**Comparison between R and Python codes**

Contents

[Install library 4](#_Toc26981177)

[Install library 4](#_Toc26981178)

[Import library 4](#_Toc26981179)

[Use library 4](#_Toc26981180)

[Retrieval from web 4](#_Toc26981181)

[Web retrieval 5](#_Toc26981182)

[Web scrape 7](#_Toc26981183)

[Using API 8](#_Toc26981184)

[Twitter API 8](#_Toc26981185)

[Read file 12](#_Toc26981186)

[Check current directory 12](#_Toc26981187)

[Read file 13](#_Toc26981188)

[CSV file 13](#_Toc26981189)

[JSON file 13](#_Toc26981190)

[XLS file 13](#_Toc26981191)

[TXT file 14](#_Toc26981192)

[SQL 15](#_Toc26981193)

[General file 17](#_Toc26981194)

[Pickle file 17](#_Toc26981195)

[SAS file 17](#_Toc26981196)

[STATA file 18](#_Toc26981197)

[HDF5 file 18](#_Toc26981198)

[MATLAB file 18](#_Toc26981199)

[Read head 20](#_Toc26981200)

[Read tail 20](#_Toc26981201)

[Read and print image 20](#_Toc26981202)

[Summary 20](#_Toc26981203)

[Describe 20](#_Toc26981204)

[Shape of dataframe 20](#_Toc26981205)

[Length of list 20](#_Toc26981206)

[Count values 21](#_Toc26981207)

[Count NA 21](#_Toc26981208)

[Number of columns 21](#_Toc26981209)

[Rename column 21](#_Toc26981210)

[Insert new column 21](#_Toc26981211)

[Drop column 21](#_Toc26981212)

[Merge dataframes 21](#_Toc26981213)

[DataFrame to list 21](#_Toc26981214)

[Print 21](#_Toc26981215)

[Data Manipulation 21](#_Toc26981216)

[Filter 22](#_Toc26981217)

[Resample time series 22](#_Toc26981218)

[Locate value 22](#_Toc26981219)

[Locate largest and smallest values 23](#_Toc26981220)

[Group 23](#_Toc26981221)

[Aggregate 23](#_Toc26981222)

[Nest 24](#_Toc26981223)

[Pivot Table 24](#_Toc26981224)

[Sort 24](#_Toc26981225)

[Replace string 24](#_Toc26981226)

[Impute numeric value 24](#_Toc26981227)

[Impute categorical value 24](#_Toc26981228)

[Reshape 24](#_Toc26981229)

[Stack 24](#_Toc26981230)

[Mutate 25](#_Toc26981231)

[Dummy variables 25](#_Toc26981232)

[Convert to dataframe 25](#_Toc26981233)

[Date manipulation 25](#_Toc26981234)

[Date extraction (Datetime) 26](#_Toc26981235)

[Month-Year extraction 26](#_Toc26981236)

[Date extraction (Pandas) 26](#_Toc26981237)

[Set date 26](#_Toc26981238)

[Functions 26](#_Toc26981239)

[If Elseif Else 27](#_Toc26981240)

[Loop 28](#_Toc26981241)

[iteritems 28](#_Toc26981242)

[Encoding / Recoding 28](#_Toc26981243)

[Encoding (Convert non-numeric to numeric) 29](#_Toc26981244)

[Min-Max Scaling 29](#_Toc26981245)

[Standard Scaling 29](#_Toc26981246)

[Dimensionality Reduction 29](#_Toc26981247)

[PCA 30](#_Toc26981248)

[K-Means Clustering 33](#_Toc26981249)

[Correlation 34](#_Toc26981250)

[Data Balancing 34](#_Toc26981251)

[Time Series 34](#_Toc26981252)

[Trend visualization 35](#_Toc26981253)

[Statistical tests 35](#_Toc26981254)

[Statistical tests 35](#_Toc26981255)

[Linear Regression 35](#_Toc26981256)

[Linear Regression 36](#_Toc26981257)

[Confidence interval 37](#_Toc26981258)

[Logistic Regression 39](#_Toc26981259)

[Train Test Split 39](#_Toc26981260)

[Logistic Regression 40](#_Toc26981261)

[Evaluation for Logistic Regression 41](#_Toc26981262)

[GridSearch 42](#_Toc26981263)

[Confidence Interval 43](#_Toc26981264)

[Decision Tree 43](#_Toc26981265)

[Decision Tree 43](#_Toc26981266)

[Train-Test-Split 44](#_Toc26981267)

[Classification 44](#_Toc26981268)

[TPOT Classification 44](#_Toc26981269)

[TPOT Classification 45](#_Toc26981270)

[Log Normalization 46](#_Toc26981271)

[Logistic Regression 46](#_Toc26981272)

[Model Comparison 47](#_Toc26981273)

[Cross-validation 47](#_Toc26981274)

[Bootstrap analysis 47](#_Toc26981275)

[Bootstrap analysis 48](#_Toc26981276)

[Text Analytics 48](#_Toc26981277)

[Word Classification 49](#_Toc26981278)

[Visualization 53](#_Toc26981279)

[Boxplot 53](#_Toc26981280)

[Bar plot 53](#_Toc26981281)

[Dual axis 53](#_Toc26981282)

[Matplotlib Visualization 53](#_Toc26981283)

[Dual axis 54](#_Toc26981284)

[Bands 54](#_Toc26981285)

[Custom bins 55](#_Toc26981286)

[Side-by-side bar 55](#_Toc26981287)

[Seaborn Visualization 55](#_Toc26981288)

[Visualization 56](#_Toc26981289)

[Bokeh visualization 56](#_Toc26981290)

[Visualization 56](#_Toc26981291)

[Network Analysis 56](#_Toc26981292)

[Networkx 57](#_Toc26981293)

[Spatial Analysis 59](#_Toc26981294)

[Import data 59](#_Toc26981295)

[Plot map 59](#_Toc26981296)

[Projection 60](#_Toc26981297)

[Extract raster information 60](#_Toc26981298)

[Data balancing 61](#_Toc26981299)

[Logistic regression with elastic net regularization 62](#_Toc26981300)

[Prediction 62](#_Toc26981301)

[Plot 63](#_Toc26981302)

[Save file 63](#_Toc26981303)

[Save file 63](#_Toc26981304)

