SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Established in collaboration with MIT

01.112 Machine Learning Fall 2017
Design Project

Jonathan Wee 1001458
Michael Lee 1001523
Wang Kunyu 1003219

Design Project

In this design project, we would like to design our sequence labelling model for informal texts using the hidden Markov model (HMM) that we have learned in class. This sequence labelling is done for informal texts that can serve as the first steps towards building a more complex, intelligent sentiment analysis system for social media text.

The idea behind sentiment analysis is to analyze the natural language texts typed, shared and read by users through services such as Twitter and Weibo and analyze such texts to infer the users' sentiment information towards certain targets. Such social texts can be different from standard texts that appear, for example, on news articles. They are often very informal, and can be very noisy. It is very essential to build machine learning systems that can automatically analyze and comprehend the underlying sentiment information associated with such informal texts.

The files have been left in their respective folders, namely *./EN*, *./FR*, *./CN*, *./SG*. With good coding practices in mind, we have created an overview HMM class with multiple methods. This allows us to not repeat ourselves in terms of repeatable codes with same functions. Running our code is simple: simply use # python hmm.py. For help, # python hmm.py -h.

Part 2

Part 2.0

Our code uses a few global variables. Tags stores the known tags from the training data, while words are a list of words that have appeared at least k times in our training dataset.

```
def __init__(self):
    self.tags = []
    self.words = []
    self.b = {}
    self.a = {}
```

In addition, our training dataset consists of t_X and t_Y. These are passed into the model when train() is called.

```
def train(self, tX, tY=[], k=3):
    self.tX = tX
    self.tY = tY

    self.setWords(tX, k)
    self.setTags(tY)
    self.calcB()
    self.calcA()
```

Part 2.1

A function is written to return the estimated the emission parameters from the training set using MLE (maximum likelihood estimation). In our code, we first create a table called b using the function calcB(). calcB() calculates with the following formula:

$$e(word|tag) = \frac{Count(tag \rightarrow word)}{Count(tag)}$$

In addition, words that are not found in our global variable *words* is treated as '#UNK#'. But this is implemented in part 2.2, so the explanation will be there.

```
def e(self, tag, word):
    word = word if word in self.words else '#UNK#'
    return self.b.get(tag, {}).get(word, 0)
```

```
def calcB(self):
    self.b = pd.DataFrame(0.0, columns=self.tags, index=self.words)
    for i in range(len(self.tX)):
        for j in range(len(self.tX[i])):
            word = self.tX[i][j]
            word = word if word in self.words else '#UNK#'
            tag = self.tY[i][j]
            self.b[tag][word] += 1
    for tag in self.tags:
        esum = self.b[tag].sum()
        self.b[tag] = self.b[tag].apply(lambda x: x/esum)
```

Part 2.2

To handle the words that appear in the test set but not in the training set, we created a modified training set that replaces these with a special token #UNK# when they appear less than *k=3* times. The function that does this is setWords(). We first count how many times each word appears in the dataset, then words that appear at least k times are added into our global variable *words*. As such when retrieving emission parameters, if a word is not found in *words*, we simply treat it as '#UNK#'.

```
def setWords(self, x, k):
    counter = {'#UNK#':k}
    to_delete = []
    for line in x:
        for word in line:
            counter[word] = counter.get(word,0) + 1
    for word in counter:
        if counter[word] < k:
            to_delete.append(word)
    for word in to_delete:
        del counter[word]
    self.words =  list(counter.keys())
```

Part 2.3

To implement a simple sentiment system, a function was written that takes in a word as the argument to predict the tag $y^* = \arg\max_y e(x|y)$ associated to word x in sequence. In this case, the tag chosen is based purely on the word given, hence we only need to check the probability of generating the word from all the tags, and pick the tag that gives the highest score. The code is given below:

```
def eval(self, test_text, part=2):
    if part == 2:
        s = ""
        opt_tags = {}

        for tag in self.tags:
            for word in self.words:
                curr = self.e(tag, word)
                if curr > opt_tags.get(word, (0,0))[1]:
                    opt_tags[word] = (tag, curr)

        for x in test_text.split('\n'):
            # end of tweet is indicated by newline
            if x == "":
                s += '\n'
                continue
            # otherwise, predict tag
            opt_tag = opt_tags.get(x, ('O', 0))[0]
            s += x + ' ' + opt_tag + '\n'
```

To evaluate our results, we use the following formulas for precision and recall scores.

The precision score is defined as follows:

$$Precision = \frac{\text{Total number of correctly predicted entities}}{\text{Total number of predicted entities}}$$

The recall score is defined as follows:

$$Recall = \frac{\text{Total number of correctly predicted entities}}{\text{Total number of gold entities}}$$

These are implemented in the eval.py script that was given to us.

