Task 1 Vocabulary Creation

Q: What is the selected threshold for unknown words replacement?

The selected threshold is 2

Q: What is the total size of your vocabulary

The size of my vocabulary is 23197

Q: what is the total occurrences of the special token < unk > after replacement?

The total occurrences of unk is 20011

Task 2: Model Learning

Q: How many transition and emission parameters in your HMM?

for threshold = 2, there are 1351 pairs of transitions, 30347 pairs of emissions

Task 3: Greedy Decoding with HMM

Q: What is the accuracy on the dev data?

Threshold: 2, Greedy accuracy on Dev Dataset: 94.482%

Task 4: Viterbi Decoding with HMM

Q: What is the accuracy on the dev data?

Threshold: 2, Viterbi accuracy on Dev Dataset: 95.233%

Solution Explanations:

Data preparation
Loaded Train data into Pandas Dataframe and utilized value_counts() to count unique values to generate vocab.
Unknown word handling:
HMM model is the only place in this HW where improvement can be made. The algorithm only "guess" the tag when encounter an unknown word to the HMM model, so I decided to categorize the unknown word.
When processing unknown words, I divide unknown words into several categories, based on observations, each category is correct 70% of the time on average:

- Word contains upper case letters is likely to be NNP
- Word contains digits, comma, dot or colon is likely to be CD
- Word with a dash is likely to be jj
- Word ends with "ed" is likely to be VBN or VBD, VBN was choose for higher count
- The rest of the words are "<unk>"

Occurrence for unknown categories:

	word_type	vocab_idx	occurrence
0	<unk>	0	6521
1	<unk_upper_nnp>	1	6963
2	<unk_nums_cd>	2	2700
3	<unk_jj>	3	2874
4	<unk_vbn>	4	953

I applied each of the above rules one at a time, and here is the comparison:

Unknown Handling	Threshold	Greedy	Viterbi
N/A	2	93.50%	94.77%
contains upper letter = NNP rest = unk	2	93.97%	95.01%
contains upper letter = NNP, contains digit = CD, rest = unk	2	94.22%	95.24%
contains upper letter = NNP, contains digit = CD, contains "-" = JJ rest = unk	2	94.37%	95.24%
contains upper letter = NNP, contains digit = CD, contains "-" = JJ suffix "ed"=VBN rest = unk	2	94.48%	95.34%

Implelementation

When each word is being predicted, I first check if the word is in my vocab, if not, the word is replaced by one of the above unknown categories.
The algorithm was implemented same way as described in slides.
For each word, the possible tags are all the unique tags from the training data, and if the product of transition and emission of the current position of each tag is equal (all zeros) the first tag will be chosen for the current predication, in my case, “NNP” is the first tag in my tags set.

For Viterbi, the data structure I used to store the path to each predication is a dictionary, similar to the one that I used to store the Pi score, except that I kept a list of tags for each possible tag at each position, and update the path that led to the best Pi score accordingly.

The jupyter note book output will be attached below:

HW3

October 20, 2021

[72]:
```
import pandas as pd
import numpy as np
import nltk, re, json
from tqdm import tqdm
from os.path import exists
```

[91]:
```
# Training dataset
df = pd.read_csv("data/train", sep='\t', names=["s_idx", "word_type", "pos"])

# The starting POS tag distribution from training dataset
start_tag_distributions = df[ df['s_idx'] == 1 ]["pos"].
 ↪value_counts(normalize=True)

# All the possible pos tags from training dataset
pos_tags = df['pos'].unique()
```

[3]:
```
# Load vocab list - for re-run the program
if exists('vocab.txt'):
    vocab = pd.read_csv('vocab.txt', sep='\t', names=['word_type', 'vocab_idx',␣
 ↪'occurrence'])
    # a set of unique words in the vocab for unk word assignment
    vocab_list = set(vocab["word_type"].unique().flatten())
```

1 Taks 1 - Generate vocab.txt

[92]:
```
# Use value_count() on word_type column to get the occurrences of word types
unique_words = df['word_type'].value_counts()
unique_words = unique_words.reset_index()
vocab = pd.DataFrame(unique_words)

# Add index column
vocab["vocab_idx"] = vocab.index

# Rename and rearrange columns
vocab.columns = ['word_type', 'occurrence', 'vocab_idx']
vocab = vocab[['word_type', 'vocab_idx', 'occurrence']]
```

```
# set unknown word threshold, and drop word types with low frequency from vocab
threshold = 2
unks = vocab[ vocab['occurrence'] < threshold ]
vocab = vocab[ vocab['occurrence'] >= threshold ]
```