| **Function** | **R** | **Python** |
| --- | --- | --- |
| Install library | | |
| Install library | install.packages('#library') | pip install #library |
| Import library | library(#library) | import #library as #abbreviation |
| Use library | #library::#function | #abbreviation.#function |
| Retrieval from web | | |
| Web retrieval |  | #Method 1:  # Import package  from urllib.request import urlretrieve  # Import pandas  import pandas as pd  # Assign url of file: url  #url = '#file'  # Save file locally  urlretrieve(#url, '#file')  # Read file into a DataFrame and print its head  #df = pd.read\_csv('#file', sep=';')  print(#df.head())  #Method 2:  # Import packages  from urllib.request import urlopen, Request  # Specify the url  url = "#url"  request = Request(url)  # Sends the request and catches the response: response  #response = urlopen(#request)  # Print the datatype of response  print(type(#response))  # Extract the response: html  #html = #response.read()  # Print the html  print(#html)  # Be polite and close the response!  #response.close()  Method 3:  # Import package  import requests  # Specify the url: url  url = '#url'  # Packages the request, send the request and catch the response: r  #r = #requests.get(#url)  # Extract the response: text  #text = #r.text  # Print the html  print(#text) |
| Web scrape |  | # Import packages  import requests  from bs4 import BeautifulSoup  # Specify url: url  #url = '#url'  # Package the request, send the request and catch the response: r  #r = #requests.get(#url)  # Extracts the response as html: html\_doc  #html\_doc = #r.text  # Create a BeautifulSoup object from the HTML: soup  #soup = BeautifulSoup(#html\_doc)  # Prettify the BeautifulSoup object: pretty\_soup  #pretty\_soup = #soup.prettify()  # Print the response  print(#pretty\_soup)  # Get the title of webpage  #title = soup.title  # Get text in webpage  #text = soup.get\_text()  # Find all 'a' tags (which define hyperlinks): a\_tags  #a\_tags = soup.find\_all('a')  # Print the URLs to the shell  for #link in #a\_tags:  print(#link.get('href')) |
| Using API |  | # Import package  import requests  # Assign URL to variable: url  url = '#url/?apikey=#apikey&search=#search'  # Package the request, send the request and catch the response: r  #r = #requests.get(url)  # Decode the JSON data into a dictionary: json\_data  #data = #r.json()  # Print each key-value pair in #data  for #key in #data.keys():  print(#key + ': ', #data[#key]) Twitter API # Import package  import tweepy  # Store OAuth authentication credentials in relevant variables  #access\_token = ""  #access\_token\_secret = ""  #consumer\_key = ""  #consumer\_secret = ""  # Pass OAuth details to tweepy's OAuth handler  #auth = tweepy.OAuthHandler(#consumer\_key, #consumer\_secret)  #auth.set\_access\_token(#access\_token, #access\_token\_secret)  class MyStreamListener(tweepy.StreamListener):  def \_\_init\_\_(self, api=None):  super(MyStreamListener, self).\_\_init\_\_()  self.num\_tweets = 0  self.file = open("tweets.txt", "w")  def on\_status(self, status):  tweet = status.\_json  self.file.write(json.dumps(tweet) + '\n')  tweet\_list.append(status)  self.num\_tweets += 1  if self.num\_tweets < 100:  return True  else:  return False  self.file.close()  # Initialize Stream listener  #l = MyStreamListener()  # Create your Stream object with authentication  #stream = tweepy.Stream(#auth, #l for listener)  # Filter Twitter Streams to capture data by the keywords:  #stream.filter(['#search\_query'])  # Import package  import json  # String of path to file: tweets\_data\_path  #tweets\_data\_path = '#file.txt'  # Initialize empty list to store tweets: tweets\_data  #tweets\_data = []  # Open connection to file  #tweets\_file = open(#tweets\_data\_path, "r")  # Read in tweets and store in list: tweets\_data  for #line in #tweets\_file:  #tweet = json.loads(#line)  #tweets\_data.append(#tweet)  # Close connection to file  #tweets\_file.close()  # Print the keys of the first tweet dict  print(#tweets\_data[0].keys())  # Import package  import pandas as pd  # Build DataFrame of tweet texts and languages  #df = pd.DataFrame(#tweets\_data, columns=['#col1','#col2'])  # Initialize list to store tweet counts  [#search\_query1, #search\_query2] = [0, 0]  #Word count  import re  def word\_in\_text(word, text):  word = word.lower()  text = text.lower()  match = re.search(word, text)  if match:  return True  return False  # Iterate through df, counting the number of tweets in which  # each candidate is mentioned  for #index, #row in #df.iterrows():  #search\_query1 += word\_in\_text('#search\_query1', row['text'])  #search\_query2 += word\_in\_text('#search\_query2', row['text'])  # Import packages  import seaborn as sns  import matplotlib.pyplot as plt  # Set seaborn style  sns.set(color\_codes=True)  # Create a list of labels  #list = ['search\_query1', 'search\_query2']  # Plot the bar chart  #ax = sns.barplot(#list, [search\_query1, search\_query2])  #ax.set(ylabel="count")  plt.show() |
| Read file | | |
| Check current directory |  | #Method 1:  # Check the name of the current folder  current\_dir = !pwd  print(current\_dir)  # List all files in this folder  file\_list = !ls  print(file\_list)  # List all files in the datasets directory  dataset\_list = !ls #dir\  print(dataset\_list)  #Method 2:  import os  #wd = os.getcwd()  os.listdir(#wd) |
| Read file | read.csv('#file') | CSV file #Method 1:  import pandas as pd  #df = pd.read\_csv('#file',  sep = "#",  encoding = 'latin-1',  nrows = #num,  header = None,  names = ['#col1', '#col2'],  parse\_dates=['#col'],  comment = '#',  sep = '|',  na\_values='Nothing')  #Method 2:  #data = np.recfromcsv(#file, delimiter=',', names=True, dtype=None) JSON file #Method 1:  # Load JSON: json\_data  with open("#file.json") as #file:  #data = json.load(#file)  # Print each key-value pair in json\_data  for #key in #data.keys():  print(#key + ': ', #data[#key]) XLS file #Method 1:  # Import package  import pandas as pd  # Assign url of file: url  #url = '#file'  # Read in all sheets of Excel file: xls  #xls = pd.read\_excel(#url, sheet\_name=None)  # Print the sheetnames to the shell  print(#xls.keys())  # Print the head of the first sheet (using its name, NOT its index)  print(#xls['#sheet'].head())  #Method 2: Xls file  # Import pandas  import pandas as pd  # Assign spreadsheet filename: file  #file = '#file.xlsx'  # Load spreadsheet: xls  #xls = pd.ExcelFile(#file)  # Print sheet names  print(#xls.sheet\_names)  # Load a sheet into a DataFrame  #df = #xls.parse('#sheet\_name', skiprows=#1, usecols=#0, names=['#new\_col\_name'])  #df = #xls.parse(0) TXT file #Method 1:  np.loadtxt('#file', delimiter='\t', skiprows=1, usecols=[0,2] , dtype=float)  #Method 2:  #data = np.genfromtxt('#file', delimiter='\t', skiprows=1, names=True, dtype=None)  np.shape(#data) SQL #Method 1:  # Import packages  from sqlalchemy import create\_engine, Table, MetaData  import pandas as pd  # Create an engine to the database: engine  engine = create\_engine(''.join(['#dialect+#driver://',  '#username:#password',  '@#host',  ':#port/#database']))  # Create a metadata object: metadata  metadata = MetaData()  # Reflect census table from the engine: census  #data = Table('#table', #metadata, autoload=True, autoload\_with=#engine)  # Print census table metadata  print(repr(#data))  # Print table names  print(engine.table\_names())  # Print the column names  print(#data.columns.keys())  # Execute query and store records in DataFrame: df  #df = pd.read\_sql\_query('#SELECT \* FROM #table', #engine)  # Print head of DataFrame  print(#df.head())  #Method 2:  # Import necessary module  from sqlalchemy import create\_engine  import pandas as pd  # Create engine: engine  #engine = create\_engine('#sqlite:///#database.sqlite')  # Save the table names to a list: table\_names  #table\_names = #engine.table\_names()  # Print the table names to the shell  print(#table\_names)  # Open engine connection: con  #con = #engine.connect()  # Perform query: rs  #rs = #con.execute('#SELECT #col1, #col2 FROM #table1 INNER JOIN #table2 ON #table1.#FK = #table2.