ANLP 2024 Assignment 1

Marked anonymously: do not add your name(s) or ID numbers.

1 Preprocessing each line (10 marks)

```
def preprocess_line(self, line):
    """
    Preprocess the input line by keeping only alphabets, spaces, and periods.
    Convert digits to '0' and lowercase all characters.

    Parameters:
    line - the input sentence to preprocess
    """
    expression = r'[a-zA-Z\s\.]+|\d+'
    matches = re.findall(expression, line)
    processed_text = ''.join(re.sub('\d', '0', match.lower()) for match in matches)
    return f'##{processed_text}#'
```

The 'preprocess_line(line)' function takes a line of the text from the training corpus and processes it by removing unwanted characters. A regular expression is used to match English letters, digits (0-9), spaces (\s), and period (\.). For each match, any digit is replaced with a zero, and uppercase letters are converted into lowercase. The join () function combines the processed characters into a single string. Finally, two hashes (##) are added at the beginning to mark the start of the sentence, and one hash (#) is added at the end to indicate the sentence's conclusion. The resulting string is returned. The hashes help the model clearly recognise sentence boundaries and determine the probability of the beginning and end of a sentence.(beginning/end of sentence)

2 Examining a pre-trained model (10 marks)

Analysis of the 'model-br-en' file suggests the model does not use Maximum Likelihood Estimation (MLE), as no trigram has zero probability. The uniform probability distribution for trigrams that are generally less likely to occur in English implies the use of a smoothing technique, likely Add-k or Good-Turing, which redistributes probability mass to trigrams with zero counts. The uniformity of probabilities indicates the model does not rely on more complex methods, such as interpolation or backoff, which would generate variable probabilities for unseen trigrams.

Example of Probability Distribution from the model file:

Trigram	Probability	Observation frequency
'ty '	0.3484	Observed frequently
'ty.'	0.6093	Observed frequently
'ty0'	0.0006211	Unseen in English
'tya'	0.0006211	Unseen in English
'tyb'	0.0006211	Unseen

In this example above, trigrams like 'ty', and 'ty.' are assigned the highest probabilities, as these are commonly found in English words like 'activity', 'type' or at the end of words ending in 'ty.'. On the other hand, combinations like 'tya', 'tyb' and 'ty0' do not form valid English words and, receive low probabilities due to the application of a smoothing algorithm. The redistribution of probability mass from seen to unseen trigrams that share a common starting bigram, such as 'ty', ensures that the total probability sums to one, creating a valid probability distribution.

3 Implementing a model: description and example probabilities (35 marks)

3.1 Model description

The language model we implemented calculates the probability of each character given its 2-character history using add-k/ alpha smoothing. This model was selected due to its effectiveness in tasks such as text classification. The smoothing technique addresses the issue of zero probabilities for trigrams not observed during training when evaluating test data. It achieves this by redistributing a portion of the probability mass, determined by the value of *k*, to n-grams not encountered during training. The value of *k* can be optimised to minimise perplexity on a validation or held-out set. This approach offers an advantage over add-one (Laplace) smoothing, which may excessively redistribute probability mass from frequent words in cases of large vocabulary sizes.

Using this method, the probability is calculated using the below formula:

$$P(c_i|c_{i-2}, c_{i-1}) = \frac{\text{Count}(c_{i-2}, c_{i-1}, c_i) + k}{\text{Count}(c_{i-2}, c_{i-1}) + k \times |V|}$$

Figure Add-k Smoothing (Jurafsky & Martin)

Here, k is typically a value between 0 and 1. The K value is a hyperparameter and needs to be optimised based on the validation set.

P(ci|ci-2, ci-1) is the probability of a trigram given a bigram.

Count(ci-2, ci-1, ci) is the count of the trigram in the corpus.

Count(ci-2, ci-1) is the count of the bigram for the given trigram, i.e. count of the prefix for a trigram in the training corpus

|V| - Vocabulary size. Since we are developing a 3-gram character model, the vocabulary size is the list of allowed characters that the pre-processing function allows, i.e., 30 characters.

Implementation: The training corpus was partitioned into a training set and a validation (held-out) set using a 90-10 split, where the validation set comprises 10% of the original corpus. The held-out data was utilised to optimise the smoothing parameter 'k'.