[93]:

```
### Define Regex for capitalization or morphology

# Non digit Regex
none_digit_regex = '^(?:[^0-9]*)$'
# word contains number tend to be CD and JJ
num_cd_regex = '^(?:[0-9.,:]*)$'
# word contains upper case letter tend to be NNP
cap_str_regex = '.*[A-Z].*'
# word contains a "-" are likely to be JJ
jj_regex = '^.*-.*$'
# Word with suffix "ed"
vbn_regex = '^.*ed$'

# unknow strings that do not contain any digit
# unk_strs = unks[ unks['word_type'].str.match(none_digit_regex)==True ]



# UPPER CASE = NNP
unk_upper_nnp = unks[ (unks['word_type'].str.match(cap_str_regex)==True)]
unk_strs = unks[ unks['word_type'].str.match(cap_str_regex)==False ]

# DIGITS = CD
unk_cd = unk_strs[ unk_strs['word_type'].str.match(num_cd_regex)==True ]
unk_strs = unk_strs[ unk_strs['word_type'].str.match(num_cd_regex)==False ]

# CONTAINS "-" = JJ
unks_jj = unk_strs[ unk_strs['word_type'].str.match(jj_regex)==True ]
unk_strs = unk_strs[ unk_strs['word_type'].str.match(jj_regex)==False ]

# SUFFIX "ed" = VBN
unk_vnb = unk_strs[ unk_strs['word_type'].str.match(vbn_regex)==True ]
unk_strs = unk_strs[ unk_strs['word_type'].str.match(vbn_regex)==False ]

# Rest of Unknowns
unks_df = pd.DataFrame([["<unk>", 0, unk_strs.occurrence.sum()]], columns =␣
↪['word_type', 'vocab_idx', 'occurrence'])

# Generate unknown categories
unk_upper_nnp = pd.DataFrame([["<unk_upper_nnp>", 0, unk_upper_nnp.occurrence.
↪sum()]], columns = ['word_type', 'vocab_idx', 'occurrence'])
```

```
unk_cd = pd.DataFrame([["<unk_nums_cd>", 0, unk_cd.occurrence.sum()]], columns␣
 ↪= ['word_type', 'vocab_idx', 'occurrence'])
unks_jj = pd.DataFrame([["<unk_jj>", 0, unks_jj.occurrence.sum()]], columns =␣
 ↪['word_type', 'vocab_idx', 'occurrence'])
unk_vnb = pd.DataFrame([["<unk_vbn>", 0, unk_vnb.occurrence.sum()]], columns =␣
 ↪['word_type', 'vocab_idx', 'occurrence'])

# Append unknown categories to the rest of unknowns
unks_df = unks_df.append(unk_upper_nnp, ignore_index=True)
unks_df = unks_df.append(unk_cd, ignore_index=True)
unks_df = unks_df.append(unks_jj, ignore_index=True)
unks_df = unks_df.append(unk_vnb, ignore_index=True)
```

[94]:
```
# creak vocab DF and store it to a file
vocab = unks_df.append(vocab, ignore_index=True)
vocab.to_csv('vocab.txt', sep='\t', header=False, index=False)



# reindex and update vocab_idx values
vocab["vocab_idx"] = vocab.index

# a set of unique words in the vocab for unk word assignment
vocab_list = set(vocab["word_type"].unique().flatten())
```

[94]:

	word_type	vocab_idx	occurrence
0	<unk>	0	6521
1	<unk_upper_nnp>	1	6963
2	<unk_nums_cd>	2	2700
3	<unk_jj>	3	2874
4	<unk_vbn>	4	953
…	…	…	…
23182	transports	23182	2
23183	employee-health	23183	2
23184	looting	23184	2
23185	diapers	23185	2
23186	precarious	23186	2

[23187 rows x 3 columns]

Threshold: 2, Vocab size: 23197, total unknown words occurrence: 20011

What is the selected threshold for unknown words replacement? The selected threshold is 2

