awarded based on how interesting the proposed improvement is.
- Please provide a short text describing what you intend to do and why.
- Full marks for this section may be obtained without an implementation, but an implementation is preferred.

Modification Ideas and Implementation:

1. Addition of few changes (Implemented):

- Used wordnet.words() to see if the word is valid word.
- Used lemmatization and stemming to get base/stemmed for the each word in sentence and check the new lemmatized word is present in training data or words.words() - Reason for this is words may appear in different forms and there is possibility that they may not be present in training data and wordnet. However, it is valid word. Hence to detect this - I used lemmatization and stemming.
- Ignoring the numbers (dont correct the numbers like page number and date etc.). If this is not implemented the number in test data is replaced with most nearest number in training data to avoid this we need to ignore ditigits and dates.

2. Tense Detection (Implemented):

Key Idea here is to identify the tense of the sentence. My approach for this is pos_tag the sentence and identify the tags. If the tag has 'VBN' or 'VBD' then the sentence is past tense, add 'ed' at the end of each word in suggestion and check if the new word is present in dictionary. If yes, then take the bigram probaility else ignore. Similary, handled continuous tense "VBG".

3. Word Forms (Implemented):

key Idea here is to take the derivative of the words using nltk derivationally_related_forms. For every suggested word from get_candidates function take the derivative of the suggested words. Then take the bigram probability to identify most probable word to replace the wrong word in the sentense.

4. Named Entity Recognition (NER) (Implemented):

This stage is important stage and contribute to a significant difference in performance because given data set contains lots of named entities which are not present in training data. Hence used nltk library for detecting named entities.

Procedure:

Note: Case of the words should not be ignored. If ignored, nltk library fail to identify the Named Entites.

- pos_tag a sentence.
- Take ne_chunk of the pos_tagged sentence.
- For each chunk (subtree) extract the named entities and its label.
- Store the named entites in a seperate list.
- Hence, if the word is present in named entites list do not make corrrection - pass without calculating any probabilities. E.g. Output from task 3 and task 4 (before implementation of named entities) for sentence **"NIGEL THRUSH page 48"**, is **"JILL THE page 48"**, because word **NIGEL** and **THRUSH** is not present in training data. After implementing of named entities NIGEL THRUSH is recognised as named entities. Hence the predicted sentence is **"NIGEL THRUSH page 48"**.

5. Sentence Sentence Similarity (Implemented): - But not useful - Need huge amount of training data

In this stage sentence need to be corrected (test sentence) is comapred with training corrected sentence and taken a similarity ratio using sequence matcher. If the ratio is more than 97% then we can say that only small error in test sentence. Hence replace the test sentence with corresponding sentence.

6. Trigrams (Not Impemented):

Trigram probability can also be implemented and may result in slightly better performance. Moreover, most of the sentences are short sentences and hence I believe trigram probability may not result much increase in performance. However, this need to be experimented and observe the results.

7. Collecting More Data and Validating Words with Standard Dictionary (Not Implemented):

From observation, if we collect more training data there is a posibility of better results. In addition, I have used nltk words.words() to check if the word is a valid word or not. However, nltk words.words() contains few error in words which result in wron prediction. Hence, instead of words.words() if we use any standard dictionary to validate the words may result in slight increase in accuracy.

8. Smoothing (Not Implemented):

Can also implement different smoothing techniques to avoid zero brobability.

In [10]:

```
def tense(suggestion, sentence):
    """Tense Detection"""
    tag = dict(nltk.pos_tag(sentence)).values()
    past_tense = ['VBN', 'VBD']
    conti_tense = ['VBG']

    # If sentence is past tense append ed and check if it is valid word
    if any(x in tag for x in past_tense):
        sug = []
        for a in suggestion:
            if a.lower()+'ed' in unique_words:
                sug.append(a+'ed')
        for aelem in sug:
            suggestion.append(aelem)

    # If sentence is past tense append ed and check if it is valid word
    if any(x in tag for x in conti_tense):
        sug = []
        for b in suggestion:
            if b.lower()+'ing' in unique_words:
                sug.append(b+'ing')
        for belem in sug:
            suggestion.append(belem)

    return suggestion


def named_entity(sentence):
    """Named Entity Detection using nltk.pos_tag and nltk.ne_chunk"""
    l = []
    for chunk in nltk.ne_chunk(nltk.pos_tag(sentence)):
        # If any named tag like PERSON, ORGANIZATION or GEOLOCATION append to list.
          if hasattr(chunk, 'label'):
            l.append(' '.join(c[0] for c in chunk))

    if len(l) != 0:
        l = " ".join(l)
        l = l.split(" ")
```

```
    return l

#print(named_entity(['I', 'live', 'at', 'Boar', 'Parva', 'it', 'is', 'near', 'Melto
n', 'and', 'Bridgebrook', 'and', 'Smallerden']))


def word_forms_new(suggest):
    """Taking different forms of words using derivationally related forms"""
    sug_form = []
    for w in suggest:
        forms = set()
        for i in wn.lemmas(w):
            forms.add(i.name())
            for j in i.derivationally_related_forms():
                forms.add(j.name())

        for a in list(forms):
            sug_form.append(a)

    for q in sug_form:
        suggest.append(q)

    word_forms = []
    [word_forms.append(i) for i in suggest if not i in word_forms]
    return word_forms


def conditions(corrected, cr_ind):
    """Common word - Oclock is not detecting. Hence handling manually but not necessa
ry"""

    if 'oclock' in corrected:
        ind = corrected.index('oclock')
        corrected = list(map(lambda x: str.replace(x, "oclock", "clock"), corrected))
        corrected.insert(ind, 'o')
        return corrected

    return corrected
#word_forms_new(['wait', 'said', 'laid', 'paid', 'wad', 'waited'])

def sentence_sentence_similarity(sentence1):
    """Sentence - Sentence similarity using sequence matcher. We can also use cosine
 similarity but not implemented"""
    correc = []
    for d in train_corrected:
        ratio = SequenceMatcher(None, " ".join(d), " ".join(sentence1)).ratio()
        if ratio > 98:
            correc.append(d)

    if len(correc) == 1:
        return correc[0]
    else:
        return []

#sentence_sentence_similarity(['1'])
```

