Project 1 Language Models

Frederick Callaway, Kang-Li Cheng

ABSTRACT

We are given lots of ebooks in .txt form from several different genres. The goal is to implement language models that will be able to predict what genre a book is from after training our language models on test sets across different genres. The process involves pre-processing the text, random sentence generation from bigrams, smoothing, calculating the perplexity, and finally classifying the genre. This report details each step and discusses the success of our language models.

1 FIRST STEPS-PREPROCESSING THE CORPUS

The books from Project Gutenberg were in .txt format and included substantial amounts of textual information about Project Gutenberg unrelated to the original work. Fortunately such text was easy to remove since they are either at the very beginning or at the very end. We used a simple search to identify when the book actually started and when it actually ended. Then we used nltk's **sent_detector** to tokenize the text by sentences and then added **SENTENCE_BOUNDARY's** between each sentence.

2 RANDOM SENTENCE GENERATION

To generate a random sentence from our ngram model, call **generate_sentence()**. The function calls **predict_next** which returns a token from the distribution of tokens that follows for the selected token. The function continues to append tokens in this way until either a "SENTENCE_BOUNDARY" is returned or until the maximum sentence length of 30 is reached.

The bigram class creates a coocurrence matrix that stores the correct probabilities from the **CounterMatrix**. Matrices were the clear choice of data structures since we chose python as our programming language. The **predict_next** function takes the current token and predicts/returns the token that is randomly chosen from the distribution suggested by the current token. We also added a feature to not generate sentences with `UNKNOWN_TOKEN` so the sentences are more readable. The code snippet below shows the implementation details.

```
def generate_sentence(self, initial="") -> str:
```

```
"""Returns a randomly generated sentence.

Optionally, the beginning of the sentence is given."""
words = initial.split()
if not words:
    words.append(self.predict_next('SENTENCE_BOUNDARY'))
for i in range(30):  # 30 is max sentence length
    next_word = self.predict_next(words[-1])

    # avoid generating sentences with UNKNOWN_TOKEN
    while next_word == 'UNKNOWN_TOKEN':
        next_word = self.predict_next(words[-1])

    if next_word == 'SENTENCE_BOUNDARY':
        break
    else:
        words.append(next_word)
    if i == 29:
        words.append('...')

return ' '.join(words) + '\n'
```

2.1 Examples

The following are some examples of output from the random sentence generation function. Note that we can specific the beginning of the sentence to be a certain string.

print(bm.generate_sentence('A'))
"A light in the little Edward in the stairs Sonia , at this is probable that every possible."

"Clovis resolved to the greatest ages of sixty without result would be struck the Nile , devoured by speaking , where the agrarian law were made himself had a collective : ..."

"At this the translators render itself , p."

"Enviously, she cried out their fortune ; [1091] The Carthaginians were the affairs of the governing and bitter period , Caesar_ , and Antemnae , did a tender his son Edward . ..."

"She paints Roman colonies – Festus , which , _Letters to contest if one hand was led his rival ."

"Razumihin stood sullenly and looked like to the sake , provided , chuckling at him ; then I often I shall walk of it could have understood."

"I had been asleep."

3 SMOOTHING AND UNKNOWN WORDS

We implemented Good-Turing smoothing in our bigram model, which uses the count of bigrams that have only been seen once to estimate the frequency of zero-count bigrams.

```
def get_good_turing_mapping(self, threshold=5) -> Dict[int, float]:
        """A dictionary mapping counts to good_turing smoothed counts."""
        total_count_counts = sum(self.count_counts.values(), Counter())
        def good_turing(c):
            return (c+1) * (total_count_counts[c+1]) / total_count_counts.get(c, 1)
        gtm = {c: good_turing(c) for c in range(threshold)}
        return {k: v for k, v in gtm.items() if v > 0}  # can't have 0 counts
```

3.1 Unknown Words

To handle unknown words in the testing corpus, we wrote a method that keeps a count of words that have not been seen before and adds them to a Python set. After the training corpus has been processed, the first occurrence of each word is labeled `UNKNOWN_TOKEN` and then the probability/co-occurrence matrix and distribution is calculated. This approach essentially classifies all unknown words as rare words.

When applying the trained bigram to the testing sets, encountering an unknown word will result in using the distribution of unknown words as calculated from the testing set. We use a **try/except** block when processing the testing corpus.

```
# first occurcence of a word is replaced with UNKNOWN_TOKEN
seen_words = set()
for i in range(len(self.tokens)):
        word = self.tokens[i]
        if word not in seen_words and word is not 'SENTENCE_BOUNDARY':
            self.tokens[i] = 'UNKNOWN_TOKEN'
        seen_words.add(word)
```

4 PERPLEXITY

To calculate perplexity we wrote a function `def perplexity(self, tokens):` that takes a test corpus (tokenized). It divides the corpus into groups of n words. Then for each group it finds the probability of the last word of the group, given the previous words. It then adds the numbers of surprisals to a running total. After going over the entire corpus the function then multiplies `pp` by $\frac{1}{N}$, the length of the corpus. Finally the perplexity is obtained by raising e to the power `pp`, e.g. `pp = math.exp(pp)`.

