case_5.1

May 13, 2020

1 How do we prepare data for use with an analytics platform?

[3]:
```
#! pip install boto3
! pip install geopy
```

```
Collecting geopy
  Downloading geopy-1.22.0-py2.py3-none-any.whl (113 kB)
     |################################| 113 kB 364 kB/s eta 0:00:01
Collecting geographiclib<2,>=1.49
  Downloading geographiclib-1.50-py3-none-any.whl (38 kB)
Installing collected packages: geographiclib, geopy
Successfully installed geographiclib-1.50 geopy-1.22.0
WARNING: You are using pip version 20.0.2; however, version 20.1 is
available.
You should consider upgrading via the '/root/.virtualenvs/ds4a-py3/bin/python -m
pip install --upgrade pip' command.
```

[4]:
```
import boto3
import base64
import datetime
import json
import os
import numpy as np
import pandas as pd
from geopy.geocoders import Nominatim
```

1.1 Introduction (5 mts)

Business Context. You are a data science consultant for a bike share company. The company has hundreds of thousands of users and has been collecting data about trips taken on each of their bikes. Since the dataset collected is quite large and increasing by the day, they have subscribed to a new analytics platform which gives them information and insights when they feed trip data into it. However, the analytics platform requires the collected data to be cleaned and converted into a certain format, for which the client requires your help.

Business Problem. Your task is to transform the raw data the client has into a format that can be fed into their analytics platform, as well as add additional features that the client wants to see in their platform, which are mentioned below.

Analytical Context. The data provided by the client is spread across 2 files. The first file is a CSV file that contains all the trip data, with features such as trip time, start and end stations, bike number, and the user details like whether it is a registered or a casual user, etc. The second file is a JSON file containing details about the stations they own. In addition to cleaning their existing data, the client wants you to add the following features:

1. Generate a unique ID for each trip
2. For each trip, calculate its duration, and based on this generate 4 more columns: `start_hour`, `end_hour`, `start_weekday`, `end_weekday`
3. For each trip, create new columns for age of the user, start and end coordinates, and municipal/city details of the start and end stations

In this case, you will: (1) fetch the raw data from Amazon S3 and take a subset for local use (because the dataset is quite large); (2) use common sense to judge likely use cases of the data and clean it; (3) add new features to the data based on client request; and finally (4) upload the data wrangling scripts we have developed as a Jupyter Notebook to Amazon EC2 so we can apply it to the entire dataset.

1.2 Fetch data from S3 (10 mts)

If the data files are provided to you via an Amazon S3 bucket, to access the bucket, you will need to get your Access Key and Secret Key. Instructions for getting your AWS key can be found here: https://help.bittitan.com/hc/en-us/articles/115008255268-How-do-I-find-my-AWS-Access-Key-and-Secret-Access-Key-

Our first step is to copy the file from S3 to your local machine. We will use the `boto3` library to do that:

[ ]:
```python
# Setting up s3 client
AWS_ACCESS_KEY = 'InsertKeyHere'
AWS_SECRET_ACCESS_KEY = 'InsertSecretHere'
S3_BUCKET_NAME = 'data-wrangling-case'

s3_client = boto3.resource(
    's3',
    aws_access_key_id=AWS_ACCESS_KEY,
    aws_secret_access_key=AWS_SECRET_ACCESS_KEY
)

s3_bucket = s3_client.Bucket(S3_BUCKET_NAME)
local_folder = '.'



# Pull the contents from the data folder into the local path
for obj in s3_bucket.objects.all():
    local_file = os.path.join(local_folder, obj.key)
    s3_bucket.download_file(obj.key, local_file)
```

1.3 Exploring the data (5 mts)

The data is now copied to the local folder. Let's read the data using the `read_csv()` method. Note that we are passing a parameter `nrows=10000` so that we only read in the first 10000 rows in the file (as mentioned earlier, we are working with a subset of the data on our local machine as the entire dataset is quite large):

[6]:
```
local_folder = '.'
# Read the data file and take a look at the data
df = pd.read_csv(os.path.join(local_folder, 'trips.csv'), nrows=10000)

df.head()
```

```
       ␣
↪---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call␣
↪last)

        <ipython-input-6-4d79f2c95194> in <module>
          1 local_folder = '.'
          2 # Read the data file and take a look at the data
    ----> 3 df = pd.read_csv(os.path.join(local_folder, 'trips.csv'),␣
↪nrows=10000)
          4
          5 df.head()


        NameError: name 'pd' is not defined
```

As discussed before, the first step we ought to take with a new dataset is to familiarize ourselves with it. Let's read through the columns present in the dataset, find out how the data is spread out across columns, etc. This will also give us a sense of the obvious cleaning steps to be performed on each column present in the dataset.

