Homework 4: pokemon or big data?

The goal of this homework is to craft a data file that could be suitable as a starting point for a binary classification machine learning model on text. This will use a "bag of words" model.

You can read a little about them here: https://en.wikipedia.org/wiki/Bag-of-words_model (https://en.wikipedia.org/wiki/Bag-of-words_model)

Basically, you create a matrix of data that counts or measures something about the text. Sometimes this is words, as you see in the name. In this case, we will be using name parts. In many cases you would go through other actions to determine exactly the impactful elements and remove certain items. We are going to focus on just the core of creating a bag of words.

We can also remind ourselves of the essential structure needed for supervised learning. Each row needs to have the numerical factors (eg you can't just toss it text, it needs the numbers), and then it needs the known values for each of the entities at question. This is usually the last column.

So it'll be something like...

sentence number	the	fluff	dog	etc...	type
1	3	0	4	...	positive
2	1	2	5	...	positive
3	1	0	1	...	negative

This type of structure is really common for text analysis, and often you need to create it. We can see the main structure of this file as:

- sentence number: the unique id of the sentence
- the - etc...: these are the unique words found within the entire corpus (all the documents or entities), and the number of times they appear in each sentence is counted within the cell.
- type: this is the known value of the sentence. For example, positive and negative, for sentiment analysis. This is used for supervised learning techniques.

Goal

The goal of this program will be to extract the names of things out of two data files, process the values, collect them up, and assemble the data file.

More specifically, https://pixelastic.github.io/pokemonorbigdata/ (https://pixelastic.github.io/pokemonorbigdata/) presents an interesting idea. Are pokemon and big data names really similar? Can we explore this with machine learning?

Now, we aren't going to actually DO the machine learning, but we are going to focus on getting the data that we need prepared.

While there are some tools that can prepare things like this, it can be rare that your data are actually ready for them.

Method

As normal, this will be broken into steps. Example outputs will be given for you to check your work. You are welcome to borrow that output to use for the next step if you get stuck on a step or want to move on. Treat each step independently.

Let's get our program file set up first. As normal, you should do your import statements and functions at the top.

Restrictions

No modules other than `csv`, `collections`, and `string` are allowed. You may use them to test things and check your work if you wish, but you will not get credit for any elements using other modules.

Step 1: Add import statements

You will need the `csv` and `string` modules, plus the `Counter` function from `collections`.

Step 2: Place the `ngram_letters(name, num)` function in

Info

You can get this from Lab 5. You can copy this function in without change.

do this stuff

Be extra sure that you copied it in without aditing it.

check

At this point you should be able to execute the following code:

```
print(ngram_letters("pikachu", 2))
```

And get this as the output:

```
['pi', 'ik', 'ka', 'ac', 'ch', 'hu']
```

Step 3: Read in the data

Use the regular boilerplate code to load the CSV data in of the normal data file that we've been using.

Pokemon:

Info

Use the regular file we've been using.

Do this stuff

Load in the pokemon data file into a variable called `pokemon`.

Check

```
print(len(headers)) #should be 3
print(len(pokemon)) # should be 980, maybe a little different if you did some data cleaning and removed ce
rtain rows. That's okay.
```

Bigdata:

Info

This is a text file where each name is on a single line and not a CSV file.

Do this stuff

Load the bigdata text file into a list of the names.

When you read the file in, a) strip any punctuation off of and b) lowercase each of the names.

Check

Once you read it in, the length should be 78.

```
print(bigdata[:5]) # printing the first 5 values
```

```
['rocket fuel inc.', 'rocket u2', 'rubicon technologies', 'salesforceiq', 'semantic research']
```

Intermission

Now that you have the data loaded, we can start crafting things. Let's take another look at what our end game goal is for how this new data should look.

Generally, you should always have a plan you are headed toward. There's a good chance it'll need to change along the way, but heading toward that goal will reveal the other needs.

To set up our bag of words model, we need to know:

- what our entities are
  - translation: what are the items we are measuring? These may also be called "documents". This may be sentences, paragraphs, chapters of a book, the entire book, etc. Determining this sort of thing is a research area itself, as each item has separate considerations. I shall leave further discussion to you and you other coursework.
- the unique "vocabulary" across the entire corpus
  - translation: all the unique things we are measuring across the total body of text we have to work with. In this case, we aren't looking at words but at pairs of letters.
- the counts for the thing we are measuring for that entity.

Once we know the answers to these questions, we can start drafting up how our data structure needs to look.

As normal, the entities will become the rows and the properties will become the columns. The measurement of thosse properties for those entities will be the content of the cells.

When you approach answering these questions for other data work, try to not think about the code at this point.

Determining the order for this can be challenging. In what order should we create these things?

Discussion 1:

In a text file, spend about a paragraph discussing how you might think through the order of assembing the data. As in, given that there are three things we need to do, what should the order be? Don't just state what I'm about to have you do. Tell us how you think through it.

