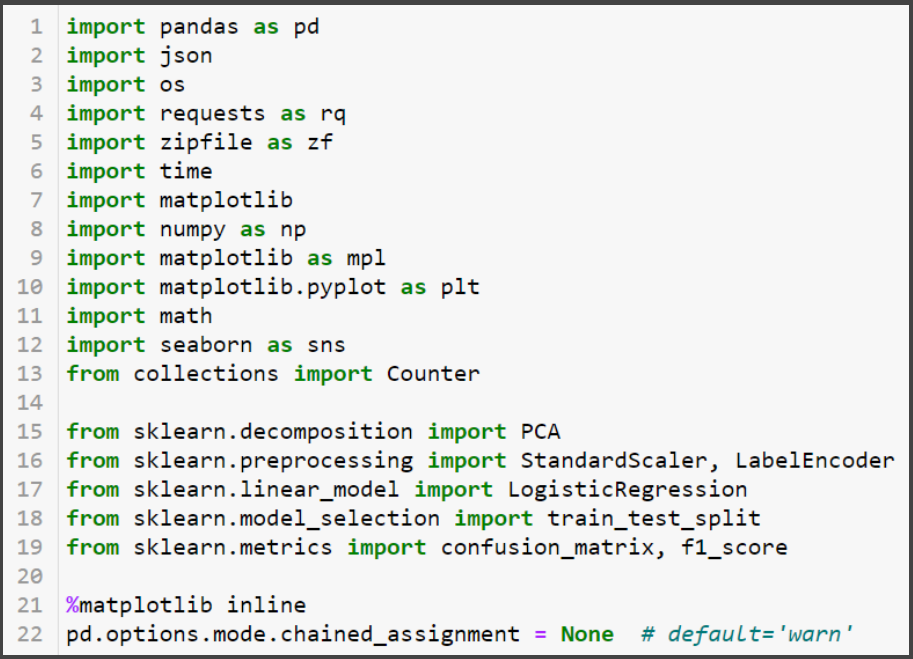
FBI Crime Data - Data Science Project - Readme

Made by: Yonatan Avizov, Asaf Dangoor

The following is the readme file for our project. In this document, you will find the "flow" of the project itself along with function explanations as to what they were given and what do they do. Along with individual cell explanations for cells that execute the functions themselves.

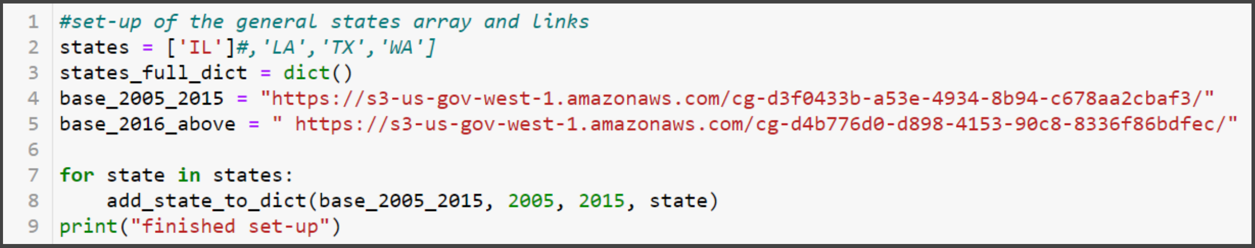
IMPORTS:



```python
import pandas as pd
import json
import os
import requests as rq
import zipfile as zf
import time
import matplotlib
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import math
import seaborn as sns
from collections import Counter

from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, f1_score

%matplotlib inline
pd.options.mode.chained_assignment = None  # default='warn'
```

Other than the usual imports for a data science project, we needed the zipfile, json, time, math imports to help us navigate through the data manipulation and downloads (since each download we had for the data was a zip file we needed to extract).

