1 Summary

In the first part of this project, we used n-grams to model corpora of speeches from Presidents Obama and Trump. We evaluated these models extrinsically by applying them to a classification task with unlabeled sentences from Obama and Trump speeches. We also evaluated the models intrinsically by calculating the perplexity of the development set. In the second part of the project, we used different sets of pre-trained word embeddings for this same classification task, as well as evaluating the embeddings on an analogy task. We chose to do this project using Python because of the optimized matrix operations available in NumPy.

2 Unsmoothed N-Grams

To train our unsmoothed n-gram models, we read the corpora from the given training data files, preprocessed it to tokenize the text, and then built up matrices tracking the unigram and bigram counts and probabilities.

2.1 Preprocessing

Although the corpora were pre-tokenized, but we decided to further preprocess this text to make it as consistent and semantically logical as possible. The rationale behind our decisions is described in detail below.

2.1.1 Start and End Tokens <s> and </s>

We decided to mark the start and end of each sentence with special tokens to improve the semantics of our models. This makes it easier to know when a sentence has logically ended, without relying on the period alone. This is helpful because some periods do not mark the end of sentences, as in the word "Jr.". It also provides a token to use as a seed for generating random sequences.

2.1.2 Handling of Contractions and Possessives

To handle the inconsistent parsing of contractions and possessives in the corpus (e.g. "you 've" vs. "you ' ve" vs. "you' ve"), we will remove all spaces on either side of all apostrophes, so that contractions will be tokenized as single words (e.g. "you've"). This will improve the consistency of our n-gram models because different parsings of the same contraction should count as the same word. We acknowledge that there are some instances of separated contractions where the apostrophe is still surrounded by letters (e.g. "ca n't"), but unfortunately there is not much we can do about this because there is no way to know programmatically that any two consecutive words are a contraction that should be combined. These instances will remain as multiple tokens (e.g. "ca" and "n't").

2.1.3 Handling of Whitespace Surrounding Periods

There are two kinds of periods in the corpora: those that mark the end of their sentence and those that do not. By and large, we feel it is a safe assumption that all sentence-ending periods will be surrounded by either a whitespace character or a newline character. Others that are bordered on one or both sides by a letter, another period, etc. are not intended to end sentences. During preprocessing, we will replace the newlines after periods with spaces, since newlines and spaces after periods seem logically equivalent.

2.2 Data Structures

2.2.1 Dictionaries for Word Type-ID Mappings

We used dictionaries to store mappings between word types and IDs so that we could translate between these in constant time. This is used when training n-gram models and generating random sentences. Each word type is assigned a unique ID corresponding to its index in n-gram matrices.

2.2.2 Bigram and Unigram Matrices

We stored probabilities and counts in separate matrices for unigrams and bigrams, because simply using the sum of bigram rows for unigram counts is complicated by smoothing. We chose to use matrices because of the optimized operations in NumPy, such as finding counts of each number in the matrix, which was used for Good-Turing smoothing. Although in the unsmoothed bigram model the matrix will be extremely sparse, storing all the zero values in a matrix makes smoothing much easier.

2.3 Training N-Grams

To train our unsmoothed bigram model, we iterated through the tokens of our processed text and incremented the value at position (i, j) in the bigram count matrix, where i and j are the indices of the last and current token being processed. Since the first token does not end a bigram and the last token does not begin a bigram, this accounts for $n - 1$ total bigrams, if there are n tokens in the corpus. We then divided each count by the sum of its row to find the bigram probabilities.

To train our unsmoothed unigram model, we simply incremented the value at position i in the unigram count matrix, where i is the index of the token being processed. We then divided each count by the total number of tokens to find the unigram probabilities.

3 Random Sentence Generation

Our next task was generating random sentences (i.e. sequences of tokens) from our unigram and bigram probabilities in order to gain insight into the models. Our results are below.

3.1 Unigram Examples

These examples are seeded with the start token.

3.1.1 Obama

1. <s> modernize some densely kennedy armed will to arise in plane pay to . of , on scratched even , </s>
2. <s> nuclear </s>
3. <s> one the individuals sick , tonight own international community deficits with not breathe . than are do were of to strongly leak with signed them price <s> joe american you a if costs from safe and any on reach makes bring and . and . end politics a broker 1999. millions and </s>
4. <s> are </s>

3.1.2 Trump

1. <s> do </s>
2. <s> magazine </s>
3. <s> , <s> to i <s> – </s>
4. <s> </s>

3.2 Bigram Examples

In this section, the last sentence for Trump and Obama was seeded with the sequence "<s> i want". The first four are seeded only with the start token.