The contents of the training set were processed to extract bigrams and trigrams from the character sequences. Their respective counts were stored in a dictionary following the necessary preprocessing steps. All possible trigram combinations ($29^2 \times 30 = 26{,}100$) were created from the defined vocabulary to generate the language model. The probabilities for each trigram were calculated based on the trigram and bigram counts derived from the training data. These trigrams and their computed probabilities were then output to a file.

Optimisation of k: To determine the optimal value of *k*, we utilised the validation set. The parameter was incrementally increased, and the value that minimised the perplexity of the validation set was selected. Perplexity, an intrinsic evaluation metric for language models, was used to measure the model's performance. The table below presents the perplexity values for different *k* values. We chose k as 0.5.

K value	perplexity
0.00001	88.6825593
0.001	63.0505822
0.001	44.8366516
0.01	31.95056
0.1	23.1489243
0.2	21.2315426
0.3	20.2654923
0.4	19.6488067
0.5	19.2092503

3.2 Model excerpt

The table below is an excerpt from the language model for English displaying all n-grams with a 2-character history "ng" and their probabilities. The sum of all probabilities of 3-character trigrams given 2-character history is 1. The trigrams which are not seen in the training set and have zero counts, such as 'nga', 'nbg', and 'ngc', have the same probability assigned due to smoothing.

nga	0.00069638
ngb	0.00069638
ngc	0.00069638
ngd	0.00487465
nge	0.0856546
ngf	0.00208914
ngg	0.00069638
ngh	0.00069638
ngi	0.00208914
ngj	0.00069638
ngk	0.00069638
ngl	0.00348189
ngm	0.00069638
ngn	0.00208914
ngo	0.00626741
ngp	0.00069638
ngq	0.00069638
ngr	0.0132312
ngs	0.01880223
ngt	0.0132312
ngu	0.00348189
ngv	0.00069638
ngw	0.00069638
ngx	0.00069638
ngy	0.00069638
ngz	0.00069638
ng0	0.00069638
ng_	0.80849582
ng.	0.02437326
ng#	0.00069638
Total	1

4 Generating from models (15 marks)

The model's data file is loaded into memory as a dictionary. To generate random text from the pre-trained language models, the algorithm begins by identifying the probabilities associated with the sequence '##', which marks the start of a sentence. The algorithm filters for all keys that begin with '##' and selects a trigram using a weighted random selection based on their respective probabilities. The predicted next character is then appended to the output string. Subsequent characters are generated by applying a sliding window over the last two characters of the current output string. This process iterates until a predefined number of characters, *n*, are produced. In each iteration, the next character is determined by the two-character context from the previously generated string.

Since the model was trained on individual sentences, it cannot predict the next character upon encountering the end-of-sentence marker ('#'). At this point, the two-character history is reset to '##', indicating the beginning of a new sentence.

Examples of Generated Outputs from the Language Models:

Language model developed using add-k smoothing:

'we its forthe of to thergo be gral oble me nesion re amisaftion tom thas faccesed so proveroust and al red wound for i whe hattencyh0bc.parm thel would 0 juse the of sulassidebassel i willy as.i the for combeire 0 as mods or wort th thavate thave wourabs degich actuall prours of the expleveret'

model-br.en:

'thing thhday we ave whates pre. beck ace. loseeppe ame dong appeeks. toons. daddy we tooddy theekaby in. what mis. yeale de. hoes frop. youumb. the canname of too that thatss tenicturropeeekayou. what. shhhromell me. youl. button to too as by beeekaysssss. whatsssse. wout welet an t'

Conclusion:

The generated outputs suggest that the two models were trained on different corpora. The pre-trained model, *model-br.en*, appears to generate shorter sequences than the language model trained on the Europarl corpus. Both language models can produce only some valid English words. Valid words are shorter and functional words such as 'end', 'was', 'that', 'the', and 'too', likely due to the higher frequency of these words in the training data.

5 Computing perplexity (15 marks)

Methodology:

Perplexity is a metric derived from the exponentiation of cross-entropy, commonly used to quantify the level of uncertainty or "surprise" in a probabilistic model. The function implemented for calculating perplexity accepts a test document as input and applies the formula in the figure below. The calculation process begins with text preprocessing of each line. Subsequently, a sliding window of size three is used to generate all possible trigrams (w_1, w_2, …, w_n). The algorithm computes the average negative log probability of each token in the test data, and the perplexity is then obtained by exponentiating this value.

$$H(W) = -\frac{1}{N}\log P(w_1 w_2 \dots w_N)$$

$$\text{Perplexity}(W) = 2^{H(W)}$$

Figure Perplexity in Language Models (Jurafsky & Martin)

N: Number of tokens in the test sequence.
H(W): is the cross-entropy.
P(w_1, w_2, …, w_n). – probability of the test sequence.

