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

CS187

Lab 1-4 – Discriminative methods for 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 a discriminative classification method to the *Federalist* papers' authorship attribution problem, the logistic regression (perceptron) model.

After this lab, you should be able to

- Derive the basic equations for the logistic regression classification method;
- Perform the forward computation to generate the class that a logistic regression model predicts for some input;
- Calculate a loss for that prediction, the cross-entropy loss;
- Update the parameters of a logistic regression model by stochastic gradient descent.

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

- `torch.exp`
- `torch.dot`

Preparation – Loading packages and data

In [ ]:
```
# Please do not change these imports because some hidden tests might depend
# You can add a cell below if you need to import anything else.
import json
import math
import matplotlib
import matplotlib.pyplot as plt
matplotlib.style.use('tableau-colorblind10')
```

```
import os
import torch
import wget

from torch import log2
from pprint import pprint
```

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 tensorize the count vectors...
for example in dataset:
    example['counts'] = torch.tensor(example['counts']).type(torch.float)

# ...and 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))
```

Logistic regression

You've read about logistic regression, a method for classification that is discriminative rather than generative. Like the generative Naive Bayes method, each example is characterized by a vector of features (the counts of word types for a text, say, as for the *Federalist* example that we've been using). In logistic regression, each such feature is assigned a weight, and the score for an example is given by weighting each feature value by its weight and summing the result; that is, the score is the dot product of the feature vector and the weight vector. Let's take an example, one of the Federalist papers. We'll

extract from the training data the counts for the first example in the training set. That's the feature vector for this example.

In [ ]:
```
training0_counts = training[0]['counts']
training0_counts
```

Suppose the weights for the features are as given by the following vector:

In [ ]:
```
weights = torch.tensor([-.1, .2, .3, -1])
```

What would the weighted sum of the counts with these weights be? Feel free to use `torch.dot` to take the dot product for you. (Alternatively, you can also use basic tensor operations such as `*` and `.sum()`.)

In [ ]:
```
#TODO: Take the weighted sum of `training0_counts` with `weights`
wtd_sum = ...
```

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

Question: What is the range of possible values that such a weighted sum can take on?

Type your answer here, replacing this text.

In order to have a standard way of comparing these numbers, it helps to be able to place them on a fixed scale, from 0 to 1, say. This way, they can be interpreted as probabilities. We use the *logistic function* (σ) to carry out this conversion:

$$\sigma(x) = \frac{1}{1 + e^{-kx}}$$

In addition to its argument x, the function takes an additional parameter k, which we will explore shortly.

Complete the definition of a function `sigma` implementing the logistic function. (We've established a default value for `k` of 1 in the header line below.)

> Hint: For exponentiation, you can use the `exp` function provided by `torch`.

In [ ]:
```
#TODO: Implement the logistic function.
def sigma(x, k=1):
    """Computes sigma(x) = 1 / (1 + exp(-kx)).
    Arguments:
      x: a tensor
    Returns: sigma applied to each element of x"""
    ...
```

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

To get a sense of the logistic function, we graph it for several values of k.

In [ ]:
```
def sigma_plot():
    x = torch.linspace(-2, 2, 100)
    fig, ax = plt.subplots()
    for k in [0, 1, 2, 10]:
        ax.plot(x, sigma(x, k), label = f"k = {k}")
    plt.title("Logistic functions")
    plt.legend()

