GIS Section

★ If you don’t have it, download QGIS and start a new project
★ Download redlining data using the attached link, and use whatever GIS compatible shapefile you would like for the map of Chicago. For this project, data was analyzed by census tract, but it could also be done by neighborhood, ZIP code, etc. The data at this link was downloaded as a shapefile
★ With the Open Data Source Manager, using the Browser option, the selected Chicago shapefile can be selected from the folder it was downloaded to on your computer. The redlining file can be connected under the ARCGis Rest Server section in the Open Data Source Manager.
★ Select ‘New’, paste in the link listed above for redlining data, and give the file a name. Click ‘Ok’, then ‘Connect’. Double click your connection then click add.
★ Right click on the Chicago map and click ‘Open Attributes Table’. Along the top bar, click ‘Field Calculator.’ Make sure ‘Create New Field’ is selected. Choose Decimal for ‘Output Field Type’. On the scrolling menu, click the dropdown menu for ‘Geometry’ and double click ‘$area’. Give the output field a useful name under ‘Output Field Name’ and click ‘Ok’.
★ Right click on ‘Urban Heat Islands’ and click ‘Open Attributes Table’. In the bottom left corner where it says ‘Show All Features’, click the ‘Field Filter’ option, select ‘CityState’, and type in Chicago. Click ‘Select All’ along the top bar.
★ After both datasets are loaded, go to the ‘Vector’ tab, navigate to ‘Geoprocessing Tools’, and click ‘Union’
★ Select either the redlining layer or the Chicago map as your Input layer and then select the unused one as the overlay layer. For the HOLC Map, click, ‘Selected Features Only’. You should also click ‘Advanced Settings’, denoted by the wrench, and change from ‘Use Default’ to ‘Skip (Ignore) Features With Invalid Geometries’. Change the temporary layer option to ‘Save to File…’ so the data isn’t lost. Click ‘Run’
★ Now open the Attribute Table for the Union layer, and repeat the area calculation process.
★ Left click the desired layers and select ‘Export’. Select ‘Save File As’, and choose the CSV option when the menu comes up. Then give the file an appropriate name in the ‘File Name’ area, and click on the three dots next to it to decide where you want your files to go.
★ The reference code for the following section is found in the Environmental Inequality Study in the Github repository. This section serves as a code walkthrough for the area calculation and dataframe creation
★ Import the CSV files into your code file as shown in the sample analysis
★ For the data frames containing total area and redlined area, which, for the sake of this explanation, will be called Census Tract Areas and Redlining Areas respectively, sort them by the column ‘name10’. This will allow for calculation of percentages of redlined, yellowlined, blueline, and greenlined areas

- ★ In our analysis, we created a smaller dataframe for Census Tract Areas and Redlining Areas that dropped extraneous columns, and renamed geoid as TractFIPS for ease of merging frames
- ★ Next, the percentages of redlining, yellowlining, etc. for each census tract will be calculated and dataframes will be created for each HOLC Grade percentage. See the sample code for method
- ★ Download environmental indicator data from EJScreen, or simply download all the files from the Github repository. Click the link to download the data and select the year you want to download data for. After this, download the first file titled EJScreen BG Columns and the fifth file, titled EJScreen_2022_StatePct_with_AS_CNMI_GU_VI
    - ○ The first file gives a brief summary of the column names, while the second contains the actual data
    - ○ The columns used in our analysis were OBJECTID, ID, ACSTOTPOP, P_PM25, P_PTRAF, P_DSLPM, P_PRMP, P_PTSDF, P_UST
    - ○ P_PWDIS was dropped because it was missing too much data
- ★ Import into your code file as shown in sample analysis
- ★ It is possible to select certain columns from the initial dataframe, so much as before, create a smaller dataframe with only the relevant columns. Do the same for the health data
- ★ From the Github Repository, download the Chicago Census Tract Health Data if you are doing a census tract analysis. Otherwise, data can be downloaded from the Chicago Health Atlas for relevant health indicators:
    - ○ Mental Health
    - ○ All cause mortality
    - ○ Childhood Cancer
    - ○ Asthma
    - ○ General Respiratory Illness
    - ○ Low Birth Weight
    - ○ Preterm Birth
    - ○ Birth Defects
    - ○ Cardiovascular Disease
    - ○ High Blood Pressure
    - ○ Kidney Disease
- ★ Merge Environmental Indicator, Health, and Redlining tables. Keep in mind that there must be a column that all the datasets can be merged along, so make sure the geographical categorizations match for each dataset (i.e. one data set isn't for neighborhood while the others are for census tracts).
- ★ In order to achieve the merging of data tables, take each HOLC Grade dataframe and add a column titled 'Name' or something similar with a range of data with 801 values.
    - ○ Make sure that if you want to display the dataframe to call it the new column is inserted, and to only run that terminal once, as a column can only be inserted once