[7]:
```
# Take a look at the columns present
df.columns
```

```
       ␣
↪---------------------------------------------------------------------------
```

```
    NameError                                 Traceback (most recent call␣
↪last)

    <ipython-input-7-248b7a9c6c59> in <module>
      1 # Take a look at the columns present
----> 2 df.columns


    NameError: name 'df' is not defined
```

The list of available featuers is as follow:

1. **status:** Status of the trip (”Ongoing”, ”Closed”)
2. **start_date:** Start time of the trip
3. **start_station:** id of the station from which this trip started
4. **end_date:** End time of the trip
5. **end_station:** id of the station from which this trip ended
6. **bike_nr:** The unique identifier of the bike used in this trip
7. **subsc_type:** Subscription type of the user (”Registered”, ”Casual”)
8. **zip_code:** If it is a registered user, their zipcode
9. **birth_date** If it is a registered user, their date of birth
10. **gender** If it is a registered user, their gender (”Male”, ”Female”)

1.4 Handling null values (25 mts)

Let’s start with the data cleaning process. As discussed in the intro Python cases, one of the first steps is to deal with null or missing values. However, previous cases only gave a passing treatment of these and resulted in dropping the rows containing null values entirely. Here, we will be more nuanced and look at ways that null or missing values can be replaced with more understandable and logical values.

Generally, null values in a specific column are dealt with in one of the following ways:

1. Any row containing a null value for that column is removed
2. If that column’s feature is a string, null values are replaced with a ”Not found” string. If that column’s feature is a number, null values are replaced either with 0 or the mean/median of the available values in the column
3. Null values are replaced with an interpolated value based on the data present in other rows

Let’s start by taking a look at the list of columns that have null values:

```
[ ]: df.isnull().any()
```

We can see that there are 5 columns that have null values. We need to decide, based on the importance of each column, whether we will be applying a blunt instrument and remove the rows that have a particular column as null, or if we will be more nuanced and replace the null values in that column with a replacement value.

1.4.1 Exercise 1:

`bike_nr` is one of the columns with missing values. Which of the above three methods do you think is most appropriate?

Answer. We can first see that method 3 is not sensible. This is because `bike_nr` is assigned before any trips are taken with that bike; hence, every other feature of the dataset is necessarily generated after the bike number is assigned. This means that there is no rhyme or reason for what `bike_nr` should be based on the other columns (because its creation preceded all the others), so there is no reason to expect a reasonable value to result from interpolation.

Given that at best we can only fill `bike_nr` with a meaningless filler value, it may be tempting to remove rows without `bike_nr` entirely. However, removing a row entirely tends to be the nuclear option and in the absence of a very clear and limited use case for the dataset going forward, it may result in us regretting our choice once we realize that some data we cut out might actually be useful. For example, if in some future analyses the client cares a fair amount about aggregate statistics relating to many or all the trips, then a specific `bike_nr` value is not important and we can still get useful analysis by replacing each `bike_nr` with a ”Not Found”, whereas we could well be missing that analysis if we cut out too much data. Thus, option 2 is the best answer.

Based on the above, let’s now replace null values for `bike_nr` with ”Not found”:

[3]:
```
df['bike_nr'].fillna('Not Found', inplace=True)
```

```
        ␣
 ↪---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call␣
 ↪last)

        <ipython-input-3-63e8c11d27a1> in <module>
    ----> 1 df['bike_nr'].fillna('Not Found', inplace=True)


        NameError: name 'df' is not defined
```

The next column we can concentrate on is `gender`. We know that `gender` can have only one of the 2 values: ”Male” or ”Female”.

1.4.2 Exercise 2:

Suppose that one thing (but not the only thing) our client cares about is a rough idea of how the number of male and female riders has changed over time. Which of the above three methods would be most appropriate?

