Project 1: Language Modeling

Scott Cambo (sac355)
Graham Harwood (gdh56)
Jennifer Doughty (jad359)
Malcolm McKinney (mim46)

Part I: Unsmoothed Unigrams and Bigrams

We constructed module **Model.py** for constructing unigram, bigram, and trigram models for a given text corpus. This module computes the desired frequency statistics for a specified n-gram. Pre-processing, such as XML tag removal, identification of sentence start locations, and tokenization of the text, is performed using the NLTK toolkit which has been incorporated into the project 1 library in **project1_lib.py**. The **trainOnTexts** method takes in a list of lists of strings representing a pre-tokenized corpus. In this method, we populate up to three separate dictionaries: **wordCounts**, for which each unique word is a key, and the corresponding word count is a value; **bigramCounts**, for which all possible combinations of 2-word tuples are the keys, and the corresponding 2-gram counts are the values; and **trigramCounts**, for which all possible combinations of 3-word tuples are the keys, and the corresponding 3-gram counts are the values.

We used Python's Counter object from Collections, which is essentially a subtype of dictionary (i.e. hash table), in order to compute the n-gram frequency counts of a corpus. This data structure was selected due to its high-performance tallying capabilities, as well as its ability to perform amortized O(1) lookups and insertions. The Counter object also has robust constructors for object creation, and allowed us to include efficient lines of code, such as: `self.bigramCounts += Counter((x,y) for x, y in zip(*[t[i:] for i in range(2)]))`.

Initially, our module also included code to compute every possible n-gram from a given list of words: `tupleCount = dict().fromkeys( list( itertools.product( self.wordCount.keys(), repeat=n) ), 0)`. Although this code worked as expected on small corpuses, the time required to compute all possible 2-grams is $O(n^2)$ and does not scale well for corpuses with a large vocabulary. Additionally, the $O(n^2)$ memory requirements resulted in memory errors in our test runs. A modified algorithm was thus created to enable random sentence generation that could include never-before-seen n-grams, as described in Part III. We also included functionality to save the state of generated models using Python's object serialization module Pickle so that computations would not need to be recomputed unnecessarily.

Part II: Smoothing, Unknown Words, and Perplexity

The purpose of smoothing is to assign some small probabilities to tuples in our n-gram models that have no word count. For example, we use smoothing for our random sentence generator to choose sequences of words that may not have occurred in the same order as in our corpus. Were we not to smooth our dictionaries, we would be limited to choosing from random word sequences that have been observed in our corpus.

Smoothing a unigram model is trivial since words are considered independently as opposed to in sequences. For bigram models, we take the **bigramCounts** dictionary and run either one of two smoothing methods: Laplacian smoothing, performed by the method **smooth**, or Good Turing smoothing, performed by the method **tsmooth**. The method **smooth** takes in an unsmoothed dictionary and the

number of unique words in the dictionary and returns the smoothed dictionary. We first set **denom** equal to the number of tokens in the corpus added to the number of unique words. This smoothing scheme is to simply "add one" to each value in the dictionary and divide the sum by **denom**. After we finish our calculation for each word in our dictionary, we return the new smoothed dictionary.

The Good Turing smoothing method, **tsmooth**, takes in an unsmoothed dictionary and the number of unique tokens as arguments and returns a Good Turing smoothed dictionary. As in method **smooth**, we set **numer** equal to the number of tokens, **tTokens,** in the corpus added to the number of unique words in the dictionary. We loop through all of the values in the dictionary, add one to the value, and multiply the sum by **numer/tTokens.** We take the new value stored at this key and keep a running sum. After we finish all of the computations, we loop through the dictionary once more and divide each value by the running sum. We finally return this smoothed dictionary.

Perplexity, the degree of unlikeliness of a given randomly generated sentence based on the model, is calculated in the method **perplexity** which takes in a model and a text. The **model** is the trained model for the given data set and the **text** is the string or line of text for which the user wishes to compute perplexity. We add the logs of the probabilities of unigrams in the text and raise two to the sum normalized by the length of the input string. This computation is similar to the built-in perplexity function of nltk but better fits the manner in which we wrote our code.

One toy perplexity calculation that we used our model to perform was the comparison of a short string to a dictionary based on a short string. We computed a model based on the string: *"We, that is to say Malcolm, Jennifer, Scott, and Graham, are working on an NLP project together on our computers and have been for a great deal of time".* The string: "*Malcolm, Jennifer, and Scott all spend a great deal of time on their computers working on a project"* has a somewhat low perplexity score according to our model due to the similarity of the corpora to the sentence. Conversely, the sentence, *"Do not go gentle into that goodnight, Old age should burn and rave at close of day; Rage, rage against the dying of the light."* has a relatively high perplexity. The perplexity for the first test string is relatively small, 19, while the second string gives a perplexity of around 362.

