Analyzing police traffic stop data

Due: Nov 22nd at 4pm

The goal of this assignment is to give you experience using the pandas data analysis library. It will also give you experience using a well-documented third-party library, and navigating its documentation in search of specific features that can help you complete your implementation.

You may work alone or in a pair on this assignment.

North Carolina traffic stop data

Much has been written about the impact of race on police traffic stops. In this assignment, we will examine and analyze trends in traffic stops that occurred in the state of North Carolina from 2000 until 2015. We will not be able to look at every single traffic stop, and will instead look at different subsets of data.

Feel free to continue exploring this data on your own after the assignment. You should have the necessary programming skills to do so! You can find additional clean data from the Stanford Open Policing Project.

Getting started

See these start-up instructions if you intend to work alone.

See these start-up instructions if you intend to work with the same partner as in a previous assignment.

See these start-up instructions if you intend to work in a *NEW* pair.

We have seeded your repository with a directory for this assignment. To pick it up, change to your `capp30121-aut-19-username` directory (where the string `username` should be replaced with your username, or usernames if you are working as a pair) and then run the command: `git pull upstream master`. You should also run `git pull` to make sure your local copy of your repository is in sync with the server.

The `pa6` directory includes a file named `traffic_stops.py`, which you will modify, a file named `pa6_helpers.py` that contains a few useful functions, and a file named `test_traffic_stops.py` with tests.

Please put all of your code for this assignment in `traffic_stops.py`. Do not add extra files and do not modify any files other than `traffic_stops.py`.

The `data/` directory in `pa6` contains a file called `get_files.sh` that will download the data files necessary for this assignment, along with some other files needed by the tests. To download these files, change into the `data/` directory and run this command from the **Linux command-line**:

```
$ sh get_files.sh
```

(Recall that we use `$` to indicate the Linux command-line prompt. You should not include it when you run this command.)

Please note that you must be connected to the network to use this script.

Do not add the data files to your repository! If you wish to use both CSIL & the Linux servers, and your VM, you will need to run the `get_files.sh` script twice: once for CSIL & the Linux servers and once for your VM.

Some of our utility code uses `seaborn`, a plotting library. This library is installed on the machines in CSIL. You will need to install it on your VM using the following command:

```
sudo -H pip3 install seaborn
```

The `sudo` command will ask for a password. Use `uccs` as the password.

We suggest that, as you work through the assignment, you keep an `ipython3` session open to test the functions you implement in `traffic_stops.py`. Run the following commands in `ipython3` to get started:

```
In [1]: %load_ext autoreload

In [2]: %autoreload 2

In [3]: import pandas as pd

In [4]: import numpy as np

In [5]: import traffic_stops as ts
```

We will use `ts` to refer to the `traffic_stops` module in our examples below.

Data

The Stanford Open Policing Project maintains a database of records from traffic stops (i.e., when a police officer pulls a driver over) around the country. We will be working with two different datasets extracted from this database.

The first dataset contains data on traffic stops that occurred in the state of North Carolina. For each stop, the dataset includes information related to the driver (gender, race, age, etc.), the stopping officer (a unique identifier for the officer), and the stop itself (a unique identifier for the stop), the date of the stop, the violation that triggered the stop, if any, etc. More specifically, the records from this dataset include the following fields:

- `stop_id`: a unique identifier of the stop
- `stop_date`: the date of the stop
- `officer_id`: a unique identifier for officers
- `driver_gender`: the driver's gender
- `driver_age`: the driver's age
- `driver_race`: a column that combines information about the driver's race and ethnicity
- `violation`: the violation for which the driver was stopped
- `is_arrested`: a boolean that indicates whether the driver was arrested
- `stop_outcome`: the outcome of a stop (arrest, citation, written warning)

The gender column presumably contains information copied from the binary classification listed on the driver's license, which may or may not match the driver's actual personal gender identity. The race column presumably contains information about what the officer perceived the driver's race to be, which may or may not match the driver's actual personal racial and ethnic identity.

We have constructed three files from this dataset for this assignment:

- The first, `all_stops_basic.csv`, contains a small hand-picked sample of the data and is used in our test code.
- The second, `all_stops_assignment.csv`, contains a random sample of records from 500K stops (out of 10M).
- The third, `all_stops_mini.csv`, contains a random sample of 20 records and will be useful for debugging.

