HW3-CSCI544

October 21, 2021

[22]:
```
#imprting libraries
import pandas as pd
import numpy as np
import json
```

1 Task 1: Vocabulary Creation

1.1 The first task is to create a vocabulary using the training data. In HMM, one important problem when creating the vocabulary is to handle unknown words. One simple solution is to replace rare words whose occurrences are less than a threshold (e.g. 3) with a special token '< unk >'.

1.2 Task. Creating a vocabulary using the training data in the file train and output the vocabulary into a txt file named vocab.txt. The format of the vocabulary file is that each line contains a word type, its index in the vocabulary and its occurrences, separated by the tab symbol '. The first line should be the special token '< unk >' and the following lines should be sorted by its occurrences in descending order. Note that we can only use the training data to create the vocabulary, without touching the development and test data. What is the selected threshold for unknown words replacement? What is the total size of your vocabulary and what is the total occurrences of the special token '< unk >' after replacement?

1.3 Ans: In the cell below we first read the train file then compute the occurence of each unique word in the file and store it under a new column named 'occ', then we replace each word whose frequency is less than 2 with a special token '<unk>'. We also display the selected threshold for unkown word replacement.

[23]:
```
#reading the training file
df = pd.read_csv("./data/train", sep = "\t", names = ['id', 'words', 'pos'])
#finding the number of occurences for each word
df['occ'] = df.groupby('words')["words"].transform('size')
#function to replace each word with <unk> whose occurrence is less than 2
def replace(row):
    if row.occ < 2:
        return "<unk>"
```

```
    else:
        return row.words
#applying the replace function on train data
df['words'] = df.apply(lambda row : replace(row), axis = 1)
print('The Selected Threshold for unkown word replacement is: 2')
```

```
The Selected Threshold for unkown word replacement is: 2
```

1.4 Ans: In cell below we store all the unique words in the train data along with their occurences in df_vocab (in descending order based on their occurences) and then put the special token (<unk>) data row above all others. Then we add index to df_vocab and rearrange it according to requirement and eventually store the vocabulary data in 'vocab.txt'. We also display the total size of our vocabulary and the total occurences of the special token '<unk>'.

[24]:
```
#storing unique words in df_vocab and arranging them in descending order of␣
 ↪their occurrence value
df_vocab = df.words.value_counts().rename_axis('words').reset_index(name =␣
 ↪'occ')
#finding the data row containing word <unk>
df_unk = df_vocab[df_vocab['words'] == "<unk>"]
#removing the data row with word as <unk> and moving it to the top
index = df_vocab[df_vocab.words == "<unk>"].index
df_vocab = df_vocab.drop(index)
df_vocab = pd.concat([df_unk, df_vocab]).reset_index(drop = True)
#storing the index of each vocabulary in df_vocab
df_vocab['id'] = df_vocab.index + 1
#rearranging the vocabulary dataset into the required format
cols = df_vocab.columns.tolist()
cols = [cols[0], cols[-1], cols[1]]
df_vocab = df_vocab[cols]
#storing the vocabulary dataset to 'vocab.txt'
df_vocab.to_csv("vocab.txt", sep="\t", header=None)
print('The total size of our vocabulary is: {}'.format(df_vocab.shape[0]))
print('The total occurences of the special token \'<unk>\': {}'.
 ↪format(int(df_vocab[df_vocab["words"] == "<unk>"].occ)))
```

```
The total size of our vocabulary is: 23183
The total occurences of the special token '<unk>': 20011
```

1.5 In the cell below we store all the unique pos tags of our training data in 'tags' variable and store our training data in a list (called sentences) consisting of lists(called sentence), where each sublist consists of tuples, where each tuple corresponds to one data row in training data

[25]:
```
#storing all the unique tags in training data into 'tags' variable
df_pos = df.pos.value_counts().rename_axis('pos').reset_index(name = 'count')
pos_dict = dict(df_pos.values)
tags = df_pos.pos.tolist()

#storing the training data in a list( called sentences) consisting of␣
 ↪lists(called sentence),
#where each sublist consists of tuples, where each tuple corresponds to one␣
 ↪data row in training data
sentences = []
sentence = []
first = 1
for row in df.itertuples():
    if(row.id == 1 and first == 0):
        sentences.append(sentence)
        sentence = []
    first = 0
    sentence.append((row.words, row.pos))
sentences.append(sentence)
#del(df_pos)
```

2 Task 2: Model Learning

2.1 The second task is to learn an HMM from the training data. Remember that the solution of the emission and transition parameters in HMM are in the following formulation:

2.2 t(s/s') = count(s->s') / count(s)

2.3 e(x/s) = count(s->x)/ count(s)

2.4 where t(.|.) is the transition parameter and e(.|.) is the emission parameter.

