Pset_2_Tuning_ngram_model

April 30, 2023

1 Tuning an n-gram model

The following is scaffolding code that you can expand to complete the problem. First, we set up the training, validation, and test datasets (for real-size modeling problems you would read these from files):

[64]:
```
# imports
import numpy as np
import warnings
from copy import deepcopy
from matplotlib import pyplot as plt
from math import log2

warnings.filterwarnings("ignore")
```

[65]:
```
training_set = [['dogs', 'chase', 'cats'],
['dogs', 'bark'],
['cats', 'meow'],
['dogs', 'chase', 'birds'],
['birds', 'chase', 'cats'],
['dogs', 'chase', 'the', 'cats'],
['the', 'birds', 'chirp']]

validation_set = [['the','cats','meow'],
                  ['the','dogs','bark'],
                  ['the','dogs','chase','the','cats']]

test_set = [['cats','meow'],['dogs','chase','the','birds']]
```

A natural way to implement a model is often to define a class that you can give model hyperparameters, and define methods for training the model, computing the most basic building-block quantity relevant for the model, and assessing overall performance of a trained model on a dataset. Below is scaffolding code for doing this. For a bigram model, the most elementary quantity is $p(w_i|w_{i-1})$ so that is what the `prob()` method gives.

You don't need to use this scaffolding code in your solution to the problem, but you may find it useful.

[66]:
```python
##TODO
class bigram_model:
  def __init__(self, alpha):
    self.alpha = alpha

    self.word_count_dict = {}
    self.pair_count_dict = {}

  def train(self, training_set):
    # your code goes here
    for sentence in training_set:
      for i in range(len(sentence)):
        if sentence[i] not in self.word_count_dict:
          self.word_count_dict[sentence[i]] = 1
        else:
          self.word_count_dict[sentence[i]] += 1

        if i != 0:
          if (sentence[i-1], sentence[i]) not in self.pair_count_dict:
            self.pair_count_dict[(sentence[i-1], sentence[i])] = 1
          else:
            self.pair_count_dict[(sentence[i-1], sentence[i])] += 1

    # <s> is not considered a word as mentioned on the forums, so we will not
    # count it in the vocabulary size
    self.M = len(self.word_count_dict.keys()) - 1


  # this function will be used in the 'bonus' question only
  def peak(self, val_set, test_set):
    """
    Gets validation set and test set and updates the vocabulary
    if those sets contain new words (that weren't seen while training).
    """
    # duplicate the list of words we saw while training
    vocabulary_dup = deepcopy(self.word_count_dict)

    # iterating through all words in each set and updating
    # the vocabulary when seeing new ones
    for sentence in val_set:
      for word in sentence:
        if word not in vocabulary_dup:
          vocabulary_dup[word] = 0

    for sentence in test_set:
      for word in sentence:
        if word not in vocabulary_dup:
          vocabulary_dup[word] = 0

    # <s> is not considered a word as mentioned on the forums, so we will not
    # count it in the vocabulary size
    self.M = len(vocabulary_dup.keys()) - 1


  def prob(self, previous_word, next_word):
    # your code goes here
    c1 = self.alpha
    if (previous_word, next_word) in self.pair_count_dict:
      c1 += self.pair_count_dict[(previous_word, next_word)]

    c2 = self.alpha * self.M
    if previous_word in self.word_count_dict:
      c2 += self.word_count_dict[previous_word]

    return c1 / c2


  def perplexity(self, heldout_set):
    """
    Calculate the perplexity according to the formula
    we proved in question 2.
    """
    N, dataset_perp= 0, 0

    for sentence in heldout_set:
      log_prob_sum = 0

      for i in range(1, len(sentence)):
        p = self.prob(sentence[i-1], sentence[i])

        log_prob_sum += log2(1/p)
        N += 1

      dataset_perp += log_prob_sum

    dataset_perp = 2 ** ((1/N) * dataset_perp)
    return dataset_perp
```

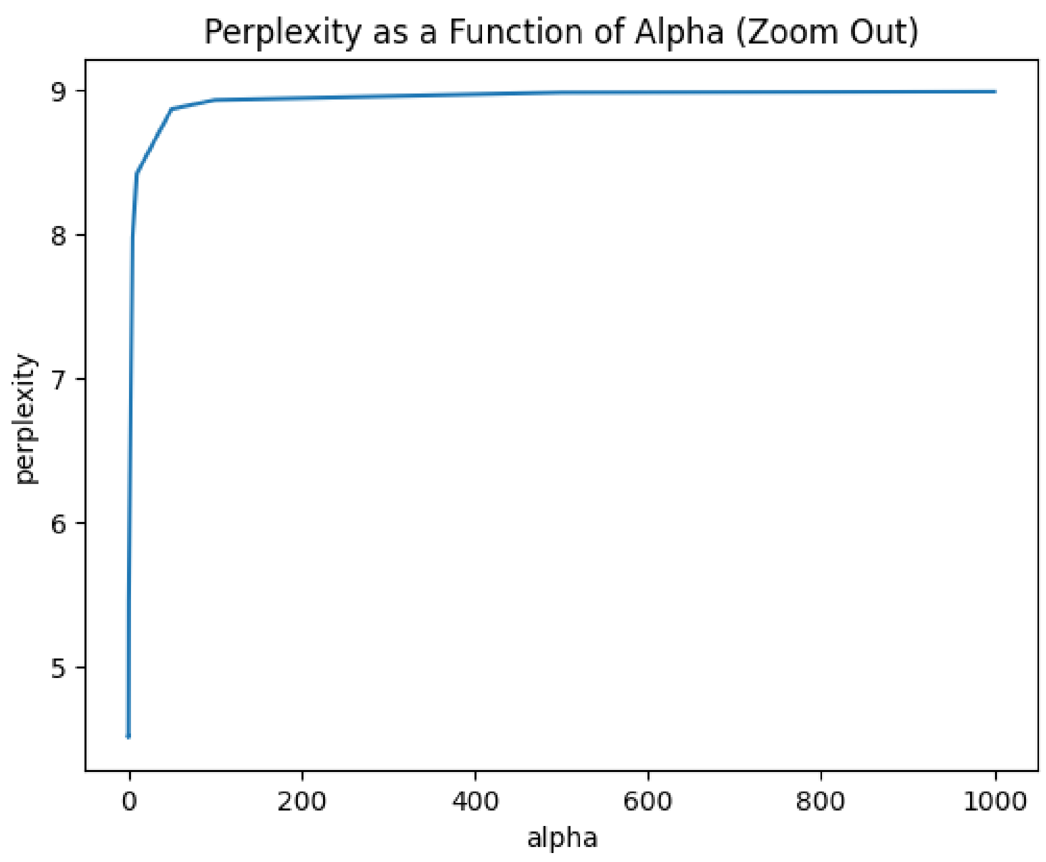
For step 1 of this problem, you need to find the value of α that optimizes validation-set perplexity; this is a simple example of what in machine learning these days "hyperparameter tuning" or "hyperparameter optimization".

