Project 3 Sequence Tagging Report

Hongfei Li - hl963, Shibo Zang sz428

Sequence Tagging Model

Implementation Details

We built an hidden markov model(HMM) from scratch to implement sequence tagging, and used viterbi algorithm for find the best fitting tags for the target sentences.

Training Process

Our HMM is able to handle both bigram language model and trigram language model. It also has a regular expression dictionary `rexDict` that we used to run smoothing on the training set. Details of the smoothing will be discussed in the next section.

To count the number of occurences of label ngrams in the training set, `uniCount biCount triCount` are the dictionaries used to count unigram, bigram, trigram, label occurences respectively.

To calculate the transition probability from a tag/tag pair to a tag, `biTransP` and `triTransP` dictionaries are used to store the transfer probability.

Finally `emissionCount` dictionary is used store the number of occurences a word tag pair.

```Python
uniCount = {}
biCount = {}
triCount = {}

biTransP = {}
triTransP = {}

emissionCount = {}
emissionP = {}

wordCount = {}
```

```
rexDict = {}
wordCategory = {}


def updateCountDict(dic, key):
    if key not in dic:
        dic[key] = 1
    else:
        dic[key] += 1


def hmm():
    with open('train.txt', 'r') as f:
        line_no = 0

        for line in f:
            if line_no == 0:
                wordTokens = line.strip().split()

            if line_no == 2:
                labels = line.strip().split()
                labels.insert(0, '<s>')
                labels.append('<end>')

                for index in range(len(labels) - 1):
                    updateCountDict(uniCount, labels[index])
                    updateCountDict(
                        biCount, labels[index] + ' ' + labels[index + 1])
                updateCountDict(uniCount, '<end>')

                labels.insert(0, '<s>')
                for index in range(len(labels) - 2):
                    updateCountDict(triCount, labels[
                                    index] + ' ' + labels[index + 1] + ' ' +
                                            labels[index + 2])

                labels.pop()
                labels.pop(0)

                for index in range(len(labels)):
                    word = wordTokens[index]
                    if word not in wordCategory:
                        updateCountDict(emissionCount, labels[
                                        index] + ' ' + word)
                    else:
```

```
                        updateCountDict(emissionCount, labels[
                                        index] + ' ' + wordCategory[word])

            line_no = (line_no + 1) % 3

    for key in biCount:
        biTransP[key] = biCount[key] / float(uniCount[key.split()[0]])

    for key in triCount:
        a = key.split()
        firstTwoWord = a[0] + ' ' + a[1]
        triTransP[key] = triCount[key] / float(biCount[firstTwoWord])

    for key in emissionCount:
        emissionP[key] = emissionCount[key] / float(uniCount[key.split()[0]])
```

Smoothing

A set of regular expressios are written to implement smoothing for emission dicitonary. During the training process, all words that occurs less or equal to 1 time are categorized in to one of the following categories. These words' counts in the `emissionCount` are moved and replaced with a tuple of category name as the key, and combined words count as the value.

```
twoDigitNum
fourDigitNum
containsDigitAndAlpha
containsDigitAndDash
containsDigitAndSlash
containsDigitAndComma
containsDigitAndPeriod
othernum
allCaps
capPeriod
initCap
lowercase
others
```

During the testing process, if all tag probability for a certain word in the viterbi table are 0 for a column, same regular expressions are used to categorize the word at that columne and run the viterbi algorithm again to calculate the tag probabilities. Detailed testing process is discuessed in

the next section.

