This executable notebook will help you complete parts of Pset 2:

1. Introductory code for making plots in Python;
2. Scaffolding code for the n-gram model tuning problem.

If you haven't used Colab before, it's very similar to Jupyter / IPython / R Notebooks: cells containing Python code can be interactively run, and their outputs will be interpolated into this document. If you haven't used any such software before, we recommend [taking a quick tour of Colab (https://colab.research.google.com/notebooks/basic_features_overview.ipynb)](https://colab.research.google.com/notebooks/basic_features_overview.ipynb).

Now, a few Colab-specific things to note about execution before we get started:

- Google offers free compute (including GPU compute!) on this notebook, but *only for a limited time*. Your session will be automatically closed after 12 hours. That means you'll want to finish within 12 hours of starting, or make sure to save your intermediate work (see the next bullet).
- You can save and write files from this notebook, but they are *not guaranteed to persist*. For this reason, we'll mount a Google Drive account and write to that Drive when any files need to be kept permanently (e.g. model checkpoints, surprisal data, etc.).
- You should keep this tab open until you're completely finished with the notebook. If you close the tab, your session will be marked as "Idle" and may be terminated.

Getting started

First, make a copy of this notebook so you can make your own changes. Click *File -> Save a copy in Drive*.

What you need to do

Read through this notebook and execute each cell in sequence, making modifications and adding code where necessary. You should execute all of the code as instructed, and make sure to write code or textual responses wherever the text **TODO** shows up in text and code cells.

When you're finished, download the notebook as a PDF file by running the script in the last cell, or alternatively download it as an .ipynb file and locally convert it to PDF.

Making plots in python

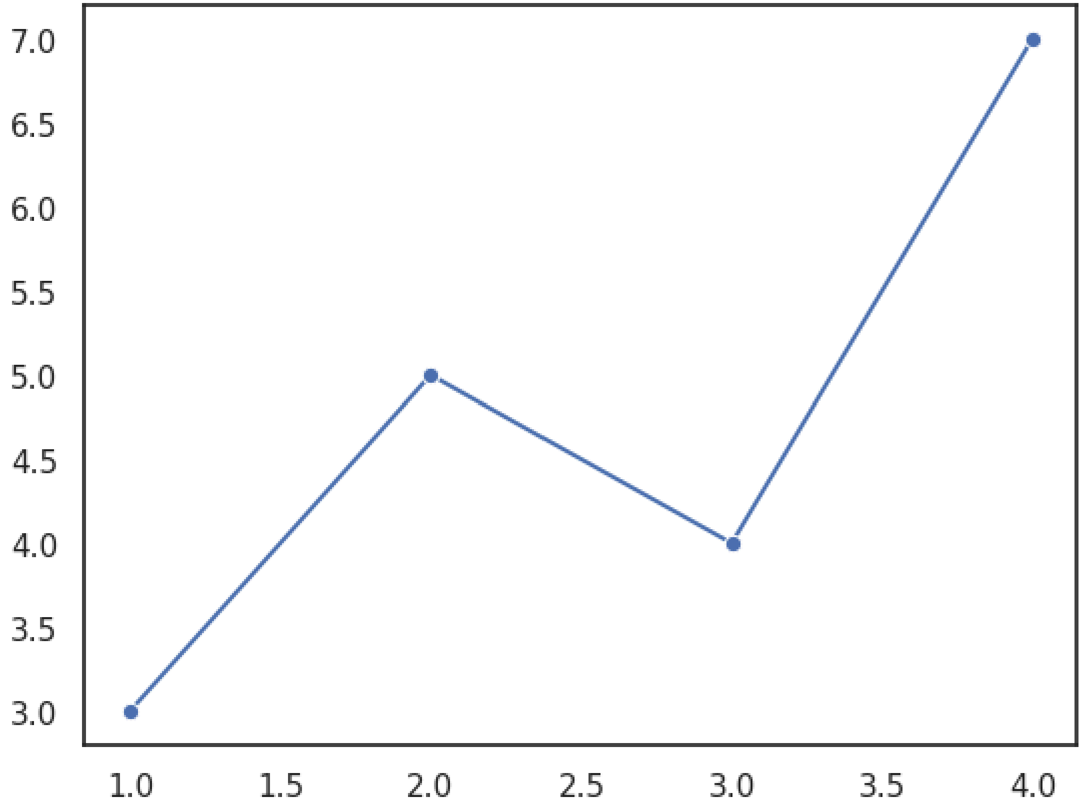
This code may help you with Part 2 of the "Tuning an n-gram language model" problem (plotting validation-set perplexity against test-set