Interpretation of Results:

Training - Corpus	Model	Perplexity
English	Add-k smoothing, k=0.5	24.40
Spanish	Add-k smoothing, k=0.5	39.67
German	Add-k smoothing, k =0.5	48.49

The English Add-k smoothing model provides the lowest perplexity value for the test set (24.40) compared to the other two language models. This indicates that the English model is better at predicting the context of the test set. The lower perplexity suggests that the text of the provided test set is written in English, as the perplexity is lower than the Spanish and German models.

Evaluate Document Language Based on Perplexity:

We are not able to determine the language of a document based only on its perplexity under the English language model. Every language model (English, German Spanish) should estimate the perplexity for the given document, as the perplexity is a relative measure. To sum up, without comparing the perplexity across models, we have no context to interpret how well a model performs at predicting the content of a given set or in classifying the text to the correct language.

6 Extra question (15 marks)

As a further step of extending our language model, we wanted to implement a more advanced smoothing method that does not assign the same probability mass to trigrams not encountered in training data which is a drawback of alpha smoothing.

We explored implementing a language model that uses interpolation, which considers the context of the bigram and unigram probabilities. This method depends on adjusting the hyperparameters (λ_1, λ_2, λ_3) which apply to different n-gram levels, weighting the probabilities of trigrams, bigrams and unigrams.

Methodology:

The function 'smoothed_trigram_probability(trigram)' initially defines three empirical values λ_1, λ_2, λ_3, which serve as weighting factors for the probabilities of the trigram, bigram, and unigram, respectively. These probabilities are calculated using the following equation.

$$\begin{aligned}\hat{P}(w_n|w_{n-2}w_{n-1}) \;=\; & \lambda_1 P(w_n) \\ & +\lambda_2 P(w_n|w_{n-1}) \\ & +\lambda_3 P(w_n|w_{n-2}w_{n-1})\end{aligned}$$

Figure Language Model Interpolation (Jurafsky & Martin)

Interpretation of Results:

Training - Corpus	Model	Perplexity
English	Interpolation, λ_1, λ_2, λ_3 = [0.1,0.2,0.7]	19.08
Spanish	Interpolation, λ_1, λ_2, λ_3 = [0.1,0.2,0.7]	35.44
German	Interpolation, λ_1, λ_2, λ_3 = [0.1,0.2,0.7]	35.94

Unlike the Add-k smoothing technique, the interpolation method does not uniformly distribute the probability mass from observed to unseen trigrams. Instead, it redistributes the probability based on the contribution of various n-gram levels. Furthermore, the perplexity value obtained using interpolation was lower than the achieved with Add-k smoothing, suggesting that interpolation leads to more reliable and effective probability estimates.

Optimization of λ values:

To determine the optimal value of λ_1, λ_2, λ_3 we used the validation set. For different values of lambdas, we calculated the perplexity and chose the values that produced the lowest perplexity.

λ1 λ2, λ3	Perplexity (English model)
[0.05 0.05 0.9]	18.145
[0.05 0.05 0.8]	15.762
[0.1 0.1 0.8]	16.528
[0.1 0.2 0.7]	16.712
[0.2 0.1 0.7]	15.347
[0.1 0.3 0.6]	17.107

Appendix:

The below code generates a 90-10 split to produce the training-validation sets from the given corpus:

```
import numpy as np


def split_file_by_line(input_file, output_file_90, output_file_10):
    # Read the content of the input file
    with open(input_file, 'r', encoding='utf-8') as f:
        lines = f.readlines()

    # Calculate the number of lines for the 90% and 10% split
    total_lines = len(lines)
    split_index = int(total_lines * 0.9)  # 90% of the total lines

    # Split the lines
    lines_90 = lines[:split_index]
    lines_10 = lines[split_index:]

    # Write 90% lines to the first output file
    with open(output_file_90, 'w', encoding='utf-8') as f_90:
        f_90.writelines(lines_90)

    # Write 10% lines to the second output file
    with open(output_file_10, 'w', encoding='utf-8') as f_10:
        f_10.writelines(lines_10)


input_file = 'training.de'        # Input file path
output_file_90 = 'training-corpus.de'  # Output file for 90% of the lines
output_file_10 = 'heldout.de'  # Output file for 10% of the lines

split_file_by_line(input_file, output_file_90, output_file_10)
```