#PK')  # Save results of the query to DataFrame: df  #df = pd.DataFrame(#rs.fetchall())  # Close connection  #con.close()  # Print head of DataFrame df  print(#df.head())  #Method 3:  # Import necessary module  from sqlalchemy import create\_engine  import pandas as pd  # Create engine: engine  #engine = create\_engine('#sqlite:///#database.sqlite')  # Open engine in context manager  # Perform query and save results to DataFrame: df  with engine.connect() as con:  #rs = #con.execute("#SELECT #col FROM #table")  #df = pd.DataFrame(#rs.fetchmany(size=#3))  #df.columns = #rs.keys()  # Print the length of the DataFrame df  print(len(#df))  # Print the head of the DataFrame df  print(#df.head())  #Method 4:  from sqlalchemy import create\_engine  #engine = create\_engine('sqlite:///#database.sqlite')  # Create a connection on engine  #connection = #engine.connect()  # Build select statement for census table: stmt  #query = '#SELECT \* FROM #table'  # Execute the statement and fetch the results: results  #results = #connection.execute(#query).fetchall()  # Print results  print(#results)  #Method 5:  # Import select  from sqlalchemy import select  # Reflect census table via engine: census  #data = Table('#database', #metadata, autoload=True, autoload\_with=engine)  # Build select statement for census table: stmt  #query = select([#table])  # Print the emitted statement to see the SQL string  print(#query)  # Add a where clause to filter the results: stmt\_filtered  #query = #query.where(#table.columns.#col == '#condition')  # Execute the statement on connection and fetch 10 records: result  #results = #connection.execute(#query).fetchmany(size=#10)  # Execute the statement and print the results  print(#results)  # Append a where clause to match all the states in\_ the list states  #query = #query.where(#table.columns.#col.in\_(['']))  # Loop over the ResultProxy and print the result  for #result in #connection.execute(#query):  print(#result.col)  # Get the first row of the results by using an index: first\_row  #first\_row = #results[0]  # Print the first row of the results  print(#first\_row)  # Print the first column of the first row by accessing it by its index  print(#first\_row.keys())  # Print the column of the first row by using its name  print(#first\_row.#col)  # Loop over the results and print the age, sex, and pop2000  for result in results:  print(result.age, result.sex, result.pop2000)  #Method 6:  # Import and\_, desc  from sqlalchemy import and\_, desc  from sqlalchemy import desc  # Build a query for the census table:  #query = select([#table.columns.#col1, #table.columns.#col2]).order\_by(#table.columns.#col1, desc(#table.columns.#col2)).limit(#5)  # Append a where clause to select only specific records using and\_  #query = #query.where(  and\_(#table.columns. #col == '#condition',  #table.columns.#col != '#condition'  )  )  # Loop over the ResultProxy  for #result in connection.execute(#query):  print(#result.#col, #result.#col)  # Print the first 20 results  print(#results[:20])  # import pandas  import pandas as pd  # Create a DataFrame from the results: df  #df = pd.DataFrame(#results)  # Set column names  #df.columns = #results[0].keys()  #Method 7:  # Import func  from sqlalchemy import func  # Build a query to count the distinct values  #query = select([func.#sum(#table.columns.#col.label('#label'))])  #query = select([func.#count(#table.columns.#col.distinct())])  # Group query by state  #query = #query.group\_by(#table.columns.#col)  # Execute the query and store the scalar result  #count = connection.execute(#query).scalar()  # Print the distinct\_state\_count  print(#count)  Method 8:  # Import create\_engine function  from sqlalchemy import create\_engine  # Create an engine to the census database  #engine = create\_engine(''.join(['mysql+pymysql://',  '#username:#password',  '@#host:#port/',  '#table']))  # Print the table names  print(#engine.table\_names())  # Build query  #query = select([#table.columns.#col1, (#table.columns.#col2 - #table.columns.#col3).label('#label')])  # Append group by: stmt\_grouped  #query = #query.group\_by(#table.columns.#col)  # Append order by: stmt\_ordered  #query = #query.order\_by(desc('#label'))  # Return only 5 results  #query = #query.limit(#5)  # Use connection to execute and fetch all results  #results = connection.execute(#query).fetchall()  # Print each record  for #result in #results:  print('{}:{}'.format(#result.#col1, #result. #col2))  #Method 9:  # import case, cast and Float from sqlalchemy  from sqlalchemy import case, cast, Float  # Build an expression to calculate  #data = func.sum(  case([  (#table.columns.#col == '#condition', #table.columns.#col)  ], else\_=0))  # Cast an expression to Float  #data\_total = cast(func.sum(#table.columns.#col), Float)  # Build a query to calculate the percentage  #query = select([#data / #data\_total \* 100])  # Execute the query and store the scalar result  #results = connection.execute(#query).scalar()  # Print the percentage  print(#results)  #Method 10:  # Build a statement to select the tables  #query = select([#table1, #table2])  # Add a select\_from clause that wraps a join for the tables  #query\_join = #query.select\_from(  #table1.join(#table2, #table1.columns.#FK == # table2.columns.#PK))  # Execute the statement and get the first result: result  #result = #connection.execute(#query\_join).first()  # Loop over the keys in the result object and print the key and value  for #key in #result.keys():  print(#key, getattr(#result, #key))  #Method 11:  # Make an alias of the employees table: managers  #table2 = #table1.alias()  # Build a query to select  #query = select(  [#table1.columns.#col.label('#label1'),  #table2.columns.#col.label('#label2')]  )  # Match  #query\_matched = #query.where(#table1.columns.#FK == #table2.columns.#PK)  # Order the statement  #query \_ordered = #query \_matched.order\_by(#table1.columns.#col)  # Execute statement: results  #results = #connection.execute(#query\_ordered).fetchall()  # Print records  for #record in #results:  print(#record)  # Build a query  #query = select([#table1.columns.#col, func.count(#table2.columns.#col)])  # Append a where clause  #query\_matched = #query.where(#table1.columns.#col == #table2.columns.#col)  # Group by  #query\_grouped = #query \_matched.group\_by((#table1.columns.#col)  # Execute statement: results  results = connection.execute(#query \_grouped).fetchall()  # Print records  for #record in #results:  print(#record)  #Method 12:  # Start a while loop checking for more results  while more\_results:  # Fetch the first 50 results from the ResultProxy: partial\_results  #partial\_results = #results\_proxy.fetchmany(#50)  # if empty list, set more\_results to False  if #partial\_results == []:  #more\_results = False  # Loop over the fetched records and increment the count  for #row in #partial\_results:  if #row.#col in #count:  #count[row.#col] += 1  else:  #count[row.#col] = 1  # Close the ResultProxy, and thus the connection  #results\_proxy.close()  # Print the count by state  print(#count) General file #Method 1:  with open('#file') as f:  print(f.read())  print(f.readline())  #Method 2:  #file = open('#file', mode = 'r / w')  #data = file.read()  #file.close()  print(#file)  print(#file.closed) Pickle file Method 1: Pickle file  # Import pickle package  import pickle  # Open pickle file and load data: d  with open('#data.pkl', mode='#rb #b for binary') as file:  #d = pickle.load(#file) SAS file Method 1:  # Import sas7bdat package  from sas7bdat import SAS7BDAT  # Save file to a DataFrame: df\_sas  with SAS7BDAT('#filename.sas7bdat') as #file:  #df = #file.to\_data\_frame() STATA file Method 1:  #df = pd.read\_stata('#file.dta') HDF5 file Method 1: HDF5 file  # Import packages  import numpy as np  import h5py  # Assign filename: file  #file = '#filename.hdf5'  # Load file: data  #data = h5py.File(#file, 'r')  # Print the datatype of the loaded file  print(type(#data))  # Print the keys of the file  for #key in #data.keys():  print(#key)  data['#col']['#col'].value MATLAB file Method 1: MATLAB file  # Import package  import scipy.io  # Load MATLAB file: mat  #mat = scipy.io.loadmat('#file.mat')  # Print the keys of the MATLAB dictionary  print(#mat.keys())  # Print the type of the value corresponding to the key  print(type(#mat['#col']))  # Print the shape of the value corresponding to the key  print(np.shape(#mat['#col']))  scipy.io.savemat('#file.mat') |
