Neel Narayan
CS 109
Fall 2021

Spell Checker for Fake News Detection
Probability Challenge

November 28, 2021

1 Project Description

Implemented below is a spell checker trained on a corpus of political rhetoric. The program uses Bayesian Probabilities and Bigrams to determine words that are incorrectly spelled in a sentence. We use additive smoothing (laplacian +1 smoothing) to get rid of zero probabilities and report high accuracy ($\sim$ 85.43333%) on sentences where the replacement words are included in the corpus. In the interest of understanding how to algorithmically detect fake news, I used a political corpus, as one of the most prominent signs of fake news is articles riddled with typos and spelling errors (1, 2, 3). In this report, we will discuss the implications of this project, the code, and different ways to improve the algorithm.
Linked is a Demonstration of The Algorithm: Demonstration

2 Implications and Impact

Fake News poses one of the largest threats to our society. In a world so tightly connected by the internet, disseminating false information through the online medium has profound consequences. Average human attention spans have declined "precipitiously" over the last 13 years (4), making it critically important that fake news is immediately found, as short attention spans will make it hard for people to differentiate between fake and real news. Disinformation campaigns undermine democracy, promote distrust in the media, and smear the integrity of established institutions. Additionally, the real world impacts of fake news are far reaching and can severely harm people. For example, YouTube recently announced that it would remove all content that "spreads misinformation about all approved vaccines" as too many people were refusing to get vaccinated (5).

One of the most common signs of fake news is misspelled words. Sites that report real news (though there might be bias), such as The New York Times, CNN, or Fox, have meticulous editors that are diligent in their editing work; it is for this reason that typos seldom occur in their articles. With fake news, articles are typically published by members of fringe movements on sites that aren't financially able to hire editors. Thus, there typically tend to be more typos. If a computer can exploit the presence of misspelled words, we can better detect fake news and get rid of it.

Here, we report a method of detecting typos in sentences. If used in conjunction with a machine learning classification algorithm, we could scrape websites and accurately determine which sites contain fake news; based on the amount of misspelled words, we could definitively classify, with high confidence, sites as containing fake news or not.

3 Algorithm

3.1 Bigram Creation

We begin by importing all necessary packages:

```
import csv # read from file
import math # exponentiation
import string # string processing
import nltk # get corpus
import time # efficiency
import heapq # 'top k'
import pandas as pd # csv processing
import numpy as np # numerical processing
from operator import itemgetter # sorting
from english_words import english_words_set # extra words
```

We then parse the 'True.csv' file, which contains 21,417 recent articles of true news (a sample article is included at the end of this report). Additionally, we append tokens at the beginning and end of sentences to create a more consistent probability model. After this, we get a total of 8,579,806 words (which is not an ideal size for a training corpus but is enough to create a functional model) in our data frame, including the start and end tokens:

```
df = pd.read_csv("True.csv")

words = []
words.append('<s>')
for i in range(len(df) - 1):
    for word in df.iloc[i][1].split():
        words.append(word.lower())
        if '.' in word and word.index('.') == (len(word) - 1):
            words.append('</s>')

print(len(words))
```

We can then create our bigrams. Bigrams are groupings of two consecutive words in a sentence. For example, for the sentence "I am a student at Stanford University," the bigrams would be, after accouting for case sensitivity, [('i', 'am'), ('am', 'a'), ('a', 'student'), ('student', 'at'), ('at', 'stanford'), ('stanford', 'university')]:

```
def create_bigrams(words):
    bigrams = []
    bigrams_freq = {}
    word_freq = {}
    for i in range(len(words) - 1):
        bigram = (words[i], words[i + 1])
        if i < len(words) - 1:
            bigrams.append(bigram)
            if bigram in bigrams_freq:
                bigrams_freq[bigram] += 1
            else:
                bigrams_freq[bigram] = 1
        if words[i] in word_freq:
            word_freq[words[i]] += 1
        else:
            word_freq[words[i]] = 1
    return bigrams, word_freq, bigrams_freq
```

Once we have our bigrams, we can calculate our bigram probabilities. A bigram probability is a condtional probability representing the probability that a word occurs given some specic word occurred before it. For the bigram ('white', 'house), the bigram probability would be calculated as

$$P(\text{house}|\text{white}) = \frac{C(\text{white house})}{C(\text{white})}$$

where $C(x)$ denotes the count, or frequency, of x. In my implementation, I use laplacian add 1 smoothing to get rid of zero probabilities. This ensures that we are able to make predictions even based on values, or in this case, words, that we haven't seen before:

```
def calculate_bigrams_probability(bigrams, word_freq, bigrams_freq):
    probabilities = {}
    for bigram in bigrams:
        word1 = bigram[0]
        probabilities[bigram] = math.log((bigrams_freq[bigram] + 1) / (word_freq[word1] +
            8579806)) # 8579806 is the amount of words in the corpus
    return probabilities
```

