hw3 (1)

October 21, 2021

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

TASK 1

[2]:
```
df = pd.read_csv("./data/train", sep = "\t", names = ['id', 'words', 'pos'])
df['occ'] = df.groupby('words')["words"].transform('size')
def replace(row):
    if row.occ <= 1:
        return "<unk>"
    else:
        return row.words
df['words'] = df.apply(lambda row : replace(row), axis = 1)
```

[3]:
```
df_vocab = df.words.value_counts().rename_axis('words').reset_index(name =␣
 ↪'occ')
df_unk = df_vocab[df_vocab['words'] == "<unk>"]
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)
df_vocab['id'] = df_vocab.index + 1
cols = df_vocab.columns.tolist()
cols = [cols[0], cols[-1], cols[1]]
df_vocab = df_vocab[cols]
df_vocab.to_csv("vocab.txt", sep="\t", header=None)
print("Selected threshold for unknown words: ", 1)
print("Total size of the vocabulary: ", df_vocab.shape[0])
print("Total occurrences of the special token <unk>: ",␣
 ↪int(df_vocab[df_vocab["words"] == "<unk>"].occ))
```

```
Selected threshold for unknown words:  1
Total size of the vocabulary:  23183
Total occurrences of the special token <unk>:  20011
```

[4]:
```
df_pos = df.pos.value_counts().rename_axis('pos').reset_index(name = 'count')
pos_dict = dict(df_pos.values)
tags = df_pos.pos.tolist() # Extracting all the distinc tags
```

```
print(len(tags))
"""
    Creating the 2D list of sentences where in each entry we have 1D list of␣
↪sentence and in each sentence we have tuples of word and
    its corresponding pos tag.
"""
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)
```

45

TASK 2 HMM LEARNING

[5]:

```
"""
    get_trans_matrix
        args: sentences (2D list from training dataseet)
              tags: list of distinct tags

        return: Transition matrix (2D numpy array square matrix)
        size: length of tags * length of tags

        row: tag at previous state
        col: tag at current state we want to calulate for

        formula: transition from s -> s' = count(s->s') / count(s)
"""

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
        args: tags (list of distinct pos tags)
              vocab (list of all distinct words in the training dataset)
              sentences (2D list of all the sentence in the training dataset)

        return: emission matrix (2D numpy array)
        size: length of tags * length of vocabualry

        row : tags
        col: vocab

        Formula: emission probability = from pos tag to a word = count(s -> x) /
↪ count(s)
"""

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
```

```
# For any entry in emission matrix or transition matrix if the entry is zero we␣
↪are inserting 1e-10 as the probability.

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

[6]:
```
"""
    Creating the dictionary for transition and emission matrix
    Each cell in either cell states either transition from one
    tag to other tag or going from a tag to a word. These states
    will be the key for their respective dictionaries
"""
def get_trans_probs(tags, tr_matrix):
    tags_dict = {}

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

    trans_prob = {}
    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

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


def get_all_prob(tags, vocab, sentences):
    tr_matrix = get_trans_matrix(sentences, tags)
    em_matrix = get_emission_matrix(tags, vocab, sentences)

    transition_probability = get_trans_probs(tags, tr_matrix)
```

```python
    emission_probability = get_emission_probs(tags, vocab, em_matrix)

    return transition_probability, emission_probability

"""
    Initial Probability
        args: df (dataframe)
              tags (list of pos)

        return: a dictionary of intital probability

        Objective: For calculating T(s1).

        Formula: T(s1) = count(s1 at the begining) / sum of all count(s at the␣
 ↪begining)
"""

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

# Extract all the probaility dictionary
trans_prob, em_prob = get_all_prob(tags, vocab, sentences)
prior_prob = get_inital_prob(df, tags)
print("Total transition parameter in our HMM model: {}".format(len(trans_prob)␣
 ↪+ len(prior_prob)))
print("Total emission paramenter in our HMM model: {}".format(len(em_prob)))
```

```
Total transition parameter in our HMM model: 2070
Total emission paramenter in our HMM model: 1043235
```

[7]:
```python
total_trans_prob = {}
for key in prior_prob:
    total_trans_prob['(' + '<s>' + ',' + key + ')'] = prior_prob[key]
```

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

TASK 3 GREEDY DECODING

[8]:
```
# Similar to training data extraction
validation_data = pd.read_csv("./data/dev", sep = '\t', names = ['id', 'words',␣
 ↪'pos'])
validation_data['occ'] = validation_data.groupby('words')['words'].
 ↪transform('size')
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

        args: trans_prob (dictionary containg transisition probability)
              em_prob (dictionary containing emission probability)
              prior_prob (dictionary containing T(s1) probability)
              valid_sentences (2D list of sentences for testing our HMM model)
              tags (list of distince pos tags)

        return: sequences (2D list of pos tag for each sentence we tested)
                total_score (2D list of score for all tags in the sequence)

        In greedy decoding for every change in states we are calculating the␣
 ↪score
        and storing tag which is giving maximum score an using it as previous␣
 ↪state.

        If a word in testing or validation dataset isn't present in the␣
 ↪training dataset
        then emission will not have an entry. So to handle that case we are␣
 ↪using the
        probablity for unknown words.
"""
def greedy_decoding(trans_prob, em_prob, prior_prob, valid_sentences, tags):
    sequences = []
```

```python
    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)
# To calculate the accuracy
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 for greedy decoding for validation dataset: {:.2f}".
 ↪format(measure_acc(sequences, valid_sentences)))
```

Accuracy for greedy decoding for validation dataset: 0.94

[9]:
```
# Similar extraction for testing dataset as for training dataset
test_data = pd.read_csv("./data/test", sep = '\t', names = ['id', 'words'])
test_data['occ'] = test_data.groupby('words')['words'].transform('size')

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)

# 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")

output_file(test_sentences, test_sequences, "greedy")
```

TASK 4 VITERBI DECODING

[10]:
```
"""
    Viterbi Decoding
```

```
        args: trans_prob (dictionary containg transisition probability)
              em_prob (dictionary containing emission probability)
              prior_prob (dictionary containing T(s1) probability)
              sen (2D list of sentences for testing our HMM model)
              tags (list of distince pos tags)

        return: Viterbi List -> A 1D list storing all the scores for the
                                previous states pos.
                Cache -> 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.
"""

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(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 = 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 = []

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

[11]:

```
"""
    After calculating all the best probalilties and storing it into cache␣
↪dictionary and viterbi list
    We are back propogating to generate the sequence of pos tags.
    Taking argmax of the viterbi list for the last word propgrating it till we␣
↪reach to the first word.

    best sequence will return a 2D list containing sequence of tags for each␣
↪sentences in a test data
"""
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

best_seq = []
```

```python
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 for viterbi decoding for validation dataset: {:.2f}".
 ↪format(measure_acc(best_seq, valid_sentences)))

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)

output_file(test_sentences, best_seq, 'viterbi')
```

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