| Read head | head(#df, #num) | #Method 1:  #df.head(#num)  #Method 2:  #df = !head -n 20 #dir |
| Read tail | tail(#df, #num) | #df.tail(#num) |
| Read and print image |  | # Import package  import numpy as np  # Assign filename to variable: file  #file = '#digits.csv'  # Load file as array: digits  #digits = np.loadtxt(#file, delimiter=',')  # Print datatype of digits  print(type(#digits))  # Select and reshape a row  #im = #digits[21, 1:]  #im\_sq = #np.reshape(#im, (28, 28))  # Plot reshaped data (matplotlib.pyplot already loaded as plt)  plt.imshow(#im\_sq, cmap='Greys', interpolation='nearest')  plt.show() |
| Summary | | |
| Describe | summary(#df)  glimpse(#df)  str(#df) | #df.describe()  #df.info() |
| Shape of dataframe |  | #df.shape()  #df.shape[0] #rows |
| Length of list |  | len(#list) |
| Count values |  | #Method 1:  #Show proportion  #df.#col.value\_counts(normalize = True)  #Method 2:  #df.#col.unique().dropna() |
| Count NA |  | #Method 1:  #df.isnull().sum()  #Method 2:  # Assert that there are no missing values  assert #df.notnull().all().all()  # Assert that all values are >= 0  assert (#df >= 0).all().all() |
| Number of columns |  | len(#df.columns) |
| Rename column | #df %>%  rename(#new = #old) | #Method 1:  #df.rename(  columns = {'#old': '#new'},  inplace = True  )  #Method 2:  $df.columns = ['#col1', '#col2'] |
| Insert new column |  | #df.insert(#n\_cols, '#col', #col\_value)  #df\_new = #df.assign(#col=#condition) |
| Drop column |  | #df = #df.drop(columns = ['#col'], axis = 1) |
| Merge dataframes | Method 1:  #df\_new = cbind(#df1, #df2)  Method 2:  #df\_new = full\_join(#df1, #df2, by = '#col') | #Method 1:  #df\_new = #df1.append(#df2).append(#df3).sort\_index(ascending=False)  #Method 2:  pd.concat([#df1, #df2], axis=#1 for col)  #Method 3: Mass concatenation  # Import necessary modules  import glob  import pandas as pd  # Write the pattern: pattern  #pattern = '#\* / ?.csv'  # Save all file matches  #files = glob.glob(#pattern)  # Create an empty list: frames  #frames = []  # Iterate over files  for #file in #files:  # Read into a DataFrame: df  #df = pd.read\_csv(#file)    # Append df to frames  #frames.append(#df)  #Method 3:  #df = pd.merge(left=#df1, right=#df2, left\_on='#col', right\_on='#col') |
| DataFrame to list |  | #df.values.tolist() |
| Print |  | print("%s " % (#var))  print("{}".format(#df[0])) |
| Data Manipulation | | |
| Filter | #Method 1:  #df %>%  dplyr::filter(str\_detect(#col, "#search\_str"))  Method 2:  #df %>%  dplyr::filter(#col %in% c(#value1, #value2))  Method 3:  #df %>%  dplyr::filter(#col %in% (#value1:#value2))  #filter(  # dplyr::between(decade, 1970, 2010))  Method 4:  #df %>%  transmute(#col = #col) | #col\_list = ['#col1', '#col2']  # Filter for rows containing these metrics  #df\_f = #df[#df.#colname.isin(#col\_list)] |
| Remove duplicates |  | # Drop the duplicates  #df\_new = #df.drop\_duplicates()  # Print info  print(#df\_new.info()) |
| Drop NA |  | #df.dropna(axis=0, how='any') |
| Fill NA |  | # Calculate the mean  #col\_mean = #df.#col.mean()  # Replace all the missing values in the column with the mean  #df['#col'] = #df.#col.fillna(#col\_mean)  # Print the info of dataframe  print(#df.info()) |
| Resample time series |  | # Prepare data for last x years  #df\_new = #df['#year':'#year']  # Calculate annual statistics  display(#df\_new.resample('A').mean()) #Annual  # Calculate weekly statistics  display(#df\_new.resample('W').mean().mean()) #Weekly  # Mean weekly counts  display(#df\_new['#col'].resample('W').count().mean()) |
| Type casting |  | #Method 1:  #df.#col = #df.#col.astype('category')  #Method 2:  #df.#col = pd.to\_numeric(#df.#col, errors='coerce') |
| Locate value |  | #df['#col'].iloc[#row\_num] |
| Locate largest and smallest values |  | #df.nlargest(#num, '#col')  #df.nsmallest(#num, '#col') |
| Split string |  | #Method 1:  #df.#col.str[#num:]  #Method 2:  #df.#col.str.split('#|').str.get(#0) |
| Concatenate |  |  |
| Group |  | #Method 1:  #df\_crosstab = #df.groupby(['#col1', '#col2']).mean()  #df\_crosstab = #df.groupby(['#col1', '#col2']).sum()  #df\_crosstab = #df.groupby(['#col1', '#col2']).describe()  # Convert the GroupBy object to a DataFrame  #new\_df = #df\_crosstab.reset\_index()  Method 2:  #df.groupby(pd.Grouper(key = '#col', freq = 'AS #for year')).size() |
| Aggregate |  | #Method 1:  #df\_agg = #df.groupby('#col') ['#col\_with\_value'].agg(['count', 'mean', 'sum', 'nunique'])  #Method 2:  #df.agg({'#col\_with\_value': 'count'})  #Method 3: (Numerical variables)  #df.groupby(['#col']).size() |
| Nest | # "Nest" the data  #df %>%  group\_by(#col) %>%  nest(.key = '#new\_col') -> #df\_nested  # Calculate the total number of records per row  #df\_nested %>%  mutate(n = map\_dbl(#new\_col, function (x) nrow(x))) |  |
| Sort | #df %>%  arrange(#col) | #df. sort\_values(by = '#col', ascending = False) |
| Replace string |  | #Method 1:  #df.replace('#string', np.NaN)  #Method 2:  #df['#col'] = #df['#col'].str.replace('#Old', '#New') |
| Regular expression |  | # Import the regular expression module  import re  # Compile the pattern  #pattern = re.compile('#\d{3}-\d{3}-\d{4}')  # See if the pattern matches  print(bool(#pattern.match('#str')))  # Find the numeric values: matches  re.findall('#\d+', '#str')  # Write the first pattern  pattern1 = bool(re.match(pattern='#\d{3}-\d{3}-\d{4}', string='#123-456-7890'))  print(pattern1)  # Write the second pattern  pattern2 = bool(re.match(pattern='#\$\d\*\.\d\*', string='#$123.45'))  print(pattern2)  # Write the third pattern  pattern3 = bool(re.match(pattern='#[A-Z]\w\*', string='#Australia'))  print(pattern3)  # Write the regular expression: pattern  #pattern = '^[A-Za-z\.\s]\*$'  # Create the Boolean vector: mask  #mask = #df.#col.str.contains(#pattern)  # Invert the mask: mask\_inverse  #mask\_inverse = ~#mask  # Subset using mask\_inverse: invalid\_df  #invalid\_df = #df.#col.loc[#mask\_inverse]  # Print  print(#invalid\_df) |
| Impute numeric value |  | #df.fillna(#df.mean(), inplace=True) |
| Impute categorical value |  | for col in #df.columns:  # Check if the column is of object type  if #df [col].dtypes == 'object':  # Impute with the most frequent value  #print(#df[col].value\_counts().index[0])  #df = #df.replace(np.NaN, #df [col].value\_counts().index[0])  # Count the number of NaNs in the dataset and print the counts to verify  #df.isnull().sum() |
| Reshape / Melt |  | # Melt  #df\_new = pd.melt(frame=#df, id\_vars=['#col1, #col2\_name\_to\_keep'], var\_name='#agg\_col\_name\_type\_to\_rows', value\_name='#agg\_col\_name\_value to\_rows', value\_vars= ['# agg\_col\_1, # agg\_col\_2']) |
| Pivot Table |  | #Method 1: #Use for duplicate entries  #df.pivot\_table(index=['#col1, #col2\_for\_pivot\_row'], columns='#col', values='#col\_for\_values', fill\_value=0, aggfunc=np.mean).reset\_index()  #Method 2: #Cannot use for duplicate entries  #df.pivot(index=['#col1, #col2\_for\_pivot\_row'], columns='#col', values='#col\_for\_values', fill\_value=0).reset\_index() |
| Stack |  | #df.stack() |
| Mutate | #df %>%  mutate(#col = ifelse(#condition, #new\_value\_for\_true, #new\_value\_for\_false)) -> #df  # Recode  #df %>%  mutate(#col = factor(#col, levels = 0:1, labels = c('#value1','#value2'))) -> #df |  |
| Dummy variables |  | #df\_dummy = pd.get\_dummies(data=#df['#col'], drop\_first=True) |
| Convert to dataframe |  | pd.DataFrame(#df) |
| Date manipulation | | |
| Date extraction (Datetime) | #Extract decade  library(lubridate)  ymd\_hms(#date) %>% round\_date("10y") %>% year() | from datetime import datetime  datetime.strptime('#date', '#format')  print(dt\_object.month)  %d day (e.g. 01)  %b abbreviated month  %B fully spelt month  %m month (e.g. 01)  %y year (e.g. 00)  %Y full year (e.g. 2000)  # Set the variable for the datetime to convert  dt = '14/02/2018'  # Create the dictionary for the month values  mm = {'01': 'January', '02': 'February', '03': 'March'}  # Split the dt string into the different parts  day, month, year = dt.split('/')  # Print the concatenated date string  print(day + ' ' + mm[month] + ' ' + year) |