In [ ]:

```
# import relevant libraries
import matplotlib.pyplot as plt
import seaborn as sns; sns.set(style="white", color_codes=True)

x_variable = [1,2,3,4]
y_variable = [3,5,4,7]
sns.lineplot(x=x_variable,y=y_variable,marker='o') # marker='o' adds points to the line graph
```

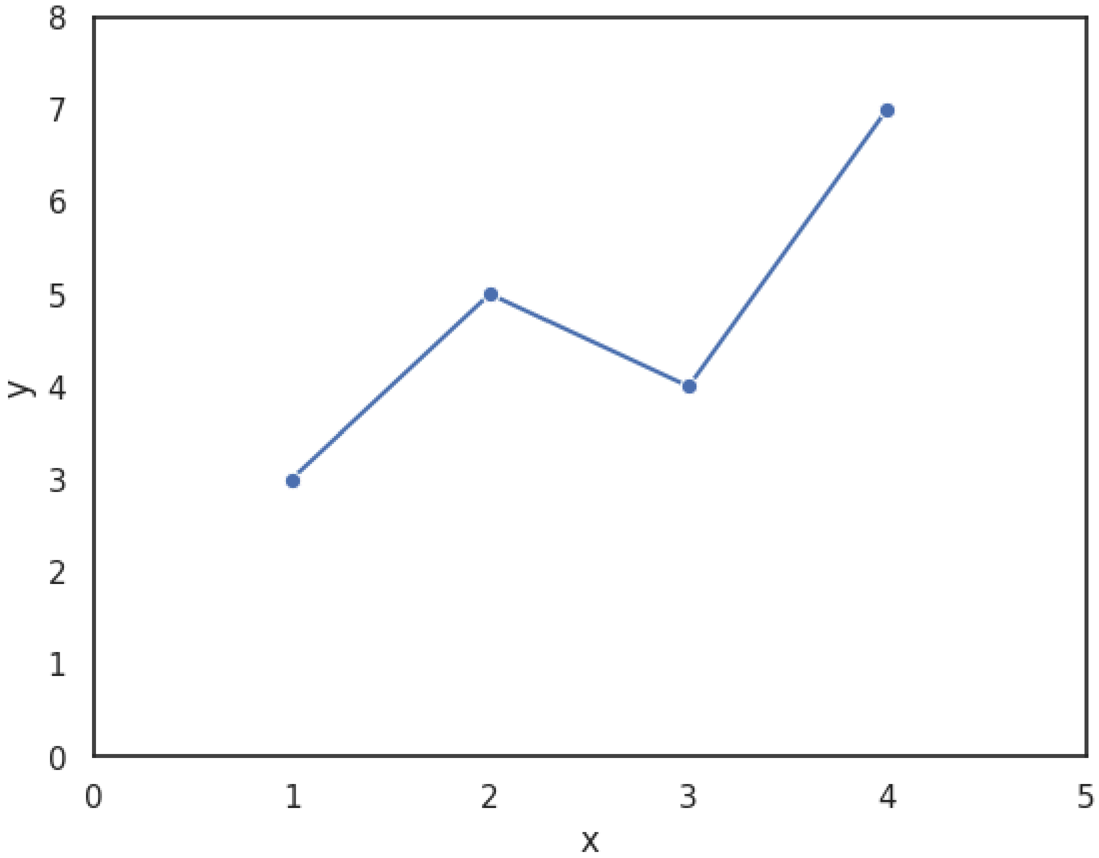
Out[36]:

```
<Axes: >
```



The first thing that you see in the output, `<matplotlib.axes._subplots.AxesSubplot at 0x7f68a607e550>` (the hexadecimal number at the end may be different for you), is a string representation of the return value of the last line of code in the code cell. In `seaborn`, the return value of plotting statements like `lineplot()`, is an object that can be subsequently modified to change the plot. In the below example, we assign this return value to a variable name, and then change the object to enrich the graph, using the `set()` method. We aren't interested in holding on to the return value of the `set() method, so to suppress the output of its string representation, we use a common Python convention of assigning it to the _`` variable (see e.g. Part 5 of [this nice article (https://medium.com/python-features/naming-conventions-with-underscores-in-python-791251ac7097)](https://medium.com/python-features/naming-conventions-with-underscores-in-python-791251ac7097))

In [ ]:
```
p = sns.lineplot(x=x_variable,y=y_variable,marker='o')
_ = p.set(xlabel="x",ylabel="y",xlim=(0,5),ylim=(0,8))
```



For multiple lines on the same plot in `seaborn`, you'll want to use `pandas`, which you are probably familiar with from previous Python experience. The `melt()` function is extremely useful for getting your data into the right format.

