CS187 Project Segment 1: Text Classification

September 29, 2024

[1]:
```
# 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/project1.git .tmp
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

[2]:
```
# Initialize Otter
import otter
grader = otter.Notebook()
```

1 CS187

1.1 Project segment 1: Text classification

In this project segment you will build several varieties of text classifiers using `PyTorch`.

1. A majority baseline.
2. A naive Bayes classifier.
3. A logistic regression (single-layer perceptron) classifier.
4. A multilayer perceptron classifier.

Preparation

[3]:
```
import copy
import re
import wget
import csv
import torch
import torch.nn as nn
import datasets

from datasets import load_dataset
from tokenizers import Tokenizer
from tokenizers.pre_tokenizers import Whitespace
from tokenizers import normalizers
from tokenizers.models import WordLevel
from tokenizers.trainers import WordLevelTrainer
from transformers import PreTrainedTokenizerFast
from collections import Counter
from torch import optim
from tqdm.auto import tqdm
```

[4]:
```
from typing import Dict
```

[5]:
```
# Random seed
random_seed = 1234
torch.manual_seed(random_seed)

## GPU check
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cpu
```

2 The task: Answer types for ATIS queries

For this and future project segments, you will be working with a standard natural-language-processing dataset, the ATIS (Airline Travel Information System) dataset. This dataset is composed

of queries about flights – their dates, times, locations, airlines, and the like.

> The ATIS dataset was generated using a "Wizard of Oz" methodology, a common approach in early NLP research and human-computer interaction more generally. Subjects were asked to interact with a "prototype of a voice-input information retrieval system. It has the same information that is contained in the Official Airline Guide (OAG) to help you make air travel plans." In reality, behind the curtain, two confederates were transcribing the queries and providing the answers. In this way, an "ecologically realistic" set of queries was obtained.

Over the years, the dataset has been annotated in all kinds of ways, with parts of speech, informational chunks, parse trees, and even corresponding SQL database queries. You'll use various of these annotations in future assignments. For this project segment, however, you'll pursue an easier classification task: **given a query, predict the answer type**.

These queries ask for different types of answers, such as

- Flight IDs: "Show me the flights from Washington to Boston"
- Fares: "How much is the cheapest flight to Milwaukee"
- City names: "Where does flight 100 fly to?"

In all, there are some 30 answer types to the queries.

Below is an example taken from this dataset:

Query:

```
show me the afternoon flights from washington to boston
```

SQL:

```
SELECT DISTINCT flight_1.flight_id FROM flight flight_1 , airport_service airport_service_1 , c
   WHERE flight_1.departure_time BETWEEN 1200 AND 1800
     AND ( flight_1.from_airport = airport_service_1.airport_code
           AND airport_service_1.city_code = city_1.city_code
           AND city_1.city_name = 'WASHINGTON'
           AND flight_1.to_airport = airport_service_2.airport_code
           AND airport_service_2.city_code = city_2.city_code
           AND city_2.city_name = 'BOSTON' )
```

In this project segment, we will consider the answer type for a natural-language query to be the target field of the corresponding SQL query. For the above example, the answer type would be *flight_id*.

2.1 Loading and preprocessing the data

> Read over this section, executing the cells, and **making sure you understand what's going on before proceeding to the next parts.**

First, let's download the dataset.

[6]:
```
data_dir = "https://raw.githubusercontent.com/nlp-course/data/master/ATIS/"
os.makedirs('data', exist_ok=True)
for split in ['train', 'dev', 'test']:
```

```
    wget.download(f"{data_dir}/{split}.nl", out='data/')
    wget.download(f"{data_dir}/{split}.sql", out='data/')
```

```
100% […] 250477 / 250477
```

Next, we process the dataset by extracting answer types from SQL queries and saving in CSV format.

[7]:
```
def get_label_from_query(query):
    """Returns the answer type from `query` by dead reckoning.
    It's basically the second or third token in the SQL query.
    """
    match = re.match(r'\s*SELECT\s+(DISTINCT\s*)?(\w+\.)?(?P<label>\w+)', query)
    if match:
        label = match.group('label')
    else:
        raise RuntimeError(f'no label in query {query}')
    return label

for split in ['train', 'dev', 'test']:
    sql_file = f'data/{split}.sql'
    nl_file = f'data/{split}.nl'
    out_file = f'data/{split}.csv'

    with open(nl_file) as f_nl:
        with open(sql_file) as f_sql:
            with open(out_file, 'w') as fout:
                writer = csv.writer(fout)
                writer.writerow(('label','text'))
                for text, sql in zip(f_nl, f_sql):
                    text = text.strip()
                    sql = sql.strip()
                    label = get_label_from_query(sql)
                    writer.writerow((label, text))
```

Let's take a look at what each data file looks like.

[8]:
```
shell('head "data/train.csv"')
```

```
label,text
flight_id,list all the flights that arrive at general mitchell international
from various cities
flight_id,give me the flights leaving denver august ninth coming back to boston
flight_id,what flights from tacoma to orlando on saturday
fare_id,what is the most expensive one way fare from boston to atlanta on
american airlines
flight_id,what flights return from denver to philadelphia on a saturday
flight_id,can you list all flights from chicago to milwaukee
flight_id,show me the flights from denver that go to pittsburgh and then atlanta
```

```
flight_id,i'd like to see flights from baltimore to atlanta that arrive before
noon and i'd like to see flights from denver to atlanta that arrive before noon
flight_id,do you have an 819 flight from denver to san francisco
```

It's a CSV file with the answer type in the first column and the query text in the second.

We use `datasets` to prepare the data, as in lab 1-5. More information on `datasets` can be found at https://huggingface.co/docs/datasets/loading.

[9]:
```
atis = load_dataset('csv', data_files={'train':'data/train.csv', \
                                       'val': 'data/dev.csv', \
                                       'test': 'data/test.csv'})
```

```
Generating train split: 0 examples [00:00, ? examples/s]

Generating val split: 0 examples [00:00, ? examples/s]

Generating test split: 0 examples [00:00, ? examples/s]
```

[10]:
```
atis
```

[10]:
```
DatasetDict({
    train: Dataset({
        features: ['label', 'text'],
        num_rows: 4379
    })
    val: Dataset({
        features: ['label', 'text'],
        num_rows: 491
    })
    test: Dataset({
        features: ['label', 'text'],
        num_rows: 448
    })
})
```

[11]:
```
train_data = atis['train']
val_data = atis['val']
test_data = atis['test']

train_data.shuffle(seed=random_seed)
```

[11]:
```
Dataset({
    features: ['label', 'text'],
    num_rows: 4379
})
```

We build a tokenizer from the training data to tokenize text and convert tokens into word ids.

[12]:
```
MIN_FREQ = 3 # words appearing fewer than 3 times are treated as 'unknown'
unk_token = '[UNK]'
```