We use log rules and Python's BigDecimal to avoid overflow during the perplexity computation. Perplexity is defined as:

$$PP = \left(\prod_{i=0}^{N} \frac{1}{P(w_i|w_{i-1}, ..., w_{i-n+1})} \right)^{1/N}.$$

But by using log rules we can rewrite it as:

$$log(PP) = \log \left(\prod_{i=0}^{N} \frac{1}{P(w_i|w_{i-1}, ..., w_{i-n+1})} \right)^{1/N} = \frac{1}{N} \sum_{i=0}^{N} \log \left(\frac{1}{P(w_i|w_{i-1}, ..., w_{i-n+1})} \right).$$

We can then simply find PP by doing $e^{log(PP)}$. In our perplexity function we use this approach, note that all logs in these calculations are natural logs (base e).

```
def perplexity(self, tokens):
        """Average surprisal or something."""
        first_surprisal = self.surprisal('SENTENCE_BOUNDARY', tokens[0])
        total_surprisal = first_surprisal + sum(self.surprisal(tokens[i], tokens[i+1])
                                                for i in range(len(tokens) - 1))

        return math.exp(total_surprisal / (len(tokens)))
```

5 GENRE CLASSIFICATION

To determine how successful our bigram model is at classifying books by genre, we assign a rule based on perplexity measures. We expect that for a bigram trained on a certain genre X, then tested against all of the other genres, the perplexity measure of the bigram model on the testing corpus of genre X will have the lowest perplexity versus the perplexity from the testing corpuses of the other genres.

An example of functions called when the bigram is trained on history then tested against other genres.

```
def test_genre_classifier():
    print('\nTesting genre classifier')
    genres = ['children', 'crime', 'history']
    train_dirs = {g: genre_directory(g) for g in genres}
    clf = GenreClassifier(train_dirs)

    for genre in genres:
        directory = genre_directory(genre, test=True)
        for file in get_text_files(directory):
            print('%s book classified as %s' % (genre, clf.classify(file)))
```

We then wrote an additional function that would classify based on this rule of lowest perplexity and return the predicted genre given a test corpus.The following table show the perplexity values for each model against the three different test genres.

		Train	Children's	Crime	History
Perplexity	Test	Children's	556.2996	564.3006	1887.7872
		Crimes	440.0556	316.5115	1344.1917
		History	1032.2186	1129.0515	1299.6966

Note that using lowest perplexity gives the correct genre two-thirds of the time. Our model will incorrectly identify "Crimes" as "Children's" since the model trained on "Children's" has the lowest perplexity when applied to "Crimes". In all of the other cases picking the test genre with the lowest perplexity results in correct classification, i.e. the model trained on "genre X" has the lowest perplexity when applied to test corpus of "genre X". We think that the results could be improved by training on more corpuses. For example, the children's books are all from a series called "Junior Classics" which may skew the results. The history corpuses are all from history of Western civilization which may explain why the perplexity values were on average quite high when the bigram model was trained on that test set. It would be intuitive to simply train on more texts and compare the results.

When we normalized each dataset by its mean perplexity we were able to classify with one-hundred percent accuracy on a held-out validation corpus. So the bigram model and the perplexity rule seems to be an acceptable method of classifying text by genre if the datasets are normalized first.

6 EXTENSION

For the extension, we chose to employ our n-gram model in the service of an NLP application. We created a Sublime Text plugin, BigramsCompleteMe[1], which sorts autocompletion results based on the probability given the previously typed word. (See the linked Github page for a demonstration and installation instructions). The plugin is designed to be used for editing text files, including plain text, LaTeX, and markdown.

The utility of the tool comes from improved `insert_best_completion` performance, with the goal of increasing typing speed by saving the writer from typing long words and phrases repeatedly. It is not clear through simple tests exactly how Sublime Text's default sorting algorithm works, but it is some combination of frequency and recency. This is highly effective for editing source code which has few unique words; however it leaves much to be desired for natural language text completion.

At present, the plugin only uses the `Distribution` class because at each completion, only the bigrams beginning with one word are relevant. It is difficult to know when a user has deleted a word, thus retraining upon every completion is the only straightforward and reliable option. This causes noticable delays when training the full model on files larger than 5000 words. However, future versions of the plugin could use pre-trained `BigramModel` objects, which could come packaged with the plugin or be user-generated with a Sublime Text command.

7 DIVISION OF LABOR

Fred Callaway wrote the bigram class and random sentence generator function. He was in charge of the choice of language and the main driver behind our coding approach. He also implemented smoothing and perplexity and the extension that sort autocompletion results based on the probability of the previously typed word, in Sublime Text.

[1] https://github.com/fredcallaway/BigramsCompleteMe

Kang-Li Cheng implemented pre-processing and wrote a method to handle unknown words. He did the testing and experiment with classification and wrote most of this report.