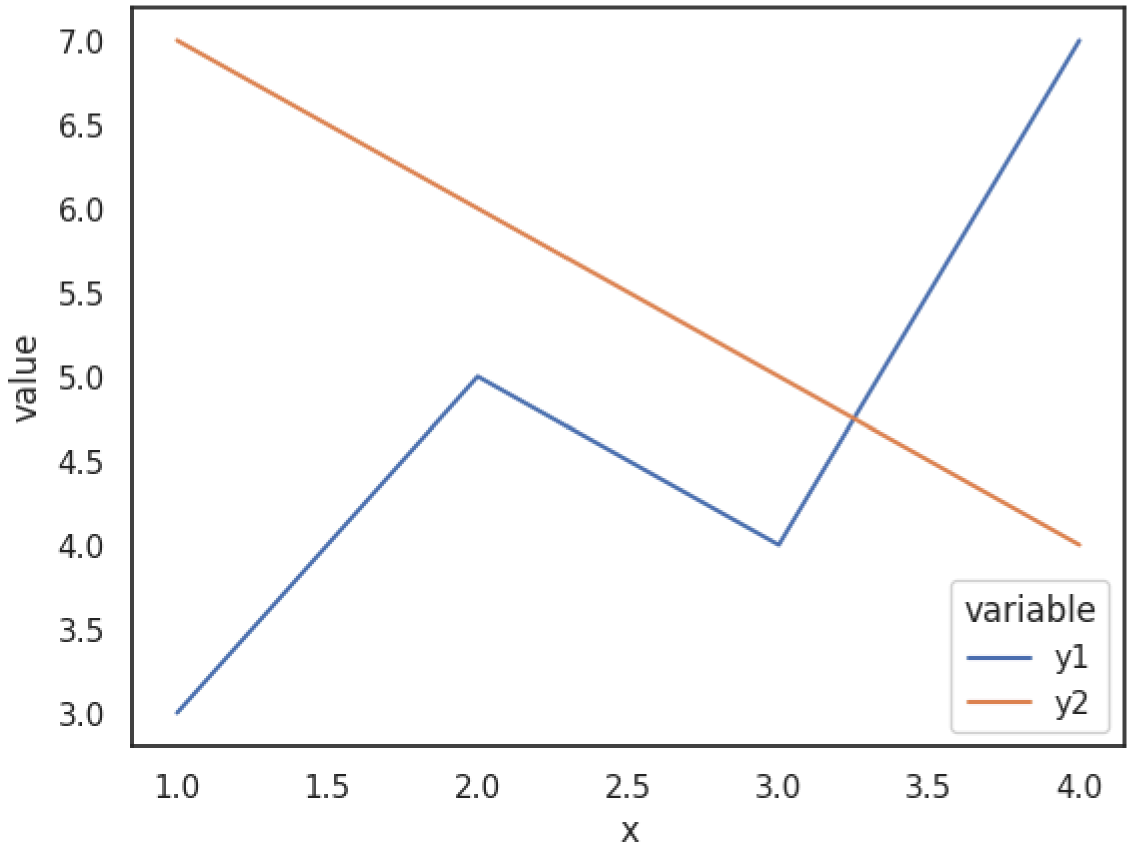
In [ ]:
```
import pandas as pd
dat = pd.DataFrame({
    'x': x_variable,
    'y1': y_variable,
    'y2': [7,6,5,4]
})
print(dat)
pd.melt(dat,['x']) # we leave this at the bottom of the cell so you can see what melt() does
```

```
   x  y1  y2
0  1   3   7
1  2   5   6
2  3   4   5
3  4   7   4
```

Out[38]:

	x	variable	value
0	1	y1	3
1	2	y1	5
2	3	y1	4
3	4	y1	7
4	1	y2	7
5	2	y2	6
6	3	y2	5
7	4	y2	4