A second function, calculate_unknown_bigrams_probability(), is also implemented, where the unknown word is added to the dictionary of words and is given a frequency of 1 (based on laplacian +1 smoothing):

```
def calculate_unknown_bigrams_probability(bigram):
    if bigram[0] not in word_freq:
        word_freq[bigram[0]] = 1
    bigram_probabilities[bigram] = math.log(1 / (word_freq[bigram[0]] + 8579806))
```

For testing purposes, we observe the probabilites of two bigrams ('donald', 'trump') and ('trump', 'donald') to ensure accuracy in our probabilistic calculations:

```
bad_bigram = ('trump', 'donald')
if bad_bigram not in bigrams:
    calculate_unknown_bigrams_probability(bad_bigram)
print(bigram_probabilities[bad_bigram])
good_bigram = ('donald', 'trump')
print(bigram_probabilities[good_bigram])
```

The log probability of the good bigram is significantly larger than that of the bad bigram, which logically holds (-15.969165271621279 for the bad bigram and -7.203158871411385 for the good bigram).

3.2 Capturing The Incorrect Word

Now that we have created our bigrams, we can test our model. For this example, we will use the following sentence:

The former president, Donald Frump, lived in The White House.

where 'Frump' should be 'Trump.' Because our bigrams are absent of punctuation and use all lowercase letters, we first need to format our sentence:

```
def format_sentence(sentence):
    return sentence.translate(str.maketrans('', '', string.punctuation)).lower()

sentence = format_sentence(original_sentence)
print(sentence)
```

Our new sentence is

the former president donald frump lived in the white house

Our bigram creation function takes in a list of words, so we feed an array of the words split at each space into the create_bigrams() function.

```
sentence_words = sentence.split()
s_bigrams, s_word_freq, s_bigrams_freq = create_bigrams(sentence_words)
```

We now can calculate our bigram probabilities for each bigram. This is done by using the bigrams in our sentence and getting each probability based on its probability calculated from the original corpus of words:

```
s_bigrams_probabilities = {}
for bigram in s_bigrams:
    if bigram not in bigram_probabilities:
        calculate_unknown_bigrams_probability(bigram)
    s_bigrams_probabilities[bigram] = bigram_probabilities[bigram]

print(s_bigrams_probabilities)
```

Our bigram probabilities are:

$$\begin{aligned}
\text{('the', 'former')} &: -8.91006 \\
\text{('former', 'president')} &: -8.98329 \\
\text{('president', 'donald')} &: -7.28443 \\
\text{('donald', 'frump')} &: -15.96612 \\
\text{('frump', 'lived')} &: -15.96492 \\
\text{('lived', 'in')} &: -10.88978 \\
\text{('in', 'the')} &: -5.36711 \\
\text{('the', 'white')} &: -7.43147 \\
\text{('white', 'house')} &: -7.10778.
\end{aligned}$$

Immediately, we notice that the two smallest probabilities of -15.96612 and -15.96492 occur with ('donald', 'frump') and ('frump', 'lived'), respectively. This makes sense, as there is a very low probability that 'frump' will occur after 'donald,' and 'lived' will occur after 'frump' because 'frump' isn't a word. We determine the two bigrams with the smallest probabilities and as expected, they are ('donald', 'frump') and ('frump', 'lived'):

```
two_smallest = sorted(s_bigrams_probabilities.items(), key=itemgetter(1))[:2]
print(two_smallest)
```

As a note, two_smallest is formatted as [((a, b), p1), ((b, c), p2)], where a, b, and c are the words, and p1 and p2 are the probabilities. Based on the way that two_smallest is formatted, we know that the incorrect word will always be the second word in the first tuple, or first word in the second tuple. Equivalently, the incorrect word will always be b. Thus, to access the word, we can simply use indexing:

```
incorrect_word = two_smallest[0][0][1] # either approach will capture the incorrect word
# incorrect_word = two_smallest[1][0][0]

print(incorrect_word)
```

'frump' is printed, as desired.

3.3 Acquiring More Words

Our corpus that we trained on contains many words, but is not comprehensive. Additionally, the correctly spelled word may not even be in the original training corpus. For this reason, we need a larger list of English Words. Here, we use the english_words_set provided by Python (4).

```
english_words = list(english_words_set)
```

This set contains only 25,487 words, which is minuscule in the world of Natural Language Processing, but because of spatial and computational limits on my laptop, this data set was chosen. For context, words like 'hello,' 'trump,' 'jump,' and 'esoteric' are included where as 'clinton' and 'obama' are not. Thus, this is not the best data set for our purposes but is used here because of ease of both usage and access.

