**Comparison between R and Python codes**

Contents

[Install library 5](#_Toc27234259)

[Install library 5](#_Toc27234260)

[Import library 5](#_Toc27234261)

[Use library 5](#_Toc27234262)

[Retrieval from web 5](#_Toc27234263)

[Web retrieval 6](#_Toc27234264)

[Web scrape 8](#_Toc27234265)

[Using API 9](#_Toc27234266)

[Twitter API 9](#_Toc27234267)

[Read file 13](#_Toc27234268)

[Check current directory 13](#_Toc27234269)

[Read file 14](#_Toc27234270)

[CSV file 14](#_Toc27234271)

[JSON file 14](#_Toc27234272)

[XLS file 14](#_Toc27234273)

[TXT file 15](#_Toc27234274)

[SQL 16](#_Toc27234275)

[General file 26](#_Toc27234276)

[Pickle file 27](#_Toc27234277)

[SAS file 27](#_Toc27234278)

[STATA file 27](#_Toc27234279)

[HDF5 file 27](#_Toc27234280)

[MATLAB file 28](#_Toc27234281)

[Read head 30](#_Toc27234282)

[Read tail 30](#_Toc27234283)

[Read and print image 30](#_Toc27234284)

[Summary 30](#_Toc27234285)

[Describe 30](#_Toc27234286)

[Shape of dataframe 30](#_Toc27234287)

[Length of list 30](#_Toc27234288)

[Count values 31](#_Toc27234289)

[Count NA 31](#_Toc27234290)

[Number of columns 31](#_Toc27234291)

[Rename column 31](#_Toc27234292)

[Insert new column 31](#_Toc27234293)

[Drop column 31](#_Toc27234294)

[Merge dataframes 32](#_Toc27234295)

[DataFrame to list 34](#_Toc27234296)

[Print 34](#_Toc27234297)

[Data Manipulation 34](#_Toc27234298)

[Filter 34](#_Toc27234299)

[Remove duplicates 34](#_Toc27234300)

[Convert to NA 34](#_Toc27234301)

[Drop NA 35](#_Toc27234302)

[Fill NA 35](#_Toc27234303)

[Resample time series 35](#_Toc27234304)

[Type casting 36](#_Toc27234305)

[Locate value 36](#_Toc27234306)

[Locate largest and smallest values 36](#_Toc27234307)

[Split string 37](#_Toc27234308)

[Concatenate 37](#_Toc27234309)

[Group 37](#_Toc27234310)

[Aggregate 38](#_Toc27234311)

[Nest 38](#_Toc27234312)

[Sort 38](#_Toc27234313)

[Replace string 38](#_Toc27234314)

[Regular expression 39](#_Toc27234315)

[Impute numeric value 41](#_Toc27234316)

[Impute categorical value 41](#_Toc27234317)

[Reshape / Melt 41](#_Toc27234318)

[Pivot Table 41](#_Toc27234319)

[Stack 42](#_Toc27234320)

[Mutate 42](#_Toc27234321)

[Dummy variables 42](#_Toc27234322)

[Convert to dataframe 42](#_Toc27234323)

[Date manipulation 42](#_Toc27234324)

[Date extraction (Datetime) 43](#_Toc27234325)

[Month-Year extraction 43](#_Toc27234326)

[Date extraction (Pandas) 43](#_Toc27234327)

[Set date 43](#_Toc27234328)

[Functions 43](#_Toc27234329)

[If Elseif Else 44](#_Toc27234330)

[Loop 45](#_Toc27234331)

[iteritems 45](#_Toc27234332)

[Function 46](#_Toc27234333)

[Encoding / Recoding 46](#_Toc27234334)

[Encoding (Convert non-numeric to numeric) 47](#_Toc27234335)

[Encoding 47](#_Toc27234336)

[Scaling (MaxAbsScaler) 47](#_Toc27234337)

[Min-Max Scaling 48](#_Toc27234338)

[Standard Scaling 48](#_Toc27234339)

[Dimensionality Reduction 48](#_Toc27234340)

[Select K Best 48](#_Toc27234341)

[PCA 49](#_Toc27234342)

[K-Means Clustering 52](#_Toc27234343)

[Correlation 53](#_Toc27234344)

[Data Balancing 54](#_Toc27234345)

[Time Series 54](#_Toc27234346)

[Trend visualization 55](#_Toc27234347)

[Statistical tests 55](#_Toc27234348)

[Statistical tests 55](#_Toc27234349)

[Train-Test-Split 55](#_Toc27234350)

[Stratified 56](#_Toc27234351)

[Multi-label 56](#_Toc27234352)

[One VS Rest Classifier 57](#_Toc27234353)

[Linear Regression 57](#_Toc27234354)

[Linear Regression 58](#_Toc27234355)

[Cross-validation 60](#_Toc27234356)

[Lasso Regression 60](#_Toc27234357)

[Ridge Regression 61](#_Toc27234358)

[Confidence interval 63](#_Toc27234359)

[Supervised learning: Classification 65](#_Toc27234360)

[K-Nearest Neighbour 66](#_Toc27234361)

[Support Vector Machine 69](#_Toc27234362)

[Supervised learning: Classification 71](#_Toc27234363)

[K-Nearest Neighbour 72](#_Toc27234364)

[Classification 75](#_Toc27234365)

[Multiple Classifications 76](#_Toc27234366)

[Visualizing decision boundaries 79](#_Toc27234367)

[TPOT Classification 80](#_Toc27234368)

[Log Normalization 81](#_Toc27234369)

[Logistic Regression 81](#_Toc27234370)

[Train Test Split 81](#_Toc27234371)

[Logistic Regression 82](#_Toc27234372)

[Hyperparameter: Regularization 83](#_Toc27234373)

[L1 Regularization 84](#_Toc27234374)

[One VS Rest Classifier (Default); Multinomial 85](#_Toc27234375)

[Evaluation for Logistic Regression 87](#_Toc27234376)

[Cross-validation 89](#_Toc27234377)

[Grid Search 90](#_Toc27234378)

[GridSearch 93](#_Toc27234379)

[Confidence Interval 94](#_Toc27234380)

[Support Vector Machine 94](#_Toc27234381)

[Linear SVM 94](#_Toc27234382)

[Grid Search 95](#_Toc27234383)

[SGD Classifier 96](#_Toc27234384)

[Decision Tree 96](#_Toc27234385)

[Decision Tree 96](#_Toc27234386)

[Random Search CV 97](#_Toc27234387)

[Logistic Regression 98](#_Toc27234388)

[Model Comparison 98](#_Toc27234389)

[Cross-validation 99](#_Toc27234390)

[Bootstrap analysis 99](#_Toc27234391)

[Bootstrap analysis 99](#_Toc27234392)

[Loss Function 99](#_Toc27234393)

[Log Loss 100](#_Toc27234394)

[Hold Out 100](#_Toc27234395)

[Hold out 101](#_Toc27234396)

[Text Analytics 101](#_Toc27234397)

[Word Classification 102](#_Toc27234398)

[Sentiment Analysis 107](#_Toc27234399)

[Join all the items in a row 108](#_Toc27234400)

[Pipeline 110](#_Toc27234401)

[Hashing 113](#_Toc27234402)

[K Best Features, N-gram, Interaction terms, Hashing 114](#_Toc27234403)

[SQL 116](#_Toc27234404)

[Create table and manual import of data 117](#_Toc27234405)

[Import data 118](#_Toc27234406)

[Update records 118](#_Toc27234407)

[Delete records 119](#_Toc27234408)

[Drop table 119](#_Toc27234409)

[Visualization 119](#_Toc27234410)

[Boxplot 120](#_Toc27234411)

[Bar plot 120](#_Toc27234412)

[Histogram 120](#_Toc27234413)

[Scatterplot 120](#_Toc27234414)

[Dual axis 120](#_Toc27234415)

[Matplotlib Visualization 120](#_Toc27234416)

[Dual axis 121](#_Toc27234417)

[Bands 121](#_Toc27234418)

[Custom bins 122](#_Toc27234419)

[Side-by-side bar 122](#_Toc27234420)

[Seaborn Visualization 122](#_Toc27234421)

[Correlation plot / Heat map 123](#_Toc27234422)

[Line / Scatter plot 123](#_Toc27234423)

[Bar plot 123](#_Toc27234424)

[Count plot 123](#_Toc27234425)

[Bokeh visualization 123](#_Toc27234426)

[Visualization 124](#_Toc27234427)

[Network Analysis 124](#_Toc27234428)

[Networkx 125](#_Toc27234429)

[Spatial Analysis 127](#_Toc27234430)

[Import data 127](#_Toc27234431)

[Plot map 127](#_Toc27234432)

[Projection 128](#_Toc27234433)

[Extract raster information 128](#_Toc27234434)

[Data balancing 129](#_Toc27234435)

[Logistic regression with elastic net regularization 130](#_Toc27234436)

[Prediction 130](#_Toc27234437)

[Plot 131](#_Toc27234438)

[Save file 131](#_Toc27234439)