No spoilers. Go write discussion 1

In [6]:

```
for i in range(20):
    if i % 2 == 0:
        print('  ' * i, '<_<')
    else:
        print('   ' * i, '>_>')
```

```
<_<
   >_>
    <_<
        >_>
       <_<
            >_>
          <_<
               >_>
             <_<
                  >_>
               <_<
                     >_>
                 <_<
                        >_>
                    <_<
                           >_>
                       <_<
                              >_>
                          <_<
                                 >_>
```

Intermission part 2

In trying to think this through, we sort of end up in a chicken and the egg problem. How can we measure the data that we want but also how do we get that without having the data?

We can do this in a few steps:

1. collect all the unique letter pairs from the pokemon data & big data names
2. use that to determine the shape/dimensions of the matrix we need to make
3. count the values and then populate our matrix

In my own solution, I combined the steps of collecting the unique values and counting the values, but I'll have you go through them in distinct steps.

Step 4: create the column names

Info

We will need to gather all the letter pairs from each dataset and determine which ones are unique. Saving this as a list, we get the core of our matrix columns.

The order that the pairs are in does not matter. So we can just accept the order they arrive in.

Do this stuff

- create an empty list called `duos` that will collect all the letter pairs (why do so many of you not use the variable names I provide in here? it makes helping with your code and grading harder.)
- loop over the pokemon names and the big data names (these will be separate for loop). Run the names through `ngram_letters` asking for `2` grams. Store those values into `duos`.
- Cast `duos` as a set to get the unique values and save it as a variable called `unique_duos`.

Check

```
print("there are", len(duos), "individual pairs")
print(duos[:10])
print('there are', len(unique_duos), 'unique letter pairs')
```

```
there are 7148 individual pairs
['bu', 'ul', 'lb', 'ba', 'as', 'sa', 'au', 'ur', 'iv', 'vy']
there are 449 unique letter pairs
```

Step 5: finish preparing the headers

Info

Now that we have the pairs, we can finish preparing things. Look back up top and remember the things that we need to add:

- the entity id
- the known answer label

Do this stuff

We are just working on the headers, so make a new variable called `headers`. Add the string `'name'` at the beginning and `'kind'` at the end. Use `unique_duos` for the middle.

Check

```
print(len(headers))
```

```
451
```

This completes our headers! We can move on to making our counts.

Step 6: think about the structure we want do make

Info

So there are many ways to do this thing. I'll be guiding you through a way you can do it as a completely separate steps.

Our first instinct might be to do something like a word count pattern. Which would produce something like this:

```
{'bu': 27, 'ul': 18, 'lb': 5, 'ba': 28, ...
```

So this would be fine, except it doesn't give us the counts for each pokemon. We've combined everything together. So we need to make a dictionary that has the counts but keeps them clustered together by entity. Again, this is the type of activity where there are so many ways to organize the datae. I'll be showing you how a method that allows you to store the data in a way that you can inspect and check on before assembling the data file. This adds more steps but is super important!

First, we want to think of the structure. What should the keys and the values be? Let's first start thinking of which values are unique to each entity.

I suggest that we use the names as the keys and the values be the counts. This will allow us to easily see the entities and the data will be pretty readable.

Discussion 2: what should be in the counts?

Now that we (well, I have declared) that the dictionary values should be the counts of the letter pairs, we have a decision. Do we want the counts to have every unique pair represented in them, or only the pairs that are present within that entity's name?

Using your text document, spend about a paragraph writing down your thoughts on it. Honestly, there are fair arguments for both sides here, even though I'll be asking you to only do the ones where they are present. What do you thing the pros and cons would be for each?

Step 7: Producing and collecting the counts

Info

We will run both sets of names through the `ngram_letter` function again, but this time using the `Counter` object from `collections` to count how many times it appears. This will be the item that we will save.

At the same time, we have two situations to handle:

1. we have no guarentee that there aren't repeat names within the corpus. This would not change the overall counts of the letter parts, but we don't want to erase the existance of a name existing in one group but not the other.
    - we need to a) make a unique name for the entities, and b) retain the original name somewhere
2. When we combine the counts for the two groups together, we lose the context for which group the entitiy came from.
    - we need to add a field to label the category that the entitiy came from

Do this stuff

Collect names

- Collect all the pokemon names and bigdata names into separate lists (you'll need separate loops). Use `set()` on each to generate a list of all the unique values. Save each as separate lists called `unique_pokemon` and `unique_bigdata` .
    - We want to get unique names from each of these datasets because we know that the names should be unique, but there are duplicates in the pokemon data. While there are none in the big data file, it can be best practice to do this as a data cleaning step. We may not want to do this if we had another research question, etc. and the duplication would be a possibly factor that would influence our model.

Count the parts

- create an empty dictionary called `counted_parts` .
- loop over `unique_pokemon` and `unique_bigdata` (in separate loops).
- Within these loops:
    - run the name through the `ngram_letters` function asking for 2 letter pairs.
    - Run those results through `Counter` from `collections` . Recast this as a dictionary. Save it as `entity_results` .

Stop here and print off the counts to check that things are working.