3.4 Calculating Minimum Edit Distances

The next step is to determine minimum edit distances between words. This will allow us to determine which word was supposed to be typed. In my implementation, I use a Levenshtein Distance Calculation, which is a type of string metric that represents the edit distance between two strings. Insertions and deletions have a cost of 1 and substitutions have a cost of 2. As an example, we can take the words 'apple' and 'trample.' The Levenshtein Distance between these two words is 4:

$$\begin{aligned} \text{insert t, apple} &\rightarrow \text{tapple } (+1) \\ \text{insert r, tapple} &\rightarrow \text{trapple } (+1) \\ \text{substitute p with m, trapple} &\rightarrow \text{trample } (+2) \end{aligned}$$

Mathematically,

$$\text{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0 \\ \begin{cases} \text{lev}_{a,b}(i-1,j)+1 & \text{otherwise} \\ \text{lev}_{a,b}(i,j-1)+1 & \text{otherwise} \\ \text{lev}_{a,b}(i-1,j-1)+1_{a_i \neq b_i} & \text{otherwise} \end{cases} \end{cases}$$

with a and b as the words and i and j as the letters. In code:

```
def minimum_edit_distance(word1, word2):
    # levenshtein distance minimum edit distance table
    amt_rows = len(word1) + 1
    amt_cols = len(word2) + 1
    edit_distance = np.zeros((amt_rows, amt_cols))

    for r in range(1, amt_rows):
        edit_distance[r][0] = r
    for c in range(1, amt_cols):
        edit_distance[0][c] = c

    for row in range(1, amt_rows):
        for col in range(1, amt_cols):
            if word1[row - 1] == word2[col - 1]:
                edit_distance[row][col] = edit_distance[row - 1][col - 1]
```

```
            else:
                edit_distance[row][col] = min(edit_distance[row][col - 1] + 1,
                                              edit_distance[row - 1][col] + 1,
                                              edit_distance[row - 1][col - 1] + 2)

    return edit_distance[amt_rows - 1][amt_cols - 1]
```

Using a tabular approach, the minimum edit distance between two words will always be at the bottom rightmost cell of the matrix. As expected, minimum_edit_distance('apple', 'trample') returns 4.

3.5 Determining The Correct Word

We have already determined our incorrect word to be 'frump.' To determine a list of potentially correct words, we can get a list of words, in this case of length 20, that have the smallest minimum edit distances to 'frump.' The words in the english corpus are compared to 'frump.'

```
distances = {}
for i in range(len(english_words) - 1):
    distances[english_words[i]] = minimum_edit_distance(english_words[i], incorrect_word)

twenty = sorted(distances.items(), key=itemgetter(1))[:20]

print(twenty[0:len(twenty)-1])
```

We get the following: [('rump', 1.0), ('forum', 2.0), ('trump', 2.0), ('rum', 2.0), ('crump', 2.0), ('grump', 2.0), ('fum', 2.0), ('ramp', 3.0), ('pump', 3.0), ('from', 3.0), ('farm', 3.0), ('dump', 3.0), ('grumpy', 3.0), ('arum', 3.0), ('jump', 3.0), ('bump', 3.0), ('fume', 3.0), ('up', 3.0), ('lump', 3.0)]. Each tuple contains the word as well as its minimum edit distance to, in this case, 'frump.' We now have a list of potentially correct words. To determine the most likely correct word, we then create new bigrams with each word and again calculate the bigram probabilities. For example, the new bigrams would be ('donald', 'rump'), ('donald', 'forum'), ('donald', 'trump'), ('donald', 'rum'), ('donald', 'crump'), ..., and ('donald', 'lump'):

```
first_word = two_smallest[0][0][0]
new_probabilities = {}

for i in range(len(twenty)):
    bigram = (first_word, twenty[i][0])
    if bigram not in bigram_probabilities:
        calculate_unknown_bigrams_probability(bigram)
    new_probabilities[bigram] = bigram_probabilities[bigram]

largest = sorted(new_probabilities.items(), key=itemgetter(1))[-1:]

print(largest)
correct_word = largest[0][0][1]
print(correct_word)
```

We create the bigrams, calculate the probabilities, and select the bigram with the largest probability. In this example, largest is printed as [(('donald', 'trump'), -7.203158871411385)] and correct_word is printed as trump. We have successfully identified our desired word as 'trump.'

3.6 Additional Experimentation

To make the program interactive, user input is also captured:

```
print("Original Sentence:", original_sentence)
print("Did you mean to replace", incorrect_word, "with", correct_word, "?")
response = input("")

final_sentence = original_sentence

if response == "yes":
    for word in original_sentence.translate(str.maketrans('', '',
        string.punctuation)).split():
        if word.lower() == incorrect_word:
            final_sentence = final_sentence.replace(word, correct_word)
            break
    print("Updated Sentence: ", final_sentence)
else:
    print("We could not find a suitable change for", incorrect_word)
```