Testing Process

Once training process has ben completed, viterbi algorithm is used for finding the best matching tags for the input words. Bellow is the viterbi that was written for bigram langualge model. The viterbi algorithm that is implemented is identical to one in the textbook, so description of it is omitted. What's worth mentioning is that we included a smoothing method for words that has never been seen with any tag in the training set. The word is categorized into one of the categories above and is seen as one of them. Specifically starting in line 7 and line 41 contains the segment of code that we used to smooth the input.

```Python
Solution = {}

def initializationHelper(smooth, stateNum, states, viterbi, backpointer, wordTokens):
    smoothLater = True
    for s in range(stateNum):
        emissionEntry = states[s] + ' ' + wordTokens[0]
        if smooth:
            if emissionEntry not in hmm.emissionP:
                emissionEntry = states[s] + ' ' + hmm.categorize(wordTokens[0])

        if emissionEntry not in hmm.emissionP:
            viterbi[s][0] = 0
        else:
            transEntry = '<s> ' + states[s]
            if transEntry not in hmm.biTransP:
                viterbi[s][0] = 0
            else:
                viterbi[s][0] = hmm.biTransP['<s> ' + states[s]] * \
                    hmm.emissionP[emissionEntry]
                smoothLater = False
        backpointer[s][0] = 0

    return smoothLater


def iterationHelper(smooth, stateNum, states, viterbi, backpointer, wordTokens, t):
    smoothLater = True
    for s in range(stateNum):
        maxProb = 0
        maxProbIndex = 0
```

```
        for u in range(stateNum):
            transEntry = states[u] + ' ' + states[s]
            prob = 0
            if transEntry in hmm.biTransP:
                prob = viterbi[u][t - 1] * hmm.biTransP[transEntry]
            if prob > maxProb:
                maxProb = prob
                maxProbIndex = u

        emissionEntry = states[s] + ' ' + wordTokens[t]
        if smooth:
            if emissionEntry not in hmm.emissionP:
                emissionEntry = states[s] + ' ' + hmm.categorize(wordTokens[t])

        if emissionEntry not in hmm.emissionP:
            viterbi[s][t] = 0
        else:
            viterbi[s][t] = maxProb * hmm.emissionP[emissionEntry]
            smoothLater = False

        backpointer[s][t] = maxProbIndex

    return smoothLater

def decoding(wordTokens):
    wordLen = len(wordTokens)
    states = []
    for key in hmm.uniCount:
        states.append(key)
    states.remove('<s>')
    states.remove('<end>')
    stateNum = len(states)

    viterbi = [[0 for x in range(wordLen)] for x in range(stateNum + 1)]
    backpointer = [[0 for x in range(wordLen)] for x in range(stateNum + 1)]

    # Initialization step
    smooth = initializationHelper(
        False, stateNum, states, viterbi, backpointer, wordTokens)
    if smooth:
        initializationHelper(True, stateNum, states,
                             viterbi, backpointer, wordTokens)

    # Recursion step
    # 0 column has been taken care of in initialization
```

```
    for t in range(1, wordLen):
    smooth = iterationHelper(
            False, stateNum, states, viterbi, backpointer, wordTokens, t)
        if smooth:
            iterationHelper(True, stateNum, states, viterbi,
                            backpointer, wordTokens, t)
    # Termination step
    maxTProb = 0
    maxTProbIndex = 0
    for s in range(stateNum):
        transEntry = states[s] + ' <end>'
        TProb = 0
        if transEntry in hmm.biTransP:
            TProb = viterbi[s][wordLen - 1] * hmm.biTransP[transEntry]
        if TProb > maxTProb:
            maxTProb = TProb
            maxTProbIndex = s
    viterbi[stateNum][wordLen - 1] = maxTProb
    backpointer[stateNum][wordLen - 1] = maxTProbIndex

    # Backtrace step
    index = backpointer[stateNum][wordLen - 1]

    result = []
    for t in range(wordLen - 1, -1, -1):
        result.insert(0, states[index])
        index = backpointer[index][t]

    with open('result.txt', 'a') as f:
        for r in result:
            f.write(r + ' ')
        f.write('\n')

    return result


def addToSolution(tag, begin, end):
    if tag not in Solution:
        Solution[tag] = []
    Solution[tag].append(str(begin) + '-' + str(end))


def main():
    if os.path.isfile('result.txt'):
```

```
        os.remove('result.txt')

    with open('test.txt', 'r') as f:
        line_no = 0

        for line in f:
            if line_no == 0:
                wordTokens = line.strip().split()
                pred = decoding(wordTokens)

            if line_no == 2:
                indexes = line.split()
                begin = -1
                end = -1
                tag = ''
                for i in range(len(indexes)):
                    if pred[i][0] == 'B':
                        if begin != -1:
                            addToSolution(tag, begin, end)
                        begin = indexes[i]
                        tag = pred[i][2:]
                        end = indexes[i]

                    elif pred[i][0] == 'I':
                        end = indexes[i]

                    else:
                        if begin != -1:
                            addToSolution(tag, begin, end)
                        begin = -1
                        end = -1
                        tag = ''

                    if i == len(indexes) - 1:
                        if begin != -1:
                            addToSolution(tag, begin, end)
                            begin = -1
                            end = -1
                            tag = ''

            line_no = (line_no + 1) % 3


if __name__ == "__main__":
    hmm.smoothing()
```

```
    hmm.hmm()
    main()

    baseline.printSolution(Solution)


    return smoothLater
```

Pre-Proecessing

No pre-processing was needed for our alogrithm. Therefore no pre-processing was implemented.