[Save file 131](#_Toc27234440)

| **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',  squeeze=False #true for Series)  #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])  #Method 2:  # Loading json module  import json  # Load data  #df = json.loads(open('#file').read())  # Pretty-printing the results  print(json.dumps(#df, indent=1))  # Extracting all from df  #df\_new = set([#x['#col\_new'] for #x in #df [0]['#col']])  [user\_mention['screen\_name'] for tweet in tweets for user\_mention in tweet['entities']['user\_mentions']] 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()  #Method 3:  # Importing modules  from collections import Counter  # Counting occcurrences/ getting frequency dist of all names and hashtags  for item in [names, hashtags]:  c = Counter(item)  # Inspecting the 10 most common items in c  print (c.most\_common(10), "\n") |
| 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) |
| Index |  | #Method 1:  # Assign the string 'MONTHS' to sales.index.name  sales.index.name = 'MONTHS'  # Print the sales DataFrame  print(sales)  # Assign the string 'PRODUCTS' to sales.columns.name  sales.columns.name = 'PRODUCTS'  # Print the sales dataframe again  print(sales)  #Method 2:  # Generate the list of months: months  months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']  # Assign months to sales.index  sales.index = months  #Method 3:  # Set the index to be the columns ['state', 'month']: sales  sales = sales.set\_index(['state', 'month'])  # Sort the MultiIndex: sales  sales = sales.sort\_index() |
| 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 4:  #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) | #Method 1:  #col\_list = ['#col1', '#col2']  # Filter for rows containing these metrics  #df\_f = #df[#df.#colname.isin(#col\_list)]  #Method 2:  # Filter 'Units' where the sum is > 35: by\_com\_filt  by\_com\_filt = by\_company.filter(lambda g:g['Units'].sum() > 35)  print(by\_com\_filt) |
| Remove duplicates |  | # Drop the duplicates  #df\_new = #df.drop\_duplicates()  # Print info  print(#df\_new.info()) |
| Convert to NA |  | # Convert '?' to NaN  #df[#df == '?'] = np.nan |
| Drop NA |  | #Method 1:  #Drop missing values  #df.dropna(axis=0, how='any')  #Method 2:  # Drop rows in df with how='any' and print the shape  print(df.dropna(how='any').shape)  # Drop rows in df with how='all' and print the shape  print(df.dropna(how='all').shape)  # Drop columns in titanic with less than 1000 non-missing values  print(titanic.dropna(thresh=1000, axis='columns').info()) |
| 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')  #Method 3:  # Define the lambda function: categorize\_label  #categorize\_label = lambda #x: #x.astype('category')  # Convert df[LABELS] to a categorical type  #df[#LABELS] = #df[#LABELS].apply(#categorize\_label, axis=0)  # Print the converted dtypes  print(#df[#LABELS].dtypes)  # Get the columns that are features in the original df  #NON\_LABELS = [#c for #c in #df.columns if #c not in #LABELS] |
| Locate value |  | #Method 1:  #df['#col'].iloc[#row\_num]  #Method 2: #Multi-index  all\_month2 = sales.loc[(slice(None), 2), :]  #Method 3:  # Slice the row labels 'Potter' to 'Perry' in reverse order: p\_counties\_rev  p\_counties\_rev = election.loc['Potter':'Perry':-1] |
| 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)  #Method 3:  #df['#col'].str.split('#.', expand=True)[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()  #Method 3:  # Create a groupby object: by\_day  by\_day = sales.groupby(sales.index.strftime('%a')) |
| 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()  #Method 4:  # Calculate number of unique values for each label  #num\_nunique = #df[#LABELS].apply(pd.Series.nunique)  #Method 5:  # Read the CSV file into a DataFrame and sort the index: gapminder  gapminder = pd.read\_csv('gapminder.csv', index\_col=['Year','region','Country']).sort\_index()  # Group gapminder by 'Year' and 'region': by\_year\_region  by\_year\_region = gapminder.groupby(level=['Year', 'region'])  # Define the function to compute spread: spread  def spread(series):  return series.max() - series.min()  # Create the dictionary: aggregator  aggregator = {'population':'sum', 'child\_mortality':'mean', 'gdp':spread}  # Aggregate by\_year\_region using the dictionary: aggregated  aggregated = by\_year\_region.agg(aggregator)  # Print the last 6 entries of aggregated  print(aggregated.tail(6))  #Method 6: Intersection of df  #df1.intersection(#df2) |
| 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')  #Method 3:  under10 = (titanic['age'] < 10).map({True:'under 10', False:'over 10'})  # Create the dictionary: red\_vs\_blue  red\_vs\_blue = {'Obama':'blue', 'Romney':'red'}  # Use the dictionary to map the 'winner' column to the new column: election['color']  election['color'] = election['winner'].map(red\_vs\_blue)  #Method 4:  # Loop through columns  for col in trends.columns:  # Only modify columns that have the "<" sign  if("<" in trends[col].to\_string()):  # Remove "<" and convert dtype to integer  trends[col] = trends[col].str.replace("<", "") |
| 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 |  | #Method 1:  #df.fillna(#df.mean(), inplace=True)  #Method 2:  # Write a function that imputes median  def impute\_median(series):  return series.fillna(series.median())  # Impute age and assign to titanic['age']  titanic.age = by\_sex\_class['age'].transform(impute\_median) |
| 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 |  | #Method 1:  # 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'])  #Method 2:  # Reshape X  X = #df.#col.values.reshape(#-1,#1) |
| 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, margins=True).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, aggfunc='count').reset\_index() |
| Stack |  | # Unstack users by 'city': bycity #One index  bycity = users.unstack('city')  # Stack bycity by 'city' and print it #Multi-column  newusers = bycity.stack('city')  # Swap the levels of the index of newusers: newusers  newusers = newusers.swaplevel(0, 1)  # Print newusers and verify that the index is not sorted  print(newusers)  # Sort the index of newusers: newusers  newusers = newusers.sort\_index() |
| 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, prefix\_sep='\_') |
| 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]) |
| Encoding |  | #Method 1: get\_dummies in pandas  # Create dummy variables  #df = pd.get\_dummies(#df, drop\_first=True)  #Method 2:  #OneHotEncoder in sklearn |
| Ordinal variable |  | pd.Categorical(values = medals.Medal, categories=['Bronze', 'Silver', 'Gold'], ordered=True) |
| Scaling (MaxAbsScaler) |  | # Import scale  from sklearn.preprocessing import scale  # Scale the features: X\_scaled  X\_scaled = scale(X) |
| 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) |
| Z-Score |  | # Import zscore  from scipy.stats import zscore  # Group gapminder\_2010: standardized  standardized = gapminder\_2010.groupby('region')['life','fertility'].transform(zscore)  # Construct a Boolean Series to identify outliers: outliers  outliers = (standardized['life'] < -3) | (standardized['fertility'] > 3)  # Filter gapminder\_2010 by the outliers: gm\_outliers  gm\_outliers = gapminder\_2010.loc[outliers]  # Print gm\_outliers  print(gm\_outliers) |
| Dimensionality Reduction | | |
| Select K Best |  | #Chi-squared test to select K best features  from sklearn.feature\_selection import SelectKBest, chi2  #X\_new = SelectKBest(chi2, k=20).fit\_transform(#X, #y) |
| 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) |
| Visualizing PCA |  | # Perform the necessary imports  import matplotlib.pyplot as plt  from scipy.stats import pearsonr  # Assign the 0th column of grains: width  width = grains[:,0]  # Assign the 1st column of grains: length  length = grains[:,1]  # Scatter plot width vs length  plt.scatter(width, length)  plt.axis('equal')  plt.show()  # Calculate the Pearson correlation  correlation, pvalue = pearsonr(width, length)  # Display the correlation  print(correlation)  # Import PCA  from sklearn.decomposition import PCA  # Create PCA instance: model  model = PCA()  # Apply the fit\_transform method of model to grains: pca\_features  pca\_features = model.fit\_transform(grains)  # Assign 0th column of pca\_features: xs  xs = pca\_features[:,0]  # Assign 1st column of pca\_features: ys  ys = pca\_features[:,1]  # Scatter plot xs vs ys  plt.scatter(xs, ys)  plt.axis('equal')  plt.show()  # Calculate the Pearson correlation of xs and ys  correlation, pvalue = pearsonr(xs, ys)  # Display the correlation  print(correlation)  # Make a scatter plot of the untransformed points  plt.scatter(grains[:,0], grains[:,1])  # Create a PCA instance: model  model = PCA()  # Fit model to points  model.fit(grains)  # Get the mean of the grain samples: mean  mean = model.mean\_  # Get the first principal component: first\_pc  first\_pc = model.components\_[0,:]  # Plot first\_pc as an arrow, starting at mean  plt.arrow(mean[0], mean[1], first\_pc[0], first\_pc[1], color='red', width=0.01)  # Keep axes on same scale  plt.axis('equal')  plt.show() |