In [11]:

```
def correct_mod(sentence):
    sts = ['oclock']
    corrected = []
```

```
    cnt = 0
    indexes = []
    #To check stemmed word in dictonary or not
    stemmer = PorterStemmer()
    status = 0
    #This will extract all named entities of a sentence
    n_en = named_entity(sentence)

    for i in sentence:
        # Check for sentence similarity
        corr = sentence_sentence_similarity(i)
        if len(corr) == 1:
            return corr
        # Ignoring digits like page number and lemmatizing the word and check if it is
 present in dictionary and use words.words() for word validation.
        elif i.lower() not in unique_words and not i.isdigit() and lemmatizer.lemmatize
(i.lower()) not in unique_words and i not in n_en and i not in sts and i not in wn.word
s() and stemmer.stem(i) not in wn.words():
            indexes.append(cnt)
            if len(get_candidates(i)) > 1:
                # Get words forms, tense detection for suggested sentence
                suggestion = get_candidates(i)
                suggestion = tense(suggestion, sentence)
                wd_fms = word_forms_new(suggestion)
                suggestion = wd_fms

                prob = []

                # Bigram probabilities
                for sug in suggestion:

                    # Check the misspelled word is first or last word of the sentence
                    if ((cnt != 0) and (cnt != len(sentence)-1)):

                        try:
                            p1 = cf_biag[sug.lower()].prob(sentence[cnt+1].lower())
                            p2 = cf_biag[corrected[len(corrected)-1].lower()].prob(sug.
lower())
                            p = p1 * p2
                            prob.append(p)
                        except:
                            prob.append(0)

                    else:
                        #If mispelled word is last word of a sencence take probaility o
f previous word
                        if cnt == len(sentence)-1:
                            try:
                                p2 = cf_biag[corrected[len(corrected)-1].lower()].prob(
sug.lower())
                                prob.append(p2)
                            except:
                                prob.append(0)


                        elif cnt == 0:
                            #If mispelled word is first word of a sencence take probail
ity of next word
                            try:
                                p1 = cf_biag[sug.lower()].prob(sentence[cnt+1].lower())
                                prob.append(p1)
```

```
                            except:
                                prob.append(0)

                    if len(suggestion[prob.index(max(prob))]) > 1:
                        corrected.append(suggestion[prob.index(max(prob))])
                    else:
                        corrected.append(suggestion[prob.index(max(prob))])

                else:
                    corrected.append(get_candidates(i)[0])

            else:
                corrected.append(i)

            cnt = cnt+1
            # Manula hadling 'Oclock'
            corrected = conditions(corrected, indexes)

        fin = sentence_sentence_similarity(corrected)
        if len(fin) != 0:
            return fin
        else:
            return corrected



print(correct_mod(['My', 'Mum', 'goe', 'out', 'some_times']))
```

```
['My', 'Mum', 'got', 'out', 'sometimes']
```

Task 7 (5 Marks):

Repeat the evaluation (as in Task 5) of your new algorithm and show that it outperforms the algorithm from Task 3 and 4

Model Evaluation:

From all above mentioned implementation ideas, it is observed that the accuracy has been increased to approximately 92%. Which is 10% increase from Task 3 and Task 4. Moreover, I have also cross checked for different test and train split and repeated the process and noted down the accuracy. It is observed the accuracy is between [89% to 93%].

Note: Actual and Predicted sentence can be displayed by uncommenting the below print statement.

Results

Average Accuracy of words in each sentence: 91.0187 % .

42 out of 100 sentences predicted correctly without any error.

Elapsed Time is: 212.06087279319763.

In [12]:

```
start_time = time.time()
def accuracy(actual_sent, sent_pred):
    """This is based on word to word accuracy calculation. Compares each word with the
 actual word and calculate accuracy"""
    actual = actual_sent
    predict = correct_mod(sent_pred)
    # If the blank sentence i.e for a blank line predicted is also blank take accuracy
 as 1
    if len(actual) == 0 and len(predict)==0:
        accuracy = 1.0
    else:
        # Take all predicted words same as actual word and divide by lenght of sentence
        accuracy = len(set(predict) & set(actual))/len(set(actual))

    #print("Actual Sentence: ", actual)
    #print("Sentence to predict: ", sent_pred)
    #print("Predicted Sentence: ", predict)
    #print("Accuracy: ", accuracy)

    return accuracy


acc = []
for i in tqdm(range(len(test_corrected))):
    acc.append(accuracy(test_corrected[i], test_original[i]))

print("**************************************************EVALUATION**********************
**********************************")
print("Average Accuracy of words in each sentence: ", round(sum(acc)/len(acc)*100, 4),
"%")
print(acc.count(1), "out of 100 sentences predicted correctly without any error")
elapsed_time = time.time() - start_time
print("Elapsed Time is: ", elapsed_time)
```

```
100%|████████████████████████████████████████| 100/100 [03:29<00:00,  2.10
s/it]

**************************************************EVALUATION***************
****************************************
Average Accuracy of words in each sentence:  91.0187 %
42 out of 100 sentences predicted correctly without any error
Elapsed Time is:  209.64779901504517
```