assignment1_part2

February 4, 2022

[ ]:
```
version = "REPLACE_PACKAGE_VERSION"
```

---

1 Assignment 1 Part 2: N-gram Language Models (Cont.) (30 pts)

In this assignment, we're going to train an n-gram language model that is able to "imitate" William Shakespeare's writing.

[ ]:
```
# Configure nltk

import nltk

nltk_data_path = "assets/nltk_data"
if nltk_data_path not in nltk.data.path:
    nltk.data.path.append(nltk_data_path)
```

[ ]:
```
# Copy and paste the functions you wrote in Part 1 here and import any␣
 ↪libraries necessary
# We have tried a more elegant solution by using
# from ipynb.fs.defs.assignment1_part1 import load_data, build_vocab,␣
 ↪build_ngrams
# but it doesn't work with the autograder...

def load_data():
    """
    Load text data from a file and produce a list of token lists
    """

    sentences = []

    # YOUR CODE HERE
    raise NotImplementedError()

    return sentences
```

```
def build_vocab(sentences):
    """
    Take a list of sentences and return a vocab
    """

    vocab = []

    # YOUR CODE HERE
    raise NotImplementedError()

    return vocab

def build_ngrams(n, sentences):
    """
    Take a list of unpadded sentences and create all n-grams as specified by␣
 ↪the argument "n" for each sentence
    """

    all_ngrams = []

    # YOUR CODE HERE
    raise NotImplementedError()

    return all_ngrams
```

1.1 Question 4: Guess the next token (20 pts)

With the help of the three functions you wrote in Part 1, let's first answer the following question as a review on n-grams.

Assume we are now working with bi-grams. What is the most likely token that comes after the sequence `<s> <s> <s>`, and how likely? Remember that a bi-gram language model is essentially a first-order Markov Chain. So, what determines the next state in a first-order Markov Chain?

Complete the function below to return a `tuple`, where `tuple[0]` is a `str` representing the mostly likely token and `tuple[1]` is a `float` representing its (conditional) probability of being the next token.

[ ]:
```
def bigram_next_token(start_tokens=("<s>", ) * 3):
    """
    Take some starting tokens and produce the most likely token that follows␣
 ↪under a bi-gram model
    """

    next_token, prob = None, None

    # YOUR CODE HERE
    raise NotImplementedError()
```

```
    return next_token, prob
```

[ ]:
```
# Autograder tests

stu_ans = bigram_next_token(start_tokens=("<s>", ) * 3)

assert isinstance(stu_ans, tuple), "Q4: Your function should return a tuple. "
assert len(stu_ans) == 2, "Q4: Your tuple should have two elements. "
assert isinstance(stu_ans[0], str), "Q4: tuple[0] should be a str. "
assert isinstance(stu_ans[1], float), "Q4: tuple[1] should be a float. "

# Some hidden tests


del stu_ans
```

1.2 Question 5: Train an n-gram language model (10 pts)

Now we are well positioned to start training an n-gram language model. We can fit a language model using the `MLE` class from `nltk.lm`. It requires two inputs: a list of all n-grams for each sentence and a vocabulary, both of which you have already written a function to build. Now it's time to put them together to work.

Complete the function below to return an `nltk.lm.MLE` object representing a trained n-gram language model.

[ ]:
```
from nltk.lm import MLE

def train_ngram_lm(n):
    """
    Train a n-gram language model as specified by the argument "n"
    """

    lm = MLE(n)

    # YOUR CODE HERE
    raise NotImplementedError()

    return lm
```

[ ]:
```
# Autograder tests

stu_n = 4
stu_lm = train_ngram_lm(stu_n)
stu_vocab = build_vocab(load_data())
```

```
assert isinstance(stu_lm, nltk.lm.MLE), "Q3b: Your function should return an␣
↪nltk.lm.MLE object. "

assert hasattr(stu_lm, "vocab") and len(stu_lm.vocab) == len(stu_vocab) + 1,␣
↪"Q3b: Your language model wasn't trained properly. "

del stu_n, stu_lm, stu_vocab
```

FINALLY, are you ready to compose sonnets like the real Shakespeare?! We provide some starter code below, but absolutely feel free to modify any parts of it on your own. It'd be interesting to see how the "authenticity" of the sonnets is related to the parameter n. Do the sonnets feel more Shakespeare when you increase n?

[ ]:
```
# Every time it runs, depending on how drunk it is, a different sonnet is␣
↪written.
n = 3
num_lines = 14
num_words_per_line = 8
text_seed = ["<s>"] * (n - 1)

lm = train_ngram_lm(n)

sonnet = []
while len(sonnet) < num_lines:
    while True:  # keep generating a line until success
        try:
            line = lm.generate(num_words_per_line, text_seed=text_seed)
        except ValueError:  # the generation is not always successful. need to␣
↪capture exceptions
            continue
        else:
            line = [x for x in line if x not in ["<s>", "</s>"]]
            sonnet.append(" ".join(line))
            break

# pretty-print your sonnet
print("\n".join(sonnet))
```