CS6320: Assignment 1

Group 18 Kevin Tak Hay Cheung (kxc220019) Humza Salman (mhs180007)

Introduction

The project aims to classify if a hotel review is truthful or deceptive using the n-gram language model. We are given truthful and deceptive hotel reviews to train and validate the models. We will build a truthful n-gram model trained on truthful data and a deceptive n-gram model trained on only deceptive reviews. Ultimately, we will pass the testing reviews to these two models and compare the perplexities. The prediction class will be associated with the model that produces lower perplexity.

The following sections will provide details of the modeling process, including data preprocessing and n-gram probability computation, perplexity calculation, and classification modeling.

Data preprocessing

Before building the models, we performed several data cleaning processes to reduce the noise in the data.

Case Folding

We converted every review to lowercase so that the tokens with the exact spelling are represented identically. For example, "Pleasant" and "pleasant" will mean the same after the conversion. Applying this normalization ensures the models will not reward or penalize the testing reviews due to different cases. Also, it reduces the number of vocabularies from the input data and makes the resulting probability less sparse.

Lemmatization

This normalization technique converts the words back to their root form (lemma). For example, the words *"is", "am", and "are"* will be converted back to *"be"*. The lemmatization algorithm will evaluate the word's context and identify the correct root. Applying this normalization ensures the same n-gram will be more representative in different scenarios and sentence structures. However, lemmatization is time-consuming and does not improve the model performance significantly. Therefore, we decided to go with a more straightforward process – stemming.

Stemming

Stemming is a rule-based process that removes the words' prefixes or suffixes to identify their root form without considering the word's context. For example, the words *"plays", "playing", "played"* will be converted to *"play"* using the suffix information. Although this method is considered inferior to lemmatization, it requires less computation time, and the implementation is more straightforward. In this project, we will use the Porter Stemmer provided by the python library "nltk".

Tokenization

Tokenization is an essential process for language models. It converts text into a list of smaller tokens. In this project, we will use the word_toknize function provided by the python library "nltk" to tokenize the reviews into words.

Stop Word Removal

Removing stop words is considered an essential step for text preprocessing. In general, certain words, such as *"those", "there", and "that"*, do not provide extra information to the models. We tried to remove these stop words from the reviews using the stop word list provided by the python library "nltk", but the model performances were worse. We believe the existence and the number of stop words may help explain the review types. For example, deceptive reviews may contain more or less stop words. Therefore, we decided not to remove any stop words from the dataset.

Unigram and Bigram Word Probability Computation

This section will discuss the computation of the unigram and bigram probabilities using the preprocessed dataset obtained from the previous section. Although a TRIE structure is considered more efficient for n-gram modeling, we decided to go with the python dictionaries due to their simplicity and familiarity.

Unigram

We must compute the count for each word (unigram) in the corpus to compute a unigram. We utilized python's dictionary to store all the counts. Our algorithm will loop through each token and check if the unigram exists in the dictionary. If it does not exist, we create a new key and set the value to 1. Otherwise, we will add 1 to the currently stored value. The details can be found in the following code snippet.

```
for word in review:
        if word not in self._ngram:
            self._ ngram [word] = 1
        else:
            self._ ngram [word] += 1
```

We imported the training corpora, and the following dictionaries show part of the frequency counts:

Truthful corpus: {'<s>': 512, 'i': 1708, 'book': 118, 'two': 128, 'room': 1131, 'four': 20, 'month': 15, …}
Deceptive corpus: {'<s>': 512, 'i': 2554, 'wa': 2057, 'here': 109, 'on': 434, 'busi': 135, 'so': 254…}

The next step is to divide each word count by the corpora's total word count to obtain the unsmoothed probabilities:

Truthful corpus: {'<s>': 0.0056, 'i': 0.0188, 'book': 0.0013, 'two': 0.0014, 'room': 0.0125…}
Deceptive corpus: {'<s>': 0.0059, 'i': 0.0297, 'wa': 0.0239, 'here': 0.0013, 'on': 0….}

Bigram

Like unigram, the algorithm will count the bigrams in the corpora. It would create a dictionary of dictionaries to store the counts and probabilities. The following tables illustrate the idea.

Truthful Bigram Count:

	i	book	two	room	four	month	in	advanc
i	0	21	0	0	0	0	0	0
book	0	0	1	0	0	0	0	0

Deceptive Bigram Count:

	i	wa	here	on	busi	so	need	to
i	0	338	0	0	0	1	18	4
wa	2	0	3	18	5	37	1	14

Each row represents a unique dictionary, the column names represent the keys, and the cell values represent the count in the dictionary. With this structure, we can easily output the unsmoothed or smoothed probabilities for debugging purposes. All dictionaries are stored in another dictionary with the row name as the key.

