In [ ]:
```
# Initialize Otter
import otter
grader = otter.Notebook()
```

CS187

Lab 1-3 – Naive Bayes classification

In [ ]:
```
# Please do not change this cell because some hidden tests might depend on i
import os

# Only install packages if not in cs187-env
if 'cs187-env' not in os.environ.get('CONDA_PREFIX', ''):
    import subprocess
    import sys
    subprocess.run([sys.executable, '-m', 'pip', 'install', '-q', '-r', 'req
                    check=True)
```

In this lab, you'll apply the naive Bayes method to the *Federalist* papers' authorship attribution problem.

After this lab, you should be able to

- Derive the basic equations for the naive Bayes classification method;
- Estimate the parameters for the naive Bayes model;
- Determine where use of the "log trick" is indicated, and apply it.

New bits of Python used for the first time in the *solution set* for this lab, and which you may therefore find useful:

- `math.log2`

Preparation – Loading packages and data

In [ ]:
```
%matplotlib inline
import json
import math
import matplotlib
import matplotlib.pyplot as plt
matplotlib.style.use('tableau-colorblind10')
import torch
import wget
import os
```

```
from collections import defaultdict
```

In [ ]:
```
# Prepare to download needed data
def download_if_needed(source, dest, filename):
    if os.path.exists(f"./{dest}{filename}"):
        print (f"Skipping {filename}")
    else:
        print (f"Downloading {filename} from {source}")
        wget.download(source + filename, out=dest)

source_path = "https://raw.githubusercontent.com/nlp-course/data/refs/heads/
dest_path = "./"
subdirectory_text = "Text/"
subdirectory_models = "Models/"

# Download data and tokenizer
os.makedirs(dest_path + subdirectory_text, exist_ok=True)
for filename in ["federalist_data_four_counts.json",
                 ]:
    download_if_needed(source_path + subdirectory_text,
                       dest_path + subdirectory_text,
                       filename)

# Read in the raw data
dataset = json.load(open(dest_path + subdirectory_text + "federalist_data_fc
```

In [ ]:
```
# As before, we extract the papers by either of Madison and Hamilton
# to serve as training data.
training = list(filter(lambda ex: ex['authors'] in ['Madison', 'Hamilton'],
                       dataset))
```

The Naive Bayes method reviewed

A quick review of the Naive Bayes (NB) method for text classification: In classification tasks, we're given a representation of some text as a vector $\mathbf{x} = \langle x_1, x_2, \ldots, x_m \rangle$ of feature values, and we'd like to determine which of a set of classes $\{y_1, y_2, \ldots, y_k\}$ the text should be classified as.

> In the case at hand, the Federalist Papers, for a given document, we'll take $\mathbf{x} = \langle x_1, x_2, \ldots, x_m \rangle$ to be the sequence of words in the document, so each x_i corresponds to a single word *token*.

We might naturally think to choose that class that has the highest probability of being correct given the text, that is, the class y_i that maximizes $Pr(y_i \mid \mathbf{x})$.

By Bayes rule (this is the "Bayes" part in the name "Naive Bayes"),

$$\begin{aligned}\underset{i}{\operatorname{argmax}} \Pr(y_i \mid \mathbf{x}) &= \underset{i}{\operatorname{argmax}} \frac{\Pr(\mathbf{x} \mid y_i) \cdot \Pr(y_i)}{\Pr(\mathbf{x})} \\ &= \underset{i}{\operatorname{argmax}} \Pr(\mathbf{x} \mid y_i) \cdot \Pr(y_i)\end{aligned}$$

Question: Why can we drop the denominator in the last step of this derivation?

Type your answer here, replacing this text.

We use the following terminology: $\Pr(y_i)$ is the *prior probability*. $\Pr(\mathbf{x} \mid y_i)$ is the *likelihood*. $\Pr(y_i \mid \mathbf{x})$ is the *posterior probability*.

By the chain rule,

$$\begin{aligned}\Pr(\mathbf{x} \mid y_i) &= \Pr(x_1, \ldots, x_m \mid y_i) \\ &= \Pr(x_1 \mid y_i) \cdot \Pr(x_2, \ldots, x_m \mid x_1, y_i) \\ &= \Pr(x_1 \mid y_i) \cdot \Pr(x_2 \mid x_1, y_i) \cdot \Pr(x_3, \ldots, x_m \mid x_1, x_2, y_i) \\ \cdots &= \prod_{j=1}^{m} \Pr(x_j \mid x_1, \ldots, x_{j-1}, y_i)\end{aligned}$$

We further assume that each feature x_j is independent of all the others given the class. (That's the "naive" part.) So

$$\Pr(x_j \mid x_1, \ldots, x_{j-1}, y_i) \approx \Pr(x_j \mid y_i)$$

Using this approximation, we'll calculate instead the class as per the following maximization:

$$\underset{i}{\operatorname{argmax}} \Pr(y_i \mid \mathbf{x}) \approx \underset{i}{\operatorname{argmax}} \underbrace{\Pr(y_i)}_{\text{the prior}} \cdot \underbrace{\prod_{j=1}^{m} \Pr(x_j \mid y_i)}_{\text{the likelihood}}$$

> This independence assumption, in the text case, amounts to ignoring the *order* and even the *cooccurence* of words in a document, a quite aggressive and unrealistic independence assumption indeed.

