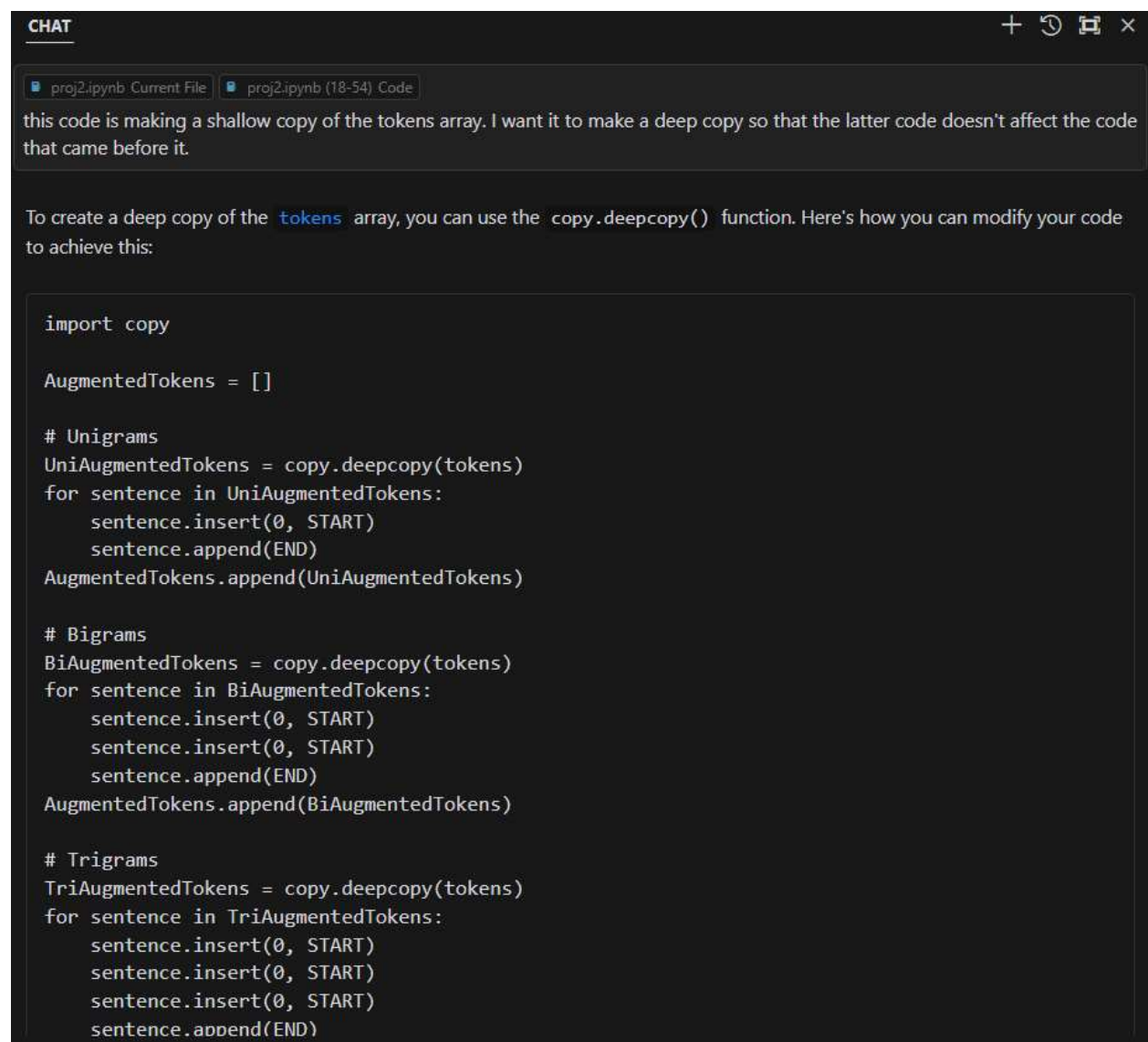


A.I. Disclaimer: Work for this assignment was completed with the aid of artificial intelligence tools and comprehensive documentation of the names of, input provided to, and output obtained from, these tools is included as part of my assignment submission.

Mainly I used AI for assisting me with parts of Python that I am unfamiliar with. I attached my usage below, including deep copy via list comprehension, better readability, refactoring my code to add a layer of dimensionality, and time complexity reduction. I did not use AI for developing my n-gram extraction or counting.