[67]:
```
##TODO: your code for step 1 goes here
# transform data
```

```
def transform_dataset(dataset):
  trans_dataset = deepcopy(dataset)

  # add tokens
  for sentence in trans_dataset:
    sentence.insert(0, '<s>')
    sentence.append('</s>')

  return trans_dataset

trans_train_set = transform_dataset(training_set)
trans_val_set = transform_dataset(validation_set)
trans_test_set = transform_dataset(test_set)
```

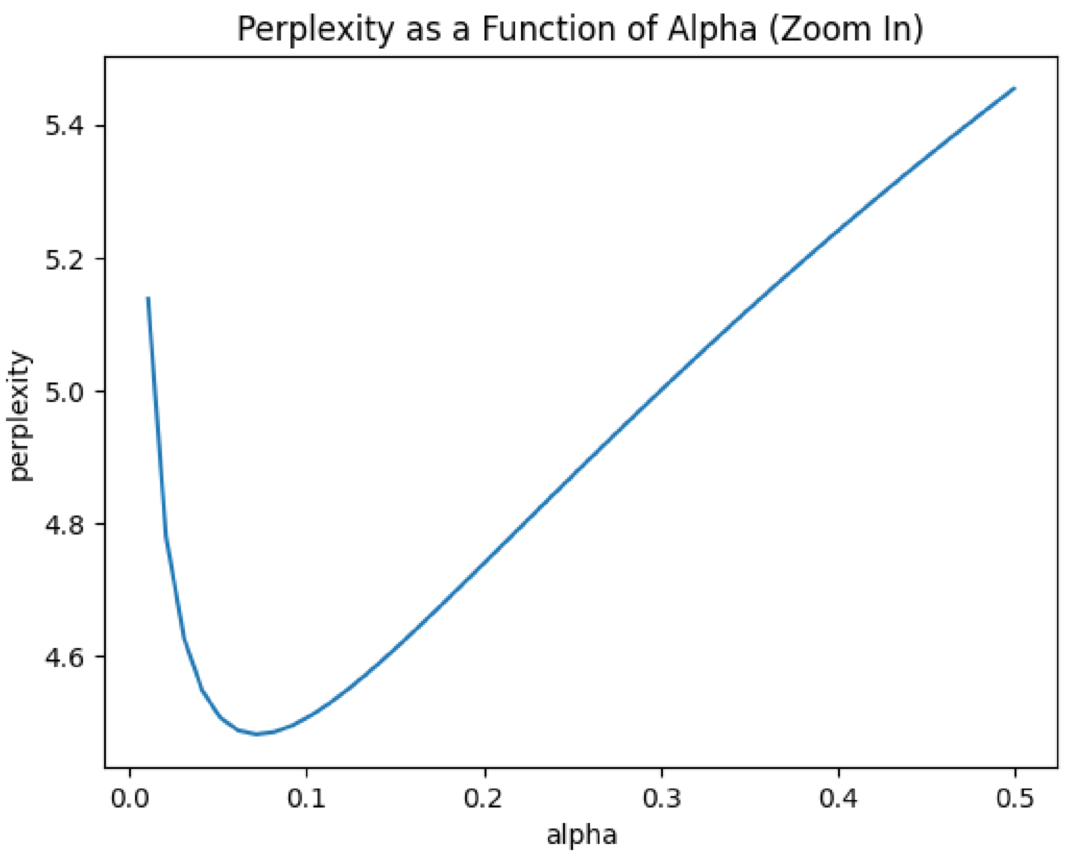
[68]:
```
def calc_perplexity(alpha, is_bonus=False):
  bgm = bigram_model(alpha)
  bgm.train(trans_train_set)

  if is_bonus:
    bgm.peak(trans_val_set, trans_test_set)

  return bgm.perplexity(trans_val_set)

def plot_perplexity(alphas, zoom, is_bonus=False):
  """
  Get a list of alpha values and plot the bigram model's (with alpha smoothing)
  perplexity as a function of alpha.
  """
  perplexities = []

  for a in alphas:
    perplexity = calc_perplexity(a, is_bonus)
    perplexities.append(perplexity)

  plt.plot(alphas, perplexities)
  plt.title(f"Perplexity as a Function of Alpha ({zoom})")
  plt.xlabel("alpha")
  plt.ylabel("perplexity")
  plt.show()
```

[69]:
```
# choosing possible values of alpha to search through
alphas = [0.05, 0.1, 0.5, 5, 10, 50, 100, 500, 1000]
plot_perplexity(alphas, zoom="Zoom Out")
```



As we can see, the lower alpha is - the lower the perplexity. Now, we will "zoom in" to see how the perplexity behaves closer to 0.