| PCA Pipeline |  | # Perform the necessary imports  from sklearn.decomposition import PCA  from sklearn.preprocessing import StandardScaler  from sklearn.pipeline import make\_pipeline  import matplotlib.pyplot as plt  # Create scaler: scaler  scaler = StandardScaler()  # Create a PCA instance: pca  pca = PCA()  # Create pipeline: pipeline  pipeline = make\_pipeline(scaler, pca)  # Fit the pipeline to 'samples'  pipeline.fit(samples)  # Plot the explained variances  features = range(0, pca.n\_components\_)  plt.bar(features, pca.explained\_variance\_)  plt.xlabel('PCA feature')  plt.ylabel('variance')  plt.xticks(features)  plt.show() |
| PCA for Sparse Matrix |  | # Import TfidfVectorizer  from sklearn.feature\_extraction.text import TfidfVectorizer  # Create a TfidfVectorizer: tfidf  tfidf = TfidfVectorizer()  # Apply fit\_transform to document: csr\_mat  csr\_mat = tfidf.fit\_transform(documents)  # Print result of toarray() method  print(csr\_mat.toarray())  # Get the words: words  words = tfidf.get\_feature\_names()  # Print words  print(words)  # Perform the necessary imports  from sklearn.decomposition import TruncatedSVD  from sklearn.cluster import KMeans  from sklearn.pipeline import make\_pipeline  # Create a TruncatedSVD instance: svd  svd = TruncatedSVD(n\_components=50)  # Create a KMeans instance: kmeans  kmeans = KMeans(n\_clusters=6)  # Create a pipeline: pipeline  pipeline = make\_pipeline(svd, kmeans)  # Import pandas  import pandas as pd  # Fit the pipeline to articles  pipeline.fit(articles)  # Calculate the cluster labels: labels  labels = pipeline.predict(articles)  # Create a DataFrame aligning labels and titles: df  df = pd.DataFrame({'label': labels, 'article': titles})  # Display df sorted by cluster label  print(df.sort\_values('label')) |
| Non-Negative Matrix Factorisation for Text Analytics |  | # Import NMF  from sklearn.decomposition import NMF  # Create an NMF instance: model  model = decomposition.NMF(n\_components=6)  # Fit the model to articles  model.fit(articles)  # Transform the articles: nmf\_features  nmf\_features = model.transform(articles)  # Print the NMF features  print(nmf\_features)  # Import pandas  import pandas as pd  # Create a pandas DataFrame: df  df = pd.DataFrame(nmf\_features, index=titles)  # Print the row for 'Anne Hathaway'  print(df.loc['Anne Hathaway'])  # Print the row for 'Denzel Washington'  print(df.loc['Denzel Washington']) |
| Non-Negative Matrix Factorisation for Image Recognition |  | # Import pandas  import pandas as pd  # Create a DataFrame: components\_df  components\_df = pd.DataFrame(model.components\_, columns=words)  # Print the shape of the DataFrame  print(components\_df.shape)  # Select row 3: component  component = components\_df.iloc[3]  # Print result of nlargest  print(component.nlargest())  # Import pyplot  from matplotlib import pyplot as plt  # Select the 0th row: digit  digit = samples[0, :]  # Print digit  print(digit)  # Reshape digit to a 13x8 array: bitmap  bitmap = digit.reshape(13, 8)  # Print bitmap  print(bitmap)  # Use plt.imshow to display bitmap  plt.imshow(bitmap, cmap='gray', interpolation='nearest')  plt.colorbar()  plt.show()  # Import NMF  from sklearn.decomposition import NMF  # Create an NMF model: model  model = NMF(n\_components=7)  # Apply fit\_transform to samples: features  features = model.fit\_transform(samples)  # Call show\_as\_image on each component  for component in model.components\_:  show\_as\_image(component)  # Assign the 0th row of features: digit\_features  digit\_features = features[0,:]  # Print digit\_features  print(digit\_features) |
| Cosine similarity |  | # Perform the necessary imports  import pandas as pd  from sklearn.preprocessing import normalize  # Normalize the NMF features: norm\_features  norm\_features = normalize(nmf\_features)  # Create a DataFrame: df  df = pd.DataFrame(norm\_features, index=titles)  # Select the row corresponding to 'Cristiano Ronaldo': article  article = df.loc['Cristiano Ronaldo']  # Compute the dot products: similarities  similarities = df.dot(article)  # Display those with the largest cosine similarity  print(similarities.nlargest())  # Perform the necessary imports  from sklearn.decomposition import NMF  from sklearn.preprocessing import Normalizer, MaxAbsScaler  from sklearn.pipeline import make\_pipeline  # Create a MaxAbsScaler: scaler  scaler = MaxAbsScaler()  # Create an NMF model: nmf  nmf = NMF(n\_components=20)  # Create a Normalizer: normalizer  normalizer = Normalizer()  # Create a pipeline: pipeline  pipeline = make\_pipeline(scaler, nmf, normalizer)  # Apply fit\_transform to artists: norm\_features  norm\_features = pipeline.fit\_transform(artists)  # Import pandas  import pandas as pd  # Create a DataFrame: df  df = pd.DataFrame(norm\_features, index=artist\_names)  # Select row of 'Bruce Springsteen': artist  artist = df.loc['Bruce Springsteen']  # Compute cosine similarities: similarities  similarities = df.dot(artist)  # Display those with highest cosine similarity  print(similarities.nlargest()) |
| 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() |
| Rolling average |  | # Smooth the data with rolling means  trends.rolling(window=12).mean().plot() |
| Statistical tests | | |
| Statistical tests | chisq.test(#df$X, #df$Y)  t.test(#df$Y ~ #df$X) |  |
| Train-Test-Split | | |
| Stratified |  | # 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() |
| Multi-label |  | # Import train\_test\_split method  from sklearn.model\_selection import multilabel\_train\_test\_split  # Get labels and convert to dummy variables: label\_dummies  label\_dummies = pd.get\_dummies(df[LABELS])  # Create training and test sets  X\_train, X\_test, y\_train, y\_test = multilabel\_train\_test\_split(numeric\_data\_only, label\_dummies, size=0.2, seed=123) |
| One VS Rest Classifier |  | # Import classifiers  from sklearn.linear\_model import LogisticRegression  from sklearn.multiclass import OneVsRestClassifier  # Get labels and convert to dummy variables: label\_dummies  label\_dummies = pd.get\_dummies(df[LABELS])  # Create training and test sets  X\_train, X\_test, y\_train, y\_test = multilabel\_train\_test\_split(numeric\_data\_only, label\_dummies, size=0.2, seed=123)  # Instantiate the classifier: clf  clf = OneVsRestClassifier(LogisticRegression())  # Fit the classifier to the training data  clf.fit(X\_train, y\_train)  # Print the accuracy  print("Accuracy: {}".format(clf.score(X\_test, y\_test))) |
| Linear Regression | | |
| Linear Regression |  | # Import LinearRegression  from sklearn.linear\_model import LinearRegression  from sklearn.metrics import mean\_squared\_error  from sklearn.model\_selection import train\_test\_split  # 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))  # Print R^2  print(#regr.score(#X\_train, #y\_train))  # Compute and print R^2 and RMSE  print("R^2: {}".format(#regr.score(#X\_test, #y\_test)))  #rmse = np.sqrt(mean\_squared\_error(#y\_test, #y\_pred))  print("Root Mean Squared Error: {}".format(#rmse))  # Create the prediction space  #prediction\_space = np.linspace(min(#X\_train), max(#X\_train)).reshape(-1,1)  # Compute predictions over the prediction space  #y\_pred = #regr.predict(#prediction\_space)  # Plot regression line  plt.plot(#prediction\_space, #y\_pred, color='#black', linewidth=#3)  plt.show() |
| Cross-validation |  | # Import the necessary modules  from sklearn.linear\_model import LinearRegression  from sklearn.model\_selection import cross\_val\_score  # Create a linear regression object: reg  reg = LinearRegression()  # Compute 5-fold cross-validation scores: cv\_scores  cv\_scores = cross\_val\_score(reg, X, y, cv=5)  # Print the 5-fold cross-validation scores  print(cv\_scores)  print("Average 5-Fold CV Score: {}".format(np.mean(cv\_scores)))  %timeit |
| Lasso Regression |  | # Import Lasso  from sklearn.linear\_model import Lasso  # Instantiate a lasso regressor: lasso  #lasso = Lasso(alpha=#0.4, normalize=#True)  # Fit the regressor to the data  lasso.fit(#X, #y)  # Compute and print the coefficients  #lasso\_coef = #lasso.coef\_  print(#lasso\_coef)  # Plot the coefficients  plt.plot(range(len(#df\_columns)), #lasso\_coef)  plt.xticks(range(len(#df\_columns)), #df\_columns.values, rotation=#60)  plt.margins(0.02)  plt.show() |
| Ridge Regression |  | # Import necessary modules  from sklearn.linear\_model import Ridge  from sklearn.model\_selection import cross\_val\_score  # Setup the array of alphas and lists to store scores  alpha\_space = np.logspace(-4, 0, 50)  ridge\_scores = []  ridge\_scores\_std = []  # Create a ridge regressor: ridge  ridge = Ridge(alpha=True, normalize=True)  # Compute scores over range of alphas  for alpha in alpha\_space:  # Specify the alpha value to use: ridge.alpha  ridge.alpha = alpha    # Perform 10-fold CV: ridge\_cv\_scores  ridge\_cv\_scores = cross\_val\_score(ridge, X, y, cv=10)    # Append the mean of ridge\_cv\_scores to ridge\_scores  ridge\_scores.append(np.mean(ridge\_cv\_scores))    # Append the std of ridge\_cv\_scores to ridge\_scores\_std  ridge\_scores\_std.append(np.std(ridge\_cv\_scores))  # Create user-defined function  def display\_plot(cv\_scores, cv\_scores\_std):  fig = plt.figure()  ax = fig.add\_subplot(1,1,1)  ax.plot(alpha\_space, cv\_scores)  std\_error = cv\_scores\_std / np.sqrt(10)  ax.fill\_between(alpha\_space, cv\_scores + std\_error, cv\_scores - std\_error, alpha=0.2)  ax.set\_ylabel('CV Score +/- Std Error')  ax.set\_xlabel('Alpha')  ax.axhline(np.max(cv\_scores), linestyle='--', color='.5')  ax.set\_xlim([alpha\_space[0], alpha\_space[-1]])  ax.set\_xscale('log')  plt.show()  # Display the plot  display\_plot(ridge\_scores, ridge\_scores\_std) |
