COMP34112, Natural Language Systems, Coursework1

Andras Geiszl

Please refer to the README for setup instructions.

1) Named-entity recognition

First, we define a method on Corpus class to use the NTLK's built-in tagger to tag the corpus.

```
# corpus.py

def get_named_entities_default(self) -> List[List[Tuple[str, str]]]:
    # Download necessary NLTK models
    print('\nDownloading NLTK resources...')
    nltk.download('punkt')
    nltk.download('averaged_perceptron_tagger')
    nltk.download('maxent_ne_chunker')
    nltk.download('words')

    print('\nExtracting named entities using the default tagger...')
    entities: List[List[Any]] = []
    for text in cast(Any, tqdm(self.texts)):
        # Join lines of the current text and tag them
        tagged_words = nltk.pos_tag(nltk.word_tokenize(' '.join(text)))
        chunks: Any = nltk.ne_chunk(tagged_words)

        # Get the named entities from the tagged entities
        text_entities = []
        for chunk in chunks:
            if type(chunk) is nltk.Tree:
                name = ''.join(leaf[0] for leaf in chunk.leaves())
                text_entities.append((name, chunk.label()))

        entities.append(text_entities)

    return entities
```

Then we also define one for taggging it with Stanford's NER tagger (we use the `stanza` package for this).

```
def get_named_entities_stanford(self) -> List[List[Tuple[str, str]]]:
    print('Downloading Stanford NER resources...')
    stanza.download('en', processors='tokenize,ner')

    tagger = stanza.Pipeline(lang='en', processors='tokenize,ner')

    print('\nExtracting named entities using the Stanford tagger...')
    entities = []
    for text in cast(Any, tqdm(self.texts)):
        # Join and tag each text
        entities.append([(entity.text, entity.type) for entity
            in cast(Any, tagger(' '.join(text)).ents)])

    return entities
```

We call both from the main script (`src/sourcework2.py`) and compare them.

```
# Load the inagural texts
inagural_corpus = Corpus('Inaugural Texts', 'data/inaugural')

# Tag with both NLTK's default and the Stanford named entity tagger
default_tagged = inagural_corpus.get_named_entities_default()
stanford_tagged = inagural_corpus.get_named_entities_stanford()

print(f'Default tagger found {len(list(chain.from_iterable(default_tagged)))} named entities')
# Print named entities of the first text
print(default_tagged[0])

stanford_count = len(list(chain.from_iterable(stanford_tagged)))
print(f'\nStanford tagger found {stanford_count} named entities')
print(stanford_tagged[0])
```

This gives the output:

```
Default tagger found 2073 named entities
[('Senate', 'ORGANIZATION'), ('House', 'ORGANIZATION'), ('AlmightyBeing', 'PERSON'), ('UnitedSt

Stanford tagger found 3432 named entities
[('Senate', 'ORG'), ('the House of Representatives', 'ORG'), ('the 14th day of the present mont
United States', 'GPE'), ('Government', 'ORG'), ('the Invisible Hand', 'LAW'), ('the United Stat
```

We can see that the Stanford tagger gives much better results. It finds more named entities, and the boundary of the entities are better determined. For example, while the default tagger determined the boundary of the entity "the House of Representatives" as "House", the Stanford model was able to recognize the whole entity. Another example is "the United State", which the default tagger recognized as only "United States". The Stanford tagger also recognized some other type of entities (e.g. DATE, EVENT, tec.), which the default couldn't.

However, the Stanford tagger also took more time to finish (default: 3.5min, stanford: 15.5min). This is probably because it's a more complex model and does more operations while tagging the text.

2) Sentiment analysis of movie reviews

Build Sentiment Lexicon

First, we need to load the seed words from a file. Then, we define `build_sentiment_lexicon` to extend this seed with more words from the corpus:

```
# corpus.py

def build_sentiment_lexicon(self, seed: Dict[str, str]) -> Dict[str, str]:
    ...
```

First, we define a few words, which we will use to determine the polarity of a new word

```
    same_polarity_words = ['and', 'also', 'as well as', 'moreover', 'plus']
    opposite_polarity_words = ['but', 'however', 'on the other hand', 'yet', 'still',
        'nevertheless', 'though', 'not']
```