Downloading the Files:



```python
#set-up of the general states array and links
states = ['IL']#,'LA','TX','WA']
states_full_dict = dict()
base_2005_2015 = "https://s3-us-gov-west-1.amazonaws.com/cg-d3f0433b-a53e-4934-8b94-c678aa2cbaf3/"
base_2016_above = " https://s3-us-gov-west-1.amazonaws.com/cg-d4b776d0-d898-4153-90c8-8336f86bdfec/"

for state in states:
    add_state_to_dict(base_2005_2015, 2005, 2015, state)
print("finished set-up")
```

In this cell we had a setup of a dictionary of dictionaries within in. Each item had the Key of the state index (IE like TX for Texas) and the value would be a dictionary of links for each year we wanted, alongside the state itself.

add_state_to_dict(base, year_bot, year_top, state_code)

Properties:
- base - The base URL for the download that we need to build upon.
- year_bot - The first year we need to take when we are going to download.
- Year_top- The last year we need to take when we are going to download.
- State_code - the index of the state itself, IE TX as in Texas.

Method:
- The method itself creates a temporary dictionary that will hold the built links.
- Once it iterates in the loop from all the base URL and years, it puts it in the global dictionary of the links.

We then had a cell that would iterate over the states list and request for different functions to start downloading from the Crime Stats API slowly (as to not get blocked by our request).

download_state_files(state_code)

Properties:
- State_code - the index of the state itself, IE TX as in Texas.

Method:
- Loops through the inner dictionary of the global links dictionary (states_full_dict). It then takes the key and the value inside it. The key would symbolize the file name, while the value is the download link itself.
- It would make sure that there is a download, and if there is none, it would remove the link from the dictionary as to be more precise.
- It would then sleep for a second before doing the next request.
- The request (once we loop) calls for the download_zip function.

download_zip(url, filename)

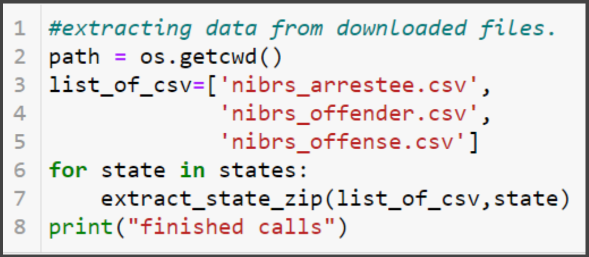
Properties:

- Url - the url path of the download that is needed.
- Filename - the name of the file we will be getting (IE TX-2010.zip)

Method:

- The method itself would do a requests.get function call for the URL we got for the method.
- If the status code would be anything but 200 (AKA a success) then we return a message declaring it was not successful. And then we returned a False boolean.
- If the status code was 200 then we'd commence the download, and open the file, requesting the specific file we wanted (AKA a TX-2010.zip)
- We'd then save it on the PC where the project is saved.

Extracting the Files:



```
#extracting data from downloaded files.
path = os.getcwd()
list_of_csv=['nibrs_arrestee.csv',
             'nibrs_offender.csv',
             'nibrs_offense.csv']
for state in states:
    extract_state_zip(list_of_csv,state)
print("finished calls")
```

This cell, after we downloaded the needed files, goes through the current saved project's path, and along with it, loops through the states list, requesting to extract the state data zip. Of course, we have a list_of_csv as a list of needed files from each .zip.

extract_state_zip(list_of_csv,state)

Properties:

- List_of_csv - a List of CSV file names that need to be extracted from the .zip
- State - the state in which we need to run, normally like TX or IL.

Method:

- The method would iterate over the global dictionary of download links (state_full_dict), more specifically the inner dictionary that the state itself is holding. It would then go through the keys of the dictionary (as they are actually file names) and build a path.
- It would call the create_path function to make a path to extract the data to.
- Once the path is made, it would loop through the list_of_csv and extract the individual files from the zip itself to the given path. It would utilize the find method to see if it exists.
- In case the file doesn't exist or there is an issue with the path, it would declare as such.

find(name, path)

Properties:

- Name - the filename
- Path - the path where the file would be.