All we need, then, for the Naive Bayes classification method is values for $\Pr(y_i)$ and $\Pr(x_j \mid y_i)$ for each feature x_j and each class y_i. These constitute the *parameters* of the model, which we will learn from a training dataset.

Naive Bayes for the Federalist papers

In applying Naive Bayes to an example in the Federalist dataset, we'll take the x_j to be the *tokens in the example*. To make the calculations easier, in this lab, we won't use *all* of the tokens, just the tokens of the four word types we've been attending to, but in an actual application of NB, we'd use (essentially) all of the word types. That is, we still treat the input as a bag-of-words representation, but the vocabulary is now limited to just four word types.

As a reminder the four word types are

In [ ]:
```
keywords = ['on', 'upon', 'there', 'whilst']
```

and the two class labels are

In [ ]:
```
classes = ['Hamilton', 'Madison']
```

Estimating the prior probabilities

Let's start with the prior probabilities $\Pr(y_i)$. In our case, there are only two class labels, for Hamilton and Madison. We estimate the probability of a class y_i by simply counting the proportion of examples that are labeled with that class. (This estimate is the *sample probability*, which is also referred to as the *maximum likelihood estimate* for reasons we'll skip for the moment.) That is, we estimate

$$\Pr(y_i) \approx \frac{\sharp(y_i)}{N}$$

where N is the number of training examples, and $\sharp(y_i)$ is the number of training examples of class y_i.

In the cell below, write code to count how many of the training examples are labeled with Hamilton and how many are labeled with Madison. Use these to provide estimates of the Hamilton and Madison prior probabilities.

In [ ]:
```
#TODO - Calculate the prior probabilities for Madison and Hamilton as floats
prior_madison = ...
prior_hamilton = ...
```

In [ ]:
```
grader.check("priors")
```

In [ ]:
```
print(f"Madison  prior: {prior_madison:.4f}\n"
      f"Hamilton prior: {prior_hamilton:.4f}")
```

Question: What do these probabilities tell us about how we might predict the class of a *Federalist* document *prior* to looking at the actual content of the document? (That's why these probabilities are called "priors".)

Type your answer here, replacing this text.

Estimating the likelihood probabilities

Now for the likelihood probabilities, $\Pr(\mathbf{x} \mid y_i)$. Recall that the overall NB likelihood is estimated by

$$\Pr(\mathbf{x} \mid y_i) \approx \prod_{j=1}^{m} \Pr(x_j \mid y_i)$$

For each particular conditional probability parameter $\Pr(x_j \mid y_i)$, we need to estimate a value. We'll do so by simply counting the number of training examples with feature value x_j that are labeled y_i (notated as $\sharp(x_j, y_i)$) as a proportion of the overall number of words labeled as y_i, that is,

$$\Pr(x_j \mid y_i) \approx \frac{\sharp(x_j, y_i)}{\sum_k \sharp(x_k, y_i)}$$

Again, for the text case, each token counts as an instance of the corresponding word type in a training example. Note that $\sum_k \sharp(x_k, y_i)$ is not the same as $\sharp(y_i)$.

We've provided a small table that shows, for each label (author) and each of the four word types of interest, how many tokens of the type occurred in training examples with that label.

In [ ]:
```
def counts(dataset, label, index):
    """Returns the total count for `index` for examples with the
       given `label`"""
    return sum([example['counts'][index]
                for example in dataset
                if example['authors'] == label])

# print a table header
print(f"{'':10}", end="")
for i in range(4):
    print(f"{keywords[i]:>8}", end="")
print()
# print table entries for each label
for label in classes:
    print(f"{label:10}", end="")
    for i in range(4):
        print(f"{counts(training, label, i):8}", end="")
    print()
```

Given the counts in this table, what would an estimate be for the probability that a given word would be "whilst" given that the document was authored by Madison, that is, $\Pr(\text{whilst} \mid \text{Madison})$?

In [ ]:
```
#TODO - Define this variable to be the specified probability.
condprob_whilst_madison = ...
```

In [ ]:
```
grader.check("prob_whilst_madison")
```

What about the probability $\Pr(\text{on} \mid \text{Hamilton})$?

