ML Project Report

December 5, 2017

```
01.112 ML Project
Done by :

Yin Ji Sheng(1001670)
Hilda Thian(1001776)
Teo Yang Rui(1001518)
```

1 Part 1

1.1 Annotation of raw data sets

1.1.1 Annotation Process

Every student of the course helped annotate 500 tweets/Weibo sentences each.

2 Part 2

2.1 Implementation of simple Sentiment Analysis using only emission parameters:

2.1.1 Loading File

To easily iterate through a desired file, we strip the white space in the string and put separate words with their tags in a whole list.

```
def load_original_train():
    with open("train", encoding="utf-8") as file:
        train_list = file.readlines()
        train_list = [x.strip() for x in train_list]
        return train_list

train_list = load_original_train()
```

2.1.2 Estimating emission parameters

- The method to estimate the emission parameters were as shown in the project description. First, we created an annotation dictionary to store counts of all possible state, {O, I-Positive, I-Negative, I-Neutral, B-Positive, B-Negative, B-Neutral}. Hence, we can call on count(y) anytime when estimating emission parameters without recounting them for every word.

- Then, to estimate emission, we created a function called estimate_emission which take in two variables, word and tag. In this function, the tag will call the annotation dictionary from the first part to yield us the relevant denominator named countBottom.
- For the numerator, a for loop through the loaded train_list above is employed. For each word with their tag, we further split them into a list with 2 elements, [word,tag]. This can only be done if the "word" is not an empty string "", found between different tweets. Creating an if statement to calculate only when we have a word and tag. We loop through the file and increase numerator count, named countTop when we have the specified word and tag. After this is done, the function returns the fraction countTop/countBottom as the estimated_emission parameter.

2.1.3 Replacing words that appear less than k times with #UNK

- To replace words that appear less than k times with #UNK#, the function modify_trainingset(k) was created. Firstly, we need to find a way to store the count of words so that we can know the words that have to be replaced. A dictionary called wordCountDict was created. We loop through the entire training set and counting the occurence of each word.
- Secondly, in order to iterate through the file just once and replace the words that needs to be replaced along the way, a list called wordToBeReplacedList was created. If the key in wordCountDict has a value less than k, it will be stored in wordToBeReplacedList.
- Thirdly, we iterate through train_list, and replace the word found in wordToBeReplacedList with #UNK#., then taking into account the original form of the file, the modified set of words and tags are written into a new file called modified_train

2.1.4 Implementing sentiment analysis system to produce tag

- In order to speed up the sentiment analysis algorithm, we created a giantEmissionDict to store all the possible emission parameters into it. This is done using the store_estimate_emission_fix function. The function is also done using the modified training set so we can count the emission of #UNK# as well.
- In sentiment analysis, we first check whether the word exists in the modified training set or not. If it does not, the word is treated as #UNK#. Then, we loop through all the possible tags and get it's emission with respect to that tag and storing them into scoreDict. The tag with the highest emission value is then picked.
- We then use the same sentiment_analysis function for all the words in the entire dev.in file and output the results to dev.p2.out.

2.1.5 Results for Part 2

CN

```
#Entity in gold data: 362
#Entity in prediction: 3318
```

```
#Correct Entity : 183
Entity  precision: 0.0552
Entity  recall: 0.5055
Entity  F: 0.0995

#Correct Sentiment : 57
Sentiment  precision: 0.0172
Sentiment  recall: 0.1575
Sentiment  F: 0.0310
```

EN

```
#Entity in gold data: 226
#Entity in prediction: 1201

#Correct Entity : 165
Entity  precision: 0.1374
Entity  recall: 0.7301
Entity  F: 0.2313

#Correct Sentiment : 71
Sentiment  precision: 0.0591
Sentiment  recall: 0.3142
Sentiment  F: 0.0995
```