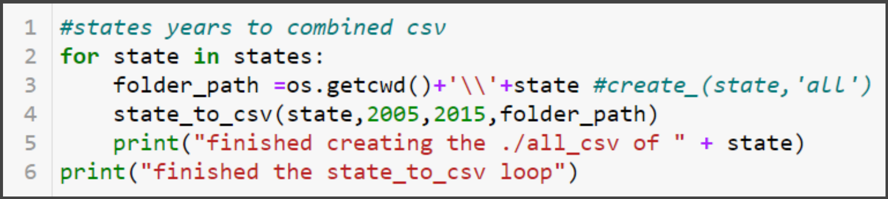
Method:
- Loops through the current folder and checks to see if the file exists while using three temporary variables. Root would be the root of the folder itself, of the project. Dirs would be a list of directories in the folder. Files would be the files currently in the folder.
- It would check to see if the file is in the files list. If it is it would return the path itself.

create_path(state,add_str)

Properties:
- Add_str - the additive path we need to make folders to.
- State - the state in which we need to run, normally like TX or IL.

Method:
- The method creates a string via formatting that is the directory we want to create.
- With the help of try-catch we attempt to create a directory, assuming it was not created before.
- If it was, we generate the correct response.
- We then return the path we generated.

Combining the extracted CSV files into one full csv file:



```python
#states years to combined csv
for state in states:
    folder_path =os.getcwd()+'\\'+state #create_(state,'all')
    state_to_csv(state,2005,2015,folder_path)
    print("finished creating the ./all_csv of " + state)
print("finished the state_to_csv loop")
```

This cell loops through the states list taking each state, and taking the folder path of the state (as all saved in logical names) and sends it to the state_to_csv method. This would create a new csv file that combines the three priorly extracted files, and places them in a “all” folder, which in the end holds files like TX_2005.csv.

state_to_csv(state,year_min,year_max,state_path)

Properties:
- State - the state in which we need to run, normally like TX or IL.
- Year_min - the smallest year we will handle (normally 2005).
- Year_max - the largest year we will handle (normally 2015).
- state_ path - the path where the file would be.

Method:
- We get the needed properties for the function, and basically right off the bat, we create a lot of variables that will help us with the adjustments of the dataframes we will be

using. For instance we have arstee_clean which is the list we will use for the .drop method in our dataframe.

- We then have arstee_thresh_hold which we will use for the adjust_dataframe method.
- We then create the “all” folder with the create_path function from before.
- We then make a list called path that holds all the file paths we want to create a combo csv from (IE ...\\IL\\2006\\...)
- We then loop through the list (path) and for each item in it we do the following:
    - We create cp which holds the name of the file path with the file.
    - We then call adjust_dataframe for offense.csv and offender.csv
    - if flag_offender is False or flag_offense is False
        - This is a check in case the folder it accesses does not have any files in it that can make a combined csv that we want. If that is the case, advance the year we want to use by one and continue without combining any information.
    - We then call combined_data_frame for the offender and offense csvs
    - We then call adjust_dataframe for the arestee.csv
    - If the flag it lifsts is false then we adjust the year, and continue (skip it).
    - We then call combined_data_frame for arestee datafram and the 1st combined dataframe we called before.
    - NOW WE HAVE COMBINED ALL THREE DATAFRAMES (CSVS)
    - We then save the file with the correct path name, and drop any duplicated values.
- At the end we return the “all” path.

adjust_dataframe(directory_string, df_threshold, df_clean_array, df_fill_flag, df_fill_array, index)

Properties:

- directory_string - the path of the csv itself.
- Df_threshold - the minimum empty dataspace we want for the dropna command.
- Df_clean_array - a list of column names we intend to use for the drop command.
- Df_fill_flag - in case we need to do fillna command once we get the dataframe.
- Df_fill_array - A dictionary of items for each column we need ot adjust data with by fillna.
- Index - The column we will use for the dataframe index.