A trie was also implemented to make searching faster. If a larger corpus was used, searching through the entire corpus and checking the minimum edit distance for every single word would be incredibly computationally heavy. The other benefit is that a trie makes it easy to find words given a specific minimum edit distance. Thus, using a trie for minimum edit distance searching is preferable:

```
class Trie:
    def __init__(self):
        self.word = None
        self.children = {}
    def add(self, word):
        node = self
        for l in word:
            if l not in node.children: # only add letter if it isn't already there
                node.children[l] = Trie()
            node = node.children[l]
        node.word = word

def find(node, word, letter, previous, size, words):
    row = [previous[0] + 1]

    for column in range(1, len(word) + 1):
        delete = previous[column] + 1 # deletions cost 1
        insert = row[column - 1] + 1 # insertions cost 1
        replace = 0

        if word[column - 1] == letter:
            replace = previous[column - 1]
        else:
            replace = previous[column - 1] + 2 # substitutions cost 2
        row.append(min(delete, insert, replace))
    if min(row) <= size:
        for letter in node.children:
            find(node.children[letter], word, letter, row, size, words)
    if node.word != None and row[len(row) - 1] <= size:
        words.append((node.word, row[len(row) - 1]))

def search(word, size):
    row = range(len(word) + 1)
    words = []
    for l in trie.children:
        find(trie.children[l], word, l, row, size, words) # recursively search
    return words

start = time.time()
trie = Trie()
for w in english_words:
    trie.add(w)
```

```
words = search("frump", 3)
end = time.time()
print(words)
print("Search took %g seconds" % (end - start))
```

Based on timing considerations, we can find words with specified minimum edit distances significantly faster using a trie rather than using a brute-force iterative approach. The way the trie works is it combines all common prefixes so that when using a tabular approach in the minimum edit distance, we only look at the last row as we change words. This creates a tree-like structure where each node contains a separate branch to a new word. For this example, the following was returned in 1.16485 seconds: [('arum', 3), ('crump', 2), ('trump', 2), ('grumpy', 3), ('grump', 2), ('pump', 3), ('fume', 3), ('fum', 2), ('form', 3), ('forum', 2), ('farm', 3), ('firm', 3), ('from', 3), ('bump', 3), ('lump', 3), ('romp', 3), ('ramp', 3), ('rumpus', 3), ('rumple', 3), ('rump', 1), ('rum', 2), ('hump', 3), ('drum', 3), ('dump', 3), ('up', 3), ('jump', 3)]. When tested with a trie, it became much more efficient to find the correct word.

*** As a side note, I implemented my first trie in 9^{th} grade. I adopted it to work with a minimum edit distance table a few weeks ago, and after completion and efficiency testing, I looked up a visualization program to make sure that my trie was implemented properly. As I was searching, I came a cross this website that has an incredibly similar implementation to my trie. I would like to make it clear that I did not know about or consult this website when I coded my trie for minimum edit distances, and did not copy code. The only feature I added from the website's implementation was timing; I think this is a good metric to gauge efficiency. I am working under the assumption that this is a common trie implementation, which is why such similarity exists.***

4 Algorithmic Improvement

4.1 Lack of Words

The most pressing issue with the current implementation is that it is not robust enough; a lack of training vocabulary as well a small word list to create new bigrams limits the different suggestions that can be returned by the program. A larger amount of words will (1) allow for more bigrams and (2) ensure more accurate probabilities. As an example, this program fails on the sentence

i consider myself a left meaning individual

where 'meaning' should be 'leaning.' It is able to correctly identify the misspelled word, 'meaning.' However, it fails in determining the correct word. When it captures the words with the minimum edit distance to 'meaning' and calculates the new bigram probabilities, it suggests changing 'meaning' to 'mange.' This is because the word 'leaning' is not in the corpus of english words. As a test, I calculated the bigram probability of 'left leaning' relative to the other bigram probabilities, and P(left leaning) was significantly higher than all other probabilities, as desired. Thus, the program is limited not in its capabilities or algorithmic strength, but rather in its lack of data.

4.2 Trigrams and N-Grams

The current approach uses bigram probability calculations. But what would happen if we used trigrams? Or n-grams with n greater than 2? This would increase our probability accuracy, but only up until a certain point. Say, for example, that we used a 9-gram. Thus, for the following sentence, "The entire human population of seven and a half billion people could fit inside of the city of Los Angeles," our 9-grams would be ('the', 'entire', 'human', 'population', 'of', 'seven', 'and', 'a', 'half'), ('entire', 'human', 'population', 'of', 'seven', 'and', 'a', 'half', 'billion'), ('human', 'population', 'of', 'seven', 'and', 'a', 'half', 'billion', 'people'), ..., and ('could', 'fit', 'inside', 'of', 'the', 'city', 'of', 'los', 'angeles'). When calculating the 9-gram probabilities, we will, for example, calculate the conditional probability of city:

$$P(\text{city}|\text{half billion people could fit inside of the}) = \frac{C(\text{half billion people could fit inside of the city})}{C(\text{half billion people could fit inside of the})}$$

Finding the specific terms in the numerator and denominator in text is very difficult, as the odds of each word in 'half billion people could fit inside of the city' consecutively occurring is incredibly small. The issue of a zero probability can be solved with laplacian smoothing, but if there are multiple such terms, their probabilities could potentially be the same. Thus, using too high of an n-gram is harmful to probabilities. If we use bigram probabilities, however, there isn't much conditioning, so probabilities and their relative magnitudes could

be miscalibrated. As such, potentially using a trigram (3-gram) or quadgram (4-gram) would give us more accurate probabilities, which would be needed if our corpus and word lists were bigger so that differentiation in probabilities between different words and phrases could occur.