The reason for the much higher perplexity in the second string is in large part due to the many words seen in the second string but not seen in the first string. We dealt with these unknown words by increasing the running sum of values greatly in the case of any unknown words. In this way, unknown words—which should be seen as highly unlikely—will increase the perplexity score of the text a great deal. As a result, the greatest perplexity factor for a model would be the occurrence of any unknown term.

Part III: Random Sentence Generation

Random sentence generation is included as a function within our **project1_lib.py**. The random component of the sentence generation comes from the **sampleFrom** method in **Model.py**. This method will, given a dictionary in which the values are integers or decimals, sample from the keys using the values as the corresponding weight. This functionality is accomplished using a random number generator which computes a number within the span of the sum of the weights. The weight associated with each word key is then subtracted from the random value until the value becomes less than or equal to zero, and then returns the key associated with the most recently subtracted weight value, as shown:

```
def sampleFrom(self,dict):
        rnd = random.random() * sum(dict.values())
        for key, w in weightedDict.iteritems():
            rnd -= w
            if rnd <= 0:
                return key
```

For example, consider the scenario in which the word "improbable" only appears once in a text of 1000 possible words, and the word "probable" appears twice; the value corresponding to "improbable" would be 0.001, and the value corresponding to "probable" would be 0.002. This key-value pattern would continue until the most probabilistic word is reached. If the random number generator happens to generate 0.0013, then the word "probable" would be selected in sampleFrom. More probable words would thus occupy a larger proportion of the sampling space compared to less probable words; thus, if numbers are generated with equal probability, then the words can be selected with the probability corresponding to the text on which the model was trained.

Random sentence generation from a unigram model is very simple: we "roll the dice" and concatenate each selected word token to a string until either a sentence terminator has been reached (i.e. ".", "?", or "!"), or the generator has selected 100 words. The resulting sentences may or may not begin with capitalized words, since the model does not consider whether or not the selected word came from the beginning of a sentence. Since absolutely no previous context is considered when selecting a word, random sentences generated from unigram models do not usually make grammatical or logical sense, as evident in **Table 1** below.

Bigram sentence generation is more complex than unigram sentence generation. Unlike for the unigram model, the selection of the next word is based on the previous word; for example, the first word in a sentence would follow our start-of-sentence marker, "^". Our initial algorithm computed all possible bigrams and used a sampling function to sample from the weighted dictionary. This scheme, however, did not scale up to the larger Wall Street Journal and Movie corporas; thus, we modified the algorithm to utilize a dictionary that contains all of the bigrams that have been seen in the corpora, as well as a "<UNSEEN>" key corresponding to the space of unobserved bigrams, as shown:

```
prevWord = '^'
while (word not in terminators) and (numWords < 100):
     bigrams = gram.bigramsStartingWith(prevWord)
     if len(bigrams) is 0:
          word = gram.sampleFrom(gram.wordCounts)
     else:
          nextgram = gram.sampleFrom(bigrams)
          if nextgram == '<UNSEEN>':
               word = gram.sampleUnknownBigram(bigrams)
          else:
               word = nextgram[1]
     sentence += word + ' '
     numWords += 1
     prevWord = word
```

The unobserved space has a probability equal to the sum of all possible bigrams from the words of the corpora, minus the total sum of bigrams seen in the corpora, divided by the total number of possible bigrams that occur as a result of the words of the corpora. If the dice falls into this space, then the set of unseen bigrams that begin with the most recently selected word is computed and one is selected from a uniform probability distribution using **sampleUnknownBigram**, as shown:

```
def sampleUnknownBigram(self,weightedDict):
     s = Set([])
     for key in weightedDict.keys():
         s.add(key[1])
     samplingSet = Set(self.wordCounts.keys()) - s
     return self.sampleFrom({x: 1 for x in samplingSet})
```

The results from some of our random sentence generation runs are shown below in **Table 1**.