To convert the bigram counts to probabilities, we can divide each cell value by its row total, and the following table shows the unsmoothed probabilities of our example.

Truthful Bigram Probabilities:

	i	book	two	room	four	month	in	advanc
i	0.0000	0.0123	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
book	0.0000	0.0000	0.0085	0.0000	0.0000	0.0000	0.0000	0.0000

Deceptive Bigram Probabilities:

	i	wa	here	on	busi	so	need	to
i	0.0000	0.1323	0.0000	0.0000	0.0000	0.0004	0.0070	0.0016
wa	0.0010	0.0000	0.0015	0.0088	0.0024	0.0180	0.0005	0.0068

Smoothing

We utilized Laplace smoothing to handle the unseen n-grams whose components are seen in the training corpus. Laplace smoothing is the technique that adds one occurrence to all possible n-grams from the training corpus. The following tables show the smoothed probabilities for both corpora.

Truthful Bigram Laplace Smoothed Probabilities:

	i	book	two	room	four	month	in	advanc
i	0.0002	0.0034	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002
book	0.0002	0.0002	0.0004	0.0002	0.0002	0.0002	0.0002	0.0002

Deceptive Bigram Laplace Smoothed Probabilities:

	i	wa	here	on	busi	so	need	to
i	0.0002	0.0517	0.0002	0.0002	0.0002	0.0003	0.0029	0.0008
wa	0.0005	0.0002	0.0007	0.0031	0.0010	0.0063	0.0003	0.0025

One problem with Laplace smoothing is that the technique may transfer a considerable portion of probability mass to the unseen n-grams. Also, the smoothed probabilities may not correctly capture the probability distribution. For example, the smoothed bigram likelihood of “book two” in the truthful model is only two times the other possible n-gram pairs, which may be highly unlikely.

To overcome the disadvantage of Laplace smoothing, we also implemented add-K smoothing to handle the unseen n-grams. It is very similar to Laplace smoothing, but it adds K occurrences to each n-gram pair instead of 1. We can adjust K so that the smoothed probability can capture the actual likelihood. To better model the truthful and deceptive reviews, we set K = 1.06 for the truthful bigram model and K = 1.46 for the deceptive bigram model. The rationale will be explained in the upcoming sections. The following tables show the add-K smoothed probabilities.

Truthful Bigram Add-K Smoothed Probabilities:

	i	book	two	room	four	month	in	advanc
i	0.0002	0.0033	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002
book	0.0002	0.0002	0.0004	0.0002	0.0002	0.0002	0.0002	0.0002

Deceptive Bigram Add-K Smoothed Probabilities:

	i	wa	here	on	busi	so	need	to
i	0.0002	0.0404	0.0002	0.0002	0.0002	0.0003	0.0023	0.0007
wa	0.0004	0.0002	0.0006	0.0025	0.0008	0.0049	0.0003	0.0020

Unknown Words

To account for unseen words in the corpora, we decided to replace words in the training data with <UNK>. We implemented two strategies – replacement by corpus frequency and random word selection. In the end, the algorithm will produce a list called an unknown list that stores all the words to be replaced with the unknown tag when we process the training data.

Replacement by Corpus Frequency

Replacement by corpus frequency replaces the occurrences of selected words with the unknown tag if the word has a total frequency below a certain threshold in the entire corpus. This threshold is determined arbitrarily and we will experiment with thresholds of 1,2, and 3 in our model to compare the results.

Replacement by Random Word Selection

Replacement by random word selection selects words at random and replaces their occurrences with the unknown tag. We utilize the random function from the python library “random” and use a uniform distribution to select a percentage of words to replace. We will experiment with 0.5%, 0.75%, and 1% for the percentage selection and compare the outcome.

Perplexity

After handling the unknown words and performing the smoothing, the two models can now estimate the perplexities for the reviews. As an example, this section will go over the calculation of the bigram perplexities using the validation set with Laplace smoothing and replacement by corpus frequency (threshold = 1).

```
def likelihood(self, txt):
    words = [self._ps.stem(word) for word in word_tokenize(txt.lower()) ]
    words = [ word if (word in self._unigram and word not in self._unk_list) else "<UNK>" for word in words ]
    tokenized_txt = ['<s>'] * max(1, (self._n - 1))  + words + ['</s>']
    prob = 0
    for i in range(len(tokenized_txt) - self._n + 1):
        prior_ngrams = " ".join(tokenized_txt[i:(i + self._n-1)])
        next_word = tokenized_txt[i + self._n - 1]

        if prior_ngrams in self._ngram_prob_table_smoothed and next_word in self._ngram_prob_table_smoothed[prior_ngrams]:
            prob += -math.log(self._ngram_prob_table_smoothed[prior_ngrams][next_word])
        else:
            prob += -math.log(self._k / (self._k * (self._V - 1) ))

    return math.exp(prob/len(tokenized_txt))
```

We created a function *likelihood* to estimate the perplexity for any input *txt*. It normalizes the inputs (with lowercasing and stemming) to ensure they are consistent with the training data. Next, it replaces the unknown word (using the unknown list or the actual unseen word) with the <UNK> tag. Then it pads the input with the start and stop signals, and finally loops through each n-gram to compute the perplexity according to the formula given.