What is the total size of your vocabulary The size of my vocabulary is 23197

what is the total occurrences of the special token $<$ unk $>$ after replacement? The total occurrences of unk is 20011

lowercase: Threshold: 2, Vocab size: 21158, unk occurrence: 17401

2 Taks 2 - Model Learning

[95]:
```
# Check if the input word is in vocab list, if not, categorize it based the␣
↪word format
def checkWord(word):
    # Non digit Regex
    none_digit_regex = '^(?:[^0-9]*)$'
    # word contains number tend to be CD and JJ
    num_cd_regex = '^(?:[0-9.,:]*)$'
    # word contains upper case letter tend to be NNP
    cap_str_regex = '.*[A-Z].*'
    # word contains a "-" are likely to be JJ
    jj_regex = '^.*-.*$'
    # Word with suffix "ed"
    vbn_regex = '^.*ed$'

    if word in vocab_list:
        return word

    if bool(re.match(cap_str_regex, word)):
        return '<unk_upper_nnp>'
    elif bool(re.match(num_cd_regex, word)):
        return '<unk_nums_cd>'
    elif bool(re.match(jj_regex, word)):
        return '<unk_jj>'
    elif bool(re.match(vbn_regex, word)):
        return '<unk_vbn>'
    else:
        return '<unk>'
```

[96]:
```
# for calculating count(tag)
pos_distributions = df['pos'].value_counts().to_dict()

# keep track of the transitions and emissions
transitions = {}
emissions = {}

# keep track of the keys in transitions and emissions
e_keys = set()
t_keys = set()

prev_pos = None
```

```python
print("Outputing hmm.json...")

for i, row in tqdm(df.iterrows(), total=df.shape[0]):
    cur_word = row['word_type']
    cur_pos = row['pos']
    # replace low frequent word with <unk>
#     if cur_word not in vocab_list:
#         cur_word = "<unk>"
    cur_word = checkWord(cur_word)

    e_key = cur_pos+","+cur_word
    if e_key in e_keys:
        emissions[e_key]+=(1/pos_distributions[cur_pos])
    else:
        emissions[e_key]=(1/pos_distributions[cur_pos])
        e_keys.add(e_key)
    # skip transition for the first word in a sentence
    if row['s_idx'] != 1:
        t_key = prev_pos+","+cur_pos
        if t_key in t_keys:
            transitions[t_key]+=(1/pos_distributions[prev_pos])
        else:
            transitions[t_key]=(1/pos_distributions[prev_pos])
            t_keys.add(t_key)
    prev_pos = cur_pos


print("Transitions: {}, Emissions:{}".format(len(transitions), len(emissions)))

# put transitions and emissions into hmm_model
hmm_model = {"transition": transitions, "emission" : emissions}
# store hmm into a file

with open('hmm.json', 'w') as fp:
    json.dump(hmm_model, fp)
```

```
Outputing hmm.json…

100%|      | 912095/912095 [00:41<00:00, 22021.43it/s]

Transitions: 1351, Emissions:30347
```

How many transition and emission parameters in your HMM? for threshold = 2, there are 1351 pairs of transitions, 30347 pairs of emissions

3 Task 3: Greedy Decoding withHMM

[97]:
```
if exists('hmm.json'):
    hmm_model = json.load(open('hmm.json',))
```

[98]:
```
# greedy decoding function, return accuracy and output_array according to␣
↪parameters
# dataset: entire dev dataset, output: boolean, is_dev: boolean
def greedy(dataset, output, is_dev):
    # for accuracy computation
    total, correct = 0, 0

    # keep track of prev state
    prev_tag = None

    # output file
    output_array = []

    # Iterate over dataset, print progress
    for i, row in tqdm(dataset.iterrows(), total=dataset.shape[0]):
        word = row['word_type']
        if output:
            output_row = [row['s_idx'], word]

        if is_dev:
            target = row['pos']

        # Check if word is in vocab, if not categorize unks
        word = checkWord(word)

        pred = None
        max_s = -1

        # For starting word:
        if row["s_idx"] == 1:
            # for each possible starting tag
            for tag in start_tag_distributions.keys():
                t,e = 0,0
                # t(s_j)
                t = start_tag_distributions[tag]
                # e(x|s) = 0 if the emission is not seen in training data
                if tag+','+word in hmm_model['emission']:
                    e = hmm_model['emission'][tag+','+word]
                s = e*t
                # argmax s and update predication
                if s > max_s:
                    max_s = s
```

```python
                    pred = tag
        # For the rest of the sentence
        else:
            # for each possible tag
            for tag in pos_tags:
                t,e = 0,0

                # t(s_j|s_j-1) = 0 if the transition is not seen in training␣
↪data
                if prev_tag+','+tag in hmm_model['transition']:
                    t = hmm_model['transition'][prev_tag+','+tag]
                    # e(x|s) = 0 if the emission is not seen in training data
                    if tag+','+word in hmm_model['emission']:
                        e = hmm_model['emission'][tag+','+word]
                s = e*t
                # argmax s and update predication
                if s > max_s:
                    max_s = s
                    pred = tag
        if(output):
            output_row.append(pred)
            # Add empty row between sentences
            if row["s_idx"] == 1:
                output_array.append([None, None, None])
            output_array.append(output_row)

        # remember the current predication as the prev_tag for next observation
        prev_tag = pred
        # increment correct if predicted right
        if is_dev:
            if pred == target:
                correct +=1
        total +=1

    # output , remove the first Nan
    return round(correct/total*100, 3), output_array
```

