CS5800 Final Project

Fall 2023

So Man Amanda Au-Yeung, Wenqiao Xu, Yian Chen

Introduction

To recall, in this project, we need to map categories between 2 files. The first file, Avalara_goods_and_services.xlsx, is a well-structured file of goods and services with somewhat uniform fields; whereas the second file, UNSPSC_English.csv is hierarchical, but the fields are messy and non-uniform.

Our task is to map categories from the reference file: Avalara_goods_and_services.xlsx to the file UNSPSC_English.csv, clean and process data, design matching algorithms and report the number of valid matches and corresponding matching percentages.

In our “Improved Brute Force” iteration, our matching percentage being roughly 2691 matches/2000 rows(test data)* 8 columns * 8 words(an approximate average number of words in each field), which is about 2%). In our trie algorithm, our matching percentage is around 90%, which is promising.

- ## Data processing
  - Converted both files to “xlsx” for processing
  - Replace NaN values in the UNSPSC file with empty strings

```
#replace nan values with an empty string
df_un["Family Title"] = df_un["Family Title"].fillna('')
df_un["Family Definition"] = df_un["Family Definition"].fillna('')
df_un["Class Title"] = df_un["Class Title"].fillna('')
df_un["Class Definition"] = df_un["Class Definition"].fillna('')
df_un["Commodity Title"] = df_un["Commodity Title"].fillna('')
df_un["Definition"] = df_un["Definition"].fillna('')
```

  - Write a text-preprocessing function using NLP libraries and regular expressions to filter out unwanted punctuation, stopwords, singularize plural words, and stem the words to keep prefixes of words. For example, the word “electronically” should be matched with “electronic” or “electrical”, so our stemming function keeps the main prefixes of words to help us find more matches in our data.

```
[55]
from nltk.corpus.reader import wordlist
def text_preprocess(text):
    text = text.encode("ascii","ignore").decode()
    # get rid of punctuations and unwanted characters (keeping hyphens and forward slash)
    r = re.compile(r"[^\w\s\-\/]+")
    res = []
    res = [r.sub("",s) for s in text]

    for i in range(len(res)):
        res[i] = res[i].lower()
        res[i] = re.sub(r"\w*\d+\w*","", res[i])
        res[i] = re.sub(r"\-"," ",res[i])
        res[i] = re.sub(r"\/"," ",res[i])
    res_word = "".join(res).split()

    # making plural words singular and removing stop words
    new_res = set()
    for word in res_word:
        if word not in stopwords_default:
            singular_word = p.singular_noun(word)
            if singular_word:
                new_res.add(singular_word)
            else:
                new_res.add(word)

    word_list = list(new_res)
    #stem the words
    porterStemming = PorterStemmer()
    stemWords = [porterStemming.stem(word) for word in word_list]

    return stemWords
```

- Filtering out stopwords:
  - Here is an example of on a sentence
  - NLTK library has a default list of stopwords.

```
stopwords_default = stopwords.words("english")
example = "Computer software that is primarily designed for something other than academic educational purposes that is transferre
```

- We may customize the stopwords by appending new words in it or a list of customized words where we observe as unnecessary in the Avalara data or UNSPSC data.
- Here is an example of the sentence when we filter out the stopwords:

```python
res = []
example_text = example.split(" ")
print(example_text)
for word in example_text:
  if word not in stopwords_default:
    res.append(word)
print(res)
```

```
['Computer', 'software', 'that', 'is', 'primarily', 'designed', 'for', 'something', 'other', 'than', 'academic', 'educational', 'purposes', 'that', 'is', 'transferre
['Computer', 'software', 'primarily', 'designed', 'something', 'academic', 'educational', 'purposes', 'transferred', 'electronically', 'customized.']
```