FR

```
#Entity in gold data: 223
#Entity in prediction: 1149

#Correct Entity : 182
Entity  precision: 0.1584
Entity  recall: 0.8161
Entity  F: 0.2653

#Correct Sentiment : 68
Sentiment  precision: 0.0592
Sentiment  recall: 0.3049
Sentiment  F: 0.0991
```

SG

```
#Entity in gold data: 1382
#Entity in prediction: 6599

#Correct Entity : 794
Entity  precision: 0.1203
Entity  recall: 0.5745
Entity  F: 0.1990

#Correct Sentiment : 315
Sentiment  precision: 0.0477
Sentiment  recall: 0.2279
Sentiment  F: 0.0789
```

3 Part 3

3.1 Implementation of Viterbi

3.1.1 Estimating transition parameters using Maximum Likelihood Estimation

- We loop through the modified training set and we created a nested list to store tags of each sentence. We also append "START" and "STOP" to the list to allow easy counting later of the transition parameters.
- We then create a dictionary that contains the count of every possible transition in the modified training set. To calculate the transition parameter, we simply divide that count by the count of the start state, which is available to us from annotationDict that we have created earlier. The result is then stored into giantTransitionDict, with key being the transition.

3.1.2 Before implementing Viterbi

- In order to implement the Viterbi algorithm, we need to first split the whole list of words read by the file into a individual tweets. We use the split_into_sentences function to split the whole list of words dev_in_list into a nested list. We first create a list named sentences that is a collection of lists of individual tweets named each_sentence. each_sentence contains words from the same tweet as its elements. The end of each tweet is recognised by an "" empty string. In iterating through the every word in dev_in_list, we check if the word is the empty string "". If the word is the empty string "", we are at the end of a tweet so we stop appending words into each_sentence and instead, append each_sentence, now a list of words containing a single completed tweet into sentences. We then have to recreate a new empty list each_sentence to accomodate for the next tweet. If the iterated word is not an empty string "", the word still belongs to the same tweet and we continue to append it to each_sentence.

The following pseudocode will split the data set into sentences:

```
def split_into_sentences(dev_in_list):
    sentences = [] #List to store all tweets, tweet by tweet
    each_sentence = [] #List of each tweet, containing words from one tweet
    for every word in each line of the file:
        if word == "": #at end of a tweet
            # append each_sentence into list of all tweets
            # recreate a new empty list each_sentence to contain words for next tweet
        else: #still within the same tweet
            # append word into list containing words from this same tweet
    return sentences
sentences = split_into_sentences(dev_in_list)
```

3.1.3 Implementing Viterbi

- There are then two parts to the Viterbi algorithm. Firstly, we have a Viterbi algorithm to recursively compute and store the scores to each tag in every layer of the sequence. We then find the optimal maximum score and using this optimal score, coupled with the stored scores per layer, we back trace our path to find the optimal parent tags to each layer using the find_backward_path function. Hence, our Viterbi function will run the base case, recursive case and finally, in running the final case, it will incorporate the find_backward_path function.

The pseudo codes explaining the algorithms are as follows:

```
def viterbi(sentence):
    beginning = True # to check for start of sentence
    piList = [] # list to contain dictionaries of scores of tags of every word

    for each word in the tweet:
        if word exists in training set:
            # no changes are made to the word
        else:
            # treat word as "#UNK#"

        # dictionary to store scores of each tag in a given layer of the sequence
        piLayer = {}
        if beginning: # base case
            for tag in all_tags:
                if exists transition probability for 'START' to tag
                AND emission probability for tag to word:
                    # calculate score of tag and store as pi
                    # add pi as a value into the piLayer with the corresponding tag as key
            # append piLayer into piList
            # update beginning to False

        else: # recursive case
            previousLayer = piList[len(piList) - 1] # latest layer of scores in piList
            if word == "": # end of sentence
                tempLayer = {}
                # final case:
                for each previous_tag in the previousLayer:
                    if there exists a transition probability for previous_tag to 'STOP':
                        # calculate new score and store as pi
                        tempLayer[tag] = pi
                if tempLayer is not empty:
                    lastLayer = {}
                    # find the tag that gave the highest score and store it to lastLayer
                    # append lastLayer in piList
                    # feed piList and sentence into find_path_backward()
                    return result from find_path_backward
                else:
```

```
            # tempLayer should not be empty; signifies an error otherwise
                    return "ERROR"

            else: # if not end of sentence
                for tag in all_tags:
                    tempLayer = {}
                    for each previous_tag in the previousLayer:
                        if there exists transition probability for previous_tag to tag
                        AND emission probability for tag to word:
                            # calculate new score and store as pi
                            tempLayer[tag] = pi
                    # if empty, it will never transit from any previous_tag to this tag
                    if tempLayer is not empty:
                        # find the tag that gave the highest score and store it to piLayer
                        # using the tag as key and score as its value
                # add piLayer into piList
```