I spent multiple hours trying many different attempts of getting my lists of tokens and augmented sentence tokens to work, but they kept using a shallow copy, so the references were overwriting the previous lists. I tried googling for answers online but using `.copy()`, `[:]` and `list()` were not working. I didn't want to import `copy.deepcopy` because it would require importing a library. I used Clause 3.5-sonnet in Cursor to discover a very efficient usage of list comprehension! I'm not very great at Python so this was cool to see it in such a compact and understandable way. After this I went to Youtube to learn more about List Comprehension.

For example, printing square(i) of 1 to 10 would be `print([i * i for i in range(1,11)])`



The screenshot shows a chat interface in the Cursor IDE. At the top, there's a 'CHAT' header with icons for adding, undo, redo, and closing. Below the header, there are two tabs: 'proj2.ipynb Current File' and 'proj2.ipynb (18-54) Code'. The chat area contains a user message asking for a deep copy solution and an AI response providing a code snippet. The code snippet uses `copy.deepcopy()` to create deep copies of token arrays for unigrams, bigrams, and trigrams, and appends them to a main `AugmentedTokens` list.

```
import copy

AugmentedTokens = []

# Unigrams
UniAugmentedTokens = copy.deepcopy(tokens)
for sentence in UniAugmentedTokens:
    sentence.insert(0, START)
    sentence.append(END)
AugmentedTokens.append(UniAugmentedTokens)

# Bigrams
BiAugmentedTokens = copy.deepcopy(tokens)
for sentence in BiAugmentedTokens:
    sentence.insert(0, START)
    sentence.insert(0, START)
    sentence.append(END)
AugmentedTokens.append(BiAugmentedTokens)

# Trigrams
TriAugmentedTokens = copy.deepcopy(tokens)
for sentence in TriAugmentedTokens:
    sentence.insert(0, START)
    sentence.insert(0, START)
    sentence.insert(0, START)
    sentence.append(END)
```

This code makes the following changes:

1. Imports the `copy` module to use `deepcopy()`.
2. Uses `copy.deepcopy(tokens)` instead of `tokens.copy()` to create a deep copy of the `tokens` list.
3. Simplifies the loop structure by iterating directly over the sentences in each augmented token list.
4. Adds the correct number of START tokens for each n-gram level.
5. Uncomments and includes the trigram and quadgram processing.
6. Improves the final print loop for clarity.

These changes ensure that modifications to one set of augmented tokens won't affect the others, as each is now a deep copy of the original `tokens` list.



proj2.ipynb Current File

Can I do it without the `copy.deepcopy()`

Certainly! You can create a deep copy of the nested list structure without using `copy.deepcopy()`. Here's how you can modify your code to achieve this using list comprehension:

```
AugmentedTokens = []

# Unigrams
UniAugmentedTokens = [[START] + sentence + [END] for sentence in tokens]
AugmentedTokens.append(UniAugmentedTokens)

# Bigrams
BiAugmentedTokens = [[START, START] + sentence + [END] for sentence in tokens]
AugmentedTokens.append(BiAugmentedTokens)
```

```
# Trigrams
TriAugmentedTokens = [[START, START, START] + sentence + [END] for sentence in tokens]
AugmentedTokens.append(TriAugmentedTokens)

# Quadgrams
QuadAugmentedTokens = [[START, START, START, START] + sentence + [END] for sentence in tokens]
AugmentedTokens.append(QuadAugmentedTokens)

print()
for i, augmented_tokens in enumerate(AugmentedTokens):
    print(f"{i+1}-grams:", augmented_tokens)
```

Run as cell

This approach creates new lists for each n-gram level, effectively making a deep copy without using the `copy` module. The list comprehension `[START] + sentence + [END]` creates a new list for each sentence, adding the appropriate number of START tokens at the beginning and END token at the end.

This method is more concise and potentially more efficient, as it creates the new structures directly without first copying and then modifying.

From these multiple different lines, I was able to further improve them (by myself) by merging them all into a single line in a for loop:

```
for i in range(len(AugmentedTokens)):
    AugmentedTokens[i] = [[START]*(i+1) + sentence + [END] for sentence in tokens]
```

Later, I also used AI to compact some code, such as this:

```
if context != None and context2 != None and context3 != None and context != END and context2 != END and context3 != END:
    if context not in (None, END) and context2 not in (None, END) and context3 not in (None, END):
```

When I was transitioning into using both sentence and paragraph modes, I introduced another layer of array dimensionality, and since the code was becoming so complex it was difficult for me to locate which exact locations needed to be updated or to find parts that I accidentally missed. I had Claude 3.5-sonnet help me identify the locations to help make my sentence and paragraph levels work as I intended. I received some extra changes in these responses that I didn't want to use, I just used these responses to locate parts of my code that I missed in my transition to using the modes dimension.