- For the Improved Brute Force: Tree, Non-Stop and Stop algorithm, added new columns which broke sentences into words.

```python
#add a column which is a list of words (part one)
list_seg_title = df_un["Segment Title"].apply(lambda x: text_preprocess(x)).values.tolist()
list_seg_def = df_un["Segment Definition"].apply(lambda x: text_preprocess(x)).values.tolist()
list_fam_title = df_un["Family Title"].apply(lambda x: text_preprocess(x)).values.tolist()
list_fam_def = df_un["Family Definition"].apply(lambda x: text_preprocess(x)).values.tolist()
list_class_def = df_un["Class Definition"].apply(lambda x: text_preprocess(x)).values.tolist()
list_class_title = df_un["Class Title"].apply(lambda x: text_preprocess(x)).values.tolist()
list_comm_title = df_un["Commodity Title"].apply(lambda x: text_preprocess(x)).values.tolist()
list_comm_def = df_un["Definition"].apply(lambda x: text_preprocess(x)).values.tolist()
```

```python
#add a column which is a list of words (part two)
df_un["list_seg_title"] = list_seg_title
df_un["list_seg_def"] = list_seg_def
df_un["list_fam_title"] = list_fam_title
df_un["list_fam_def"] = list_fam_def
df_un["list_comm_title"] = list_comm_title
df_un["list_comm_def"] = list_comm_def
df_un["list_class_title"] = list_class_title
df_un["list_class_def"] = list_class_def
```

```python
df_un
```

Segment Title	Segment Definition	Family	Family Title	Family Definition	Column1	Class	...	Synonym	Acronym	list_seg_title	list_seg_def	list_fam_title	list_fam_def	list_comm_title	list_comm_def	list_class_title	list_class_def
Live Plant and Animal Material and Accessories...	This segment includes live, wild and domestica...	NaN			NaN	NaN	...	活植物、活动物及其附件和用品, 活植物、活動物及其附件和用品, Levende og ani...	NaN	[supply, live, accessory, animal, material, pl...	[segment, include, used, equipment, live, mate...	[]	[]	[]	[]	[]	[]
Live Plant and Animal Material and Accessories...	This segment includes live, wild and domestica...	10100000	Live animals		NaN	NaN	...	活动物, 活動物, Levende dyr, Levende dieren, Animaux...	NaN	[supply, live, accessory, animal, material, pl...	[segment, include, used, equipment, live, mate...	[live, animal]	[]	[]	[]	[]	[]
Live Plant and Animal Material and Accessories...	This segment includes live, wild and domestica...	10100000	Live animals		NaN	10101500	...	家畜类, 家畜類, Besætning, Vee, Bétail, Viehbestand,...	NaN	[supply, live, accessory, animal, material, pl...	[segment, include, used, equipment, live, mate...	[live, animal]	[]	[]	[]	[livestock]	[]
Live Plant and Animal Material and Accessories...	This segment includes live, wild and domestica...	10100000	Live animals		NaN	10101500	...	猫, 貓, Katte, Katten, Chats, Katzen, Macskák, G...	NaN	[supply, live, accessory, animal, material, pl...	[segment, include, used, equipment, live, mate...	[live, animal]	[]	[cat]	[]	[livestock]	[]
Live Plant and Animal Material and Accessories...	This segment includes live, wild and domestica...	10100000	Live animals		NaN	10101500	...	狗, 狗, Hunde, Honden, Chiens, Hunden, Kutyák, C...	NaN	[supply, live, accessory, animal, material, pl...	[segment, include, used, equipment, live, mate...	[live, animal]	[]	[dog]	[]	[livestock]	[]
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Drugs and Pharmaceutical Products	This segment includes natural or synthetic	51170000	Drugs affecting the gastrointestinal system	This classification denotes drugs that act on ...	NaN	51171800	...	NaN	NaN	[pharmaceutical, product, drug]	[segment, compound, include, used, disease, sy...	[affecting, gastrointestinal, system, drug]	[include, structure, digestive, denote, drug, ...	[]	[]	[antivertigo, antiemetic, agent, antinauseant]	[medullary, nausea, center, receptor, peripher...