```
pad_token = '[PAD]'

tokenizer = Tokenizer(WordLevel(unk_token=unk_token))
tokenizer.pre_tokenizer = Whitespace()
tokenizer.normalizer = normalizers.Lowercase()

trainer = WordLevelTrainer(min_frequency=MIN_FREQ, special_tokens=[pad_token,␣
 ↪unk_token])
tokenizer.train_from_iterator(train_data['text'], trainer=trainer)
```

We use `datasets.Dataset.map` to convert text into word ids. As shown in lab 1-5, first we need to wrap `tokenizer` with the `transformers.PreTrainedTokenizerFast` class to be compatible with the `datasets` library.

[13]:
```
hf_tokenizer = PreTrainedTokenizerFast(tokenizer_object=tokenizer,
                                       pad_token=pad_token,
                                       unk_token=unk_token)
```

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-
packages/transformers/tokenization_utils_base.py:1601: FutureWarning:
`clean_up_tokenization_spaces` was not set. It will be set to `True` by default.
This behavior will be depracted in transformers v4.45, and will be then set to
`False` by default. For more details check this issue:
https://github.com/huggingface/transformers/issues/31884
  warnings.warn(
```

[14]:
```
def encode(example):
    return hf_tokenizer(example['text'])

train_data = train_data.map(encode)
val_data = val_data.map(encode)
test_data = test_data.map(encode)
```

```
Map:   0%|          | 0/4379 [00:00<?, ? examples/s]
Map:   0%|          | 0/491 [00:00<?, ? examples/s]
Map:   0%|          | 0/448 [00:00<?, ? examples/s]
```

We also need to convert label strings into label ids.

[15]:
```
# Add a new column `label_id`
train_data = train_data.add_column('label_id', train_data['label'])
val_data = val_data.add_column('label_id', val_data['label'])
test_data = test_data.add_column('label_id', test_data['label'])

# Convert feature `label_id` from strings to integer ids
train_data = train_data.class_encode_column('label_id')

# Use the label vocabulary on training data to convert val and test sets
```

```
label2id = train_data.features['label_id']._str2int
val_data = val_data.class_encode_column('label_id')
val_data = val_data.align_labels_with_mapping(label2id, "label_id")
test_data = test_data.class_encode_column('label_id')
test_data = test_data.align_labels_with_mapping(label2id, "label_id")
```

```
Casting to class labels:   0%|          | 0/4379 [00:00<?, ? examples/s]
Casting to class labels:   0%|          | 0/491 [00:00<?, ? examples/s]
Aligning the labels:   0%|          | 0/491 [00:00<?, ? examples/s]
Casting to class labels:   0%|          | 0/448 [00:00<?, ? examples/s]
Aligning the labels:   0%|          | 0/448 [00:00<?, ? examples/s]
```

[16]:
```
print(val_data[:5])
```

```
{'label': ['flight_id', 'flight_id', 'state_code', 'flight_id', 'flight_id'],
'text': ['what flights are available tomorrow from denver to philadelphia',
'show me the afternoon flights from washington to boston', 'list all arrivals
from any airport to baltimore on thursday morning arriving before 9am', 'flights
from phoenix to milwaukee', "i'd like to fly from philadelphia to san francisco
through dallas"], 'input_ids': [[7, 4, 28, 59, 133, 3, 14, 2, 25], [11, 8, 5,
63, 4, 3, 34, 2, 10], [29, 24, 435, 3, 99, 86, 2, 23, 6, 79, 37, 57, 61, 288],
[4, 3, 125, 2, 96], [12, 27, 46, 26, 2, 40, 3, 25, 2, 13, 17, 295, 22]],
'token_type_ids': [[0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0], [0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0]], 'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1,
1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]], 'label_id': [15, 15, 24, 15, 15]}
```

[17]:
```
# Compute size of vocabulary
text_vocab = tokenizer.get_vocab()
label_vocab = train_data.features['label_id']._str2int
vocab_size = len(text_vocab)
num_labels = len(label_vocab)
print(f"Size of vocab: {vocab_size}")
print(f"Number of labels: {num_labels}")

id_to_label = [None] * num_labels
for label, int_id in label_vocab.items():
    assert id_to_label[int_id] is None
    id_to_label[int_id] = label
assert all(word is not None for word in id_to_label)
```

```
Size of vocab: 514
Number of labels: 30
```

To get a sense of the kinds of things that are asked about in this dataset, here is the list of all of the answer types in the training data, along with their label IDs.

[18]:
```
for label in label_vocab:
    print(f"{label_vocab[label]:2d} {label}")
```

```
 0 advance_purchase
 1 aircraft_code
 2 airline_code
 3 airport_code
 4 airport_location
 5 arrival_time
 6 basic_type
 7 booking_class
 8 city_code
 9 city_name
10 count
11 day_name
12 departure_time
13 fare_basis_code
14 fare_id
15 flight_id
16 flight_number
17 ground_fare
18 meal_code
19 meal_description
20 miles_distant
21 minimum_connect_time
22 minutes_distant
23 restriction_code
24 state_code
25 stop_airport
26 stops
27 time_elapsed
28 time_zone_code
29 transport_type
```

2.2 Handling unknown words

Note that we mapped words appearing fewer than 3 times to a special *unknown* token (we're using `[UNK]`). We do this for two reasons:

1. Due to the scarcity of such rare words in training data, we might not be able to learn generalizable conclusions about them.
2. Introducing an unknown token allows us to deal with out-of-vocabulary words in the test data as well: we just map those words to `[UNK]`.

[19]:
```
print (f"Unknown token: {unk_token}")
unk_index = text_vocab[unk_token]
print (f"Unknown token id: {unk_index}")
```

```
# UNK example
example_unk_token = 'IAmAnUnknownWordForSure'
print (f"An unknown token: {example_unk_token}")
print (f"Mapped back to word id: {hf_tokenizer(example_unk_token).input_ids}")
print (f"Mapped to [UNK]'s?: {all([id == unk_index for id in␣
 ↪hf_tokenizer(example_unk_token).input_ids])}")
```

```
Unknown token: [UNK]
Unknown token id: 1
An unknown token: IAmAnUnknownWordForSure
Mapped back to word id: [1]
Mapped to [UNK]'s?: True
```

To facilitate batching sentences of different lengths into the same tensor as we'll see later, we also reserved a special padding symbol `[PAD]`.