The table below shows the perplexities of the first five reviews from truthful and deceptive validation sets calculated by both truthful and deceptive models. The prediction is favored to the model that produces the lower perplexity.

Review Type	Truthful Model	Deceptive Model	Prediction
Truthful	260.00	253.91	Deceptive
Truthful	296.79	339.06	Truthful
Truthful	448.49	362.15	Deceptive
Truthful	337.12	352.37	Truthful
Truthful	260.46	261.98	Truthful
Deceptive	247.08	176.13	Deceptive
Deceptive	186.82	151.20	Deceptive
Deceptive	310.82	256.92	Deceptive
Deceptive	358.44	248.97	Deceptive
Deceptive	448.92	397.02	Deceptive

As you may notice, the truthful model does not perform very well. After passing truthful reviews to both models, the truthful bigram model sometimes returns a higher perplexity than the deceptive model. Therefore, we may need to tune the parameters for the truthful n-gram models to improve their performances.

Opinion Spam Classification with Language Models Methodology

This section will create a classification model using the n-gram models. The classification depends on the lower perplexity score returned by the n-gram models. We will train the truthful and deceptive models using the two sets of training corpora, tune the parameters using the validation set if needed and perform a final evaluation using the testing data.

To start with, we created several baseline models without any parameter tuning. The following table shows the unigram and bigram models with the best performances on the validation sets.

Model Type	Smoothing K in Truthful Model	Smoothing K in Deceptive Model	Unknown Words	Validation Overall Accuracy	Validation Accuracy Rate (Truthful Set)	Validation Accuracy Rate (Deceptive Set)
Unigram	0.1	0.1	Random Word Selection - 0.75%	65%	69%	61%
Unigram	0.1	0.1	Random Word Selection - 1%	66%	49%	84%
Unigram	0.1	0.3	Random Word Selection - 0.75%	65%	50%	80%
Unigram	1	1.3	Random Word Selection - 0.75%	68%	60%	76%
Unigram	1	1.3	Random Word Selection - 1%	66%	54%	77%
Unigram	1	1.5	Random Word Selection - 0.5%	65%	44%	86%
Bigram	0.1	0.1	Random Word Selection - 0.5%	76%	54%	97%
Bigram	0.1	0.1	Random Word Selection - 0.75%	75%	54%	96%
Bigram	0.1	0.1	Random Word Selection - 1%	76%	57%	96%
Bigram	0.1	0.1	Replacement by corpus frequency - threshold = 1	75%	57%	93%
Bigram	0.1	0.1	Replacement by corpus frequency - threshold = 2	79%	64%	94%
Bigram	0.1	0.1	Replacement by corpus frequency - threshold = 3	79%	66%	93%
Bigram	1	1	Replacement by corpus frequency - threshold = 2	66%	36%	97%
Bigram	1	1.3	Random Word Selection - 0.5%	81%	67%	96%
Bigram	1	1.3	Random Word Selection - 0.75%	86%	77%	94%
Bigram	1	1.3	Random Word Selection - 1%	83%	71%	94%
Bigram	1	1.3	Replacement by corpus frequency - threshold = 1	77%	66%	89%
Bigram	1	1.3	Replacement by corpus frequency - threshold = 2	79%	73%	86%
Bigram	1	1.3	Replacement by corpus frequency - threshold = 3	77%	70%	84%

These baseline models provide several insights. First, the unigram models generally perform worse than the bigram models. It makes sense because bigram pairs convey more information than unigram alone. Second, with the same set of Ks, the model using random word selection performs better than the models with corpus frequency replacement. The reason is that the replacement by corpus frequency may drop the rare but significant key n-grams, which may significantly impact the model performance.

We also noticed that the classification model performs better when the two K values are different – to be exact when the deceptive K value is larger than the truthful K value. Due to the fundamental differences between the truthful and deceptive corpora (such as vocabulary size and word choice), the resulting n-gram models may be biased - such as one model returning a higher perplexity score on average. That would explain why the truthful model would return a higher perplexity score in the truthful review in the previous section. To ensure that the models only produce lower or higher perplexities due to the truthful or deceptive features of the review, the smoothing factor can be used as a normalizing factor to offset the fundamental differences in the dataset, and the differences in the smoothing Ks can bring the two models to level ground.