[99]:
```python
# Dev dataset evaluation
dev_df = pd.read_csv("data/dev", sep='\t', names=["s_idx", "word_type", "pos"])

is_output, is_dev = False, True

print("[Greedy] Testing on Dev data...")

accuracy, output = greedy(dev_df, is_output, is_dev)

print("Dev Dataset Accuracy: {}%".format(accuracy))
```

```
[Greedy] Testing on Dev data…

100%|      | 131768/131768 [00:11<00:00, 11362.14it/s]

Dev Dataset Accuracy: 94.482%
```

[100]:
```
# Test dataset predication
test_df = pd.read_csv("data/test", sep='\t', names=["s_idx", "word_type"])

is_output, is_dev = True, False

print("[Greedy] Generating predications on Test data  greedy.out...")
accuracy, output = greedy(test_df, is_output, is_dev)

# Remove 1st empty line from output, due to the way output was generated
Predictions = np.array(output[1:])

# Store predications to greedy.out
greedy_output_df = pd.DataFrame(Predictions, columns = ["s_idx", "word_type",␣
↪"pos"])
greedy_output_df.to_csv('greedy.out', sep='\t', header=False, index=False)
```

```
[Greedy] Generating predications on Test data  greedy.out…

100%|      | 129654/129654 [00:12<00:00, 10348.60it/s]
```

What is the accuracy on the dev data? Threshold: 2, Greedy accuracy on Dev Dataset: 94.482%

4 Task 4: Viterbi Decoding withHMM

[105]:
```
# Viterbi Algorithm Implementation, take dataset contains one sentences from␣
↪pos 1 to pos m
# dataset: df, output: boolean, is_dev: boolean
def viterbi(dataset):
    # pi table
    pi = {}
    # path table
    paths = {}
    # best tag for each position
    best = None
    # position
    j = 0

    # Iterate over dataset, print progress
#     for i, row in tqdm(dataset.iterrows(), total=dataset.shape[0]):
    for i, row in dataset.iterrows():
```

```
        j = row['s_idx']
        # the <unk> tag was already replaced in input dataset
        word = row['word_type']
        # Keep track of the pi value for each pos tag
        # {"DT" : 0.001  }
        pi_pos = {}
        # keep track of the path for each pos in pi_pos
        pi_path = {}

        if j == 1:
            # reset pi for each sentence
            pi = {}
            paths = {}

            for tag in pos_tags:
                # initialize transition and emission
                t,e = 0,0
                # start word t = start tag distribution
                if tag in start_tag_distributions.keys():
                    t = start_tag_distributions[tag]
                    if tag+','+word in hmm_model['emission']:
                        e = hmm_model['emission'][tag+','+word]
                # record pi for each starting tag
                pi_pos[tag] = e*t
                # start the path for each starting tag
                pi_path[tag] = [tag]
            # record the pi and paths at each position j
            pi[j] = pi_pos
            paths[j] = pi_path

        else:
            for cur_tag in pos_tags:
                pi_pos[cur_tag] = -1

                # pi[j-1][prev_tag], t(cur_tag|prev_tag), e(word|cur_tag)
                for prev_tag in pos_tags:
                    prev_pi = pi[j-1][prev_tag]
                    # Skip to the next prev_tag if pi[j-1, prev_tag] = 0
                    cur_pi = 0
                    # cur_pi = 0 if pre_pi = 0 so only consider the non zero␣
↪prev_pi
                    if prev_pi != 0:
                        t,e = 0,0
                        # if transition exists
                        if prev_tag+','+cur_tag in hmm_model['transition']:
                            t =␣
↪hmm_model['transition'][prev_tag+','+cur_tag]
```

```
                                # if emission exists
                                if cur_tag+','+word in hmm_model['emission']:
                                    e = hmm_model['emission'][cur_tag+','+word]

                        cur_pi = prev_pi*e*t

                    # update pi at each position as well as update the path to␣
 ↪get to the best pi
                    if cur_pi > pi_pos[cur_tag]:
                        pi_pos[cur_tag] = cur_pi
                        pi_path[cur_tag] = paths[j-1][prev_tag][:]
                        pi_path[cur_tag].append(cur_tag)

            # record the pi and paths at each position j
            pi[j] = pi_pos
            paths[j] = pi_path

    # with j being the last position, get the best tag with highest pi value
    best = max(pi[j], key=pi[j].get)
    # return the sequence that led to the best tag
#     print(pi)
#     print(paths)
    return paths[j][best]
```