3.2.1 Obama

1. <s> and kept our destiny . </s>
2. <s> so has a way into the muslim and our power grids , and grenades but we should . </s>
3. <s> oh , alternatively , america is life-changing . </s>
4. <s> i want to the market until they lost loved one that we saw that offer social security demands by the young minds of support from wall . </s>

3.2.2 Trump

1. <s> so they would have a speech was taken care of which is hard . </s>
2. <s> i love that if the state building . </s>
3. <s> you're going to . </s>
4. <s> i want to be applied to put on a lot . </s>

3.3 Analysis

The bigram model is clearly the better model for creating a probability distribution that yields more coherent sentences. The unigram models for both corpora result in more 0- or 1-word sentences, because the end-of-sentence tokens are understandably one of the more prevalent word types in the corpora. When only the straight count affects the probabilities, this end token will be randomly chosen with more frequency. Also, when the unigram-generated sentences do make it longer than a few tokens, they are much more disjointed, as consecutive words appear that do not agree on tense, gender, plurality, etc. In bigram-generated sentences, this does not happen as much, because conesecutive words have to have appeared in the corpus in order to have any probability of being in a randomly generated sentence. This results in much more logical and meaningful sentences. Also, there are no periods or illogical consecutive punctuation in the middle of bigram-generated sentences, because all periods in the corpus are followed by end tokens.

All random sentences are seeded with the start token, but we also experimented using words and phrases for seeds for the bigram-generated sentences. Seeding does not affect unigram sentences because the probabilities of the next word are not dependent on the previous word. Displayed above are one sentence from each bigram model seeded with "<s> i want". Only the last token impacts the randomly generated sentence, which can be seen in the Obama example. Both models randomly choose "to" first, as it often appears after "want", but in the Obama example it appears as a prepositional phrase instead of a verb phrase, which its wider context suggests.

4 Smoothing and Unknown Words

Smoothing is necessary to avoid overfitting the model to the training data such that it performs poorly on test data. For example, without smoothing, bigrams that are unseen in the training data will have a probability of zero, even if the individual words appear on their own in the training corpus. In general, better performance is achieved when some of the probability of bigrams and unigrams with higher counts is shifted to those with lower counts.

In order to experiment with different parameters for our model, we implemented several different types of add-k smoothing, as well as Good-Turing smoothing. As shown in class, add-k smoothing simply consists of adding a constant k to all n-gram counts before calculating probabilities. This scales very easily to different values of k. Good-Turing smoothing is a bit more complicated - it involves updating counts only of less frequently seen n-grams, with the increment dependent on how many different n-grams had that count. This algorithm was designed to encode the pattern typically seen in which probabilities of rare n-grams are underestimated in the training data. We implemented the algorithm as presented in class.

Additionally, in order to calculate the probability of a test corpus, we needed to devise a strategy to handle unknown word types that did not appear at all in the training set. This is important, because punishing unknown words too much or too little will cause inaccuracies in classification. At first, we tried setting the probability of unknown n-grams to be $\frac{1}{N}$ where N is the total number of n-grams in the corpus. This proved to be too large a probability, hence not punishing unknown words enough, and it classified almost all test trials as Trump speeches because Trump's test data had more unknown words. We then tweaked it to $\frac{1}{N+V}$ where V is the size of the training vocabulary. This essentially treats unknown words the same as unseen bigrams whose individual tokens have been seen separately in the test data. This improved our accuracy significantly, but it still was not quite punishing unknown words enough, since we realized that these should be considered as less likely to appear than unseen bigrams. Therefore, we tried dividing the numerator by 2 and 10, achieving the best accuracy with 10. We also realized that, instead of using 1, the probability needed to take the smoothing constant into account for add-k smoothing. Since the probability of unseen bigrams would automatically be made $\frac{k}{N+k*v}$ by smoothing, we made the probability of unknown bigrams $\frac{0.1*k}{N+k*v}$

to make them 10% as probable as unseen bigrams. See section 6.2 for comparative results on the classification task. **NOTE: Due this methodology of handling the unknown words, our perplexity values are somewhat high. This is because it punishes unknown and unseen n-grams more than some other methodologies. However, we found this implementation to maximize our accuracy on the classification task.**