3.1.4 After running Viterbi

The following pseudocode will derive the best path given the piList derived from the Viterbi algorithm:

```
def find_path_backward(piList, sentence):
    path = []
    for i in range(len(piList)):
        if i == 0:
            # find last layer from piList
            # find the tag that produced max value in dictionary lastLayer
            # add 'end' as first element to path

        elif i == 1:
                # add the tag found previously to path

        # start of recursive algorithm, back tracing to find optimal parent tag
        else:
            # find the current layer at position N-i in piList and store as currentLayer
            # find the tag (position N-i +1) of interest from path and let it be targetedTag
            scoreDict = {}
            for each parent tag in currentLayer:
                if there exists a transition from parent tag to targetedTag:
                    # calculate the score as scoreOfTag
                    # add it to scoreDict with corresponding parent tag as key

            # find the tag with the highestKey
            # add it into path

    singleStringToBeWritten = "" # new empty string
    for every word in the single tweet:
        # attach every word to the best tag from the path
        # note that the path is derived backwards
        # so first word is attached to last tag from path and so on
    return singleStringToBeWritten
```

3.1.5 Results for Part 3

CN

```
#Entity in gold data: 362
#Entity in prediction: 158

#Correct Entity : 30
Entity  precision: 0.1899
Entity  recall: 0.0829
Entity  F: 0.1154

#Correct Sentiment : 22
Sentiment  precision: 0.1392
Sentiment  recall: 0.0608
Sentiment  F: 0.0846
```

EN

```
#Entity in gold data: 226
#Entity in prediction: 155

#Correct Entity : 99
Entity  precision: 0.6387
Entity  recall: 0.4381
Entity  F: 0.5197

#Correct Sentiment : 62
Sentiment  precision: 0.4000
Sentiment  recall: 0.2743
Sentiment  F: 0.3255
```

FR

```
#Entity in gold data: 223
#Entity in prediction: 166

#Correct Entity : 112
Entity  precision: 0.6747
Entity  recall: 0.5022
Entity  F: 0.5758

#Correct Sentiment : 72
Sentiment  precision: 0.4337
Sentiment  recall: 0.3229
Sentiment  F: 0.3702
```

SG

```
#Entity in gold data: 1382
#Entity in prediction: 723

#Correct Entity : 135
Entity  precision: 0.1867
Entity  recall: 0.0977
Entity  F: 0.1283

#Correct Sentiment : 73
Sentiment  precision: 0.1010
Sentiment  recall: 0.0528
Sentiment  F: 0.0694
```

4 Part 4

4.1 Implementation of Max Marginal algorithm

4.1.1 Implementing Max Marginal

- First, we create an empty list called opti_path to store the best sequence of tags as we move along the max_marginal algorithm later on.
- As the max marginal algorithm involves calculating the best expected path from start to state u at the ith sequence, where u is a set comprising of all possible tags, the all_tags list was created so that we can loop through all the tags to determine the best tag with highest Alpha(u,i)*Beta(u,i) for observation i.
- In calculating alpha, we first initialised 2 empty dictionary, forward and alpha_base. Forward is a dictionary initialised so that we can easily extract the alpha values we have calculated by referencing the index of the word in question or index of observation. For word 1, the alpha values will be stored in the forward dictionary as forward={0:{O:alpha value, B-negative alpha value...}, 1:{O:....}...N:{O:...}} . Alpha base is a dictionary use to store the first alpha values, which is just the transition from start to the first tag. Below is a discription of how the code works.