Then we look for all the occurrences of the seed words in the text and try to find others in their close proximity, for which we can determine their polarity.

```
    found_words: List[Tuple[str, str]] = []

    for text in tagged_texts:
        for line in text:
            # Get the text of the word from all the tokens in current line
            words = [word.text for word in line]

            for index, word in enumerate(words):
                # Get new subjective words from the neighbors of the seed words
                if word in seed:
                    ...
```

If we found a word that is in the seed lexicon, use the words above to determine its polarity.

```
if word in seed:
    # Look at the words after the seed word
    if index + 2 < len(line) and line[index + 2].pos_ == 'ADJ':
        if (line[index + 1].text in same_polarity_words):
            found_words.append((words[index + 2].text, seed[word]))
        if line[index + 1].text in opposite_polarity_words:
            found_words.append((words[index + 2].text, (
                'positive' if seed[word] == 'negative' else 'negative'
            )))

    # Look at the words before the seed word
    if index - 2 > 0 and line[index - 2].pos_ == 'ADJ':
        if line[index - 1] in same_polarity_words:
            found_words.append((words[index - 2].text, seed[word]))
        if line[index - 1] in opposite_polarity_words:
            found_words.append((words[index - 2].text, (
                'positive' if seed[word] == 'negative' else 'negative'
            )))
```

We can resolve the possibly conflicting polarities (i.e. a word is determined once to be 'positive' and once to be 'negative') by looking at both of their frequencies and only adding the larger one to the dictionary.

```
    # Add all the words found to the dictionary once
    # The words and their polarity are sorted by the number of occurrences, so if a word appear
    # as both 'positive' and 'negative', the one which appears more frequenty will be added to
    # the dictionary
    dictionary: Dict[str, str] = {}
    for word_tuple in dict(Counter(found_words)):

        if word_tuple[0] not in dictionary:
            dictionary[word_tuple[0]] = word_tuple[1]
```

Another pattern to find more adjectives would be finding all the already known positive and negative words in the context of the current word and deciding on the polarity of the word based on the frequencies of each (we are using `get_word_context` from coursework 1; see `utilites.py` for the implementation).

```
if line[index].pos_ == 'ADJ':
  # Find known words in context of the current adjective and if they are in
  # the seed, then the current word probably also has the same polarity
  polarities_in_context = [seed[context_word] for context_word
      in get_word_context(index, words, 4) if context_word in seed]

  found_words += [(line[index].text, polarity) for polarity
      in polarities_in_context]
```

This builds the following lexicon:

```
{'old': 'positive', 'chilly': 'negative', 'disturbing': 'positive', 'true': 'positive', 'claust
'bizarre': 'positive', 'uplifting': 'positive', 'brassy': 'negative', 'unusual': 'negative', 's
'sensual': 'positive', 'rocky': 'positive', 'enticing': 'positive', 'sexual': 'positive', 'feat
'farcical': 'negative', 'infantile': 'negative', 'inconsequential': 'positive', 'pointless': 'n
'lazy': 'negative', 'manipulative': 'negative', 'strong': 'positive', 'fake': 'negative', 'dish
```


Most of the words are correct, however there are some, which are obviously wrong. This would need more analysis and more sophisticated rules to fix.

Baseline classifier

For the baseline of sentiment classification, we implement a basic model, which uses the number of positive and negative words in the text to determine its polarity.

```
# utilities.py

def classify_sentiment(
    text: str,
    lexicon: SentimentLexicon,
    tagger: Language,
    stemmer: SnowballStemmer,
    is_use_stem: bool = True,
    is_ignore_pos: bool = False,
) -> int:
    polarities: Dict[str, Dict[str, SentimentEntry]] = {'positive': {}, 'negative': {}}

    # Lexicon has different POS format, so we need to convert it to the standard one
    pos_values = {
        'adverb': 'ADV',
        'noun': 'NOUN',
        'verb': 'VERB',
        'adj': 'ADJ',
    }

    for word_token in cast(Any, tagger(text)):
        word = word_token.text
        stemmed_word = cast(Any, stemmer).stem(word)

        # If the current word satisfies the rules, add it to the corresponding polarity dictior
        if (
            word in lexicon
                and lexicon[word]['priorpolarity'] in ['positive', 'negative']
                and (not is_ignore_pos and lexicon[word]['pos1'] == 'anypos'
                    or pos_values[lexicon[word]['pos1']] == word_token.pos_)
        ):
            polarities[lexicon[word]['priorpolarity']][word] = lexicon[word]

        if (
            is_use_stem and stemmed_word in lexicon
                and lexicon[stemmed_word]['priorpolarity'] in ['positive', 'negative']
                and (is_ignore_pos or lexicon[stemmed_word]['stemmed1'] == 'y')
        ):
            word = stemmed_word
            polarities[lexicon[word]['priorpolarity']][word] = lexicon[word]

    return 0 if len(polarities['positive']) > len(polarities['negative']) else 1
```

