01.112 Machine Learning Project Report

Eric Teo 1001526 Keong Jo Hsi 1001685

December 6, 2017

1 Introduction

Please read the code, there are many comments, so it should be understandable which part is doing what assuming you know python. Using debug mode will also make the running of the program easier to view.

Note: For parts 2,3,4, we used python Fractions datatype, which have arbitary accuracy for rational numbers. This will solve any possible underflow issues. The processing time will be significantly slower than normal, but the program runs sufficiently fast.

Our emission parameter and transition parameter notation differs from notes. This should not affect meaning. We use this notation as it is more intuitive and thus easier to see.

Parameter	Notes	Our notation
Emission	$b_i(o)$	$e(o\|i)$
Transition	$a_{i,j}$	$t(j\|i)$

Also, note that python is 0-indexed. Thus, 0 is the first index. We will use $n-1$ to refer to the last index. (n is length of sequence)

1.1 How to run

There is a script for each part (2,3,4,5) of the project.

The syntax for each script is as follows:

```
python part#.py -t trainfile -i infile -o outfile -f folder [-d]
```

Where:

- `#` in `part#.py` is the part number

- `trainfile` is the name of the file used for training (defaults to `train`)
- `infile` is the name of the file with validation data to be annotated (defaults to `dev.in`)
- `outfile` is the name of the output file to write annotated data to (defaults to `dev.p#.out`)
- `folder` is the path of the folder where trainfile, infile and outfile are at. (defaults to `.` the current folder)
  This makes it more convenient to run the script. No need write the full path of files. (i.e. `trainfile`, `infile` and `outfile` are relative paths from folder.)
- `-d` is an option for debug mode. The script will print (large amounts of) debug info while calculating.

Example execution:

```
python part4.py -f "C:\Users\Eric\ML_project\EN" -d
```

Assuming that the EN files are stored in the `C:\Users\Eric\ML_project\EN` folder, This will run the part4 script on the EN files with debug information. The output will be in `C:\Users\Eric\ML_project\EN\dev.p4.out`.

Notes: The script was tested in windows. It should work under linux, but we did not check.

1.2 Results

	EN		FR		SG		CN	
Part	Entity F-score	Sentiment F-score	Entity F-score	Sentiment F-score	Entity F-score	Sentiment F-score	Entity F-score	Sentiment F-score
Part 2	0.2313	0.0995	0.2653	0.0991	0.1990	0.0789	0.0995	0.0310
Part 3	0.5361	0.3299	0.5758	0.3702	0.3667	0.2318	0.2462	0.1808
Part 4	0.5387	0.3441	0.5707	0.3687	0.3639	0.2383	0.2432	0.1706

2 Part 2 - Mixture model

In part 2, each word is treated independently as being generated from a Multinomial mixture model.

2.1 Parameter estimation

The emission parameters $e(x|y)$ are calculated by:

We use t in the following equations to index each sequence in the data: (x_t, y_t). i indexes a single (word, tag) pair in the sequence: $(x_{t,i}, y_{t,i})$. (Note: There is no variable t in the code as it is not necessary.)

$$e(a|u) = \frac{\sum_t \sum_i [[x_{t,i} = a \wedge y_{t,i} = u]]}{\sum_t \sum_i [[y_{t,i} = u]]}$$

We simply iterate through every sentence and every (word, tag) pair in the sentence and count the 2 sums above. The counts are stored in 2 python dicts, the (word, tag) counts are accessed as `counts_e[x][y]`. The word counts are accessed as `counts_y[y]`.

Note that python is 0-indexed. Thus, $i \in \{0, ..., n-1\}$ where n is the number of words in the sentence.

Emission parameters are stored in a nested dict. `e[x][y]` is the emission probability of (tag `y`)→(word `x`).

```
e = {'word1': {'B-negative': emission_prob,
               'B-neutral': emission_prob,
               ...
               'O': emission_prob},
     ...
     '#UNK#': {'B-negative': emission_prob,
               'B-neutral': emission_prob,
               ...
               'O': emission_prob}
    }
```

2.2 Unknown characters

To handle unknown characters, the code simply loops through the `counts_e` dict for every word `x` and checks if the count is lower than the threshold. If it is, the count is accumulated into a count for `#UNK#`. All the accumulated words are then removed from the counts dict (and therefore the emissions dict).

In all decoding parts, when getting the emission parameter for a word, if a word does not exist in the emission parameters dict, the emission parameters for `#UNK#` are used instead.