Why did I get this error?

The error you're encountering is due to trying to access a key '<s>' (the start token) that doesn't exist in your dictionary. This is likely happening because you're trying to count the start token as a regular word in your unigrams, which it shouldn't be.

To fix this, you need to modify your code to skip the start and end tokens when counting unigrams. Here's how you can adjust the relevant part of your code:

Ask Copy Apply

```
if gram == 0: # Calculate Unigrams
    context = ()
    gramsMode[mode][gram][context] = {} # Declare the unigrams to be a
    for tokenList in AugmentedTokens[mode][gram]: #0 context words
        for word in tokenList:
            if word not in (START, END): # Skip start and end tokens
                if word not in gramsMode[mode][gram][context]:
```


This change ensures that you're not trying to count the start and end tokens as regular words in your unigrams. The start and end tokens are useful for higher-order n-grams (bigrams, trigrams, etc.), but for unigrams, we typically don't include them in the counts.

Make sure to apply this change to both the sentence-level and paragraph-level processing in your code. This should resolve the `KeyError` you're encountering.



proj2.ipynb Current File

```
{
  "name": "KeyError",
  "message": "(' <s>')",
  "stack": "-----
KeyError                                Traceback (most recent call last)
Cell In[167], line 57
      55 if context not in (None, END):
      56 \tbigramContext = (context,) # bigram dictionary key
--> 57 \tincrementWordCount(mode, gram, bigramContext, word)
      58 context = word
      59 contextCountMode[mode][gram] += 1

Cell In[167], line 32, in incrementWordCount(mode, gramIndex, context, word)
      30 if context not in gramsSen[gramIndex]: \t\t# if the context isn't in the gram dict,
      31 \tgramsMode[mode][gramIndex][context] = {} \t\t# create an empty dictionary
--> 32 if word not in gramsMode[mode][gramIndex][context]: \t# check if word is
already found in context
      33 \tgramsMode[mode][gramIndex][context][word] = 1 \t# Initialize count as 1
      34 else:

KeyError: (' <s>')
}
```

I apologize for the confusion earlier. The error is now occurring in the bigram calculation, specifically when trying to use '<s>' (start token) as part of the context. Let's modify the `incrementWordCount` function and the bigram calculation to handle this case. Here's the updated code:

proj2.ipynb

Ask

Copy

Apply

```
# Helper function for repeating code
def incrementWordCount(mode, gramIndex, context, word):
    if START in context: # Skip contexts that include the start token
        return
    if context not in gramsMode[mode][gramIndex]:
        gramsMode[mode][gramIndex][context] = {}
    if word not in gramsMode[mode][gramIndex][context]:
        gramsMode[mode][gramIndex][context][word] = 1
    else:
        gramsMode[mode][gramIndex][context][word] += 1

# ... (keep your existing code for unigrams)

if gram == 1: # Calculate Bigrams
    context = None
    for tokenList in AugmentedTokens[mode][gram]: #1 context word
        for word in tokenList:
            if context not in (None, START, END) and word != END:
                bigramContext = (context,) # bigram dictionary key
                incrementWordCount(mode, gram, bigramContext, word)
            context = word
            if word not in (START, END):
                contextCountMode[mode][gram] += 1
```

Run as cell

These changes do the following:

1. In the `incrementWordCount` function, we now skip any context that includes the start token. This prevents trying to create or access dictionary entries with the start token as a key.

2. In the bigram calculation:

- We now also check if the context is not START before creating a bigram.
- We only increment the word count if the current word is not START or END.
- We skip creating bigrams where the second word is END.

These modifications should prevent the `KeyError` you're encountering by ensuring that we don't try to use START or END tokens in ways that would cause issues in your data structures.

Make sure to apply similar logic to your trigram and quadgram calculations as well, adjusting the conditions as needed for the increased context size.



proj2.ipynb Current File

I don't think that is correct, I need to count how many START tokens I have. Please try again for a different solution

I apologize for the misunderstanding. You're right, we do need to count the START tokens for proper n-gram modeling. Let's approach this differently. The error suggests that we're trying to access a context that doesn't exist yet. We need to ensure that we create the context before trying to increment the word count.