2.5 Task. Learning a model using the training data in the file train and output the learned model into a model file in json format, named hmm.json. The model file should contain two dictionaries for the emission and transition parameters, respectively. The first dictionary, named transition, contains items with pairs of (s, s') as key and t(s/s') as value. The second dictionary, named emission, contains items with pairs of (s, x) as key and e(x/s) as value. How many transition and emission parameters in your HMM?

2.6 Ans: In the cell below the get_trans_matrix function computes the transition matrix ,using the formula provided, on the train data. Similarly, the get_emission_matrix function computes the emission matrix ,using the formula provided, on the train data.

[26]:
```
#get_trans_matrix function computes the transition matrix for the given␣
↪sentences
def get_trans_matrix(sentences, tags):
    tr_matrix = np.zeros((len(tags),len(tags)))

    tag_occ = {}
    for tag in range(len(tags)):
        tag_occ[tag] = 0

    for sentence in sentences:
        for i in range(len(sentence)):
            tag_occ[tags.index(sentence[i][1])] += 1
            if i == 0: continue
            tr_matrix[tags.index(sentence[i - 1][1])][tags.
↪index(sentence[i][1])] += 1

    for i in range(tr_matrix.shape[0]):
        for j in range(tr_matrix.shape[1]):
            if(tr_matrix[i][j] == 0) : tr_matrix[i][j] = 1e-10
            else: tr_matrix[i][j] /= tag_occ[i]

    return tr_matrix
```

```
#get_emission_matrix function computes the emission matrix for the given␣
 ↪sentences
def get_emission_matrix(tags, vocab, sentences):
    em_matrix = np.zeros((len(tags), len(vocab)))

    tag_occ = {}
    for tag in range(len(tags)):
        tag_occ[tag] = 0

    for sentence in sentences:
        for word, pos in sentence:
            tag_occ[tags.index(pos)] +=1
            em_matrix[tags.index(pos)][vocab.index(word)] += 1

    for i in range(em_matrix.shape[0]):
        for j in range(em_matrix.shape[1]):
            if(em_matrix[i][j] == 0) : em_matrix[i][j] = 1e-10
            else: em_matrix[i][j] /= tag_occ[i]

    return em_matrix

vocab = df_vocab.words.tolist()
```

2.7 Ans: In the cell below the get_trans_probs function coverts the transition matrix to transition dictionary and the get_emission_probs function coverts the emission matrix to emission dictionary, while the get_inital_prob function calculates the initial transition probability for each tag and the get_all_prob function generates the transition matrix,emission matrix, transition dictionary and the emission dictionary