We use this method on all the texts in corpus 2 to get the mean accuracy of our classifier.

```
# corpus.py

def get_baseline_sentiment_metrics(self, sentiment_lexicon: SentimentLexicon) -> float:
    ...

    # Assuming first text contains positive and second the negative samples
    labels = [1] * len(self.texts[0]) + [0] * len(self.texts[1])
    # Classify the sentiment of each line in the texts using the given sentiment lexicon
    baseline_predicitons = [classify_sentiment(line, sentiment_lexicon, tagger, stemmer)
        for line in chain.from_iterable(self.texts)]

    # Get the accuracy of the sentiment classification
    return cast(float, accuracy_score(labels, baseline_predicitons))
```

The output is the following:

```
Baseline accuracy: 0.35921965860063776
```

This result looks really bad, it is worse than guessing the polarity.

Bag-of-words sentiment classifier

For this task, I made a simple neural network to classify the sentiment of a text. I used the PyTorch machine learning library to implement the model.

```
# classifier.py

class BagOfWords(nn.Module):
    """A PyTorch neural network class using a simple bag-of-words model"""
    __doc__ += nn.Module.__doc__  # type: ignore

    def __init__(self, vocabulary_size: int, embedding_size: int):
        super().__init__()

        # As simple model with 2 linear layers
        self.embedding = nn.Embedding(vocabulary_size, embedding_size)
        self.linear1 = nn.Linear(embedding_size, 128)
        self.activation_function1 = nn.ReLU()
        self.linear2 = nn.Linear(128, 1)

    def forward(self, *input: Tensor, **kwargs: Any):
        # Sum the tensor on the first dimension to feed into linear layer
        embedding = cast(Tensor, sum(self.embedding(*input))).view(1, -1)

        output = self.linear1(embedding)
        output = self.activation_function1(output)

        # Remove unnecessary dimension at index 0
        return self.linear2(output.squeeze(0))
```

For this model, I wrote a wrapper class and implemented training, evaluation, prediction and other functionalities on that. To train the classifier, I also created a script called `train.py`. This script first loads the review corpora and tokenizes it.

```
# train.py

# Load polarity corpora
positive_review_corpus = Corpus(
    'Positive Movie Reviews',
    'data/rt-polaritydata/rt-polarity.pos',
)
negative_review_corpus = Corpus(
    'Negative Movie Reviews',
    ['data/rt-polaritydata/rt-polarity.neg'],
    tagger=positive_review_corpus.get_tagger(),
    stemmer=positive_review_corpus.get_stemmer(),
)

# Tokenize corpora to get the words in them
print('')
positive_texts = positive_review_corpus.get_tagged_texts()
negative_texts = negative_review_corpus.get_tagged_texts()
```

Then, I need to get the vocabulary of the texts to create a new classifier instance. The vocabulary size is used for the input layer size of the model and the indices of the words in it are used to encode the text for the model to predict on (e.g. if dictionary has 'something', 'happened', 'just', the text, "something just happened" is encoded as [0, 2, 1]).

To get the vocabulary of a text, we use `get_vocabulary` method of Corpus class.

```
# corpus.py

def get_vocabulary(self) -> Set[str]:
    tagged_texts = self.get_tagged_texts()
    return set(word.text for word in chain.from_iterable(chain.from_iterable(tagged_texts)))
```

```
# train.py

# Generate the vocabulary of all texts and create a classifier from it
print('Generating vocabulary...')
vocabulary = positive_review_corpus.get_vocabulary().union(
    negative_review_corpus.get_vocabulary()
)
classifier = Classifier(vocabulary, 100)
```

Once we have initiated the classifier, we need to process the texts so that the model can take it as input. For this, we use `preprocess_data` method of the Classifier class.

```
# classifier.py

def preprocess_data(
    self, texts: List[Any], text_labels: List[int],
) -> Tuple[List[Tensor], List[Tensor]]:
    data: List[Tensor] = []
    labels: List[Tensor] = []

    for index, text in enumerate(texts):
        # Create tensors from the word list and the label
        data.append(self.__create_words_vector([word.text for word in text]))
        labels.append(torch.tensor([text_labels[index]], dtype=torch.float, device=self.device)

    return data, labels
```

