In [103]:
```
import numpy as np
import pandas as pd
import argparse
import math
import random
from tqdm import tqdm
import matplotlib.pyplot as plt
import os
import operator
from collections import defaultdict as dd

```

In [101]:
```
masked_txt = pd.read_csv (r'/Users/nehakardam/Documents/UWclasses /CSE
masked_txt.to_csv (r'/Users/nehakardam/Documents/UWclasses /CSE NLP/A6/
lm_txt = pd.read_csv (r'/Users/nehakardam/Documents/UWclasses /CSE NLP/
lm_txt.to_csv (r'/Users/nehakardam/Documents/UWclasses /CSE NLP/A6/data
```

```
<ipython-input-101-32cd5041a44b>:1: ParserWarning: Falling back to the 'p
ython' engine because the 'c' engine does not support regex separators (s
eparators > 1 char and different from '\s+' are interpreted as regex); yo
u can avoid this warning by specifying engine='python'.
  masked_txt = pd.read_csv (r'/Users/nehakardam/Documents/UWclasses /CSE
NLP/A6/data_A6/15pctmasked.txt',sep='delimiter', names = ["Sentences"])
<ipython-input-101-32cd5041a44b>:3: ParserWarning: Falling back to the 'p
ython' engine because the 'c' engine does not support regex separators (s
eparators > 1 char and different from '\s+' are interpreted as regex); yo
u can avoid this warning by specifying engine='python'.
  lm_txt = pd.read_csv (r'/Users/nehakardam/Documents/UWclasses /CSE NLP/
A6/data_A6/lm.txt', sep='delimiter', names = ["Words", "Prob"])
```

In [109]:
```
blm = BigramModel('/Users/nehakardam/Documents/UWclasses /CSE NLP/A6/da
v = Viterbi('/Users/nehakardam/Documents/UWclasses /CSE NLP/A6/data_A6/
            '/Users/nehakardam/Documents/UWclasses /CSE NLP/A6/data_A6/
complete_sentences = v.compute_missing_characters(blm)
v.write_sentences_to_file(complete_sentences)
```

```
Running Viterbi Algorithm: 100%|██████████| 5785/5785 [04:58<00:00, 19.36
it/s]
```

In [129]:
```
print(blm)
```

```
<__main__.BigramModel object at 0x7f87c8fbbf40>
```