2.3 Decoding

The `predict` function does the decoding. It treats each word individually and simply finds the maximum probability $e(x|y)$ $\forall y$.

If $x_{t,i} \in$ `e` dictionary:

$$y_{t,i}* = \arg\max_u \left(e(x_{t,i}|u)\right)$$

Else: (treat as `#UNK#`)

$$y_{t,i}* = \arg\max_u \left(e(x_{t,i}|\texttt{\#UNK\#})\right)$$

2.4 Results

We used the `EvalResult.py` script provided to calculate the F-score. Detailed results are below:

	EN	FR	SG	CN
Gold data #Entity	226	223	1382	362
Prediction #Entity	1201	1149	6599	3318
#Correct Entity	165	182	794	183
Entity Precision	0.1374	0.1584	0.1203	0.0552
Entity Recall	0.7301	0.8161	0.5745	0.5055
Entity F	0.2313	0.2653	0.1990	0.0995
#Correct Sentiment	71	68	315	57
Sentiment Precision	0.0591	0.0592	0.0477	0.0172
Sentiment Recall	0.3142	0.3049	0.2279	0.1575
Sentiment F	0.0995	0.0991	0.0789	0.0310

3 Part 3 - Viterbi

Instead of treating each word independently, the input data is first chunked into sentences with `sentence_gen()`. It is a python generator that loops through each line of the input file and uses blank lines to delimit sentences.

We have converted the algorithm in the notes given into code, while accomodating python's language features.

3.1 Parameter estimation

The emission parameters $e(x|y)$ are again calculated by:

(t indexes each sequence in the data: (x_t, y_t). i indexes a single (word, tag) pair in the sequence: $(x_{t,i}, y_{t,i})$. Also, $i \in 0, ..., n-1$ where n is number of words in sentence t.)

$$e(a|u) = \frac{\sum_t \sum_i [[x_{t,i} = a \wedge y_{t,i} = u]]}{\sum_t \sum_i [[y_{t,i} = u]]}$$

The transition parameters $t(v|u)$ are calculated by:

$$t(v|u) = \frac{\sum_t \sum_i [[y_{t,i} = v \wedge y_{t,i-1} = u]]}{\sum_t \sum_i [[y_{t,i-1} = u]]}$$

Also, START and STOP are handled specially.

$$t(v|START) = \frac{\sum_t [[y_{t,0} = v]]}{\sum_t 1}$$

$$t(STOP|u) = \frac{\sum_t [[y_{t,n-1} = u]]}{\sum_t 1}$$

In the program, the calculation of $t(v|u)$ requires storing the value of $y_{t,i-1}$. This is stored in `prev_y` variable. By initializing `prev_y` to START, calculation of $t(v|START)$ is automatically handled.

Transition parameters are stored in a nested dict. `t[y][y_prev]` is the emission probability of (tag `y_prev`)→(tag `y`).

```
t = {'B-negative': {'B-negative': transition_prob,
                    'B-neutral': transition_prob,
                    ...
                    'START': transition_prob},
     ...
    'STOP':       {'B-negative': transition_prob,
                   'B-neutral': transition_prob,
                    ...
                   'START': transition_prob}
    }
```

3.2 Decoding

The Viterbi algorithm first generates a table of $\pi(i, y)$ values, where i is the index in the sentence and y is a tag. Again, note that pyhton is 0-indexed. The meaning of each $\pi(i, y)$ is the (maximum probability of all sequences of length i ending with tag y).

Viterbi Table:

	0	$\cdots$	i	$\cdots$	$n-1$
O	$\pi(0, \text{O})$		$\cdots$		$\pi(n-1, \text{O})$
B-positive	$\pi(0, \text{B-positive})$		$\cdots$		$\pi(n-1, \text{B-positive})$
$\vdots$			$\cdots$		
y	$\cdots$		$\pi(i, y)$		$\cdots$
$\vdots$			$\cdots$		

This table is represented as a list of dict in python, where each column is a dict:

```
viterbi_table = [{'B-negative': max P(Yi='B-negative', Yi-1, ..., Y0, X0, ..., Xi),
                  ...,
                  'O': max P(Yi='O', Yi-1, ..., Y0, X0, ..., Xi)},
                 ...
                ]
```