5 Perplexity

Perplexity is an intrinsic method of evaluating a language model. A lower perplexity indicates a higher probability of a test corpus, and thus a better model. We calculated perplexity as described in class. This was a very useful heuristic for us in terms of tweaking the parameters of our model. We tested our models on the development set repeatedly by making a tweak, calculating the perplexity, and finding the combination of parameters and implementation decisions that minimized the perplexity on this held-out set. This was instrumental in helping us decide which n value, smoothing types, preprocessing implementations, and methods of handling unknown words we should spend our limited Kaggle submissions on to test using the test data. The development set perplexities for different combinations of n and smoothing type, given our latest implementation of everything else, are below.

5.1 Comparative Results of Perplexity Heuristic

These are perplexity values calculated on the development sets with different parameters for our model:

n	Smoothing Type	Obama PP	Trump PP
2	Add-0.1	11388	1273
2	Add-0.3	10609	1239
2	Add-0.5	10492	1265
2	Add-1	10501	1340
2	Add-2	10642	1460
2	Good-Turing	72557	5493
1	Add-0.1	2703	663
1	Add-1	2618	656

Although this chart seemingly indicates the superior performance of unigrams to bigrams, this is somewhat deceiving - the unigram perplexities are lower only because there are only V terms instead of V^2 for bigram perplexity, not because unigram is the better model. We chose bigram because it is logically better to take some context into account when assessing probabilities. Within bigram perplexities, Add-0.3 and Add-0.5 performed the best, so that's what we focused on the most. As discussed in class, this is because with k values of 1 or 2, too much probability gets shifted, and the smoothing over-corrects. The perplexity values for Good-Turing are artificially high here because the way we handle unknown words in calculating perplexity is specific to an Add-k model. Since all Add-k models were similar and we could experiment more with those than Good-Turing, we decided not to spend a lot of time modifying our handling of unknown words to work the best for Good-Turing.

6 Speech Classification Using N-Grams

Classification of test data as either Obama or Trump speeches was the main extrinsic evaluation tool of this project. We were trying to optimize our performance on this task.

6.1 Approach

Once we implemented the calculation of perplexity, all we needed to do for each test sample was to calculate perplexity on the Obama and Trump models and classify it as whichever model gave the lower perplexity.

6.2 Comparative Results of Classification Task

As mentioned in section 4, we changed our handling of unknown words in the perplexity calculation to improve our accuracy. The results below are classification accuracy values on the test set with different n and k parameters, for our original implementation, with the probability of unknown words = $\frac{1}{N+V}$:

k	n	Accuracy
0.1	1	0.94
0.1	2	0.98
0.3	2	0.97
1	2	0.91
2	2	0.88

After changing our probability of unknown bigrams to $\frac{0.1*k}{N+k*V}$, we received the accuracy scores below:

k	n	Accuracy
0.05	2	0.98
0.1	2	0.98
0.5	2	0.975
1	2	0.95

This change in handling unknown words tangibly increased our performance on the classification task, which we expected since the new model is more logically sound, as described in section 4. We also see the trend of accuracy peaking between k values of 0.1 and 0.5. This mirrors the trend we saw in our tests against the validation set, which allowed us to narrow the scope of our test set tests to this range of optimal parameters.

7 Evaluating Word Embeddings

In the second part of the project, we evaluated different pre-trained sets of word embeddings. One was the Google dataset Word2vec [1], and the other other was the Stanford Glove dataset [2].

7.1 Dataset and Task

Our first task with the word embeddings was an analogy task where, given the first three words, the model predicts the fourth word that has the same relation to the third word as does the second to the first. For example: "boy girl brother sister". The fourth word vector $\mathbf{v}_4$ is guessed by maximizing the cosine similarity $cos(\mathbf{v}_4, \mathbf{v}_2 - \mathbf{v}_1 + \mathbf{v}_3)$. To improve the runtime, we traversed the vocabulary of the analogy test file, not the entire set of embeddings, to find the best guess for $\mathbf{v}_4$. We ran this evaluation on a set of 13,618 trials in analogy_test.txt.

