Accelerated Natural Language Processing
Assignment 1
Report

s1882930

s1842611

Q1.

Here are the codes:

```
# judge the character belongs to the alphabet
def is_alphabet(char):
    if (char >= 'a' and char <= 'z'
            or
            char >= 'A' and char <= 'Z'
    ):
        return True
    else:
        return False

def preprocess_line(a):
    b = ['##']
#set  '##'as start of a sentence
#in order to get the probability of first character when we use bigram
#and trigram model
    for i in range(len(a)):
        if a[i].isdigit():
#get the digit of the sentence and change it into '0'
            b.append('0')
        if a[i] == " " or a[i] == '.' or is_alphabet(a[i]):
#judge it if is the target characters
            b.append(a[i])
    b.append('#')
#set  '#' as end of a sentence
#in order to calculate the probability of the trigrams
#it will be convenient when computing perplexity
    c = ''.join(b).lower().
return c
```

Q2.

I guess that it uses maximum likelihood estimation and add-one smoothing, when calculating the probabilities of trigrams. Reasons are following.
Firstly, from the distribution of the data given by **model_br.en**, there are 26100 different trigrams in the model. The probabilities range from 1.021e-5 to 0.9986, which means that there is no zero probability. Secondly, given the previous same two history words, the sum probabilities of different trigram is one. Thirdly, the mode(the most frequent probability value) value of all data is 0.0333 and it occurs 12632 times. Besides, 0.0333 is very important because it approximates 1/30, whose denominator is just the type number (the vocabulary size) of each trigram model. According to the mode, I assume that the language model uses the add-one smoothing method to get the probability

of each trigram.

Q3.

We use interpolation to build up the language model with **training.en** data, mixing the probability estimates from all the n-gram estimators, weighing and combining the trigram, bigram, and unigram. Because when there is only a small training set, using the ordinary equation

$$P(w_n|w_{n-2}w_{n-1}) = \frac{counts(w_{n-2}w_{n-1}w_n)}{counts(w_{n-2}w_{n-1})}$$

to get the trigram probability will generate many zero probabilities. Besides, the counts of two previous characters (bigram) may also be zero, which means the probability value is infinite.
Compared with add-one smoothing and add-α smoothing which distribute the same probability to each unknown trigram which doesn't appear in the given corpus, the interpolation method can distribute different probabilities to each unknown trigram according to the bigram model and unigram model.
In order to solve it, we use interpolation method to set up language model.

$$P(w_n|w_{n-2}w_{n-1}) = \lambda_1 P(w_n|w_{n-2}w_{n-1}) + \lambda_2 P(w_n|w_{n-1}) + \lambda_3 P(w_n)$$
$$where\ \lambda_1 + \lambda_2 + \lambda_3 = 1$$
$$0 < \lambda_i < 1$$