Here, for example, is the data from `all_stops_basic.csv`:

```
stop_id,stop_date,officer_id,driver_gender,driver_age,driver_race,ethnicity,violatic
2168033,2004-05-29,10020,M,53.0,White,N,Registration/plates,False,Written Warning
4922383,2009-09-04,21417,M,22.0,Hispanic,H,Other,False,Citation
924766,2001-08-13,10231,M,38.0,White,N,Other,False,Citation
8559541,2014-05-25,11672,F,19.0,White,N,Other,False,Citation
8639335,2014-07-05,21371,F,76.0,White,N,Other,False,Citation
6198324,2011-04-30,11552,M,35.0,White,N,DUI,True,Arrest
```

```
58220,2000-02-09,,F,42.0,Black,N,Other,False,Citation
5109631,2009-12-23,11941,M,65.0,Black,N,Seat belt,False,Citation
```

Keep in mind that even "clean" data often contains irregularities. You'll notice when you look at these files that some values are missing. For example, the `officer_id` is missing in the eighth line of the file. When you load the data into a dataframe, missing values like these will be represented with `NaN` values.

The second dataset contains information specific to those stops from the first dataset that resulted in a search. Each record in this dataset includes fields for:

- `stop_id`: the stop's unique identifier
- `search_type`: the type of search (e.g., incident to arrest or protective frisk)
- `contraband_found`: indicates whether contraband was found during the search
- `search_basis`: the reason for the search (e.g., erratic behavior or official information)
- `drugs_related_stop`: indicates whether the stop was related to drugs

Here are the first ten lines from `search_conducted_mini.csv`:

```
stop_id,search_type,contraband_found,search_basis,drugs_related_stop
4173323,Probable Cause,False,Observation Suspected Contraband,
996719,Incident to Arrest,True,Observation Suspected Contraband,
5428741,Incident to Arrest,False,Other Official Info,
824895,Incident to Arrest,False,Erratic Suspicious Behaviour,
816393,Protective Frisk,False,Erratic Suspicious Behaviour,
5657242,Incident to Arrest,False,Other Official Info,
4534875,Incident to Arrest,False,Suspicious Movement,
4733445,Incident to Arrest,False,Other Official Info,
1537273,Incident to Arrest,False,Other Official Info,
```

As with the first dataset, some values are missing and will be represented with `NaN` values when you load the data into a dataframe.

Please note that a stop from the first dataset will be represented in this second dataset only if it resulted in a search.

Pandas

You could write the code for this assignment using the `csv` library, lists, dictionaries, and loops. The purpose of this assignment, however, is to help you become more comfortable using `pandas`. As a result, you are required to use `pandas` data frames to store the data and `pandas` methods to do the necessary computations. If you use `pandas` methods efficiently and effectively, functions should be short and will likely use multiple `pandas` methods.

Some of the tasks we will ask you to do require using `pandas` features that have not been covered in class. This is by design: one of the goals of this assignment is for you to learn to read and use API documentation. So, when figuring out these tasks, you are allowed (and, in fact, encouraged) to look at the Pandas documentation. Not just that, any code you find in the Pandas documentation can be incorporated into your code without attribution. (For your own convenience, though, we encourage you to include citations for any code you get from the documentation that is more than one or two lines.) If, however, you find Pandas examples elsewhere on the Internet, and use that code either directly or as inspiration, you *must* include a code comment specifying its origin.

When solving the tasks in this assignment, you should assume that you can use a series of Pandas operations to perform the required computations. Before trying to implement anything, you should spend some time trying to find the right methods for the task. We also encourage you to experiment with them in `ipython3` before you incorporate them into your code.

Our implementation used filtering and vector operations, as well as methods like `agg`, `apply`, `cut`, `to_datetime`, `fillna`, `groupby`, `isin`, `loc`, `merge`, `read_csv`, `rename`, `size`, `transform`, `unstack`, `np.mean`, `np.where`, along with a small number of lists and loops. Do not worry if you are not using all of these methods!

Your tasks

Task 1: Reading in CSV files

Before we analyze our data, we must read it in. Often, we also need to process the data to make it analysis-ready. It is usually good practice to define a function to read and process your data. In this task, you will complete two such functions, one for each type of data.