Method:

- Gets the directory of a csv, then calls load_csv to load it into a dataframe.
- If the dataframe is Empty it will return None and a false flag indication that there was a issue.
- We then call clean_csv to clean the dataframe.
- We then set the index of the dataframe.
- We then return the dataframe alongside with True for the flag itself.

load_csv(file_dir)

Properties:

- file_dir - the path of the csv itself.

Method:

- Attempts to create a dataframe from a csv, if there is no csv it will make a flag and return nothing.
- If it manages, then it returns the dataframe.

clean_csv(df_csv, thresh_num, clean_colums, is_fill, fill_colums)

Properties:

- df_csv - the dataframe itself
- thres_num - the threshold for the dropna how
- clean_colums - the list of columns we need to drop from the dataframe.
- Is_fill - flag to see if we need to commence a fillna protocol.
- Fill_colums - a list of columns to go through and fill with the fillna command

Method:

- Makes a copy of the csv and then drops the columns that are not needed with the drop command.
- Removes any na values in the incident_id column (since it is a critical section of our code we can’t afford empty spaces).
- Proceeds to then do dropna in case there are any outliers with the given threshold. Ont the row axis of course.
- If the is_fill is True then it proceeds to fill the dataframe.
- Returns the fixed dataframe.

combined_data_frame(data1,data2)

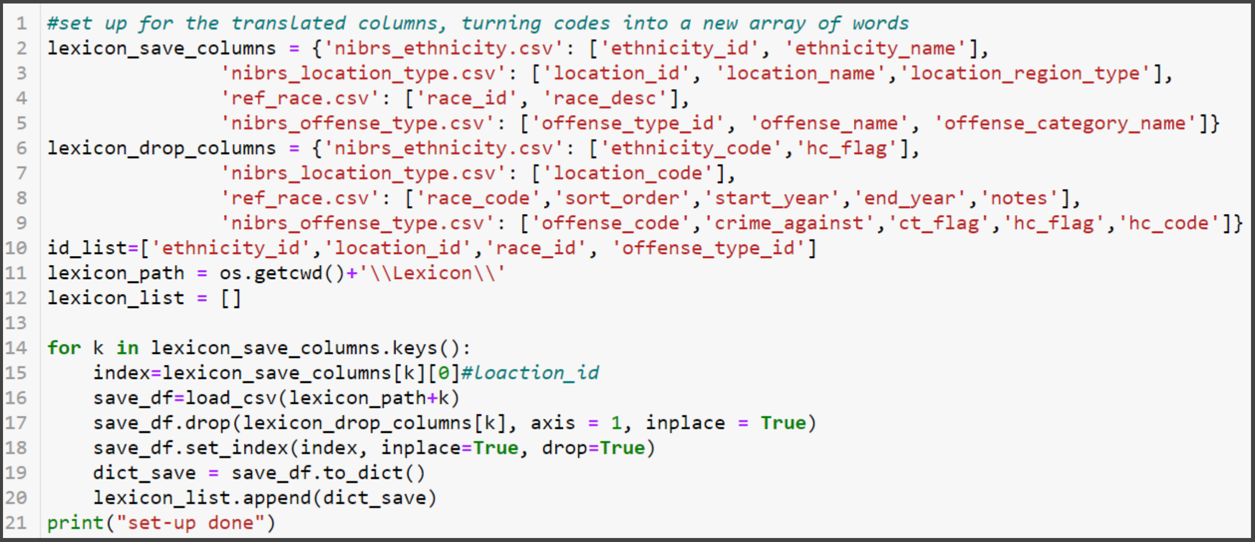
Properties:

- Data1 - a dataframe
- Data2 - another dataframe

Method:

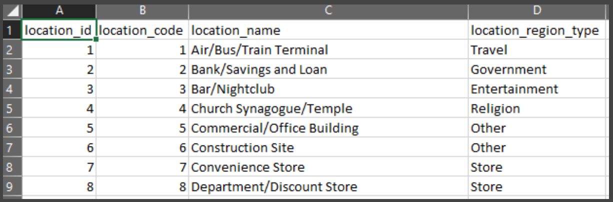
- Does the merge command for the two dataframes (as they are already fixed with their index)
- Does dropna for any missing elements.
- Returns the merged dataframe.

