CSC 383-Project1

Writing Shakespeare

c = 2

```
    m = create_ngram_model(NgramModel, "shakespeare_input.txt", 2)
    print(m.random_text(250))
```

```
output:
Firats histrest prelst? as fult the resse quareir's nobe
How it? his trumas wheet th
To ostaiddruck whim the frot sk ing evoy, am whe trould
armar dal ve negre verson captent:
'Tim to be amestare a diek, damost: the you no mou?

By all
To con monotir
```

c = 3

```
    m = create_ngram_model(NgramModel, "shakespeare_input.txt", 3)
    print(m.random_text(250))
```

```
output:
First fathe he for took of bit say ducates and do't; 'As it them besentiendition,
you,
and leasong lovice, away and use,
For his were, is feeble and betting!
Thy be scutest thous the dignion, lieven a grouse is do to o' the him,
beggarmonumbs, so vil
```

c = 4

```
    m = create_ngram_model(NgramModel, "shakespeare_input.txt", 4)
    print(m.random_text(250))
```

```
output:
First Gentlement you are me archers leave near that go inforce
Is, like a repenteous days' way for he do fetter
apes, he need: yet him abide word quickly earl of woo immortal perselvet; it was we
all I thing Nilus! Shall ever saw him that my lays bea
```

c = 7

```
    m = create_ngram_model(NgramModel, "shakespeare_input.txt", 7)
    print(m.random_text(250))
```

```
First Citizen:
Where reside
But why did you cannot, no, nor the whole world increases, his youth, how my grief;
And, being down before, draw you out? say thus I conceive
And laugh; but that was most mighty strength.

VALENTINE:
How! traitor.

CYMBELI
```

The initial word in every output is a variation of "first" because that is the opening word in the training set. This happens because the `start_pad()` function is only called once when creating n-grams such that the only word that really gets padded is "first"

Perplexity

I first trained the model on `shakespeare_input.txt` for different values of `c` and then first tested the model on `shakespeare_sonnets.txt` . Results are as following

```
c = 1  = 12.2419
c = 2  = 7.9101
c = 3  = 6.1222
c = 4  = 6.4856
c = 5  = 8.9371
c = 6  = 13.9203
c = 7  = 21.6897
c = 8  = 31.7205
c = 9  = 42.1431
c = 10  = 51.1800
```

After this I tested the same model on `nytimes_article.txt` . Results are as follows

```
c = 1  = 15.0042
c = 2  = 11.0188
c = 3  = 9.7889
c = 4  = 10.8516
c = 5  = 15.6087
c = 6  = 23.7098
c = 7  = 34.0777
c = 8  = 44.5696
c = 9  = 53.1590
c = 10  = 59.0892
```

We see here that the NY Times article has higher perplexity than Shakespeare's sonnets for corresponding values of `c` . This is because the model was first trained on Shakespeare. Therefore the model is having a tough time with testing this data.

Different lambdas

Here I have tried out different values of lambdas by either changing the value of `self.c` or by using the `set_lambda()`

```
m1 = NgramModelWithInterpolation(1, 0) #lambda = [0.5,0.5]
m1.update('abab')
print(m1.prob("a","a")) #output = 0.25
print(m1.prob("a","b")) #output = 0.75
```

```
m3 = NgramModelWithInterpolation(2, 1) #lambda = [2/3,1/6,1/6]
m3.set_lambdas([2/3,1/6,1/6])
m3.update('abab')
m3.update('abcd')
print(m3.prob("~a","b"))  #output = 0.4841269841269841
print(m3.prob("ba","b"))  #output = 0.4174603174603174
print(m3.prob("~c","d"))  #output =  0.26111111111111107
print(m3.prob("bc","d"))  #output = 0.3611111111111111
```

```
m2 = NgramModelWithInterpolation(2, 1) #lambda = [1/3,1/3,1/3]
m2.update('abab')
m2.update('abcd')
print(m2.prob("~a","b"))  #output = 0.46825396825396826
print(m2.prob("ba","b"))  #output = 0.43492063492063493
print(m2.prob("~c","d"))  #output = 0.2722222222222222
print(m2.prob("bc","d"))  #output = 0.3222222222222222
```

```
m4 = NgramModelWithInterpolation(2, 1) #lambda = [0,2/3,1/6]
m4.set_lambdas([0,2/3,1/6])
m4.update('abab')
m4.update('abcd')
print(m4.prob("~a","b"))  #output = 0.31746031746031744
print(m4.prob("ba","b"))  #output = 0.31746031746031744
print(m4.prob("~c","d"))  #output = 0.17777777777777776
print(m4.prob("bc","d"))  #output = 0.17777777777777776
```