The below code snippet generates perplexity for a given test document using a language model

```
import NgramModel

# Instantiate object of the model trained with English corpus
ngram_model = NgramModel.NGramModel("training-corpus.en")
ngram_model.train_model()

# Instantiate object of the model trained with German corpus
ngram_model_de = NgramModel.NGramModel("training-corpus.de")
ngram_model_de.train_model()


# Instantiate object of the model trained with Spanish corpus
ngram_model_es = NgramModel.NGramModel("training-corpus.es")
ngram_model_es.train_model()


# Caculate perplexity with the different models
with open('test', 'r') as file:
    test_data = file.read()

    print("Perplixity with en test data add_k", ngram_model.perplexity(test_data, "add_k"))
    print("Perplexity with en test data interpolation", ngram_model.perplexity(test_data, "interpolation"))

    print("Perplixity with es test data add_k", ngram_model_es.perplexity(test_data, "add_k"))
    print("Perplexity with es test data interpolation",  ngram_model_es.perplexity(test_data, "interpolation"))

    print("Perplixity with de test data add_k",  ngram_model_de.perplexity(test_data, "add_k"))
    print("Perplexity with de test data interpolation",  ngram_model_de.perplexity(test_data, "interpolation"))
```

The below code snippets optimise hyperparameters k and lambdas using the validation corpus

```
import NgramModel


# Optimize the value of 'k' for add_k smoothing
with open('heldout.en', 'r') as file:
    text = file.read()
    model = NgramModel.NGramModel("training-corpus.en")
    model.train_model()
    model.optimize_k(text)
```

```
import NgramModel

# Optimize the value of lambdas for interpolation
with open('heldout.en', 'r') as file:
    text = file.read()
    model = NgramModel.NGramModel("training-corpus.en")
    model.train_model()
    model.optimize_lambdas(text)
```

The below code snippet generates random output from a language model:

```
import RandomTextGenerator

# Instantiate an object of RandomTextGenerator with the path of the language model file
ngram_text_generator = RandomTextGenerator.RandomTextGenerator(r'model-br.en', '##')

# Call the function to generate output from language model for 300 characters
random_output = ngram_text_generator.generate_from_lm(300)

print("Model: model-br.en", random_output, "Len: ", len(random_output))


# Instantiate an object of RandomTextGenerator with the path of the language model file
ngram_text_generator = RandomTextGenerator.RandomTextGenerator(r'model/model_en_add_k_smoothing', '##')

# Call the function to generate output from language model for 300 characters
random_output = ngram_text_generator.generate_from_lm(300)

print("Model: model_en_add_k_smoothing", random_output, "Len: ", len(random_output))
```

GenerateLanguageModel.py: This code snippet generates the language models for the different training sets

```python
import NgramAddKSmoothingModel
import InterpolationSmoothingModel
import NgramTrainModel
import InterpolationSmoothingModel

#This class generates the language model files. The
######################################## MODELS ADD_K / INTERPOLATION SMOOTHING ################################

# Vocabulary containing 30 characters
vocabulary = 'abcdefghijklmnopqrstuvwxyz0_.#'

# The training corpus containing the 90% split of the provided corpus
list_of_training_corpus = ['training-corpus.en','training-corpus.de','training-corpus.es']

# Output file names for language model files produced using add_k smoothing
list_of_models_add_k = ['model_en_add_k_smoothing','model_de_add_k_smoothing','model_es_add_k_smoothing']

# Output file names for language model files produced using interpolation
list_of_models_interpolation = ['model_en_interpolation_smoothing','model_de_interpolation_smoothing','model_es_interpolation_smoothing']

for training_corpus, model_add_k, model_interpolation in zip(list_of_training_corpus,list_of_models_add_k,list_of_models_interpolation):

    # Train the model on the respective corpus
    ngram_train_model = NgramTrainModel.NgramTrainModel(training_corpus)

    # Get the trigram, bigram and unigram counts
    trigram_counts, bigram_counts, unigram_counts = ngram_train_model.train_model()

    # Generate probabilities for all trigrams using add-k/alpha smoothing and output to a file
    ngram_model_add_k = NgramAddKSmoothingModel.NgramAddKSmoothingModel(trigram_counts,bigram_counts,unigram_counts)
    ngram_model_add_k.write_trigram_probabilities_to_file(vocabulary,model_add_k)


    # Generate probabilities for all trigrams using interpolation and output to a file
    ngram_model_interpolation = InterpolationSmoothingModel.InterpolationSmoothingModel(trigram_counts,bigram_counts,unigram_counts)
    ngram_model_interpolation.write_interpolation_smoothed_probabilities(vocabulary,model_interpolation)
```