You may find `pd.read_csv`, `pd.to_datetime`, `pd.cut`, and `np.where` along with dataframe methods, such as `fillna` and `isin`, useful for Tasks 1a and 1b.

Task 1a: Building a dataframe from the stops CSV files

Your first task is to complete the function `read_and_process_allstops` in `traffic_stops.py`. This function takes the name of a CSV file that pertains to the `all_stops` dataset and should return a `pandas` dataframe, if the file exists.

If the file does not exist, your function should return `None`. (You can use the library function `os.path.exists` to determine whether a file exists or a `try` block (see R&S Exceptions) that returns `None` when the file cannot be opened.

Note about reading the data

The pandas `read_csv` function allows you to read a CSV file into a dataframe. When you use this function, it is good practice to specify data types for the columns. You can do so by specifying a dictionary that maps column names to types using the `dtypes` parameter. The set of types available for this purpose is a little primitive. In particular, you can specify `str`, `int`, `float`, and `bool` (or their `np` equivalents) as initial column types. In some cases, you will need to adjust the types after you read in the data.

For this assignment (and in general), you should be very thoughtful about how you specify column data types. Here are a few guidelines to consider:

- A number that can begin with `0` should be read as a string so that any leading zeros are preserved.
- Be mindful about how you import columns that contain missing values. There is no integer representation of `NaN`.

The `officer_id` column, for example, starts with a zero in some cases and has a missing value in others.

Here's a sample use of `pd.read_csv` that uses the `dtype` keyword parameter:

```
col_types = {'col1_name': str, 'col2_name': int}
df = pd.read_csv("some_file.csv", dtype=col_types)
```

While it is possible to specify a column to use as a row index when calling `pd.read_csv`, for this assignment we will be using the default range index for rows.

Data preparation

In addition to reading the data into a dataframe, your `read_and_process_allstops` function should prepare the data for analysis by:

1. Converting the type of the `stop_date` column to `datetime` using `pd.to_datetime`.
2. Adding new columns for the year (`stop_year`) and the month (`stop_month`) of the stop.
3. Adding a column for season (`stop_season`) as defined by:
    - December, January, February: `"winter"`
    - March, April, May: `"spring"`
    - June, July, August: `"summer"`
    - September, October, November: `"fall"`
4. Converting all ages to discretized categories ('age_category') as defined by:
    - (0, 21]: `"juvenile"`

    - (21 - 36]: `"young_adult"`
    - (36 - 50]: `"adult"`
    - (50 - 65]: `"middle_aged"`
    - (65 - 100]: `"senior"`
5. Adding a boolean new column, named `arrest_or_citation`, for whether the stop ended in an arrest or citation. (This type of variable is often called a *dummy* variable in social sciences.)
6. Converting missing values in the `officer_id` column to `"UNKNOWN"`
7. Converting the columns listed in `CATEGORICAL_COLS` from strings to categoricals.

At the top of `traffic_stops.py` you will find several constants we have defined for many (but not all) of the values that you will need to do this task: the column names, the age brackets and labels, the mapping of months to seasons, etc. Make sure to use these constants!

We suggest that you start by trying code in your `ipython3` session. A good starting point is to load either the "basic" or the "mini" dataset using `read_csv`:

```
In [6]: df = pd.read_csv("data/all_stops_basic.csv")
```

If you print out the names of the columns and the first few rows of the dataframe itself, you should see that it contains the same data as in the CSV file:

```
In [7]: df.columns
Out[7]:
Index(['stop_id', 'stop_date', 'officer_id', 'driver_gender', 'driver_age',
       'driver_race', 'ethnicity', 'violation', 'is_arrested', 'stop_outcome'],
      dtype='object')

In [8]: df.head()
Out[8]:
   stop_id   stop_date      ...         is_arrested     stop_outcome
   0  2168033  2004-05-29      ...                False  Written Warning
   1  4922383  2009-09-04      ...                False         Citation
   2   924766  2001-08-13      ...                False         Citation
   3  8559541  2014-05-25      ...                False         Citation
   4  8639335  2014-07-05      ...                False         Citation


 [5 rows x 10 columns]
```

