CS187 Lab 1-2: Text classification and evaluation methodology

September 5, 2023

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

# Otter grader does not handle ! commands well, so we define and use our
# own function to execute shell commands.
def shell(commands, warn=True):
    """Executes the string `commands` as a sequence of shell commands.

       Prints the result to stdout and returns the exit status.
       Provides a printed warning on non-zero exit status unless `warn`
       flag is unset.
    """
    file = os.popen(commands)
    print (file.read().rstrip('\n'))
    exit_status = file.close()
    if warn and exit_status != None:
        print(f"Completed with errors. Exit status: {exit_status}\n")
    return exit_status

shell("""
ls requirements.txt >/dev/null 2>&1
if [ ! $? = 0 ]; then
 rm -rf .tmp
 git clone https://github.com/cs187-2021/lab1-2.git .tmp
 mv .tmp/tests ./
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

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

After this lab, you should be able to

- Understand the distinction between training and test corpora, and why both are needed;

- Understand the role of gold labels;
- Implement a majority class baseline as a benchmark to compare other methods;
- Implement nearest neighbor classification, and understand the role of distance metrics in its operation;
- Compare multiple methods for acccuracy.

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

- `collections.Counter`
- `collections.Counter.most_common`
- `torch.float`
- `torch.Tensor.type`

1 Preparation – Loading packages and data

[ ]:
```
# Please do not change these imports because some hidden tests might depend on them.
# You can add a cell below if you need to import anything else.
import collections
import copy
import json
import math
from pprint import pprint
import torch
import wget
```

2 The Federalist Papers

The *Federalist* papers is a collection of 85 essays written pseudonymously by Alexander Hamilton, John Jay, and James Madison following the Constitutional Convention of 1787, promoting the ratification of the nascent United States Constitution.

The authorship of many of the individual papers has been well established and acknowledged by the various authors, but a number of the papers have been contentious, with both Madison and Hamilton as possible authors. Determining the authorship of these disputed papers is a classic *text classification* problem, and one that has received great attention. The seminal work on the problem is that of Mosteller and Wallace, who applied then-novel statistical methods to the problem. In this lab, we'll use the *Federalist* data to experiment with some of the ideas about distance metrics and classification methods that you've read about. (It's also an excuse to make some points about proper testing methodology.)

Mosteller and Wallace used the frequencies of various words in the papers as the raw data for determining authorship. We've provided access to a heavily pre-digested version of this data. (If you're interested, you can find the raw data – all 85 papers – and the notebook used to generate the pre-digested data in the course's `data` github repository.)

Start by evaluating the cells below to load the data and view a sample.

[ ]:
```
# Retrieve the Federalist data
os.makedirs('data', exist_ok=True)
wget.download('https://github.com/nlp-course/data/raw/master/Federalist/
 ↪federalist_data.json', out='data/')
# Read the json data into a data structure
with open('data/federalist_data.json', 'r') as fin:
    dataset = json.load(fin)
# Convert counts to tensors of floats
for example in dataset:
    example['counts'] = torch.tensor(example['counts']).type(torch.float)
```

[ ]:
```
# View a sample of the data
print(f"Number of papers in the dataset: {len(dataset)}")
print("Some examples:")
pprint(dataset[:3])
```

You'll see above that the dataset is a list of *examples*, one for each paper, each a Python dictionary providing the paper number, its title and author(s), and the raw counts for a few important words in the papers. From the last lab, you'll recognize the `counts` field as a bag-of-words representation of the document. The `counts` field is the document representation that we will be wanting to classify, and the `authors` field contains the pertinent class label for each example.

For your reference, here are the words that were used to derive the counts:

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

Thus in the first example paper, *Federalist 1*, there were 9 tokens of "on", 6 of "upon", 2 of "there", and none of "whilst".

The `authors` field takes on various values. Here's a table of the frequency of each of the values. (This will come in handy later.)