To obtain a better classification model using the bigram models, we created a tool that can optimize the overall accuracy rate by adjusting the two k parameters used in the truthful and deceptive bigram models using the validation set. Since the optimization tool is not the focus of this project, we will not go further in this report, but more details can be found in the attached code. It may be relevant to mention that utilizing the optimization tool increases the run-time of the code. The following table shows the optimization results with the best performing bigram models and their corresponding validation accuracies.

Model Type	Truthful K	Deceptive K	Unknown Words	Validation Overall Accuracy	Validation Accuracy Rate (Truthful Set)	Validation Accuracy Rate (Deceptive Set)
Bigram	1.06	1.46	Random Word Selection - 0.5%	84%	70%	97%
Bigram	1.06	1.46	Random Word Selection - 0.75%	89%	93%	86%
Bigram	1.06	1.46	Random Word Selection - 1%	83%	67%	99%

The best performing bigram model is the one with truthful K = 1.06 and deceptive K = 1.46 using randomly chosen 0.75% of the vocabulary as unknown word replacement. However, since the unknown word handling is a random process, there is no guarantee that it would achieve such a high accuracy rate in the testing set. One of the reasons is that the handling method may accidentally pick some very representative words for a particular class and replace them with <UNK>, which may significantly impact the model performance. To overcome the issue, we decided to ensemble the classification models by creating 50 sets of truthful and deceptive n-gram models. When we pass a review into the model, these models will vote and pick the class with the most votes as the final prediction. The following table shows the validation and testing accuracies.

Model Type	Truthful K	Deceptive K	Unknown Words	Validation Overall Accuracy	Validation Accuracy Rate (Truthful Set)	Validation Accuracy Rate (Deceptive Set)	Test Set Accuracy
Bigram	1.06	1.46	0.75% Random Unknown Word	87%	80%	94%	83%

The final evaluation with the test set produces an accuracy rate of 83%. We output the misclassified reviews, pulled the voting results from our ensemble model, and attached the details in the table below.

Review	Test_Labels	Truthful_Votes	Deceptive_Votes
1	1	50	0
11	1	40	10
20	1	50	0
38	1	48	2
49	1	47	3
55	1	49	1
60	0	8	42
67	0	11	39
73	0	23	27
79	0	1	49
90	0	1	49
92	0	2	48
93	0	1	49
96	0	13	37
100	0	20	30
101	0	8	42
105	0	23	27
106	0	12	38
108	0	16	34
112	0	6	44

The table shows that most misclassified reviews received overwhelming votes from the other models. For example, after passing the first review, which is supposed to be a deceptive review, to all 50 sets of bigram models, the truthful bigram models returned a lower perplexity all 50 times. Since the models remove n-grams randomly each time, it is doubtful that the unknown word selection accidentally removed some representative n-grams. A possible explanation is that there are some features other than n-grams that are not considered in the classification models, such as extravagant tone, misspelling and grammatical mistakes.

Programming Library Usage

- ***ntlk*** is used to perform text preprocessing, including work_tokenize, porter_stemmer
- ***copy*** is used to create deep copies of the dictionaries storing the language model
- ***math*** is used the compute the perplexity
- ***random*** is used to perform the unknown word selection
- ***scipy*** is used to identify the optimal k values (using optimize, basinhopping)
- ***numpy*** & ***panda*** are used for general data processing

Contributions of Each Member

Kevin: Data Preprocessing (80%), N-gram probability calculation (50%), Smoothing & Unknown word handling (30%), Perplexity (50%), Classification Modeling (40%), Report (65%)

Humza: Data Preprocessing (20%), N-gram probability calculation (50%), Smoothing & Unknown word handling (70%), Perplexity (50%), Classification Modeling (60%), Report (35%)

Feedback of Project

The project difficulty is at the right level and appropriate time. Although there was confusion from the start, we could research the topics and resolve the issues on our own. However, there were several suggestions that we would like to bring up:

- We are not sure about the format of the code submission – should we have the entire code ready from the data preprocessing to the prediction? Or do we need to create a class of functions to achieve the language modeling?
- We are unclear about what the grader expects in each section – we did all the requirements, but we do not know what to show in this report. Should we provide evidence that we did all these smoothings and unknown word handling in the actual modeling process, or can we just describe our understanding of each section and claim we accomplished the task in the code?
- A general expectation of the result would be appreciated – we achieved an accuracy rate of 83%, but we are not sure if it is a good or bad sign. It may be unreasonably low or high, which usually signals that we mess up something in the process.
- Overall, it is a fun project; we learned a lot through the process. Thank you!