| Month-Year extraction |  | #df['#month\_year'] = #df.apply(lambda x: str(x['date'].year) + '-' + str(x['date'].month), axis = 1) |
| Date extraction (Pandas) |  | #df['#col'] = pd.DatetimeIndex(#df['#col']).year  #df.#col.dt.weekday\_name |
| Set date |  | #date = pd.to\_datetime('#date', unit = 's') |
| Functions | | |
| If Elseif Else |  | # Create the computevariance function  def computevariance(amount, sentiment):  if (sentiment < 0.6):  res = amount + (amount \* 0.02)  elif (sentiment > 0.8):  res = amount + (amount \* 0.07)  else:  res = amount + (amount \* 0.05)  return res |
| Loopiteritems |  | # Set the index to start at 0  index = 0  # Create the dictionary for the months  tt = {'Jan': 0, 'Feb': 0, 'Mar': 0}  # Create a for loop that will iterate the date and amount values in the dataset  for date, amount in df.iteritems():  # Create the if statement to split the day and month, then add it to the new tt variable  if index > 0:  day, month = date.split('-')  tt[month] +=float(amount[0])  index += 1  print(tt)  totals = {'Jan': 0, 'Feb': 0, 'Mar': 0}  calendar = {'01': 'Jan', '02': 'Feb', '03': 'Mar'}  for date, amount in df1.iteritems():  day, month, year = date.split('-')  totals[month] +=float(amount[0])  for date, amount in df2.iteritems():  day, month, year = date.split('/')  totals[calendar[month]] += float(amount[0])  print(totals) |
| Function |  | # Define function  def function(#arg):  # Return 0 if arg is false  if #arg == '#condition1':  return 0    # Return 1 if arg is true  elif #arg == '#condition2':  return 1    # Return np.nan  else:  return np.nan  # Apply the function to the column  #df['#col'] = #df.#col.apply(#function)  # Apply lambda function  #df['#col'] = #df.#col.apply(#lambda x: x.function('#arg')) |
| Encoding / Recoding | | |
| Encoding (Convert non-numeric to numeric) |  | # Import LabelEncoder  import sklearn  from sklearn.preprocessing import LabelEncoder  # Instantiate LabelEncoder  le = LabelEncoder()  # Iterate over all the values of each column and extract their dtypes  for col in #df.columns:  # Compare if the dtype is object  if #df[col].dtypes == 'object':  # Use LabelEncoder to do the numeric transformation  le.fit(#df[col])  #df[col]=le.transform(#df[col]) |
| Min-Max Scaling |  | # Import MinMaxScaler  from sklearn.preprocessing import MinMaxScaler  # Instantiate MinMaxScaler and use it to rescale X\_train and X\_test  scaler = MinMaxScaler(feature\_range = (0, 1))  rescaledX\_train = scaler.fit\_transform(X\_train)  rescaledX\_test = scaler. fit\_transform(X\_test) |
| Standard Scaling |  | from sklearn.preprocessing import StandardScaler  scaler = StandardScaler()  features\_scaled = scaler.fit\_transform(features) |
| Dimensionality Reduction | | |
| PCA |  | # Standardize and center the feature columns  from sklearn.preprocessing import StandardScaler  scaler = StandardScaler()  features\_scaled = scaler.fit\_transform(features)  # Import the PCA class function from sklearn  from sklearn.decomposition import PCA  pca = PCA()  # Fit the standardized data to the pca  pca.fit(features\_scaled)  #Method 1 of plotting:  # Get our explained variance ratios from PCA using all features  pca = PCA()  pca.fit(scaled\_train\_features)  exp\_variance = pca.explained\_variance\_ratio\_  # plot the explained variance using a barplot  fig, ax = plt.subplots()  ax.bar(range(1, pca.n\_components\_ + 1), pca.explained\_variance\_ratio\_)  ax.set\_xlabel('Principal Component #')  # Import numpy  import numpy as np  # Calculate the cumulative explained variance  cum\_exp\_variance = np.cumsum(pca.explained\_variance\_ratio\_)  # Plot the cumulative explained variance and draw a dashed line at 0.90.  fig, ax = plt.subplots()  plt.plot(range(1, pca.n\_components\_ + 1), cum\_exp\_variance, color = 'grey')  ax.set(xlabel='#label', ylabel='label')  ax.axhline(y=0.9, linewidth = 1, linestyle='--')  n\_components = 6  # Perform PCA with the chosen number of components and project data onto components  pca = PCA(n\_components, random\_state=10)  pca.fit(scaled\_train\_features)  pca\_projection = pca.transform(scaled\_train\_features)  #Method 2 of plotting:  # Plot the proportion of variance explained on the y-axis of the bar plot  import matplotlib.pyplot as plt  plt.bar(range(1, pca.n\_components\_ + 1), pca.explained\_variance\_ratio\_)  plt.xlabel('Principal component #')  plt.ylabel('Proportion of variance explained')  plt.xticks([1, 2, 3])  # Compute the cumulative proportion of variance explained by the first two principal components  two\_first\_comp\_var\_exp = pca.explained\_variance\_ratio\_[0] + pca.explained\_variance\_ratio\_[1]  print("The cumulative variance of the first two principal components is {}".format(  round(two\_first\_comp\_var\_exp, 5)))  # Transform the scaled features using two principal components  pca = PCA(n\_components=2)  p\_comps = pca.fit\_transform(features\_scaled)  # Extract the first and second component to use for the scatter plot  p\_comp1 = p\_comps[:, 0]  p\_comp2 = p\_comps[:, 1]  # Plot the first two principal components in a scatter plot  plt.scatter(p\_comp1, p\_comp2) |
| K-Means Clustering |  | # Import KMeans from sklearn  from sklearn.cluster import KMeans  # A loop will be used to plot the explanatory power for up to 10 KMeans clusters  ks = range(1, 10)  inertias = []  for k in ks:  # Initialize the KMeans object using the current number of clusters (k)  km = KMeans(n\_clusters=k, random\_state=8)  # Fit the scaled features to the KMeans object  km.fit(features\_scaled)  # Append the inertia for `km` to the list of inertias  inertias.append(km.inertia\_)  # Plot the results in a line plot  plt.plot(ks, inertias, marker='o')  # Create a KMeans object with 3 clusters, use random\_state=8  km = KMeans(n\_clusters=3, random\_state=8)  # Fit the data to the `km` object  km.fit(features\_scaled)  # Create a scatter plot of the first two principal components  # and color it according to the KMeans cluster assignment  plt.scatter(p\_comps[:, 0], p\_comps[:, 1], c=km.labels\_) |
| Correlation |  | # Compute the correlation coefficent for all column pairs  #df\_corr = #df.corr()  #df\_corr.style.background\_gradient()  # import seaborn and make plots appear inline  import seaborn as sns  %matplotlib inline  # Create a pairwise scatter plot to explore the data  sns.pairplot(#df) |
| Data Balancing |  | # Subset only the hip-hop tracks, and then only the rock tracks  #df1\_new = #df.loc[#df['#col'] == '#condition']  #df2\_new = #df.loc[#df['#col'] == '#condition']  # sample the df1\_new to be the same number as there are df2\_new  df2\_new = df2\_new.sample(len(df1\_new), random\_state=10)  # concatenate the dataframes  df12\_new = pd.concat([#df1\_new, #df2\_new])  # The features, labels, and pca projection are created for the balanced dataframe  features = #df12\_new.drop(['#col'], axis=1)  labels = #df12\_new['#col']  pca\_projection = pca.fit\_transform(scaler.fit\_transform(features))  # Redefine the train and test set with the pca\_projection from the balanced data  train\_features, test\_features, train\_labels, test\_labels = train\_test\_split(pca\_projection, labels, random\_state=10) |
| Time Series | | |
| Trend visualization |  | # Import required library  import statsmodels.api as sm  # Prepare data  #df\_new = #df['#year':'#year']['#col'].resample('W').bfill()  decomposed = sm.tsa.seasonal\_decompose(#df\_new, extrapolate\_trend=1, freq=52)  # Create plot  fig = plt.figure(figsize=(12, 5))  # Plot and customize  ax = decomposed.trend.plot(label='Trend', linewidth=2)  ax = decomposed.observed.plot(label='Observed', linewidth=0.5)  ax.legend()  ax.set\_title('#Title')  # Show plot  plt.show() |
| Statistical tests | | |
| Statistical tests | chisq.test(#df$X, #df$Y)  t.test(#df$Y ~ #df$X) |  |
| Linear Regression | | |