In [116]:

```
START = '<start>'
MASK = '<mask>'
SPACE = '<s>'
EOS = '<eos>'

class BigramModel:
    def __init__(self, file_path):
        if not os.path.exists(file_path):
            print(f"Path: {file_path} does not exist")
            return None

        with open(file_path, "r") as f:
            bigrams = f.readlines()

        self.blm = dd(dict)

        for bigram in bigrams:
            bigram, prob = bigram.split('\t')[0], bigram.split('\t')[1

            w1, w2 = bigram.split(' ')[0], bigram.split(' ')[1]
            prob = float(prob.strip())

            self.blm[w1][w2] = prob

        self._create_labels_dict()

    def _create_labels_dict(self):
        labels = list(self.blm.keys())
        labels.sort()
        labels.append(labels.pop(labels.index(EOS))) # add EOS to end
        labels = [labels.pop(labels.index(START))] + labels # add STAR

        self.labels = labels
        self.label_to_idx = {k: v for v, k in enumerate(labels)}
        self.idx_to_label = {v: k for v, k in enumerate(labels)}

    def get_labels(self):
        return self.labels

    def get_labels_to_index(self):
        return self.label_to_idx

    def get_index_to_labels(self):
        return self.idx_to_label

    def get_bigram_prob(self, w1, w2) -> float:
        if w1 in self.blm.keys() and w2 in self.blm[w1].keys():
            return self.blm[w1][w2] # p(w2 | w1) = prob
        return 0

    def get_w2_given_w1(self, w1, w2, is_log_prob=True):
        if w1 in self.blm.keys() and w2 in self.blm[w1].keys():
            if is_log_prob: return np.log(self.blm[w1][w2]) # p(w2 | w
            else: return self.blm[w1][w2] # p(w2 | w1) = ln(prob)

        if is_log_prob: return float('-inf')
```

```
        return 0

    def get_max_from_key(self, w1) -> str:
        return max(self.blm[w1].items(), key=operator.itemgetter(1))

class Viterbi:
    def __init__(self, input_file_path, output_file_path):
        if not os.path.exists(input_file_path):
            print(f"Path: {input_file_path} does not exist")
            return None

        # Read and parse the input file
        with open(input_file_path, "r") as f:
            l = f.readlines()

        self.masked_sentences = []
        for sentence in l:
            sentence = sentence.split()
            self.masked_sentences.append(sentence)

        self.output_file_path = output_file_path

    def compute_missing_characters(self, blm: BigramModel):
        states = blm.get_labels()
        complete_sentences = []

        for sentence in tqdm(self.masked_sentences, desc="Running Vite
            best_path = self.viterbi_algorithm(sentence, states, blm)
            complete_sentences.append(best_path)

        return complete_sentences

    def viterbi_algorithm(self, observation, states, blm: BigramModel)

        sentence = observation[:]

        idx_to_state = blm.get_index_to_labels()
        state_to_idx = blm.get_labels_to_index()
        R, C = len(states), len(sentence)

        # To hold p. of each state given each sentence.
        trellis = np.full((R, C), -np.inf)

        # to hold the back pointers for cell
        back_pointer = np.zeros((R, C), dtype='int32')

        # Determine each hidden state's p. at time 0
        for i in range(R-1):
            if states[i] == START:
                trellis[i][0] = blm.get_w2_given_w1(START, START) # in

        # and now, assuming each state's most likely prior state, k
        for j in range(1, C-1):
            w1, w2 = sentence[j-1], sentence[j]

            for i in range(R-1):
                label = states[i]
```

```
114
115                 # Case 1: w1 and w2 are both known characters
116                 if w1 != MASK and w2 != MASK:
117                     label_idx = state_to_idx[w2]
118                     max_prev_trellis_value = max(trellis[:, j-1])
119                     max_prev_trellis_label_idx = np.argmax(trellis[:,
120
121                     trellis[label_idx][j] = max_prev_trellis_value + bl
122                     back_pointer[label_idx][j] = max_prev_trellis_label
123                     break
124
125                 # Case 2: curr is MASK and prev column is known
126                 elif w1 != MASK and w2 == MASK:
127
128                     max_prev_trellis_value = max(trellis[:, j-1])
129                     max_prev_trellis_label_idx = np.argmax(trellis[:,
130
131                     trellis[i][j] = max_prev_trellis_value + blm.get_w2
132                     back_pointer[i][j] = max_prev_trellis_label_idx
133
134                 # Case 3: curr is known and prev column is MASK
135                 elif w2 != MASK and w1 == MASK:
136                     label_idx = state_to_idx[w2]
137
138                     t1 = np.full((R-1, ), -np.inf)