Actually, each λ weight is computed by conditioning on the context. But considering that we have one small training set, we make the assumption on that each λ makes the same contribution to the language model, and set $\lambda_1 = \lambda_2 = \lambda_3 = 1/3$.
Codes are following:

```
# get the ngram model and compute the counts of n_grams from the training set
#line_counts is the total number of the whole passage
def ngram_counts(n, file):
    counts = defaultdict(int)
    line_count = 0
    with open(file) as f:
        for line in f:
            line = preprocess_line(line)
            line_count += 1
            if len(line) > n:
                for i in range(3 - n, len(line) - (n - 1), 1):
                    gram = line[i:i + n]
# ngram model counts
                    counts[gram] += 1
return counts, line_count

# calculate the unigram probability
#the input is the counts of unigrams in the training set
def u_p(origin_counts):
    p_model = defaultdict(float)
```

```python
    tokens = 0
    types = 0
    for i in origin_counts:
        tokens += origin_counts[i]
    for i in origin_counts:
        p_model[i] = origin_counts[i] / tokens
    types = len(origin_counts)
    return p_model


#uc is an dictionary contains the the counts of unigrams and probabilities
#origin_b_c is the counts of bigrams in the training set
#return the the whole dictionary containing all bigrams
def b_model_cor(uc, origin_b_c):
    bmc = defaultdict(int)
    for i in uc:
        for j in uc:
            temp = []
            temp.append(i)
            temp.append(j)
            word = ''.join(map(str, temp))
            if (word in origin_b_c.keys()):
                bmc[word] = origin_b_c[word]
            else:
#if the previous characters donnot appear in the passage, it is assigned to 0
                bmc[word] = 0
    del bmc['##']
    return bmc

# get  the counts of trigrams
# tc is the original counts of the trigram in the training set
def t_model_cor(uc, tc):
    tmc = defaultdict(int)
    for i in uc:
        for j in uc:
            for m in uc:
                temp = []
                temp.append(i)
                temp.append(j)
                temp.append(m)
                word = ''.join(map(str, temp))
                if (word in tc.keys()):
                    tmc[word] = tc[word]
                else:
                    tmc[word] = 0
#delete the impossible trigrams
    for i in uc:
        for j in uc:
            for m in uc:
                if (i == '#' and m == '#'):
                    del tmc[i + j + m]
    for i in uc:
        for j in uc:
            for m in uc:
                if (i != '#'):
                    if (j == '#'):
                        del tmc[i + j + m]
    return tmc

#   bcp is a whole counts for bigram dictionary
def b_gram(bcp, uc):
    p_model = defaultdict(float)
    tokens = 0
    types = len(bcp)
    for i in bcp:
        tokens += bcp[i]
    for i in bcp:
        p_model[i] = bcp[i] / uc[i[0]]
    return p_model

#set up the trigram model contain the zero probabilities
#tcp is a dictionary containing the probability of the trigrams
#bcp is a dictionary containing the probability of the bigrams
def t_gram(tcp, bcp):
    p_model = defaultdict(float)
    tokens = 0
    types = len(tcp)
    line_number = 0
    for i in tcp:
        tokens += tcp[i]
    for i in tcp:
        if (i[:2] == '##'):
            line_number = line_number + tcor_c[i]  # calculate the line number
    for i in tcp:
        if (bcp[i[:2]] != 0):
            p_model[i] = tcp[i] / bcp[i[:2]]
        else:
            if (i[:2] == '##'):
```

```
# '##'represents the start of a sentence
                p_model[i] = tcp[i] / line_number
            else:
                p_model[i] = 0
    return p_model

# using the a1,a2,a3 to set up the interpolation
def interplotaion(u, b, t, a1, a2, a3):
    p_model = defaultdict(float)
    for i in t:
        p_model[i] = a1 * t[i] + a2 * b[i[1:]] + a3 * u[i[2:]]
    return p_model
#-------main body-------------------

# generate the basic counts dictionary
u_counts, line_count = ngram_counts(1, "/Users/chenwenjun/PycharmProjects/w2/venv/training.en")
b_counts, line_count = ngram_counts(2, "/Users/chenwenjun/PycharmProjects/w2/venv/training.en")
t_counts, line_count = ngram_counts(3, "/Users/chenwenjun/PycharmProjects/w2/venv/training.en")
#generate the unigram model
ucor_c = defaultdict(int)
ucor_c = u_counts

# generate unigram model
ucor_p = defaultdict(float)
ucor_p = u_p(ucor_c)


# generate bigram model
bcor_c = defaultdict(int)
bcor_c = b_model_cor(u_counts, b_counts)
bcor_p = defaultdict(float)
bcor_p = b_gram(bcor_c, u_counts)

# generate trigram model
tcor_c = defaultdict(int)
tcor_c = t_model_cor(u_counts, t_counts)
tcor_p = defaultdict(float)
tcor_p = t_gram(tcor_c, bcor_c)

# generate interpolation
f1 = defaultdict(float)
f1 = interplotaion(ucor_p, bcor_p, tcor_p,1/3, 1/3, 1/3)
```

Here is the dictionary of **ng** :

'ng#':	0.0027488455108938257
'ng.':	0.014032523221172578
'ng ':	0.428113494670319
'ng0':	0.0015301686809280684
'nga':	0.03763821654873327
'ngb':	0.0034305047502100434
'ngc':	0.008716150714147225
'ngd':	0.011948246928464492
'nge':	0.10665589096829017
'ngf':	0.006691569056128392
'ngg':	0.0065382569120528596,
'ngh':	0.032971380415634334
'ngi':	0.0696990250537054
'ngj':	0.00042612292380275323
'ngk':	0.0013515413946875202,
'ngl':	0.012165034364289826
'ngm':	0.008158747495637563
'ngn':	0.02429830303543762
'ngo':	0.042925607838841084
'ngp':	0.0072139598009435795
'ngq':	0.0003271246687778711,
'ngr':	0.06909447177113728
'ngs':	0.027610229783556043,
'ngt':	0.032780557480811844
'ngu':	0.027213327243800767
'ngv':	0.002881710075615589,
'ngw':	0.004304271957603568,
'ngx':	0.0005552510825308602
'ngy':	0.006884771668779344
'ngz':	0.00009469398306727848