[30]:
```
#get_trans_probs function coverts the transition matrix to transition dictionary
def get_trans_probs(tags, tr_matrix,prior_prob):
    tags_dict = {}

    for i, tags in enumerate(tags):
        tags_dict[i] = tags

    trans_prob = {}
    for i in range(tr_matrix.shape[0]):
        trans_prob['(' + '<\S>' + ',' + tags_dict[i] + ')'] =␣
 ↪prior_prob[tags_dict[i]]
    for i in range(tr_matrix.shape[0]):
        for j in range(tr_matrix.shape[1]):
            trans_prob['(' + tags_dict[i] + ',' + tags_dict[j] + ')'] =␣
 ↪tr_matrix[i][j]
```

```
    return trans_prob

#get_emission_probs function coverts the emission matrix to emission dictionary
def get_emission_probs(tags, vocab, em_matrix):
    tags_dict = {}

    for i, tags in enumerate(tags):
        tags_dict[i] = tags

    emission_probs = {}

    for i in range(em_matrix.shape[0]):
        for j in range(em_matrix.shape[1]):
            emission_probs['(' + tags_dict[i] + ', ' + vocab[j] + ')'] =␣
 ↪em_matrix[i][j]

    return emission_probs

#get_all_prob function generates the transition matrix,emission matrix,
#transition dictionary and the emission dictionary
def get_all_prob(tags, vocab, sentences,prior_prob):
    tr_matrix = get_trans_matrix(sentences, tags)
    em_matrix = get_emission_matrix(tags, vocab, sentences)

    transition_probability = get_trans_probs(tags, tr_matrix,prior_prob)
    emission_probability = get_emission_probs(tags, vocab, em_matrix)

    return transition_probability, emission_probability

#get_inital_prob function calculates the initial transition probability for␣
 ↪each tag
def get_inital_prob(df, tags):
    tags_start_occ = {}
    total_start_sum = 0
    for tag in tags:
        tags_start_occ[tag] = 0

    for row in df.itertuples():
        if(row[1] == 1):
            tags_start_occ[row[3]]+=1
            total_start_sum += 1

    prior_prob = {}
    for key in tags_start_occ:
        prior_prob[key] = tags_start_occ[key] / total_start_sum
```

```
    return prior_prob

prior_prob = get_inital_prob(df, tags)
trans_prob, em_prob = get_all_prob(tags, vocab, sentences,prior_prob)
```

2.8 Ans: In the cell below we display the Transition and Emission parameters of our HMM model and store the Transition and Emission dictionaries in a json file named 'hmm.json'

[37]:
```
print('The number of Transition Parameters are: {}'.format(len(trans_prob)))
print('The number of Emission Parameters are: {}'.format(len(em_prob)))
#we store the transition and emission dictionaries in a json file named 'hmm.
 ↪json'
with open('hmm.json', 'w') as f:
    json.dump({"transition": trans_prob, "emission": em_prob}, f,␣
 ↪ensure_ascii=False, indent = 4)
```

```
The number of Transition Parameters are: 2070
The number of Emission Parameters are: 1043235
```

3 TASK 3: Greedy Decoding with HMM

3.1 The third task is to implement the greedy decoding algorithm with HMM.

3.2 Task. Implementing the greedy decoding algorithm and evaluate it on the development data. What is the accuracy on the dev data? Predicting the part-of-speech tags of the sentences in the test data and output the predictions in a file named greedy.out, in the same format of training data.

3.3 Ans: In the cell below we read the validation file, store the validation data in a list(called sentences) consisting of lists(called sentence), where each sublist consists of tuples, where each tuple corresponds to one data row in validation data. We also write a greedy_decoding function that computes the state sequence for our HMM Model using the greedy decoding technique. Similarly, we define a measure_acc function that computes the accuracy of our model by comparing the the predicted tag sequence with groundtruth tag sequence. We then use the greedy_decoding function to get the predicted tags for our validation data and then compute and display the accuracy using the measure_acc function.

[32]:
```
#reading the validation file
validation_data = pd.read_csv("./data/dev", sep = '\t', names = ['id', 'words',␣
 ↪'pos'])
validation_data['occ'] = validation_data.groupby('words')['words'].
 ↪transform('size')
```

```
#storing the validation data in a list( called sentences) consisting of␣
 ↪lists(called sentence),
#where each sublist consists of tuples, where each tuple corresponds to one␣
 ↪data row in validation data
valid_sentences = []
sentence = []
first = 1
for row in validation_data.itertuples():
    if(row.id == 1 and first == 0):
        valid_sentences.append(sentence)
        sentence = []
    first = 0
    sentence.append((row.words, row.pos))
valid_sentences.append(sentence)

#greedy_decoding function computes the state sequence for our HMM Model using␣
 ↪the greedy decoding technique
def greedy_decoding(trans_prob, em_prob, prior_prob, valid_sentences, tags):
    sequences = []
    total_score = []
    for sen in valid_sentences:
        prev_tag = None
        seq = []
        score = []
        for i in range(len(sen)):
            best_score = -1
            for j in range(len(tags)):
                state_score = 1
                if i == 0:
                    state_score *= prior_prob[tags[j]]
                else:
                    if str("(" + prev_tag  + "," + tags[j] + ")") in trans_prob:
                        state_score *= trans_prob["(" + prev_tag  + "," +␣
 ↪tags[j] + ")"]

                if str("(" + tags[j] + ", " + sen[i][0] + ")") in em_prob:
                    state_score *= em_prob["(" + tags[j] + ", " + sen[i][0] +␣
 ↪")"]
                else:
                    state_score *= em_prob["(" + tags[j] + ", " + "<unk>" + ")"]

                if(state_score > best_score):
                    best_score = state_score
                    highest_prob_tag = tags[j]

            prev_tag = highest_prob_tag
            seq.append(prev_tag)
```

```
            score.append(best_score)
        sequences.append(seq)
        total_score.append(score)

    return sequences, total_score

sequences, total_score = greedy_decoding(trans_prob, em_prob, prior_prob,␣
 ↪valid_sentences, tags)

#measure_acc function computes the accuracy of our model by comparing the the␣
 ↪predicted tag sequence
#with groundtruth tag sequence
def measure_acc(sequences, valid_sentences):
    count = 0
    corr_tag_count = 0
    for i in range(len(valid_sentences)):
        for j in range(len(valid_sentences[i])):

            if(sequences[i][j] == valid_sentences[i][j][1]):
                corr_tag_count += 1
            count +=1

    acc = corr_tag_count / count
    return acc

print('Accuracy of our Greedy Decoding HMM model on Development data: {}'.
 ↪format(measure_acc(sequences, valid_sentences)))
```

```
Accuracy of our Greedy Decoding HMM model on Development data:
0.9351132293121244
```