- For the XXX algorithm, combined the following columns in the UNSPSC file, then we converted the following to a list data type for data preprocessing.
    - Segment Title, Segment Definition
    - Family Title, Family Definition
    - Class Title, Class Definition
    - Commodity Title, Definition

- # Python libraries reference
  - Pandas: https://pandas.pydata.org/docs/
    - Goal: manipulate our data so our algorithm can be easily implemented
  - NLTK: https://www.nltk.org/
    - Goal: utilize NLTK library to filter words for matching values and for stemming words to get word prefixes
  - Inflect: https://pypi.org/project/inflect/
    - Goal: Generate plurals, singular, ordinals, and convert numbers to words.
  - NumPy: https://numpy.org/
    - Goal: Support data manipulation of both data
  - Tqdm: https://pypi.org/project/tqdm/
    - Goal: Create Progress Meters or Progress Bars.
  - Sklearn: https://pypi.org/project/sklearn/
    - Goal: implement TF-IDF vector.
  - AnyTree: https://anytree.readthedocs.io/en/latest/index.html
    - Goal: To easily create a tree for searching purposes.

- # Algorithms
  - ## Brute Force: Modified Longest Common Subsequence

```
[ ] def lcs(data1, data2):
      arr = [[None] * len(data1 + 1) for i in range(len(data2 + 1))]
      for i in range(len(data1) + 1):
        for j in range(len(data2) + 1):
          if i == 0 or j == 0:
            arr[i][j] = 0
          elif data1[i - 1] == data2[j - 1]:
            arr[i][j] = arr[i - 1][j - 1] + 1
          else:
            arr[i][j] = max(arr[i - 1][j], arr[i][j - 1])
      return arr[len(data1)][len(data2)]
```

    - Algorithm: Instead of finding the longest subsequence of strings, find the words that occur the most frequently in the target arrays. To speed up the string matching process, we cleaned the UNSPSC data similar to that of a trie structure.

    - Runtime analysis: It will run all columns of the Avalara files against UNSPSC to find matches in strings, however the run time would be

O(avalara data columns*avalara data rows)*O(UNSPSC columns* UNSPSC rows), which would be way too inefficient.

