

Zesty.AI Analyst Challenge: Part 1

Import Libraries & DataFrames

I begin the project by importing the Pandas library and the data sets provided for this case. I then run a few commands to understand how the data looks.

```
In [1]: import pandas as pd  
test = pd.read_csv(r'/Users/fatim_/Downloads/test.csv')  
clean = pd.read_csv(r'/Users/fatim_/Downloads/all_addresses.csv')
```

```
In [2]: test.dtypes  
print(test.dtypes)
```

```
uid      int64  
address   object  
city     object  
state    object  
zip     float64  
dtype: object
```

```
In [3]: clean.dtypes  
print(clean.dtypes)
```

```
address   object  
city     object  
state    object  
zip     object  
dtype: object
```

```
In [4]: len(test)
```

```
Out[4]: 99249
```

```
In [5]: len(clean)
```

```
Out[5]: 130000
```

A few observations can already be made. Firstly, I can see that the clean dataframe (i.e. the list of correctly formatted addresses) is larger than the test dataframe. That implies we cannot do any row to row comparisons between the two dataframes, as there is no guarantee the dataframes are ordered.

Find the starting coverage rate

Let's first start simple with street addresses and see how many exact matches there are from the test dataframe to the clean dataframe.

```
In [6]: exact_matches = test[test['address'].isin(clean['address'])]  
exact_matches_count = len(exact_matches)  
print(f"Number of exact matches: {exact_matches_count}")
```

```
Number of exact matches: 72096
```

Great! We can see that there are 72,096 out of 99,249 street addresses from our test dataframe that are an exact match to the clean dataframe. However, we've excluded the other location fields such as city, state and zip. To provide an accurate current coverage rate, we need to see how many complete addresses are exact matches between the two dataframes.

Let's do this by concatenating all the property fields together to create a full address.

```
In [7]: #convert test.zip from float to string, and remove the extra characters at the end.  
test["zip"] = test["zip"].astype(str).str[:-2]  
  
#creating full addresses for each dataframe  
test["fulladdress"] = test['address'] + ", " + test['city'] + ", " + test['state'] + " " + test['zip']  
clean["fulladdress"] = clean['address'] + ", " + clean['city'] + ", " + clean['state'] + " " + clean['zip']
```

Now let us see how many exact full address matches there are between the test and clean data set.

```
In [8]: exact_matches_fulladdress = test[test['fulladdress'].isin(clean['fulladdress'])]  
exact_matches_fulladdress_count = len(exact_matches_fulladdress)  
print(f"Number of exact full address matches: {exact_matches_fulladdress_count}")
```

```
Number of exact full address matches: 64526
```

There are currently 64,526 exact full address matches from the test to clean data set. This means the current coverage rate is 65% .. we have ways to go! Before we jump right in to clean and format the test dataframe to maximize it's coverage rate, let's do a pre-liminary analysis and locate the data that is currently incorrect.

Locating Missing Data & Data Inconsistencies

Is our test dataframe missing data? Is there only a single column that has incorrect data? Are there multiple? Let's see! Let's first do a quick check for missing data.

```
In [9]: test.isnull().sum()
```

```
Out[9]: uid      0  
address   0  
city     1282  
state    7433  
zip      0  
fulladdress  8715  
dtype: int64
```

Empty entries exist for both city and state columns. On an interesting note, there are 8,715 incomplete fulladdresses, which is a summation of missing city and state entries. This means that records can either have a missing city entry or a missing state entry, but they never have both.

Another test you can do for inconsistencies, is run "test['address'] = test['address'].str.lower()" and "clean['address'] = clean['address'].str.lower()", and do the same matching exercise we did previously against the test and clean dataframe. If more matches show up after applying the lowercase function to your string columns, that means there are in-correct casings in your data that you need to correct for. To save space in this paper, I have done this test aside and can conclude there are no incorrect casings in the test dataframe.