3.4 Ans: In the cell below we read the test file, store the test data in a list(called sentences) consisting of lists(called sentence), where each sublist consists of tuples, where each tuple corresponds to one data row in test data and then use the greedy_decoding function to get the predicted tags for our test data. Furthermore we use the output_file function to store the predicted tags and words of the test data ,in the required format, in 'greedy.out' file.

[33]:
```
#reading the test file
test_data = pd.read_csv("./data/test", sep = '\t', names = ['id', 'words'])
test_data['occ'] = test_data.groupby('words')['words'].transform('size')
test_data['words'] = test_data.apply(lambda row : replace(row), axis = 1)

#storing the test data in a list( called sentences) consisting of lists(called␣
 ↪sentence),
#where each sublist consists of tuples, where each tuple corresponds to one␣
 ↪data row in test data
```

```
test_sentences = []
sentence = []
first = 1
for row in test_data.itertuples():
    if(row.id == 1 and first == 0):
        test_sentences.append(sentence)
        sentence = []
    first = 0
    sentence.append(row.words)
test_sentences.append(sentence)

test_sequences, test_score = greedy_decoding(trans_prob, em_prob, prior_prob,␣
↪test_sentences, tags)

#output_file function stores the predicted tag sequence of our HMM model on␣
↪test dataset in an output file,
#in the required format
def output_file(test_inputs, test_outputs, filename):
    res = []
    for i in range(len(test_inputs)):
        s = []
        for j in range(len(test_inputs[i])):
            s.append((str(j+1), test_inputs[i][j], test_outputs[i][j]))
        res.append(s)

    with open(filename + ".out", 'w') as f:
        for ele in res:
            f.write("\n".join([str(item[0]) + "\t" + item[1] + "\t" + item[2]␣
↪for item in ele]))
            f.write("\n\n")

#we use the output_file function to store the predictions of greedy decoding by␣
↪our HMM model
#on the test data in 'greedy.out'
output_file(test_sentences, test_sequences, "greedy")
```

4 Task 4: Viterbi Decoding with HMM

4.1 The fourth task is to implement the viterbi decoding algorithm with HMM.

4.2 Task. Implementing the viterbi decoding algorithm and evaluate it on the development data. What is the accuracy on the dev data? Predicting the part-of-speech tags of the sentences in the test data and output the predictions in a file named viterbi.out, in the same format of training data.

4.3 Ans: In the cell below we define the viterbi_decoding function that computes the probabilty for each word in a sentence having a tag from the group of all tags based on the dynamic programming algorithm called viterbi decoding. Then we use the function to calculate these probabilities for all sentences in the validation data.