Experiments and Results

We seperated our training set to a training set and validation set. Bellow is the alogorithm we used to test the validity of our HMM and viberti algorithm.

Python

```
fa = open('result.txt', 'r')
fb = open('validation.txt', 'r')

correct = 0
total = 0

for line in fa:
    fb.readline()
    fb.readline()

    pred = line.strip().split()
    act = fb.readline().strip().split()

    assert len(act) == len(pred)

    for index in range(len(act)):
        if pred[index] == act[index]:
            correct += 1
        total += 1

fa.close()
fb.close()

print correct / float(total)
```

One experiment we have conducted was whether using smoothing will affect the result precision.

We expected a lower score than our 0.77373 from normal algorithm with smoothing. In fact, without smoothing, we only received 0.05249 for running our algorithm with bigram model. It was quite a surpise for us.

Another experiment we conducted was changing the smoothing word count thresh hold. We changed the threshold to 3 instead of 1, and we received 0.75425 on Kaggle which is not too surprising to us because it is reasonable that our smoothing is putting too many words into the same category.

Competition Score

Our team name is AHAHAHAHAH_HMM, and our highest score so far is 0.78536.


Extensions

Explain the extensions that you decided to do. Include the implementation details, the experiments, and the results similar to the previous section. If your extensions contribute to the competition score, analyze why they help. If not, try to explain why it was not useful.

For the extension, we incoorporated trigram language model into our HMM and viterbi, and we were able to get a score of 0.78536 which is a improvement of 0.01163 from our bigram model with smoothing threshold 1. We think the reason is not only because trigram takes in consider of more words and tags but also the implementation of deleted interpolation smoothing to prevent the denominator of trigram probability calculation to be zero.

Experiment and Result for Trigram

Smooth Threshold	Score
1	0.78536
2	0.77445

Below is the detail code of our trigram viterbi algorithm.

```python
Solution = {}
lambda1 = 0
lambda2 = 0
lambda3 = 0


def interpolation(transEntry):
    labels = transEntry.split()
    prob = 0

    if transEntry in hmm.triTransP:
        prob += lambda3 * hmm.triTransP[transEntry]
    if labels[1] + ' ' + labels[2] in hmm.biForTriP:
        prob += lambda2 * hmm.biForTriP[labels[1] + ' ' + labels[2]]
    if labels[2] in hmm.uniTransP:
        prob += lambda1 * hmm.uniTransP[labels[2]]

    assert prob != 0

    return prob
```

This part implements the interpolation method of trigram language model. Lambda1, lambda2, and lambda3 was calculated during the training process. The calculation method is according to the delete interpolation method from textbook.

```
def firstInitializationHelper(smooth, stateNum, states, viterbi, backpointer, wordTokens):
    smoothLater = True
    # First time -- * * s
    for s in range(stateNum):
        emissionEntry = states[s] + ' ' + wordTokens[0]
        if smooth and emissionEntry not in hmm.emissionP:
            emissionEntry = states[s] + ' ' + hmm.categorize(wordTokens[0])

        if emissionEntry not in hmm.emissionP:
            for u in range(stateNum):
                viterbi[u][s][0] = 0
        else:
            transEntry = '<s> <s> ' + states[s]
            prob = interpolation(transEntry)
            prob *= hmm.emissionP[emissionEntry]

            for u in range(stateNum):
                viterbi[u][s][0] = prob
            smoothLater = False

        for u in range(stateNum):
            backpointer[u][s][0] = -1

    return smoothLater
```

The firstInitializationHelper method is used to initialize the viterbi matrix given the first word -- <s> <s> 'A'.

```
def secondInitializationHelper(smooth, stateNum, states, viterbi, backpointer, wordTokens):
    smoothLater = True
    # Second Time  -- * u s
    for s in range(stateNum):
        emissionEntry = states[s] + ' ' + wordTokens[1]
        if smooth and emissionEntry not in hmm.emissionP:
            emissionEntry = states[s] + ' ' + hmm.categorize(wordTokens[1])

        if emissionEntry not in hmm.emissionP:
            for u in range(stateNum):
                viterbi[u][s][1] = 0
        else:
            for u in range(stateNum):
                transEntry = '<s> ' + states[u] + ' ' + states[s]
                prob = interpolation(transEntry)

                viterbi[u][s][1] = viterbi[0][u][0] * \
                    prob * hmm.emissionP[emissionEntry]
                smoothLater = False

        for u in range(stateNum):
            backpointer[u][s][1] = -1

    return smoothLater
```