[20]:
```
print (f"Padding token: {pad_token}")
pad_index = text_vocab[pad_token]
print (f"Padding token id: {pad_index}")
```

```
Padding token: [PAD]
Padding token id: 0
```

2.3 Batching the data

To load data in batches, we use `torch.utils.data.DataLoader`. This enables us to iterate over the dataset under a given `BATCH_SIZE` which specifies how many examples we want to process at a time.

[21]:
```
BATCH_SIZE = 32

# Defines how to batch a list of examples together
def collate_fn(examples):
    batch = {}
    bsz = len(examples)
    label_ids = []
    for example in examples:
        label_ids.append(example['label_id'])
    label_batch = torch.LongTensor(label_ids).to(device)
    input_ids = []
    for example in examples:
        input_ids.append(example['input_ids'])
    max_length = max([len(word_ids) for word_ids in input_ids])
    text_batch = torch.zeros(bsz, max_length).long().fill_(pad_index).to(device)
    for b in range(bsz):
        text_batch[b][:len(input_ids[b])] = torch.LongTensor(input_ids[b]).
 ↪to(device)

    batch['label_ids'] = label_batch
```

```
    batch['input_ids'] = text_batch
    return batch

train_iter = torch.utils.data.DataLoader(train_data, batch_size=BATCH_SIZE,␣
 ↪collate_fn=collate_fn)
val_iter =   torch.utils.data.DataLoader(val_data,   batch_size=BATCH_SIZE,␣
 ↪collate_fn=collate_fn)
test_iter =  torch.utils.data.DataLoader(test_data,  batch_size=BATCH_SIZE,␣
 ↪collate_fn=collate_fn)
```

Let's look at a single batch from one of these iterators.

[22]:
```
batch = next(iter(train_iter))
text = batch['input_ids']
print (f"Size of text batch: {text.size()}")
print (f"Third sentence in batch: {text[2]}")
print (f"Mapped back to string: {hf_tokenizer.decode(text[2])}")
print (f"Mapped back to string skipping padding: {hf_tokenizer.decode(text[2],␣
 ↪skip_special_tokens=True)}")

label = batch['label_ids']
label_vocab_itos = train_data.features['label_id']._int2str # map from label␣
 ↪ids to strs
print (f"Size of label batch: {label.size()}")
print (f"Third label in batch: {label[2]}")
print (f"Mapped back to string: {label_vocab_itos[label[2].item()]}")
```

```
Size of text batch: torch.Size([32, 31])
Third sentence in batch: tensor([  7,   4,   3, 180,   2, 114,   6, 119,   0,
0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0])
Mapped back to string: what flights from tacoma to orlando on saturday [PAD]
[PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
[PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
Mapped back to string skipping padding: what flights from tacoma to orlando on
saturday
Size of label batch: torch.Size([32])
Third label in batch: 15
Mapped back to string: flight_id
```

You might notice some padding tokens `[PAD]` when we convert word ids back to strings, or equivalently, padding ids `0` in the corresponding tensor. The reason why we need such padding is because the sentences in a batch might be of different lengths, and to save them in a 2D tensor for parallel processing, sentences that are shorter than the longest sentence need to be padded with some placeholder values. Later during training you'll need to make sure that the paddings do not affect the final results.

Alternatively, we can also directly iterate over the individual examples in `train_data`, `val_data`

and `test_data`. Here the returned values are the raw sentences and labels instead of their corresponding ids, and you might need to explicitly deal with the unknown words, unlike using bucket iterators which automatically map unknown words to an unknown word id.

[23]:
```
for _, example in zip(range(5), train_data):
  print(f"{example['label']:10} -- {example['text']}")
```

```
flight_id  -- list all the flights that arrive at general mitchell international
from various cities
flight_id  -- give me the flights leaving denver august ninth coming back to
boston
flight_id  -- what flights from tacoma to orlando on saturday
fare_id    -- what is the most expensive one way fare from boston to atlanta on
american airlines
flight_id  -- what flights return from denver to philadelphia on a saturday
```

2.4 Notations used

In this project segment, we'll use the following notations.

- Sequences of elements (vectors and the like) are written with angle brackets and commas ($\langle w_1, \ldots, w_M \rangle$) or directly with no punctuation ($w_1 \cdots w_M$).
- Sets are notated similarly but with braces, ($\{v_1, \ldots, v_V\}$).
- Maximum indices (M, N, V, T, and X in the following) are written as uppercase italics.
- Variables over sequences and sets are written in boldface ($\mathbf{w}$), typically with the same letter as the variables over their elements.

In particular,

- $\mathbf{w} = w_1 \cdots w_M$: A text to be classified, each element w_j being a word token.
- $\mathbf{v} = \{v_1, \ldots, v_V\}$: A vocabulary, each element v_k being a word type.
- $\mathbf{x} = \langle x_1, \ldots, x_X \rangle$: Input features to a model.
- $\mathbf{y} = \{y_1, \ldots, y_N\}$: The output classes of a model, each element y_i being a class label.
- $\mathbf{T} = \langle \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(T)} \rangle$: The training corpus of texts.
- $\mathbf{Y} = \langle y^{(1)}, \ldots, y^{(T)} \rangle$: The corresponding gold labels for the training examples in T.

3 To Do: Establish a majority baseline

A simple baseline for classification tasks is to always predict the most common class. Given a training set of texts $\mathbf{T}$ labeled by classes $\mathbf{Y}$, we classify an input text $\mathbf{w} = w_1 \cdots w_M$ as the class y_i that occurs most frequently in the training data, that is, specified by

$$\operatorname*{argmax}_i \sharp(y_i)$$

and thus ignoring the input entirely (!).

Implement the majority baseline and compute test accuracy using the starter code below. For this baseline, and for the naive Bayes classifier later, we don't need to use the validation set since we don't tune any hyper-parameters.