| 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) |
| Supervised learning: Classification | | |
| K-Nearest Neighbour |  | #Require dataframe, number of neighbours, features need to be continuous, no missing values  # Import necessary modules  from sklearn.neighbors import KNeighborsClassifier  from sklearn.model\_selection import train\_test\_split  from sklearn.metrics import classification\_report, confusion\_matrix  # Create arrays for the features and the response variable  y = df['#Y'].values  X = df.drop('#col\_categorical', axis=1).values  # Split into training and test set  X\_train, X\_test, y\_train, y\_test = train\_test\_split(#X, #y, test\_size = #0.2, random\_state=#43, stratify=#y)  # Create a k-NN classifier with 6 neighbors  #knn = KNeighborsClassifier(n\_neighbors=#6)  # Fit the classifier to the data  #knn.fit(#X\_train, #y\_train)  # Predict the labels for the training data X  #y\_pred = #knn.predict(#X\_test)  # Predict and print the label for the new data point X\_new  #new\_prediction = #knn.predict(#X\_new)  print("Prediction: {}".format(#new\_prediction))  # Print the accuracy  print(#knn.score(#X\_test, #y\_test))  # Setup arrays to store train and test accuracies  #neighbors = np.arange(1, 9)  #train\_accuracy = np.empty(len(#neighbors))  #test\_accuracy = np.empty(len(#neighbors))  # Loop over different values of k  for #i, #k in enumerate(#neighbors):  # Setup a k-NN Classifier with k neighbors: knn  #knn = KNeighborsClassifier(n\_neighbors=#k)  # Fit the classifier to the training data  #knn.fit(#X\_train, #y\_train)    #Compute accuracy on the training set  #train\_accuracy[i] = #knn.score(#X\_train, #y\_train)  #Compute accuracy on the testing set  #test\_accuracy[i] = #knn.score(#X\_test, #y\_test)  # Generate plot  plt.title('k-NN: Varying Number of Neighbors')  plt.plot(#neighbors, #test\_accuracy, label = '#Testing Accuracy')  plt.plot(#neighbors, #train\_accuracy, label = '#Training Accuracy')  plt.legend()  plt.xlabel('#Number of Neighbors')  plt.ylabel('#Accuracy')  plt.show()  # Generate the confusion matrix and classification report  print(confusion\_matrix(y\_test, y\_pred))  print(classification\_report(y\_test, y\_pred))  # Import the necessary modules  from sklearn.preprocessing import StandardScaler  from sklearn.pipeline import Pipeline  # Setup the pipeline steps: steps  steps = [('scaler', StandardScaler()),  ('knn', KNeighborsClassifier())]    # Create the pipeline: pipeline  pipeline = Pipeline(steps)  # Create train and test sets  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)  # Fit the pipeline to the training set: knn\_scaled  knn\_scaled = pipeline.fit(X\_train, y\_train)  # Instantiate and fit a k-NN classifier to the unscaled data  knn\_unscaled = KNeighborsClassifier().fit(X\_train, y\_train)  # Compute and print metrics  print('Accuracy with Scaling: {}'.format(knn\_scaled.score(X\_test, y\_test)))  print('Accuracy without Scaling: {}'.format(knn\_unscaled.score(X\_test, y\_test))) |
| Support Vector Machine |  | # Import the Imputer module  from sklearn.preprocessing import Imputer  from sklearn.svm import SVC  # Setup the Imputation transformer: imp  imp = Imputer(missing\_values='NaN', strategy='most\_frequent', axis=0)  # Instantiate the SVC classifier: clf  clf = SVC()  # Setup the pipeline with the required steps: steps  steps = [('imputation', imp),  ('SVM', clf)]  # Import necessary modules  from sklearn.preprocessing import Imputer, Scale  from sklearn.pipeline import Pipeline  from sklearn.svm import SVC  # Setup the pipeline steps: steps  steps = [('imputation', Imputer(missing\_values='NaN', strategy='most\_frequent', axis=0)), ('scaler', StandardScaler())  ('SVM', SVC())]  # Create the pipeline: pipeline  pipeline = Pipeline(steps)  # Specify the hyperparameter space  parameters = {'SVM\_\_C':[1, 10, 100],  'SVM\_\_gamma':[0.1, 0.01]}  # Create training and test sets  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)  # Instantiate the GridSearchCV object: cv  cv = GridSearchCV(pipeline, parameters)  # Fit the pipeline to the train set  cv.fit(X\_train, y\_train)  # Predict the labels of the test set  y\_pred = pipeline.predict(X\_test)  # Compute metrics  print(classification\_report(y\_test, y\_pred))  # Compute and print metrics  print("Accuracy: {}".format(cv.score(X\_test, y\_test)))  print(classification\_report(y\_test, y\_pred))  print("Tuned Model Parameters: {}".format(cv.best\_params\_)) |
| Unsupervised Learning: Classification | | |
| K Means Clustering |  | # Import KMeans  from sklearn.cluster import KMeans  ks = range(1, 6)  inertias = []  for k in ks:  # Create a KMeans instance with k clusters: model  model = KMeans(n\_clusters=k)    # Fit model to samples  model.fit(samples)    # Append the inertia to the list of inertias  inertias.append(model.inertia\_)    # Plot ks vs inertias  plt.plot(ks, inertias, '-o')  plt.xlabel('number of clusters, k')  plt.ylabel('inertia')  plt.xticks(ks)  plt.show()  # Determine the cluster labels of new\_points: labels  labels = model.predict(new\_points)  # Print cluster labels of new\_points  print(labels)  # Import pyplot  import matplotlib.pyplot as plt  # Assign the columns of new\_points: xs and ys  xs = new\_points[:,0]  ys = new\_points[:,1]  # Make a scatter plot of xs and ys, using labels to define the colors  plt.scatter(xs, ys, c=labels, alpha=0.5)  # Assign the cluster centers: centroids  centroids = model.cluster\_centers\_  # Assign the columns of centroids: centroids\_x, centroids\_y  centroids\_x = centroids[:,0]  centroids\_y = centroids[:,1]  # Make a scatter plot of centroids\_x and centroids\_y  plt.scatter(centroids\_x, centroids\_y, marker='D', s=50)  plt.show()  # Use fit\_predict to fit model and obtain cluster labels: labels  labels = model.fit\_predict(samples)  # Create a DataFrame with labels and varieties as columns: df  df = pd.DataFrame({'labels': labels, 'varieties': varieties})  # Create crosstab: ct  ct = pd.crosstab(df['labels'], df['varieties'])  # Display ct  print(ct) |
| K Means Clustering Pipeline |  | # Perform the necessary imports  from sklearn.pipeline import make\_pipeline  from sklearn.preprocessing import StandardScaler  from sklearn.cluster import KMeans  # Import Normalizer  from sklearn.preprocessing import Normalizer  # Create a normalizer: normalizer #Rescale each sample independently of other category of samples  normalizer = Normalizer()  # Create scaler: scaler #Standardize features – scale to unit variance  scaler = StandardScaler()  # Create KMeans instance: kmeans  kmeans = KMeans(n\_clusters=4)  # Create pipeline: pipeline  pipeline = make\_pipeline(normalizer / scaler, kmeans)  # Import pandas  import pandas as pd  # Fit the pipeline to samples  pipeline.fit(samples)  # Calculate the cluster labels: labels  labels = pipeline.predict(samples)  # Create a DataFrame with labels and species as columns: df  df = pd.DataFrame({'labels': labels, 'species': species})  # Create crosstab: ct  ct = pd.crosstab(df['labels'], df['species'])  # Display ct  print(ct)  # Display df sorted by cluster label  print(df.sort\_values(['labels'])) |
| Hierarchical clustering |  | # Perform the necessary imports  from scipy.cluster.hierarchy import linkage, dendrogram  import matplotlib.pyplot as plt  # Import normalize  from sklearn.preprocessing import normalize  # Normalize the movements: normalized\_movements  normalized\_samples = normalize(samples)  # Calculate the linkage: mergings  mergings = linkage(normalized\_samples, method='complete')  # Plot the dendrogram, using varieties as labels  dendrogram(mergings,  labels=varieties,  leaf\_rotation=90,  leaf\_font\_size=6,  )  plt.show()  # Perform the necessary imports  import pandas as pd  from scipy.cluster.hierarchy import fcluster  # Use fcluster to extract labels: labels  labels = fcluster(mergings, 6, criterion='distance')  # Create a DataFrame with labels and varieties as columns: df  df = pd.DataFrame({'labels': labels, 'varieties': varieties})  # Create crosstab: ct  ct = pd.crosstab(df['labels'], df['varieties'])  # Display ct  print(ct) |
| t-SNE |  | # Import TSNE  from sklearn.manifold import TSNE  # Create a TSNE instance: model  model = TSNE(learning\_rate=200)  # Apply fit\_transform to samples: tsne\_features  tsne\_features = model.fit\_transform(samples)  # Select the 0th feature: xs  xs = tsne\_features[:,0]  # Select the 1st feature: ys  ys = tsne\_features[:,1]  # Scatter plot, coloring by variety\_numbers  #plt.scatter(xs, ys, c=variety\_numbers)  plt.scatter(xs, ys)  plt.show()  # Annotate the points  for x, y, company in zip(xs, ys, companies):  plt.annotate(company, (x, y), fontsize=5, alpha=0.75)  plt.show() |
| Classification | | |