★ Merge the HOLC Grade dataframes one at a time by merging left on the column ‘Name’ in order to ensure the correct order is maintained. The dataframes have already calculated their area percentages in the correct order for the dataframes to be merged.
★ It is also important to ensure that the columns are given comprehensive names, as when they are merged, they all end up with similar names. We gave the columns descriptive names each time a dataframe was merged
★ After all of the HOLC Grade dataframes are merged, use the same method to merge the health and environmental indicator dataframes, this time merging along the column ‘TractFIPS’, but still merging left. In the environmental indicator dataset, the column ‘ID’, which had identical values to the TractFIPS column, was renamed as TractFIPS to allow for merging
★ When those are merged, repeat the process of adding a Name column with values of 0 to 800, then merge left with the completed HOLC dataframe
★ Drop the extraneous columns that were used only for merging dataframes, such as Name, TractFIPS, and PlaceFIPS. At this point, we have 27 columns, though this number may vary based on data availability
★ Let it be noted that in this analysis, from the EJScreen data, unaltered percentiles for environmental indicators were used. The EJ Index percentiles of environmental indicators could also be used if one so chose, as those percentiles account for demographic.

VIF Analysis

Background:
VIF stands for Variance Inflation Factors, and is a tool used to explore multicollinearity. Multicollinearity occurs when independent variables are correlated. VIF measures the correlation between multiple independent variables in a least squared linear regression model. Each independent variable will be given a value by VIF to indicate how much multicollinearity exists between it and the other independent variables. While there is a wide range of values that can be considered indicators of significant multicollinearity, a good general rule is to consider VIF values of 5 and above as indicators of significant multicollinearity

★ First, it is important to note that VIF analysis cannot be done on non-numerical values, so any row with values of NaN must be dropped
★ We did this in our analysis using the function dataframe.dropna(), which drops any rows including NaN values. This removes only two rows from the dataframe, which should not significantly affect the analysis
★ For the VIF analysis, the redlining percentages and environmental indicators will be treated as the independent variables, while the health impacts are the dependent variables. Each of the health impacts will have its own VIF analysis
★ To begin the VIF analysis, from statsmodels.stats.outliers_influence, import variance_inflation_factor
★ From the merged data table, create an independent variable set. The independent variables will be columns:

    - Redlined_Area
    - Yellowlined_Area
    - Bluelined_Area
    - Greenlined_Area
    - Percentile for Particulate Matter 2.5 (P_PM25)
    - Percentile for Traffic Proximity (P_PTRAF)
    - Percentile for Diesel Particulate Matter (P_DSLPM)
    - Percentile for RMP Facility Proximity (P_PRMP)
    - Percentile for Hazardous Waste Proximity (P_PTSDF)
    - Percentile for Underground Storage Tanks (P_UST)
- Set an empty data frame called vif_data or something similar, and set the features of the independent variable set as the columns of that new data frame
- Take vif_data["VIF"] and set it equal to the function: variance_inflation_factor
    - The VIF function takes two parameters: variance_inflation_factor(exog, exog_idx)
        - exog sets the array containing the independent variables
        - exog_idx sets the index of the features whose effects on the other features are being measured
    - For the parameter exog, use the values of the independent variable set. For the parameter exog_idx use i and set a for loop to cycle through the lengths of the columns for the independent variable set.
- Print vif_data and analyze. Anything with a value greater than 5 has significant multicollinearity.

Ridge Regression

Background:
Ridge regression is a form of linear regression, which is modeled by:
$y = w[0] \times x[0] + w[1] \times x[1] + \ldots + w[n] \times x[n]$
This describes a model with *n* number of features, where w is the coefficient, or weight, assigned to each feature. This coefficient describes how heavily each feature will affect the model. Linear models use a cost function to calculate errors between predictions and actual values of data. The cost function is a single real valued number representing the average error across the dataset, and is represented by:
$\text{Loss} = (y - y_{\text{prediction}})^2$
Due to this, coefficients can become very large and cause overfitting of a model. When there are multiple predictors, some may have much larger coefficients, thus influencing the model more heavily (Datacamp).
Ridge regression is a method of tackling this problem, as it introduces a penalty to the loss function that will include an additional cost for functions with larger coefficients. The specific penalty used by Ridge Regression is called the L2 Penalty, which penalizes a model based on the sum of the squared coefficient values. The size of all of the coefficients is minimized, but none of them are allowed to reach zero. The weighting of the penalty is controlled by a hyperparameter lambda (sometimes defined as alpha in code) and is defined by:

Ridge Loss= loss + (lambda * L2_penalty)
(Machine Learning Mastery)

