UDACITY

Logout

‹ Return to Classroom

Part of Speech Tagging

REVIEW | HISTORY

Meets Specifications

Dear Student,

I am really impressed with the amount of effort you've put into the project. You deserve applaud for your hardwork!

🎉 Finally, Congratulations on completing this project. You are one step closer to finishing your Nanodegree.

Wishing you good luck for all future projects 🎉

Some general suggestions

Use of assertions and Logging:

- Consider using Python assertions for sanity testing - assertions are great for catching bugs. This is especially true of a dynamically type-checked language like Python where a wrong variable type or shape can cause errors at runtime
- Logging is important for long-running applications. Logging done right produces a report that can be analyzed to debug errors and find crucial information. There could be different levels of logging or logging tags that can be used to filter messages most relevant to someone. Messages can be written to the terminal using print() or saved to file, for example using the Logger module. Sometimes it's worthwhile to catch and log exceptions during a long-running operation so that the operation itself is not aborted.

Debugging:

- Check out this guide on debugging in python

General Requirements

- Includes `HMM Tagger.ipynb` displaying output for all executed cells
- Includes `HMM Tagger.html`, which is an HTML copy of the notebook showing the output from executing all cells

All required files are included in the submission zip.
✅ `Jupyter Notebook`
✅ `HTML Export`

Suggestion:

You can export your conda environment into `environment.yaml` file so that you can recreate your conda environment later while practicing on your own system. Use the following command -

```
conda env export -f environment.yaml
```

Submitted notebook has made no changes to test case assertions

The test cases are intact throughout the notebook.

Baseline Tagger Implementation

Emission count test case assertions all pass.

- **The emission counts dictionary has 12 keys, one for each of the tags in the universal tagset**
- **"time" is the most common word tagged as a NOUN**

Both test cases associated with `emission_counts` are passing:
1) The dictionary contains the required 12 keys.
2) `time` is the most common word and appropriately tagged as a `noun`.

Suggestion: Here are some alternate implementations of `pair_counts()` function

```
def pair_counts(sequences_A, sequences_B):

    map = defaultdict(Counter)
    for i in range(len(sequences_A)):
        for key, value in zip(sequences_A[i],sequences_B[i]):
                map[key][value] += 1

    return map
```

```
def pair_counts(sequences_A, sequences_B):

    for (tag, word) in zip(sequences_A, sequences_B):
        tag_word_list[tag].append(word)

    for tag in tag_word_list.keys():
        map[tag] = Counter(tag_word_list[tag])

    return map
```

```
def pair_counts(sequences_A, sequences_B):

    map={}

    for item in set(sequences_A):
        map[item]={}
        for word in set(sequences_B):
            map[item][word]=0

    for idx, item in enumerate(sequences_B):
            map[sequences_A[idx]][item]+=1

    return map
```

Using Pandas

```
def pair_counts(sequences_A, sequences_B):

    # Create pandas of pairs and count occurences
    pairs = pd.DataFrame({'tags':sequences_A,'words':sequences_B, 'count':1
})\
        .groupby(['words','tags'])['count']\
        .count()\
        .reset_index()

    pair_counts = pairs.groupby('tags')[['words','count']]\
        .apply(lambda g: dict(g.values.tolist())).to_dict()

    return pair_counts
```

Baseline MFC tagger passes all test case assertions and produces the expected accuracy using the universal tagset.