| Multiple Classifications |  | from sklearn.linear\_model import LogisticRegression  from sklearn.svm import SVC, LinearSVC  from sklearn.neighbors import KNeighborsClassifier  # Define the classifiers  classifiers = [LogisticRegression(), LinearSVC(), SVC(), KNeighborsClassifier()]  # Fit the classifiers  for c in classifiers:  c.fit(X, y)  # Plot the classifiers  plot\_4\_classifiers(X, y, classifiers)  plt.show()  import numpy as np  import matplotlib.pyplot as plt  from sklearn import svm, datasets  def make\_meshgrid(x, y, h=.02):  """Create a mesh of points to plot in  Parameters  ----------  x: data to base x-axis meshgrid on  y: data to base y-axis meshgrid on  h: stepsize for meshgrid, optional  Returns  -------  xx, yy : ndarray  """  x\_min, x\_max = x.min() - 1, x.max() + 1  y\_min, y\_max = y.min() - 1, y.max() + 1  xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h),  np.arange(y\_min, y\_max, h))  return xx, yy  def plot\_contours(ax, clf, xx, yy, \*\*params):  """Plot the decision boundaries for a classifier.  Parameters  ----------  ax: matplotlib axes object  clf: a classifier  xx: meshgrid ndarray  yy: meshgrid ndarray  params: dictionary of params to pass to contourf, optional  """  Z = clf.predict(np.c\_[xx.ravel(), yy.ravel()])  Z = Z.reshape(xx.shape)  out = ax.contourf(xx, yy, Z, \*\*params)  return out  def plot\_4\_classifiers(X, y, clf)  # Set-up 2x2 grid for plotting.  fig, sub = plt.subplots(2, 2)  plt.subplots\_adjust(wspace=0.4, hspace=0.4)  X0, X1 = X[:, 0], X[:, 1]  xx, yy = make\_meshgrid(X0, X1)  for clf, title, ax in zip(models, titles, sub.flatten()):  plot\_contours(ax, clf, xx, yy,  cmap=plt.cm.coolwarm, alpha=0.8)  ax.scatter(X0, X1, c=y, cmap=plt.cm.coolwarm, s=20, edgecolors='k')  ax.set\_xlim(xx.min(), xx.max())  ax.set\_ylim(yy.min(), yy.max())  ax.set\_xlabel('Sepal length')  ax.set\_ylabel('Sepal width')  ax.set\_xticks(())  ax.set\_yticks(())  ax.set\_title(title)  plt.show() |
| Visualizing decision boundaries |  | # Set the coefficients  model.coef\_ = np.array([[-1,0.5]])  model.intercept\_ = np.array([-2])  # Plot the data and decision boundary  plot\_classifier(X,y,model)  # Print the number of errors  num\_err = np.sum(y != model.predict(X))  print("Number of errors:", num\_err) |
| 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 | | |
| 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  from sklearn.metrics import confusion\_matrix, classification\_report  # Instantiate a LogisticRegression classifier with default parameter values  logreg = LogisticRegression(C = 0.1, penalty='l1', random\_state = 10, multi\_class='multinomial', solver='lbfgs')  # Fit logreg to the train set  logreg.fit(rescaledX\_train, y\_train)  # Predict the labels of the test set: y\_pred  y\_pred = logreg.predict(X\_test)  # Compute and print the confusion matrix and classification report  print(confusion\_matrix(y\_test, y\_pred))  print(classification\_report(y\_test, y\_pred)) |
| Hyperparameter: Regularization |  | # Train and validaton errors initialized as empty list  train\_errs = list()  valid\_errs = list()  # Loop over values of C\_value  for C\_value in [0.001, 0.01, 0.1, 1, 10, 100, 1000]:  # Create LogisticRegression object and fit  lr = LogisticRegression(C = C\_value)  lr.fit(X\_train, y\_train)    # Evaluate error rates and append to lists  train\_errs.append( 1.0 - lr.score(X\_train, y\_train) )  valid\_errs.append( 1.0 - lr.score(X\_valid, y\_valid) )    # Plot results  plt.semilogx(C\_values, train\_errs, C\_values, valid\_errs)  plt.legend(("train", "validation"))  plt.show() |
| L1 Regularization |  | # Specify L1 regularization  lr = LogisticRegression(penalty=  'l1')  # Instantiate the GridSearchCV object and run the search  searcher = GridSearchCV(lr, {'C':[0.001, 0.01, 0.1, 1, 10]})  searcher.fit(X\_train, y\_train)  # Report the best parameters  print("Best CV params", searcher.best\_params\_)  # Find the number of nonzero coefficients (selected features)  best\_lr = searcher.best\_estimator\_  coefs = best\_lr.coef\_  print("Total number of features:", coefs.size)  print("Number of selected features:", np.count\_nonzero(coefs)) |
| One VS Rest Classifier (Default); Multinomial |  | # Fit one-vs-rest logistic regression classifier  lr\_ovr = LogisticRegression()  lr\_ovr.fit(X\_train, y\_train)  print("OVR training accuracy:", lr\_ovr.score(X\_train, y\_train))  print("OVR test accuracy :", lr\_ovr.score(X\_test, y\_test))  # Fit softmax classifier  lr\_mn = LogisticRegression(multi\_class='multinomial', solver = "lbfgs")  lr\_mn.fit(X\_train, y\_train)  print("Softmax training accuracy:", lr\_mn.score(X\_train, y\_train))  print("Softmax test accuracy :", lr\_mn.score(X\_test, y\_test))  # Print training accuracies  print("Softmax training accuracy:", lr\_mn.score(X\_train, y\_train))  print("One-vs-rest training accuracy:", lr\_ovr.score(X\_train, y\_train))  # Create the binary classifier (class 1 vs. rest)  lr\_class\_1 = LogisticRegression(C=100)  lr\_class\_1.fit(X\_train, y\_train==1)  # Plot the binary classifier (class 1 vs. rest)  plot\_classifier(X\_train, y\_train==1, lr\_class\_1)  # We'll use SVC instead of LinearSVC from now on  from sklearn.svm import SVC  # Create/plot the binary classifier (class 1 vs. rest)  svm\_class\_1 = SVC()  svm\_class\_1.fit(X\_train, y\_train==1)  plot\_classifier(X\_train, y\_train==1, svm\_class\_1) |
| 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))  #Method 3:  # Import necessary modules  from sklearn.metrics import roc\_curve  # Compute predicted probabilities: y\_pred\_prob  y\_pred\_prob = logreg.predict\_proba(X\_test)[:,1]  # Generate ROC curve values: fpr, tpr, thresholds  fpr, tpr, thresholds = roc\_curve(y\_test, y\_pred\_prob)  # Plot ROC curve  plt.plot([0, 1], [0, 1], 'k--')  plt.plot(fpr, tpr)  plt.xlabel('False Positive Rate')  plt.ylabel('True Positive Rate')  plt.title('ROC Curve')  plt.show() |
| Cross-validation |  | # Import necessary modules  from sklearn.metrics import roc\_auc\_score  from sklearn.model\_selection import cross\_val\_score  # Compute predicted probabilities: y\_pred\_prob  y\_pred\_prob = logreg.predict\_proba(X\_test)[:,1]  # Compute and print AUC score  print("AUC: {}".format(roc\_auc\_score(y\_test, y\_pred\_prob)))  # Compute cross-validated AUC scores: cv\_auc  cv\_auc = cross\_val\_score(logreg, X, y, cv=5, scoring='roc\_auc')  # Print list of AUC scores  print("AUC scores computed using 5-fold cross-validation: {}".format(cv\_auc)) |
| Grid Search |  | # Import necessary modules  from sklearn.linear\_model import LogisticRegression  from sklearn.linear\_model import ElasticNet  from sklearn.metrics import mean\_squared\_error  from sklearn.model\_selection import GridSearchCV, train\_test\_split  # Create the hyperparameter grid  c\_space = np.logspace(-5, 8, 15)  param\_grid = {'C': c\_space, 'penalty': ['l1', 'l2']}  # Instantiate a logistic regression classifier: logreg  logreg = LogisticRegression()  # Create train and test sets  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.4, random\_state=42)  # Instantiate the GridSearchCV object: logreg\_cv  logreg\_cv = GridSearchCV(logreg, param\_grid, cv=5)  # Fit it to the training data  logreg\_cv.fit(X\_train, y\_train)  # Print the tuned parameters and score  print("Tuned Logistic Regression Parameters: {}".format(logreg\_cv.best\_params\_))  print("Best score is {}".format(logreg\_cv.best\_score\_)) |
| Elastic Net |  | # Import necessary modules  from sklearn.linear\_model import ElasticNet  from sklearn.metrics import mean\_squared\_error  from sklearn.model\_selection import GridSearchCV, train\_test\_split  # Create train and test sets  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.4, random\_state=42)  # Create the hyperparameter grid  l1\_space = np.linspace(0, 1, 30)  param\_grid = {'l1\_ratio': l1\_space}  # Instantiate the ElasticNet regressor: elastic\_net  elastic\_net = ElasticNet()  # Setup the GridSearchCV object: gm\_cv  gm\_cv = GridSearchCV(elastic\_net, param\_grid, cv=5)  # Fit it to the training data  gm\_cv.fit(X\_train, y\_train)  # Predict on the test set and compute metrics  y\_pred = gm\_cv.predict(X\_test)  r2 = gm\_cv.score(X\_test, y\_test)  mse = mean\_squared\_error(y\_test, y\_pred)  print("Tuned ElasticNet l1 ratio: {}".format(gm\_cv.best\_params\_))  print("Tuned ElasticNet R squared: {}".format(r2))  print("Tuned ElasticNet MSE: {}".format(mse))  # Setup the pipeline steps: steps  steps = [('imputation', Imputer(missing\_values='NaN', strategy='mean', axis=0)),  ('scaler', StandardScaler()),  ('elasticnet', ElasticNet())]  # Create the pipeline: pipeline  pipeline = Pipeline(steps)  # Specify the hyperparameter space  parameters = {'elasticnet\_\_l1\_ratio':np.linspace(0,1,30)}  # Create train and test sets  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.4, random\_state=42)  # Create the GridSearchCV object: gm\_cv  gm\_cv = GridSearchCV(pipeline, parameters)  # Fit to the training set  gm\_cv.fit(X\_train, y\_train)  # Compute and print the metrics  r2 = gm\_cv.score(X\_test, y\_test)  print("Tuned ElasticNet Alpha: {}".format(gm\_cv.best\_params\_))  print("Tuned ElasticNet R squared: {}".format(r2)) |
| 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 |  |
| Support Vector Machine | | |