4.3 False Positives

In this program, a least likely bigram is always returned. What happens if all words in a given sentence are spelled properly, however? The program will incorrectly return a bigram with both words spelled correctly. This is because each bigram probability is compared relative to the others in the sentence. To combat the issue of false positives and incorrectly returning correctly spelled words, I decided to set a probability threshold, denoted here as x (in the program, it was around -15.1), calculated as the average of tens of thousands of bigram probabilities. Here, a bigram would only be returned if its probability was below x. The issue with this approach arises when certain probabilities in a given sentence are low, which typically occur if rare words are used. Take the following sentence:

The obelus, representative of the division symbol (÷), is also used grammatically to mark questionable text.

Though all words are spelled properly, the probability of 'representative' occurring given that 'obelus' occured before it is incredibly low. Specific to this program, because the corpus is politically related, words like 'state' or 'political' would occur before 'representative' with a much higher probability than that of 'obelus.' Thus, the program would incorrectly return the word of 'obelus', even though that is the desired word. It is for this reason that the threshold approach doesn't work. As a potential solution, a machine learning algorithm could use Naïve Bayes Classification or Logistic Regression to determine whether or not a bigram is actually incorrect; given training data in the form of a probability and a corresponding binary classification representing if a bigram is valid or not, we could set up a boundary, or threshold value, to help in accurate classification of misspelled words.

WASHINGTON (Reuters) - Transgender people will be allowed for the first time to enlist in the U.S. military starting on Monday as ordered by federal courts, the Pentagon said on Friday, after President Donald Trump's administration decided not to appeal rulings that blocked his transgender ban. Two federal appeals courts, one in Washington and one in Virginia, last week rejected the administration's request to put on hold orders by lower court judges requiring the military to begin accepting transgender recruits on Jan. 1. A Justice Department official said the administration will not challenge those rulings. "The Department of Defense has announced that it will be releasing an independent study of these issues in the coming weeks. So rather than litigate this interim appeal before that occurs, the administration has decided to wait for DOD's study and will continue to defend the president's lawful authority in District Court in the meantime," the official said, speaking on condition of anonymity. In September, the Pentagon said it had created a panel of senior officials to study how to implement a directive by Trump to prohibit transgender individuals from serving. The Defense Department has until Feb. 21 to submit a plan to Trump. Lawyers representing currently-serving transgender service members and aspiring recruits said they had expected the administration to appeal the rulings to the conservative-majority Supreme Court, but were hoping that would not happen. Pentagon spokeswoman Heather Babb said in a statement: "As mandated by court order, the Department of Defense is prepared to begin accessing transgender applicants for military service Jan. 1. All applicants must meet all accession standards." Jennifer Levi, a lawyer with gay, lesbian and transgender advocacy group GLAD, called the decision not to appeal "great news." "I'm hoping it means the government has come to see that there is no way to justify a ban and that it's not good for the military or our country," Levi said. Both GLAD and the American Civil Liberties Union represent plaintiffs in the lawsuits filed against the administration. In a move that appealed to his hard-line conservative supporters, Trump announced in July that he would prohibit transgender people from serving in the military, reversing Democratic President Barack Obama's policy of accepting them. Trump said on Twitter at the time that the military "cannot be burdened with the tremendous medical costs and disruption that transgender in the military would entail." Four federal judges - in Baltimore, Washington, D.C., Seattle and Riverside, California - have issued rulings blocking Trump's ban while legal challenges to the Republican president's policy proceed. The judges said the ban would likely violate the right under the U.S. Constitution to equal protection under the law. The Pentagon on Dec. 8 issued guidelines to recruitment personnel in order to enlist transgender applicants by Jan. 1. The memo outlined medical requirements and specified how the applicants' sex would be identified and even which undergarments they would wear. The Trump administration previously said in legal papers that the armed forces were not prepared to train thousands of personnel on the medical standards needed to process transgender applicants and might have to accept "some individuals who are not medically fit for service." The Obama administration had set a deadline of July 1, 2017, to begin accepting transgender recruits. But Trump's defense secretary, James Mattis, postponed that date to Jan. 1, 2018, which the president's ban then put off indefinitely. Trump has taken other steps aimed at rolling back transgender rights. In October, his administration said a federal law banning gender-based workplace discrimination does not protect transgender employees, reversing another Obama-era position. In February, Trump rescinded guidance issued by the Obama administration saying that public schools should allow transgender students to use the restroom that corresponds to their gender identity.