[24]:
```
# TODO
def majority_baseline_accuracy(train_data, test_data):
    """Returns the most common label in the training set, and the accuracy of
     the majority baseline on the test set.
    """
    (label_id, _) = torch.mode(torch.tensor(train_data["label_id"]))
    mode_label = id_to_label[label_id.item()]
    matched = torch.sum((torch.tensor(test_data["label_id"]) == label_id).int())
    return mode_label, matched / len(test_data)
```

How well does your classifier work? Let's see:

[25]:
```
# Call the method to establish a baseline
most_common_label, test_accuracy = majority_baseline_accuracy(train_data,␣
 ↪test_data)
# For comparison, evaluate it on the training data as well
_, train_accuracy = majority_baseline_accuracy(train_data, train_data)

print(f'Most common label: {most_common_label}\n'
      f'Test accuracy:     {test_accuracy:.3f}\n'
      f'Train accuracy:    {train_accuracy:.3f}')
```

```
Most common label: flight_id
Test accuracy:     0.683
Train accuracy:    0.733
```

4 To Do: Implement a Naive Bayes classifier

4.1 Review of the naive Bayes method

Recall from lab 1-3 that the Naive Bayes classification method classifies a text $\mathbf{w} = \langle w_1, w_2, \ldots, w_M \rangle$ as the class y_i given by the following maximization:

$$\operatorname*{argmax}_i \Pr(y_i \mid \mathbf{w}) \approx \operatorname*{argmax}_i \Pr(y_i) \cdot \prod_{j=1}^{M} \Pr(w_j \mid y_i)$$

or equivalently (since taking the log is monotonic)

$$\operatorname*{argmax}_i \Pr(y_i \mid \mathbf{w}) = \operatorname*{argmax}_i \log \Pr(y_i \mid \mathbf{w}) \tag{1}$$

$$\approx \operatorname*{argmax}_i \left(\log \Pr(y_i) + \sum_{j=1}^{M} \log \Pr(w_j \mid y_i) \right) \tag{2}$$

All we need, then, to apply the Naive Bayes classification method is values for the various log probabilities: the priors $\log \Pr(y_i)$ and the likelihoods $\log \Pr(w_j \mid y_i)$, for each feature (word) w_j and each class y_i.

We can estimate the prior probabilities $\Pr(y_i)$ by examining the empirical probability in the training set. That is, we estimate

$$\Pr(y_i) \approx \frac{\sharp(y_i)}{\sum_j \sharp(y_j)}$$

We can estimate the likelihood probabilities $\Pr(w_j \,|\, y_i)$ similarly by examining the empirical probability in the training set. That is, we estimate

$$\Pr(w_j \,|\, y_i) \approx \frac{\sharp(w_j, y_i)}{\sum_{j'} \sharp(w_{j'}, y_i)}$$

To allow for cases in which the count $\sharp(w_j, y_i)$ is zero, we can use a modified estimate incorporating add-δ smoothing:

$$\Pr(w_j \,|\, y_i) \approx \frac{\sharp(w_j, y_i) + \delta}{\sum_{j'} \sharp(w_{j'}, y_i) + \delta \cdot V}$$

4.2 Two conceptions of the naive Bayes method implementation

We can store all of these parameters in different ways, leading to two different implementation conceptions. We review two conceptions of implementing the naive Bayes classification of a text $\mathbf{w} = \langle w_1, w_2, \ldots, w_M \rangle$, corresponding to using different representations of the input $\mathbf{x}$ to the model: the index representation and the bag-of-words representation.

Within each conception, the parameters of the model will be stored in one or more matrices. The conception dictates what operations will be performed with these matrices.

4.2.1 Using the index representation

In the first conception, we take the input elements $\mathbf{x} = \langle x_1, x_2, \ldots, x_M \rangle$ to be the *vocabulary indices* of the words $\mathbf{w} = w_1 \cdots w_M$. That is, each word token w_i is of the word type in the vocabulary $\mathbf{v}$ at index x_i, so

$$v_{x_i} = w_i$$

In this representation, the input vector has the same length as the word sequence.

We think of the likelihood probabilities as forming a matrix, call it $\mathbf{L}$, where the i, j-th element stores $\log \Pr(v_j \,|\, y_i)$.

$$\mathbf{L}_{ij} = \log \Pr(v_j \,|\, y_i)$$

Similarly, for the priors, we'll have

$$\mathbf{P}_i = \log \Pr(y_i)$$

Now the maximization can be implemented as

$$\underset{i}{\text{argmax}} \log \Pr(y_i) + \sum_{j=1}^{M} \log \Pr(w_j \,|\, y_i) = \underset{i}{\text{argmax}} \mathbf{P}_i + \sum_{j=1}^{M} \mathbf{L}_{i,x_j} \tag{3}$$

Implemented in this way, we see that the use of each input x_i is as an *index* into the likelihood matrix.

4.2.2 Using the bag-of-words representation

Notice that since each word in the input is treated separately, the order of the words doesn't matter. Rather, all that matters is how frequently each word type occurs in a text. Consequently, we can use the bag-of-words representation introduced in lab 1-1.

Recall that the bag-of-words representation of a text is just its frequency distribution over the vocabulary, which we will notate $bow(\mathbf{w})$. Given a vocabulary of word types $\mathbf{v} = \langle v_1, v_2, \ldots, v_V \rangle$, the representation of a sentence $\mathbf{w} = \langle w_1, w_2, \ldots, w_M \rangle$ is a vector $\mathbf{x}$ of size V, where

$$bow(\mathbf{w})_j = \sum_{i=1}^{M} 1[w_i = v_j] \quad \text{for } 1 \leq j \leq V$$

We write $1[w_i = v_j]$ to indicate 1 if $w_i = v_j$ and 0 otherwise. For convenience, we'll add an extra $(V+1)$-st element to the end of the bag-of-words vector, a single 1 whose use will be clear shortly. That is,

$$bow(\mathbf{w})_{V+1} = 1$$

Under this conception, then, we'll take the input $\mathbf{x}$ to be $bow(\mathbf{w})$. Instead of the input having the same length as the text, it has the same length as the vocabulary.

As described in lecture, represented in this way, the quantity to be maximized in the naive Bayes method

$$\log \Pr(y_i) + \sum_{j=1}^{M} \log \Pr(w_j \,|\, y_i)$$

can be calculated as

$$\log \Pr(y_i) + \sum_{j=1}^{V} x_j \cdot \log \Pr(v_j \,|\, y_i)$$

which is just $\mathbf{U}\mathbf{x}$ for a suitable choice of $N \times (V+1)$ matrix $\mathbf{U}$, namely

$$\mathbf{U}_{ij} = \begin{cases} \log \Pr(v_j \,|\, y_i) & 1 \leq i \leq N \text{ and } 1 \leq j \leq V \\ \log \Pr(y_i) & 1 \leq i \leq N \text{ and } j = V+1 \end{cases}$$

Under this implementation conception, we've reduced naive Bayes calculations to a single matrix operation. This conception is depicted in the figure at right.

You are free to use either conception in your implementation of naive Bayes.