Answer. Again, removing rows entirely is the nuclear option, and there is no reason to do so without a very clear and limited objective for this dataset. Now, we may be tempted to fill in the missing values with ”Not found”. However, what if the missing data is disproportionately concentrated in later dates? Then later on, when the client tries to use their software to plot this trend, they will get a growth rate result that is far too low compared to real life. Thus, the best option is to fill in the missing data using some interpolation method to preserve the integrity of the growth rate data.

Now, how do we go about doing this? Well, we can use the `interpolate()` function in `pandas`. The `interpolate()` function uses linear interpolation, which is a mathematical method for filling in unknown points based on building a linear regression model on the non-missing points. We can then use this model to estimate the values of the missing points. This is very different from substituting null values with random or meaningless values, as it preserves aspects of the distribution of the data, which can be very important for certain analyses.

But gender is a string; how can we apply a mathematical model to a string? Well luckily, gender can only take on two values, so we can convert it to a `category` type and then run the interpolation:

```
[ ]: # Convert gender to a category type
     df['gender'] = df['gender'].astype('category')
```

After converting `gender` to be a category column, let's see what the data looks like:

```
[ ]: df['gender'].cat.categories
```

```
[8]: # Let's see the codes assigned to the values present in the DataFrame.
     df['gender'].cat.codes
```

```
      ␣
 ↪---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call␣
 ↪last)

        <ipython-input-8-827a6c983ba9> in <module>
          1 # Let's see the codes assigned to the values present in the␣
 ↪DataFrame.
    ----> 2 df['gender'].cat.codes


        NameError: name 'df' is not defined
```

You can see there are 3 codes here: -1, 0, 1. But in the categories list, there are only 2 values: "Male" and "Female". This is because -1 represents the NaN values. In order to interpolate the values, we need to convert the -1 into actual NaN, as interpolation works only on NaN values:

```
[ ]: # The below code replaces the value -1 with NaN.
     gender = df['gender'].cat.codes.replace(-1, np.nan)
     gender
```

```
[ ]: # We now call the interpolate function that actually fills the NaN values with␣
     ↪either a 0 or 1
     gender = gender.interpolate()
```

```
[ ]: gender = gender.astype(int).astype('category')
     gender = gender.cat.rename_categories(df['gender'].cat.categories)
     df['gender'] = gender
     df['gender']
```

In the above snippet, we are converting the interpolated values into category type and then replacing the category names with the names from the existing DataFrame column. So now the gender column will contain ”Male” and ”Female” as its values (rather than 0 and 1).

1.4.3 Exercise 3:

Come up with a proper replacement scheme for null values in `zip_code`, `birth_date`, and `end_station`.

Answer. Let’s start with `zip_code`. Again, removing rows is a nuclear option and we should avoid it unless we have a very clear and limited objective for the dataset. So we can either replace zip code with a meaningless filler value, or attempt to interpolate it. However, we have no other features in this dataset that can sufficiently narrow down the bike rider’s/user’s zip code in any significant way. Therefore, any interpolation attempt would involve a lot of hand-wavey guessing. Thus, we will go ahead and replace null `zip_code` values with a meaningness filler value. Since zip codes are effectively strings (they are numbers, but they have no natural ordering; i.e. the fact that one zip code number is larger than another is meaningless), we will replace null values with ”Not found”.

The same line of reasoning leads us to fill in ”Not found” for missing values in the `end_station` column.

For `birth_date`, a similar line of reasoning as above tells us we should fill it with a meaningless filler value as well. Now, because `birth_date` contains only the year of birth, it can be considered to be a numeric field. So we default the null `birth_date` values to 0:

```
[ ]: df['zip_code'].fillna('Not found', inplace=True)
     df['zip_code']
     df['end_station'].fillna('Not found', inplace=True)
     df['end_station']
     df['birth_date'].fillna(0, inplace=True)
     df['birth_date']
```

Did you notice the `inplace=True` parameter passed to the `fillna()` function? By passing this parameter we are asking `pandas` to make the changes in memory, instead of creating a new object that contains the result of this operation. The above statement is technically the equivalent of:

```
df['gender'] = df['gender'].fillna('No Gender')
```

1.5 Correcting erroneous string values (10 mts)

After cleaning the null values in the dataset, the next process we usually do is to check for and deal with erroneous values in the dataset. For string columns, this can manifest in a few ways:

1. Unncessary spaces at the start or end of the string
2. Mixed case (e.g. "value 1" and "Value 1")
3. Spelling mistakes

However, this can be quite difficult to clean for in practice. In particular, number 3 (spelling mistakes) are nearly impossible to find if that string column is meant to be free-form; any legitimate English word would be a legal entry, and you would have to cross-check each entry against the entire English dictionary!