- Improved Brute Force: Tree, non-stop and stop
  - Ideas: as mentioned in the project requirement and out of our own inspection, the UNSPSC data has a form that is close to a 4-level tree. The initial thought was to take advantage of this and to make the tree structured file ready to use. However, later exploration reveals that lower levels do not necessarily have more info than the upper levels.
  - Algorithm: reused the data processing from the last algorithm, try top-down and bottom-up order of traversal of the entire file and set no stop conditions. This way, the algorithm will scan through the entire Alvara file and UNSPSC file to find matches. This is the “baseline tree” shown below. After discussion with Prof.Lama, we decided to add a non-stop version of the algorithm, which will stop iterating through the current row if it does not find any matches at the current cell, i.e. it does not go to deeper levels of the tree.
  - Results: results are displayed as below, we could see that the results of two order(top-down and bottom-up) for non-stop algorithm are the same, and that the number of matches the algorithm that will stop after no-match at current level based on a top-down order found is way lower than that of non-stop’s, the matching percentage being roughly 133 matches/65526 rows* 25 columns(matching percentage lower than 1%), and the top down stop algorithm finds no matches for the test slice of data.

```
[21] def compare(list1,list2,threshold):
       #list1 is from df, list2 is from df_un
       #result_flag is binary value indicating whether list2 has over %threshold of words in list1
       result_flag = False
       count = 0
       if not list1 or not list2:
         return [result_flag,0]
       for word1 in list1:
         for word2 in list2:
           if word1 == word2:
             count += 1
       if count//len(list1) > threshold:
           result_flag = True
       return [result_flag, count//len(list1)]
```

```
test_df = df[:200]
test_df_un = df_un[:5000]
```

```python
#baseline tree
from prompt_toolkit.shortcuts.progress_bar.formatters import D
#count number of matches in two columns, each in one file
from tqdm import tqdm
count = 0
#follow the order of commodity->class->family->segment columns
lists = ["list_comm_title","list_comm_def","list_class_title","list_class_def"\
         ,"list_fam_title","list_fam_def","list_seg_title","list_seg_def"][::-1]
for df_list in tqdm(test_df["list_tax_code_des"]):
  #iterate through row number
  for df_un_row in range(len(test_df_un)):
    #iterate through column lables
    for df_un_col in lists:
      current_cell = test_df_un.iloc[df_un_row][df_un_col]
      if compare(df_list,current_cell,threshold=0.5)[0]:
        count += 1
print(count)
```

```
100%|██████████| 200/200 [12:11<00:00,  3.66s/it]2691
```

```python
#baseline tree
from prompt_toolkit.shortcuts.progress_bar.formatters import D
#count number of matches in two columns, each in one file
from tqdm import tqdm
count = 0
#follow the order of commodity->class->family->segment columns
lists = ["list_comm_title","list_comm_def","list_class_title","list_class_def"\
         ,"list_fam_title","list_fam_def","list_seg_title","list_seg_def"]
for df_list in tqdm(test_df["list_tax_code_des"]):
  #iterate through row number
  for df_un_row in range(len(test_df_un)):
    #iterate through column lables
    for df_un_col in lists:
      current_cell = test_df_un.iloc[df_un_row][df_un_col]
      if compare(df_list,current_cell,threshold=0.5)[0]:
        count += 1
print(count)
```

```
100%|██████████| 200/200 [12:15<00:00,  3.68s/it]2691
```

```
[28] #a tree that will stop at levels
     from prompt_toolkit.shortcuts.progress_bar.formatters import D
     #count number of matches in two columns, each in one file
     from tqdm import tqdm
     count = 0
     # break_flag = False
     lists = ["list_comm_title","list_comm_def","list_class_title","list_class_def"\
              ,"list_fam_title","list_fam_def","list_seg_title","list_seg_def"][::-1]
     for df_list in tqdm(test_df["list_tax_code_des"]):
       #iterate through row number
       for df_un_row in range(len(test_df_un)):
         #iterate through column lables
         for df_un_col in lists:
           current_cell = test_df_un.iloc[df_un_row][df_un_col]
           #if not a single word match, we choose not to look at the remaining levels
           ismatch, match_percent = compare(df_list,current_cell,threshold=0.5)[0], compare(df_list,current_cell,threshold=0.5)[1]
           if match_percent == 0:
             break
           #if targets match according to our algo that has a threshold parameter, increae match count by one
           elif ismatch:
             count += 1
         # if break_flag:
         #   break_flag = False
         #   break
     print(count)

     100%|██████████| 200/200 [01:41<00:00,  1.98it/s]0
```

```
#a tree that will stop at levels
from prompt_toolkit.shortcuts.progress_bar.formatters import D
#count number of matches in two columns, each in one file
from tqdm import tqdm
count = 0
# break_flag = False
lists = ["list_comm_title","list_comm_def","list_class_title","list_class_def"\
         ,"list_fam_title","list_fam_def","list_seg_title","list_seg_def"]
for df_list in tqdm(test_df["list_tax_code_des"]):
  #iterate through row number
  for df_un_row in range(len(test_df_un)):
    #iterate through column lables
    for df_un_col in lists:
      current_cell = test_df_un.iloc[df_un_row][df_un_col]
      #if not a single word match, we choose not to look at the remaining levels
      ismatch, match_percent = compare(df_list,current_cell,threshold=0.5)[0], compare(df_list,current_cell,threshold=0.5)[1]
      if match_percent == 0:
        break
      #if targets match according to our algo that has a threshold parameter, increae match count by one
      elif ismatch:
        count += 1
    # if break_flag:
    #   break_flag = False
    #   break
print(count)

100%|██████████| 200/200 [01:37<00:00,  2.06it/s]113
```

- Runtime: stop algo is way faster than non-stop algo because it saves unnecessary effort in iterating through the entire database, which is actually the brute force approach, and the former one is an improvement of the latter one.