Language Classification

I first trained the models on the test data

The class "language_identification" is used for identifying the language of a given text. The class has several methods including **init**, identify_word, classifier_accuracy and evaluate.

The **init** method initializes the models dictionary and creates n-gram models for each language using the 'create_ngram_model_lines' function, which takes a path to a training dataset, the order of the n-gram model and the interpolation parameter as its inputs. The path to the training data is constructed by concatenating the language code with the string "cities_train/train/" and '.txt'.

The identify_word method takes a text as an input and returns the language code of the language that the text is most likely to be in. It does this by looping through the models dictionary and for each model, it calculates the probability of the given text using the 'prob' method of the n-gram model. The calculated probabilities are then stored in a probability_code list along with the corresponding language codes. The max function is then used to find the language code with the highest probability.

The classifier_accuracy method takes a language code as an input and returns the accuracy of the model on the test data for the given language code. It does this by reading the test data from a file using the language code to construct the path, and then calling the identify_word method on each line of the test data. The number of correct classifications is then divided by the total number of test examples to get the accuracy.

The evaluate method loops through all language codes and calls the classifier_accuracy method for each one, printing out the accuracy for each language.

Overall, the class uses n-gram models with interpolation to identify the language of a given text and evaluate the accuracy of the models using test datasets. However, the accuracy that the class reports may be different from the actual performance of the models because the dataset that the class uses may not be representative of the real world. Also, the accuracy is not reported so we can't make a judgement about the performance of the models.

```
with c = 3, k = 1, lambda = [0.1, 0.2,0.3,0.4]
results:
af :  56.00000000000001 %
cn :  94.0 %
de :  71.0 %
fi :  79.0 %
fr :  82.0 %
in :  52.0 %
ir :  51.0 %
pk :  68.0 %
za :  57.99999999999999 %
```

```
with c = 3, k = 1, lambda = [1/4,1/4,1/4,1/4]
results:
af :  57.99999999999999 %
cn :  95.0 %
de :  71.0 %
fi :  77.0 %
fr :  80.0 %
in :  54.0 %
ir :  52.0 %
pk :  69.0 %
za :  57.99999999999999 %
```

```
with c = 4, k = 1, lambda = [1/5,1/5,1/5,1/5,1/5]
results:
af :  63.0 %
cn :  95.0 %
de :  71.0 %
fi :  80.0 %
fr :  82.0 %
in :  52.0 %
ir :  51.0 %
pk :  69.0 %
za :  60.0 %
```

```
with c = 4, k = 0.5, lambda = [1/5,1/5,1/5,1/5,1/5]
results:
af :  62.0 %
cn :  95.0 %
de :  72.0 %
fi :  80.0 %
fr :  81.0 %
in :  49.0 %
ir :  55.00000000000001 %
pk :  68.0 %
za :  64.0 %
```

```
with c = 5, k = 1, lambda = [1/6,1/6,1/6,1/6,1/6,1/6]
results:
af :  61.0 %
cn :  95.0 %
de :  72.0 %
fi :  78.0 %
fr :  81.0 %
in :  49.0 %
ir :  51.0 %
pk :  68.0 %
za :  57.99999999999999 %
```

These are all results I achieved by changing the values. Changing these values increases a particular countries accuracy while decreasing another countries accuracy hence I do not have a conclusive set of values that will increase the accuracy of each and every country on the list