Here is a sample use of `read_and_process_allstops`:

```
In [9]: basic_df = ts.read_and_process_allstops('data/all_stops_basic.csv')

In [10]: basic_df.head()
Out[10]:
   stop_id  stop_date officer_id driver_gender       ...         stop_year stop_mc
   0  2168033 2004-05-29      10020             M       ...              2004
   1  4922383 2009-09-04      21417             M       ...              2009
   2   924766 2001-08-13      10231             M       ...              2001
   3  8559541 2014-05-25      11672             F       ...              2014
   4  8639335 2014-07-05      21371             F       ...              2014

   [5 rows x 15 columns]
```

Once you believe that your `read_and_process_allstops` function is in good shape, you can try out the automated tests that we have provided to check your function.

Tests for `read_and_process_allstops`

Test	Filename	Purpose
1	data/all_stops_basic.csv	Check column names and types
2	data/all_stops_basic.csv	Check seasons set properly
3	data/all_stops_basic.csv	Check age category set properly

Test	Filename	Purpose
4	data/all_stops_basic.csv	Check missing officer IDs handled properly
5	data/all_stops_basic.csv	Check arrest_citation dummy variable
6	data/all_stops_basic.csv	Check full dataframe
7	data/all_stops_mini.csv	Check full dataframe
8	data/all_stops_assignment.csv	Check full dataframe
9	bad_file_name.csv	Check missing file

You can run these tests using the Linux command:

```
$ py.test -x -v -k allstops
```

Recall that the `-x` flag indicates that `py.test` should stop as soon as one test fails. The `-v` flag indicates that you want the output in verbose form. And finally, the `-k` flag allows you to limit the set of tests that is run. For example, using `-k allstops` will only run the tests that include `allstops` in their names.

Remember that running the automated tests should come towards the end of your development process! Debugging your code based on the output of the tests will be challenging, so make sure you've done plenty of manual testing using `ipython3` first. We encourage you to do quick sanity checks and manual testing as you work through the assignment.

Task 1b: Reading in the searches CSV files

The second part of your Task 1 is to read and process the `search_conducted` files in the `read_and_process_searches` function. You will use this data for Task 5.

Your function should take a filename and a fill dictionary and return a pandas dataframe if the specified file exists and `None` if the specified file does not exist. A fill dictionary maps column names to values to use in place of missing values (`NaN`) for those columns. For example, the following dictionary:

```
{'drugs_related_stop': False,
 'search_basis': "UNKNOWN"}
```

indicates that values missing from the `drugs_related_stop` column should be replaced by `False` and values missing from the `search_basis` column should be replaced with the string `"UNKNOWN"`.

Unlike Task 1a, you do *not* have to convert any of the columns to categoricals.

Here is a sample use of this function:

```
In [11]: searches = ts.read_and_process_searches(
    ...:                'data/search_conducted_mini.csv',
    ...:                {'drugs_related_stop': False,
    ...:                 'search_basis': 'UNKNOWN'})
Out[11]:

In [12]: searches.head()
   stop_id         search_type  contraband_found                      search_basis
0  4173323      Probable Cause             False  Observation Suspected Contraband
1   996719  Incident to Arrest              True  Observation Suspected Contraband
2  5428741  Incident to Arrest             False               Other Official Info
3   824895  Incident to Arrest             False        Erratic Suspicious Behaviour
4   816393     Protective Frisk             False        Erratic Suspicious Behaviour
```

As with Task 1a, we have provided automated tests for your implementation of `read_and_process_searches`.

Tests for `read_and_process_searches`

Test	Filename	Purpose
1	data/search_conducted_mini.csv	Check small file with no fill

Test	Filename	Purpose
2	data/search_conducted_mini.csv	Check small file, with fill value for the boolean column (`'drug_related_stop'`) that has missing values
3	data/search_conducted_mini.csv	Check small file, with fill value for the string column (`'search_basic'`) that has missing values
4	data/search_conducted_mini.csv	Check small file, with fill values for multiple columns
5	bad_file_name.csv	Test missing file
6	data/search_conducted_assignment.csv	Check large file

To run these tests, use the following command on the Linux command line:

```
$ py.test -x -v -k searches
```

Task 2: Creating filter functions

Often when working with large datasets, you will want to filter for rows that have specific properties. For example, you might want all rows that contain information on stops of men between the ages of 15 and 30 whose race is recorded as `Black` or `Hispanic`. Your next task is to complete two functions, `apply_val_filters` and `apply_range_filters` that will do the heavy lifting of this type of filtering for you.