[106]:
```
def viterbi_decoder(dataset, is_output, is_dev):

    # Represent each sentence key = word, value = pos, e.g. {"word", 1}
    sentence = []
    targets = []

    # Compute accuracy
    correct = 0
    total = 0

    # Output
    output = []

    for i, row in tqdm(dataset.iterrows(), total=dataset.shape[0]):
        # increment total
        total += 1
        # Get word and sentence_idx of each row
        j = row['s_idx']
        if is_dev:
            target = row['pos']
        word = row['word_type']

        #if word not in vocab_list:
```

```
        word = checkWord(word)

        # hold sentence from each row
        w_row = [j, word]

        if j == 1:
            # Process previous complete sentence stored in dict sentence
            if sentence:
                sen_df = pd.DataFrame(sentence, columns =['s_idx', 'word_type'])
                #print(sen_df)
                # get predictions from viterbi
                preds = np.array(viterbi(sen_df))
                # compare preds and target, increment correct
                if is_dev:
                    targets = np.array(targets)
                    correct += np.sum(preds == targets)
                # Generate output
                for i in range(len(preds)):
                    o_row = sentence[i][:]
                    o_row.append(preds[i])
                    output.append(o_row)
                output.append([None,None,None])
                # empty sentence and targets for the next sentence
                sentence = []
                targets = []
        sentence.append(w_row)
        if is_dev:
            targets.append(target)

    # Process the last complete sentece from input df
    if sentence:
        sen_df = pd.DataFrame(sentence, columns =['s_idx', 'word_type'])
        #print(sen_df)
        # get predictions from viterbi
        preds = np.array(viterbi(sen_df))
        # compare preds and target, increment correct
        if is_dev:
            targets = np.array(targets)
            correct += np.sum(preds == targets)
        # Generate output
        for i in range(len(preds)):
            o_row = sentence[i][:]
            o_row.append(preds[i])
            output.append(o_row)
#          output.append([None,None,None])
    accuracy = round(correct/total*100, 3)
```

```
    return accuracy, output
```

[107]:
```
dev_df = pd.read_csv("data/dev", sep='\t', names=["s_idx", "word_type", "pos"])
print("[Viterbi] Testing on Dev data...")
accuracy, output = viterbi_decoder(dev_df, False, True)
print("Dev Dataset Accuracy: {}%".format(accuracy))
```

```
[Viterbi] Testing on Dev data…

100%|          | 131768/131768 [01:41<00:00, 1297.83it/s]

Dev Dataset Accuracy: 95.338%
```

What is the accuracy on the dev data? Threshold: 2, Viterbi accuracy on Dev Dataset: 95.233%

[108]:
```
# Output a viterbi.out on test dataset
test_df = pd.read_csv("data/test", sep='\t', names=["s_idx", "word_type",␣
↪"pos"])
print("[Viterbi] Generating predications on Test data viterbi.out...")
# Accuracy is 0 here, output contains predications for each sentence, separated␣
↪by rows with None values
accuracy, output = viterbi_decoder(test_df, True, False)

# Store predications to greedy.out
Predictions = np.array(output)
greedy_output_df = pd.DataFrame(Predictions, columns = ["s_idx", "word_type",␣
↪"pos"])
greedy_output_df.to_csv('viterbi.out', sep='\t', header=False, index=False)
```

```
[Viterbi] Generating predications on Test data viterbi.out…

100%|          | 129654/129654 [01:38<00:00, 1318.07it/s]
```