In [ ]:
```
_ = sns.lineplot(x='x',y='value',hue='variable',data=pd.melt(dat,['x']))
```



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):

In [ ]:
```
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 hyper-parameters, 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.

In [ ]:
```
class bigram_model:
    def __init__(self, alpha):
        self.alpha = alpha
        self.bigram_counts = {}
        self.unigram_counts = {}
        self.vocab = set()
        self.vocab_size = 0

    def train(self, training_set):
        processed_sentences = []
        for sentence in training_set:
            processed_sentences.append(['<s>'] + sentence + ['</s>'])

        for sentence in processed_sentences:
            for i in range(len(sentence)):
                word = sentence[i]
                self.vocab.add(word)
                self.unigram_counts[word] = self.unigram_counts.get(word, 0) + 1

                if i < len(sentence) - 1:
                    next_word = sentence[i+1]
                    bigram = (word, next_word)
                    self.bigram_counts[bigram] = self.bigram_counts.get(bigram, 0) + 1

        self.vocab_size = len(self.vocab)
        return None

    def prob(self, previous_word, next_word):
        bigram_count = self.bigram_counts.get((previous_word, next_word), 0)
        unigram_count = self.unigram_counts.get(previous_word, 0)

        numerator = bigram_count + self.alpha
        denominator = unigram_count + (self.alpha * self.vocab_size)

        return numerator / denominator

    def perplexity(self, heldout_set):
        processed_sentences = []
        for sentence in heldout_set:
            processed_sentences.append(['<s>'] + sentence + ['</s>'])
```

```
        total_log_prob = 0
        total_tokens = 0

        for sentence in processed_sentences:
            for i in range(1, len(sentence)):  # Start from 1 to skip the first token
                previous_word = sentence[i-1]
                current_word = sentence[i]

                # Skip if either word is not in vocabulary
                if previous_word not in self.vocab or current_word not in self.vocab:
                    continue

                prob = self.prob(previous_word, current_word)
                total_log_prob += -1 * math.log2(prob)
                total_tokens += 1

        # Calculate perplexity
        if total_tokens == 0:
            return float('inf')

        return 2 ** (total_log_prob / total_tokens)
```

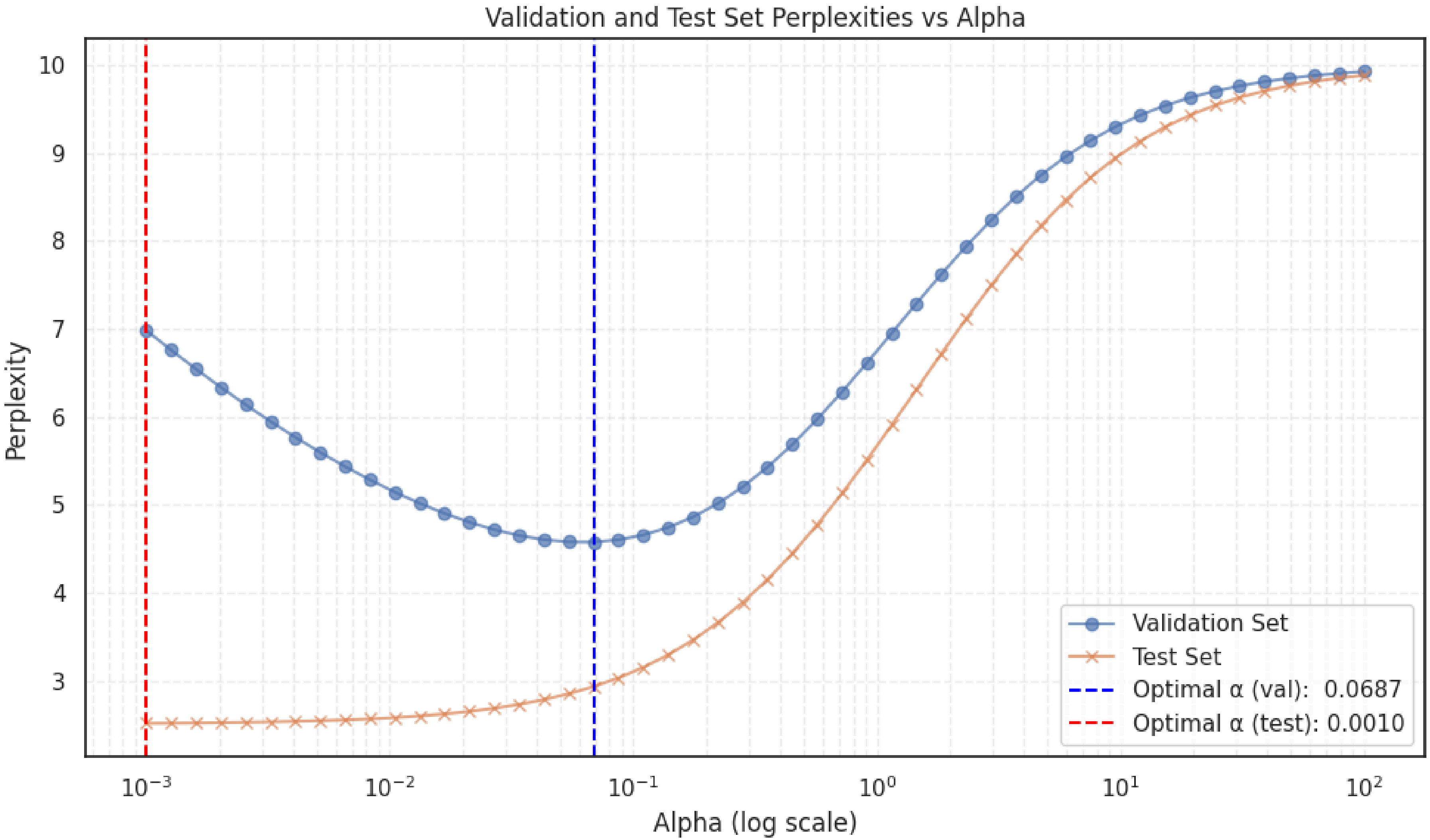
For step 1 of this problem, you need to find the value of α that optimizes validation-set perplexity; this is a simple example of what in machine learning these days "hyperparameter tuning" or "[hyperparameter optimization (https://en.wikipedia.org/wiki/Hyperparameter_optimization)](https://en.wikipedia.org/wiki/Hyperparameter_optimization)".

In [ ]:
```
##TODO: your code for step 1 goes here
import math
import numpy as np
import matplotlib.pyplot as plt