4.3 Implement the naive Bayes classifier

For the implementation, we ask you to implement a Python class `NaiveBayes` that will have (at least) the following three methods:

1. `__init__`: An initializer that takes `text_vocab`, `label_vocab`, and `pad_index` as inputs.
2. `train`: A method that takes a training data iterator and estimates all of the log probabilities $\log \Pr(y_i)$ and $\log \Pr(v_j \mid y_i)$ as described above. Perform add-δ smoothing with $\delta = 1$. These parameters will be used by the `evaluate` method to evaluate a test dataset for accuracy, so you'll want to store them in some data structures in objects of the class.
3. `evaluate`: A method that takes a test data iterator and evaluates the accuracy of the trained model on the test set.

You can organize your code using either of the conceptions of Naive Bayes described above.

You should expect to achieve about an **86% test accuracy** on the ATIS task.

[26]:
```
class NaiveBayes():
    def __init__ (self, text_vocab: Dict[str, int], label_vocab: Dict[str,␣
↪int], pad_index: int):
        self.pad_index = pad_index
        self.V = len(text_vocab) # vocabulary size
        self.N = len(label_vocab) # the number of classes

        self.delta_smooth = 1
        self.scores: torch.Tensor = None


    def train(self, iterator: torch.utils.data.DataLoader):
        """Calculates and stores log probabilities for training dataset␣
↪`iterator`."""
        label_cts = torch.zeros(self.N)
        text_cts = torch.zeros(self.N, self.V + 1)

        for group in iter(iterator):
            label_cts += torch.bincount(group["label_ids"], minlength=self.N)
            for idx, label_id in enumerate(group["label_ids"]):
                text_cts[label_id] += torch.bincount(group["input_ids"][idx],␣
↪minlength=(self.V + 1))

        text_cts[:, self.pad_index] = 0
        row_totals = (torch.sum(text_cts, 1) + (self.delta_smooth * self.V)).
↪view(-1, 1)
```

```
        text_scores = torch.log((text_cts + self.delta_smooth) / row_totals)

        priors = torch.log(label_cts / torch.sum(label_cts))
        text_scores[:, self.V] = priors

        self.scores = text_scores

        return self


    def evaluate(self, iterator: torch.utils.data.DataLoader):
        """Returns the model's accuracy on a given dataset `iterator`."""
        correct = 0
        total = 0
        for group in iter(iterator):
            for idx, label_id in enumerate(group["label_ids"]):
                total += 1
                input_ids = group["input_ids"][idx]
                bow = torch.bincount(input_ids, minlength=(self.V + 1))
                bow[self.pad_index] = 0
                bow[self.V] = 1
                correct += int(torch.argmax((self.scores * bow).sum(1)) ==␣
↪label_id)
        return correct / total
```

[ ]:

[27]:
```
# Instantiate and train classifier
nb_classifier = NaiveBayes(text_vocab, label_vocab, pad_index)
nb_classifier.train(train_iter)

# Evaluate model performance
print(f'Training accuracy: {nb_classifier.evaluate(train_iter):.3f}\n'
      f'Test accuracy:     {nb_classifier.evaluate(test_iter):.3f}')
```

```
Training accuracy: 0.897
Test accuracy:     0.864
```

5 To Do: Implement a logistic regression classifier

In this part, you'll complete a PyTorch implementation of a logistic regression (equivalently, a single layer perceptron) classifier. We review logistic regression here highlighting the similarities to the matrix-multiplication conception of naive Bayes. Thus, we take the input $\mathbf{x}$ to be the bag-of-words representation $bow(\mathbf{w})$. But as before you are free to use either implementation approach.

5.1 Review of logistic regression

Similar to naive Bayes, in logistic regression, we assign a probability to a text $\mathbf{x}$ by merely multiplying an $N \times V$ matrix $\mathbf{U}$ by it. However, we don't stipulate that the values in the matrix $\mathbf{U}$ be estimated from the training corpus in the "naive Bayes" manner. Instead, we allow them to take on any value, using a training regime to select good values.

In order to make sure that the output of the matrix multiplication $\mathbf{Ux}$ is mapped onto a probability distribution, we apply a nonlinear function to renormalize the values. We use the softmax function, a generalization of the sigmoid function from lab 1-4, defined by

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{N} \exp(z_j)}$$

for each of the indices i from 1 to N.

In summary, we model $\Pr(y \mid \mathbf{x})$ as

$$\Pr(y_i \mid \mathbf{x}) = \text{softmax}(\mathbf{Ux})_i$$

The calculation of $\Pr(y \mid \mathbf{x})$ for each text $\mathbf{x}$ is referred to as the *forward* computation. In summary, the forward computation for logistic regression involves a linear calculation ($\mathbf{Ux}$) followed by a nonlinear calculation (softmax). We think of the perceptron (and more generally many of these neural network models) as transforming from one representation to another. A perceptron performs a linear transformation from the index or bag-of-words representation of the text to a representation as a vector, followed by a nonlinear transformation, a softmax or sigmoid, giving a representation as a probability distribution over the class labels. This single-layer perceptron thus involves two *sublayers*. (In the next part of the project segment, you'll experiment with a multilayer perceptron, with two perceptron layers, and hence four sublayers.)

The loss function you'll use is the negative log probability $-\log \Pr(y \mid \mathbf{x})$. The negative is used, since it is convention to minimize loss, whereas we want to maximize log likelihood.

The forward and loss computations are illustrated in the figure at right. In practice, for numerical stability reasons, PyTorch absorbs the softmax operation into the loss function `nn.CrossEntropyLoss`. That is, the input to the `nn.CrossEntropyLoss` function is the vector of sums $\mathbf{Ux}$ (the last step in the box marked "your job" in the figure) rather than the vector of probabilities $\Pr(y \mid \mathbf{x})$. That makes things easier for you (!), since you're responsible only for the first sublayer.

Given a forward computation, the weights can then be adjusted by taking a step opposite to the gradient of the loss function. Adjusting the weights in this way is referred to as the *backward* computation. Fortunately, `torch` takes care of the backward computation for you, just as in lab 1-5.

The optimization process of performing the forward computation, calculating the loss, and performing the backward computation to improve the weights is done repeatedly until the process converges on a (hopefully) good set of weights. You'll find this optimization process in the `train_all` method that we've provided. The trained weights can then be used to perform classification on a test set. See the `evaluate` method.

5.2 Implement the logistic regression classifier

For the implementation, we ask you to implement a logistic regression classifier as a subclass of `torch.nn.Module`. You need to implement the following methods:

1. `__init__`: an initializer that takes `text_vocab`, `label_vocab`, and `pad_index` as inputs.