[ ]:
```
# Generate a table of the number of papers by each author label
cnt = collections.Counter(map(lambda ex: ex['authors'],
                              dataset))
for author, count in cnt.items():
    print(f"{count:3d} ({100.0*count/len(dataset):4.1f}%) {author}")
```

As you can see, some of the papers are of known authorship by one of Madison or Hamilton. We can use these as training data.

[ ]:
```
# Extract the papers by either of Madison and Hamilton
training = list(filter(lambda ex: ex['authors'] in ['Madison', 'Hamilton'],
                       dataset))
```

[ ]:
```
# View a sample of the training data
print(f"Number of papers in the dataset: {len(training)}")
```

```
print("Some examples:")
pprint(training[:3])
```

Others of the papers are of ambiguous authorship. They are shown as having `'Hamilton or Madison'` as author. These will be the elements that we want to test our models on.

[ ]:
```
# Extract the papers of unknown authorship
testing = list(filter(lambda ex: ex['authors'] == 'Hamilton or Madison',
                      dataset))
```

[ ]:
```
# View a sample of the data
print(f"Number of papers in the dataset: {len(testing)}")
print("Some sample elements:")
pprint(testing[:3])
```

3 Models for text classification

We can think of a *model* for a text classification problem as a function taking a test example and returning a class label for the test example. Generating the model will rely on a corpus of training data.

With a model in hand, we can evaluate its *accuracy* on a test corpus by computing the proportion of test examples that the model correctly classifies, that is, the model assigns to a test example the author that the test example specifies. Define a higher-order function `accuracy` that takes a test corpus (like `testing`) and a model (which is a function, remember), and returns the accuracy of the model on that corpus.

> For you CS51 aficionados, `accuracy` is a *higher-order function* since it *takes a function as an argument*. Yes, higher-order functions are possible in Python.

[ ]:
```
#TODO -- Define the `accuracy` function.
def accuracy(test_corpus, model):
    """Computes the accuracy of a model on a corpus.
    Arguments:
      `test_corpus`: a list of test examples, such as `testing`
      `model`: a function whose input is an example from the corpus (such as
               `testing[0]`, and whose output is the predicted author
    Returns:
      accuracy, a float number.
    """
    ...
```

3.1 Majority class classification

An especially simple classification model labels each test example with whichever label happens to occur most frequently in the training data. It completely ignores the test example that it classifies!

By examination of the table provided above, what is the majority class label for the training dataset?

[ ]:
```
#TODO -- Set this variable to the majority class label for the training set.
maj_class_label = ...
```

[ ]:
```
grader.check("maj_class_label")
```

Rather than determining the majority class by inspection, it's better to have a function to compute it for us. Define a function `majority_class_label` that returns the majority class label for a training set.

[ ]:
```
#TODO -- Define the `majority_class_label` function.
def majority_class_label(training):
    """Find the majority class label for a training set.
    Arguments:
      `training`: a list of training examples, such as `training`
    Returns:
      the majority class label, a string.
    """
    ...
```

[ ]:
```
grader.check("majority_class_label")
```

What proportions of the *training* examples do you think would be classified correctly by the majority class model?

[ ]:
```
#TODO -- Define this variable to be what you think the
#        accuracy of the majority class model would be
#        on the training data.
maj_class_accuracy_guess = ...
```

[ ]:
```
grader.check("maj_class_accuracy_guess")
```

Now define a function `majority_class` that takes a single argument (a test example) and returns the particular class label that is most frequent in the training data `training` (regardless of what the test example is).

[ ]:
```
#TODO - Define the `majority_class` model.
def majority_class(example):
    """Defines a majority class model.
    Arguments:
      `example`: an example, such as `testing[0]`
    Returns:
      the majority class in the *training* set, a string.
    """
    ...
```

[ ]:
```
grader.check("majority_class")
```

Now we can see how well this majority class model works by trying it out on some examples. Use the `accuracy` function to determine the model's accuracy when applied to the task of labeling the *training* data.

[ ]:
```
#TODO -- Define `maj_class_on_train` to be the accuracy of the majority
#        class model on the training data.
accuracy_maj_class_train = ...
```

[ ]:
```
grader.check("accuracy_maj_class_train")
```

[ ]:
```
print(f"Accuracy of the majority class model on training data: "
      f"{accuracy_maj_class_train:.3f}")
```

Was your guess from above right?