[70]:
```
# decide on a range of alpha values: [0, 0.5]
alphas = np.linspace(0, 0.5, 50)
plot_perplexity(alphas, zoom="Zoom In")
```



As we can see, the perplexity function is unimodal w.r.t alpha (in the range of [0, 0.5]). Therefore, we will use the "golden section" method to approximate the optimal value of alpha with percision of 4 decimal points.

```
[71]: def generic_gs(f, l, u, eps, is_bonus=False):
          """
          Find an estimation for the x value of the minimum of f in [l, u], and the␣
       ↪series f values found in the process.

          :param f: the target function
          :param l: lower bound
          :param u: upper bound
          :param eps: constant to measure accuracy
          :is_bonus: boolean of whether the function is being used for the bonus␣
       ↪question
          :return: value of estimation for minimum of f in [l, u], and a list␣
       ↪containing the series of f values
          """
          # c is 'tao' (golden ratio)
          c = (-1 + np.sqrt(5)) / 2
```

```
    x2 = c * l + (1 - c) * u
    fx2 = f(x2, is_bonus)
    x3 = (1 - c) * l + c * u
    fx3 = f(x3, is_bonus)

    while np.abs(u - l) >= eps:
        if fx2 < fx3:
            u = x3
            x3 = x2
            fx3 = fx2
            x2 = c * l + (1 - c) * u
            fx2 = f(x2, is_bonus)

        else:
            l = x2
            x2 = x3
            fx2 = fx3
            x3 = (1 - c) * l + c * u
            fx3 = f(x3, is_bonus)

    return l
```

[72]:
```
best_alpha = generic_gs(calc_perplexity, l=0, u=0.5, eps=10**(-4))
print(f"Alpha that minimizes the held-out perplexity of \
the validation set: {best_alpha:.4f}")
print(f"Minimal perplexity: {calc_perplexity(best_alpha):.4f}")
```

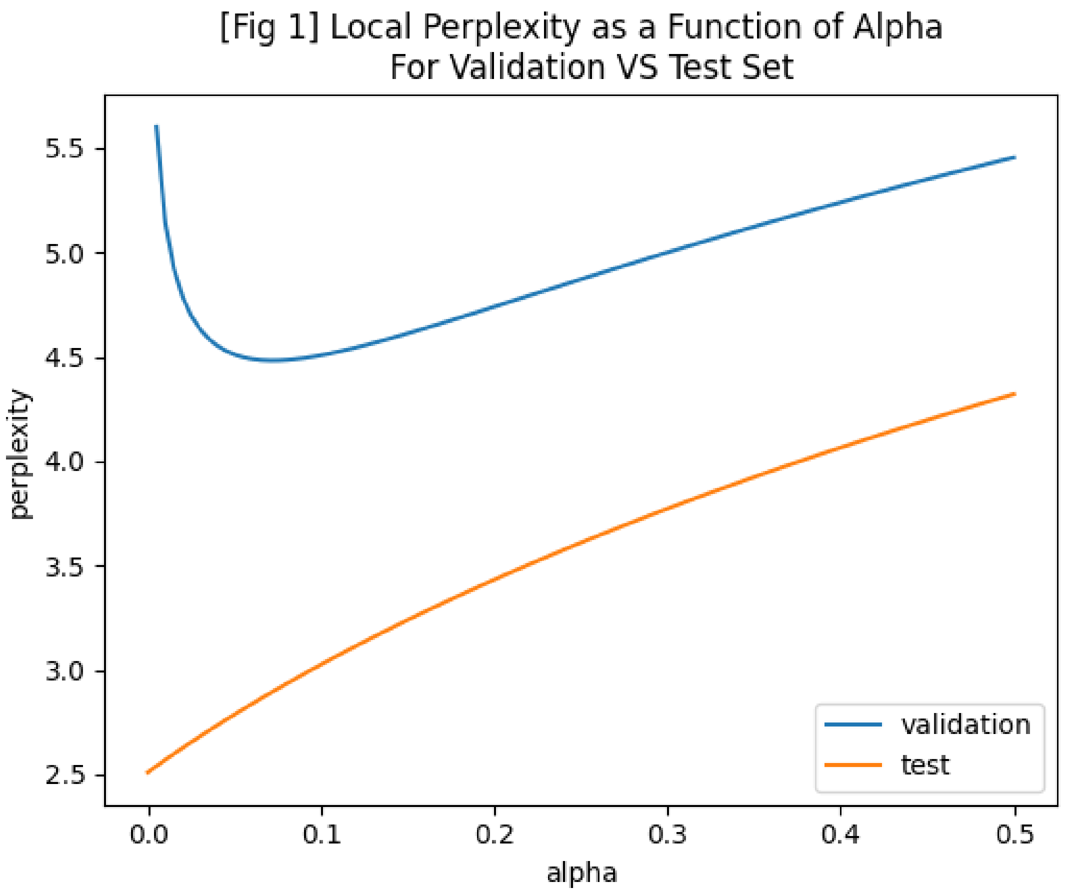
```
Alpha that minimizes the held-out perplexity of the validation set: 0.0721
Minimal perplexity: 4.4824
```