Lexicon Setup:



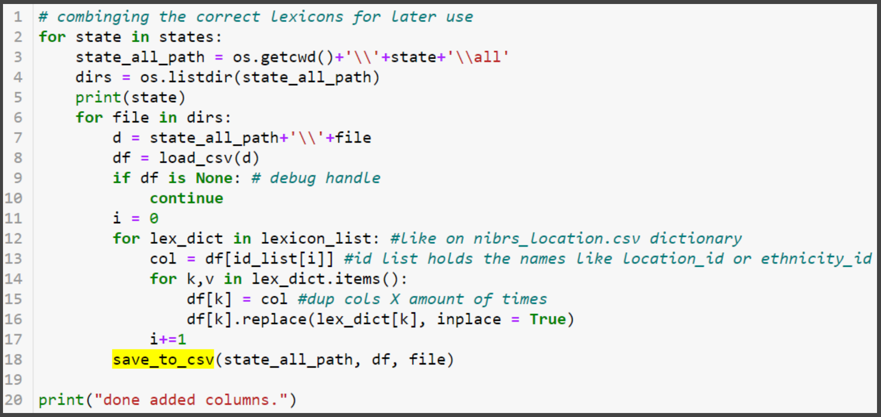
```python
#set up for the translated columns, turning codes into a new array of words
lexicon_save_columns = {'nibrs_ethnicity.csv': ['ethnicity_id', 'ethnicity_name'],
               'nibrs_location_type.csv': ['location_id', 'location_name','location_region_type'],
               'ref_race.csv': ['race_id', 'race_desc'],
               'nibrs_offense_type.csv': ['offense_type_id', 'offense_name', 'offense_category_name']}
lexicon_drop_columns = {'nibrs_ethnicity.csv': ['ethnicity_code','hc_flag'],
               'nibrs_location_type.csv': ['location_code'],
               'ref_race.csv': ['race_code','sort_order','start_year','end_year','notes'],
               'nibrs_offense_type.csv': ['offense_code','crime_against','ct_flag','hc_flag','hc_code']}
id_list=['ethnicity_id','location_id','race_id', 'offense_type_id']
lexicon_path = os.getcwd()+'\\Lexicon\\'
lexicon_list = []

for k in lexicon_save_columns.keys():
    index=lexicon_save_columns[k][0]#loaction_id
    save_df=load_csv(lexicon_path+k)
    save_df.drop(lexicon_drop_columns[k], axis = 1, inplace = True)
    save_df.set_index(index, inplace=True, drop=True)
    dict_save = save_df.to_dict()
    lexicon_list.append(dict_save)
print("set-up done")
```

This cell dealt with creating a lexicon from the lexicon part of the csvs. Meaning, there were about 30+ given csvs, along those 30 we had csvs that converted codes into words, for example:



location_id	location_code	location_name	location_region_type
1	1	Air/Bus/Train Terminal	Travel
2	2	Bank/Savings and Loan	Government
3	3	Bar/Nightclub	Entertainment
4	4	Church Synagogue/Temple	Religion
5	5	Commercial/Office Building	Other
6	6	Construction Site	Other
7	7	Convenience Store	Store
8	8	Department/Discount Store	Store

And so we loaded those CSVs here into a dictionary we will be using down the line to add to the big csvs.