- >95.5% accuracy on the training sentences
- 93% accuracy the test sentences

```
mfc_training_acc = accuracy(data.training_set.X, data.training_set.Y, mfc_m
odel)
print("training accuracy mfc_model: {:.2f}%".format(100 * mfc_training_ac
c))

mfc_testing_acc = accuracy(data.testing_set.X, data.testing_set.Y, mfc_mode
l)
print("testing accuracy mfc_model: {:.2f}%".format(100 * mfc_testing_acc))

assert mfc_training_acc >= 0.955, "Uh oh. Your MFC accuracy on the training
set doesn't look right."
assert mfc_testing_acc >= 0.925, "Uh oh. Your MFC accuracy on the testing s
et doesn't look right."
HTML('<div class="alert alert-block alert-success">Your MFC tagger accuracy
looks correct!</div>')
```

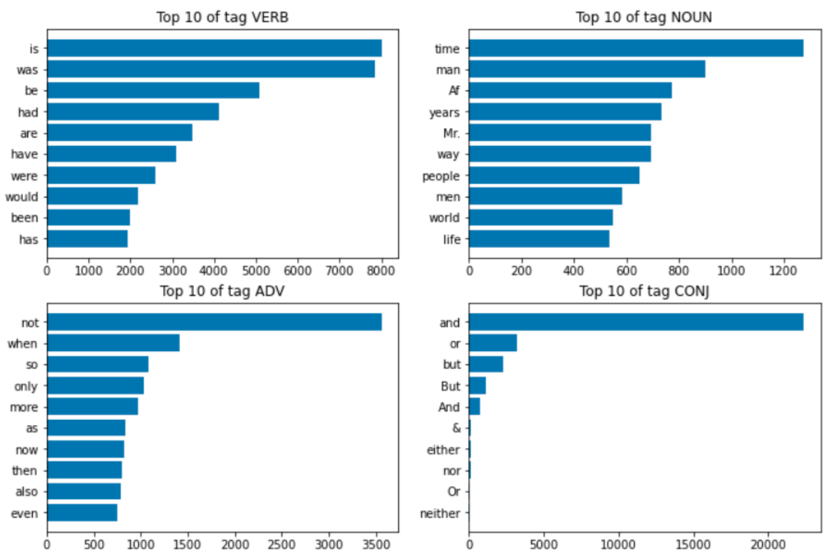
```
training accuracy mfc_model: 95.72%
testing accuracy mfc_model: 93.01%
```

Your MFC tagger accuracy looks correct!

Good job! 👍
The accuracy on training sentences is `95.72%`
Test sentence accuracy is `93.01%`

You can also consider plotting the top counts in each of the TAG sets as follows

```
def plot_word_count(word_counts, tag, max_count, ax):
    words =  dict(sorted(word_counts[tag].items(), key=lambda item: item[1]
))
    #print(words)
    top_words = {r[0]:r[1] for r in list(words.items())[-max_count:]}

    ax.barh(*zip(*top_words.items()))
    ax.set_title(f'Top {max_count} of tag {tag}')
    #ax.show()
fig, axs = plt.subplots(2, 2, figsize=(12,8))
plot_word_count(word_counts, 'VERB', 10, axs[0,0])
plot_word_count(word_counts, 'NOUN', 10, axs[0, 1])
plot_word_count(word_counts, 'ADV', 10, axs[1, 0])
plot_word_count(word_counts, 'CONJ', 10, axs[1, 1])
```



Suggestion: itertools.chain function can be used to merge tuples of words and sequences.

Calculating Tag Counts

All unigram test case assertions pass

All test cases passed!

N-grams of texts are used extensively in NLP and text mining tasks. An n-gram is a contiguous sequence of n items from a given sample of text or speech data. n-gram is just set of words occurring within a given window so when

- n=1 it is Unigram
- n=2 it is bigram
- n=3 it is trigram and so on
  When n > 3 this is usually referred to as four grams or five grams and so on.