Luckily, certain types of string columns are far easier to deal with. For example, `subscription_type`, `gender` and `status` are categorical; i.e. their values are supposed to come from a defined set of options. This means that we can do some basic data summarizing to see whether they contain any values they are not supposed to:

[1]:
```
print("Subscription Type: ", df['subsc_type'].unique())
print("Gender:", df['gender'].unique())
print("Status:", df['status'].unique())
```

```
               ␣
 ↪---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call␣
 ↪last)

        <ipython-input-1-0920ec10c5cb> in <module>()
    ----> 1 print("Subscription Type: ", df['subsc_type'].unique())
          2 print("Gender:", df['gender'].unique())


        NameError: name 'df' is not defined
```

The data shows that `gender` and `status` only have 2 values, as expected.

However, in the `subscription_type` column, you will see that there are 3 `Regular`s and 3 `Casual`s. This is because of leading and trailing spaces. You can also see that this column has `regular` and `casual` as values. Though `Regular` and `regular` are same, they are identified as 2 different values because of their mixed case.

Let's go ahead and fix both of these issues:

[ ]:
```
# Let's fix the space issue. We will use the strip function to remove the␣
 ↪leading and trailing spaces
# for each row in the subsc_type column
```

```
df['subsc_type'] = df['subsc_type'].str.strip()

df['subsc_type'].unique()
```

[ ]:
```
df['subsc_type'] = df['subsc_type'].str.upper()

df['subsc_type'].unique()

# We have cleansed both the gender and subscription type columns to contain␣
 ↪proper values.
```

In practice, data scientists will apply the `strip()` and `upper()` (or `lower()`, either is fine) functions even to free-form, non-categorical string columns to avoid any unintentional duplicates (though as mentioned earlier, spelling mistakes need to be dealt with separately). An example of this is if you had a column specifying the city someone was from – you would not want "New York City, New York" to be treated differently from "new york city, New York".

There are many ways that values within a column can be erroneous, and we have only covered a few of them here. In future cases, you will learn about other ways erroneous values can creep into your dataset (say, in numeric or `datetime` columns) and how to deal with them. For now, let's move on.

1.6 Generating a unique ID for each trip (10 mts)

Let's get going on the specifics that the client wants. The first request relates to unique identifiers. When generating unique identifiers for datasets like this, we should make sure the generation process is idempotent (i.e. the same ID should be generated for each trip no matter how many times you run the script). The idempotency is required because there may be chances that the same trip is input into this tool multiple times. For example, the customer first uploads the data set for the first week of the month (may be for testing purposes, or based on data availability, etc.) and then uploads the data for the entire month. Now if the same trip is assigned different IDs on each run, then it might result in the analytics platform interpreting this as two different trips and this will skew the analysis.

1.6.1 Exercise 4:

Describe how you would generate a unique ID per trip while guaranteeing idempotence, then write code to do this.

Answer. Before it gets assigned an ID, a trip can only be uniquely identified via a suitable combination of its features. We reason that `start date`, `end date`, and `bike_nr` will always be unique, because any given bike can only be taking one trip at a time. Therefore, we will define a one-to-one function of these three quantities to generate the unique ID. (A one-to-one function is one such that for any given output, there can only be one possible input that generates that output.)

Now let's go ahead and write code for this:

```
[ ]: # Let's generate an id for each trip, in order to uniquely identify each trip.
     # The trip id can be a combination of start_date, end_date, start_station,␣
      ↪end_station and bike number
     df['id'] = df.apply(lambda x: ':'.join([str(x['start_date']),␣
      ↪str(x['end_date']), str(x['bike_nr'])]), axis=1)

     # In order for the id to look actually like a unique identifier, let's use␣
      ↪base64 encode to convert the id to a base64 string
     # The command converts the newly created id column into bytes, and then gets␣
      ↪the base64 encoded value for the same.
     # Then the base64 value is converted to string again and then stored in the id␣
      ↪column.
     df['id'] = df['id'].apply(lambda x: base64.b64encode(x.encode()).decode())

     df['id'].unique()
```