   During initialization, you'll want to define a tensor of weights, wrapped in `torch.nn.Parameter`, initialized randomly, which plays the role of **U**. The elements of this tensor are the parameters of the `torch.nn` instance in the following special technical sense: It is the parameters of the module whose gradients will be calculated and whose values will be updated. Alternatively, **you might find it easier** to use the `nn.Embedding` module which is a wrapper to the weight tensor with a lookup implementation.

2. `forward`: given a text batch of size `batch_size X max_length`, return a tensor of logits of size `batch_size X num_labels`. That is, for each text **x** in the batch and each label y, you'll be calculating **Ux** as shown in the figure, returning a tensor of these values. Note that the softmax operation is absorbed into `nn.CrossEntropyLoss` so you won't need to deal with that.

3. `train_all`: A method that performs training. You might find lab 1-5 useful.

4. `evaluate`: A method that takes a test data iterator and evaluates the accuracy of the trained model on the test set.

Some things to consider:

1. The parameters of the model, the weights, need to be initialized properly. We suggest initializing them to some small random values. See `torch.uniform_`.

2. You'll want to make sure that padding tokens are handled properly. What should the weight be for the padding token?

3. In extracting the proper weights to sum up, based on the word types in a sentence, we are essentially doing a lookup operation. You might find `nn.Embedding` or `torch.gather` useful.

You should expect to achieve about **90%** accuracy on the ATIS classificiation task.

[28]:
```
class LogisticRegression(nn.Module):
    def __init__(self, text_vocab, label_vocab, pad_index):
        super().__init__()
        self.pad_index = pad_index
        # Keep the vocabulary sizes available
        self.N = len(label_vocab)  # num_classes
        self.V = len(text_vocab)  # vocab_size
        # Specify cross-entropy loss for optimization
        self.criterion = nn.CrossEntropyLoss()
        # TODO: Create and initialize a tensor for the weights,
        #       or create an nn.Embedding module and initialize
        embedding_dim = 100
        self.embedding = torch.nn.Embedding(self.V, embedding_dim)
        self.weights = torch.nn.Parameter(torch.Tensor(embedding_dim, self.N).
 ↪uniform_())
```

```
    def forward(self, text_batch: torch.Tensor) -> torch.Tensor:
        # TODO: Calculate the logits (Ux) for the `text_batch`,
        #       returning a tensor of size batch_size x num_labels
        # (# samples, # words, 1)
        pad_mask = (text_batch != self.pad_index).float().unsqueeze(2)
        # (# samples, # words, embedding size)
        text_embeddings = self.embedding(text_batch)
        # (#samples, embedding size) - mega embedding representing the entire␣
↪word sequence
        masked_emb_groups = (text_embeddings * pad_mask).mean(dim=1)
        # (#samples, #classes)
        return torch.mm(masked_emb_groups, self.weights)

    def train_all(self, train_iter, val_iter, epochs=8, learning_rate=3e-3):
        # Use Adam to optimize the parameters
        optim = torch.optim.Adam(self.parameters(), lr=learning_rate)
        best_validation_accuracy = -float('inf')
        best_model = None

        # Run the optimization for multiple epochs
        with tqdm(range(epochs), desc='train', position=0) as pbar:
            for epoch in pbar:

                # Switch the module to training mode (each epoch, since
                # `self.evaluate` call below resets it to evaluation mode)
                self.train()

                c_num = 0
                total = 0
                running_loss = 0.0

                for batch in tqdm(train_iter, desc='batch', leave=False):
                    # TODO: set labels, compute logits (Ux in this model),
                    #       loss, and update parameters
                    optim.zero_grad()

                    labels = batch["label_ids"]
                    logits = self.forward(batch["input_ids"])
                    loss = self.criterion(logits, labels)

                    loss.backward()
                    optim.step()

                    # Prepare to compute the accuracy
                    predictions = torch.argmax(logits, dim=1)
                    total += predictions.size(0)
```

```
                    c_num += (predictions == labels).float().sum().item()
                    running_loss += loss.item() * predictions.size(0)

                # Evaluate and track improvements on the validation dataset
                validation_accuracy = self.evaluate(val_iter)
                if validation_accuracy > best_validation_accuracy:
                    best_validation_accuracy = validation_accuracy
                    self.best_model = copy.deepcopy(self.state_dict())
                epoch_loss = running_loss / total
                epoch_acc = c_num / total
                pbar.set_postfix(epoch=epoch+1, loss=epoch_loss,
                                 train_acc=epoch_acc,␣
↪val_acc=validation_accuracy)
        return self

    def evaluate(self, iterator: torch.utils.data.DataLoader):
        """Returns the model's accuracy on a given dataset `iterator`."""
        self.eval()    # switch the module to evaluation mode
        total = 0
        correct = 0
        for sample in iter(iterator):
            total += len(sample["label_ids"])
            fwd = self.forward(sample["input_ids"])
            decisions = torch.argmax(fwd, dim=1)
            correct += (decisions == sample["label_ids"]).int().sum()
        return correct / total

    def explore_failures(self, iterator: torch.utils.data.DataLoader,␣
↪label_vocab: Dict[str, int]):
        self.eval()
        for sample in iter(iterator):
            fwd = self.forward(sample["input_ids"])
            decisions = torch.argmax(fwd, dim=1)
            for idx, correct in enumerate(sample["label_ids"]):
                if correct == decisions[idx]:
                    continue
                words = tokenizer.decode(list(sample["input_ids"][idx]))
                expected_label = None
                got_label = None
                for label, l_id in label_vocab.items():
                    if l_id == correct:
                        expected_label = label
                    elif l_id == decisions[idx]:
                        got_label = label
                print(f"expected {expected_label}, got {got_label}: {words}")
```

[29]:
```
# Instantiate the logistic regression classifier and run it
model = LogisticRegression(text_vocab, label_vocab, pad_index).to(device)
model.train_all(train_iter, val_iter)
model.load_state_dict(model.best_model)
test_accuracy = model.evaluate(test_iter)
print (f'Test accuracy: {test_accuracy:.4f}')
```

```
train:   0%|          | 0/8 [00:00<?, ?it/s]

batch:   0%|          | 0/137 [00:00<?, ?it/s]

batch:   0%|          | 0/137 [00:00<?, ?it/s]

batch:   0%|          | 0/137 [00:00<?, ?it/s]

batch:   0%|          | 0/137 [00:00<?, ?it/s]

batch:   0%|          | 0/137 [00:00<?, ?it/s]

batch:   0%|          | 0/137 [00:00<?, ?it/s]

batch:   0%|          | 0/137 [00:00<?, ?it/s]

batch:   0%|          | 0/137 [00:00<?, ?it/s]

Test accuracy: 0.9308
```