139                     for k in range(R-1):
140                         prev_label = states[k]
141                         t1[k] = blm.get_w2_given_w1(w1=prev_label, w2=w
142
143                     trellis[label_idx][j] = max(t1)
144                     back_pointer[label_idx][j] = np.argmax(t1)
145                     break
146
147                 # Case 4: w1 and w2 are both MASK characters
148                 else:
149                     t1 = np.full((R-1, ), -np.inf)
150
151                     '''
152                     Since the prev column is also a mask, we pick the
153                     all the other lables as well. Then find the max
154                     '''
155                     for k in range(R-1):
156                         prev_label = states[k]
157                         t1[k] = blm.get_w2_given_w1(w1=prev_label, w2=
158                     trellis[i][j] = max(t1)
159                     back_pointer[i][j] = np.argmax(t1)
160
161         # Fill in the prob for <eos>
162         trellis[R-1][C-1] = blm.get_w2_given_w1(w1=EOS, w2=EOS)
163         back_pointer[R-1][C-1] = np.argmax(trellis[:, C-2])
164
165         # np.savetxt('back_pointer.out', np.vstack((['header'] + sente
166         # np.savetxt('trellis.out', np.vstack((['header'] + sentence,
167
168         # get the back pointers
169         guessed_sentence = [EOS]
170         label_idx = back_pointer[R-1][C-1]
```

```
171         for j in reversed(range(C-1)):
172             guessed_sentence.append(states[label_idx])
173             label_idx = back_pointer[label_idx][j]
174 
175         return list(reversed(guessed_sentence))
176 
177     def write_sentences_to_file(self, sentence_list):
178         sentence_list_strings = []
179 
180         for sentence in sentence_list:
181             s = ' '.join(sentence)
182             sentence_list_strings.append(s)
183 
184         s = '\n'.join(sentence_list_strings)
185 
186         with open(self.output_file_path, "w") as f:
187             f.write(s)
188 
189 def sanity_check_output(masked_sentences, un_masked_sentences):
190     print("Performing sanity check on output")
191 
192     # Ensure the first sentence matches the correct output:
193     correct_out = ['<start>', 'I', '<s>', 'p', 'e', '<s>', 'm', 'a', '
194     for correct_char, unmasked_char in zip(correct_out, un_masked_sent
195         if correct_char != unmasked_char:
196             print(f'Error! Viterbi Algorithm is incorrect for sentence
197             return
198 
199     for masked_sentence, unmasked_sentence in zip(masked_sentences, un_
200 
201         # Ensure the length of 2 sentence is the same (same number of 
202         lm, lum = len(masked_sentence), len(unmasked_sentence)
203         if lm != lum:
204             print(f"Error! The length of the sentences do not match")
205             print(f"Masked Sentence: {masked_sentence}")
206             print(f"Unmasked Sentence: {unmasked_sentence}")
207 
208         # Ensure we only changed the <mask> characters
209         for i in range(lm):
210             c_m, c_um = masked_sentence[i], unmasked_sentence[i]
211 
212             if c_m != MASK and (c_m != c_um):
213                 print(f"Error! Changed a known character")
214                 print(f"Changed {c_m} -> {c_um} at index: {i}")
215 
216             elif c_m == MASK and c_um == START:
217                 print(f"Changed a masked char to <start> token!")
218                 print(masked_sentence)
219                 print(unmasked_sentence)
220                 print("")
221 
222 def parse_output_file(output_file_path):
223     un_masked_sentences = []
224 
225     # Read in output file if it exists
226     if os.path.exists(output_file_path):
227         with open(output_file_path, "r") as f:
```

```
            l = f.readlines()
        for sentence in l:
            sentence = sentence.split()
            un_masked_sentences.append(sentence)

    return un_masked_sentences

# if __name__ == "__main__":
#     print("NLP - A4")

#     parser = argparse.ArgumentParser(description='Viterbi Algorithm'
#     parser.add_argument("-lm", "--lang-model", dest="lang_model_path
#     parser.add_argument("-ip", "--input-file", dest="input_file_path
#     parser.add_argument("-op", "--output-file", dest="output_file_pa
#     parser.add_argument("-t", "--sanity-check", dest="perform_sanity_
#     args = parser.parse_args()

#     blm = BigramModel(args.lang_model_path)
#     v = Viterbi(input_file_path=args.input_file_path, output_file_pa

#     complete_sentences = v.compute_missing_characters(blm)
#     v.write_sentences_to_file(complete_sentences)

#     if args.perform_sanity_check:
#         sanity_check_output(v.masked_sentences, parse_output_file(ar
```

In [ ]:

```
1
```