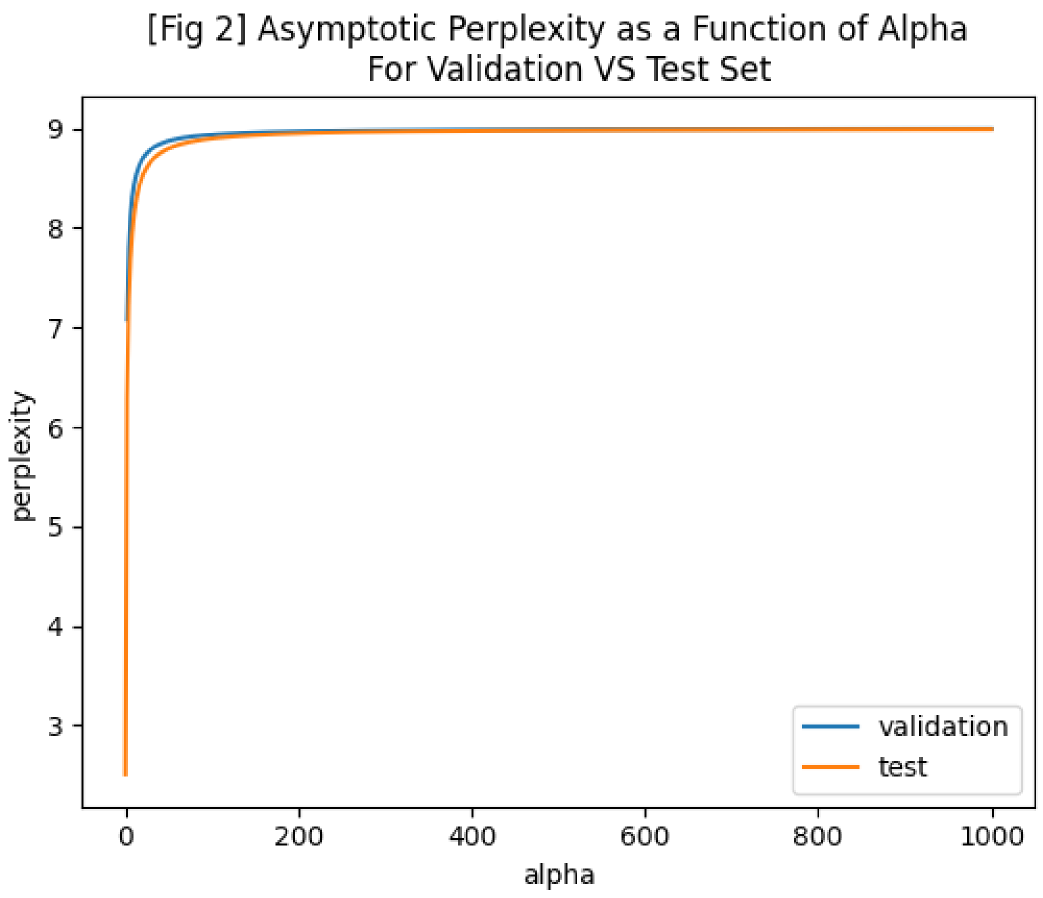
Thus, the optimal alpha is: $\alpha = 0.0721$

For step 2, write a function which returns the perplexities of the validation and test sets for a given alpha. Next, graph in a lineplot the validation and test set perplexities for a range of alphas that reveals the full relationship between validation and test set perplexities.

[73]:
```
##TODO
def perplexities_from_alpha(alpha, is_bonus=False):
  ##your code here
  bgm = bigram_model(alpha)
  bgm.train(trans_train_set)

  if is_bonus:
    bgm.peak(trans_val_set, trans_test_set)

  validation_perplexity = bgm.perplexity(trans_val_set)
  test_perplexity = bgm.perplexity(trans_test_set)
  return((validation_perplexity, test_perplexity))


## your code for graphing the perplexities here
def plot_perplexities(alphas, val_perplexities, test_perplexities, st):
  plt.plot(alphas, val_perplexities, label="validation")
  plt.plot(alphas, test_perplexities, label="test")
  plt.legend()
  plt.xlabel("alpha")
  plt.ylabel("perplexity")

  if st == "Local":
    fig = "[Fig 1]"
  else:
    fig = "[Fig 2]"

  plt.title(f"{fig} {st} Perplexity as a Function of Alpha\n\
  For Validation VS Test Set")
  plt.show()
```

[74]:
```
# take a look at the local behaviour of the perplexity (as a function of alpha)
alphas = np.linspace(0, 0.5, 100)
val_perplexities, test_perplexities = [], []

for a in alphas:
  val_perp, test_perp = perplexities_from_alpha(a)
  val_perplexities.append(val_perp)
  test_perplexities.append(test_perp)

plot_perplexities(alphas, val_perplexities, test_perplexities, st="Local")

# check also for the asymptotic behaviour of the perplexity
alphas = np.linspace(0, 1000, 500)
val_perplexities, test_perplexities = [], []

for a in alphas:
  val_perp, test_perp = perplexities_from_alpha(a)
  val_perplexities.append(val_perp)
  test_perplexities.append(test_perp)

plot_perplexities(alphas, val_perplexities, test_perplexities, st="Asymptotic")
```


[Fig 1] Local Perplexity as a Function of Alpha
For Validation VS Test Set
perplexity
alpha
validation
test
5.5
5.0
4.5
4.0
3.5
3.0
2.5
0.0
0.1
0.2
0.3
0.4
0.5



We chose to display Fig 1 in order to see the behavior of the perplexity on the test set around the minimum point of the perplexity on the validation set. We saw that the perplexity of the test set is monotonically increasing (w.r.t alpha) in this section, thus we chose to display Fig 2 to show that the behavior we saw in Fig 1 remains the same (asymptotically).