| Linear SVM |  | # Train a linear SVM  svm = SVC(kernel="linear")  svm.fit(X, y)  plot\_classifier(X, y, svm, lims=(11,15,0,6))  # Make a new data set keeping only the support vectors  print("Number of original examples", len(X))  print("Number of support vectors", len(svm.support\_))  X\_small = X[svm.support\_]  y\_small = y[svm.support\_]  # Train a new SVM using only the support vectors  svm\_small = SVC(kernel="linear")  svm\_small.fit(X\_small, y\_small)  plot\_classifier(X\_small, y\_small, svm\_small, lims=(11,15,0,6)) |
| Grid Search |  | # Instantiate an RBF SVM  svm = SVC()  # Instantiate the GridSearchCV object and run the search  parameters = {'C':[0.1, 1, 10], 'gamma':[0.00001, 0.0001, 0.001, 0.01, 0.1]}  searcher = GridSearchCV(svm, parameters)  searcher.fit(X\_train, y\_train)  # Report the best parameters and the corresponding score  print("Best CV params", searcher.best\_params\_)  print("Best CV accuracy", searcher.best\_score\_)  # Report the test accuracy using these best parameters  print("Test accuracy of best grid search hypers:", searcher.score(X\_test, y\_test)) |
| SGD Classifier |  | # We set random\_state=0 for reproducibility  linear\_classifier = SGDClassifier(random\_state=0)  # Instantiate the GridSearchCV object and run the search  parameters = {'alpha':[0.00001, 0.0001, 0.001, 0.01, 0.1, 1],  'loss':['hinge', 'log'], 'penalty':['l1', 'l2']}  searcher = GridSearchCV(linear\_classifier, parameters, cv=10)  searcher.fit(X\_train, y\_train)  # Report the best parameters and the corresponding score  print("Best CV params", searcher.best\_params\_)  print("Best CV accuracy", searcher.best\_score\_)  print("Test accuracy of best grid search hypers:", searcher.score(X\_test, y\_test)) |
| 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) |
| Random Search CV |  | # Import necessary modules  from scipy.stats import randint  from sklearn.tree import DecisionTreeClassifier  from sklearn.model\_selection import RandomizedSearchCV  # Setup the parameters and distributions to sample from: param\_dist  param\_dist = {"max\_depth": [3, None],  "max\_features": randint(1, 9),  "min\_samples\_leaf": randint(1, 9),  "criterion": ["gini", "entropy"]}  # Instantiate a Decision Tree classifier: tree  tree = DecisionTreeClassifier()  # Instantiate the RandomizedSearchCV object: tree\_cv  tree\_cv = RandomizedSearchCV(tree, param\_dist, cv=5)  # Fit it to the data  tree\_cv.fit(X,y)  # Print the tuned parameters and score  print("Tuned Decision Tree Parameters: {}".format(tree\_cv.best\_params\_))  print("Best score is {}".format(tree\_cv.best\_score\_)) |
| 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 |
| Loss Function | | |
| Log Loss |  | def compute\_log\_loss(predicted, actual, eps=1e-14):  """ Computes the logarithmic loss between predicted and actual when these are 1D arrays. :param predicted: The predicted probabilities as floats between 0-1 :param actual: The actual binary labels. Either 0 or 1. :param eps (optional): log(0) is inf, so we need to offset our predicted values slightly by eps from 0 or 1. """  predicted = np.clip(predicted, eps, 1 - eps)  loss = -1 \* np.mean(actual \* np.log(predicted) + (1 - actual) \* np.log(1 - predicted))    return loss  compute\_log\_loss(predicted\_values, actual\_labels) |
| Hold Out | | |
| Hold out |  | # Instantiate the classifier: clf  clf = OneVsRestClassifier(LogisticRegression())  # Fit it to the training data  clf.fit(X\_train, y\_train)  # Load the holdout data: holdout  holdout = pd.read\_csv('HoldoutData.csv', index\_col=0)  # Generate predictions: predictions  predictions = clf.predict\_proba(holdout[NUMERIC\_COLUMNS].fillna(-1000) )  # Generate predictions: predictions  predictions = clf.predict\_proba(holdout[NUMERIC\_COLUMNS].fillna(-1000))  # Format predictions in DataFrame: prediction\_df  prediction\_df = pd.DataFrame(columns=pd.get\_dummies(df[LABELS]).columns, index=holdout.index, data=predictions)  # Save prediction\_df to csv  prediction\_df.to\_csv('predictions.csv')  # Submit the predictions for scoring: score  score = score\_submission(pred\_path='predictions.csv')  # Print score  print('Your model, trained with numeric data only, yields logloss score: {}'.format(score)) |
| Text Analytics | | |
| Web-scrapping, Stopword removal, Text processing, Word Frequency |  | # Importing requests, BeautifulSoup and nltk  import requests  import nltk  from bs4 import BeautifulSoup  # Getting the Moby Dick HTML  r = requests.get('url')  # Setting the correct text encoding of the HTML page  r.encoding = 'utf-8'  # Extracting the HTML from the request object  html = r.text  # Printing the first 2000 characters in html  print(html[0:2000])  # Creating a BeautifulSoup object from the HTML  soup = BeautifulSoup(html)  # Getting the text out of the soup  text = soup.text  # Printing out text between characters 32000 and 34000  print(text[32000:34000])  # Importing library  import nltk  # Creating a tokenizer  tokenizer = nltk.tokenize.RegexpTokenizer('\w+')  # Tokenizing the text  tokens = tokenizer.tokenize(text)  # Printing out the first 8 words / tokens  print(tokens[:8])  # A new list to hold the lowercased words  words = []  # Looping through the tokens and make them lower case  for word in tokens:  words.append(word.lower())  # Printing out the first 8 words / tokens  print(words[0:8])  # Importing library  nltk.download('stopwords')  from nltk.corpus import stopwords  # Getting the English stop words from nltk  sw = stopwords.words('english')  # Printing out the first eight stop words  print(sw[0:8])  # A new list to hold Moby Dick with No Stop words  words\_ns = []  # Appending to words\_ns all words that are in words but not in sw  for word in words:  if (word not in sw):  words\_ns.append(word)  else:  words\_ns  # Printing the first 5 words\_ns to check that stop words are gone  words\_ns[:5]  # This command display figures inline  %matplotlib inline  # Creating the word frequency distribution  freqdist = nltk.FreqDist(words\_ns)  # Plotting the word frequency distribution  freqdist.plot(25) |
|  |  | # Import modules  import numpy as np  import pandas as pd  import nltk  # Set seed for reproducibility  np.random.seed(5)  # Word Tokenize first sentence from sent\_tokenized, save as words\_tokenized  words\_tokenized = [word for word in nltk.word\_tokenize(sent\_tokenized[0])]  # Remove tokens that do not contain any letters from words\_tokenized  import re  filtered = [word for word in words\_tokenized if re.search('\w', word)]  # Display filtered words to observe words after tokenization  filtered  # Import the SnowballStemmer to perform stemming  from nltk.stem.snowball import SnowballStemmer  # Create an English language SnowballStemmer object  stemmer = SnowballStemmer("english")  # Print filtered to observe words without stemming  print("Without stemming: ", filtered)  # Stem the words from filtered and store in stemmed\_words  stemmed\_words = [stemmer.stem(word) for word in filtered]  # Print the stemmed\_words to observe words after stemming  print("After stemming: ", stemmed\_words)  # Define a function to perform both stemming and tokenization  def tokenize\_and\_stem(text):    # Tokenize by sentence, then by word  tokens = [word for sent in nltk.sent\_tokenize(text) for word in nltk.word\_tokenize(sent)]    # Filter out raw tokens to remove noise  filtered\_tokens = [token for token in tokens if re.search('[a-zA-Z]', token)]    # Stem the filtered\_tokens  stems = [stemmer.stem(word) for word in filtered\_tokens]    return stems  words\_stemmed = tokenize\_and\_stem("Today (May 19, 2016) is his only daughter's wedding. \ n Tomorrow is May 20, 2016")  print(words\_stemmed)  # Import TfidfVectorizer to create TF-IDF vectors  from sklearn.feature\_extraction.text import TfidfVectorizer  # Instantiate TfidfVectorizer object with stopwords and tokenizer  # parameters for efficient processing of text  tfidf\_vectorizer = TfidfVectorizer(max\_df=0.8, max\_features=200000,  min\_df=0.2, stop\_words='english',  use\_idf=True, tokenizer=tokenize\_and\_stem,  ngram\_range=(1,3))  # Fit and transform the tfidf\_vectorizer with the "plot" of each movie  # to create a vector representation of the plot summaries  tfidf\_matrix = tfidf\_vectorizer.fit\_transform([x for x in movies\_df["plot"]])  print(tfidf\_matrix.shape)  # Import k-means to perform clusters  from sklearn.cluster import KMeans  # Create a KMeans object with 5 clusters and save as km  km = KMeans(n\_clusters=5)  # Fit the k-means object with tfidf\_matrix  km.fit(tfidf\_matrix)  clusters = km.labels\_.tolist()  # Create a column cluster to denote the generated cluster for each movie  movies\_df["cluster"] = clusters  # Display number of films per cluster (clusters from 0 to 4)  movies\_df['cluster'].value\_counts()  # Import cosine\_similarity to calculate similarity of movie plots  from sklearn.metrics.pairwise import cosine\_similarity  # Calculate the similarity distance  similarity\_distance = 1 - cosine\_similarity(tfidf\_matrix)  # Import matplotlib.pyplot for plotting graphs  import matplotlib.pyplot as plt  # Configure matplotlib to display the output inline  %matplotlib inline  # Import modules necessary to plot dendrogram  from scipy.cluster.hierarchy import linkage, dendrogram  # Create mergings matrix  mergings = linkage(similarity\_distance, method='complete')  # Plot the dendrogram, using title as label column  dendrogram\_ = dendrogram(mergings,  labels=[x for x in movies\_df["title"]],  leaf\_rotation=90,  leaf\_font\_size=16,  )  # Adjust the plot  fig = plt.gcf()  \_ = [lbl.set\_color('r') for lbl in plt.gca().get\_xmajorticklabels()]  fig.set\_size\_inches(108, 21)  # Show the plotted dendrogram  plt.show() |
