hw3

October 21, 2021

[1]:
```python
import pandas as pd
import numpy as np
import json
```

0.0.1 Task 1: Vocabulary Creation

Here, we're filtering out words that appears only once in train dataset and replacing with unk tag. Then we're removing unk from middle of vocabulary and concatinating it on top. Rest all vocb is sorted by number of occurances.

[6]:
```python
def filter_by_occ(row):
    if row.occur < 2:
        return "<unk>"
    return row.word

df_train = pd.read_csv("./data/train", sep = "\t", names = ['idx_sent', 'word',␣
 ↪'tag'])
df_train['occur'] = df_train.groupby('word')["word"].transform('size')
df_train["word"] = df_train.apply(lambda row: filter_by_occ(row), axis=1)

vocab = df_train.word.value_counts().rename_axis("unique_words").
 ↪reset_index(name='occur')
df_unk = vocab[vocab['unique_words'] == "<unk>"]
index = vocab[vocab.unique_words == "<unk>"].index
vocab = vocab.drop(index)
vocab = pd.concat([df_unk, vocab]).reset_index(drop = True)
vocab['idx'] = vocab.index + 1
vocab = vocab[["unique_words", "idx", "occur"]]
np.savetxt('./vocab.txt', vocab, fmt = "%s", delimiter="\t")
```

[7]:
```python
print("Threshold for unknown words: ", 1)
print("Total size of Vocabulary: ", vocab.shape[0])
print("Occurances of <unk>: ", int(vocab[vocab.unique_words == "<unk>"].occur))
```

```
Threshold for unknown words:  1
Total size of Vocabulary:  23183
Occurances of <unk>:  20011
```

Creating dictionaries for tags and reformatting sentences in the form of 2D list from the train dataframe. The list inside the list would contain tuples with word and its corresponding tag in the sentence.

[9]:
```
def df_to_list(df_train):
    train = []
    tmp = []
    for row in df_train.values:
        if row[0] == 1:
            if len(tmp) != 0:
                train.append(tmp)
                tmp = []
        tmp.append((row[1], row[2]))

    train.append(tmp)
    return train

sentences_train = df_to_list(df_train)
print(len(sentences_train))

df_tag = df_train.tag.value_counts().rename_axis('tag').reset_index(name =␣
 ↪'count')
tag_dict = dict(df_tag.values)
tags = df_tag.tag.tolist()
print(len(tags))
```

```
38218
45
```

0.0.2 Task 2: HMM Learning

Below two functions are used in creating 2D matrices of transition probability and emission probability repectively. They use the formula mentioned in the question itself.

To improve the results, I have add some minor probability to zero probability cases so that those caes can be considered.

[10]:
```
def get_transition_mat(sentences, tags):
    tag_occur = dict()
    trans_mat = np.zeros((len(tags),len(tags)))

    for tag in range(len(tags)):
        tag_occur[tag] = 0

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

    for i in range(trans_mat.shape[0]):
        for j in range(trans_mat.shape[1]):

            # removing cases that have zero probability and putting minimum␣
↪probability
            if trans_mat[i][j] == 0:
                trans_mat[i][j] = 1e-10
            else:
                trans_mat[i][j] /= tag_occur[i]

    return trans_mat


def get_emission_mat(tags, vocab, sentences):
    tag_occur = dict()
    em_mat = np.zeros((len(tags), len(vocab)))

    for tag in range(len(tags)):
        tag_occur[tag] = 0

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

    for i in range(em_mat.shape[0]):
        for j in range(em_mat.shape[1]):

            # removing cases that have zero probability and putting minimum␣
↪probability
            if em_mat[i][j] == 0:
                em_mat[i][j] = 1e-10
            else:
                em_mat[i][j] /= tag_occur[i]

    return em_mat
```

Creating dictionaries to get transition probabilty from one tag to other and emision probability for a word and tag.

they keys are the comma separated values. For transition, it is tags sepearted by comma between parenthesis. For emission, it is tag and wor separated by comma between parentheis.

[57]:
```
def get_transition_prob(tags, trans_mat):

    tag_dict = dict()
    for i, tag in enumerate(tags):
        tag_dict[i] = tag

    trans_prob = dict()

    for i in range(trans_mat.shape[0]):
        for j in range(trans_mat.shape[1]):
            trans_prob['(' + tag_dict[i] + ',' + tag_dict[j] + ')'] =␣
↪trans_mat[i][j]

    return trans_prob


def get_emission_prob(tags, vocab, em_mat):

    tag_dict = dict()
    for i, tag in enumerate(tags):
        tag_dict[i] = tag

    em_prob = dict()
    for i in range(em_mat.shape[0]):
        for j in range(em_mat.shape[1]):
            em_prob['(' + tag_dict[i] + ',' + vocab[j] + ')'] = em_mat[i][j]

    return em_prob
```

this function is used to calculate initial probability T(s1) for the first word tag.

[58]:
```
def get_inital_prob(df, tags):

    tags_start_occ = dict()
    for tag in tags:
        tags_start_occ[tag] = 0

    total_start_sum = 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
```

Calculating all probabilities and storing them in json file.