We expect that the sum of all conditional probability value should be one, and the real sum of all probabilities is 0.9986. Because when we pin down the first two characters, the sum of all conditional probabilities should be one. The reason why there is a little differences is that computer uses float variable to store the probability values, and it will lost accuracy.
Overall, the probability values extracted from the language model we build match what we expect.

Q4.

Firstly we create a possible generated letter and save them as a list, which removes the symbol '#', because once generated # means that is the end of the characters generation, so removing the '#' is to ensure the number of characters generated, and then set the initial character '# #'. Next, we set two empty lists to store the different trigram types and corresponding probability, and then extract the last two characters of the currently generated string, searching for the corresponding different type of trigram from our established language model and storing corresponding probabilities in the previously set empty list, and then store the probability value in the form of an array to facilitate subsequent selection by probability. Then, since we remove the case of generating ‘#’, the sum value of the different type trigrams with given the same two history characters is not 1, and the function of the latter **random.choice** cannot be used, so normalization is performed, and each trigram probability value is divided by the sum of the probability values of all such trigrams. This doesn’t affect the generation of the next character, because the probability values of all cases are expanded by the same times, and then we use the **random.choice** function with the previously extracted probability value to generate the corresponding character. Finally, the generated character is connected with the previously generated string, and the loop is repeated until the desired number of characters are generated.

```python
def generate_from_LM(distribution,N):
    y=['r','e','s','u','m','p','t','i','o','n',' ',
        'f','h','d','c','l','a','j','y','0','b','w','k','g','v','.','q','x','z']
    generate=[]
    w='##'
    for j in range(N):
        outcomes=[]
        probs=[]
        for i in y :
            a=[w[-2:]]
            a.append(i)
            a=''.join(a)
            outcomes.append(a)
            probs.append(distribution.get(a))
        probs = np.array(probs)
        summ=np.sum(probs)
        probs=probs/summ
        index = np.random.choice(outcomes, p = probs.ravel())
        generate.append(index)
        w=w+generate[-1][-1:]
    return print(w)
generate_from_LM(f1,300)
generate_from_LM(tr_frombr,300)
```

sequence generated from the model trained from “training.en”:

##fartiew ht ha of epa tontosindonov. tihow prin emowor nne reievere gucyssheflisshi ii inelide ds meral mpetety f f at c daiven tly amif blar exemrttteptisin t do teunthan a encecl this sut inrt.albe wme of mldidider tme icoralt un orolighelheso buracot t iilite po weangomenfipsew ein ahatte t tunt

sequence generated from the model trained from “mode-br.en”:
##bou dont oner.bcwn.auld ou theres.his it there can his is fore some.oh ah bood i samel a like falk.iyuirdalike thers. done.turn thats andy an to you like yought at.ozgvfnight.h0zy dfkhqhcphat.ekaboy.srbwair dons.vqs the this gook to pre thwuoats to thi ch hem.mxize dtie.gcqn.y hing way now way opere

We noticed that the sequence generated more ‘.’ character based on teacher’s model than our own model. And there are more real English words generated in teacher’s model. Because the teacher’s model is established on a much larger corpus than mine which is just based on an article. The teacher’s model has more sentences, so simultaneously the model has more possibility of the character followed by ‘.’. Besides much larger corpus means more precise probability, so based on the teacher’s model we can generate word better.

Q5.

