CS187 Project Segment 1: Text Classification

May 1, 2023

[ ]:

```python
# 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/cs236299-2023-spring/project1.git .tmp
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

[ ]:

```python
# Initialize Otter
import otter
grader = otter.Notebook()
```

1 Preparation

[ ]:

```
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
```

[ ]:

```
# 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)
```

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.

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 ,
   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.

[ ]:

```python
data_dir = "https://raw.githubusercontent.com/nlp-236299/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/')
```

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

[ ]:

```python
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 the data file looks like.

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

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.

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

[ ]:
```
atis
```

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

train_data.shuffle(seed=random_seed)
```

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

[ ]:
```
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.

[ ]:

```
hf_tokenizer = PreTrainedTokenizerFast(tokenizer_object=tokenizer,␣
 ↪pad_token=pad_token, unk_token=unk_token)
```

[ ]:

```
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)
```

We also need to convert label strings into label ids.

[ ]:

```
# 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")
```

[ ]:

```
# 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}")
```

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.

[ ]:

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

2.2 Handling unknown words

Note that we mapped words appearing fewer than 3 times to a special *unknown* token (we're using `[UNK]`) 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].

[ ]:
```
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])}")
```

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

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

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.

[ ]:
```
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.

[ ]:

```
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()]}")
```

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.

[ ]:

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

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.

[ ]:

```
# 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.
  """
  ...
  return most_common_label, test_accuracy
```

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

[ ]:

```
# Call the method to establish a baseline
most_common_label, test_accuracy = majority_baseline_accuracy(train_data,␣
 ↪test_data)

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

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)

$$\begin{aligned} \operatorname*{argmax}_i \Pr(y_i \mid \mathbf{w}) &= \operatorname*{argmax}_i \log \Pr(y_i \mid \mathbf{w}) && (1) \\ &\approx \operatorname*{argmax}_i \left(\log \Pr(y_i) + \sum_{j=1}^{M} \log \Pr(w_j \mid y_i) \right) && (2) \end{aligned}$$

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 $\Pr(w_j \mid y_i)$ similarly by examining the empirical probability in the training set. That is, we estimate

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

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:

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

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 $\log \Pr(v_j \mid y_i)$.

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

Similarly, for the priors, we'll have

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

Now the maximization can be implemented as

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

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

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

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

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

can be calculated as

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

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

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

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.

[ ]:
```
class NaiveBayes():
  def __init__ (self, text_vocab, label_vocab, pad_index):
    self.pad_index = pad_index
    self.V = len(text_vocab) # vocabulary size
    self.N = len(label_vocab) # the number of classes
    # TODO: Add your code here
    ...


  def train(self, iterator):
    """Calculates and stores log probabilities for training dataset `iterator`.
 ↪"""
    # TODO: Implement this method.
    ...

  def evaluate(self, iterator):
    """Returns the model's accuracy on a given dataset `iterator`."""
    # TODO: Implement this method.
    ...
```

[ ]:

```
# 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}')
```

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 $\mathbf{U}\mathbf{x}$ 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

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

for each of the indices i from 1 to N.

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

$$\Pr(y_i \mid \mathbf{x}) = \mathrm{softmax}(\mathbf{U}\mathbf{x})_i$$

The calculation of $\Pr(y \mid \mathbf{x})$ for each text **x** is referred to as the *forward* computation. In summary, the forward computation for logistic regression involves a linear calculation ($\mathbf{U}\mathbf{x}$) 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 $\mathbf{U}\mathbf{x}$ (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 $\mathbf{U}\mathbf{x}$ 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.

[ ]:

```
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
    ...

  def forward(self, text_batch):
    # TODO: Calculate the logits (Ux) for the `text_batch`,
    #       returning a tensor of size batch_size x num_labels
    ...

  def train_all(self, train_iter, val_iter, epochs=8, learning_rate=3e-3):
    # Switch the module to training mode
    self.train()
    # 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:
        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
          ...
          labels = ...
          logits = ...
          loss = ...
          ...
          # 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)

  def evaluate(self, iterator):
    """Returns the model's accuracy on a given dataset `iterator`."""
    self.eval()   # switch the module to evaluation mode
    # TODO: Compute accuracy
    ...
```

[ ]:

```
# 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}')
```

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.

[ ]:

```
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
```

```python
    # Specify cross-entropy loss for optimization
    self.criterion = nn.CrossEntropyLoss()
    # TODO: Create and initialize neural modules
    ...

  def forward(self, text_batch):
    # TODO: Calculate the logits for the `text_batch`,
    #       returning a tensor of size batch_size x num_labels
    ...

  def train_all(self, train_iter, val_iter, epochs=8, learning_rate=3e-3):
    # Switch the module to training mode
    self.train()
    # 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:
        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
          ...
          labels = ...
          logits = ...
          loss = ...
          ...
          # 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)
  def evaluate(self, iterator):
    """Returns the model's accuracy on a given dataset `iterator`."""
    # TODO: Compute accuracy
    ...
```

[ ]:

```
# 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}')
```

7 Lessons learned

Take a look at some of the examples that were classified correctly and incorrectly by your best method.

Question: Do you notice anything about the incorrectly classified examples that might indicate *why* they were classified incorrectly?

Type your answer here, replacing this text.

[ ]:

```
...
```

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:

- 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?

Type your answer here, replacing this text.

9 Instructions for submission of the project segment

This project segment should be submitted to Gradescope at https://rebrand.ly/project1-submit-code and https://rebrand.ly/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 `https://rebrand.ly/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 `https://rebrand.ly/project1-submit-pdf`.

10 End of project segment 1