1.0.1 Interpret the results

- What value of worked the best for the validation set?
- Was it the same that would have worked best for the test set?

TODO:

As we saw in sub-question 1, the value of alpha that worked best for the validation set is 0.0721 (with precision of 4 decimal points).

In addition, we can see from the Fig 1 that the best alpha value for the validation set (0.0721) does not work best for the test set.

From both Fig 1 and Fig 2 we can see that the perplexity of the test set is monotonically increasing (w.r.t alpha). Thus, the optimal value of alpha for the test set is 0. In other words, we get the best perplexity on the test set when our language model does not incorporate alpha-smoothing at all.

Moreover, for alpha values greater than the optimal value for the validation set, both functions behave the same (monotonically increasing), and asymptoically as well.

The Meaning of the Results

To reiterate our findings: the best perplexity for the test set is received when alpha is 0, meaning there is no alpha smoothing at all.

As we can see in the given data sets, the test set is "contained" in the training set (all bigram combinations that appear in the test set also appear in the training set). Thus, we conclude:

- Feasibility - As seen in the lectures, smoothing helps us deal with unseen bigram combinations. However, the test set is "contained" in the training set, which allows zero to be a feasible value of alpha.
- Optimality - As seen in the lecture, smoothing can be viewed as a form of regularization - it distributes probability mass to unseen bigram combinations which helps us tackle them. However, the test set is "contained" in the training set. Therefore, there aren't any unseen bigram combinations. Thus, the distribution of probability mass to irrelevant bigram combinations (that do not appear in the test set anyway) deducts from the probabilities learned in the training phase. This increases the perplexity of the model, and thus better be avoided ($\alpha = 0$).

1.1 Bonus Question

In this part, we will repeat our solution for the question above, but for a larger datasets ('Wikitext-2' and 'Wikitext-103').

1.1.1 Wikitext-2 Dataset

Installing the Data

[75]:
```python
import re
import zipfile
from pathlib import Path
```

[76]:
```python
with zipfile.ZipFile('wikitext-2-v1.zip', 'r') as zip_ref:
    zip_ref.extractall('/content/')

# Read train, val, and test sets into string objects
train_data = Path('/content/wikitext-2/wiki.train.tokens').read_text()
val_data = Path('/content/wikitext-2/wiki.valid.tokens').read_text()
test_data = Path('/content/wikitext-2/wiki.test.tokens').read_text()
```

Cleaning the Data

[77]:
```python
# Store regular expression pattern to search for wikipedia article headings
heading_pattern = '( \n \n = [^=]*[^=] = \n \n )'

# Split out train headings and articles
```

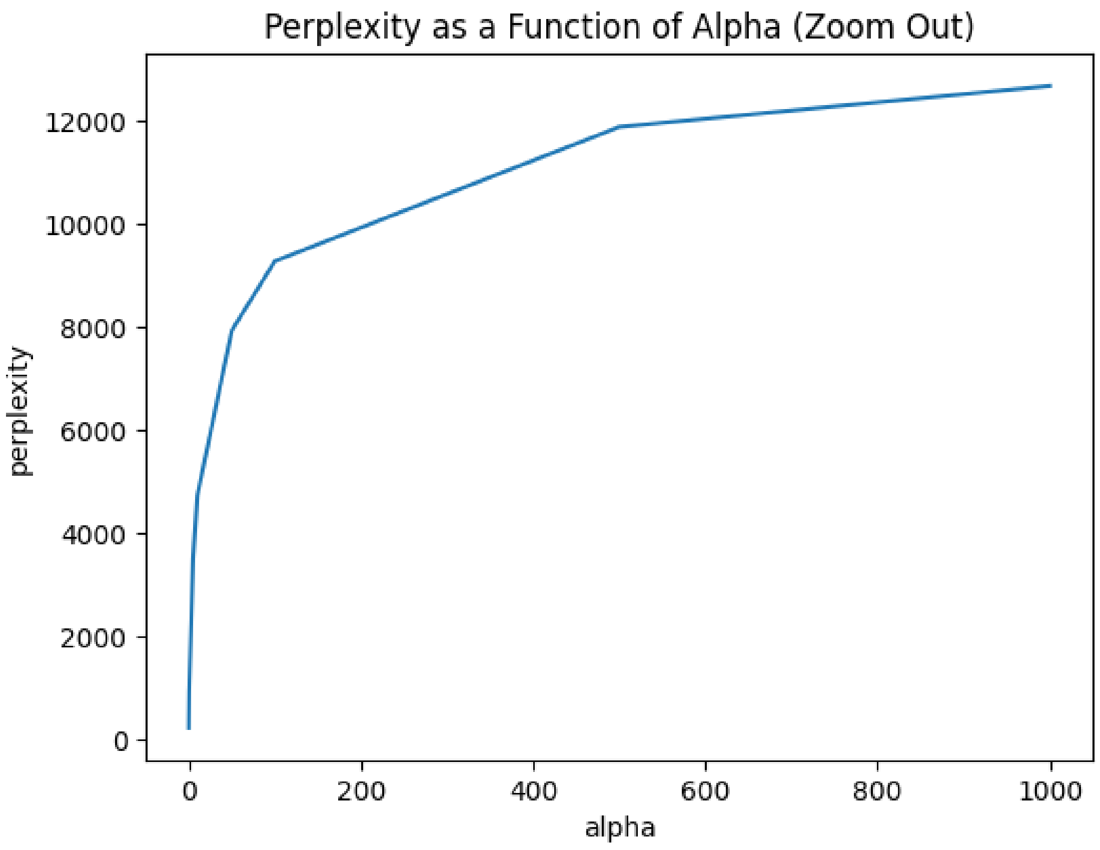
```
train_split = re.split(heading_pattern, train_data)
train_headings = [x[7:-7] for x in train_split[1::2]]
train_articles = [x for x in train_split[2::2]]

# Split out validation headings and articles
val_split = re.split(heading_pattern, val_data)
val_headings = [x[7:-7] for x in val_split[1::2]]
val_articles = [x for x in val_split[2::2]]

# Split out test headings and articles
test_split = re.split(heading_pattern, test_data)
test_headings = [x[7:-7] for x in test_split[1::2]]
test_articles = [x for x in test_split[2::2]]
```