Part 2 Results

	EN SET	FR SET	CN SET	SG SET
#Entity in gold data	226	223	362	1382
#Entity in prediction	713	727	1741	2582
#Correct Entity	128	173	80	352
Precision Score	0.1795	0.2380	0.0460	0.1363
Recall Score	0.5664	0.7758	0.2210	0.2547
F Score	0.2726	0.3642	0.0761	0.1776
#Correct Sentiment	40	64	49	183
Precision Score	0.0561	0.0880	0.0281	0.0709
Recall Score	0.1770	0.2870	0.1354	0.1324
F Score	0.0852	0.1347	0.0466	0.0923

Part 3

Part 3.1

A function was written to estimate transition parameters using MLE (maximum likelihood estimation). The calculation is simple: to calculate the probability of transitioning from yp to y, we simply count the number of times $\boldsymbol{yp} \rightarrow \boldsymbol{y}$, and divide by the total number of times $\boldsymbol{yp}$ appeared. We implemented special cases for $\boldsymbol{yp}$ == START and $\boldsymbol{yp}$ == STOP. In $\boldsymbol{yp}$ == START, $\boldsymbol{Count}(\boldsymbol{yp})$ is simply the number of datapoints, and $\boldsymbol{Count}(\boldsymbol{yp})$ gives the number of times $\boldsymbol{y}$ is the first tag. In $\boldsymbol{y}$ == STOP, a similar approach is taken, where $\boldsymbol{Count}(\boldsymbol{yp} \rightarrow \boldsymbol{y})$ = number of times $\boldsymbol{yp}$ is the last tag in each datapoint. The formula is as follows:

$$q(y_i|y_{i-1}) = \frac{Count(y_{i-1}, y_i)}{Count(y_{i-1})}$$

```
def q(self, yp, y):
    count_yp = 0.0
    count_yp_y = 0.0
    if yp == 'START':
        count_yp = len(self.tY)
        for yy in self.tY:
            if yy[0] == y:
                count_yp_y += 1
    elif y == 'STOP':
        for i in range(len(self.tY)):
            for j in range(len(self.tY[i])):
                if self.tY[i][j] == yp:
                    count_yp += 1
                    if j+1 == len(self.tY[i]):
                        count_yp_y += 1
    else:
        for i in range(len(self.tY)):
            for j in range(len(self.tY[i])):
                if self.tY[i][j] == yp:
                    count_yp += 1
                    next_y = self.tY[i][j+1] if j+1 < len(self.tY[i]) else 0
                    if next_y == y:
                        count_yp_y += 1
    return count_yp_y/count_yp
```

Part 3.2

Using the estimated transition and emission parameters, the Viterbi Algorithm is used to perform tag predictions on sentences.

The function can be classified into 3 parts:

1. Initialization
2. Forward step of Algorithm
3. Backtracking step of Algorithm

Initialization:

1. Each word of the list of the input sentence is stored into a data table, with number of columns based on the number of x words, and initialized y tags of rows.

```
elif part == 3:
    # predict y1*,...,yn* = arg max(y1,...,yn) p(x1, . . . , xn, y1, . . . , yn)
    s = ""
    x = []
    for word in test_text.split('\n'):
        if word == "":
            if x:
                y = self.viterbi(tuple(x))
                s += self.getText(x,y) + '\n'
                x = []
        else:
            x.append(word)
```

Forward Step:

We created the function $pii(x, k, v)$ to model the $\pi(k, v)$ function taught in class. The function uses the LRUcache to significantly reduce running timings.

$$\text{Base case: } \pi(x, 0, u) = \begin{cases} 1 \; if \; u = \text{START} \\ 0 \text{ otherwise} \end{cases}$$

$$\text{Recursive case: } \pi(x, k, v) = \max_{u \in T} \{\pi(x, k-1, u) \cdot a_{u,v} \cdot b_v(x_k)\}$$

```
@lru_cache(maxsize=None)
def pii(self, x, k, v):
    if k == -1:
        if v == 'START': return 1
        return 0
    T = self.tags + ['START', 'STOP']
    return max([self.pii(x, k-1,u) * self.a[u][v] * self.e(v, x[k]) for u in T])
```

1. Using a double for loop, value of each tag is generated for each column and row.
2. Unless the k value equals to -1, the $pii(x, k, v)$ function would then perform recursively and return the maximum value for each tag in the current state and store it.