For both cases, the functions will take a dataframe and some filter information as inputs, and return a filtered dataframe in which all rows satisfy the filter requirements. Given a filter for a bad column (i.e. a column that does not exist in the dataframe), these functions should return `None`.

You should begin each function by experimenting with different pandas methods on the small dataframes in your `ipython3` session before you start writing code. We strongly suggest that you begin with the small data files. It is difficult to check that a filter is working properly on a large dataset; if you start with smaller files, you can manually verify that a filter performs as expected.

Task 2a

The `apply_val_filters` function should take a dataframe and a value filter and return a filtered dataframe. A value filter is a dictionary that maps column names to lists of values.

For example, we could represent the first two parts of our above example using the following dictionary:

```
{ts.DRIVER_RACE: ['Black', 'Hispanic'],
 ts.DRIVER_GENDER: ['M']}
```

Given this filter, a row should be included in the result only if the value of the `driver_race` field is either `'Black'` or `'Hispanic'` *and* the value of the `driver_gender` field is `'M'`. The values are case-sensitive, so we will consider `'Hispanic'` to be different from `'hispanic'` or `'HISPANIC'`.

Below is a use of this function that uses the basic all stops dataset and the sample value filter shown above. The result has only two rows:

```
 In [13]: basic_df = ts.read_and_process_allstops("data/all_stops_basic.csv")
 In [14]: val_filter = {ts.DRIVER_RACE: ['Black', 'Hispanic'],
     ...:               ts.DRIVER_GENDER: ['M']}
 In [15]: filtered_df = ts.apply_val_filters(basic_df, val_filter)

 In [16]: filtered_df
 Out[16]:
     stop_id  stop_date officer_id driver_gender        ...          stop_year stop_
 1   4922383 2009-09-04      21417             M        ...               2009
 7   5109631 2009-12-23      11941             M        ...               2009
```

```
[2 rows x 15 columns]
```

Keep in mind that it is possible for a valid value filter to yield a dataframe with zero rows. Also, it is OK to do multiple filter operations. You do *not* need to figure out how to combine the different value filters into a single Pandas filter operation.

Also, recall that calling `apply_val_filter` with a value filter that includes a bad column name, such as:

```
{'race': ['Black', 'Hispanic'],
 'gender': ['M']}
```

should yield the value `None`.

Once you have tested your functions by hand, you can run our tests using the Linux command:

```
$ py.test -x -v -k val
```

Most of the tests use the basic all stops dataset and test for different cases: a filter for one column, a filter with more than one column, an empty filter, a filter that yields the whole dataframe, a filter that yields an empty dataframe, and a filter with bad column names. We also run a test using the full all stops dataset.

For this and the following tasks, if you fail a particular test, you can infer the specific details of the failed test from the error message. For instance, here is part of the output for a failed test:

```
=================================== FAILURES ===================================
___________________________ test_apply_val_filters_1 ___________________________

    def test_apply_val_filters_1():
        ''' Purpose: test one value filter '''
        helper_apply_filters(ts.apply_val_filters,
                             "data/all_stops_basic_expected.gzip",
                             "data/all_stops_basic_gender_only.gzip",
>                            {"driver_gender":["M"]})
```

Observe that in this output, there is a purpose for the test function that gives a brief summary of the test. Within the code shown, there is a call to a helper function that is part of the tests, but one of the parameters is the function you are writing, `ts.apply_val_filters`. Further, the specific dictionary that is being used for the filtering is visible later: `{"driver_gender":["M"]}`. Although internal testing data files are referenced, we can still deduct that this particular test is using the `basic` data set. With this information, we can then run our function with the same filter in an `ipython3` session and investigate the failure.

Task 2b

The `apply_range_filters` function should take a data frame and a range filter and return a filtered dataframe. A range filter maps column names to a lower and upper bound (inclusive) for the column. As a reminder, calling this function with a range filter that includes a bad column name should yield `None`.

As with `apply_value_filters`, it is OK to do multiple filter operations.