Here's a revised approach:

Here's a revised approach:

proj2.ipynb

Ask

Copy

Apply

```
# Helper function for repeating code
def incrementWordCount(mode, gramIndex, context, word):
    if context not in gramsMode[mode][gramIndex]:
        gramsMode[mode][gramIndex][context] = {}
    if word not in gramsMode[mode][gramIndex][context]:
        gramsMode[mode][gramIndex][context][word] = 1
    else:
        gramsMode[mode][gramIndex][context][word] += 1

# ... (keep your existing code for unigrams)

if gram == 1: # Calculate Bigrams
    for tokenList in AugmentedTokens[mode][gram]:
        for i in range(len(tokenList) - 1): # We go up to the second-t
            context = (tokenList[i],)
            word = tokenList[i+1]
            incrementWordCount(mode, gram, context, word)
            contextCountMode[mode][gram] += 1

# Similar changes for trigrams and quadgrams
if gram == 2: # Calculate Trigrams
    for tokenList in AugmentedTokens[mode][gram]:
        for i in range(len(tokenList) - 2):
            context = (tokenList[i], tokenList[i+1])
            word = tokenList[i+2]
            incrementWordCount(mode, gram, context, word)
            contextCountMode[mode][gram] += 1

if gram == 3: # Calculate Quadgrams
    for tokenList in AugmentedTokens[mode][gram]:
        for i in range(len(tokenList) - 3):
            context = (tokenList[i], tokenList[i+1], tokenList[i+2])
            word = tokenList[i+3]
            incrementWordCount(mode, gram, context, word)
            contextCountMode[mode][gram] += 1
```

This approach:

1. Keeps the `incrementWordCount` function as is, which will create new contexts as needed.
2. For bigrams, trigrams, and quadgrams, we iterate through the token list, creating appropriate contexts and incrementing counts.
3. We don't skip any tokens, including START and END, ensuring all are counted properly.

This should resolve the `KeyError` by ensuring that all contexts are created before we try to access them, while also correctly counting all tokens including START and END.

Remember to adjust the range in the for loops based on the number of context words needed for each n-gram type. This approach should work for both sentence-level and paragraph-level processing.



proj2.ipynb Current File

Is there anything obvious I'm missing? This was working before I switched to using the `modes[]` when I was only doing sentence level. Now I added another dimension to the entire loop so I can do paragraph mode and now I get this error.

```
KeyError                                Traceback (most recent call last)
Cell In[176], line 57
     55 if context not in (None, END):
     56     bigramContext = (context,) # bigram dictionary key
--> 57     incrementWordCount(mode, gram, bigramContext, word)
     58     context = word
     59     contextCountMode[mode][gram] += 1
```

```
Cell In[176], line 32
     30 if context not in gramsSen[gramIndex]:                                # if the context isn't
```


I apologize for the confusion. You're right, the error is likely related to the transition to using modes. Let's take a closer look at your code and make some adjustments:

proj2.ipynb

Ask

Copy

Apply

```
# Helper function for repeating code
def incrementWordCount(mode, gramIndex, context, word):
    if context not in gramsMode[mode][gramIndex]:
        gramsMode[mode][gramIndex][context] = {}
    if word not in gramsMode[mode][gramIndex][context]:
        gramsMode[mode][gramIndex][context][word] = 1
    else:
        gramsMode[mode][gramIndex][context][word] += 1

for mode in range(2): # Sentence then Paragraph Level
    for gram in range(4): # 4 gram types
        if gram == 0: # Calculate Unigrams
            context = ()
            gramsMode[mode][gram][context] = {}
            for tokenList in AugmentedTokens[mode][gram]:
                for word in tokenList:
                    incrementWordCount(mode, gram, context, word)
                    contextCountMode[mode][gram] += 1

        if gram == 1: # Calculate Bigrams
            for tokenList in AugmentedTokens[mode][gram]:
                for i in range(len(tokenList) - 1):
                    context = (tokenList[i],)
                    word = tokenList[i+1]
                    incrementWordCount(mode, gram, context, word)
                    contextCountMode[mode][gram] += 1

        # Similar changes for trigrams and quadgrams
        if gram == 2: # Calculate Trigrams
            for tokenList in AugmentedTokens[mode][gram]:
                for i in range(len(tokenList) - 2):
                    context = (tokenList[i], tokenList[i+1])
```

Key changes:

1. We're now using `gramsMode[mode][gramIndex]` consistently instead of `gramsSen[gramIndex]`.
2. We've simplified the logic for each n-gram type, iterating through the tokens directly.
3. We're creating the context tuples explicitly for each n-gram type.
4. We're not skipping any tokens, including START and END.

This approach should handle both sentence and paragraph modes correctly, and it should avoid the `KeyError` you're encountering. The error was likely caused by inconsistent use of `gramsMode` vs `gramsSen`, or by trying to access contexts that weren't properly initialized.