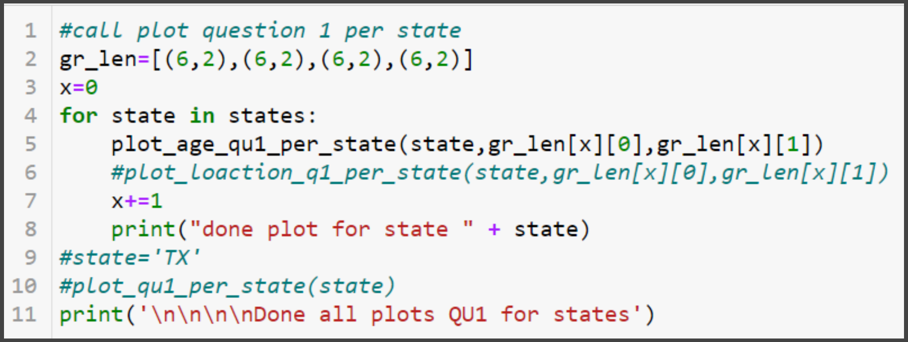


```python
# combinging the correct lexicons for later use
for state in states:
    state_all_path = os.getcwd()+'\\'+state+'\\all'
    dirs = os.listdir(state_all_path)
    print(state)
    for file in dirs:
        d = state_all_path+'\\'+file
        df = load_csv(d)
        if df is None: # debug handle
            continue
        i = 0
        for lex_dict in lexicon_list: #like on nibrs_location.csv dictionary
            col = df[id_list[i]] #id list holds the names like location_id or ethnicity_id
            for k,v in lex_dict.items():
                df[k] = col #dup cols X amount of times
                df[k].replace(lex_dict[k], inplace = True)
            i+=1
        save_to_csv(state_all_path, df, file)

print("done added columns.")
```

We take the lexicon dicti0onary from before and now we go through it, iterating, by making a copy column of the coded columns, and then we make a description column for each of them, creating a friendly to read csv.
We call the save_to_csv function that makes sure that the csv that is given exists, and then proceeds to adjust it.

save_to_csv(file_dir, dataframe, file_name)
Properties: - File_dir - the directory of the file we need to save to. - Dataframe - the dataframe we need to override over that csv. - File_name - the name of the file we need to find and override.
Method: - We check to see if all the information we are given are acceptable, if not we turn a error. - If the path exists we override the csv with the new dataframe information on top of that. - If the path does not exist we send a error.

Question 1.1 and 1.2 functions:



```python
#call plot question 1 per state
gr_len=[(6,2),(6,2),(6,2),(6,2)]
x=0
for state in states:
    plot_age_qu1_per_state(state,gr_len[x][0],gr_len[x][1])
    #plot_loaction_q1_per_state(state,gr_len[x][0],gr_len[x][1])
    x+=1
    print("done plot for state " + state)
#state='TX'
#plot_qu1_per_state(state)
print('\n\n\n\nDone all plots QU1 for states')
```

This cell handles the graph generation for each of the questions we needed. For the sake of need we just muted/commented-out each major function and called what we needed at that moment.
We do as we always did, we loop through the states in the global states array.
We had the idea of a dynamic size of the graphs, but for the sake of simplicity we kept them all at the same size of a grid 2x6, of course for larger year stems we suggest enlarging it.

Question 1.1

plot_age_qu1_per_state(state,x_len,y_len)
Properties: - State - the state abbreviation that we will use to access the other files. - X_len - the amount of columns we need for the subplots. - Y_len - the amount of rows we need for the subplots.
Method: - We first adjust the subplots to maintain the proper grid we will be using. We then make some counters to traverse this grid properly. - We then snatch a list of dirs within the “all” folder of the given state, so that we can iterate the following CSV files. - We loop through the different files that are there, we call load_csv and put it into a df

dataframe variable.