Now let's see if there are certain columns that are inconsistent with the others when comparing across the test and clean dataframes. Let's start off with zipcode. The exercise is as follows: If there is a record where the fields address, city and state, all together match between the test and clean data frame, the zipcode should hypothetically match as well. If it does not, then the zipcode is incorrect. Let's see how many incorrect zipcodes we have in the test data frame.

```
In [10]: #Finding out if there are records between test and clean, where zipcode doesn't match but address, city and state all match.  
#Match address, city and state from the test data frame to the clean data frame  
matches_no_zip = test[test[['address', 'city', 'state']].isin(clean[['address', 'city', 'state']]).all(axis=1)]  
  
#Match address, city, state and zip from the test dataframe to the clean data frame.  
matches_zip = test[test[['address', 'city', 'state', 'zip']].isin(clean[['address', 'city', 'state', 'zip']]).all(axis=1)]  
  
#From test, find and print rows that are in matches_no_zip but not in matches_zip
```

```

rows_to_print = test[matches_no_zip[~matches_no_zip.isin(matches_zip)].dropna().index]
print(rows_to_print)

Empty DataFrame
Columns: []
Index: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, ...]

[99249 rows x 0 columns]

Our dataframe is empty, which means zipcodes for the properties that match atleast address, city, and state, are correct! Let's do this same exercise independently for address, city and state as well. We'll start off with city, state then finish with address.

In [11]: #Finding out if there are records between test and clean, where city doesn't match but address, state and zip all match.
#Match address, state and zip from the test dataframe to the clean data frame. Exclude city.
matches_no_city = test[test[['address', 'state', 'zip']].isin(clean[['address', 'state', 'zip']]).all(axis=1)]

#Match address, city, state and zip from the test dataframe to the clean data frame.
matches_city = test[test[['address', 'city', 'state', 'zip']].isin(clean[['address', 'city', 'state', 'zip']]).all(axis=1)]

#From test, find and print rows from test that are in matches_no_city but not in matches_city
rows_to_print = test[matches_no_city[~matches_no_city.isin(matches_city)].dropna().index]

print(rows_to_print)

Empty DataFrame
Columns: []
Index: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, ...]

[99249 rows x 0 columns]

In [12]: #Finding out if there are records between test and clean, where state doesn't match but address, city and zip all match.
#Match address, city and zip from the test dataframe to the clean data frame. Exclude state.
matches_no_state = test[test[['address', 'city', 'zip']].isin(clean[['address', 'city', 'zip']]).all(axis=1)]

#Match address, city, state and zip from the test dataframe to the clean data frame.
matches_state = test[test[['address', 'city', 'state', 'zip']].isin(clean[['address', 'city', 'state', 'zip']]).all(axis=1)]

#From test, find and print rows from test that are in matches_no_state but not in matches_state
rows_to_print = test[matches_no_state[~matches_no_state.isin(matches_state)].dropna().index]

print(rows_to_print)

Empty DataFrame
Columns: []
Index: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, ...]

[99249 rows x 0 columns]

In [13]: #Finding out if there are records between test and clean, where address doesn't match but state, city and zip all match.
#Match address, city and zip from the test dataframe to the clean data frame. Exclude address.
matches_no_address = test[test[['city', 'state', 'zip']].isin(clean[['city', 'state', 'zip']]).all(axis=1)]

#Match address, city, state and zip from the test dataframe to the clean data frame.
matches_address = test[test[['address', 'city', 'state', 'zip']].isin(clean[['address', 'city', 'state', 'zip']]).all(axis=1)]

#From test, find and print rows from test that are in matches_no_address but not in matches_address
rows_to_print = matches_no_address[~matches_no_address.isin(matches_address.index)]
```

uid	address	city	state	zip
3	7 ucaylptus	Newport Beach	CA	92657
5	725 ounain View St	Altadena	CA	91001
7	431 6th St	Sacramento	CA	95820
9	3640 Oak Creek Dr, Unit 10	Ontario	CA	91761
11	5571Moringside Dr	Clayton	CA	94517
...
94740	1610 Shern Way	Petaluma	CA	94954
95833	1356 San Madele Ave	Fresno	CA	93711
96595	27012 Cordero Ln, Unit 96596	Mission Viejo	CA	92691
96623	9788 Monterey Rd	Morgan Hill	CA	95037
98554	18588 Caminito Pasadero	San Diego	CA	92128
	fulladdress			
3	7 ucaylptus, Newport Beach, CA 92657			
5	725 ounain View St, Altadena, CA 91001			
7	431 6th St, Sacramento, CA 95820			
9	3640 Oak Creek Dr, Unit 10, Ontario, CA 91761			
11	5571Moringside Dr, Clayton, CA 94517			
...	...			
94740	1610 Shern Way, Petaluma, CA 94954			
95833	1356 San Madele Ave, Fresno, CA 93711			
96595	27012 Cordero Ln, Unit 96596, Mission Viejo, C...			
96623	9788 Monterey Rd, Morgan Hill, CA 95037			
98554	18588 Caminito Pasadero, San Diego, CA 92128			

[71 rows x 6 columns]

Looks like there are no records where city, state, or zip are incorrect but all the other fields match! Great, this means no cleaning for these columns (but we're not done with these columns - we'll get back to this in the next section).