Here's sample use of the `apply_range_filters` function:

```
In [17]: range_filter = {ts.DRIVER_AGE: (15, 30)}
In [18]: range_df = ts.apply_range_filters(basic_df, range_filter)
In [19]: range_df
Out[19]:
   stop_id  stop_date officer_id driver_gender        ...         stop_year stop_mo
1  4922383 2009-09-04      21417             M        ...              2009
3  8559541 2014-05-25      11672             F        ...              2014

[2 rows x 15 columns]
```

Note that we can combine calls to the two filter functions to get a dataframe with rows that contain information on stops of men between the ages of 20 and 30 whose race is recorded as `Black` or `Hispanic`

```
In [20]: val_filter = {ts.DRIVER_RACE: ['Black', 'Hispanic'],
    ...:               ts.DRIVER_GENDER: ['M']}
In [21]: filtered_df = ts.apply_val_filters(basic_df, val_filter)

In [22]: range_filter = {ts.DRIVER_AGE: (15, 30)}
In [23]: combined_df = ts.apply_range_filters(filtered_df, range_filter)

In [24]: combined_df
Out[24]:
   stop_id  stop_date officer_id driver_gender  driver_age        ...         age_ca
1  4922383 2009-09-04      21417             M        22.0        ...          young

[1 rows x 15 columns]
```

Keep in mind that it is possible for a valid range filter to yield a dataframe with zero rows.

Once you have tested your functions by hand, you can run our basic tests using the Linux command:

```
$ py.test -x -v -k range
```

The tests are similar to the previous task except they use range filters rather than value filters.

Task 3: Producing a dataframe of aggregations

Looking for differences among subgroups is a common data analysis task. Generally, we might be interested in what differences exist in our data by race, age, type of stop, or any other variable. For example, we might want to know if the median age of Asian drivers who get stopped is lower or higher than that of white drivers. In this task, you will implement the function `get_summary_statistics` that will allow you to specify a set of grouping characteristics and get back summary statistics for each group.

Specifically, `get_summary_statistics` should take a pandas dataframe, a list of column names to group by, and a column to summarize (this column must be numeric; we've defaulted to age) as parameters and return a dataframe that contains the columns that define the subgroups along with the median, mean, and difference from the global mean (in that order) for each subgroup.

It will be somewhat simple to find the median and the mean for this function. We are also asking you to compute the difference from the global mean as a custom aggregation. We will define this statistic as the difference between a group's mean age and the mean age of every single person in the data set. Such a statistic allows us to quickly see how the average age of a group member compares to the average age overall in our dataset.

To test that you have computed the `mean_diff` correctly, you should manually compare your calculated group means to the mean across the dataframe. For example:

```
In [24]: all_stops = ts.read_and_process_allstops("data/all_stops_assignment.csv")

In [25]: all_stops[ts.DRIVER_AGE].mean()
Out[25]: 35.832132
```

This result tells us that if the mean age for Asian drivers is `34.306827`, then the mean difference will be `34.306827 - 35.832132`, or about `-1.525305`. In other words, the average Asian driver who is stopped is about 1.5 years younger than the average general driver who is stopped.

Here are some sample uses of this function:

```
In [26]: ts.get_summary_statistics(all_stops, [ts.DRIVER_RACE])
Out[26]:
             median       mean  mean_diff
driver_race
Asian          32.0  34.306827  -1.525305
Black          34.0  35.911102   0.078970
```

```
Hispanic       29.0  31.071711  -4.760421
Other          31.0  33.402386  -2.429746
White          34.0  36.408409   0.576277


In [27]: ts.get_summary_statistics(all_stops,
    ...:                            [ts.DRIVER_RACE, ts.DRIVER_GENDER])
Out[27]:
                          median       mean  mean_diff
driver_race driver_gender
Asian       F               31.0  33.793215  -2.038917
            M               32.0  34.537124  -1.295008
Black       F               32.0  34.378343  -1.453789
            M               35.0  36.786237   0.954105
Hispanic    F               30.0  31.496343  -4.335789
            M               29.0  30.970611  -4.861521
Other       F               30.0  32.148493  -3.683639
            M               31.0  33.976765  -1.855367
White       F               32.0  35.017182  -0.814950
            M               35.0  37.108272   1.276140
```

Notice that the output in the second example has a hierarchical index, which is the natural result of using `groupby` with a list of columns.