[59]:
```
vocab_list = vocab.unique_words.tolist()

init_prob = get_inital_prob(df_train, tags)
trans_mat = get_transition_mat(sentences_train, tags)
em_mat = get_emission_mat(tags, vocab_list, sentences_train)
trans_prob = get_transition_prob(tags, trans_mat)
em_prob = get_emission_prob(tags, vocab_list, em_mat)

with open('hmm.json', 'w') as f:
    json.dump({"transition": trans_prob, "emission": em_prob}, f,␣
 ↪ensure_ascii=False, indent = 4)
```

[60]:
```
print('Transition Parameters: {}'.format(len(trans_prob) + len(init_prob)))
print('Emission Parameters: {}'.format(len(em_prob)))
```

```
Transition Parameters: 2070
Emission Parameters: 1043235
```

0.0.3 Task 3: HMM using Greedy Decoding

[25]:
```
# Preprocessing validation data in the required format
df_valid = pd.read_csv("./data/dev", sep = "\t", names = ['idx_sent', 'word',␣
 ↪'tag'])
df_valid['occur'] = df_valid.groupby('word')["word"].transform('size')

sentences_valid = df_to_list(df_valid)
print(len(sentences_valid))
```

```
5527
```

Following function is used to calculate the accuracy

[26]:
```
def get_accuracy(pred_op, orig_op):
    count = 0
    corr_count = 0
    for i in range(len(orig_op)):
        for j in range(len(orig_op[i])):

            if(pred_op[i][j] == orig_op[i][j][1]):
                corr_count += 1
            count +=1

    return corr_count / count
```

Greedy Decoding fxn: In greedy decoding for every change in states we are calculating the score and storing tag which is giving maximum score and using it as previous state. best_score and state_score are used to keep track of the score.

[28]:
```
def greedy_decoding(trans_prob, em_prob, init_prob, sentences_valid, tags):
    res = []

    for sentence in sentences_valid:

        prev_tag = None
        seq = []
        for i in range(len(sentence)):

            best_score = -1
            for j in range(len(tags)):

                state_score = 1
                if i == 0:
                    state_score *= init_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] + "," + sentence[i][0] + ")") in em_prob:
                    state_score *= em_prob["(" + tags[j] + "," + sentence[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)

        res.append(seq)

    return res

greedy_valid_op = greedy_decoding(trans_prob, em_prob, init_prob,␣
 ↪sentences_valid, tags)
print("Accuracy for greedy decoding for validation dataset: {:.2f}".
 ↪format(get_accuracy(greedy_valid_op, sentences_valid)))
```

```
Accuracy for greedy decoding for validation dataset: 0.94
```

```
[35]: # Generating outfile
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")
```

Preprocessing for test data.

```
[39]: df_test = pd.read_csv("./data/test", sep = "\t", names = ['idx_sent', 'word'])
df_test['occur'] = df_test.groupby('word')["word"].transform('size')

sentences_test = []
sentence = []
first = 1
for row in df_test.itertuples():
    if(row.idx_sent == 1 and first == 0):
        sentences_test.append(sentence)
        sentence = []
    first = 0
    sentence.append(row.word)
sentences_test.append(sentence)
print(len(sentences_test))
```

```
5462
```

Calculating and storing result of greedy ddecoding on test

```
[40]: greedy_test_op = greedy_decoding(trans_prob, em_prob, init_prob,␣
 ↪sentences_test, tags)
output_file(sentences_test, greedy_test_op, "greedy")
```

0.0.4 Task 4: Viterbi Decoding

```
[45]: def viterbi_decoding(trans_prob, em_prob, prior_prob, s, tags):

    n = len(tags)
    viterbi_list = []
    cache = {}
    # cache is a dictionary storing all indides of pos and "pos" as a key and␣
↪value as score or cumulative probability
    # Dictionary will only make update when for any state we find that a␣
↪transition for one tag to another is better than other
    # transition mapping.

    for si in tags:
        if str("(" + si + "," + s[0][0] + ")") in em_prob:
            viterbi_list.append(prior_prob[si] * em_prob["(" + si + "," +␣
↪s[0][0] + ")"])

        else:
            viterbi_list.append(prior_prob[si] * em_prob["(" + si + "," +␣
↪"<unk>" + ")"])

    for i, l in enumerate(s):

        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 = 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
```

[46]:
```
c, v = [], []
for sentence in sentences_valid:
    a, b = viterbi_decoding(trans_prob, em_prob, init_prob, sentence, tags)
    c.append(a)
    v.append(b)
```

[47]:
```
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
```

[49]:
```
viterbi_val_op = []
best_seq_score = []
for cache, viterbi_list in zip(c, v):

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

print("Accuracy for viterbi decoding for validation: {:.2f}".
 ↪format(get_accuracy(viterbi_val_op, sentences_valid)))
```

```
Accuracy for viterbi decoding for validation: 0.95
```

Calculating Tags for Test data via Viterbi and saving them in viterbi.out file

[50]:
```
c, v = [], []

for sentence in sentences_test:
    a, b = viterbi_decoding(trans_prob, em_prob, init_prob, sentence, tags)
    c.append(a)
    v.append(b)

viterbi_test_op= []
for cache, viterbi_list in zip(c, v):
    a, b = viterbi_backward(tags, cache, viterbi_list)
    viterbi_test_op.append(a)

output_file(sentences_test, viterbi_test_op, 'viterbi')
```