3.2 Nearest neighbor classification

Recall that nearest neighbor classification classifies a test example with the label of the nearest training example. To calculate nearest neighbors, we need a distance metric between the representations of the documents. Below we've provided two such metrics, familiar from the previous lab, for Euclidean distance and cosine distance.

> Note: In order to allow full use of `torch` operations, these functions assume that the vectors are provided as tensors of type `float`. (That's why we tensorified the `counts` data as we loaded the dataset at the top of this notebook.) When you call them, you'll want to make sure of this. They also return singleton tensors, not floats.

[ ]:
```
def euclidean_distance(v1, v2):
    """Returns the Euclidean distance between two vectors"""
    return torch.linalg.norm(v1 - v2)
```

[ ]:
```
def safe_acos(x):
    """Returns the arc cosine of `x`. Unlike `math.acos`, it
       does not raise an exception for values of `x` out of range,
       but rather clips `x` at -1..1, thereby avoiding math domain
       errors in the case of numerical errors."""
    return math.acos(math.copysign(min(1.0, abs(x)), x))

def cosine_distance(v1, v2):
    """Returns the cosine distance between two vectors"""
    dot_product = (v1 * v2).sum()
    v1_norm = (v1 * v1).sum().sqrt()
    v2_norm = (v2 * v2).sum().sqrt()
    return safe_acos(dot_product / (v1_norm * v2_norm)) / math.pi
```

Here's an example of the use of these distance metrics:

```
[ ]: t1 = torch.tensor([1., 2.])
     t2 = torch.tensor([3., 4.])

     print("Testing on two different tensors\n"
           f"Euclidean: {euclidean_distance(t1, t2)}\n"
           f"Cosine   : {cosine_distance(t1, t2)}\n\n"
           "Testing on two identical tensors\n"
           f"Euclidean: {euclidean_distance(t1, t1)}\n"
           f"Cosine   : {cosine_distance(t1, t1)}")
```

3.2.1 Generating nearest neighbor models

To specify a nearest neighbor model, we need both a training corpus (like `training`) and a distance metric (like `euclidean_distance` or `cosine_distance` defined just above).

Define a function called `define_nearest_neighbor` that takes a training corpus and a distance metric and returns a model – that is, a function that classifies a single test example. The model should return the class *label* of that training example whose *counts vector* is closest to that of the test example according to the metric.

> Again, harkening to CS51, `define_nearest_neighbor` is a higher-order function since it *returns a function*. Yes, higher-order functions are possible in Python.

```
[ ]: #TODO -- Define this function that generates nearest neighbor models.
     def define_nearest_neighbor(corpus, metric):
         """Generates a nearest neighbor model from a training corpus and a
         distance metric.
         Arguments:
           `corpus`: a training corpus, such as `training`
           `metric`: a metric function which takes two tensors as input and
                     returns their distance, such as `euclidean_distance`
         Returns:
           a model, which is a function that takes in a test example (such as
           `testing[0]`) and returns the author of the nearest example in the
           training set, where distances are measured on the counts vector
           using `metric`.
         """
         ...
```

We can use the `define_nearest_neighbor` function to define two new models for nearest neighbor classification, one using Euclidean distance and one using cosine distance.

```
[ ]: nearest_neighbor_euclidean_model = \
         define_nearest_neighbor(training, euclidean_distance)

     nearest_neighbor_cosine_model = \
         define_nearest_neighbor(training, cosine_distance)
```

3.2.2 Testing the nearest neighbor models on the training data

How accurate are these models when used to label the training data (as we did for the majority class model above)? Use the `accuracy` function above to calculate the accuracy of `nearest_neighbor_euclidean_model` in labeling the *training* data (not the test data), and similarly for `nearest_neighbor_cosine_model`.