- We call a auxiliary function called age_bin that will make binary columns for the age itself, and will adjust the data when needed.
- We sum the data into a total_bahor variable that we will use for the final graph.
- Per each axis location we create a pie graph with all the needed things.
- For the last graph after the loop we genearet a total average graph, that is also a pie chart.

age_bin(df,take,c1,c2)

Properties:

- Df - the dataframe we use.
- Take - the column we need to duplicate for the project
- C1 - the column that we will use,.
- C2 - a column we will override if need be.

Method:

- We duplicate the column front he dataframe.
- We make conditions per the column and the needed data IE like under the age op18 we put binary 1 there, anything else would be 0 or 2 depending on the split.
- We then create a new column and retunrning it.

Question 1.2

plot_loaction_q1_per_state(state,x_len,y_len)

Properties:

- State - the state abbreviation that we will use to access the other files.
- X_len - the amount of columns we need for the subplots.
- Y_len - the amount of rows we need for the subplots.

Method:

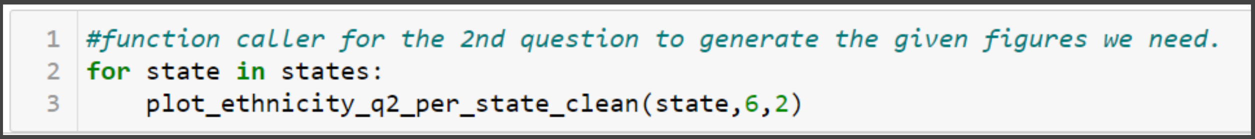
- We first adjust the subplots to maintain the proper grid we will be using. We then make some counters to traverse this grid properly.
- We then snatch a list of dirs within the “all” folder of the given state, so that we can iterate the following CSV files.
- We loop through the different files that are there, we call load_csv and put it into a df dataframe variable.
- We remove any data from the age_num column that is under the age of 18 due the fact that we wish only to sum up the amounts of criminal activity per region per underaged individuals.
- We adjusyted the data by dropping additional information.
- We then started calling a adjust_value function to the bahor, that we then added to a total_bahor to generate a total average pie chart.
- Afterwards we iterated through the axis and genearted the graphs per what we needed.

fix_value_count(bahor,labels)

Properties:
- Bahor - the column we need
- Labels - all the labels you need.

Method:
- We duplicate the column front he dataframe.
- We make conditions per the column and the needed data IE like under the age op18 we put binary 1 there, anything else would be 0 or 2 depending on the split.
- We then create a new column and returning it.

Question 2:



```
#function caller for the 2nd question to generate the given figures we need.
for state in states:
    plot_ethnicity_q2_per_state_clean(state,6,2)
```

Basically here like before we prompt the function to make our graphs for the second question (stacked bar graphs). As yo can see we are using the same logic of a state list.

plot_ethnicity_q2_per_state_clean(state,x_len,y_len)

Properties:
- State - the state abbreviation that we will use to access the other files.
- X_len - the amount of columns we need for the subplots.
- Y_len - the amount of rows we need for the subplots.

Method:
- We prepare the ground by making indexing arrays along with the subplot areas.
- We then iterate over all the files in the “all” folder via a osdirs, and create a graph per each year.
- We take a column “Location ID” and loop through the list of other value labels such as “Airport” and so on as we’d calculate the amounts alongside the list of races. This is all the while we call upon fix_value_count_clean
- Once that is fitted, we take the columns and we push the values into bahor, from it we make the graph itself, and continue to the next year.