| Linear Regression |  | # Import LinearRegression  from sklearn.linear\_model import LinearRegression  # Build a linear regression model  regr = LinearRegression()  target = #df['#col']  features = #df.drop('#col', axis=1)  features = #df [#df.columns.difference([target.name])]  # Fit regr to the dataset  regr.fit(X = #df\_dummy, y = #df['#col'])  regr.fit(X = features, y = target)  # Get estimated intercept and coefficient values  #col1 = regr.intercept\_  #col2 = regr.coef\_[[0]][0]  # Inspect the estimated intercept and coefficient values  print((#col1, #col2)) |
| Confidence interval |  | # Import a module  import numpy as np  # Create an array of indices to sample from  #inds = np.arange(len(#df['#col']))  # Initialize 500 replicate arrays  size = 500  #sample1 = np.empty(size)  #sample2 = np.empty(size)  # Generate replicates  for i in range(size):  # Resample the indices  #bs\_inds = np.random.choice(#inds, len(#inds))    # Get the sampled genre and sampled adjusted gross  #bs\_X = #df['#col'][bs\_inds]  #bs\_Y = #df['#col'][bs\_inds]    # Convert sampled genre to dummy variables  #bs\_dummies = pd.get\_dummies(data=#df['#col'], drop\_first=True)    # Build and fit a regression model  regr = LinearRegression().fit(#bs\_dummies, #bs\_Y)    # Compute replicates of estimated intercept and coefficient  #X0[i] = regr.intercept\_  #X1[i] = regr.coef\_[[0]][0]  # Compute 95% confidence intervals for intercept and coefficient values  #confidence\_interval\_1 = np.percentile(#X0, [2.5, 97.5])  #confidence\_interval\_2 = np.percentile(#X1, [2.5, 97.5])    # Inspect the confidence intervals  print(#confidence\_interval\_1)  print(#confidence\_interval\_2) |
| Logistic Regression | | |
| Train Test Split |  | # Import train\_test\_split  from sklearn.model\_selection import train\_test\_split  # Drop the features 11 and 13 and convert the DataFrame to a NumPy array  #df = #df.drop([11, 13], axis=1).values  # Segregate features and labels into separate variables  X,y = #df[:,0:12] , #df[:,13]  # Split into train and test sets  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.33, random\_state=42) |
| Logistic Regression | # use glm function from base R and specify the family argument as binomial  #model <- glm(data = #df, #Y ~ #X1 + #X2, family='binomial')  # extract the model summary  summary(#model)  # get the predicted probability in our dataset using the predict() function  #pred\_prob <- predict(#model, #df, type = "response")  # create a decision rule using probability 0.5 as cutoff and save the predicted decision into the main data frame  #df$#pred\_col <- ifelse(#pred\_prob >= 0.5, 1, 0)  # create a newdata data frame to save a new case information  #df\_new <- data.frame(#col = #value)  # predict probability for this new case and print out the predicted value  #pred <- predict(#model, #df\_new, type = "response")  #pred | # Import LogisticRegression  from sklearn.linear\_model import LogisticRegression  # Instantiate a LogisticRegression classifier with default parameter values  logreg = LogisticRegression(random\_state = 10)  # Fit logreg to the train set  logreg.fit(rescaledX\_train, y\_train) |
| Evaluation for Logistic Regression | # load Metrics package  library(Metrics)  # calculate auc, accuracy, clasification error  #True as 1st argument, Predicted as 2nd argument  auc <- auc(#df$Y, #df$pred\_Y)  accuracy <- accuracy(#df$Y, #df$pred\_Y)  classification\_error <- ce(#df$Y, #df$pred\_Y)  # print out the metrics on to screen  print(paste("AUC=", auc))  print(paste("Accuracy=", accuracy))  print(paste("Classification Error=", classification\_error))  # confusion matrix  table(#df$Y, #df$pred\_Y, dnn=c('True Status', 'Predicted Status')) # confusion matrix | #Method 1: # Import confusion\_matrix  from sklearn.metrics import confusion\_matrix  # Use logreg to predict instances from the test set and store it  y\_pred = logreg.predict(rescaledX\_test)  # Get the accuracy score of logreg model and print it  print("Accuracy of logistic regression classifier: ", ...)  # Print the confusion matrix of the logreg model  print(logreg.score(rescaledX\_test, y\_test))  confusion\_matrix(y\_test, y\_pred)  #Method 2:  # Create the classification report for both models  from sklearn.metrics import classification\_report  class\_rep\_tree = classification\_report(test\_labels, logreg.predict(test\_features)) |
| GridSearch |  | # Import GridSearchCV  from sklearn.model\_selection import GridSearchCV  # Define the grid of values for tol and max\_iter  tol = [0.01, 0.001, 0.0001]  max\_iter = [100, 150, 200]  # Create a dictionary where tol and max\_iter are keys and the lists of their values are corresponding values  param\_grid = dict(tol = tol, max\_iter = max\_iter)  # Instantiate GridSearchCV with the required parameters  grid\_model = GridSearchCV(estimator = logreg, param\_grid = param\_grid, cv = 5)  # Use scaler to rescale X and assign it to rescaledX  rescaledX = scaler. fit\_transform (X)  # Fit data to grid\_model  grid\_model\_result = grid\_model. fit(rescaledX, y)  # Summarize results  best\_score, best\_params = grid\_model\_result.best\_score\_, grid\_model\_result.best\_params\_  print("Best: %f using %s" % (best\_score, best\_params)) |
| Confidence Interval | # load the broom package  library(broom)  # tidy up the coefficient table  #df\_tidy <- tidy(#model)  #df\_tidy  # calculate OR  #df\_tidy$#OR <- exp(#df$#col)  # calculate 95% CI and save as lower CI and upper CI  #df\_tidy$lower\_CI <- exp(#df\_tidy$estimate - 1.96 \* #df\_tidy$std.error)  #df\_tidy$upper\_CI <- exp(#df\_tidy$estimate + 1.96 \* #df\_tidy$std.error)  # display the updated coefficient table  #df\_tidy |  |
| Decision Tree | | |
| Decision Tree |  | # Import train\_test\_split function and Decision tree classifier  from sklearn.model\_selection import train\_test\_split  from sklearn.tree import DecisionTreeClassifier  # Split our data  train\_features, test\_features, train\_labels, test\_labels = train\_test\_split(pca\_projection, labels, random\_state = 10)  # Train our decision tree  tree = DecisionTreeClassifier()  tree.fit(train\_features, train\_labels)  # Predict the labels for the test data  pred\_labels\_tree = tree.predict(test\_features) |
| Train-Test-Split | | |
| Classification |  | # Import train\_test\_split method  from sklearn.model\_selection import train\_test\_split  # Split DataFrame into  # X\_train, X\_test, y\_train and y\_test datasets,  # stratifying on the `target` column  X\_train, X\_test, y\_train, y\_test = train\_test\_split(  #df.drop(columns='#colname'),  #df.target,  test\_size=0.25,  random\_state=42,  stratify=#df.#colname  )  # Print out the first 2 rows of X\_train  X\_train.head() |
| TPOT Classification | | |
| TPOT Classification |  | # Import TPOTClassifier and roc\_auc\_score  from tpot import TPOTClassifier  from sklearn.metrics import roc\_auc\_score  # Instantiate TPOTClassifier  tpot = TPOTClassifier(  generations=5,  population\_size=20,  verbosity=2,  scoring='roc\_auc',  random\_state=42,  disable\_update\_check=True,  config\_dict='TPOT light'  )  tpot.fit(X\_train, y\_train)  # AUC score for tpot model  tpot\_auc\_score = roc\_auc\_score(y\_test, tpot.predict\_proba(X\_test)[:, 1])  print(f'\nAUC score: {tpot\_auc\_score:.4f}')  # Print best pipeline steps  print('\nBest pipeline steps:', end='\n')  for idx, (name, transform) in enumerate(tpot.fitted\_pipeline\_.steps, start=1):  # Print idx and transform  print(f'{idx}. {transform}') |
| Log Normalization |  | # Import numpy  import numpy as np  # Copy X\_train and X\_test into X\_train\_normed and X\_test\_normed  X\_train\_normed, X\_test\_normed = X\_train.copy(), X\_test.copy()  # Specify which column to normalize  col\_to\_normalize = '#col'  # Log normalization  for df\_ in [X\_train\_normed, X\_test\_normed]:  # Add log normalized column  df\_['#col'] = np.log(df\_[col\_to\_normalize])  # Drop the original column  df\_.drop(columns=col\_to\_normalize, inplace=True)  # Check the variance for X\_train\_normed  X\_train\_normed.var().round(3) |
| Logistic Regression |  | # Importing modules  from sklearn import linear\_model  # Instantiate LogisticRegression  logreg = linear\_model.LogisticRegression(  solver='liblinear',  random\_state=42  )  # Train the model  logreg.fit(X\_train\_normed, y\_train)  # AUC score for tpot model  logreg\_auc\_score = roc\_auc\_score(y\_test, logreg.predict\_proba(X\_test\_normed)[:, 1])  print(f'\nAUC score: {logreg\_auc\_score:.4f}') |