★ First, create a dataframe with only the most useful columns of data, in this case, the columns including health data and redlining percentages
★ Use the function pairplot() to show graphs of each of the columns plotted against both themselves and each of the other columns. Essentially, pairplot() takes the simplified dataframe as input and outputs every possible graph combination. This is why it's important to only include the most useful columns, as including every column takes too long to run
★ Next, for ridge regression, set features and targets from dataframes as X and y respectively. The features will be the HOLC Grade and Environmental Indicator columns, while the targets will be the health impacts, with a separate analysis being run for each of the health impact columns. We started with the 3rd column, indexed at 2 due to python's zero indexing
★ After X and y are defined, split the dataset into testing and training sections using train_test_split()
    ○ train_test_split() takes parameters (*arrays, test_size, train_size, random_state, shuffle, stratify)
    ○ Arrays is where you will input the arrays or datasets you want to use for testing, in this case X and y.
    ○ Test_size defaults to none and allows a decimal specification of the percent of the dataset that should be used for testing
    ○ Train_size is the same as test_size, but if not defined, the value is autoset to complement that of test_size
    ○ Random_state controls shuffling applied to data before split
    ○ Shuffle determines whether or not data should be shuffled before splitting. And takes a boolean value
    ○ Stratify determines whether data is split in a stratified fashion, and defaults to None
★ You can print out the shapes of the test and train dimensions to see how much of the dataset will be used for each
★ Scale features using StandardScalar(). Machine learning algorithms need a standardized dataset, and the function standardizes features by removing mean and scaling to unit variance. Use scaler.fit_transform() to standardize X_train and X_test.
★ Using the function LinearRegression(), create a non-penalized ridge regression model, fitting it with the training data as the x and y variables, in that order
★ Use the linear regression model.fit(X_train, y_train) with the training X and y data, and make a prediction with the test data using linear regression model.predict(X_test)
★ Show test scores and train scores for regular linear regression
    ○ In a good model, test scores and train scores will be very close together. Test and train scores that are far apart may indicate over or underfitting
★ Next will be the ridge regression section, for which the optimal value of alpha should be calculated

★ Calculate optimal coefficients for ridge regression. This section will start by defining cv using the function RepeatedKFold(). This function estimates the performance of the machine learning model, repeating a K-Fold Cross Validation procedure multiple times, as single runs can result in noisy estimates. The multiple runs give the truest estimate of performance, calculated using standard error. RepeatedKFold has parameters (n_splits, n_repeats, random_state)
  - n_splits determines the amount of folds, and a good default for this value is 10
  - n_repeats determines how many times the validation repeats, and 3, 5, or 10 is a good start, with more than 10 likely not being required
  - random_state can be set to 1
  - Source

★ Use the function RidgeCV() to define a variable ‘model’. The function is used to automatically find the optimal hyperparameters for Ridge Regression. It has parameters (alphas, fit_intercept, scoring, cv, gcv_mode, store_cv_values, alpha_per_target)
  - alphas provides an array of alpha values to try. Set the first value in arange() slightly above zero, as a value of zero can sometimes cause an error in the running of RidgeCV()
  - fit_intercept is a boolean value that defaults to true, defining whether or not the model’s intercept is calculated
  - Scoring has a default value of None. It is a string or a scorer callable object with a signature: scorer(estimator, X, y). When the value is set to its default, the negative mean squared error if cv is ‘auto’ or None (i.e. when using leave-one-out cross-validation), and r2 score otherwise.
  - cv is used to specify the cross validation fitting strategy, and can have values including None, an integer, CV Splitter, or an iterable yielding (train, test) splits as arrays of indices.
  - gcv_mode has a default value of ‘auto’ and indicates the strategy to use when performing Leave-One-Out Cross Validation
  - store_cv_values is a boolean value that defaults to false. It indicates if the cross-validation values corresponding to each alpha should be. It determines whether to optimize an alpha value for each target individually. When set to False, it uses one alpha value for all the targets.
  - Source

★ Fit the model to X and y

★ Print out the ideal model alpha using model.alpha

★ Next, the actual ridge regression will be run. Use the function Ridge(), which takes a parameter for alpha. Set alpha as the ideal value calculated previously.

★ Fit the ridge regression model to the X and y training data that was previously separated out using regression model.fit(X_train, y_train)

★ Find the training and testing scores using model.score(X_train, y_train) and model.score(X_test, y_test)
  - Print these scores and compare them to those achieved from the linear regression model. Ideally, the ridge regression training and testing scores should be closer together than those from the standard linear regression model

- ★ Plot the coefficients of each of the features gained from linear regression and ridge regression on the same graph and compare the results. This is done by optionally setting plot size, then using plt.plot(). This function takes quite a few parameters, many of which are optional.
  - ○ The first mandatory parameters are the x and y data, and in this case, the features and coefficients are those parameters. ‘features’ comes first and coefficients can be found using a regression model.coef_
  - ○ The transparency of the plot can be changed with a value alpha from 0 to 1, though we have neglected this as it makes the code less clear due to a more important value being defined as alpha
  - ○ The rest of the parameters in the plot() function in our code are purely aesthetic, and can be changed to the user’s liking
- ★ Repeat these steps to graph the linear regression coefficient. Show the plots and legend with plt.legend() and plt.show()
- ★ Plot the ridge regression model, using the same y variable as was defined as the target at the start of the analysis. For the x variable, write an equation using the ridge regression model: $y = w[0] \times x[0] + w[1] \times x[1] + \ldots + w[n] \times x[n]$
  - ○ To write the coefficients, use coef_[i] starting at zero and going to nine
  - ○ To write the x values, use X[:,i] starting at zero and going to nine
- ★ Plot using plt.scatter(x,y) and show your plot. The relationship should ideally be linearized
- ★ Repeat this value for any health factor you want to analyze. Just remember to change the index of the target to the index of the health column you want to run analysis on.