sigma_plot()
```

Question: What does the k parameter do to the logistic function?

Type your answer here, replacing this text.

Question: In the figure, the logistic function looks to be radially symmetric. In particular, it appears that $\sigma(x) = 1 - \sigma(-x)$. (If so, we can use a simple thresholding for classification purposes, $\sigma(x) > 0.5$ to capture $x > 0$.)

Prove the identity $\sigma(x) = 1 - \sigma(-x)$.

> Hint: *Don't* get hung up on this problem during the lab session. You can always come back to it later.

Type your answer here, replacing this text.

The logistic function, when applied to the weighted average above is greater than 0.5.

In [ ]:
```
sigma(wtd_sum)
```

In the *Federalist Papers* classification problem, there are only two classes, so we can use this single value to determine the classification. For an input feature vector $\vect{x}$ and weight vector $\vect{w}$, we'll take the model to predict the probability of the author being Hamilton (say), as

$$\Prob(\text{Hamilton} \given \mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x})$$

Therefore,

$$\Prob(\text{Madison} \given \mathbf{x}) = 1 - \sigma(\mathbf{w} \cdot \mathbf{x})$$

since there are only two classes.

> When there are more than two classes, we'd use a generalization of the sigmoid function, called softmax. We'll use that in later labs.

Define a function `predict_lr` (the "lr" for "logistic regression") that calculates the probability of Hamilton being the author of an example text, given a weight vector and

the feature vector for the example text.

In [ ]:
```
#TODO: Calculate the probability of Hamilton being the author
def predict_lr(weights, features):
    ...
```

This is the *forward computation* for the logistic regression model, calculating its output prediction from some inputs. Next we turn to the *backward computation*, calculating the updates to the parameters based on any error in the predicted output, as measured by a loss function.

Using a logistic regression model to predict Federalist authorship

Consider the following two training examples (examples 0 and 9) from the *Federalist* training dataset:

In [ ]:
```
for example in [0, 9]:
    pprint(training[example])
```

As above, a logistic regression model is defined by a vector of weights $\mathbf{w}$, like this weight vector that we defined above:

In [ ]:
```
weights
```

Calculate the predicted Hamilton probabilities for the two examples (examples 0 and 9) above.

In [ ]:
```
#TODO
prob_hamilton_example0 = ...
prob_hamilton_example9 = ...
```

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

In [ ]:
```
print(f"example 0 prob of Hamilton: {prob_hamilton_example0:.3f}\n"
      f"example 9 prob of Hamilton: {prob_hamilton_example9:.3f}")
```

Question: What does this model predict about the two training examples? Is it correct?

Type your answer here, replacing this text.

Training a logistic regression model

Of course, this is just one of a gazillion models (weight vectors), since we could have set the weights in all kinds of ways. How should we come up with a *good* model, i.e., a good

setting of the weights? Ideally, we'd try to find a set of weights that predicts all of the training data well. This is the problem of *training* a logistic regression model. Let's try another set of weights:

In [ ]:
```
weights_new = torch.tensor([0.1, 0.2, 0.3, -5.])
```

Calculate the probabilities generated by the model for these weights.

In [ ]:
```
#TODO
prob_hamilton_example0_new = ...
prob_hamilton_example9_new = ...
```

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

In [ ]:
```
print(f"example 0 prob: {prob_hamilton_example0_new:.3f}\n"
      f"example 9 prob: {prob_hamilton_example9_new:.3f}")
```

Question: Is this a better model, a worse model, or neither, *at least as far as the two sample examples are concerned*?

Type your answer here, replacing this text.

An ideal model would give a probability of 1 to the Hamilton examples and a 0 to the Madison examples. For the two sample examples, you'll have noticed that the new weights generate not 1 and 0, respectively, but numbers quite a bit closer to 1 and 0.

We could continue trying different weight values to try to improve the performance of the model on these training examples (and others) by trial and error, but a more systematic method is needed. We define a *loss function*, which specifies how bad a model is, and try to minimize the loss function by *stochastic gradient descent*.

We'll do a few steps of the process here by hand. It's sufficiently tedious that it's far better to deploy computers on the task, which we'll do in the next lab.

Cross-entropy loss function

We'll use the cross-entropy loss function. For an example i, we'll use $\mathbf{x}^{(i)}$ for the feature vector, and $y^{(i)}$ for the actual (gold) label (1 or 0, 1 for Hamilton and 0 for Madison). The predicted probability of the label for the i-th example being 1 will be as per the logistic regression model $\sigma(\mathbf{w} \cdot \mathbf{x}^{(i)})$, which we will call $\hat{y}^{(i)}$.

The cross-entropy loss for example i as per a model $\mathbf{w}$ is

$$L_{CE}(\mathbf{w}) = -(y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}))$$

Question: What is the minimum possible value of the cross-entropy loss above? When is that achieved? (Note that while $0 \log 0$ is undefined, $\lim_{x \to 0^+} x \log x = 0$.)

Type your answer here, replacing this text.

We define a function to compute the cross-entropy loss for a particular model (weight vector), example (feature vector), and gold label:

In [ ]:
```
def loss(weights, features, correct):
    """Returns the cross-entropy loss for a weight vector, a feature
       vector and a gold label `correct`."""
    y_hat = predict_lr(weights, features)
    return -(correct * log2(y_hat)
             + (1 - correct) * log2(1 - y_hat))
```

Use the `loss` function to determine the cross-entropy loss for example 0 for the original model that we used (`weights`), and for the new model (`weights_new`).

In [ ]:
```
#TODO: Calculate loss for training[0] under `weights` and `weights_new`
loss_example0_old = ...
loss_example0_new = ...
```

In [ ]:
```
grader.check("loss_example0_old_and_new")
```

In [ ]:
```
print(f"Old model loss: {loss_example0_old:.3f}\n"
      f"New model loss: {loss_example0_new:.3f}")
```

Question: Which of the models is better (at least on this example)?

Type your answer here, replacing this text.

Optimizing the weights with the gradient of the loss function

We want to find the weights that minimize the loss function. We use gradient descent:

1. Find the gradient of the loss function, the direction in which it is increasing fastest.
2. Take a step in the opposite direction.
3. Repeat.

For cross-entropy loss, recall that the partial derivative of the loss function with respect to a single weight w_j is

$$\frac{\partial L_{CE}(\mathbf{w})}{\partial w_j} = (\hat{y} - y) \cdot x_j$$

> At the end of this subsection, we give a problem that explores how this gradient is derived, but you can just assume it for the time being.

The gradient combines these partial derivatives for all of the weights.

$$\nabla L_{CE}(\mathbf{w}) = \begin{bmatrix} \frac{\partial L_{CE}(\mathbf{w})}{\partial w_1} \\ \frac{\partial L_{CE}(\mathbf{w})}{\partial w_2} \\ \vdots \\ \frac{\partial L_{CE}(\mathbf{w})}{\partial w_m} \end{bmatrix}$$

Let's work out an example, using example 0. The counts for example 0 are

In [ ]:
```
training[0]['counts']
```

and the original weights, recall, were

In [ ]:
```
weights
```

What is the gradient vector for these `weights` and training example `training[0]`?

In [ ]:
```
grad_vector = ...
grad_vector
```

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

Deriving the gradient of the cross-entropy loss

> You should skip this section until after you've done the rest of the lab.

As shown in the Jurafsky and Martin textbook (Section 5.10), the derivation for the gradient of the cross-entropy loss relies on the salutary fact that the derivative of the sigmoid has an especially simple form:

$$\frac{d\sigma(z)}{dz} = \sigma(z) \cdot (1 - \sigma(z))$$

Question: Prove this identity. For simplicity, you can assume that $k = 1$.

> **Hint:** You may find some of the following standard formulas for the derivative of various functions – reviewed from your calculus course – useful. (In these schematic identities, u and v are metavariables over functions of z, and a and n are metavariables over constants.)

$$\frac{d}{dz}a = 0$$
$$\frac{d}{dz}(u+v) = \frac{du}{dz} + \frac{dv}{dz}$$
$$\frac{d}{dz}(u\,v) = v\frac{du}{dz} + u\frac{dv}{dz}$$
$$\frac{d}{dz}\left[\frac{1}{u}\right] = -\frac{1}{u^2}\frac{du}{dz}$$
$$\frac{d}{dz}u^n = nu^{n-1}\frac{du}{dz}$$
$$\frac{d}{dz}e^u = e^u\frac{du}{dz}$$
$$\frac{d}{dz}\log_a u = (\log_a e)\frac{1}{u}\frac{du}{dz}$$

Type your answer here, replacing this text.

Adjusting weights against the gradient

Step 2 is to adjust the weights in the *opposite* direction of the gradient. In this case, we compute the new weight vector $\mathbf{w}'$ by adding to the weight vector a fraction of the negative gradient:

$$\mathbf{w}' = \mathbf{w} - \eta\nabla L_{CE}(\mathbf{w})$$

Here, η is the *learning rate*. The larger η is, the more we move in each step, but if η is too large we risk overshooting. We'll use a learning rate of 0.1 for now. (Setting good learning rates is one aspect of the black arts of machine learning.)

In [ ]:
```
learning_rate = 0.1
```

Calculate the new weight vector using `learning_rate` and `grad_vector`. Use `weights` (instead of `weights_new`) as the initial weights.

In [ ]:
```
weights_updated = ...
```

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

In [ ]:
```
print(weights_updated)
```

How do these weights perform on the training example we've been using? Let's see.

In [ ]:
```
loss_example0_updated = loss(weights_updated, training[0]['counts'], 1)
```

In [ ]:
```
print(f"Old model loss: {loss_example0_old:.3f}\n"
      f"New model loss: {loss_example0_new:.3f}\n"
```