NgramModel.py: This class is used to train, pre-process, calculate perplexity and optimise parameters:

```
class NGramModel:
    def __init__(self, model_name):
        """
        Initialize the class with the model name (file to read).
        """
        self.model_name = model_name
        self.trigram_counts = {}
        self.bigram_counts = {}
        self.unigram_counts = {}
        self.model_probabilities = {}
        self.starting_bigram = ""

    def preprocess_line(self, line):
        """
        Preprocess the input line by keeping only alphabets, spaces, and periods.
        Convert digits to '0' and lowercase all characters.

        Parameters:
        line - the input sentence to preprocess
        """
        expression = r'[a-zA-Z\s\.]+|\d+'
        matches = re.findall(expression, line)  # Extract alphabets, spaces, periods, and digits
        processed_text = ''.join(re.sub('\d', '0', match.lower()) for match in matches)  # Replace digits and lowercase
        return f'##{processed_text}#'
```

```
def ngrams(self, output, input, n):
    """
    Generate n-grams and update the output dictionary with their counts.

    Parameters:
    output - dictionary to store n-gram counts
    input - list of tokens (processed words)
    n - the size of n-grams to generate
    """
    for i in range(len(input) - n + 1):
        g = ''.join(input[i:i + n])
        output.setdefault(g, 0)
        output[g] += 1
    return output

def train_model(self):
    """
    This function reads the corpus from a file, preprocesses and get the counts for character level trigrams, bigrams and unigrams
    """

    # Read all the lines in a file
    with open(self.model_name, "r") as file:
        lines = file.readlines()

    # Process each line in the file
    for l_num in lines:
        process_sentence = self.preprocess_line(l_num.strip())
        tokens = list(process_sentence)  # Convert processed sentence into a list of characters (tokens)

        # Count the n-grams and update the respective dictionaries
        self.trigram_counts = self.ngrams(self.trigram_counts, tokens, 3)
        self.bigram_counts = self.ngrams(self.bigram_counts, tokens, 2)
        self.unigram_counts = self.ngrams(self.unigram_counts, tokens, 1)

    return self.trigram_counts, self.bigram_counts, self.unigram_counts
```

NgramModel.py

```python
def perplexity(self, text, model, hyper_paramater=None):
    """
    This function calculates perplexity of a document against the trained models

    Parameters:
    text - The text content of the document
    model - The type of model (Add-k / Interpolation)
    hyper_paramater - Hyper parameter to be passed to the model
    """

    ngram_model_add_k = NgramAddKSmoothingModel.NgramAddKSmoothingModel(self.trigram_counts,self.bigram_counts,self.unigram_counts)
    ngram_model_interpolation = InterpolationSmoothingModel.InterpolationSmoothingModel(self.trigram_counts,self.bigram_counts,self.unigram_counts)

    # Window size is 3 to get all the trigrams
    window_size = 3
    trigrams = []
    n = 0


    for line in text.split():
        preprocess_text = self.preprocess_line(line)
        for i in range(len(preprocess_text) - window_size + 1):
            trigrams.append(preprocess_text[i:i+window_size])

    log_sum = 0
    for trigram in trigrams:
        # Get the probability of the trigram
        probability = ngram_model_add_k.add_k_smoothing_probability(trigram, hyper_paramater) if model == "add_k" else ngram_model_interpolation.smoothed_trigram_probability(trigram, hyper_paramater)
        # Sum of the logs of corresponding probabilities
        log_sum = log_sum + math.log(probability,2)
        # Count the number of trigrams
        n = n + 1
    if n > 0:
        # Average negative log probability
        avg_log_prob = -(log_sum) / float(n)
    # Average negative log probability exponentiated to 2
    result = pow(2, avg_log_prob)
    return result
```