def find_optimal_alpha(training_set, validation_set, test_set, alphas):
    val_perps = []
    test_perps = []
    best_val = float('inf')
    best_alpha = None

    for alpha in alphas:
        model = bigram_model(alpha)
        model.train(training_set)

        val_p = model.perplexity(validation_set)
        test_p = model.perplexity(test_set)

        val_perps.append(val_p)
        test_perps.append(test_p)

        if val_p < best_val:
            best_val = val_p
            best_alpha = alpha

    return best_alpha, best_val, val_perps, test_perps
```

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.

In [ ]:
```
alphas = np.logspace(-3, 2, 50)
optimal_alpha, min_val_perp, val_perps, test_perps = \
    find_optimal_alpha(training_set, validation_set, test_set, alphas)

# best α on the test set
test_idx       = np.argmin(test_perps)
test_opt_alpha = alphas[test_idx]
test_min_perp  = test_perps[test_idx]

# report
print(f"Optimal α (val):  {optimal_alpha:.6f}, perplexity: {min_val_perp:.4f}")
print(f"Optimal α (test): {test_opt_alpha:.6f}, perplexity: {test_min_perp:.4f}")

# plot
plt.figure(figsize=(10, 6))
plt.semilogx(alphas, val_perps,  label='Validation Set', marker='o', linestyle='-', alpha=0.7)
plt.semilogx(alphas, test_perps, label='Test Set',       marker='x', linestyle='-', alpha=0.7)

plt.axvline(x=optimal_alpha,   color='blue', linestyle='--',
            label=f'Optimal α (val):  {optimal_alpha:.4f}')
plt.axvline(x=test_opt_alpha,  color='red',  linestyle='--',
            label=f'Optimal α (test): {test_opt_alpha:.4f}')

plt.xlabel('Alpha (log scale)')
plt.ylabel('Perplexity')
plt.title('Validation and Test Set Perplexities vs Alpha')
plt.legend()
plt.grid(True, which="both", ls="--", alpha=0.3)
plt.tight_layout()
plt.show()
```