In [ ]:
```
#TODO - Define this variable to be the specified probability.
condprob_on_hamilton = ...
```

In [ ]:
```
grader.check("prob_on_hamilton")
```

Consider a sample text

> **whilst** depending neither **on** the American government nor **on** the British

What would the Naive Bayes method estimate be for the likelihood of this sentence given it was by Hamilton? By Madison? (You should of course ignore all the tokens in our little sample text except for tokens of the four keyword types. (We've boldfaced their occurrences.) With a full-blown NB analysis, we'd be using *all* of the words in the text.)

In [ ]:
```
#TODO - Define the variables to be the corresponding likelihood probabilitie
likelihood_hamilton = ...
likelihood_madison = ...
```

In [ ]:
```
grader.check("likelihoods")
```

In [ ]:
```
print(f"Madison  likelihood: {likelihood_madison:4f}\n"
      f"Hamilton likelihood: {likelihood_hamilton:4f}")
```

Posterior probabilities

We're almost there. We simply need to combine the prior probabilities and the likelihood probabilities for each class to form the posterior, and select the largest one. As a reminder, we don't actually calculate the posterior *probability* because we aren't dividing through by $\Pr(\mathbf{x})$. Instead, we get something like a posterior *score*.

Calculate the posteriors for the two classes, and then specify which class – Hamilton or Madison – the NB method would predict for the sample text.

In [ ]:
```
#TODO - Define the variables to be the corresponding posterior probabilities
#       and the classification of the sample phrase.
posterior_madison = ...
posterior_hamilton = ...
sample_classification = ...
```

In [ ]:
```
grader.check("posteriors")
```

In [ ]:
```
print(f"Madison  posterior: {posterior_madison:4f}\n"
      f"Hamilton posterior: {posterior_hamilton:4f}\n"
      f"Sample classification: {sample_classification}")
```

Question: Is the NB-predicted classification the same as or different from the classification based on the priors? Why?

Type your answer here, replacing this text.

A practical issue and the "log trick"

The computations of what we've been calling the posterior scores

$$\Pr(y_i \mid \mathbf{x}) \approx \Pr(y_i) \cdot \prod_{j=1}^{m} \Pr(x_j \mid y_i)$$

involve the multiplication of many extremely small numbers. This is a recipe for *arithmetic underflow*, leading to garbage outputs, like the following:

In [ ]:
```
# An example of an arithmetic underflow
2 ** (-1100)
```

Instead, rather than maximizing the posterior, we can maximize its logarithm. Since the logarithm function is monotonic (see the next cell for a figure), whichever i maximizes the posterior maximizes its log as well.

In [ ]:
```
def log_plot():
    x = torch.linspace(1e-10, 1, 100)
    fig, ax = plt.subplots()
    ax.plot(x, torch.log2(x), label = "log_2")
    plt.title("Monotonicity of logarithm")
    plt.legend()

log_plot()
```

The log of the posterior is

$$\log\left(\Pr(y_i) \cdot \prod_{j=1}^{m} \Pr(x_j \mid y_i)\right) = \log \Pr(y_i) + \sum_{j=1}^{m} \log \Pr(x_j \mid y_i)$$

so that the calculation now involves the sum of a bunch of numbers rather than the product. In practice, this computation is much more robust. This is the "log trick"; we'll use it extensively in the future.

> A log-of-probability value is referred to, colloquially if not quite accurately, as a *logit*, because of a resemblance to the values of the logit function.

Calculate the log of the posterior for Madison for the sample text by summing up all of the pertinent logits, and similarly for Hamilton. Use the base 2 logarithm.

In [ ]:
```
#TODO - Calculate the log of the posterior for Madison by summing up all
#       of the pertinent parts.
log_posterior_madison = ...
log_posterior_hamilton = ...
```

In [ ]:
```
grader.check("log_posteriors")
```

In [ ]:
```
print(f"Madison  log posterior: {log_posterior_madison:8.3f}\n"
      f"Hamilton log posterior: {log_posterior_hamilton:8.3f}")
```

Question: Which one of the two is larger? Does this accord with your expectation?

Type your answer here, replacing this text.

Lab debrief

Question: We're interested in any thoughts you have about this lab so that we can improve this lab for later years, and to inform later labs for this year. Please list any issues that arose or comments you have to improve the lab. Useful things to comment on include the following, but you're not restricted to these:

- Was the lab too long or too short?
- Were the readings appropriate for the lab?
- Was it clear (at least after you completed the lab) what the points of the exercises were?
- Are there additions or changes you think would make the lab better?

Type your answer here, replacing this text.

End of lab 3

To double-check your work, the cell below will rerun all of the autograder tests.

In [ ]:
```
grader.check_all()
```