COMP34112, Natural Language Systems, Coursework1

Andras Geiszl

Please refer to the README for setup instructions.

1) Part of speech tagging

Here, I am using Spacy to POS-tag the corpus texts. The program loads the model data from a pip mirror package:

```
# main.py

TAGGER = en_core_web_sm.load()
```

Then proceeds to tag all the texts from different files in `calculate_vb_nn_probabilities` method..

```
# corpus.py

tagged_texts = [tagger(' '.join(text)) for text in self.texts]
```

Then, using these tags, we can calculate the word likelihood and transition probabilities for disambiguation as follows.

Word Likelihood

To calculate word likelyhood for VB, we first get all the words tagged as VB (`vb_tokens`) and from that count the number of words, which are "race" (`vb_race_count`), then divide the latter by the former.

```
# corpus.py

vb_word_likelihood = vb_race_count / len(vb_tokens)
```

We can do the same with words tagged as NN to get NN word likelihood.

```
nn_word_likelihood = nn_race_count / len(nn_tokens)
```

Word Transition Probabilities

To get word transition probabilities for VB, we need to count the number of times VB follows "DT" and when "IN" follows VB (`dt_vb_count` and `vb_in_count` respectively) and also get the number of words tagged as "DT" (`dt_count`). From that, we can calculate the two transition probabilities.

```
# corpus.py

dt_vb_probability = dt_vb_count / dt_count
vb_in_probability = vb_in_count / len(vb_tokens)
```

Word Disambiguation

We can use these two type of probabilities to disambiguate between the different meanings of "race" in the given context. For this, we just need to multiply all the obtained probabilities and choose the larger one to get the most likely tag for "race".

```
# main.py

disambiguation_probability = word_likelyhood * to_transition_likelihood \
    * from_transition_likelihood
```

In our case, from `Corpus A`, all the probabilities for NN came up bigger than those for VB and we got `0.0` and `0.00031925035649292907` for the overall probability for VB and NN respectively. This means, that NN tag is more likely for "race" considering the current corpus. In the sample sentence, this is the correct disambiguation outcome, so the process was successful.

2) Distributional Semantics

In the second part, we use co-occurrance patterns of words to separate them into several clusters based on their similarity of semantics. For this, we first need to build a 2D co-occurrence array (in `create_clusters` function).

```
# utilities.py

# Initiate a 2D numpy array for the word x word matrix filled with zeroes
word_x_word_matrix = numpy.zeros((len(target_words), len(vocabulary)), numpy.int32)

# For all words in all lines
for line in lines:
    for index, word in enumerate(line):
        # Get context of current word
        context = get_word_context(index, line, context_size)

        # If the word is among the target words
        if word in target_words:
            context_indexes = [vocabulary_word_to_index[context_word]
                for context_word in context]

            # increment all co-occurrence counts for the context words
            word_x_word_matrix[target_word_to_index[word], context_indexes]
```

This will count the contexts for all target words. To get the context for a specific word in the text, we use `get_word_context`.

```
# Get index of first element in context
start_index = word_index - context_size // 2
start_index = start_index if start_index >= 0 else 0

# Get index of last element in context
end_index = word_index + context_size - (word_index - start_index)
end_index = end_index if end_index < len(words) else len(words) - 1

# Calculate current context size
context = [words[index] for index in range(start_index, end_index + 1) if index != word_index]
current_size = len(context)

# If we are at the end of the list and the size of the context is less than what's given
if current_size <= context_size:
    # Add more elements at the start if possible
    start_index = start_index - (context_size - current_size)
    start_index = start_index if start_index >= 0 else 0

# Get items from start to end index not including the target word itself
return [words[index] for index in range(start_index, end_index + 1) if index != word_index]
```

After we built the co-occurrance matrix, we can use it to cluster the words. I use SKLearn's `KMeans` function here, which needs to be initiated first with the number of clusters it needs to create, then call `fit_predict` to assign each target word in the matrix to a cluster.

```
return KMeans(cluster_count).fit_predict(word_x_word_matrix)
```

Clustering Evaluation

To evaluate the accuracy of the created clusters, we use a pseudoword disambiguation approach. This means that we substitute half of the words in the text with their reverses and count how many of them and their reversed counterparts are assigned to the same cluster. First, we reverse all words with a 50% probability.

```
# corpus.py

for index, line in enumerate(lines):
    lines[index] = list(map(lambda word: word[::-1] if random() > 0.5 else word, line))
```

Then, we add the reversed words to the target words.

```
test_target_words = target_words + [word[::-1] for word in target_words]
```

Then, we use the method described above to generate the clusters.

```
clusters = create_clusters(test_target_words, cluster_count, context_size, lines)
```

For these clusters, we count how many times a word got into the same cluster as its reverse.

```
correct_count = sum(1 for index, cluster in enumerate(clusters[:len(clusters) // 2])
    if cluster == clusters[word_to_index[test_target_words[index][::-1]]])
```

From this, we can calculate an average accuracy and show it to the user.

```
print(f'Correct pairs: {correct_count}/{cluster_count}'
    f', average accuracy: {correct_count / cluster_count * 100}%')
```

Clustering Analysis

We can change the parameters of our clustering and analyze its effect on the accuracy of the clustering.

The first thing that we can notice is that the corpus has a huge effect on the clustering accuracy. For corpus B, it barely goes over 60%, but on corpus C, it has around 80%. For the two corpora together, it sometimes goes beyond 90%.

We can also use the stems of the words as context (instead of the full word). This however seems to lower the accuracy of the model. This hints that the forms of words are also important and not just the stem itself.

Another thing we can change is the context window size. This seems to have an optimal value, as at size 4 it tends to be better than both size 2 and size 10 (although results vary). The optimal windows size should be between these two values.

We could also change the types of features the clustering is using, for example instead of the co-occurrance count, we could create an embedding for the word by training a neural network and use that embedding to cluster the words. In this case, we could also adjust the dimension of the embedding (size of feature vector) and see if that has any effect on the clustering.