| Text removal |  | # Load the regular expression library  import re  # Print the titles of the first rows  print(papers['title'].head())  # Remove punctuation  papers['title\_processed'] = papers['title'].map(lambda x: re.sub('[,\.!?]', '', x))  # Convert the titles to lowercase  papers['title\_processed'] = papers['title\_processed'].str.lower()  # Print the processed titles of the first rows  papers['title\_processed'][1]  # Import the wordcloud library  import wordcloud  # Join the different processed titles together.  long\_string = ''.join(papers['title\_processed'])  # Create a WordCloud object  wordcloud = wordcloud.WordCloud()  # Generate a word cloud  wordcloud.generate(long\_string)  # Visualize the word cloud  wordcloud.to\_image() |
| Word Cloud |  | # Import the wordcloud library  import wordcloud  # Join the different processed titles together.  long\_string = ' '.join(papers['title\_processed'])  # Create a WordCloud object  wordcloud = wordcloud.WordCloud()  # Generate a word cloud  wordcloud.generate(long\_string)  # Visualize the word cloud  wordcloud.to\_image() |
| 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['#col']  # Create the basic token pattern  TOKENS\_BASIC = '\\S+(?=\\s+)'  # Create the token pattern: TOKENS\_ALPHANUMERIC  TOKENS\_ALPHANUMERIC = '[A-Za-z0-9]+(?=\\s+)'  # 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, token\_pattern=TOKENS\_ALPHANUMERIC)  # Print the number of tokens and first 15 tokens  msg = "There are {} tokens in Position\_Extra if we split on non-alpha numeric"  print(msg.format(len(vec\_alphanumeric.get\_feature\_names())))  print(vec\_alphanumeric.get\_feature\_names()[:15])  # 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) |
| Join all the items in a row |  | # Define combine\_text\_columns()  def combine\_text\_columns(data\_frame, to\_drop=NUMERIC\_COLUMNS + LABELS):  """ converts all text in each row of data\_frame to single vector """    # Drop non-text columns that are in the df  to\_drop = set(to\_drop) & set(data\_frame.columns.tolist())  text\_data = data\_frame.drop(to\_drop, axis=1)    # Replace nans with blanks  text\_data.fillna('', inplace=True)    # Join all text items in a row that have a space in between  return text\_data.apply(lambda x: " ".join(x), axis=1)  # Import the CountVectorizer  from sklearn.feature\_extraction.text import CountVectorizer  # Create the basic token pattern  TOKENS\_BASIC = '\\S+(?=\\s+)'  # Create the alphanumeric token pattern  TOKENS\_ALPHANUMERIC = '[A-Za-z0-9]+(?=\\s+)'  # Instantiate basic CountVectorizer: vec\_basic  vec\_basic = CountVectorizer(token\_pattern=TOKENS\_BASIC, ngram\_range=(1,2))  # Instantiate alphanumeric CountVectorizer: vec\_alphanumeric  vec\_alphanumeric = CountVectorizer(token\_pattern=TOKENS\_ALPHANUMERIC, ngram\_range=(1,2))  # Create the text vector  text\_vector = combine\_text\_columns(df)  # Fit and transform vec\_basic  vec\_basic.fit\_transform(text\_vector)  # Print number of tokens of vec\_basic  print("There are {} tokens in the dataset".format(len(vec\_basic.get\_feature\_names())))  # Fit and transform vec\_alphanumeric  vec\_alphanumeric.fit\_transform(text\_vector)  # Print number of tokens of vec\_alphanumeric  print("There are {} alpha-numeric tokens in the dataset".format(len(vec\_alphanumeric.get\_feature\_names()))) |
| Pipeline |  | #Method 1:  # Import Pipeline  from sklearn.pipeline import Pipeline  # Import other necessary modules  from sklearn.model\_selection import train\_test\_split  from sklearn.linear\_model import LogisticRegression  from sklearn.multiclass import OneVsRestClassifier  from sklearn.preprocessing import Imputer  # Split and select numeric data only, no nans  X\_train, X\_test, y\_train, y\_test = train\_test\_split(sample\_df[['numeric', 'with\_missing']], pd.get\_dummies(sample\_df['label']), random\_state=456)  # Instantiate Pipeline object: pl  pl = Pipeline([  ('imp', Imputer()),  ('vec', CountVectorizer()),  ('clf', OneVsRestClassifier(LogisticRegression()))  ])  # Fit the pipeline to the training data  pl.fit(X\_train, y\_train)  # Compute and print accuracy  accuracy = pl.score(X\_test, y\_test)  print("\nAccuracy on sample data - all numeric, incl nans: ", accuracy)  # Import FunctionTransformer  from sklearn.preprocessing import FunctionTransformer  # Obtain the text data: get\_text\_data  get\_text\_data = FunctionTransformer(lambda x: x['text'], validate=False)  # Obtain the numeric data: get\_numeric\_data  get\_numeric\_data = FunctionTransformer(lambda x: x[['numeric', 'with\_missing']], validate=False)  # Fit and transform the text data: just\_text\_data  just\_text\_data = get\_text\_data.fit\_transform(sample\_df)  # Fit and transform the numeric data: just\_numeric\_data  just\_numeric\_data = get\_numeric\_data.fit\_transform(sample\_df)  # Print head to check results  print('Text Data')  print(just\_text\_data.head())  print('\nNumeric Data')  print(just\_numeric\_data.head())  #Method 3:  # Import random forest classifer  from sklearn.ensemble import RandomForestClassifier  # Import FeatureUnion  from sklearn.pipeline import FeatureUnion  # Split using ALL data in sample\_df  X\_train, X\_test, y\_train, y\_test = train\_test\_split(sample\_df[['numeric', 'with\_missing', 'text']], pd.get\_dummies(sample\_df['label']), random\_state=22)  # Create a FeatureUnion with nested pipeline: process\_and\_join\_features  process\_and\_join\_features = FeatureUnion(  transformer\_list = [  ('numeric\_features', Pipeline([  ('selector', get\_numeric\_data),  ('imputer', Imputer())  ])),  ('text\_features', Pipeline([  ('selector', get\_text\_data),  ('vectorizer', CountVectorizer())  ])),  ('clf', RandomForestClassifier(n\_estimators=15))  ]  )  # Instantiate nested pipeline: pl  pl = Pipeline([  ('union', process\_and\_join\_features),  ('clf', OneVsRestClassifier(LogisticRegression()))  ])  # Fit pl to the training data  pl.fit(X\_train, y\_train)  # Compute and print accuracy  accuracy = pl.score(X\_test, y\_test)  print("\nAccuracy on sample data - all data: ", accuracy) |
| Hashing |  | # Import HashingVectorizer  from sklearn.feature\_extraction.text import HashingVectorizer  # Get text data: text\_data  text\_data = combine\_text\_columns(X\_train)  # Create the token pattern: TOKENS\_ALPHANUMERIC  TOKENS\_ALPHANUMERIC = '[A-Za-z0-9]+(?=\\s+)'  # Instantiate the HashingVectorizer: hashing\_vec  hashing\_vec = HashingVectorizer(token\_pattern=TOKENS\_ALPHANUMERIC)  # Fit and transform the Hashing Vectorizer  hashed\_text = hashing\_vec.fit\_transform(text\_data)  # Create DataFrame and print the head  hashed\_df = pd.DataFrame(hashed\_text.data)  print(hashed\_df.head()) |
| K Best Features, N-gram, Interaction terms, Hashing |  | # Import pipeline  from sklearn.pipeline import Pipeline  # Import classifiers  from sklearn.linear\_model import LogisticRegression  from sklearn.multiclass import OneVsRestClassifier  # Import CountVectorizer  from sklearn.feature\_extraction.text import CountVectorizer  # Import other preprocessing modules  from sklearn.preprocessing import Imputer  from sklearn.feature\_selection import chi2, SelectKBest  # Import SparseInteractions  import https://github.com/drivendataorg/box-plots-sklearn/blob/master/src/features/SparseInteractions.py  # Import the hashing vectorizer  from sklearn.feature\_extraction.text import HashingVectorizer  # Select 300 best features  chi\_k = 300  # Import functional utilities  from sklearn.preprocessing import FunctionTransformer, MaxAbsScaler  from sklearn.pipeline import FeatureUnion  # Perform preprocessing  get\_text\_data = FunctionTransformer(combine\_text\_columns, validate=False)  get\_numeric\_data = FunctionTransformer(lambda x: x[NUMERIC\_COLUMNS], validate=False)  # Create the token pattern: TOKENS\_ALPHANUMERIC  TOKENS\_ALPHANUMERIC = '[A-Za-z0-9]+(?=\\s+)'  # Instantiate the winning model pipeline: pl  pl = Pipeline([  ('union', FeatureUnion(  transformer\_list = [  ('numeric\_features', Pipeline([  ('selector', get\_numeric\_data),  ('imputer', Imputer())  ])),  ('text\_features', Pipeline([  ('selector', get\_text\_data),  ('vectorizer', HashingVectorizer(token\_pattern=TOKENS\_ALPHANUMERIC,  non\_negative=True, norm=None, binary=False,  ngram\_range=(1,2))),  ('dim\_red', SelectKBest(chi2, chi\_k))  ]))  ]  )),  ('int', SparseInteractions(degree=2)),  ('scale', MaxAbsScaler()),  ('clf', OneVsRestClassifier(LogisticRegression()))  ]) |
| Sentiment Analysis |  | # Get the indices of the sorted cofficients  inds\_ascending = np.argsort(lr.coef\_.flatten())  inds\_descending = inds\_ascending[::-1]  # Print the most positive words  print("Most positive words: ", end="")  for i in range(5):  print(vocab[inds\_descending[i]], end=", ")  print("\n")  # Print most negative words  print("Most negative words: ", end="")  for i in range(5):  print(vocab[inds\_ascending[i]]], end=", ")  print("\n") |