1.7 Trip timing details (15 mts)

The second requirement of your client is to create a few additional features. The first of these is trip duration. Now, trip duration is defined as `end time - start time`, so naturally we would consider using those columns. But if you take a look at the start and end date columns, you can see that they are not listed as `datetime` columns; rather, they are just shown as strings. In order to calculate the trip duration, we will have to convert the strings into `datetime` objects. We will be using the `pd.datetime()` function to convert all values of a column into date objects:

```
[ ]: # Then with the remaining values, convert them to datetime objects
     df['start_date'] = pd.to_datetime(df['start_date'], format='%m/%d/%Y %H:%M:%S')
     df['end_date'] = pd.to_datetime(df['end_date'], format='%m/%d/%Y %H:%M:%S')

     # You can now see that the start date and end date are converted to datetime␣
      ↪objects.
     df[['start_date', 'end_date']]
```

With start date and end date being datetime objects now, it is easy to calculate the trip duration for each trip.

```
[ ]: df['trip_duration'] = (df['end_date'] - df['start_date'])

     df['trip_duration'] = df['trip_duration'].apply(lambda x: x.seconds)
```

```
[ ]: # trip_duration column will contain the duration of the trip in seconds
     df['trip_duration']
```

1.7.1 Exercise 5:

Generate the 4 other time-related columns the client wants: `start_weekday`, `start_hour`, `end_weekday`, `end_hour`.

Answer. One possible solution is given below:

```
[ ]: df['start_hour'] = df['start_date'].dt.hour
     df['end_hour'] = df['end_date'].dt.hour
     df['start_weekday'] = df['start_date'].dt.weekday
     df['end_weekday'] = df['end_date'].dt.weekday
```

1.7.2 Exercise 6:

Replicate the process we used to calculate trip duration in order to calculate the age of the registered users.

Answer. One possible solution is given below:

```
[ ]: dob_users = df[df['birth_date'].notnull()]
     current_year = datetime.datetime.now().year
     df.loc[df['birth_date'].notnull(), 'age'] = current_year -␣
      ↪dob_users['birth_date']

     df['age']
```

1.8 Adding stations data (20 mts)

The last thing we need to do is add municipal details as well as coordinates of the start and end stations. To get the latitude and longitude of each station, we will be using the address of the stations present in the JSON file provided along with the CSV data.

The JSON file contains the station ID, station name, and municipal. We will generate the address by concatenating the station and municipal fields and then use the `Nominatim`, which is a geocoding library provided and maintained by OpenStreetMap:

```
[ ]: # With the trip data in place, next we can pull the JSON data of the stations.
     stations_data = []
     with open(os.path.join(local_folder, 'stations.json')) as f:
         stations_data = json.load(f)
```

```
[9]: # For each of the stations, let's get the latitude and longitude for each of␣
      ↪the stations, based on the address
     geolocator = Nominatim(user_agent="Address Predictor", timeout=10)
     for station in stations_data:
         address = station['station'] + ',' +  station['municipal']
         print(address)
```

```
    lat_long = geolocator.geocode(address)
    if lat_long:
        print(lat_long.latitude)
        location = ','.join([str(lat_long.latitude), str(lat_long.longitude)])
    else:
        location = 'None,None'
    station['coordinates'] = location
```

```
        ␣
↪---------------------------------------------------------------------------

    NameError                                 Traceback (most recent call␣
↪last)

    <ipython-input-9-2ad72fa10a46> in <module>
      1 # For each of the stations, let's get the latitude and longitude for␣
↪each of the stations, based on the address
----> 2 geolocator = Nominatim(user_agent="Address Predictor", timeout=10)
      3 for station in stations_data:
      4     address = station['station'] + ',' +  station['municipal']
      5     print(address)


    NameError: name 'Nominatim' is not defined
```

1.8.1 Exercise 7:

`stations_data` now contains the coordinates for each station. Use this to add the start and end coordinates to each trip in the trips data. Add the municipal data for each trip as well.