[34]:
```
#viterbi_decoding function computes the probabilty for each word in a sentence␣
↪having a tag from the group of all tags
#based on the dynamic programming algorithm called viterbi decoding
def viterbi_decoding(trans_prob, em_prob, prior_prob, sen, tags):

    n = len(tags)
    viterbi_list = []
    cache = {}
    for si in tags:
        if str("(" + si + ", " + sen[0][0] + ")") in em_prob:
            viterbi_list.append(prior_prob[si] * em_prob["(" + si + ", " +␣
↪sen[0][0] + ")"])
        else:
            #viterbi_list.append(1)
            viterbi_list.append(prior_prob[si] * em_prob["(" + si + ", " +␣
↪"<unk>" + ")"])

    for i, l in enumerate(sen):
        word = l[0]
        if i == 0: continue
        temp_list = [None] * n
        for j,tag in enumerate(tags):
            score = -1
            val = 1
            for k, prob in enumerate(viterbi_list):
                if str("(" + tags[k] + "," + tag + ")") in trans_prob and␣
↪str("(" + tag + ", " + word + ")") in em_prob:
                    val = prob * trans_prob["(" + tags[k] + "," + tag + ")"] *␣
↪em_prob["(" + tag + ", " + word + ")"]
                else:
                    # val = 1
                    val = prob * trans_prob["(" + tags[k] + "," + tag + ")"] *␣
↪em_prob["(" + tag + ", " + "<unk>" + ")"]
```

```
                if(score < val):
                    score = val
                    cache[str(i) + ", " + tag] = [tags[k], val]
            temp_list[j] = score
        viterbi_list = [x for x in temp_list]

    return cache, viterbi_list


c = []
v = []
#we apply viterbi decoding function to all sentences in Validation data
for sen in valid_sentences:
    a, b = viterbi_decoding(trans_prob, em_prob, prior_prob, sen, tags)
    c.append(a)
    v.append(b)
```

4.4 Ans: In the cell below we define the function viterbi_backward that finds the best possible tag sequence for each sentence based on the probabilites calculated by the viterbi decoding function, we then apply the viterbi backward function to all sentences in Validation data and print the accuracy of our viterbi decoding HMM model using measure_acc function.

[35]:
```
#viterbi_backward function than finds the best possible tag sequence for each␣
 ↪sentence based on the
#probabilites calculated by the viterbi decoding function
def viterbi_backward(tags, cache, viterbi_list):

    num_states = len(tags)
    n = len(cache) // num_states
    best_sequence = []
    best_sequence_breakdown = []
    x = tags[np.argmax(np.asarray(viterbi_list))]
    best_sequence.append(x)

    for i in range(n, 0, -1):
        val = cache[str(i) + ', ' + x][1]
        x = cache[str(i) + ', ' + x][0]
        best_sequence = [x] + best_sequence
        best_sequence_breakdown =  [val] + best_sequence_breakdown

    return best_sequence, best_sequence_breakdown

#we apply viterbi backward function to all sentences in Validation data
best_seq = []
best_seq_score = []
```

```
for cache, viterbi_list in zip(c, v):

    a, b = viterbi_backward(tags, cache, viterbi_list)
    best_seq.append(a)
    best_seq_score.append(b)

print('Accuracy of our Viterbi Decoding HMM model on Development data: {}'.
 ↪format(measure_acc(best_seq, valid_sentences)))
```

```
Accuracy of our Viterbi Decoding HMM model on Development data:
0.9480905834496994
```

4.5 Ans: In the cell below we apply the viterbi_decoding function followed by the viterbi_backward function for all sentences in the test data. Then, we use the output_file function to store the predicted tags and words of the test data ,in the required format, in 'viterbi.out' file.

[36]:
```
#applying the viterbi decoding algorithm on all sentences in the test data␣
 ↪using the functions viterbi_decoding
#and viterbi_backward
c = []
v = []

for sen in test_sentences:
    a, b = viterbi_decoding(trans_prob, em_prob, prior_prob, sen, tags)
    c.append(a)
    v.append(b)

best_seq = []
best_seq_score = []
for cache, viterbi_list in zip(c, v):

    a, b = viterbi_backward(tags, cache, viterbi_list)
    best_seq.append(a)
    best_seq_score.append(b)

#we use the output_file function to store the predictions of Viterbi decoding␣
 ↪by our HMM model
#on the test data in 'viterbi.out'
output_file(test_sentences, best_seq, 'viterbi')
```

[ ]:
```

```