Using AnyTree to build a tree

- Idea: Using a hashmap to map each UNSPSC code to each combined column as mentioned before, then build a tree for iteration as one of the possible implementations.

```
[133] from anytree import Node, RenderTree, PreOrderIter
```

```
root = Node("")
seg = Node("Segment", parent=root)
family = Node("Family", parent=seg)
Class = Node("Class", parent=family)
Commodity = Node("Commodity", parent=Class)
print(RenderTree(root))
```

```
Node('/')
└── Node('//Segment')
    └── Node('//Segment/Family')
        └── Node('//Segment/Family/Class')
            └── Node('//Segment/Family/Class/Commodity')
```

```
[131] from collections import defaultdict
from pprint import pprint # allows printing large data with a limit of 5000 lines
Segment = {}
Family = {}
Class = {}
Commodity = {}
for index, row in df_un.iterrows():
  Segment[row["Segment"]] = row["list_seg"]
  Family[row["Family"]] = row["list_fam"]
  Class[row["Class"]] = row["list_class"]
  Commodity[row["Commodity"]] = row["list_comm"]
# print(Segment)
# print(Family)
# print(Class)
# pprint(Commodity)
root = Node("")
seg_sub_tree = Node(Segment, parent=root)
family_sub_tree = Node(Family, parent=seg_sub_tree)
class_sub_tree = Node(Class, parent=family_sub_tree)
print(RenderTree(root))
commodity_sub_tree = Node(Commodity, parent=class_sub_tree)
# pprint(RenderTree(root))
```

```
Node('/')
└── Node("//{50000000: ['miner', 'food', 'flavor', 'segment', 'tobacco', 'includ', 'product', 'condiment', 'co
    └── Node("//{50000000: ['miner', 'food', 'flavor', 'segment', 'tobacco', 'includ', 'product', 'condiment',
        └── Node("//{50000000: ['miner', 'food', 'flavor', 'segment', 'tobacco', 'includ', 'product', 'condime
```

- Levenshtein ratio, with and without
  - Algorithm:

    Compare two strings and find their ration based on how many edits are needed to change from one string to another. 1.0 ratio means an exact match. We tested our matching algorithm that included string matches with Levenshtein ratio of 0.80 and above. Also compared it with results using exact matches.

- ```
  [286] # Calculates levenshtein distance between two strings
  def levenshtein_ratio(string1,string2,ratio_calc = False):
      #initialize matrix of zeroes
      rows = len(string1)+1
      cols = len(string2)+1
      distance = np.zeros((rows,cols),dtype=int)

      #populate matrix of zeroes with indices of each characters of both strings
      for i in range(1,rows):
          for j in range(1,cols):
              distance[i][0] = i
              distance[0][j] = j

       # Iterate over the matrix to compute the cost of deletions,insertions and/or substitutions
      for col in range(1,cols):
          for row in range(1,rows):
              if string1[row-1] == string2[col-1]:
                  cost = 0 #if the characters are same in the two strings then cost=0
              else:
                  if ratio_calc == True:
                      cost = 2
                  else:
                      cost = 1
              distance[row][col] = min(distance[row-1][col]+1,distance[row][col-1]+1,distance[row-1][col-1]+cost)

      #if ratio_cal is true then compute the levenshtein distance ratio of similarity between the two strings
      #if false, show how many edits needed to change string1 to string2 or vice versa
      if ratio_calc == True:
          ratio = ((len(string1)+len(string2))-distance[row][col])/(len(string1)+len(string2))
          return ratio
      else:
          return f"the strings are {distance[row][col]} edits away"
  ```
- Runtime: Assuming that Levenshtein algorithm has an overall slower run time while performing against two different lists. However, it can increase accuracy.
- Conclusion: If we preprocessed text enough (used stemming to get prefix words), applying the Levenshtein algorithm won't necessarily be faster.