NgramModel.py:

```
def optimize_k(self, validation_corpus):
    """
    Function to optimise the value of K for alpha smoothing
    """
    k_values = [0.00001, 0.0001, 0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
    best_k = None
    best_perplexity = float('inf')

    for k in k_values:
        perplexity = self.perplexity(validation_corpus, "add_k", k)
        print("K value:", k, " Perplexity", perplexity)
        if(perplexity < best_perplexity):
            best_perplexity = perplexity
            best_k = k

    print('best k: ',best_k)


def optimize_lambdas(self, validation_corpus):
    """
    Function to find the lambda values with lower perplexity
    """
    lambda_values = [(0.9, 0.05), (0.8, 0.05), (0.8, 0.1), (0.7, 0.2), (0.7, 0.1), (0.6, 0.3), (0.4, 0.3)]

    best_lambda = None
    best_perplexity = float('inf')

    for value in lambda_values:
        perplexity = self.perplexity(validation_corpus, "interpolation", value)
        print("lambdas:", value, "perplexity: ", perplexity)
        if(perplexity < best_perplexity):
            best_perplexity = perplexity
            best_lambda = value

    print('best lambda:', best_lambda)
```

NgramTrainModel.py: This class is used for pre-processing:

```python
import re

class NgramTrainModel:
    def __init__(self, model_name):
        """
        Initialize the NgramTrainModel with the name of the model (training data file).

        Parameters:
        model_name - the name of the file containing the training data
        """
        self.model_name = model_name
        self.trigram_counts = {}
        self.bigram_counts = {}
        self.unigram_counts = {}

    def preprocess_line(self, line):
        """
        Preprocess the input line by keeping only alphabets, spaces, and periods.
        Convert digits to '0' and lowercase all characters.

        Parameters:
        line - the input sentence to preprocess
        """
        expression = r'[a-zA-Z\s\.]+|\d+'
        matches = re.findall(expression, line)
        processed_text = ''.join(re.sub('\d', '0', match.lower()) for match in matches)
        processed_text = re.sub(' ', "_", processed_text)
        return f'##{processed_text}#'

    def ngrams(self, output, input_tokens, n):
        """
        Generate n-grams from the input tokens and update the output dictionary with their counts.

        Parameters:
        output - dictionary to store n-gram counts
        input_tokens - list of tokens (processed characters)
        n - the size of n-grams to generate
        """
        for i in range(len(input_tokens) - n + 1):
            g = ''.join(input_tokens[i:i + n])  # Join n tokens to form the n-gram
            output.setdefault(g, 0)
            output[g] += 1
        return output
```

NgramTrainModel

```
def train_model(self):
    """
    Read the training data from the model file and process each line to generate unigram, bigram, and trigram counts.
    """
    with open(self.model_name, "r") as file:
        lines = file.readlines()

    # Process each line in the file
    for l_num in lines:
        process_sentence = self.preprocess_line(l_num.strip())  # Preprocess each line
        tokens = list(process_sentence)  # Convert the processed sentence into a list of characters (tokens)

        # Count the n-grams and update the respective dictionaries
        self.ngrams(self.trigram_counts, tokens, 3)
        self.ngrams(self.bigram_counts, tokens, 2)
        self.ngrams(self.unigram_counts, tokens, 1)

    return self.trigram_counts, self.bigram_counts, self.unigram_counts
```

RandomTextGenerator.py: This class is used to generate output from a language model:

```python
import string
import random
import re

class RandomTextGenerator:

    model_file_path = ""
    model_probabilities = {}
    starting_bigram = ""

    # Allowed characters from vocabulary
    characters = 'abcdefghijklmnopqrstuvwxyz0_.'

    def __init__(self, model_file_path, starting_bigram):
        """
        Initializes the model file path and starting bigram to start generation

        Parameters:
        model_file_path - Path to the model file
        starting_bigram - Starting bigram for generation of text
        """
        self.model_file_path = model_file_path
        self.starting_bigram = starting_bigram

        with open(model_file_path, "r") as f:
            for line in f:
                split_line = line.split()
                if(len(split_line) > 1):
                    key = split_line[0]
                    value = split_line[1]
                    self.model_probabilities[key] = value
                else:
                    print("Line not splittable", split_line)
```