Handle the unique name situation (still inside these loops)

- Append a string like "-p" for pokemon and "-b" for big data to the end of the names to disambiguate the names for each group. Save this as a new variable called `name_id` . This will now work as our new entity id.
    - **Brief intermission: stop here and test your code. Print off the name, name id, and the counted results to check that they make sense. Take a breath. More to add.**

Add the type values to the results

- Now that you have the new key (`name_id`) and the start of the results (the counted letter pairs), we can add the original name and the type label to our dictionary. This will complete our data storage for this entity.
- Add these two items to your `entity_results` dictionary:
  - the original name using `'name'` as the key.
  - the entity type using `'type'` as the key.
  - (note: this is fine to mix the labels and the data together becasue the letter pairs will only be 2 letters long, so our 4 letter long keys won't interfere with the data.)

Check

Check your work by running this: `print(entity_results)` . You should see results like: `{'de': 1, 'ew': 1, 'wg': 1, 'go': 1, 'on': 1, 'ng': 1, 'name': 'Dewgong', 'type': 'pokemon'}` and `{'om': 1, 'ma': 1, 'as': 1, 'st': 1, 'ta': 1, 'ar': 1, 'name': 'Omastar', 'type': 'pokemon'}` .

Collect up the completed dictionaries

Finally, once all that is done, add these final dictionaries to your `counted_parts` dictionary. Remember to use `name_id` as the key and `entity_results` as the value. When you adapt your code for the other group, be sure that you update the name disambiguation and the type label to match that group.

Check things

```
print(len(unique_pokemon), len(unique_bigdata))
```

```
950 78
```

```
print(len(counted_parts))
print(counted_parts['Pikachu-p'])
print(counted_parts['splunk-b'])
```

```
1028
{'pi': 1, 'ik': 1, 'ka': 1, 'ac': 1, 'ch': 1, 'hu': 1, 'name': 'Pikachu', 'type': 'pokemon'}
{'sp': 1, 'pl': 1, 'lu': 1, 'un': 1, 'nk': 1, 'name': 'splunk', 'type': 'bigdata'}
```

Double check the values in each of the fields for these results.

Okay, almost there. I promise. This one was the worst part.

Step 7: assemble the rows for the new matrix of data

Info

At this point we have all the data values that we want collected and assembled into a single data structure. Unfortunately, that structure isn't what our final file needs to look like. But that's okay! we can transform dictionary structure to csvs.

Here are the tools that we are going to use:

- we can use the `items` dictionary method to loop through the stuff in `counted_parts` . This will give us access to the unique id, the counted letter values, the original name, and the type. We can also use `items` to loop over the dictionary values within this main list. Getting access to the field names and values for each entitiy.
- Remember the `headers` list that we created? We can use the field values we are getting in the subloop and the `index` list method on headers to get the index position for where that value needs to be placed within the columns.
- There also is a syntax trick that you can use to make an arbitrary number of values within a list. Play with this in a separate session to understand it.
  - Evaluating the following: `[0] * 5` will give you a list back like this: `[0, 0, 0, 0, 0]` . You can populate a list with a value and use the `*` operator and a number (n) to return a new list with that value repeated n number of times. This will work if you have more values as well, just duplicating the order. (protip: this also work with things like `[ [] ]` if you want to create a 2d list of empty lists. Not for this assignment, but a handy trick.)

Anyhow, we can use this trick to create a list prepopulated with as many `0` s as there are values in headers.

Why `0` ? Well, we will only be changing the values that are present for our specific entity. This gives us the advantage that any values the entity does not have will remain a zero. You can read more about this here: https://en.wikipedia.org/wiki/Sparse_matrix (https://en.wikipedia.org/wiki/Sparse_matrix)

Also remember that we are working with a nested loop here. The outside loop is looking at the entire entity. The inside loop is looking at the entity's individual values. Recall how we craft list accumulators within nested loops. The "rows" that we are creating are at the entity level, thus creating the baseline row list will be within our outer loop. Then we populate the row from within the inner loop.

Do this stuff

I'll give you this much, this is the for loop you need to use:

```
for entity, values in counted_parts.items():
    for field, cell_value in values.items():
        print(field, cell_value)
```

The entries you see should look something like this:

```
te 1
er 1
ra 1
ad 1
da 1
at 1
ta 1
name teradata
type bigdata
```

So, within this entire nested loop contraption:

- create your baseline list called `row` and populate it with as many `0` s as there are items in header. (use the `*` thing with the length of `headers`)
- When looping over the field names, use the `index` method to look up the position where that field name is within `headers` . Set that index position within `row` to have the `cell_value` .

Check

At this point, you should test what you've made. At the end of your outside loop (pay attention to all those words there), print off row. The last thing you should see is something like this (your entity might be different):

```
['tubemogul', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 'bigdata']
```

At last!

Do this one last thing

The final thing we need to for this is collect all the final instances of `row` into another list called `bag` . Add a list accumulator pattern that will collect all the final values of `row` .

Check

```
print(len(bag))
print(bag[305])
```

```
1028
['Castform', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'pokemon']
```

Step 8: write out the csv file

Use the normal csv boilerplate code to write out a csvfile called `pokemon_bigdata_bow.csv` . Use `headers` and `bag` for the content.

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```