Followings are about the function about how to calculate the perplexity with the model generated from the “training.en”document

```
def perplexity(file, n, model):
    counts = defaultdict(int)
#set up a dictionary which contains the probabilities and the trigrams
    with open(file) as f:
        for line in f:
            line = preprocess_line(line)
            if len(line) > n:
                for i in range(3 - n, len(line) - (n - 1), 1):
                    gram = line[i:i + n]
                    if (model[gram] > 0):
                        counts[gram] = -log(model[gram], 2)
#compute the log and add them together
                    else:
                        counts[gram] = 0
 # ngram model counts
    return  2**(sum(counts.values()) / len(counts)  )
```

When we input our language model and the test set document, it is fast to get the perplexity value.

```
print(perplexity('/Users/chenwenjun/PycharmProjects/w2/venv/test',3,f1))
print(perplexity('/Users/chenwenjun/PycharmProjects/w2/venv/training.es',3,f1))
print(perplexity('/Users/chenwenjun/PycharmProjects/w2/venv/training.de',3,f1))
```

Here are the outputs :
17.933215118487485
26.46348580088351

31.686271894350266

From the output, we can see that the perplexity of English article is the lowest just the same as what we expect, because we train it from the English training set. But if we use other file to calculate the perplexity and get a lower result, it is unreasonable to prove that it belongs to English. Because all the sentences have to be processed by '**preprocess_line**' function, and in this circumstance, many characters have been lost. For example, if we use a paragraph which contains Chinese, English, and Japanese, the '**preprocess_line'** function will filter the Chinese and Japanese. So we can't only use perplexity as evidence to judge the type of language.

Q6.

In the previous program, we assume that unigram, bigram, and trigram have the same contribution to the language model. Under this assumption, we set the parameters all the same. However, if we want to find the better parameters($\lambda_1, \lambda_2, \lambda_3$) to build up a stable and robust models, it is evitable to use other methods to solve this problem.

Based on this question, we try to find the best parameters to set up a better language model with lower perplexity.

At first, we use loop to find better parameters:

```
#min is the lowest perplexity
#min_i,min_j,min_m represent the parameter  λ1,λ2,λ3
min=10000
min_i=min_j=min_m=0
for i in np.arange(0.1,0.9,0.01):
    for j in np.arange(0.1, 0.9, 0.01):
        if(i+j<1):
            m=1-i-j
            f1 = interplotaion(ucor_p, bcor_p, tcor_p, i, j, m)
            per=pe('/Users/chenwenjun/PycharmProjects/w2/venv/validation_set', 3, f1)
            if(per<min):
                min=per
                min_i=i
                min_j=j
                min_m = m
```

This function wants to find the hyperparameters within a defined step size and returns the minimum perplexity value. Besides, the right selection of smaller step sizes boosts the accuracy of the perplexity value returned by the algorithm.

We extract a validation set using 20% the original training set 'training.en'. At the same time, we set the step size 0.01 and let the computer do the loop until find the best parameters. Finally, we can get the parameters: 0.72 for trigram(λ_1), 0.15 for bigram(λ_2), and 0.13 for unigram(λ_3).

Then we use the new parameters to test the other languages and find the differences are more obvious.

```
f2=interplotaion(ucor_p, bcor_p, tcor_p, 0.72, 0.15, 0.13)
print(perplexity('/Users/chenwenjun/PycharmProjects/w2/venv/test', 3, f2))
print(perplexity('/Users/chenwenjun/PycharmProjects/w2/venv/training.es', 3,
 f2))
print(perplexity('/Users/chenwenjun/PycharmProjects/w2/venv/training.de', 3,
 f2))
```

Here are the results:

```
18.888578304183984
34.784594585819754
46.416519514021324
```

We think if we can have some extra data that is different from training data and test data. We can turn this data into development data. We define c'(u, v, w) to represent the parameters that the c' (u, v, w) appears in the development dataset. It is easy to calculate the log-likelihood values on the development dataset. The specific formula is as follows:

$$\begin{aligned} L(\lambda_1, \lambda_2, \lambda_3) &= \sum_{u,v,w} c'(u, v, w) \log q(w|u, v) \\ &= \sum_{u,v,w} c'(u, v, w) \log \left(\lambda_1 \times q_{ML}(w|u, v) + \lambda_2 \times q_{ML}(w|v) + \lambda_3 \times q_{ML}(w)\right) \end{aligned}$$

We choose the value of λ so that the larger the value of $L(\lambda_1, \lambda_2, \lambda_3)$ is, the better outcome we can get. Therefore, the value of λ is as follows:

$$\arg \max_{\lambda_1, \lambda_2, \lambda_3} L(\lambda_1, \lambda_2, \lambda_3)$$