3. Equation used: $\pi(k, v) = \max_{u \in T}\{\pi(k-1, u) \cdot a_{u,v} \cdot b_v(x_k)\}$

```
# generate the table
for i in range(len(x)):
    for tag in self.tags:
        df[i][tag] = self.pii(x, i, tag)
```

Backtracking Step:

1. Backtracking is used to find the most likely sequence of tags y_n^*.
2. For each tag, value of tag is calculated using $y_{n-1}^* = arg\ \max_u\{\pi(n-1, u) \cdot a_{u,y_n^*}\}$
3. This value is then compared to all the other possible values, to give the highest values of each tag y_n^*.

```
# generate the tags by backtracking
opt_y = ['O'] * len(x)
opt_y.append('STOP')
for i in range(len(x))[::-1]:
    # get the tag with the highest score
    prev_score = 0
    for tag in self.tags:
        score = self.a[tag][opt_y[i+1]] * df[i][tag]
        if score > prev_score:
            opt_y[i] = tag
            prev_score = score
return opt_y[:-1]
```

Part 3 Results

	EN SET	FR SET	CN SET	SG SET
#Entity in gold data	226	223	362	1382
#Entity in prediction	162	166	158	723
#Correct Entity	104	112	64	386
Precision Score	0.6420	0.6747	0.4051	0.5339
Recall Score	0.4602	0.5022	0.1768	0.2793
F Score	0.5361	0.5758	0.2462	0.3667
#Correct Sentiment	64	72	47	244
Precision Score	0.3951	0.4337	0.2975	0.3375
Recall Score	0.2832	0.3229	0.1298	0.1766
F Score	0.3299	0.3702	0.1808	0.2318

Part 4

Using the estimated transition and emission parameters, an alternative max-marginal decoding algorithm using the forward-backward algorithm is implemented to determine the tags $\boldsymbol{y_n^*}$.

The function can be classified into 3 parts:

1. Initialization
2. Forward Probability
3. Backward Probability

Main Initialization

1. An empty list of optimal y_n^* is first instantiated.
2. For each word x_j and for each tag y_j^*, score is calculated from the summation of $\alpha_u(j)$ and $\beta_u(j)$, namely $alpha(x,u,j)$ and $beta(x,u,j)$. This would call for the forward probability function and the backward probability function.
3. The best score value from among all the tags would be saved as the optimal y_j^* value.

```
def mmd(self, x):
    opt_y = ['O'] * len(x)
    #checkscore = [0] * len(x)
    for i in range(len(x)):
        opt_score = 0
        for tag in self.tags:
            score = self.alpha(x,tag,i) * self.beta(x,tag,i)
            #checkscore[i] += score
            if score > opt_score:
                opt_score = score
                opt_y[i] = tag
    #print(checkscore) # every element in checkscore should be identical
    #to pass the checkscore test
    return opt_y
```

Forward Probability

1. If index j equals 0, then function $a_{START,u}$ with inputs $START, u$ is called.
2. For the following indexes j, the sum of scores of all paths from 'START' to state (j,u) $\alpha_u(j)$ is returned.
    a. Equation used: $\alpha_u(j+1) = \sum_v \alpha_u(j) \cdot a_{v,u} \cdot b_v(x_j)$

```
@lru_cache(maxsize=None)
def alpha(self, x, u, j):
    if j == 0:
        return self.a['START'][u]
    return sum([self.alpha(x,v,j-1) * self.a[v][u] * self.e(v,x[j-1]) for v in self.tags
```

Backward Probability

1. If index j equals the number of words x, then function $b_{u,STOP}$ with inputs $u, STOP$ is called.
2. For the following indexes j, the sum of scores of all paths from state (j, u) to final state 'STOP' $\beta_u(j)$ is returned.
    a. Equation used: $\beta_u(j) = \sum_v \alpha_{u,v} \cdot b_u(x_j) \cdot \beta_v(j+1)$

```
@lru_cache(maxsize=None)
def beta(self, x, u, j):
    if j == len(x):
        return self.a[u]['STOP'] * self.e(u,x[j-1])
    return sum([self.a[u][v] * self.e(u,x[j]) * self.beta(x,v,j+1) for v in self.tags])
```

In addition, we also implemented a way to check if our function is working as intended. The lines that were commented out calculate checkscore, which is given by $\sum_u \alpha_u(j)\beta_u(j)$. These scores should be constant throughout for each datapoint in x. In our implementation, our code passed the checkscore test.

Part 4 Results

	EN SET	FR SET
#Entity in gold data	226	223
#Entity in prediction	175	173
#Correct Entity	107	113
Precision Score	0.6114	0.6532
Recall Score	0.4735	0.5067
F Score	0.5337	0.5707
#Correct Sentiment	69	73
Precision Score	0.3943	0.4220
Recall Score	0.3053	0.3274
F Score	0.3441	0.3687

Part 5

To develop an improved sentiment analysis system, we have made several adjustments.

Firstly, the syntactic structure in English can help in the prediction of the tags associated to words. For example, prepositions such as *after, in, to, on, and with,* are words that usually appear before the nouns or pronouns and show the relationships between nouns and the other words in the sentence. This implies that they more likely to be 'O' and may signal the appearance of a named entity.