| Model Comparison |  | # Importing itemgetter  from operator import itemgetter  # Sort models based on their AUC score from highest to lowest  sorted(  [('tpot', tpot\_auc\_score), ('logreg', logreg\_auc\_score)],  key=itemgetter(1),  reverse=True  ) |
| Cross-validation |  | from sklearn.model\_selection import KFold, cross\_val\_score  # Set up our K-fold cross-validation  kf = KFold(10, random\_state = 10)  tree = DecisionTreeClassifier(random\_state=10)  logreg = LogisticRegression(random\_state=10)  # Train our models using KFold cv  tree\_score = cross\_val\_score(tree, pca\_projection, labels, cv = kf)  logit\_score = cross\_val\_score(logreg, pca\_projection, labels)  # Print the mean of each array of scores  print("Decision Tree:", tree\_score.mean(), "Logistic Regression:", logit\_score.mean()) |
| Bootstrap analysis | | |
| Bootstrap analysis |  | # A bootstrap analysis of the reduction of deaths due to handwashing  boot\_mean\_diff = []  for i in range(3000):  boot\_before = before\_proportion.sample(frac = 1, replace = True)  boot\_after = after\_proportion.sample(frac = 1, replace = True)  boot\_mean\_diff.append(boot\_after.mean() - boot\_before.mean())  # Calculating a 95% confidence interval from boot\_mean\_diff  confidence\_interval = pd.Series(boot\_mean\_diff).quantile([0.025, 0.975])  confidence\_interval |
| Text Analytics | | |
| Word Classification |  | # Set seed for reproducibility  import random; random.seed(53)  # Import all we need from sklearn  from sklearn.feature\_extraction.text import CountVectorizer, TfidfVectorizer  from sklearn.model\_selection import train\_test\_split  from sklearn.naive\_bayes import MultinomialNB  from sklearn.svm import LinearSVC  from sklearn import metrics  import pandas as pd  # Load data  tweet\_df = pd.read\_csv('#file')  # Create target  y = tweet\_df['#author']  # Split training and testing data  X\_train, X\_test, y\_train, y\_test = train\_test\_split(#tweet\_df['#status'], y, random\_state=53, test\_size=.33)  # Initialize count vectorizer  count\_vectorizer = CountVectorizer(stop\_words='english', min\_df=0.05, max\_df=0.9)  # Create count train and test variables  count\_train = count\_vectorizer.fit\_transform(X\_train)  count\_test = count\_vectorizer.transform(X\_test)  # Initialize tfidf vectorizer  tfidf\_vectorizer = TfidfVectorizer(stop\_words='english', min\_df=0.05, max\_df=0.9)  # Create tfidf train and test variables  tfidf\_train = tfidf\_vectorizer.fit\_transform(X\_train)  tfidf\_test = tfidf\_vectorizer.transform(X\_test)  # Create a MulitnomialNB model  tfidf\_nb = MultinomialNB()  tfidf\_nb.fit(tfidf\_train, y\_train)  # Run predict on your TF-IDF test data to get your predictions  tfidf\_nb\_pred = tfidf\_nb.predict(tfidf\_test)  # Calculate the accuracy of your predictions  tfidf\_nb\_score = metrics.accuracy\_score(y\_test, tfidf\_nb\_pred)  # Create a MulitnomialNB model  count\_nb = MultinomialNB()  count\_nb.fit(count\_train, y\_train)  # Run predict on your count test data to get your predictions  count\_nb\_pred = count\_nb.predict(count\_test)  # Calculate the accuracy of your predictions  count\_nb\_score = metrics.accuracy\_score(y\_test, count\_nb\_pred)  print('NaiveBayes Tfidf Score: ', tfidf\_nb\_score)  print('NaiveBayes Count Score: ', count\_nb\_score)  %matplotlib inline  from datasets.helper\_functions import plot\_confusion\_matrix  # Calculate the confusion matrices for the tfidf\_nb model and count\_nb models  tfidf\_nb\_cm = metrics.confusion\_matrix(y\_test, tfidf\_nb\_pred, labels=[#label1', ' label2'])  count\_nb\_cm = metrics.confusion\_matrix(y\_test, count\_nb\_pred, labels=[#label1', ' label2'])  # Plot the tfidf\_nb\_cm confusion matrix  plot\_confusion\_matrix(tfidf\_nb\_cm, classes=[#label1', ' label2'], title="TF-IDF NB Confusion Matrix")  # Plot the count\_nb\_cm confusion matrix without overwriting the first plot  plot\_confusion\_matrix(count\_nb\_cm , classes=[#label1', ' label2'], title="Count NB Confusion Matrix", figure=1)  # Create a LinearSVM model  tfidf\_svc = LinearSVC()  tfidf\_svc.fit(tfidf\_train, y\_train)  # Run predict on your tfidf test data to get your predictions  tfidf\_svc\_pred = tfidf\_svc.predict(tfidf\_test)  # Calculate your accuracy using the metrics module  tfidf\_svc\_score = metrics.accuracy\_score(y\_test, tfidf\_svc\_pred)  print("LinearSVC Score: %0.3f" % tfidf\_svc\_score)  # Calculate the confusion matrices for the tfidf\_svc model  svc\_cm = metrics.confusion\_matrix(y\_test, tfidf\_svc\_pred, labels=[#label1', ' label2'])  # Plot the confusion matrix using the plot\_confusion\_matrix function  plot\_confusion\_matrix(svc\_cm, classes=[#label1', ' label2'], title="TF-IDF LinearSVC Confusion Matrix")  from datasets.helper\_functions import plot\_and\_return\_top\_features  # Import pprint from pprint  from pprint import pprint  # Get the top features using the plot\_and\_return\_top\_features function and your top model and tfidf vectorizer  top\_features = plot\_and\_return\_top\_features(tfidf\_svc, tfidf\_vectorizer)  # pprint the top features  pprint(top\_features)  # Write two tweets as strings, one which you want to classify as Trump and one as Trudeau  #tweet1 = "#tweet"  #tweet2 = "#tweet"  # Vectorize each tweet using the TF-IDF vectorizer's transform method  # Note: `transform` needs the string in a list object (i.e. [tweet1])  #tweet1\_vectorized = tfidf\_vectorizer.transform([#tweet1])  #tweet2\_vectorized = tfidf\_vectorizer.transform([#tweet2])  # Call the predict method on your vectorized tweets  tweet1\_pred = tfidf\_svc.predict(#tweet1\_vectorized)  tweet2\_pred = tfidf\_svc.predict(#tweet2\_vectorized)  print("Predicted tweet 1", #tweet1\_pred)  print("Predicted tweet 2", #tweet2\_pred) |
| SQL | | |
| Create table and manual import of data |  | # Import Table, Column, String, Integer, Float, Boolean from sqlalchemy  from sqlalchemy import Table, Column, String, Integer, Float, Boolean  # Define a new table  #data = Table('data', metadata,  Column('#col1', String(255) , unique=True , nullable=False),  Column('#col2', Integer(), default=1),  Column('#col3', Float()),  Column('#col4', Boolean(), default=False)  )  # Use the metadata to create the table  #metadata.create\_all(#engine)  # Print table details  print(repr(#data))  # Build a list of dictionaries  #list = [  {'#col1': '#value1', '#col2': '#value2'},  {'#col1': '#value1', '#col2': '#value2'}  ]  # Build an insert statement for the data table  #query = insert(#table)  # Execute  #results = connection.execute(#query, #list)  # Print rowcount  print(#results.rowcount) |
| Import data |  | # import pandas  import pandas as pd  # read census.csv into a dataframe  #df = pd.read\_csv("#file", header=None)  # rename the columns of the dataframe  #df.columns = ['#col1', '#col2']  # append the data from df to the table via connection  #df.to\_sql(name='#table', con=#connection, if\_exists='#append/ fail / replace', index=#False) |
| Update records |  | # Build a statement to update  #update\_query = update(#table).values(#col = #condition)  # Append a where clause to limit it to records  #update\_query = #update\_ query.where(#table.columns.#col == '#condition')  # Execute the statement  #update\_results = #connection.execute(#update\_query)  # Print rowcount  print(results.rowcount) |
| Delete records |  | # Import delete, select  from sqlalchemy import delete, select  # Build a statement to empty the table  #delete\_query = delete(#table).where(  # Build a statement to delete records from the census table: delete\_stmt  delete\_stmt = delete(census)  # Append a where clause to target Men ('M') age 36: delete\_stmt  delete\_stmt = delete\_stmt.where(  and\_(#table.columns.#col1 == '#condition',  #table.columns.#col2 == '#condition')  )  # Execute the statement  #results = connection.execute(#delete\_query)  # Print affected rowcount  print(#results.rowcount) |