The first column is calculated as so:

$$\pi(0, v) = t(v|START) \times e(x_{t,0}|v) \quad \forall v \in tags$$

Subsequent columns are calculated as:
For i from 1 to $n-1$:

$$\pi(i, v) = \max_u \left(\pi(i-1, u) \times t(v|u)\right) \times e(x_{t,i}|v) \quad \forall v \in tags$$

Note that $e(x_{t,i}|v)$ in the above equations is the `word_emissions` dict in the following code ($x_{t,i}$ is `word` variable. `e` is a dict of dict of emission probabilities):

```
word_emissions = e.get(word)
if word_emissions is None:
    word_emissions = e["#UNK#"]
```

In the backtracking step, the optimal y^* are predicted in reverse order:
$y_{t,n} = STOP$
For i from $n-1$ to 0:

$$y^*_{t,i} = \arg\max_v \left(\pi(i-1, v) \times t(y^*_{t,i+1}|v)\right)$$

The $y^*_{t,i}$ for each sentence t are rearranged in a python list to be in the correct order.

3.3 Results

Results from `EvalResult.py`:

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

4 Part 4 - Max marginal

Parameter estimations are exactly the same as Part 3. As with the Viterbi algorithm, the algorithm used is from the notes.

Max-marginal decoding uses the forward backward algorithm to generate the marginal probabilities of each tag at a specific index.
(i.e. $P(Y_{t,i} = u|x_t)$ where $Y_{t,i}$ is the distribution for tag $y_{t,i}$. The notation for x_t is abused to mean that the distribution of words result in the sequence x_t)

2 tables are constructed: the α table and the β table.

4.1 Alpha table

First the α table, where $\alpha(i, y) = P(Y_{t,i} = y, x_0, ...x_{i-1})$.
Note that our convention differs in that u is not subscripted but a parameter of alpha. We believe this is clearer when converted into code. This convention does not affect the meaning.

Alpha Table:

word	–	$x_{t,0}$	$\cdots$	$x_{t,i-1}$	$\cdots$	$x_{t,n-2}$
	0	1	$\cdots$	i	$\cdots$	$n-1$
O	$\alpha(0, \text{O})$	$\alpha(1, \text{O})$		$\cdots$		$\alpha(n-1, \text{O})$
B-positive	$\alpha(0, \text{B-positive})$	$\alpha(1, \text{B-positive})$		$\cdots$		$\alpha(n-1, \text{B-positive})$
$\vdots$				$\cdots$		
y		$\cdots$		$\alpha(i, y)$		$\cdots$
$\vdots$				$\cdots$		

This table is represented as a list of dict in python, where each column is a dict:

```
alpha_table = [{'B-negative': P(Yi='B-negative', X0, ..., Xi-1),
                ...,
                'O': P(Yi='O', X0, ..., Xi-1)},
               ...
              ]
```

The first column is calculated as so:

$$\alpha(0, u) = t(u|START) \quad \forall u \in tags$$

Subsequent columns are calculated as:
For i from 1 to $n-1$:

$$\alpha(i, u) = \sum_v (\alpha(i-1, v) \times t(u|v) \times e(x_{t,i-1}|v)) \quad \forall u \in tags$$

Note that the word $x_{t,i-1}$ is offset by -1 from i. This is also shown in the table.

4.2 Beta table

Nest is the β table, where $\beta(i, y) = P(x_i, ..., x_n | Y_{t,i} = y)$

Beta Table:

word	$x_{t,0}$	$\cdots$	$x_{t,i-1}$	$\cdots$	$x_{t,n-1}$
	0	$\cdots$	i	$\cdots$	$n-1$
O	$\beta(0, \text{O})$		$\cdots$		$\beta(n-1, \text{O})$
B-positive	$\beta(0, \text{B-positive})$		$\cdots$		$\beta(n-1, \text{B-positive})$
$\vdots$			$\cdots$		
y	$\cdots$		$\beta(i, y)$		$\cdots$
$\vdots$			$\cdots$		

This table is represented as a list of dict in python, where each column is a dict:

```
beta_table = [{'B-negative': P(Xi, ..., Xn | Yi='B-negative'),
               ...,
               'O': P(Xi, ..., Xn | Yi='O')},
              ...
             ]
```

Unlike viterbi or α, the last $(n-1)$ column is calculated first.