However, for address we see there are 71 rows where the city, state and zip match between the rows of the two dataframes, but the address is incorrect. Is this something we can correct for? Let's give this a shot in the next section.

Data Cleaning & Transformations

We know that there are issues with the accuracy of our street addresses. If we do a data preview using `test.head(100)` there are a few things that stand out immediately that we can do to increase our coverage rate.

Cleaning street addresses.

1. Remove unit numbers from street addresses. (ex. "3640 Oak Creek Dr, Unit 10" to "3640 Oak Creek Dr")
2. Add a space after the street number and street name (ex. "9690Canon Way" to "9690 Canon Way")
3. Add missing letters to street names (ex. "525 heynne Dr" to "525 Cheyenne Dr")

We'll get to points 1. and 2. in a moment, but let's first address 3. (no pun intended). In order to add in the missing letters to street names, you will have to find the correct street address using external mapping libraries. For example, a Google Maps API connection can be made where we can search and fill in the street address based on the other four fields (city, state and zip), assuming zipcode is unique to the street address. However, that is beyond the scope of this assignment so we will not be actioning on 3). Note, this is one of the factors that will prevent us from reaching a full coverage rate.

Without further ado, let's action on points 1. and 2..

```
In [14]: #1 . Remove unit numbers from street addresses
test['address'] = test['address'].str.replace(r'\d+', '')

/tmp/ipykernel_15366/2349279892.py:2: FutureWarning: The default value of regex will change from True to False in a future version.
test['address'] = test['address'].str.replace(r'\d+', '')

In [15]: #2. Add a space after the Street Number and Street Name.
test['address'] = test['address'].str.replace(r'(\d+)\s*([A-Za-z]+)', r'\1 \2')

/tmp/ipykernel_15366/3280016895.py:2: FutureWarning: The default value of regex will change from True to False in a future version.
test['address'] = test['address'].str.replace(r'(\d+)\s*([A-Za-z]+)', r'\1 \2')

If you do a data preview using test.head(10), it looks like adding a space between the numbers and letters of address resulted in a little mishap where street names that include a number in the name, got split up. (ex. '431 6th St' became '431 6 th St'). Let's quickly correct for this.

In [16]: #Remove spaces where there is a number of digits, followed by one or more spaces, and the letters "th" or "nd"
def remove_space_between_number(test, column):
    test[column] = test[column].str.replace(r'(\d+)\s+(th|nd)\b', r'\1\2')

# Apply the function to the 'address' column
remove_space_between_number(test, 'address')

/tmp/ipykernel_15366/3397150675.py:3: FutureWarning: The default value of regex will change from True to False in a future version.
test[column] = test[column].str.replace(r'(\d+)\s+(th|nd)\b', r'\1\2')
```

Completing missing city information

We noted at the beginning of our pre-liminary analysis that both city and state contain null values. A transformation step that can be taken here is to fill in the empty cells based on the other fields. We noted previously that a record will never be missing both city and state, it will only exclude one or the other. So you can index against the non-missing field alongside zipcode, to fill in the missing field. However, due to similar reasoning we mentioned for missing letters in street addresses, that is beyond the scope of this assignment and will be skipped in this analysis.

Final Coverage Rate

We've done a few manipulations to our data set, now let's see how much we've increased our coverage rate by!

```
In [17]: #creating a new full addresses column for each dataframe
test["fulladdress2"] = test['address'] + ", " + test['city'] + ", " + test['state'] + " " + test['zip']
clean["fulladdress2"] = clean['address'] + ", " + clean['city'] + ", " + clean['state'] + " " + clean['zip']

In [18]: exact_matches_fulladdress2 = test[test['fulladdress2'].isin(clean['fulladdress2'])]
exact_matches_fulladdress2_count = len(exact_matches_fulladdress2)
print(f"Number of exact full address matches: {exact_matches_fulladdress2_count}")

Number of exact full address matches: 71755
```

We increased the number of exact address matches from 64,526 to 71,755 hoorah! Hence our coverage rate increased from **65% to 72%**. That is an **increase of coverage rates by 7%**. Let's go analyze some properties!