Extracting the Datasets

[78]:
```
def extract_dataset(articles_lst):
  """
  Gets a list of articles and returns a dataset (corpus), in the
  form of a list of sentences (and each sentence is a list of its words).
  """
  dataset = []

  for article in test_articles:
    sentences_in_article = article.split(" . ")

    for sentence in sentences_in_article:
      sentence_lst = sentence.split(" ")
      dataset.append(sentence_lst)

  return dataset

training_set = extract_dataset(train_articles)
validation_set = extract_dataset(val_articles)
test_set = extract_dataset(test_articles)
```

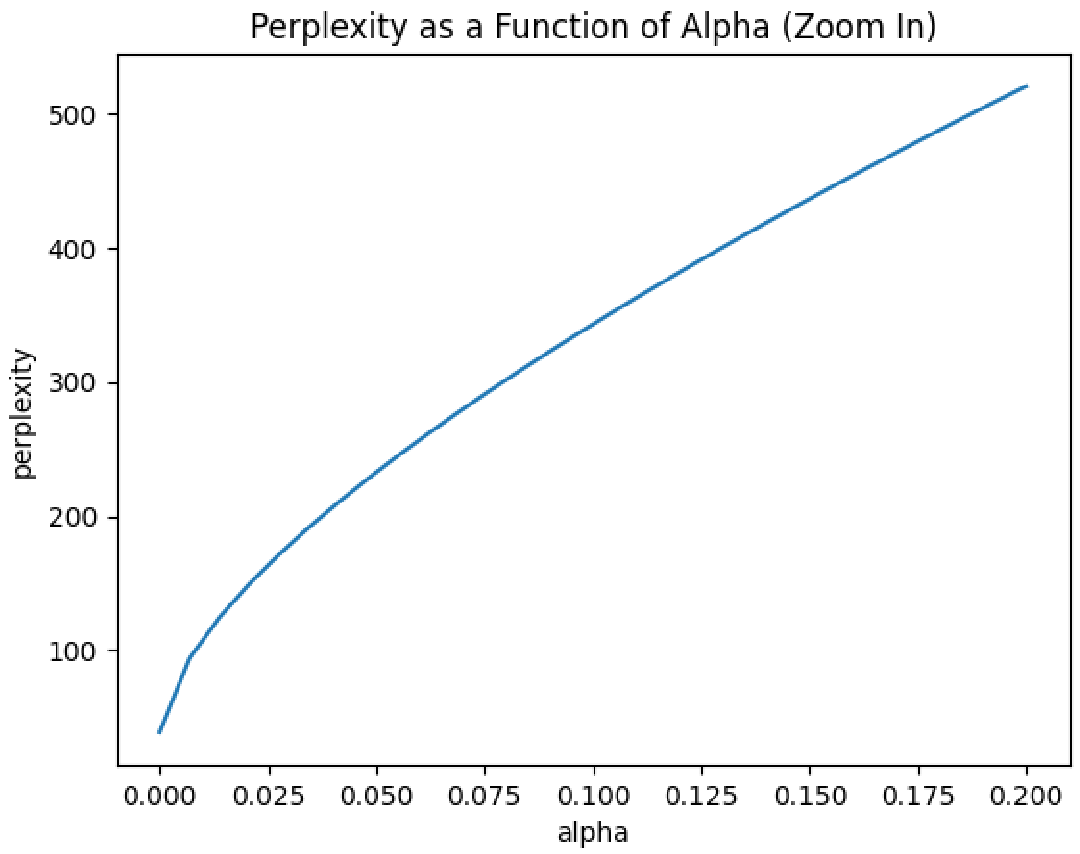
[79]:
```
# transforming the datasets
trans_train_set = transform_dataset(training_set)
trans_val_set = transform_dataset(validation_set)
trans_test_set = transform_dataset(test_set)
```

[80]:
```
# choosing possible values of alpha to search through
alphas = [0.05, 0.1, 0.5, 5, 10, 50, 100, 500, 1000]
plot_perplexity(alphas, zoom="Zoom Out", is_bonus=True)
```



As we can see, the lower alpha is - the lower the perplexity. Now, we will "zoom in" to see how the perplexity behaves closer to 0.