[30]:
```
model.explore_failures(test_iter, label_vocab)
```

```
expected fare_id, got flight_id: i need the fares on flights from washington to
toronto on a saturday
expected fare_id, got flight_id: get saturday fares from washington to boston
expected fare_id, got flight_id: get saturday fares from washington to montreal
expected fare_id, got flight_id: get saturday fares from washington to toronto
expected fare_id, got flight_id: get the saturday fare from washington to
toronto
expected flight_number, got flight_id: i need flight numbers and airlines for
flights departing from oakland to salt lake city on thursday departing before
8am
expected basic_type, got aircraft_code: what type of aircraft are flying from
cleveland to dallas before noon
expected fare_id, got flight_id: i need a ticket from nashville to seattle
expected fare_id, got flight_id: i need a ticket from nashville tennessee to
seattle
expected fare_id, got flight_id: i ' d like a one way ticket from milwaukee to
orlando either wednesday evening or thursday morning
expected booking_class, got fare_basis_code: what does fare code f mean
expected booking_class, got fare_basis_code: what does fare code h mean
expected booking_class, got fare_basis_code: what does fare code f mean
expected booking_class, got fare_basis_code: what does fare code h mean
expected miles_distant, got flight_id: list distance from airports to downtown
in new york
expected city_code, got flight_id: list la
```

```
expected city_code, got flight_id: list la
expected basic_type, got aircraft_code: list type of aircraft for all flights
from charlotte
expected fare_id, got flight_id: list for first class round trip from detroit to
st . petersburg
expected aircraft_code, got flight_id: list seating of delta flights from
seattle to salt lake city
expected city_code, got airport_code: what cities does northwest fly out of
expected city_code, got flight_id: list the cities from which northwest flies
expected city_code, got flight_id: what cities does northwest fly to
expected aircraft_code, got airport_code: what is a
expected aircraft_code, got airport_code: what is a
expected meal_code, got city_code: are served on air
expected airline_code, got flight_id: list the airlines with flights to or from
denver
expected aircraft_code, got airport_code: what is
expected state_code, got transport_type: is there ground transportation from the
memphis airport into when if i arrive at in the morning
expected airline_code, got flight_id: is there one airline that flies from
burbank to milwaukee milwaukee to st . louis and from st . louis to burbank
expected airport_code, got flight_id: tell me all the airports in the new york
city area
```

6 To Do: Implement a multilayer perceptron

6.1 Review of multilayer perceptrons

In the last part, you implemented a perceptron, a model that involved a linear calculation (the sum of weights) followed by a nonlinear calculation (the softmax, which converts the summed weight values to probabilities). In a multi-layer perceptron, we take the output of the first perceptron to be the input of a second perceptron (and of course, we could continue on with a third or even more).

In this part, you'll implement the forward calculation of a two-layer perceptron, again letting PyTorch handle the backward calculation as well as the optimization of parameters. The first layer will involve a linear summation as before and a **sigmoid** as the nonlinear function. The second will involve a linear summation and a softmax (the latter absorbed, as before, into the loss function). Thus, the difference from the logistic regression implementation is simply the adding of the sigmoid and second linear calculations. See the figure for the structure of the computation.

6.2 Implement a multilayer perceptron classifier

For the implementation, we ask you to implement a two layer perceptron classifier, again as a subclass of the `torch.nn` module. You might reuse quite a lot of the code from logistic regression. As before, you need to implement the following methods:

1. `__init__`: An initializer that takes `text_vocab`, `label_vocab`, `pad_index`, and `hidden_size` specifying the size of the hidden layer (e.g., in the above illustration, `hidden_size` is `D`).

During initialization, you'll want to define two tensors of weights, which serve as the parameters of this model, one for each layer. You'll want to initialize them randomly.

The weights in the first layer are a kind of lookup (as in the previous part), mapping words to a vector of size `hidden_size`. The `nn.Embedding` module is a good way to set up and make use of this weight tensor.

The weights in the second layer define a linear mapping from vectors of size `hidden_size` to vectors of size `num_labels`. The `nn.Linear` module or `torch.mm` for matrix multiplication may be helpful here.

2. `forward`: Given a text batch of size `batch_size X max_length`, the `forward` function returns a tensor of logits of size `batch_size X num_labels`.

   That is, for each text $\mathbf{x}$ in the batch and each label c, you'll be calculating $MLP(bow(\mathbf{x}))$ as shown in the illustration above, returning a tensor of these values. Note that the softmax operation is absorbed into `nn.CrossEntropyLoss` so you don't need to worry about that.

   For the sigmoid sublayer, you might find `nn.Sigmoid` useful.

3. `train_all`: A method that performs training. You might find lab 1-5 useful.

4. `evaluate`: A method that takes a test data iterator and evaluates the accuracy of the trained model on the test set.

You should expect to achieve at least **90%** accuracy on the ATIS classificiation task.