RandomTextGenerator.py

```
def generate_from_lm(self, n):
    """
    Generate text given a language model

    Parameters:
    n - number of characters to generate
    """
    generated_string = self.starting_bigram
    starting_random_bigram = self.starting_bigram

    while len(generated_string)<300:
        # Filter out all potential trigrams given a starting bigram
        filtered_dict = {k: v for k, v in self.model_probabilities.items() if k.startswith(starting_random_bigram)}

        if(len(filtered_dict) > 0):
            v = list(filtered_dict.values())
            k = list(filtered_dict.keys())
            normalized_probs = [float(p) for p in filtered_dict.values()]

            # Weighted random pick of trigram based on probabilities
            predicted_trigram = random.choices(list(filtered_dict.keys()), normalized_probs, k=1)

            # The output string
            generated_string = generated_string + predicted_trigram[0][-1]
            starting_random_bigram = generated_string[-2:]

        else:
            # This condition signals end of a sentence. Reset the starting bigram to look for beginning of the next sentence
            print("No trigram found for", starting_random_bigram)
            starting_random_bigram = "##"

    # Replace '_' with space
    generated_string = re.sub('_', " ", generated_string)

    # Replace beginning and end of sentence markers
    generated_string = re.sub('##', " ", generated_string)
    generated_string = re.sub('#', "", generated_string)
    return generated_string
```

NgramAddKSmoothingModel.py: This class is used to calculate probabilities and generate a language model using alpha/add-k smoothing:

```
import math
import re
import itertools
import os

class NgramAddKSmoothingModel:
    def __init__(self, trigram_counts=None, bigram_counts=None, unigram_counts=None):
        """
        Initialize the NgramAddKSmoothingModel with the necessary attributes.

        Parameters:
        model_name - the name of the model (file containing training data)
        """
        self.trigram_counts = trigram_counts if trigram_counts is not None else {}
        self.bigram_counts = bigram_counts if bigram_counts is not None else {}
        self.unigram_counts = unigram_counts if unigram_counts is not None else {}


    def add_k_smoothing_probability(self, trigram, k=None):
        """
        Calculate trigram probability with add-k smoothing.

        Parameters:
        trigram - the trigram tuple (e.g., ('a', 'b', 'c'))
        """
        if(k is None):
            k=0.5
        if('#' in trigram[:-1]):
            total_vocabulary_size = 29  # Size of the vocabulary
        else:
            total_vocabulary_size = 30
        bigram_count = self.bigram_counts.get(trigram[:-1], 0)  # Get the bigram (first two elements of trigram)
        trigram_prob = (self.trigram_counts.get(trigram, 0) + float(k)) / (bigram_count + (k * total_vocabulary_size))
        return trigram_prob
```

NgramAddKSmoothingModel

```
def write_trigram_probabilities_to_file(self, characters, output_file):
    """
    Generate all possible trigrams from the list of characters and write their add-k smoothed
    probabilities to a file.

    Parameters:
    characters - list of characters to generate trigrams from
    output_file - name of the output file to save trigram probabilities
    """
    # trigrams = [''.join(gram) for gram in itertools.product(characters, repeat=3)]  # Generate all possible trigrams
    # trigram_probabilities = {}

    # with open(output_file, "w") as model_file:
    #     for gram in trigrams:
    #         # Calculate the k-smoothing probability
    #         probability = self.add_k_smoothing_probability(gram)  # Pass as a string
    #         trigram_probabilities[gram] = probability
    #         # Write the trigram and its probability to the file
    #         model_file.write(f"{gram}   {probability}\n")

    output_dir = "model"
    os.makedirs(output_dir, exist_ok=True)

    # Generate all possible trigrams from the given character set
    trigrams = [''.join(gram) for gram in itertools.product(characters, repeat=3)]
    filtered_trigrams = []

    for trigram in trigrams:
        # Exclude trigrams like a## (first char is anything, followed by ##)
        if trigram[1] == '#' and trigram[2] == '#':
            continue
        # Exclude trigrams like ###
        if trigram[0] == '#' and trigram[1] == '#' and trigram[2] == '#':
            continue
        # Exclude trigrams like a#a (same first and third char, # in the middle)
        if trigram[0] !='#' and trigram[2] != '# 'and trigram[1] == '#':
            continue
        # Exclude trigrams like #a# (first and third are #, anything in the middle)
        if trigram[0] == '#' and trigram[2] == '#':
            continue

        # Append valid trigrams
        filtered_trigrams.append(trigram)


    # Construct the full path to the output file inside the "model" folder
    output_path = os.path.join(output_dir, output_file)

    with open(output_path, "w") as model_file:
        for trigram in filtered_trigrams:
            # Calculate the smoothed trigram probability using the interpolation model
            probability = self.add_k_smoothing_probability(trigram)

            # Write the trigram and its probability to the file
            model_file.write(f"{trigram}\t{probability}\n")
```