[81]:
```python
# decide on a range of alpha values: [0, 0.2]
alphas = np.linspace(0, 0.2, 30)
plot_perplexity(alphas, zoom="Zoom In", is_bonus=True)
```



As we can see, the perplexity function is unimodal w.r.t alpha (in the range of [0, 0.2]). Therefore, we will use the "golden section" method to approximate the optimal value of alpha with percision of 4 decimal points.

[82]:
```
best_alpha = generic_gs(calc_perplexity, l=0, u=0.2, eps=10**(-4),␣
 ↪is_bonus=True)
print(f"Alpha that minimizes the held-out perplexity of \
the validation set: {best_alpha:.4f}")
print(f"Minimal perplexity: {calc_perplexity(best_alpha, is_bonus=True):.4f}")
```

```
Alpha that minimizes the held-out perplexity of the validation set: 0.0000
Minimal perplexity: 39.1814
```

Thus, the optimal alpha is: $\alpha = 0$

1.1.2 Wikitext-103 Dataset

Installing the Data

[83]:
```
with zipfile.ZipFile('wikitext-103-v1.zip', 'r') as zip_ref:
    zip_ref.extractall('/content/')
```

```
# Read train, val, and test sets into string objects
train_data = Path('/content/wikitext-103/wiki.train.tokens').read_text()
val_data = Path('/content/wikitext-103/wiki.valid.tokens').read_text()
test_data = Path('/content/wikitext-103/wiki.test.tokens').read_text()
```

Cleaning the Data

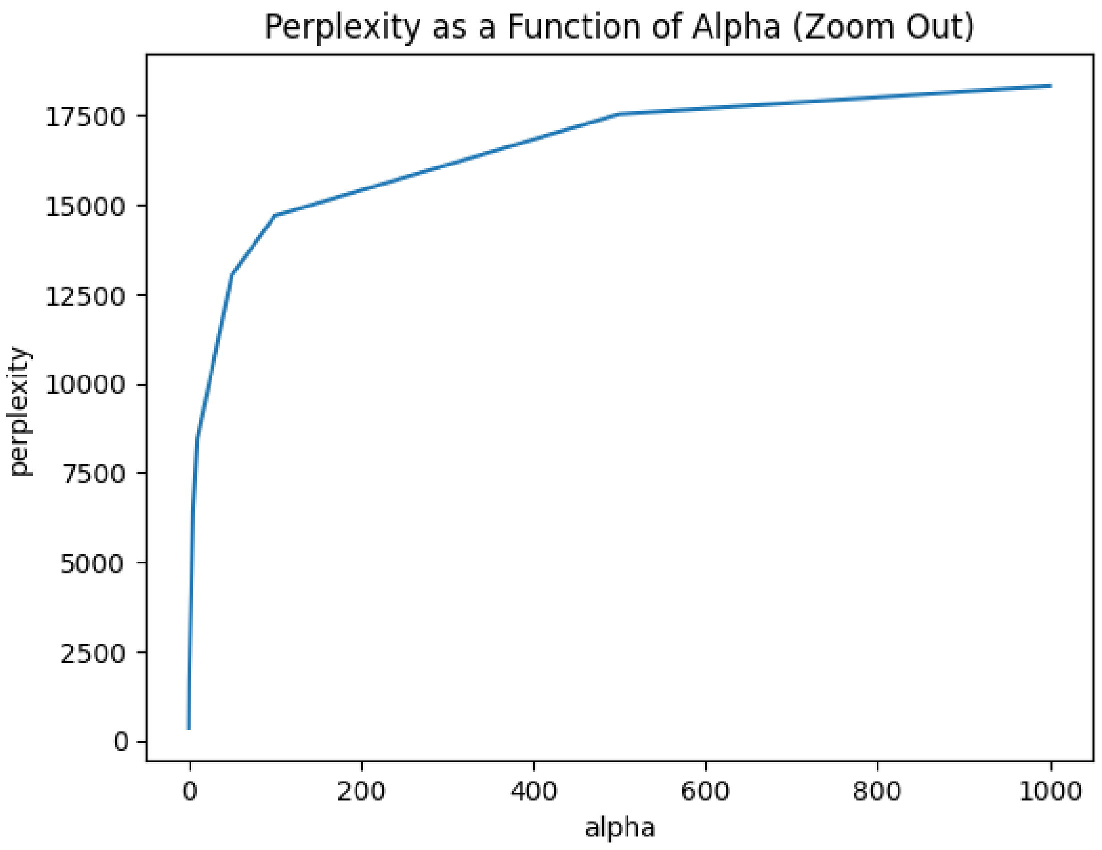
[84]:
```
# Store regular expression pattern to search for wikipedia article headings
heading_pattern = '( \n \n = [^=]*[^=] = \n \n )'

# Split out train headings and articles
train_split = re.split(heading_pattern, train_data)
train_headings = [x[7:-7] for x in train_split[1::2]]
train_articles = [x for x in train_split[2::2]]

# Split out validation headings and articles
val_split = re.split(heading_pattern, val_data)
val_headings = [x[7:-7] for x in val_split[1::2]]
val_articles = [x for x in val_split[2::2]]

# Split out test headings and articles
test_split = re.split(heading_pattern, test_data)
test_headings = [x[7:-7] for x in test_split[1::2]]
test_articles = [x for x in test_split[2::2]]
```

Extracting the Datasets

[85]:
```
training_set = extract_dataset(train_articles)
validation_set = extract_dataset(val_articles)
test_set = extract_dataset(test_articles)
```