Secondly, we can make use of adjectives that describe the pronoun, like *good, best, worst, wonderful,* to identity the incoming named entity with its corresponding tag.

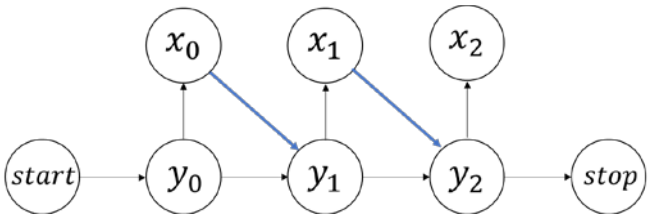
```
count_pxy = count_pxy[['good','best','worst','wonderful','great','was','were','is','it',
                       'are','after','before','in','to','on','up','into','with','or','of',
                       'by','about','like']]
```

```
        for i in range(1, len(x)):
                for j2 in range(len(Y)):
                        for j1 in range(len(Y)):
                                if pi_probability.iloc[j1, i-1] * q(Y[j1], Y[j2]) * e(Y[j2],
                                        pi_probability.iloc[j2, i] = pi_probability.iloc[j1,
                                        pi_path.iloc[j2, i] = Y[j1]
```

	good	best	worst	wonderful	great	was	were	is	it	are	\
O	128	41	5	12	95	424	125	461	250	131	
B-neutral	0	0	0	0	0	0	0	0	0	0	
B-negative	2	0	3	0	1	0	0	0	0	0	
B-positive	28	21	0	3	43	0	0	1	0	0	
I-neutral	0	0	0	0	0	0	0	0	0	0	
I-negative	0	0	0	0	0	0	0	0	0	0	
I-positive	0	0	0	0	0	0	0	0	0	0	

	...	to	on	up	into	with	or	of	by	about	like
O	...	465	144	36	12	128	63	278	59	50	59
B-neutral	...	0	1	0	0	0	0	0	0	0	0
B-negative	...	0	1	0	0	2	0	4	0	2	0
B-positive	...	12	0	0	1	0	0	14	0	1	1
I-neutral	...	0	0	0	0	0	0	1	0	0	0
I-negative	...	1	0	0	0	0	2	4	1	0	0
I-positive	...	1	3	0	0	8	0	13	1	0	0

Using all these prior knowledge, we enable our x_0 observation to directly influence the prediction of the y_1 tag.



Thirdly, we have identified several smoothing techniques used for generative models. There are techniques like Add-One Smoothing, Good-Turing Smoothing, Kneser-Ney Smoothing. Add-One Smoothing is easy to implement, but a large dictionary of words would lead to novel events in test sets too probable. Hence we tried to implement Good-Turing Smoothing, but to implementation complexities, we were unable to achieve the full implementation of the smoothing technique.

In our case, we noticed that data for words that have not appeared at least k times is channeled into data for '#UNK#'. Hence, we decided not to discard this information, instead adding the information for such words into our emission table b. Hence, we will have both information on all words that have appeared in the training dataset as well as a rough gauge on how to handle words that have not been seen before. The implementation is as follows:

```
# -------------- part 5 --------------
def e5(self, tag, word):
    word = word if word in self.words5 else '#UNK#'
    return self.b5.get(tag, {}).get(word, 0)
def calcB5(self):
    self.b5 = pd.DataFrame(0.0, columns=self.tags, index=self.words5)
    for i in range(len(self.tX)):
        for j in range(len(self.tX[i])):
            word = self.tX[i][j]
            tag = self.tY[i][j]
            self.b5[tag][word] += 1
            if word not in self.words:
                self.b5[tag]['#UNK#'] += 1
    for tag in self.tags:
        esum = self.b5[tag].sum()
        self.b5[tag] = self.b5[tag].apply(lambda x: x/esum)
```

The first and second methods have been implemented in p5(only works for EN).py, and works only for the EN datasets. The 3rd method has been implemented under hmm.py, using the switch -p 5.

Part 5 Results

From our implementation, we noticed the Good-Turing smoothing technique we implemented has achieved the best score over all the datasets, including CN, SG, FR datasets. The scores are as follows:

	EN SET	FR SET	CN SET	SG SET
#Entity in gold data	226	223	362	1382
#Entity in prediction	189	188	216	915
#Correct Entity	111	125	93	515
Precision Score	0.5873	0.6649	0.4306	0.5628
Recall Score	0.4912	0.5605	0.2569	0.3726
F Score	0.5349	0.6083	0.3218	0.4484
#Correct Sentiment	76	81	64	320
Precision Score	0.4021	0.4309	0.2963	0.3497
Recall Score	0.3363	0.3632	0.1768	0.2315
F Score	0.3663	0.3942	0.2215	0.2786