Table 1

Model	**Smoothing Algorithm**	**Sentence Generated**	**Perplexity**
Unigram	**Laplacean**	*Report sure hagen notes unit City step over because Airlines victim upon the company , Nazi operations official 1970s abroad.*	11.02
	Good Turing	*has of marked was desperate from bill extremely , poor 87 and School .*	22.75
Bigram	**Laplacean**	*Inexperienced dark-haired diving popsicle early.*	5.33
	Good Turing	*Among traveling Kuwait 1987 duties cancel inadvertent defeated miscellaneous distinctly construct singles.*	4.06

Part IV: Email Author Prediction

For Email Author Prediction, we import the training set and parse it to a dictionary in which the key is an author and the value is a list of all of that author's emails. To create a usable model, we made dictionaries with the key as the author and the value as a dictionary of words and their respective frequencies (# of occurrences / # of total words from this author) for both bigram and unigram frequencies. We next imported both the validation set and the test set as respective lists in which each value is an email from the set. Next, we iterate through the unknown email list and call a method which creates both the unigram and bigram dictionaries for the email and compares each frequency for a word or n-gram to its corresponding frequency in each author's frequency distribution dictionaries. To tally the score for a match for a word or n-gram that does exist in the corresponding distribution model, we take the absolute value of the difference between the frequency of the model and the frequency in the unknown email. If the word or n-gram does not exist in the model, then we take the full frequency distribution for that word in the unknown email and add it to the score. Our system then takes the reciprocal of this so that the author with the highest score is the best match.

Before determining if an author is a better match than another, we experimented with adding both unigram and bigram scores. However, we found that using the bigram scores alone worked best. Ultimately, we only achieved a 58% detection rate, but this might have been improved if we could go beyond mere n-gram frequency comparisons and also consider the location within a sentence an n-gram occurs. For example, a particular author might have a tendency to start a sentence with “However, …” while another might have a preference for “But, …”.

We also implemented several additional algorithms to experimentally calculate author similarity scores. The first algorithm was fairly simple; it went through the dictionary of one author as well as the dictionary created out of the word counts of the unknown emails, and counted similar words. The similar word count was then normalized to the total length of the unknown email. The logic behind this algorithm was that each author has a certain set of words he or she is more likely to use. As a result, any new unknown document written by that author would probably show a greater likelihood for that same author.

The other additional similarity calculation algorithm was loosely based upon the Euclidean distance between two vectors. The values in the dictionaries that corresponded to words made up the vectors. The **euclidDistCompare** method compares the dictionary of the author to the dictionary created by the modeling of the unknown email, and calculates the Euclidean distance between the two vectors. The method treats any word in the unknown email model as adding the value itself because that is the distance that the value would have to a 0 in the corresponding author vector. The inverse of the total for the email is returned in order to have the lowest possible score be the best match.

Part V: Extension

For our extension, we implemented two different types of smoothing. First, we implemented LaPlacian smoothing in order to understand how smoothing would affect the word counts. This understanding allowed us to test other functions that required a probabilistically smoothed data set. We then decided to implement a Turing smoothed model after noticing the greater accuracy of the Turing smoothed model in lecture. Specifically, we wanted to compare the results of LaPlacian and Good Turing smoothing with regards to random sentence generation. As evident in **Table 1**, the results varied and generally Turing smoothing provided slightly more intelligible sentences than LaPlacian smoothing.

Part VI: Contributions:

Note that all team members contributed to the final report.

Jennifer Doughty	Jennifer worked on unigram, bigram, and trigram model generation; random sampling and sentence generation; consolidated code from the team; and edited the report.
Malcolm Mckinney	Malcolm wrote the final report and was the resident Python expert/debugger.
Scott Cambo	Scott worked on importing data from the corpora, user interfaces, unigram sentence generation, and author detection methods.
Graham Harwood	Graham worked on the smoothing, perplexity, and the Euclidean distance between two dictionaries (comparison), simple author comparison, testing and debugging. He also wrote the initial report.

Toolkits used : NLTK 2.0 library (nltk.org)

Semantics:	Dan Garrette <http://www.cs.utexas.edu/~dhg/>, Austin, USA (nltk.sem,nltk.inference)
Parsing:	Peter Ljunglöf <http://www.cse.chalmers.se/~peb/>, Gothenburg, Sweden (nltk.parse, nltk.featstruct)
Metrics:	Joel Nothman <http://joelnothman.com/>, Sydney, Australia (nltk.metrics,nltk.tokenize.punkt)
Python 3:	Mikhail Korobov <http://kmike.ru/>, Ekaterinburg, Russia
Integration:	Morten Minde Neergaard <http://8d.no/>, Oslo, Norway
Releases:	Steven Bird <http://estive.net>, Melbourne, Australia