Suggestion: Here are some alternate implementations of `unigram_counts()` function

```
def unigram_counts(sequences):
        return Counter(sequences)
```

```
def unigram_counts(sequences):
        return Counter(chain(*sequences))
```

```
def unigram_counts(sequences):

    map = Counter(tag for sentence in sequences for tag in sentence)

    return map
```

```
def unigram_counts(sequences):

    map={}
    List=list(itertools.chain.from_iterable(sequences))

    for item in set(List):
```

```
        map[item]=0
    for item in List:
        map[item]+=1

    return map
```

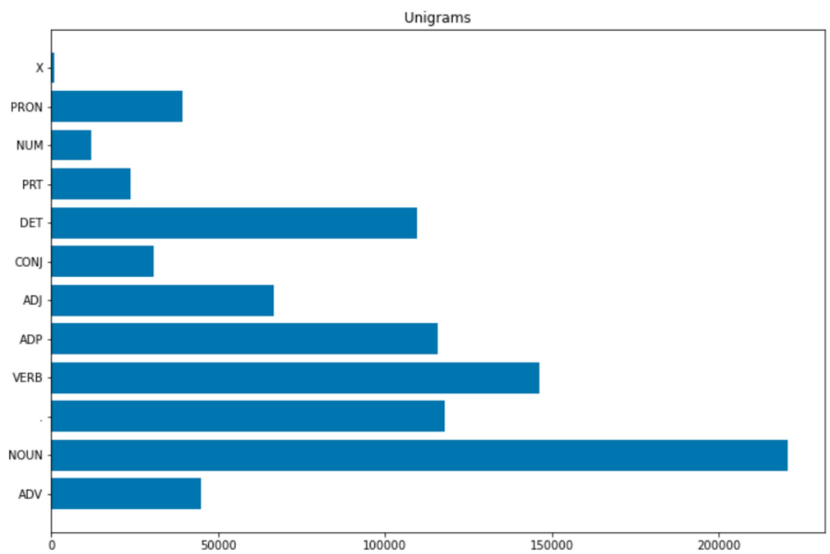
Using Pandas

```
def unigram_counts(sequences):

    unigram_counts = pd.Series(sequences).value_counts().to_dict()

    return unigram_counts
```

💡 Suggestion

You can also consider plotting the Unigram counts as follows

```
def plot_dictionary(unigrams, ax, title):
    ax.barh(*zip(*unigrams.items()))
    ax.set_title(title)

fig, axs = plt.subplots(1, 1, figsize=(12,8))
plot_dictionary(tag_unigrams, axs, 'Unigrams')
```



All bigram test case assertions pass

All test cases passed!

Suggestion: Here are some alternate implementations of `bigram_counts()` function

```
def bigram_counts(sequences):
    return Counter([pair for sequence in sequences for pair in
zip(sequence, sequence[1:])])
```

```
def bigram_counts(sequences):

    bigram_tag = dict(Counter(sequences))
```

```
    return bigram_tag
```

```
def bigram_counts(sequences):
    count = None
    for item in sequences:
        bigram = zip(item, item[1:])
        if (count is not None):
            count += Counter(bigram)
        else:
            count = Counter(bigram)

    return dict(count)
```

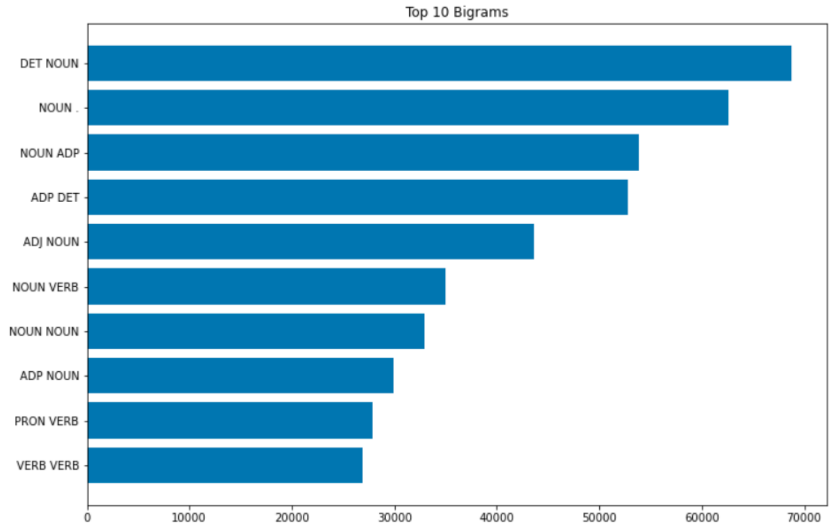
```
def bigram_counts(sequences):
    counts = Counter()
    counts.update(chain(*(zip(s[:-1], s[1:]) for s in sequences)))
    return counts
```

```
def bigram_counts(sequences):
    map = {}

    tagSet = set(list(itertools.chain.from_iterable(sequences)))

    for t1 in tagSet:
        for t2 in tagSet:
            map[(t1,t2)]=0

    for i in sequences:
        for j in range(len(i)-1):
              map[(i[j],i[j+1])]+=1

    return map
```