| Drop table |  | # Drop the table  #table.drop(#engine)  # Check to see if table exists  print(#table.exists(#engine))  # Drop all tables  metadata.drop\_all(#engine)  # Check to see if table exists  print(#engine.table\_names()) |
| Visualization | | |
| Boxplot | ggplot(data = #df, aes(x = #col, y = #col)) + geom\_boxplot() | #df.boxplot(column=['#Y'], by='#X')  plt.show() |
| Bar plot | ggplot(data = #df) + aes(x = #col, fill = #col) + geom\_bar(position='fill') + ylab('#label') | #df.plot.bar(x='#col') |
| Histogram |  | #df['#col'].plot(kind='hist', rot=#70, logx=True, logy=True) |
| Scatterplot |  | df.plot(kind='scatter', x='#X', y='#Y', rot=#70) |
| Dual axis |  | #Method 1:  #ax = #df.plot(x='#col1', y='#col1', label='#col')  #df.plot(x='#col1', y='#col1', label='#col', ax=#ax, kind='bar/line')  #ax.set\_ylabel('#label')  #Method 2:  #df\_aggregated.unstack().plot(title='#title', legend=True) |
| Matplotlib Visualization | | |
| Dual axis |  | # Prepare data  # Create plot  fig, (ax1, ax2) = plt.subplots(2, 1, sharex = True, figsize = (12,8))  # Plot and customize first subplot  #df1.plot(ax = ax1)  ax1.set(ylabel='#label', title=#title')  ax1.axhline(#df1.mean(), color='blue', linewidth=1, linestyle='-.')  # Plot and customize second subplot  #df2.plot(ax=ax2, color='gray')  ax2.set(xlabel='#label', ylabel='#label')  ax2.axhline(#df2.mean(), color = 'blue', linewidth = 1, linestyle = '-.')  # Show plot  plt.show() |
| Bands |  | # Create plot  fig = plt.figure(figsize = (8, 5))  # Plot and customize  ax = #df\_new.plot(marker='\*', markersize=14, linewidth=0, color='blue')  ax.set(ylim=['#y\_lower\_limit', '#y\_upper\_limit'],  xlim=['#x\_lower\_limit', '#x\_upper\_limit'],  ylabel='#label',  xlabel='#label',  title='#title')  ax.axhspan(#lower\_limit, #upper\_limit, color='green', alpha=#0.4)  # Show plot  plt.show() |
| Custom bins |  | # Prepare data  #bin = [#bin\_values]  #bin\_names = ['#bin\_names']  #bin\_colors = ['#green', 'yellow', 'orange', 'tomato', 'red']  # Create plot  fig, ax = plt.subplots(figsize=(8,5))  # Plot and customize  n, bins, zones = ax.hist(#df, bins=#bin, alpha=0.5)  for i in range(0, len(zones)):  zones[i].set\_facecolor(zone\_colors[i])  ax.set(title='#title', ylabel='#label')  ax.xaxis.set(ticks=#bin)  ax.set\_xticklabels(labels=#bin\_names, rotation='-30', ha='left')  # Show plot  plt.show() |
| Side-by-side bar |  | # Plotting 1st graph  plt.bar(#X1, #Y, width=0.25, color='#lightblue')  # Plotting 2nd graph  #X2 = [#X + 0.25 for #X in #list]  plt.bar(#X2, #Y, width=0.25, color='#pink') |
| Seaborn Visualization | | |
| Visualization |  | # Import seaborn library  import seaborn as sns  # Plot the data  sns.relplot(data = #df, x = '#col', y = '#col', kind = 'line / scatter', hue='#col')  # Plot the data  sns.barplot(data=#df, x='#col', y='#col', estimator=sum, ci=None) |
| Bokeh visualization | | |
| Visualization |  | import bokeh  from bokeh.plotting import output\_notebook, figure, show  output\_notebook(bokeh.resources.INLINE)  # Set up figure  p = figure(plot\_width=900, plot\_height=450, x\_axis\_type='datetime', tools='lasso\_select, box\_zoom, save, reset, wheel\_zoom',  toolbar\_location='above', x\_axis\_label='Date', y\_axis\_label='#col',  title='#title')  # Plot on figure  p.circle(#df['#col1'], #df['#col2'], color='black', nonselection\_fill\_alpha=0.2, nonselection\_fill\_color='grey')  p.line(#df['#col1'], #df['#col2'], color='black', alpha=1, line\_width=2, legend='#legend')  show(p) |
| Network Analysis | | |
| Networkx |  | # Importing modules  import networkx as nx  #Open multiple files  #g = [#list1]  #file\_list = ['#list2', '#list3']  for #file in #file\_list:  #df = pd.read\_csv(#file)  #g1 = nx.Graph()  for \_, edge in #df.iterrows():  #g1.add\_edge(edge['Source'], edge['Target'], weight=edge['weight'])  #g.append(#g1)  #g\_deg = nx.degree\_centrality(#g)  #g\_deg\_sorted = sorted(#g\_deg.items(), key=lambda x: x[1], reverse=True)  #dc = [nx.degree\_centrality(#graph) for #graph in #g]  #dc\_df = pd.DataFrame.from\_records(#dc)  #dc\_df[['#col']].plot()  # Creating a list of pagerank, betweenness centrality, degree centrality  #measures = [nx.pagerank(#graph),  nx.betweenness\_centrality(#graph, weight='weight'),  nx.degree\_centrality(#graph)]  # Creating the correlation DataFrame  #cor = pd.DataFrame.from\_records(#measures)  # Calculating the correlation  #cor.T.corr()  #p\_rank, #b\_cent, #d\_cent = #cor.idxmax(axis=1)  print(#p\_rank, #b\_cent, #d\_cent) |
| Spatial Analysis | | |
| Import data | # Load in the tidyverse, raster, and sf packages  library(tidyverse, sf, raster)  # Read the climate data from an rds file  #df <- read\_rds('#file')  # Have a look at the variables in the climate data  colnames(#df)  # Convert to SpatialPixelDataFrame for plotting  #df <- mutate(  .data = #df,  rasters = map(  .x = rasters,  ~ as\_tibble(as(.x, "SpatialPixelsDataFrame")))) %>%  unnest(cols = c(rasters)) |  |
| Plot map | library(ggthemes)  # Filter the data to plot  #df %>%  filter(#col %in% c(#value1, #value2)) %>%  # Create the plot  ggplot(aes(x = x, y = y)) + geom\_tile(aes(fill = #col)) +  # Style the plot with options ideal for maps  theme\_map() +  coord\_equal() +  facet\_grid(~ #col) + scale\_fill\_distiller(palette = "Spectral") +  theme(legend.title = element\_blank(), legend.position = "bottom") +  labs(title = "#title", caption = '#caption') |  |
| Projection | # Define geographical projections  #proj\_latlon <- st\_crs("+proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0")  #proj\_ukgrid <- st\_crs("+init=epsg:27700")  # Convert records to spatial points and project them  #df\_new <- mutate(#df,  #new\_col = map(#new\_col, ~ .x %>%  # Specify the current projection  st\_as\_sf(coords = c("x", "y"), crs = #proj\_latlon ) %>%  # Transform to new projection  st\_transform(crs = #proj\_ukgrid))) |  |
| Extract raster information | #df\_new <- map2\_df(  .x = #df[["#rasters"]],  .y = #df[["#col"]],  # extract the raster values at presence locations  ~ raster::extract(.x, .y) %>%  as\_tibble() %>%  mutate(observation = "#col")) |  |
| Data balancing | # Define helper function for creating pseudo-absence data  create\_pseudo\_absences <- function(rasters, n, ...) {  set.seed(12345)  sampleRandom(rasters, size = n \* 5, sp = TRUE) %>%  raster::extract(rasters, .) %>% as\_tibble() %>%  mutate(observation = "pseudo\_absence")  }  # Create pseudo-absence proportional to the total number of records per decade  #pseudo\_absence\_data <- pmap\_df(.l = #df, .f = create\_pseudo\_absences)  # Combine the two datasets  model\_data <- bind\_rows(#df1, #pseudo\_absence\_data) %>%  mutate(observation = factor(observation)) %>% na.omit() |  |
| Logistic regression with elastic net regularization | # Load caret and set a reproducible seed  library(caret)  set.seed(12345)  # Create a tuning grid with sets of hyperparameters to try  tuneGrid <- expand.grid(alpha = c(#0, #0.5, #1), lambda = c(#.003, #.01, #.03, #.06))  # Create settings for model training  trControl <- trainControl(method = 'repeatedcv', number = 5, repeats = 1,  classProbs = TRUE, verboseIter = FALSE, summaryFunction = twoClassSummary)  # Fit a statistical model to the data and plot  #model\_fit <- train(  observation ~ ., data = #model\_data,  method = "glmnet", family = "binomial", metric = "ROC",  tuneGrid = tuneGrid, trControl = trControl)  plot(#model\_fit) |  |
| Prediction | # Use our model to make a prediction  #df[["#prediction"]] <- predict(  object = #model\_fit,  newdata = #df,  type = "prob")[["#col"]]  head(#df) |  |
| Plot | library(viridis)  # Create the plot  #df %>%  ggplot(aes(x = x, y = y)) +  geom\_tile(aes(fill = #col)) +  # Style the plot with the appropriate settings for a map  theme\_map() +  coord\_equal() +  #scale\_fill\_distiller(palette = "Spectral") +  #theme(legend.title = element\_blank(), legend.position = "bottom") +  scale\_fill\_viridis(option = "A") + theme(legend.position = "bottom") +  # Add faceting by decade  facet\_grid(~ #col) +  labs(title = '#title', subtitle = '#subtitle',  caption = '#caption',  fill = '#legend')  # Display the plot  ggp\_changemap |  |
| Save file | | |
| Save file | write.csv(#df, ‘#file’) | #df.to\_csv('#file') |