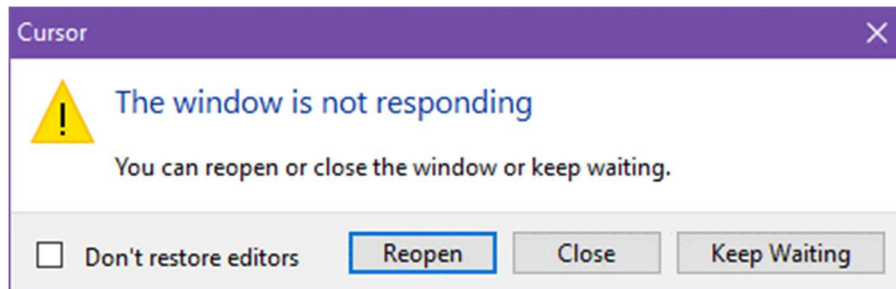
If you're still encountering issues, it might be helpful to add some debug print statements to see exactly what contexts and words are being processed when the error occurs. For example:

```
def incrementWordCount(mode, gramIndex, context, word):  
    print(f"Debug: mode={mode}, gramIndex={gramIndex}, context={context}  
    # ... rest of the function
```

Run as cell

This will help pinpoint exactly where the error is occurring and what data is being processed at that point.

Given that my code was taking 8 minutes to calculate the probabilities I investigated my time complexity of the main probability loop. It was also not responding and crashed.



```
def calcContextTotal(grams, gram):
    contextTotal = 0
    for word in grams[gram]:
        contextTotal += grams[gram][word]
    return contextTotal
```

```
for mode in range(len(modes)):
    if mode == 0:
        print("Sentence level:")
    elif mode == 1:
        print("\nParagraph level:")

    for ngram in range(len(gramsMode[mode])):
        print(f"{gramsPrintStrings[ngram]} probability table")
        for ctx in gramsMode[mode][ngram]:
            for word in gramsMode[mode][ngram][ctx]:
                prob = calcGramProb(mode, ngram, ctx, word)
                probModeGram[mode][ngram].append(prob)
                if debug:
                    occurrences = str(gramsMode[mode][ngram][ctx][word])
                    contextTotal = calcContextTotal(gramsMode[mode][ngram], ctx)
                    print(f"\tWord: {word:<12} \t Occurances: {occurrences:<3} \t Context total: {contextTotal}")
            if debug: print()
```

✓ 8m 8.1s

To which Cursor-small responded that I should precompute my context totals rather than summing every single time, which I didn't even think about before. I used most of the changes it suggested as they were greatly beneficial for the time reduction, but I did have to modify the results to properly work with my codebase.

To improve the time complexity of the provided code, we can optimize the `calcContextTotal` function and the way probabilities are calculated. Here are some suggestions:

1. **Precompute Context Totals:** Instead of calculating the context total every time you need it, compute it once and store it in a dictionary.
2. **Avoid Repeated Dictionary Lookups:** Store references to frequently accessed dictionary items to reduce lookup time.

Here's a simplified version of the relevant code with these optimizations:

```
c:\Users\drew1\Documents\Classes\NLP\Projects\Project2\proj2.ipynb
debug = False

# Precompute context totals
contextTotals = [{}, {}] # [sentence context totals, paragraph context totals]

def calcContextTotal(grams, gram):
    if gram in contextTotals[0]: # Check if already computed
        return contextTotals[0][gram]
    contextTotal = sum(grams[gram].values())
    contextTotals[0][gram] = contextTotal # Store the computed total
    return contextTotal
```

```

for mode in range(len(modes)):
    if mode == 0:
        print("Sentence level:")
    elif mode == 1:
        print("\nParagraph level:")

    for ngram in range(len(gramsMode[mode])):
        print(f"{gramsPrintStrings[ngram]} probability table")
        for ctx in gramsMode[mode][ngram]:
            word_dict = gramsMode[mode][ngram][ctx] # Store reference to avoid
            contextTotal = calcContextTotal(gramsMode[mode][ngram], ctx) # Prec
            for word in word_dict:
                prob = word_dict[word] / contextTotal # Use precomputed context
                probModeGram[mode][ngram].append(prob)
            if debug:
                occurrences = str(word_dict[word])
                print(f"\tWord: {word:<12} \t Occurances: {occurrences:<3} \t")
        if debug: print()

```

Run as cell

Explanation of Changes:

- **Precomputation:** The context totals are computed once and stored in a dictionary, reducing redundant calculations.
- **Reduced Lookups:** By storing the reference to the word dictionary, we avoid repeated dictionary lookups, which can be costly.

These changes should help lower the time complexity, especially if the number of n-grams and contexts is large.

Which effectively reduced the time to less than half a minute!

✓ 21.7s