This method is then used to preprocess the review texts. We also shuffle the result, so that positive and negative samples won't stay together.

```
# Preprocess texts to be in the correct input format for the classifier model
# (i.e. create input data and label tensors)
print('\nPreprocessing data...')
processed_data, processed_labels = classifier.preprocess_data(
    list(chain.from_iterable(positive_texts + negative_texts)),
    [0] * len(positive_texts[0]) + [1] * len(negative_texts[0]),
)

# Shuffle texts, so that the positive and negative samples are not together
shuffled = list(zip(processed_data, processed_labels))
random.shuffle(shuffled)
data, labels = cast(
    Tuple[List[Tensor], List[Tensor]],
    zip(*shuffled),
)
```

Then we use `sklearn`'s `KFold` class to k-fold validate the model.

```
# Create k-fold splits
fold_count = 5
kfold = KFold(fold_count, shuffle=True)
cumulative_accuracy = 0.0
best_validation_loss = float('inf')

print('\nTraining classifier...')
for fold, (training_indices, validation_indices) in enumerate(cast(
    List[Tuple[List[int], List[int]]],
    kfold.split(data, labels),
)):
    ...
```

In each fold we get the training and validation data from the given indices and train the classifier for 5 epochs (usually it converges in about 4-5).

```
    # For each fold, get the training and validation data
    training_data = [data[index] for index in training_indices]
    training_labels = [labels[index] for index in training_indices]
    validation_data = [data[index] for index in validation_indices]
    validation_labels = [labels[index] for index in validation_indices]

    # Create a new classifier (so it won't be evaluated on samples that it's already trained or
    classifier = Classifier(vocabulary, 100)
    optimizer = torch.optim.Adam(classifier.model.parameters())

    # Train for 5 epochs
    for epoch in range(5):
        ...
```

For each epoch, we first train the model on the training data, then evaluate it on the validation data and get the loss and accuracy of both.

```
        start_time = time.time()

        # Train it on all training texts
        training_loss, train_accuracy = classifier.train(
            training_data,
            training_labels,
            optimizer
        )

        # Validate training on validation texts
        validation_loss, validation_accuracy = classifier.evaluate(
            validation_data,
            validation_labels
        )
```

Finally, calculate the time it took for an epoch to pass and if the current validation loss is better than the best one, update it. Then print all statistics for the current epoch

```
        epoch_mins = time.time() - start_time
        epoch_secs = int(epoch_mins - (int(epoch_mins / 60) * 60))

        # If the validation loss is less than the best, update it and save the model
        if validation_loss < best_validation_loss:
            best_validation_loss = validation_loss
            torch.save(classifier.model.state_dict(), 'sentiment_bow.pt')

        # Print the statistics for the current epoch
        print(f'Epoch: {epoch + 1:02} | Epoch Time: {epoch_mins}m {epoch_secs}s')
        print(f'\tTrain Loss: {training_loss:.3f} | Train Acc: {train_accuracy * 100:.2f}%')
        print(f'\t Val. Loss: {validation_loss:.3f}'
            + f' |  Val. Acc: {validation_accuracy * 100:.2f}%\n')
```

We also track the mean accuracy at the end of each fold, which we print when all the folds are finished.

```
        # Add the accuracy to the cumulative accuracy at the end of each fold
        if epoch == 4:
            cumulative_accuracy += validation_accuracy

    print(f'Average accuracy across folds: {cumulative_accuracy / 5}')
```

At the end, this gives the following output:

```
Average accuracy across folds: 0.7295062226831291
```

This is far better than the baseline accuracy, it's in itself is a good result as it means that almost 3/4 of the times it get the polarity of the text right.

We can also add additional features to the input of the model, for example instead of using the words' indices, we could create a different category for the negative and positive words in the lexicon.

For this, we need to add two more symbols to the vocabulary

```
# classifier.py

def __init__(self, vocabulary: Set[str], embedding_size: int):
    self.vocabulary = vocabulary
    self.vocabulary.add('<pos>', '<neg>')
```

and change the word vector creator to use this when it's in the lexicon.

```
word_indices = [lexicon[word] if word in lexicon else self.word_to_index[word] for word in word
return torch.tensor(word_indices, dtype=torch.long, device=self.device)
```