The last column is calculated as so:

$$\beta(n-1, u) = t(STOP|u) \times e(x_{t,n-1}|u) \quad \forall u \in tags$$

Subsequent columns are calculated as:
For i from $n-2$ to 0:

$$\beta(i, u) = \sum_v (\beta(i+1, v) \times t(v|u) \times e(x_{t,i}|u)) \quad \forall u \in tags$$

4.3 Marginals

$$\alpha(i, y) \times \beta(i, y) = P(Y_{t,i} = y, x_0, ..., x_n)$$
$$\sum_y \alpha(i, y) \times \beta(i, y) = P(x_0, ..., x_n)$$

$\sum_y \alpha(i, y) \times \beta(i, y)$ is independent of y. Debug mode checks that this is correct when decoding.

The prediction for each optimal y^* uses the maximum $\alpha(i, y) \times \beta(i, y)$ score. As seen in notes, $P(Y_{t,i} = y|x_0, ..., x_n) \propto P(Y_{t,i} = y, x_0, ..., x_n)$.

Predicting y^*:
For i from 0 to $n-1$:

$$y^*_{t,i} = \arg\max_u (\alpha(i, u) \times \beta(i, u))$$

4.4 Results

Results from `EvalResult.py`:

	EN	FR	SG	CN
Gold data #Entity	226	223	1382	362
Prediction #Entity	175	173	767	189
#Correct Entity	108	113	391	67
Entity Precision	0.6171	0.6532	0.5098	0.3545
Entity Recall	0.4779	0.5067	0.2829	0.1851
Entity F	0.5387	0.5707	0.3639	0.2432
#Correct Sentiment	69	73	256	47
Sentiment Precision	0.3943	0.4220	0.3338	0.2487
Sentiment Recall	0.3053	0.3274	0.1852	0.1298
Sentiment F	0.3441	0.3687	0.2383	0.1706

5 Part 5 - challenge attempt CRF

We initially tried to implement conditional random fields, based on exiting literature from:

- `http://blog.echen.me/2012/01/03/introduction-to-conditional-random-fields/`
- `http://www.cs.columbia.edu/~mcollins/`
- `http://pages.cs.wisc.edu/~jerryzhu/cs838/CRF.pdf`
- `http://homepages.inf.ed.ac.uk/csutton/publications/crf-tutorial.pdf`
- `http://homepages.inf.ed.ac.uk/csutton/publications/crftut-fnt.pdf` (expanded version of above)

However, after implementing and attempting with only HMM features, we encountered numerical overflow from the $e^{\theta \cdot f(y_{i-1}, y_i, \vec{x}, i)}$ calculation which we could not figure out how to handle. The L-BFGS-B solver used for gradient descent then failed to converge.

Thus, we modified the HMM model instead.

6 Part 5 - challenge Modified Max Marginal

Considering that the training and validation data sets are extremely noisy, we decided to use the max marginal algorithm for decoding.

Max marginal is used as it is expected to be less sensitive to noise. While Viterbi attempts to find the maximum probability sequence of tags, max marginal simply finds the maximum probability of a single tag given the sentence. This makes the result less dependent on the entire sequence.

We made 4 modifications to test:

- Lowercase all words via `.casefold()`. (`.casefold()` is simply a more aggressive version of `.lower()`, it should not make a significant different from using `.lower()`)
- Define a new tag "THE". If a word is "the" or similar in the language, and is tagged "O", retag it as "THE". This many named entities have a word "the" before the entity. This is intended to try to account for this feature.
- Define a new tag "PUNCT". If a word is punctuation (non-alphanumeric and length 1), and is tagged "O", retag it as "PUNCT". This is intended to try to account for punctuation as a feature.
- Change the `#UNK#` threshold to a different value.

In order to implement these modifications, the training and prediction data are simply preprocessed to include this changes.

After testing with various combinations, it seems that:

- For the EN dataset, `#UNK#` threshold = 2 and lowercasing gives the best results.
- For the FR dataset, `#UNK#` threshold = 2, lowercasing and the "PUNCT" tag gives the best results.

6.1 Results

Results for the 2 best performing combinations of modifications from `EvalResult.py`:

	EN	FR
Gold data #Entity	226	223
Prediction #Entity	186	180
#Correct Entity	113	119
Entity Precision	0.6075	0.6611
Entity Recall	0.5000	0.5336
Entity F	0.5485	0.5906
#Correct Sentiment	76	84
Sentiment Precision	0.4086	0.4667
Sentiment Recall	0.3363	0.3767
Sentiment F	0.3689	0.4169