7.2 Results

The results of the analogy task for each embedding set is shown below, where success represents correctly guessing the fourth word in the analogy in analogies_test.txt:

	# Correct	# Trials	Accuracy
Word2vec	11557	13618	0.849
Glove	11674	13618	0.857

Our biggest obstacle limiting our accuracy in this task was that, since there was often only a small difference between the first and second vectors, the vector whose distance from the third vector was closest to this small difference was the third vector itself. This problem occurred with such frequency that in our first tests we received only 0.235 and 0.377 accuracy values with Word2vec and Glove, respectively. Manually excluding the first three vectors from the search for the optimal fourth vector pushed our accuracy up to the 80% range. In order to maximize our accuracy, we decided to not truncate the number of dimensions of the word vectors. Doing so would have boosted the runtime at the expense of some accuracy, but we felt we were able to spend the extra time required to process all dimensions of the vectors because we were iterating over only the vocabulary of the analogies file. Some of the wrong predicted examples are:
Input: Beijing China Brussels Belgium **Predicted:** Europe
Input: dream dreams man men **Predicted:** woman
The example shows that the method sometime fails and gives antonyms which have embedding vectors very close to each other.

7.3 Analogy Task Evaluation

We also evaluated the performance of the embeddings on the analogy task with three novel types of semantic relations not found in the analogy_test.txt file:
tool-function: eye see ear hear; gun shoot phone call; shovel dig knife cut
type-category: apple fruit broccoli vegetable; chair furniture three number; pug dog america country
adjective antonyms: beautiful ugly good bad; loud quiet bright dim; light dark white black
The accuracy rates for each embedding set on each type of analogy are below:

	Tool-Function	Type-Category	Adj. Antonyms
Word2vec	0.667	1.00	1.00
Glove	1.00	1.00	0.67

This is admittedly an extremely small sample size of only nine analogies, but each embedding set performed roughly as well on average as it did with analogy_test.txt. In order to help the accuracy, we tried to think about not only choosing the best words for our analogies, but putting them in the best order. For example, we thought "beautiful ugly good bad" would perform better than "beautiful ugly bad good" because in the former, the first word of each pair has a positve connotation and the second in each has a negative connotation.

7.4 Antonyms Task Analysis

In order to demonstrate that word embeddings sometimes encode antonyms as similar vectors, we found the 10 most similar words to a few verbs according to Word2vec:

Word	Nearest Embeddings
arrive	arriving, arrived, arrives, depart, Arriving, arrival, arive, Arrive, departs, disembark
give	giving, gave, Give, gives, provide, given, Giving, take, get, togive
increase	decrease, increases, increased, reduction, increasing, decreases, rise, decreasing, decline, increasein

In addition to some misspellings and different capitalizations and forms of the words, there are some antonyms that appear among the nearest embeddings. This happens because antonyms like "increase" and "decrease" are used in almost exactly the same set of contexts, so their vector representations are similar as a result.

8 Speech Classification with Word Embeddings

8.1 Approach

In this classification task, we computed the vector for every speech in the combined data set of train+development. For this task we used Glove as pre-trained embeddings as we got a better result for Glove while working with analogy data set. We also did a simple preprocessing task of handling contractions and possessives (e.g. "you 've" vs. "you 've" vs. "you' ve"), We then calculated cosine similarity with the test speech vector and tried to find the maximum. We assigned the label as Obama or Trump based on the nearest match from the list of vectors.

8.2 Comparative Results of Classification Task

While computing the classification tasks, we first calculated the vector of the complete training text and computed its cosine similarity with the test vector. This process gave us an accuracy rate of 0.815 on the Glove embedding.

Then realizing that taking the average of all the speeches may result in averaging out the vectors and may not give a good result. We tried computing the vectors for each speech in the training dataset and comparing it to the test vector. The results we got improved drastically, as we achieved 0.900 accuracy on Glove.

Combining Train + Development data sets resulted in slightly improving the accuracy.

	Accuracy
Word2vec	0.905
Glove	0.905

We got similar results for both Glove and Word2Vec embedding models, showing that even over different large corpora, the same trends in word and n-gram probabilities emerge.

9 Workflow

The three of us all worked very closely on this project, using git and Overleaf and meeting frequently to allow us all to contribute equally to the code and the report. Almost all code written was discussed beforehand by two or three of us.

References

[1] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[2] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.