[31]:
```
class MultiLayerPerceptron(nn.Module):
    def __init__(self, text_vocab, label_vocab, pad_index, hidden_size=128):
        super().__init__()
        self.pad_index = pad_index
        self.hidden_size = hidden_size
        # Keep the vocabulary sizes available
        self.N = len(label_vocab)  # num_classes
        self.V = len(text_vocab)  # vocab_size
        # Specify cross-entropy loss for optimization
        self.criterion = nn.CrossEntropyLoss()
        # TODO: Create and initialize neural modules
        self.embedding = torch.nn.Embedding(self.V, hidden_size,␣
↪padding_idx=pad_index)
        self.layer1 = torch.nn.Linear(hidden_size, hidden_size)
        self.sigmoid = torch.nn.Sigmoid()
        self.layer2 = torch.nn.Linear(hidden_size, self.N)

    def forward(self, text_batch):
        # TODO: Calculate the logits for the `text_batch`,
        #       returning a tensor of size batch_size x num_labels
        # (# samples, # words, embedding size)
        embedding = self.embedding(text_batch).mean(dim=1)
        after_layer1 = self.sigmoid(self.layer1(embedding))
        return self.layer2(after_layer1)
```

```
  def train_all(self, train_iter, val_iter, epochs=8, learning_rate=3e-3):
      # Use Adam to optimize the parameters
      optim = torch.optim.Adam(self.parameters(), lr=learning_rate)
      best_validation_accuracy = -float('inf')
      best_model = None
      # Run the optimization for multiple epochs
      with tqdm(range(epochs), desc='train', position=0) as pbar:
          for epoch in pbar:
              # Switch the module to training mode
              self.train()
              c_num = 0
              total = 0
              running_loss = 0.0
              for batch in tqdm(train_iter, desc='batch', leave=False):
                  # TODO: set labels, compute logits (Ux in this model),
                  #       loss, and update parameters
                  optim.zero_grad()

                  labels = batch["label_ids"]
                  logits = self.forward(batch["input_ids"])
                  loss = self.criterion(logits, labels)

                  loss.backward()
                  optim.step()

                  # Prepare to compute the accuracy
                  predictions = torch.argmax(logits, dim=1)
                  total += predictions.size(0)
                  c_num += (predictions == labels).float().sum().item()
                  running_loss += loss.item() * predictions.size(0)

              # Evaluate and track improvements on the validation dataset
              validation_accuracy = self.evaluate(val_iter)
              if validation_accuracy > best_validation_accuracy:
                  best_validation_accuracy = validation_accuracy
                  self.best_model = copy.deepcopy(self.state_dict())
              epoch_loss = running_loss / total
              epoch_acc = c_num / total
              pbar.set_postfix(epoch=epoch+1, loss=epoch_loss,
                               train_acc=epoch_acc,␣
↪val_acc=validation_accuracy)
      return self

  def evaluate(self, iterator: torch.utils.data.DataLoader):
      """Returns the model's accuracy on a given dataset `iterator`."""
```

```
        self.eval()   # switch the module to evaluation mode
        total = 0
        correct = 0
        for sample in iter(iterator):
            total += len(sample["label_ids"])
            fwd = self.forward(sample["input_ids"])
            decisions = torch.argmax(fwd, dim=1)
            correct += (decisions == sample["label_ids"]).int().sum()
        return correct / total
```

[32]:
```
# Instantiate classifier and run it
model = MultiLayerPerceptron(text_vocab, label_vocab, pad_index,␣
 ↪hidden_size=128).to(device)
model.train_all(train_iter, val_iter)
model.load_state_dict(model.best_model)
test_accuracy = model.evaluate(test_iter)
print (f'Test accuracy: {test_accuracy:.4f}')
```

```
train:   0%|          | 0/8 [00:00<?, ?it/s]
batch:   0%|          | 0/137 [00:00<?, ?it/s]
batch:   0%|          | 0/137 [00:00<?, ?it/s]
batch:   0%|          | 0/137 [00:00<?, ?it/s]
batch:   0%|          | 0/137 [00:00<?, ?it/s]
batch:   0%|          | 0/137 [00:00<?, ?it/s]
batch:   0%|          | 0/137 [00:00<?, ?it/s]
batch:   0%|          | 0/137 [00:00<?, ?it/s]
batch:   0%|          | 0/137 [00:00<?, ?it/s]
batch:   0%|          | 0/137 [00:00<?, ?it/s]
Test accuracy: 0.9174
```

7 Lessons learned

Question: Rank the methods as to their performance. What lessons do you learn from the ranking?

For me, performance goes in the order Majority -> Naive Bayes -> Multi Layer Perceptron -> Logistic Regression, with the last two almost identical. If my models are correctly implemented, then my conclusion is that a more complex model doesn't always perform better on a given training + test set. A similar observation I made is that ramping some parameters doesn't really help model performance. For example, making my embeddings >125 didn't really improve logistic regression, and increasing the number of epochs to 20 didn't really improve MLP.

Question: Take a look at some of the examples that were classified correctly and incorrectly by your best method. Do you notice anything about the incorrectly classified examples that might indicate *why* they were classified incorrectly?

Not all, but a decent number of the test sequences that failed were super ambiguous. For example "what is", "what is a", "are served on air". A lot of the other failures used word choice that makes even me struggle to classify. For example "list the airline flights from burbank" was supposed to yield flight_id, but my classifier gave it airline_code. However, the text uses both "airline" and "flights". Since "airline" is probably a lot more common in airline queries, it's reasonable that my classifier gave it an airline assignment. Another example is "what does fare code f mean". It wanted booking_class, but my classifier gave it fare_basis_code. The text uses both "fare" and "code", so I don't really fault the classifier for assigning fare_basis_code. Overall, my conclusion from the failures is that text classification can have some really tricky outliers, and it makes sense that neither Logistic Regression nor MLR were really able to crack past ~93% accuracy.

[33]:
```
...
```

[33]:
```
Ellipsis
```

8 Debrief

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

- Was the project segment clear or unclear? Which portions?
- Were the readings appropriate background for the project segment?
- Are there additions or changes you think would make the project segment better?

I really like the structure of this PSET, and I think I learned a lot from it. My main request is that there might be more ways to validate that what I have is correct. I reached ~85% accuracy on a Logistic Regression attempt with a completely wrong fundamental mechanic, so it seems there are more ways than one to build models that approach target accuracy. Even now I'm not completely sure I implemented the models correctly, I just tried stuff until they came out performing >90%. If they need to be implemented correctly to meet that level, a comment about that might be nice. Also, if the pset is released on Thursday or Friday, at least one weekend office hours would be helpful.

9 Instructions for submission of the project segment

This project segment should be submitted to Gradescope at http://go.cs187.info/project1-submit-code and http://go.cs187.info/project1-submit-pdf, which will be made available some time before the due date.

Project segment notebooks are manually graded, not autograded using otter as labs are. (Otter is used within project segment notebooks to synchronize distribution and solution code however.) **We will not run your notebook before grading it.** Instead, we ask that you submit the already freshly run notebook. The best method is to "restart kernel and run all cells", allowing time for all cells to be run to completion. You should submit your code to Gradescope at the code submission assignment at http://go.cs187.info/project1-submit-code.

We also request that you **submit a PDF of the freshly run notebook**. The simplest method is to use "Export notebook to PDF", which will render the notebook to PDF via LaTeX. If that

doesn't work, the method that seems to be most reliable is to export the notebook as HTML (if you are using Jupyter Notebook, you can do so using `File -> Print Preview`), open the HTML in a browser, and print it to a file. Then make sure to add the file to your git commit. Please name the file the same name as this notebook, but with a `.pdf` extension. (Conveniently, the methods just described will use that name by default.) You can then perform a git commit and push and submit the commit to Gradescope at http://go.cs187.info/project1-submit-pdf.

End of project segment 1