In [1]:
```python
import csv # read from file
import math # exponentiation
import string # string processing
import nltk # get corpus
import time # efficiency
import heapq # 'top k'
import pandas as pd # csv processing
import numpy as np # numerical processing
from operator import itemgetter # sorting
from english_words import english_words_set # extra words
```

In [2]:
```python
nltk.download('words')
from nltk.corpus import words
```

```
[nltk_data] Downloading package words to
[nltk_data]     /Users/neelnarayan/nltk_data...
[nltk_data]   Package words is already up-to-date!
```

In [3]:
```python
df = pd.read_csv("True.csv")

words = []
words.append('<s>')
for i in range(len(df) - 1):
    for word in df.iloc[i][1].split():
        words.append(word.lower())
        if '.' in word and word.index('.') == (len(word) - 1):
            words.append('</s>')

print(len(words))
```

```
8579806
```

In [4]:
```python
# print sample article
print(df.iloc[1][1])
```

```
WASHINGTON (Reuters) - Transgender people will be allowed for the first time to enlist in the U.S. military starting
on Monday as ordered by federal courts, the Pentagon said on Friday, after President Donald Trump’s administration de
cided not to appeal rulings that blocked his transgender ban. Two federal appeals courts, one in Washington and one i
n Virginia, last week rejected the administration’s request to put on hold orders by lower court judges requiring the
military to begin accepting transgender recruits on Jan. 1. A Justice Department official said the administration wil
l not challenge those rulings. “The Department of Defense has announced that it will be releasing an independent stud
y of these issues in the coming weeks. So rather than litigate this interim appeal before that occurs, the administra
tion has decided to wait for DOD’s study and will continue to defend the president’s lawful authority in District Cou
rt in the meantime,” the official said, speaking on condition of anonymity. In September, the Pentagon said it had cr
eated a panel of senior officials to study how to implement a directive by Trump to prohibit transgender individuals
from serving. The Defense Department has until Feb. 21 to submit a plan to Trump. Lawyers representing currently-serv
ing transgender service members and aspiring recruits said they had expected the administration to appeal the rulings
to the conservative-majority Supreme Court, but were hoping that would not happen. Pentagon spokeswoman Heather Babb
said in a statement: “As mandated by court order, the Department of Defense is prepared to begin accessing transgende
r applicants for military service Jan. 1. All applicants must meet all accession standards.” Jennifer Levi, a lawyer
with gay, lesbian and transgender advocacy group GLAD, called the decision not to appeal “great news.” “I’m hoping it
means the government has come to see that there is no way to justify a ban and that it’s not good for the military or
our country,” Levi said. Both GLAD and the American Civil Liberties Union represent plaintiffs in the lawsuits filed
against the administration. In a move that appealed to his hard-line conservative supporters, Trump announced in July
that he would prohibit transgender people from serving in the military, reversing Democratic President Barack Obama’s
policy of accepting them. Trump said on Twitter at the time that the military “cannot be burdened with the tremendous
medical costs and disruption that transgender in the military would entail.” Four federal judges - in Baltimore, Wash
ington, D.C., Seattle and Riverside, California - have issued rulings blocking Trump’s ban while legal challenges to
the Republican president’s policy proceed. The judges said the ban would likely violate the right under the U.S. Cons
titution to equal protection under the law. The Pentagon on Dec. 8 issued guidelines to recruitment personnel in orde
r to enlist transgender applicants by Jan. 1. The memo outlined medical requirements and specified how the applicant
s’ sex would be identified and even which undergarments they would wear. The Trump administration previously said in
legal papers that the armed forces were not prepared to train thousands of personnel on the medical standards needed
to process transgender applicants and might have to accept “some individuals who are not medically fit for service.”
The Obama administration had set a deadline of July 1, 2017, to begin accepting transgender recruits. But Trump’s def
ense secretary, James Mattis, postponed that date to Jan. 1, 2018, which the president’s ban then put off indefinitel
y. Trump has taken other steps aimed at rolling back transgender rights. In October, his administration said a federa
l law banning gender-based workplace discrimination does not protect transgender employees, reversing another Obama-e
ra position. In February, Trump rescinded guidance issued by the Obama administration saying that public schools shou
ld allow transgender students to use the restroom that corresponds to their gender identity.
```

In [5]:
```python
def create_bigrams(words):
    bigrams = []
    bigrams_freq = {}
    word_freq = {}
    for i in range(len(words) - 1):
        bigram = (words[i], words[i + 1])
        if i < len(words) - 1:
            bigrams.append(bigram)
            if bigram in bigrams_freq:
                bigrams_freq[bigram] += 1
            else:
                bigrams_freq[bigram] = 1
        if words[i] in word_freq:
            word_freq[words[i]] += 1
        else:
            word_freq[words[i]] = 1
    return bigrams, word_freq, bigrams_freq
```

In [6]:
```python
bigrams, word_freq, bigrams_freq = create_bigrams(words)
```

In [7]:
```python
def calculate_bigrams_probability(bigrams, word_freq, bigrams_freq):
    probabilities = {}
    for bigram in bigrams:
        word1 = bigram[0]
        probabilities[bigram] = math.log((bigrams_freq[bigram] + 1) / (word_freq[word1] + 8579806))
    return probabilities
```

In [8]:
```python
bigram_probabilities = calculate_bigrams_probability(bigrams, word_freq, bigrams_freq)
```

In [9]:
```python
def calculate_unknown_bigrams_probability(bigram):
    if bigram[0] not in word_freq:
        word_freq[bigram[0]] = 1
    bigram_probabilities[bigram] = math.log(1 / (word_freq[bigram[0]] + 8579806))
```