The pseudocode below follows the implementation of max marginal algorithm to do decoding:

```
def max_marginal(sentence):
    opti_path=[]
    # to store all possible alphas to be called during max marginal
    forward={}
    alpha_base={}
    for u in all_tags:
        if start to u transition in giantTransitionDict:
```

```
        alpha start to u= transition start to u
    else:
        alpha start to u = 0
forward[0]=alpha_base

for word in sentence:
    # if word is not in ModifiedwordDict, word=#UNK#
    # dictionary for values of all tags
    tempAlpha={}
    for u in all tags:
        # initialise new alpha
        alpha=0
        # run through all possible transition and emission probabilities to get alpha
        for v in all tags:
            a=0
            b=0
            if transition v to u in giantTransitionDict:
                a=transition v to u
            # do the same for emission v emit word using giantemissiondict
            # call index of word and appropriate tag from forward dictionary
            alpha+= forward[index of word][v]*a*b
        # store all alpha u in that observation.
        tempAlpha[u]=Alpha
    # build forward dictionary full of alphas
    forward[index of observation]= tempAlpha

# same for backwards, where base case starts with lastword
lastWord= sentence[len(sentence)-1]:
# if lastword in modifiedWordDict, Lastword=lastword, else lastword=UNK
# calculate beta_base
for i in range(len(sentence),0,-1):
    # to move from last word to first word sequentially
    # same procedure as forward

# officially doing max marginal
#for each word:
    # iterate through all_tags to calculate alpha(tag)*beta(tag)
    # store each calculated value in a dictionary
    #append the tag with highest probability to opti_path


# attach tag to word
for sentence in sentences:
    #call a function that combines the tags and words together
    sentence_with_tags = combine_path(opti_path, sentence)
return sentence_with_tags
```

4.1.2 After running Max Marginal

We create a function called combine_path that will help us attach the best tag to each word later after the max marginal decoding has been run. The following code is the implementation:

```
def combine_path(best_path, sentence):
    result = ""
    sentence.append("")
    for i in range(len(sentence)):
        if (sentence[i] == ''):
            result += " \n"
        else:
            result += sentence[i] + " " + best_path[i] + "\n"
    return result
```

4.1.3 Results for Part 4

Results for CN and SG are omitted as they are not required as part of the submission.

EN

```
#Entity in gold data: 226
#Entity in prediction: 172

#Correct Entity : 104
Entity  precision: 0.6047
Entity  recall: 0.4602
Entity  F: 0.5226

#Correct Sentiment : 69
Sentiment  precision: 0.4012
Sentiment  recall: 0.3053
Sentiment  F: 0.3467
```

FR

```
#Entity in gold data: 223
#Entity in prediction: 173

#Correct Entity : 113
Entity  precision: 0.6532
Entity  recall: 0.5067
Entity  F: 0.5707

#Correct Sentiment : 73
Sentiment  precision: 0.4220
Sentiment  recall: 0.3274
Sentiment  F: 0.3687
```

5 Part 5

5.1 Implementation of Viterbi with second order dependencies

5.1.1 Before implementing Second Order Viterbi

We had to create a new transition parameter to takes into account the tags two states before. Thus, we had to create a new `giantSecondTransitionDict`. It stores all the possible transition two states before and its value. Below is the pseudocode for the function that was used to create it:

```
def store_second_order_transition():
    startStateCount = {}
    transitionCount = {}
    giantSecondTransitionDict = {}
    for each_sentence_with_only_tags in entire_data_set_with_only_tags:
        for i in range(len(each_sentence_with_only_tags)-1):
            # count the occurence of the first two states
            # store it to startStateCount, e.g START O
        for i in range(len(each_sentence_with_only_tags)-2):
            # count the occurence of each transition, e.g, START O O,
            # store it to transitionCount
    for each_second_order_transition in transitionCount:
        # calculate the transition parameter
        # divide corresponding values of transitionCount by startStateCount
        # store result into giantSecondTransitionDict
    return giantSecondTransitionDict
```