Answer. One possible solution is given below:

```
[ ]: # Let's create a dictionary with the id of the station as key, to be able to␣
      ↪get the station details for each row
     stations_dict = {}
     for station in stations_data:
         stations_dict[station['id']] = station

     stations_dict
```

```
[ ]: # We can now get the starting coordinates and ending coordinates for each trip␣
      ↪into this DataFrame.
     df['start_coordinates'] = df['start_station'].map(lambda x:␣
      ↪stations_dict[x]['coordinates'] if x != 'Not found' else None)
```

```
df['end_coordinates'] = df['end_station'].map(lambda x:␣
↪stations_dict[x]['coordinates'] if x != 'Not found' else None)

df[['start_coordinates', 'end_coordinates']]
```

[ ]:
```
df['start_municipal'] = df['start_station'].map(lambda x:␣
↪stations_dict[x]['municipal'] if x != 'Not found' else None)
df['end_municipal'] = df['end_station'].map(lambda x:␣
↪stations_dict[x]['municipal'] if x != 'Not found' else None)

df['end_municipal'].describe()
```

1.9 Split the coordinates into separate latitude and longitude columns (5 mts)

We have calculated `start_coordinates` and `end_coordinates` for each trip, but they are currently residing in the same column separated by a comma. The client has asked for them to be in separate columns:

[ ]:
```
# Notice the expand=True parameter. That is required since we are assigning the␣
↪output of the split function to 2 columns
df[['start_latitude', 'start_longitude']] = df['start_coordinates'].str.
↪split(',', expand=True)
df[['end_latitude', 'end_longitude']] = df['end_coordinates'].str.split(',',␣
↪expand=True)
df[['start_latitude', 'start_longitude', 'end_latitude', 'end_longitude']]
```

1.10 Set up Jupyter notebook in an EC2 instance (15 mts)

The next step is to copy the notebook file from the local machine to the EC2 instance. It can be done via an scp command, or using winSCP. There are different ways to SCP (secure copy) the file into the server:

1. If you are using a Linux or a Mac, type the following command to copy the file:

```
scp -i <path_to_pem_file> </path/to/jupyter/notebook> <username>@<EC2IP>:</destination/path>
```

2. If you are using Windows, you can either use WinSCP - https://winscp.net/eng/index.php or if you are using Putty, you can use the pscp command:

```
pscp -i <path_to_ppk_file> <\path\to\jupyter\notebook\> <username>@<EC2IP>:</destination/path>
```

1.11 Execute on the entire dataset in the server (5 mts)

Once the file is copied, you can now access the Jupyter Notebook from `http:server_ip:8889`. Once you are able to access the file, just remove the `nrows=10000` parameter from the `read_csv` function call, in order to read the entire dataset. You can now click on `Cells > Run All` in order to run the same set of steps on the entire dataset.

1.12 Conclusions (5 mts)

In this case, we cleaned up some messy client data and also added some new features based on their requests. We first took a small sample of the overall data so we could work with it locally. However, cleaning up the data was not merely some rote mechanical work. We had to use our common sense and judgment of likely use cases of this data in order to determine how to fix it. Additionally, we had to consider the potential impact of our changes and whether it would adversely impact potential uses of the data in the future, even if those were not likely uses at this time.

After cleaning the data, we uploaded our cleaning scripts to Amazon EC2 so that they could be applied to the entire dataset.

Data wrangling & cleaning is arguably the most important step of the entire data science process, without which we could have wrong or corrupt results. In practice, data scientists can spend upwards of 60 - 70% of their time on cleaning and organizing the data.

1.13 Takeaways (5 mts)

When cleaning data, one very common problem is missing data. We learned a few ways of dealing with missing data and when these methods tend to be appropriate:

1. Removing the rows with missing data entirely is a nuclear option that only makes sense when you have a clear and very limited use case for the dataset.
2. Replacing the missing values with a meaningless filler like ”Not found” or 0 makes sense if you need to preserve the other data in those rows but either cannot sensibly fill in or don’t particularly care about the missing values in that specific column.
3. Interpolating the missing values makes sense when you need to preserve elements of the underlying distribution of the data in that column.

We also saw that for large datasets, working with a small subset of the data is a useful tool for gaining intuition about the dataset and rapidly iterating on the cleaning process. This intuition gathering element cannot be emphasized enough; in fact, many steps in our cleaning process involved elements of exploratory data analysis, via generating summary statistics. Running cleaning steps on the entirety of a very large dataset is time consuming and can bog down the EDA that is essential to cleaning. In future cases, you will see further evidence of EDA in action during wrangling & cleaning.