InterpolationSmoothing.py: This class is used to calculate probabilities and generate a language model using interpolation:

```python
class InterpolationSmoothingModel:
    def __init__(self, trigram_counts, bigram_counts, unigram_counts):
        """
        Initialize the InterpolationSmoothingModel with the necessary n-gram counts.

        Parameters:
        trigram_counts - dictionary storing trigram counts
        bigram_counts - dictionary storing bigram counts
        unigram_counts - dictionary storing unigram counts
        """
        self.trigram_counts = trigram_counts
        self.bigram_counts = bigram_counts
        self.unigram_counts = unigram_counts



    def raw_trigram_probability(self, trigram):
        """
        Returns the raw trigram probability.
        """
        if self.bigram_counts.get(trigram[:2], 0) != 0:
            return self.trigram_counts.get(trigram, 0) / self.bigram_counts[trigram[:2]]
        else:
            return 0.0

    def raw_bigram_probability(self, bigram):
        """
        Returns the raw bigram probability.
        """
        if self.unigram_counts.get(bigram[0], 0) != 0:
            return self.bigram_counts.get(bigram, 0) / self.unigram_counts[bigram[0]]
        else:
            return 0.0

    def raw_unigram_probability(self, unigram):
        """
        Returns the raw unigram probability.
        """
        return self.unigram_counts.get(unigram, 0) / sum(self.unigram_counts.values())

    def smoothed_trigram_probability(self, trigram, hyper_parameter=None):
        """
        Returns the smoothed trigram probability using linear interpolation.
        """
        if(hyper_parameter is None):
            lambda1 = 0.7
            lambda2 = 0.1
            lambda3 = 0.2
        else:
            lambda1 = hyper_parameter[0]
            lambda2 = hyper_parameter[1]
            lambda3 = 1 - (lambda1 + lambda2)

        smoothed = 0.0
        smoothed += lambda1 * self.raw_trigram_probability(trigram)
        smoothed += lambda2 * self.raw_bigram_probability(trigram[1:])  # Trigram without the first character
        smoothed += lambda3 * self.raw_unigram_probability(trigram[2:])  # Trigram without the first two characters

        return smoothed
```

```python
def write_interpolation_smoothed_probabilities(self, characters, output_file):


# Generate all possible trigrams from the given character set
    # trigrams = [''.join(gram) for gram in itertools.product(characters, repeat=3)]

    # with open(output_file, "w") as model_file:
    #     for trigram in trigrams:
    #         # Calculate the smoothed trigram probability using the interpolation model
    #         probability = self.smoothed_trigram_probability(trigram)  # Pass trigram as a string

    #         # Write the trigram and its probability to the file
    #         model_file.write(f"{trigram}   {probability}\n")
    output_dir = "model"
    os.makedirs(output_dir, exist_ok=True)

    # Generate all possible trigrams from the given character set
    trigrams = [''.join(gram) for gram in itertools.product(characters, repeat=3)]

    filtered_trigrams = []

    for trigram in trigrams:
        # Exclude trigrams like a## (first char is anything, followed by ##)
        if trigram[1] == '#' and trigram[2] == '#':
            continue
        # Exclude trigrams like ###
        if trigram[0] == '#' and trigram[1] == '#' and trigram[2] == '#':
            continue
        # Exclude trigrams like a#a (same first and third char, # in the middle)
        if trigram[0] !='#' and trigram[2] != '# 'and trigram[1] == '#':
            continue
        # Exclude trigrams like #a# (first and third are #, anything in the middle)
        if trigram[0] == '#' and trigram[2] == '#':
            continue

        # Append valid trigrams
        filtered_trigrams.append(trigram)


    # Construct the full path to the output file inside the "model" folder
    output_path = os.path.join(output_dir, output_file)

    with open(output_path, "w") as model_file:
        for trigram in filtered_trigrams:
            # Calculate the smoothed trigram probability using the interpolation model
            probability = self.smoothed_trigram_probability(trigram)

            # Write the trigram and its probability to the file
            model_file.write(f"{trigram}\t{probability}\n")
```