5.1.2 Implementing Second Order Viterbi

The pseudocode below follows the implementation of the second order viterbi using the `giantSecondTransitionDict` created earlier:

```
def second_order_viterbi(sentence):
    for each_word in sentence:
        if each_word not in training set:
            # treat it as #UNK#

        if at first word:
            for each tag:
                # piLayer = transition from START to tags
                # store piLayer into piList
        elif at second word:
            for each tag:
                # piLayer = transition from START and previous_tag to tags
                # store piLayer into piList
        else:
            if word == "":
                for previous_tags in previousLayer:
                    # piLayer = transition from previous_tags to STOP
                    # store piLayer into piList
                # run backtrack_second_order(piList, sentence)
                # in order to recover the path

            else:
                for each tag:
                    for previous_tags in previousLayer:
                        # piLayer = transition from previous_tags to tags
                        # store piLayer into piList
```

5.1.3 Backtracking in Second Order Viterbi

The following pseudocode allows the discovery of the optimal path by backtracking from the optimal score stored at each word:

```
def backtrack_second_order(piList, sentence):
    path = []
    # each key in piList's piLayer:"O STOP": 0.001
    for i in range(len(piList)-1):
        if i == 0:
            lastLayer = piList[len(piList) - (i+1)]
            # pick the highest value, and discard "STOP"
            # tag and pick the tag before "STOP"
            # store that tag into path
        else:
            currentLayer = piList[len(piList) - (i+1)]
            targetTag = path[i-1]
            scoreDict = {}
            for tags in currentLayer:
                # calculate the score
                # score = currentLayer[tags]*giantSecondTransitionDict[tags + targetTag]
                # store the value into scoreDict
            # find the highest score in scoreDict, and pick that tag
            # the second tag in the tag pair will be appended to the path
            # e.g O B-negative, B-negative is appended
            # if detected that the first tag in the tag pair == "START":
                # path.append("START")
    path.remove("START")
    path.remove("STOP")
    result = ""
    for each word in sentence:
        # attach the last element of path to first word of sentence, and so on
        # concatenate result with the attached word tag pair
    return result
```

5.1.4 Result for Part 5

Results for SG and CN are omitted as they are not required for submission.

5.1.5 EN

```
#Entity in gold data: 226
#Entity in prediction: 141

#Correct Entity : 77
Entity  precision: 0.5461
Entity  recall: 0.3407
Entity  F: 0.4196

#Correct Sentiment : 48
Sentiment  precision: 0.3404
Sentiment  recall: 0.2124
Sentiment  F: 0.2616
```

5.1.6 FR

```
#Entity in gold data: 223
#Entity in prediction: 159

#Correct Entity : 35
Entity  precision: 0.2201
Entity  recall: 0.1570
Entity  F: 0.1832

#Correct Sentiment : 19
Sentiment  precision: 0.1195
Sentiment  recall: 0.0852
Sentiment  F: 0.0995
```

5.1.7 Analysis of Accuracy

- Upon evaluation, it is observed that the score produced by Second-order Viterbi as an alternative is lower than the original Viterbi. This reduced score may be attributed to the fact that second-order viterbi is much stricter in its estimation as it now has to depend on an a sequence of two previous consecutive states instead of one. Using second-order Viterbi, there should be 7ˆ3 possible transitions altogether since there are 7 different sentiment tags. However, there are only 56 transitions available from the training data. Given that the training set provided was very small and limited, the efficiency and accuracy of the second-order viterbi is largely restricted. Assuming that the limitations of the dataset outweighs the supposed increased level of accuracy in the second-order viterbi, we believe that the second-order viterbi is still better than the original.