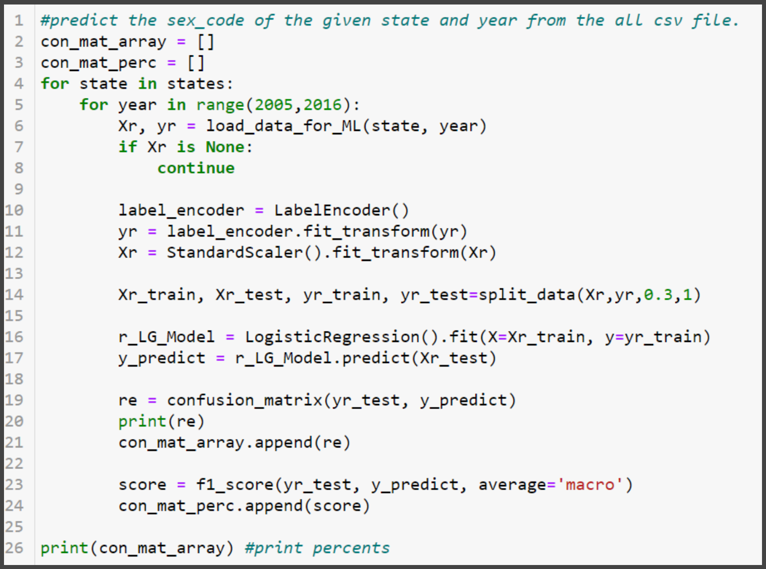
fix_value_count_clean(bahor,labels)

Properties:
- Bahor - the needed column from the datafram we need to adjust.
- Labels - the list of labels we need to loop through alongside the column itself.

Method:

- We copy all the data we have, along with the keys of the given dataframe.
- We then loop through them, to see which label is missing, if it is missing we remove it, to condense the graph itself.

Machine Learning Question:



```python
#predict the sex_code of the given state and year from the all csv file.
con_mat_array = []
con_mat_perc = []
for state in states:
    for year in range(2005,2016):
        Xr, yr = load_data_for_ML(state, year)
        if Xr is None:
            continue

        label_encoder = LabelEncoder()
        yr = label_encoder.fit_transform(yr)
        Xr = StandardScaler().fit_transform(Xr)

        Xr_train, Xr_test, yr_train, yr_test=split_data(Xr,yr,0.3,1)

        r_LG_Model = LogisticRegression().fit(X=Xr_train, y=yr_train)
        y_predict = r_LG_Model.predict(Xr_test)

        re = confusion_matrix(yr_test, y_predict)
        print(re)
        con_mat_array.append(re)

        score = f1_score(yr_test, y_predict, average='macro')
        con_mat_perc.append(score)

print(con_mat_array) #print percents
```

In this cell we activate the machine learning techniques we learned in class, mostly here we attempt to predict the crime per the gender of the involved in the incident.

Here we do LogisticRegression model, by adjusting the data properly, while looping per each state per each year.
We adjust the dataframe and the prediction column Xr Yr and if any are null we skip.
Once all is well we create a labelencoder to better the adjustments for the fit command for the LogisticRegression.

We call two function within this, we call load_data_for_ML and split_data which we will explain below.
Once we make the model and predict we then utilize confusion matrix to see the f1 score as well with it, see what went wrong and what did not. As per the powerpoint presentation, we concluded that our combination of data was insufficient and perhaps the question itself was faulty at its core.

load_data_for_ML(state,year)

Properties:

- State - the state index to read the file from
- Year - the year itself to build the file name from

Method:

- We call load_csv to read the data into a dataframe and then start processing from there.
- We adjust the sex_code column to be binary and not M or F. We have also in the past replacted any empty cells with U but we ultimately drop them for the sake of the fairity.
- We dropna anything that doesnt suit us (though in theory our csvs are full to the brim with data, with no NAs)
- We do a dynamic removal of all columns we do not need in case they pop up back again.
- We then do final adjustments to the data before returning the smaller dataframe with the column to its side.

split_data(X,y,split, ratio)

Properties:

- X - a dataframe that is adjusted.
- Y - a column from a dataframe that was adjusted.
- how split - the ratio to split (IE 0.3 is a 30% split)
- Ratio - the ratio for the random access split. Normally we kepot it 0 or 1, though any other adjustment gave no better result.

Method:

- We used the train_test_split method to split the data per what we were given and returned the splitted data back.

That is all.
Thanks for reading!

Asaf Dangoor
Yonatan Avizov