```
Optimal α (val):  0.068665, perplexity: 4.5759
Optimal α (test): 0.001000, perplexity: 2.5154
```


Validation and Test Set Perplexities vs Alpha
Perplexity
10
9
8
7
6
5
4
3
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
Alpha (log scale)
Validation Set
Test Set
Optimal α (val): 0.0687
Optimal α (test): 0.0010

Interpret the results

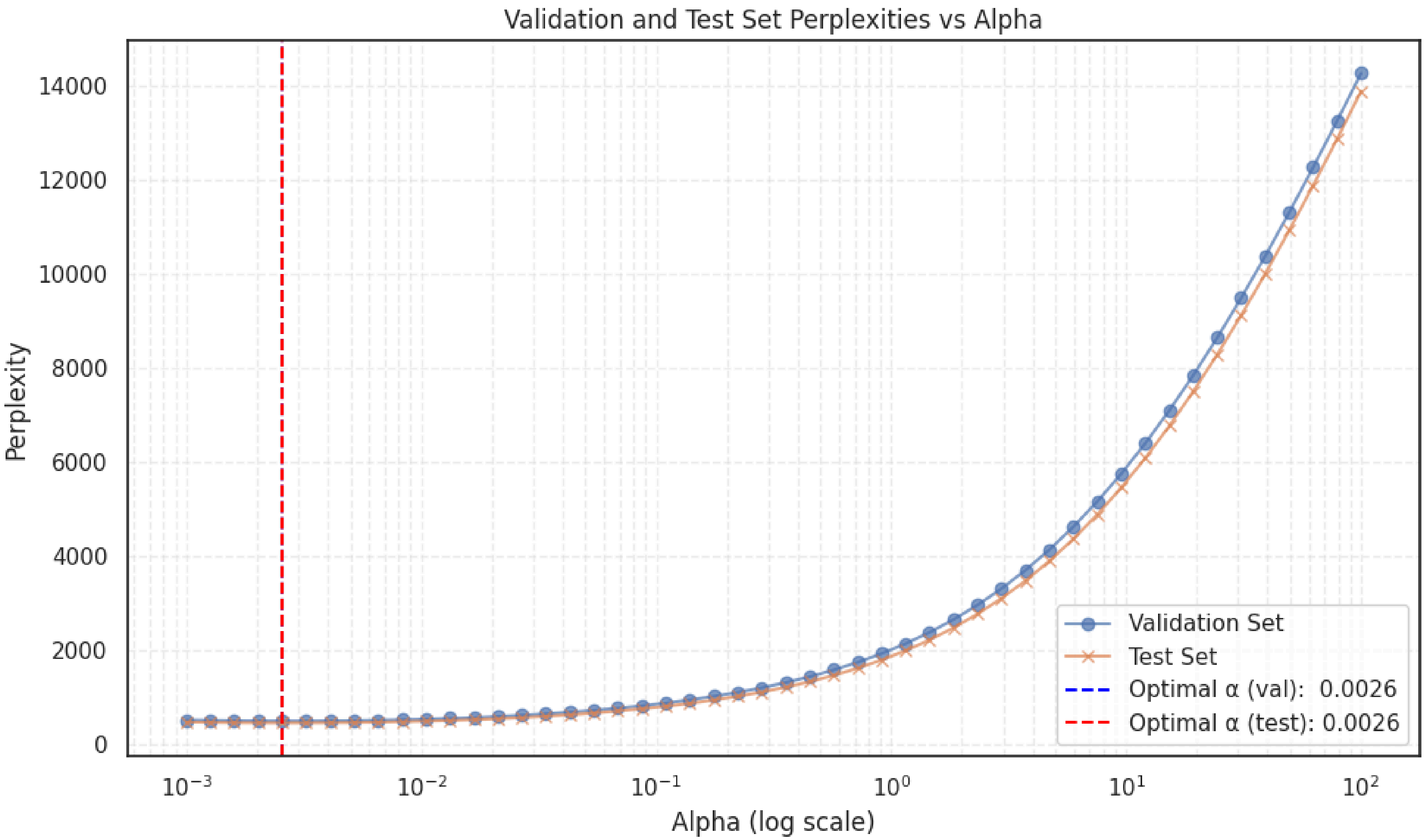
- What value of α worked the best for the validation set?
- Was it the same that would have worked best for the test set?

TODO: Optimal alpha for validation set: 0.068665 Minimum validation perplexity: 4.5759 Optimal alpha for test set: 0.001 Minimum test perplexity: 2.5154 Therefore it's not the same that would of worked for the test set.

In [ ]:
```python
def load_data(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        data = file.readlines()
    # Tokenize each line if it's not already tokenized
    # This assumes each line is a sentence and tokens are space-separated
    return [line.strip().split() for line in data]