Term Frequency-Inverse Document Frequency(TF-IDF) and Similarity Scores

- Algorithm:
  - Generate a corpus, which is an empty array
  - Put the string values in the two informative fields in the Avalara file, which are "AvaTax System Tax Code Description" and "Additional AvaTax System Tax Code Information", in the array
  - Put the string values in the eight fields(four levels, Segment, Family, Class, Commodity, each with its title and definition fields) in the UNSPSC file, in the array
  - Now the length of the corpus is 2 fields * 2543 rows + 8 fields * 65516 rows, which is 525214.
  - Generate the 525214 * 525214 tf-idf vector using

```python
from sklearn.feature_extraction.text import TfidfVectorizer

# Create TfidfVectorizer object
vectorizer = TfidfVectorizer()

# Generate matrix of word vectors
tfidf_matrix = vectorizer.fit_transform(ted)
```
```

And it should look like

Similarity Matrix

	String 1 from Avalara	String 2 from Avalara	String N from Avalara	String 5086 from Avalara	String 1 from UNSPSC	String 2 from UNSPSC	String N from UNSPSC	String 524128 from UNSPSC
String 1 from Avalara	1	0.21704584	0.18314713	0.18435251	0.11203887	0.15704584	0.22437219	0.18437219
String 2 from Avalara	0.21704584	1	...	...	0.15203687	0.25423219	...	...
String N from Avalara	0.18314713	...	...	...	...	0.22437219	...	...
String 5086 from Avalara	0.18435251	...	...	...	...	...	...	...
String 1 from UNSPSC	0.11203887	0.15203687	...	...	1	...	...	...
String 2 from UNSPSC	0.15704584	0.25423219	...	...	...	1	...	...
String N from UNSPSC	0.22437219	0.22437219	...	...	...	...	1	...
String 524128 from UNSPSC	0.18437219	...	...	...	...	...	...	1

- Set a threshold for similarity scores, for example, 0.5
- Store the strings from Avalara file whose similarity score is larger than the threshold in a result array

■ Pros and Cons

- Pros
  - Easy to implement, using the library and most work is in dealing with the similarity matrix
  - No need for data cleaning beforehand
  - No need for breaking sentences into words, thus more precise in finding matches
- Cons
  - It computes document similarity directly in the word-count space, which may be slow for large vocabularies.
  - It assumes that the counts of different words provide independent evidence of similarity.
  - It makes no use of semantic similarities between words.[1]

○ Top Down: Trie

■ Algorithm: to speed up the matching process, we tried turning the UNSPSC data into a Trie data structure. Each "Trie Node" will consist of the word list of the UNSPSC data according to its levels (Segment, Family, Class, Commodity), in that order. However, after many trials we found that the Family level was too broad and not very helpful, so we only worked with Segment, Class and Commodity level. The first level of the Trie is the Segment level. To increase matches we merged Segment Titles with Segment Definitions, Class Titles with Commodity Titles for the second level, and Commodity Titles for the third level of matching. Finally at the end of the Trie we have the corresponding UNSPSC Commodity code.

```
from collections import defaultdict
```

1 chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.cs.toronto.edu/~hinton/csc2515/notes/retrieval.pdf

```python
def createTree():
   main = defaultdict(dict)
   for idx,row in df_un.iterrows():
       category = None
       for key,item in row.items():
           if key == "list_seg_title":
               seg = tuple(item)
               if seg in main:
                   category = main[seg]
               else:
                   main[seg] = defaultdict(dict)
                   category = main[seg]

           if key == "list_clas_comm":
               class_comm_title = tuple(item)
               if class_comm_title in category:
                   category = category[class_comm_title]
               else:
                   category[class_comm_title] = defaultdict(dict)
                   category = category[class_comm_title]


           if key == "comm_title":
               comm_title = tuple(item)
               if comm_title in category:
                   category = category[comm_title]
               else:
                   category[comm_title] = defaultdict(dict)
                   category = category[comm_title]

           if key == "commodity_code":
               if item:
                   comm_code = item
                   category["comm_code"] = comm_code
   return main
```

We implemented the Trie data structure using nested dictionaries.

Essentially, our structure would look like the following:
{Segment:{Commodity+Class:{Commodity Title:{Commodity Code:{12345 }}}}}

An example of the trie would look like this:

```
                              {'comm_code': 51132234})}),
      ('non',
       'combin',
       'opioid',
       'oxid',
       'aspirin',
       'analges',
       'magnesium',
       'carbon',
       'calcium'): defaultdict(dict,
                  {('oxid',
                    'aspirin',
                    'magnesium',
                    'carbon',
                    'calcium'): defaultdict(dict,
                              {'comm_code': 51132235})}),
      ('combin',
       'opioid',
       'aspirin',
       'analges',
       'dihydroxyaluminum',
       'aminoacet',
       'non'): defaultdict(dict,
                  {('aspirin',
                    'dihydroxyaluminum',
                    'aminoacet'): defaultdict(dict,
                              {'comm_code': 51132236})}),
```

Word list match algorithm:

Using Levenshtein algorithm:
In order to find string matches, we wrote a Levenshtein function to match the ratio of two strings

```
import copy
from prompt_toolkit.shortcuts.progress_bar.formatters import D
from tqdm import tqdm

def compareWordList(newTree,data_list):

   def get_max_score(tree,data_list,score):
       matches = {k:score for k in tree.keys()}
       # matches = defaultdict(int)
       curr_tree = tree
       for word in data_list:
           for key,val in curr_tree.items():
               for k in key:
```

```
                    ratio = levenshtein_ratio(word,k,True)
                    if ratio > 0.85:
                        matches[key] += 1
                # if word in set(key):
                #     matches[key]+= 1


        # print("matches",matches)
        max_score = max(matches.values())
        #when there are ties in number of matches, put all candidates in a list
        get_max_match = [(k,v,curr_tree[k]) for k,v in matches.items() if v ==
max_score]
        # print("get_max_match",get_max_match)

        return get_max_match

    #search trie with highest match score
    max_match = get_max_score(newTree,data_list,0)

    #level 2 search: commodity and class
    for key,score,level in max_match:
        new_max_match = get_max_score(level,data_list,score)
        # print("new_max_match",new_max_match)

    #level 3 search: commodity level matching
    curr_res = []
    for key,score,level in new_max_match:
        comm_max_match = get_max_score(level,data_list,score)
        curr_res.extend(copy.deepcopy(comm_max_match))

    # print(curr_res)
    max_value = -1
    max_level = None
    for key,score,level in curr_res:
        if score > max_value:
            max_value = score
            max_level = level
    return int(max_level["comm_code"]) if type(max_level["comm_code"]) ==
"float" else ""
```

Using stemming in text-preprocessing to get prefixes:
We found that using stemming in text-preprocessing speeds up our matching runtime, so instead of using the Levenshtein algorithm, we used stemming for our final output

```python
import copy
from prompt_toolkit.shortcuts.progress_bar.formatters import D
from tqdm import tqdm

def compareWordList(newTree,data_list):

   def get_max_score(tree,data_list,score):
       matches = {k:score for k in tree.keys()}
       # matches = defaultdict(int)
       curr_tree = tree
       for word in data_list:
           for key,val in curr_tree.items():

               if word in set(key):
                   matches[key]+= 1


       # print("matches",matches)
       max_score = max(matches.values())
       #when there are ties in number of matches, put all candidates in a list
       get_max_match = [(k,v,curr_tree[k]) for k,v in matches.items() if v ==
max_score]
       # print("get_max_match",get_max_match)

       return get_max_match

   #search trie with highest match score
   max_match = get_max_score(newTree,data_list,0)

   #level 2 search: commodity and class
   for key,score,level in max_match:
       new_max_match = get_max_score(level,data_list,score)
       # print("new_max_match",new_max_match)

   #level 3 search: commodity level matching
   curr_res = []
   for key,score,level in new_max_match:
       comm_max_match = get_max_score(level,data_list,score)
       curr_res.extend(copy.deepcopy(comm_max_match))

   # print(curr_res)
   max_value = -1
   max_level = None
   for key,score,level in curr_res:
       if score > max_value:
```

```
        max_value = score
        max_level = level
return max_level["comm_code"]
```