[ ]:
```
#TODO - Define the variable to be the calculated accuracy.
accuracy_nn_euclidean_train = ...
accuracy_nn_cosine_train = ...
```

[ ]:
```
grader.check("accuracy_train")
```

[ ]:
```
print(f"Accuracy of the nearest neighbor euclidean model tested on training␣
 ↪data: "
      f"{accuracy_nn_euclidean_train:.3f}")
print(f"Accuracy of the nearest neighbor cosine model tested on training data: "
      f"{accuracy_nn_cosine_train:.3f}")
```

Question: Does the performance of these classifiers on the training data seem to you to be representative of how good a classifier each is? Why or why not?

Type your answer here, replacing this text.

Testing the nearest neighbor models on the testing data To get a better sense of how the nearest neighbor models perform, let's try them out on the testing data that we have. (Recall that the testing data in `testing` were the ambiguously-authored Federalist papers, where the `authors` field was `'Hamilton or Madison'`.)

We start by looking in detail at the predictions generated by the two nearest neighbor models. Print out a table that lists, for each `testing` example, the paper number and the authors predicted under the nearest neighbor Euclidean model and the nearest neighbor cosine model. It might look something like

```
49 Madison  Madison
50 Hamilton Madison
51 Madison  Madison
...
```

[ ]:
```
#TODO - Print out the requested table.
...
```

What do you notice about the two models?

Type your answer here, replacing this text.

3.2.3 Testing the nearest neighbor models on the training data

Now use the `accuracy` function to calculate the accuracy of the two nearest neighbor models as you did above, but this time calculating accuracy on the *testing* corpus rather than the training corpus. (Expect to find a surprising result. Read ahead for an explanation if you're confused.)

[ ]:
```
#TODO -- Define the variables to be, respectively, the calculated accuracy of␣
 ↪the nearest
#        neighbor Euclidean model and cosine model on the testing data.
accuracy_nn_euclidean_test = ...
accuracy_nn_cosine_test = ...
```

[ ]:
```
grader.check("accuracy_test")
```

[ ]:
```
print(f"Accuracy of the nearest neighbor euclidean model tested on testing data:
 ↪ "
      f"{accuracy_nn_euclidean_test:.3f}")
print(f"Accuracy of the nearest neighbor cosine model tested on testing data: "
      f"{accuracy_nn_cosine_test:.3f}")
```

Question: Does the performance of these classifiers on the testing data seem to you to be representative of how good a classifier each is? Why or why not?

Type your answer here, replacing this text.

3.2.4 The importance of gold labels

In order to evaluate the accuracy of the nearest neighbor model – and any model – we need to have the correct labels for the testing corpus, the so-called *gold* labels. What shall we use for gold labels? Mosteller and Wallace's much more extensive analysis concluded that all of the papers of ambiguous origin were penned by Madison, so we'll use that. We should use a version of the `testing` corpus with the gold labels.

Write some code to generate a version of the testing corpus with the gold labels.

> Hint: In defining `testing_gold`, you'll want to be careful not to change `testing`. Otherwise, some unit tests that use `testing` may fail. The `copy.deepcopy` function may be useful.

[ ]:
```
#TODO - Write code that defines `testing_gold`, which is the same
# as `testing` except that it has the correct gold labels.
# Note: be careful to not change `testing`.
...
```

[ ]:
```
grader.check("get_gold")
```

Now we can rerun the accuracy calculations.

```
[ ]: accuracy_nn_euclidean_test_with_gold = accuracy(testing_gold,␣
     ↪nearest_neighbor_euclidean_model)
     accuracy_nn_cosine_test_with_gold = accuracy(testing_gold,␣
     ↪nearest_neighbor_cosine_model)
```

```
[ ]: print(f"Accuracy of the nearest neighbor euclidean model tested on testing data:
     ↪ "
          f"{accuracy_nn_euclidean_test_with_gold:.3f}")
     print(f"Accuracy of the nearest neighbor cosine model tested on testing data: "
          f"{accuracy_nn_cosine_test_with_gold:.3f}")
```

Do these results make more sense?

4 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.

5 End of lab 1-2

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

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