💡 Suggestion

You can also consider plotting the top 10 bigrams as follows

```
def plot_bigrams(bigrams, max_count, ax):
    bigs =  dict(sorted(bigrams.items(), key=lambda item: item[1]))
    top_bigs = {f'{r[0][0]} {r[0][1]}':r[1] for r in list(bigs.items())[-ma
x_count:]}
    print(top_bigs)
    ax.barh(*zip(*top_bigs.items()))
    ax.set_title(f'Top {max_count} Bigrams')

fig, axs = plt.subplots(1, 1, figsize=(12,8))
plot_bigrams(tag_bigrams,  10, axs)
```



All start and end count test case assertions pass

The test cases for all three `Tag Counting` implementations (Unigram tagging, Bigram Tagging, Sequence Starting & Ending counts) are passing.

Suggestion: Here are some alternate implementations of `starting_counts()` function

```
def starting_counts(sequences):
    return Counter(next(zip(*sequences)))
```

```
def starting_counts(sequences):

    for tag in data.training_set.tagset:
        starting_tags[tag] = len([seq[0] for seq in sequences if seq[0]==ta
g])
    return starting_tags
```

Suggestion: Here are some alternate implementations of `ending_counts()` function

```
def ending_counts(sequences):
    return Counter([sequence[-1] for sequence in sequences])
```

```
def ending_counts(sequences):

    for tag in data.training_set.tagset:
        ending_tags[tag] = len([seq[-1] for seq in sequences if seq[-1]==ta
g])
    return ending_tags
```

Basic HMM Tagger Implementation

All model topology test case assertions pass

Excellent work. You've implemented an appropriate topology for your HMM Tagger. 👍

```
# finalize the model
basic_model.bake()

assert all(tag in set(s.name for s in basic_model.states) for tag in data.t
raining_set.tagset), \
       "Every state in your network should use the name of the associated t
ag, which must be one of the training set tags."
assert basic_model.edge_count() == 168, \
       ("Your network should have an edge from the start node to each stat
e, one edge between every " +
        "pair of tags (states), and an edge from each state to the end nod
e.")
HTML('<div class="alert alert-block alert-success">Your HMM network topolog
y looks good!</div>')
```

Your HMM network topology looks good!

Basic HMM tagger passes all assertion test cases and produces the expected accuracy using the universal tagset.

- >97% accuracy on the training sentences
- >95.5% accuracy the test sentences

Awesome! The final model attains the required training and test accuracies to pass this project. 👏

```
training accuracy basic hmm model: 97.54%
testing accuracy basic hmm model: 96.16%
```

📖 Additional Reading - Here are some of my favourite resources on Hidden Markov Models. Hope you find them useful.

- https://nadesnotes.wordpress.com/2016/04/20/natural-language-processing-nlp-fundamentals-hidden-markov-models-hmms/
- https://www.freecodecamp.org/news/an-introduction-to-part-of-speech-tagging-and-the-hidden-markov-model-953d45338f24/
- https://medium.com/@postsanjay/hidden-markov-models-simplified-c3f58728caab

DOWNLOAD PROJECT

RETURN TO PATH