# Load data
train_file = "wiki.train.tokens"
valid_file = "wiki.valid.tokens"
test_file = "wiki.test.tokens"

train_data = load_data(train_file)
valid_data = load_data(valid_file)
test_data = load_data(test_file)

alphas = np.logspace(-3, 2, 50)
optimal_alpha, min_val_perp, val_perps, test_perps = \
    find_optimal_alpha(train_data, valid_data, test_data, alphas)

# best α on the test set
test_idx      = np.argmin(test_perps)
test_opt_alpha = alphas[test_idx]
test_min_perp  = test_perps[test_idx]

# report
print(f"Optimal α (val):  {optimal_alpha:.6f}, perplexity: {min_val_perp:.4f}")
print(f"Optimal α (test): {test_opt_alpha:.6f}, perplexity: {test_min_perp:.4f}")

# plot
plt.figure(figsize=(10, 6))
plt.semilogx(alphas, val_perps,  label='Validation Set', marker='o', linestyle='-', alpha=0.7)
plt.semilogx(alphas, test_perps, label='Test Set',       marker='x', linestyle='-', alpha=0.7)

plt.axvline(x=optimal_alpha,   color='blue', linestyle='--',
            label=f'Optimal α (val):  {optimal_alpha:.4f}')
plt.axvline(x=test_opt_alpha,  color='red',  linestyle='--',
            label=f'Optimal α (test): {test_opt_alpha:.4f}')

plt.xlabel('Alpha (log scale)')
plt.ylabel('Perplexity')
```

```
plt.title('Validation and Test Set Perplexities vs Alpha')
plt.legend()
plt.grid(True, which="both", ls="--", alpha=0.3)
plt.tight_layout()
plt.show()
```

```
Optimal α (val):  0.002560, perplexity: 504.5238
Optimal α (test): 0.002560, perplexity: 464.9524
```



Bonus (5 points)

Results: Optimal alpha for validation set: 0.002560 Minimum validation perplexity: 504.5238 Optimal alpha for test set: 0.002560 Minimum test perplexity: 464.9524

Export to PDF

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

In [ ]:
```
# 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 automatica

!wget -nc https://raw.githubusercontent.com/lacclab/096222-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.
```

```
---------------------------------------------------------------------------
MessageError                              Traceback (most recent call last)
<ipython-input-49-0e654ef69c3d> in <cell line: 0>()
     10 # E.g. in your case the file name may be "Copy of XXXX.ipynb"
     11 
---> 12 colab_pdf(file_name='Pset_2_Tuning_ngram_model.ipynb', notebookpath="drive/MyDrive/Colab Notebook
s")

/content/colab_pdf.py in colab_pdf(file_name, notebookpath)
     16         from google.colab import drive
     17 
---> 18         drive.mount(drive_mount_point)
     19 
     20     # Check if the notebook exists in the Drive.

/usr/local/lib/python3.11/dist-packages/google/colab/drive.py in mount(mountpoint, force_remount, timeout_m
s, readonly)
     98 def mount(mountpoint, force_remount=False, timeout_ms=120000, readonly=False):
     99   """Mount your Google Drive at the specified mountpoint path."""
--> 100   return _mount(
    101       mountpoint,
    102       force_remount=force_remount,

/usr/local/lib/python3.11/dist-packages/google/colab/drive.py in _mount(mountpoint, force_remount, timeout_
ms, ephemeral, readonly)
    135   )
    136   if ephemeral:
--> 137     _message.blocking_request(
    138         'request_auth',
    139         request={'authType': 'dfs_ephemeral'},

/usr/local/lib/python3.11/dist-packages/google/colab/_message.py in blocking_request(request_type, request,
timeout_sec, parent)
    174       request_type, request, parent=parent, expect_reply=True
    175   )
--> 176   return read_reply_from_input(request_id, timeout_sec)

/usr/local/lib/python3.11/dist-packages/google/colab/_message.py in read_reply_from_input(message_id, timeo
ut_sec)
    101     ):
    102       if 'error' in reply:
```

```
--> 103         raise MessageError(reply['error'])
    104       return reply.get('data', None)
    105

MessageError: Error: credential propagation was unsuccessful
```