You can run our tests on your function by running the Linux command:

```
$ py.test -x -v -k summary
```

The first two tests use the basic all stops dataframe with `[ts.DRIVER_RACE]` and `[ts.DRIVER_RACE, ts.DRIVER_GENDER]` for grouping respectively. The second pair of tests uses the full all stops dataframe with the same two groupings. The final tests check corner cases for the grouping columns: a bad column name and no columns.

Task 4: Getting dataframe shares by group

We'd like to be able to answer questions like (1) how does the fraction of stops that result in arrests or citations differ by race? and (2) are women or men in different age categories more likely to be arrested when stopped? In order to answer such questions, we need some way to compute and compare rates across different groupings in our data.

Your fourth task, to complete the function `get_rates`, is designed to help answer these types of questions. This function takes a dataframe, a list of column names to group by, and the outcome column of interest and returns a dataframe of the share of each group for each possible outcome. Note that if there are no instances of the given outcome for a subgroup, the dataframe should have a `0` for that subgroup (rather than `NaN`)

Here is an example use of this function:

```
In [28]: ts.get_rates(all_stops, [ts.STOP_SEASON], ts.ARREST_CITATION)
Out[28]:
arrest_or_citation     False     True
stop_season
fall                0.196276  0.803724
spring              0.190865  0.809135
summer              0.187372  0.812628
winter              0.212569  0.787431
```

In this call, we have found the share by season of all stops that resulted or did not result in an arrest or citation. For example, of all stops that occurred in the summer, 81.27% ended with an arrest or citation.

Here is another use of this function:

```
In [29]: ts.get_rates(all_stops,
    ...:              [ts.AGE_CAT, ts.DRIVER_GENDER],
    ...:              ts.IS_ARRESTED)
```

```
Out[29]:
is_arrested                    False     True
age_category driver_gender
juvenile     F              0.994462  0.005538
             M              0.984669  0.015331
young_adult  F              0.990855  0.009145
             M              0.979265  0.020735
adult        F              0.991970  0.008030
             M              0.983174  0.016826
middle_aged  F              0.995749  0.004251
             M              0.989827  0.010173
senior       F              0.998697  0.001303
             M              0.995823  0.004177
```

Since this information can be difficult to read, we have provided a function (`pa6_helper.visualize_rate_series`) to help you visualize it. To use the function, you will need to select which outcome to plot (here, this would be the `True` rate or the `False` rate). This function takes two arguments: a rate series you want to visualize, and an output filename.

Here is a sample use of these two functions:

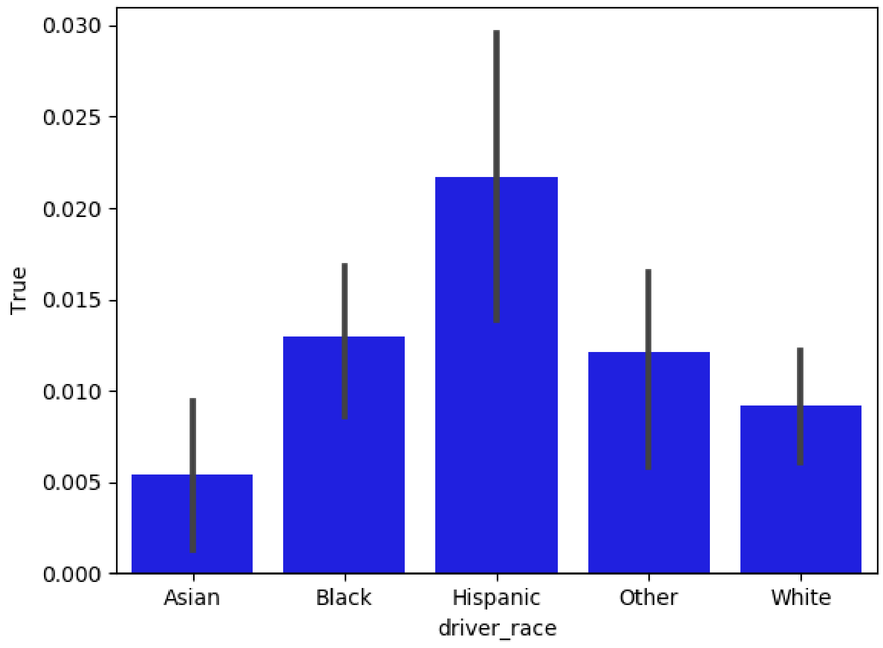
```
In [30]: import pa6_helper

In [31]: example_frame = ts.get_rates(all_stops,
    ...:                              [ts.DRIVER_RACE],
    ...:                              ts.IS_ARRESTED)

In [32]: example_series = example_frame[True]

In [33]: pa6_helper.visualize_rate_series(example_series,
    ...:                                  filename='barplot.png')
```