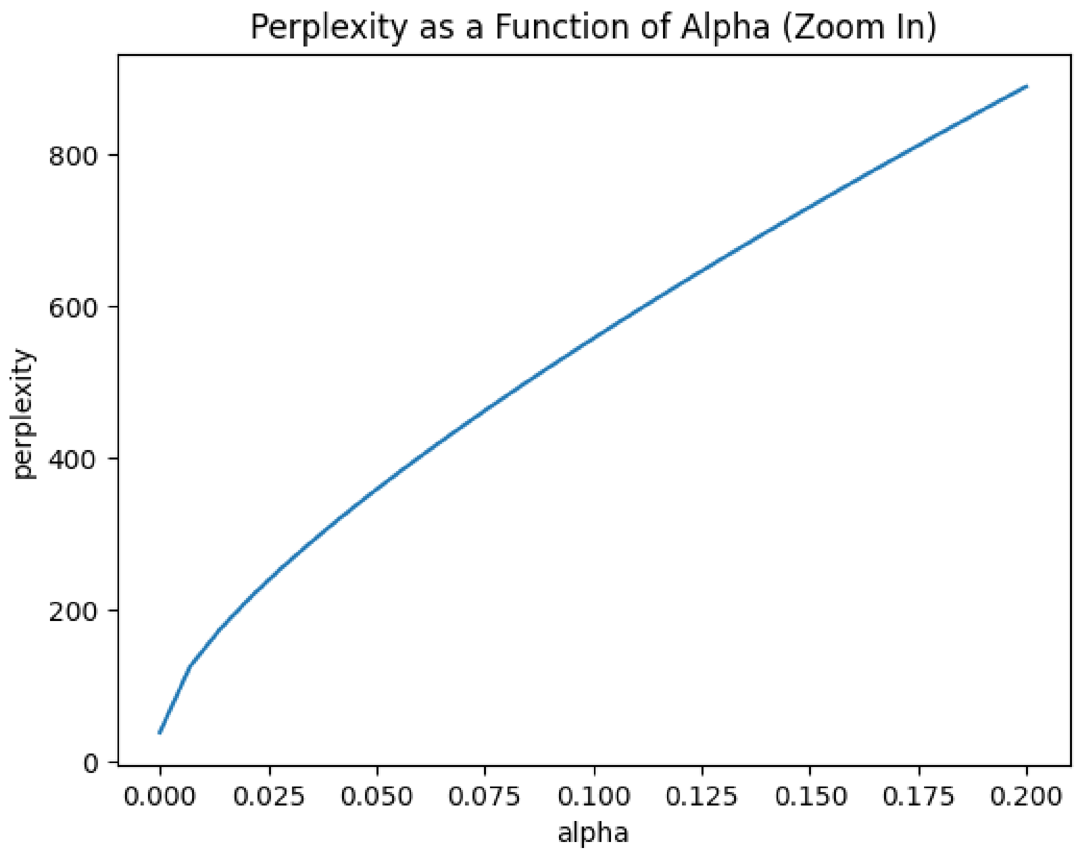
[86]:
```
# transforming the datasets
trans_train_set = transform_dataset(training_set)
trans_val_set = transform_dataset(validation_set)
trans_test_set = transform_dataset(test_set)
```

[87]:
```
# choosing possible values of alpha to search through
alphas = [0.05, 0.1, 0.5, 5, 10, 50, 100, 500, 1000]
plot_perplexity(alphas, zoom="Zoom Out", is_bonus=True)
```



As we can see, the lower alpha is - the lower the perplexity. Now, we will “zoom in” to see how the perplexity behaves closer to 0.

[88]:
```python
# decide on a range of alpha values: [0, 0.2]
alphas = np.linspace(0, 0.2, 30)
plot_perplexity(alphas, zoom="Zoom In", is_bonus=True)
```



As we can see, the perplexity function is unimodal w.r.t alpha (in the range of [0, 0.2]). Therefore, we will use the "golden section" method to approximate the optimal value of alpha with percision of 4 decimal points.

[89]:
```
best_alpha = generic_gs(calc_perplexity, l=0, u=0.2, eps=10**(-4),␣
  ↪is_bonus=True)
print(f"Alpha that minimizes the held-out perplexity of \
the validation set: {best_alpha:.4f}")
print(f"Minimal perplexity: {calc_perplexity(best_alpha, is_bonus=True):.4f}")
```

```
Alpha that minimizes the held-out perplexity of the validation set: 0.0000
Minimal perplexity: 38.0902
```

Thus, the optimal alpha is: $\alpha = 0$

2 Export to PDF

Run the following cell to download the notebook as a nicely formatted pdf file.

```
[ ]: # Add to a new cell at the end of the notebook and run the follow code,
     # which will save the notebook as pdf in your google drive (allow the␣
      ↪permissions) and download it automatically.

     !wget -nc https://raw.githubusercontent.com/scaperex/colab-pdf/master/colab_pdf.
      ↪py

     from colab_pdf import colab_pdf

     # If you saved the notebook in the default location in your Google Drive,
     #  and didn't change the name of the file, the code should work as is. If not,␣
      ↪adapt accordingly.
     # E.g. in your case the file name may be "Copy of XXXX.ipynb"

     colab_pdf(file_name='Pset_2_Tuning_ngram_model.ipynb', notebookpath="drive/
      ↪MyDrive/Colab Notebooks")
```

```
File ‘colab_pdf.py’ already there; not retrieving.


WARNING: apt does not have a stable CLI interface. Use with caution in scripts.
```