In [10]:
```python
bad_bigram = ('trump', 'donald')
if bad_bigram not in bigrams:
    calculate_unknown_bigrams_probability(bad_bigram)
print(bigram_probabilities[bad_bigram])
good_bigram = ('donald', 'trump')
print(bigram_probabilities[good_bigram])
```

```
-15.969165271621279
-7.203158871411385
```

In [11]:
```python
# prints most common bigram
print(max(bigram_probabilities, key = bigram_probabilities.get))

# prints top 25 bigrams from the article with the largest probabilities
largest_25 = heapq.nlargest(25, bigram_probabilities, key = bigram_probabilities.get)
for l in largest_25:
    print(l, ": ", bigram_probabilities[l])
```

```
('of', 'the')
('of', 'the') :  -5.216751290905708
('</s>', 'the') :  -5.227535124695011
('in', 'the') :  -5.3671079102834485
('to', 'the') :  -5.984616689707037
('said.', '</s>') :  -5.987726681362507
('(reuters)', '-') :  -6.003846840240963
('in', 'a') :  -6.176614412566284
('on', 'the') :  -6.2631386141651815
('for', 'the') :  -6.333667049584559
('the', 'united') :  -6.466039735431085
('and', 'the') :  -6.583244863306866
('with', 'the') :  -6.653984591211452
('the', 'u.s.') :  -6.663033999578238
('at', 'the') :  -6.685055012917045
('by', 'the') :  -6.715700917703136
('said', 'on') :  -6.757655965479918
('to', 'be') :  -6.82078160220548
('that', 'the') :  -6.835175007126235
('</s>', 'in') :  -6.880772222711231
('of', 'a') :  -6.882181824725299
('from', 'the') :  -6.886744683121814
('said', 'the') :  -6.8921399083808375
('united', 'states') :  -6.915384600964873
('</s>', 'he') :  -6.959703707834659
('said', 'in') :  -6.9872111545108755
```

In [12]:
```python
# test sentence
original_sentence = "The former president, Donald Frump, lived in the White House."
```

In [13]:
```python
def format_sentence(sentence):
    return sentence.translate(str.maketrans('', '', string.punctuation)).lower()

sentence = format_sentence(original_sentence)
print(sentence)
```

```
the former president donald frump lived in the white house
```

In [14]:
```python
sentence_words = sentence.split()
s_bigrams, s_word_freq, s_bigrams_freq = create_bigrams(sentence_words)
```

In [15]:
```python
print(s_bigrams)
```

```
[('the', 'former'), ('former', 'president'), ('president', 'donald'), ('donald', 'frump'), ('frump', 'lived'), ('live
d', 'in'), ('in', 'the'), ('the', 'white'), ('white', 'house')]
```

In [16]:
```python
print(s_word_freq)
```

```
{'the': 2, 'former': 1, 'president': 1, 'donald': 1, 'frump': 1, 'lived': 1, 'in': 1, 'white': 1}
```

In [17]:
```python
print(s_bigrams_freq)
```

```
{('the', 'former'): 1, ('former', 'president'): 1, ('president', 'donald'): 1, ('donald', 'frump'): 1, ('frump', 'liv
ed'): 1, ('lived', 'in'): 1, ('in', 'the'): 1, ('the', 'white'): 1, ('white', 'house'): 1}
```

In [18]:
```python
s_bigrams_probabilities = {}
for bigram in s_bigrams:
    if bigram not in bigram_probabilities:
        calculate_unknown_bigrams_probability(bigram)
    s_bigrams_probabilities[bigram] = bigram_probabilities[bigram]

print(s_bigrams_probabilities)
```

```
{('the', 'former'): -8.910056249919954, ('former', 'president'): -8.983292059499819, ('president', 'donald'): -7.2844
29390975789, ('donald', 'frump'): -15.966117792178112, ('frump', 'lived'): -15.964921977038653, ('lived', 'in'): -10.
889778698156023, ('in', 'the'): -5.3671079102834485, ('the', 'white'): -7.431466730542328, ('white', 'house'): -7.107
781685333634}
```

In [19]:
```python
smallest_probability = min(s_bigrams_probabilities, key=s_bigrams_probabilities.get)
print(smallest_probability)
```

```
('donald', 'frump')
```

In [20]:
```python
two_smallest = sorted(s_bigrams_probabilities.items(), key=itemgetter(1))[:2]

print(two_smallest)
```

```
[(('donald', 'frump'), -15.966117792178112), (('frump', 'lived'), -15.964921977038653)]
```

In [21]:
```python
incorrect_word = two_smallest[0][0][1]
# incorrect_word = two_smallest[1][0][0]

print(incorrect_word)
```

```
frump
```

In [22]:
```python
english_words = list(english_words_set)
```

In [23]:
```python
print('hello' in english_words)
print('trump' in english_words)
print('clinton' in english_words)
print('obama' in english_words)
print('jump' in english_words)
print('esoteric' in english_words)
```

```
True
True
False
False
True
True
```