The function will store a bar plot in the specified file. In this case, the bar plot looks like:



The function `get_rates` is short, but requires an understanding of how to use `groupby` and how to work with hierarchical indexes. We suggest that you think through the steps necessary to convert counts to shares before you begin to code.

We have provided automated tests for this function. You can run them using the command:

```
$ py.test -x -v -k rates
```

The first couple of tests use the basic all stops data set with a couple of different single category groups (`['age_category']` and `[ts.STOP_SEASON]`) with `ts.ARREST_CITATION` as the outcome column. The third test uses a single category group `[ts.STOP_SEASON]` on the full all stops data set with `ts.IS_ARRESTED` as the outcome column. The fourth test uses the full all stops data set with two columns. The fifth test uses a bad group column name and the sixth test uses a bad outcome column name. In both cases, the expected result is `None`.

Task 5: Determining officers most likely to conduct searches

Often, you may need to join datasets in order to answer questions that draw on multiple data sources. For example, we may want to answer, "what share of drivers by race are searched when they are stopped?" Since we do not have driver race in our `searches` dataset, for example, we would need to merge our stops and our searches datasets in order to answer this question. This task will give you practice merging two datasets and then analyzing the result.

More specifically, you will answer, "which officer IDs have the highest search rate for their stops?" Keep in mind that since our stops represent a random subset of all North Carolina traffic stop data, our answers here may not be representative by officer ID of the whole dataset. In order to complete this task, you will need to merge dataframes, add a dummy column for whether a search was conducted, filter the resulting dataframe, determine rates using Task 4, and sort the result.

You will complete the function `compute_search_share`, which takes in your stops dataframe, your searches dataframe, and a threshold `min_stops` that is described below, and returns a dataframe (or `None`; see below).

You will merge the stops dataframe (left) with the search dataframe (right) using the `stop_id` as the merge key. The merged data should include a row for *every* stop, which means that you will need to do a *left* merge. Since the searches dataframe does not include a row for every stop, the merge result will be missing some values (which will be represented with `NaN`). You should replace these `NaN` values in the `search_conducted` column of the merged data frame with `False`.

How can you check that you have merged dataframes correctly? This is very tricky, since not every record in `searches` has a corresponding record in `all_stops`. You should start by tracing specific records from the merged dataframe, making sure the correct information is included (from the `all_stops` and the `searches` dataframes). Next, you may want to identify if any `stop_id` from `all_stops` does not appear in your final dataframe. Finally, you should ensure that no new `stop_id` values from searches appear in the merged dataframe. You may find the `sample()` and `isin` methods particularly useful when checking your results.

Next, you will add a dummy column for whether a search was conducted (`True`) or not. Be careful here! There are columns that may have unexpected missing values.

The remaining three steps of this task are simple: filter out all officer IDs with fewer than M stops, find the rates of searches conducted, and then return a sorted dataframe. However, if no officers meet the criterion of having at least the minimum number of stops, return `None` instead.

In sum, your function will:

1. Left merge the two datasets
2. Add a dummy column for whether a search was conducted or not
3. Drop officers who have fewer than `min_stops` stops in the dataset
4. If, after doing so, there are no rows left, return `None` immediately
5. Determine the rates (using Task 4) of your desired variable
6. Return a dataframe sorted in *non-increasing* order by `True` rate

Here is a sample use of this function:

```
In [32]: all_stops = ts.read_and_process_allstops("data/all_stops_assignment.csv")

In [33]: searches = ts.read_and_process_searches('data/search_conducted_assignment.c
    ...:                                         {'drugs_related_stop': False,
```