```
      f"Updated model loss: {loss_example0_updated:.3f}")
```

If you did this all right, the loss for the updated model, which was generated by taking a single step opposite the gradient from the old model is not only better than the old model, but better than the new model we used above as well.

What about the loss on the other example we've been using (example 9)? Calculate the loss for example 9 with the old model (`weights`), the new model (`weights_new`), and the updated model (`weights_updated`):

In [ ]:
```
#TODO
loss_example9_old = ...
loss_example9_new = ...
loss_example9_updated = ...
```

In [ ]:
```
grader.check("weights_updated_9")
```

In [ ]:
```
print(f"Old model loss: {loss_example9_old:.3f}\n"
      f"New model loss: {loss_example9_new:.3f}\n"
      f"Updated model loss: {loss_example9_updated:.3f}")
```

Question: Did the update to the model improve its performance on example 9 or make it worse? Why?

Type your answer here, replacing this text.

Repeating the process

Step 3 is to repeat the process for this and other training examples. We could recalculate the gradient and take another step to improve further, and take steps to improve the other training examples, and so on and so forth, eventually converging on a model that works well over the entire training set. But doing so manually in this way is too tedious. We need to be able to do these kinds of computations at scale. Fortunately, in the next lab we'll be turning to packages that allow specifying these larger-scale computations.

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 1-4

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

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