In [24]:
```python
def minimum_edit_distance(word1, word2):
    # levenshtein distance minimum edit distance table
    amt_rows = len(word1) + 1
    amt_cols = len(word2) + 1
    edit_distance = np.zeros((amt_rows, amt_cols))

    for r in range(1, amt_rows):
        edit_distance[r][0] = r
    for c in range(1, amt_cols):
        edit_distance[0][c] = c

    for row in range(1, amt_rows):
        for col in range(1, amt_cols):
            if word1[row - 1] == word2[col - 1]:
                edit_distance[row][col] = edit_distance[row - 1][col - 1]
            else:
                edit_distance[row][col] = min(edit_distance[row][col - 1] + 1,
                                              edit_distance[row - 1][col] + 1,
                                              edit_distance[row - 1][col - 1] + 2)

    return edit_distance[amt_rows - 1][amt_cols - 1]
```

In [25]:
```python
print(minimum_edit_distance('apple', 'trample'))
```

```
4.0
```

In [26]:
```python
print(len(english_words))
```

```
25487
```

In [27]:
```python
distances = {}
for i in range(len(english_words) - 1):
    distances[english_words[i]] = minimum_edit_distance(english_words[i], incorrect_word)

twenty = sorted(distances.items(), key=itemgetter(1))[:20]

print(twenty[0:len(twenty)-1])
```

```
[('rump', 1.0), ('fum', 2.0), ('trump', 2.0), ('forum', 2.0), ('rum', 2.0), ('grump', 2.0), ('crump', 2.0), ('jump',
3.0), ('firm', 3.0), ('farm', 3.0), ('romp', 3.0), ('ramp', 3.0), ('grumpy', 3.0), ('drum', 3.0), ('rumpus', 3.0),
('from', 3.0), ('pump', 3.0), ('up', 3.0), ('bump', 3.0)]
```

In [28]:
```python
first_word = two_smallest[0][0][0]
new_probabilities = {}

for i in range(len(twenty)):
    bigram = (first_word, twenty[i][0])
    if bigram not in bigram_probabilities:
        calculate_unknown_bigrams_probability(bigram)
    new_probabilities[bigram] = bigram_probabilities[bigram]

largest = sorted(new_probabilities.items(), key=itemgetter(1))[-1:]

print(largest)
correct_word = largest[0][0][1]
print(correct_word)
```

```
[(('donald', 'trump'), -7.203158871411385)]
trump
```

In [29]:
```python
print("Original Sentence:", original_sentence)
print("Did you mean to replace", incorrect_word, "with", correct_word, "?")
response = input("")

final_sentence = original_sentence

if response == "yes":
    for word in original_sentence.translate(str.maketrans('', '', string.punctuation)).split():
        if word.lower() == incorrect_word:
            final_sentence = final_sentence.replace(word, correct_word)
            break
    print("Updated Sentence: ", final_sentence)
else:
    print("We could not find a suitable change for", incorrect_word)
```

```
Original Sentence: The former president, Donald Frump, lived in the White House.
Did you mean to replace frump with trump ?
yes
Updated Sentence:  The former president, Donald trump, lived in the White House.
```

In [30]:
```python
print('reporter' in english_words)
```

```
False
```

In [31]:
```python
# trie to make searching faster
```

In [32]:
```python
class Trie:
    def __init__(self):
        self.word = None
        self.children = {}
    def add(self, word):
        node = self
        for l in word:
            if l not in node.children: # only add letter if it isn't already there
                node.children[l] = Trie()
            node = node.children[l]
        node.word = word

def find(node, word, letter, previous, size, words):
    row = [previous[0] + 1]

    for column in range(1, len(word) + 1):
        delete = previous[column] + 1 # deletions cost 1
        insert = row[column - 1] + 1 # insertions cost 1
        replace = 0

        if word[column - 1] == letter:
            replace = previous[column - 1]
        else:
            replace = previous[column - 1] + 2 # substitutions cost 2
        row.append(min(delete, insert, replace))
    if min(row) <= size:
        for letter in node.children:
            find(node.children[letter], word, letter, row, size, words)
    if node.word != None and row[len(row) - 1] <= size:
        words.append((node.word, row[len(row) - 1]))

def search(word, size):
    row = range(len(word) + 1)
    words = []
    for l in trie.children:
        find(trie.children[l], word, l, row, size, words) # recursively search
    return words

start = time.time()
trie = Trie()
for w in english_words:
    trie.add(w)
words = search("frump", 3)
end = time.time()
print(words)
print("Search took %g seconds" % (end - start))
```

```
[('drum', 3), ('dump', 3), ('crump', 2), ('pump', 3), ('bump', 3), ('hump', 3), ('fume', 3), ('fum', 2), ('form', 3),
('forum', 2), ('farm', 3), ('firm', 3), ('from', 3), ('ramp', 3), ('rumpus', 3), ('rumple', 3), ('rump', 1), ('rum',
2), ('romp', 3), ('arum', 3), ('grumpy', 3), ('grump', 2), ('lump', 3), ('trump', 2), ('up', 3), ('jump', 3)]
Search took 1.46755 seconds
```