The secondInitializationHelper method is used to initialize the viterbi matrix given the first word -- <s> 'A' 'B'.

```
def recursionHelper(smooth, stateNum, states, viterbi, backpointer, wordTokens, t):
    smoothLater = True
    # v u s
    for s in range(stateNum):
        emissionEntry = states[s] + ' ' + wordTokens[t]
        if smooth and emissionEntry not in hmm.emissionP:
            emissionEntry = states[s] + ' ' + hmm.categorize(wordTokens[t])

        if emissionEntry not in hmm.emissionP:
            for u in range(stateNum):
                viterbi[u][s][t] = 0
        else:
            for u in range(stateNum):
                maxProb = 0
                maxProbIndex = 0

                for w in range(stateNum):
                    transEntry = states[w] + ' ' + states[u] + ' ' + states[s]
                    prob = interpolation(transEntry)

                    prob *= viterbi[w][u][t - 1]
                    if prob > maxProb:
                        maxProb = prob
                        maxProbIndex = w

                viterbi[u][s][t] = maxProb * hmm.emissionP[emissionEntry]
                backpointer[u][s][t] = maxProbIndex

            smoothLater = False

    return smoothLater
```

The recursioinHelper method is used to initialize the viterbi matrix given the first word -- 'A' 'B' 'C'.

```
def decoding(wordTokens):
    wordLen = len(wordTokens)
    states = []
    for key in hmm.uniCount:
        states.append(key)
    states.remove('<s>')
    states.remove('<end>')
    stateNum = len(states)
```

```
    viterbi = [[[0 for x in range(wordLen)] for x in range(
        stateNum)] for x in range(stateNum)]
    backpointer = [[[0 for x in range(wordLen)] for x in range(
        stateNum)] for x in range(stateNum)]

    # Initialization step
    smooth = firstInitializationHelper(
        False, stateNum, states, viterbi, backpointer, wordTokens)
    if smooth:
        firstInitializationHelper(
            True, stateNum, states, viterbi, backpointer, wordTokens)

    smooth = secondInitializationHelper(
        False, stateNum, states, viterbi, backpointer, wordTokens)
    if smooth:
        secondInitializationHelper(
            True, stateNum, states, viterbi, backpointer, wordTokens)

    # Recursion step
    for t in range(2, wordLen):  # 0 and1 column has been taken care of in initialization
        smooth = recursionHelper(
            False, stateNum, states, viterbi, backpointer, wordTokens, t)
        if smooth:
            recursionHelper(True, stateNum, states, viterbi,
                            backpointer, wordTokens, t)

    # Termination step
    maxTProb = 0
    maxTProbIndexS = 0
    maxTProbIndexU = 0
    for s in range(stateNum):
        for u in range(stateNum):
            transEntry = states[u] + ' ' + states[s] + ' <end>'
            TProb = interpolation(transEntry)

            TProb *= viterbi[u][s][wordLen - 1]
            if TProb > maxTProb:
                maxTProb = TProb
                maxTProbIndexS = s
                maxTProbIndexU = u

    result = []
    result.append(states[maxTProbIndexU])
    result.append(states[maxTProbIndexS])
```

```
    # Backtrace step
    for k in range(wordLen - 3, -1, -1):
        yk = backpointer[maxTProbIndexU][maxTProbIndexS][k + 2]
        maxTProbIndexS = maxTProbIndexU
        maxTProbIndexU = yk
        result.insert(0, states[yk])

    with open('result.txt', 'a') as f:
        for r in result:
            f.write(r + ' ')
        f.write('\n')

    return result


def addToSolution(tag, begin, end):
    if tag not in Solution:
        Solution[tag] = []
    Solution[tag].append(str(begin) + '-' + str(end))


if __name__ == "__main__":
    hmm.smoothing()
    hmm.hmm()
    lambda1, lambda2, lambda3 = hmm.deleted_interpolation()

    baseline.printSolution(Solution)
```

The decoding method was implemented according to viterbi method. We thought carefully about how and to smooth our language model.

Individual Member Contribution

Hongfei Li wrote the baseline algorithm, and wrote the report. Shibo Zang was the main brain of the team. Hongfei and Shibo both contributed to writing the viterbi algorithm and hmm models. Shibo also worked on the extension and figured out how to convert our algorithms to work with trigram.