| LDA |  | # Load the library with the CountVectorizer method  from sklearn.feature\_extraction.text import CountVectorizer  import numpy as np  # Helper function  def plot\_10\_most\_common\_words(count\_data, count\_vectorizer):  import matplotlib.pyplot as plt  words = count\_vectorizer.get\_feature\_names()  total\_counts = np.zeros(len(words))  for t in count\_data:  total\_counts+=t.toarray()[0]    count\_dict = (zip(words, total\_counts))  count\_dict = sorted(count\_dict, key=lambda x:x[1], reverse=True)[0:10]  words = [w[0] for w in count\_dict]  counts = [w[1] for w in count\_dict]  x\_pos = np.arange(len(words))  plt.bar(x\_pos, counts,align='center')  plt.xticks(x\_pos, words, rotation=90)  plt.xlabel('words')  plt.ylabel('counts')  plt.title('10 most common words')  plt.show()  # Initialise the count vectorizer with the English stop words  count\_vectorizer = CountVectorizer(stop\_words='english')  # Fit and transform the processed titles  count\_data = count\_vectorizer.fit\_transform(papers['title\_processed'])  # Visualise the 10 most common words  plot\_10\_most\_common\_words(count\_data, count\_vectorizer)  import warnings  warnings.simplefilter("ignore", DeprecationWarning)  # Load the LDA model from sk-learn  from sklearn.decomposition import LatentDirichletAllocation as LDA    # Helper function  def print\_topics(model, count\_vectorizer, n\_top\_words):  words = count\_vectorizer.get\_feature\_names()  for topic\_idx, topic in enumerate(model.components\_):  print("\nTopic #%d:" % topic\_idx)  print(" ".join([words[i]  for i in topic.argsort()[:-n\_top\_words - 1:-1]]))    # Tweak the two parameters below (use int values below 15)  number\_topics = 8  number\_words = 10  # Create and fit the LDA model  lda = LDA(n\_components=number\_topics)  lda.fit(count\_data)  # Print the topics found by the LDA model  print("Topics found via LDA:")  print\_topics(lda, count\_vectorizer, number\_words) |
| Similarity matrix |  | # Load the library allowing similarity computations  from gensim import similarities  # Compute the similarity matrix (pairwise distance between all texts)  sims = similarities.MatrixSimilarity(model[bows])  # Transform the resulting list into a dataframe  sim\_df = pd.DataFrame(list(sims)) |
| 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') |
| Heatmap |  | # Importing modules  import matplotlib.pyplot as plt  import pandas as pd  # Create a DataFrame and visualize the data in a pretty and insightful format  df = pd.DataFrame(retweets, columns=['Retweets','Favorites', 'Followers', 'ScreenName', 'Text'])  df.groupby(['ScreenName','Text','Followers']).sum().sort\_values(by=['Followers'], ascending=False)  df.style.background\_gradient() |
| Seaborn Visualization | | |
| Correlation plot / Heat map |  | # Import seaborn library  import seaborn as sns  plt.style.use('ggplot')  #Correlation plot  pd.scatter\_matrix(#df, c = #Y variable for target, figsize = [#8, #8], s=#150, marker = '#D')  #Heat map  sns.heatmap(#df.corr(), square=#True, cmap='RdYlGn') |
| Line / Scatter plot |  | # Plot the data  sns.relplot(data = #df, x = '#col', y = '#col', kind = 'line / scatter', hue='#col') |
| Bar plot |  | # Plot the data  sns.barplot(data=#df, x='#col', y='#col', estimator=sum, ci=None) |
| Scatter plot, KDE plot |  | # Create a figure with two scatter plots of launch speed vs. launch angle, one for each player's home runs  fig1, axs1 = plt.subplots(ncols=2, sharex=True, sharey=True)  sns.regplot(x='launch\_speed', y='launch\_angle', fit\_reg=False, color='tab:blue', data=judge\_hr, ax=axs1[0]).set\_title('Aaron Judge\nHome Runs, 2015-2017')  sns.regplot(x='launch\_speed', y='launch\_angle', fit\_reg=False, color='tab:blue', data=stanton\_hr, ax=axs1[1]).set\_title('Giancarlo Stanton\nHome Runs, 2015-2017')  # Create a figure with two KDE plots of launch speed vs. launch angle, one for each player's home runs  fig2, axs2 = plt.subplots(ncols=2, sharex=True, sharey=True)  sns.kdeplot(judge\_hr['launch\_speed'], judge\_hr['launch\_angle'], cmap="Blues", shade=True, shade\_lowest=False, ax=axs2[0]).set\_title('Aaron Judge\nHome Runs, 2015-2017')  sns.kdeplot(stanton\_hr['launch\_speed'], stanton\_hr['launch\_angle'], cmap="Blues", shade=True, shade\_lowest=False, ax=axs2[1]).set\_title('Giancarlo Stanton\nHome Runs, 2015-2017') |
| Count plot |  | plt.figure()  sns.countplot(x='#col', hue='#col', data=#df, palette='#RdBu')  plt.xticks([0,1], ['#No', '#Yes'])  plt.show() |
| Bokeh visualization | | |
| Line graph |  | 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) |
| Scatter plot |  | from bokeh.io import show, output\_notebook, push\_notebook  from bokeh.plotting import figure  from bokeh.models import ColumnDataSource, HoverTool  output\_notebook()  # Make a source and a scatter plot  source = ColumnDataSource(moisturizers\_dry)  plot = figure(x\_axis\_label = 'T-SNE 1',  y\_axis\_label = 'T-SNE 2',  width = 500, height = 400)  plot.circle(x = 'X',  y = 'Y',  source = source,  size = 10, color = '#FF7373', alpha = .8)  # Create a HoverTool object  hover = HoverTool(tooltips = [('Item', '@Name'),  ('Brand', '@Brand'),  ('Price', '$@Price'),  ('Rank', '@Rank')])  plot.add\_tools(hover)  # Plot the map  show(plot) |
| Plotly Visualization | | |
| Bar chart |  | import plotly  plotly.offline.init\_notebook\_mode(connected=True)  import plotly.graph\_objs as go  # Print the total number of unique categories  num\_categories = len(apps['Category'].unique())  print('Number of categories = ', num\_categories)  # Count the number of apps in each 'Category' and sort them in descending order  num\_apps\_in\_category = apps['Category'].value\_counts().sort\_values(ascending = False)  data = [go.Bar(  x = num\_apps\_in\_category.index, # index = category name  y = num\_apps\_in\_category.values, # value = count  )]  plotly.offline.iplot(data) |
| Histogram |  | # Average rating of apps  avg\_app\_rating = apps['Rating'].mean()  print('Average app rating = ', avg\_app\_rating)  # Distribution of apps according to their ratings  data = [go.Histogram(  x = apps['Rating']  )]  # Vertical dashed line to indicate the average app rating  layout = {'shapes': [{  'type' :'line',  'x0': avg\_app\_rating,  'y0': 0,  'x1': avg\_app\_rating,  'y1': 1000,  'line': { 'dash': 'dashdot'}  }]  }  plotly.offline.iplot({'data': data, 'layout': layout}) |
| Box Plot |  | trace0 = go.Box(  # Data for paid apps  y=apps['Installs'][apps['Type'] == 'Paid'],  name = 'Paid'  )  trace1 = go.Box(  # Data for free apps  y=apps['Installs'][apps['Type'] == 'Free'],  name = 'Free'  )  layout = go.Layout(  title = "Number of downloads of paid apps vs. free apps",  yaxis = dict(  type = 'log',  autorange = True  )  )  # Add trace0 and trace1 to a list for plotting  data = [trace0, trace1]  plotly.offline.iplot({'data': data, 'layout': layout}) |
| Strip plot |  | import matplotlib.pyplot as plt  fig, ax = plt.subplots()  fig.set\_size\_inches(15, 8)  # Select a few popular app categories  popular\_app\_cats = apps[apps.Category.isin(['GAME', 'FAMILY', 'PHOTOGRAPHY',  'MEDICAL', 'TOOLS', 'FINANCE',  'LIFESTYLE','BUSINESS'])]  # Examine the price trend by plotting Price vs Category  ax = sns.stripplot(x = popular\_app\_cats['Price'], y = popular\_app\_cats['Category'], jitter=True, linewidth=1)  ax.set\_title('App pricing trend across categories')  # Apps whose Price is greater than 200  apps\_above\_200 = popular\_app\_cats[['Category', 'App', 'Price']][popular\_app\_cats['Price'] > 200]  apps\_above\_200 |
| Joint Plot |  | %matplotlib inline  import seaborn as sns  sns.set\_style("darkgrid")  import warnings  warnings.filterwarnings("ignore")  # Subset for categories with at least 250 apps  large\_categories = apps.groupby(apps['Category']).filter(lambda x: len(x) >= 250).reset\_index()  # Plot size vs. rating  plt1 = sns.jointplot(x = large\_categories['Rating'], y = large\_categories['Size'], kind = 'hex')  # Subset out apps whose type is 'Paid'  paid\_apps = apps[apps['Type'] == 'Paid']  # Plot price vs. rating  plt2 = sns.jointplot(x = paid\_apps['Price'], y = paid\_apps['Rating']) |
| Network Analysis | | |
| Node properties |  | # Nodes of graph  #T.nodes()  # Edge of graph  #T.edges(data=True)  list(#T.edges(data=True))  #Size of graph  len(#T) |
| Draw network graph |  | # Import necessary modules  import matplotlib.pyplot as plt  import networkx as nx  # Set the weight of the edge  #T.edges[1, 10]['weight'] = 2  # Iterate over all the edges (with metadata)  for #u, #v, #d in #T.edges(data=True):  # Check if node 293 is involved  if 293 in [#u,#v]:  # Set the weight to 1.1  #T.edges[#u, #v]['weight'] = 1.1  # Draw the graph to screen  #nx.draw(#T)  plt.show() |
| Find self-loops |  | # Define find\_selfloop\_nodes()  def find\_selfloop\_nodes(G):  """  Finds all nodes that have self-loops in the graph G.  """  nodes\_in\_selfloops = []  # Iterate over all the edges of G  for u, v in G.edges():  # Check if node u and node v are the same  if u == v:  # Append node u to nodes\_in\_selfloops  nodes\_in\_selfloops.append(u)  return nodes\_in\_selfloops  # Check whether number of self loops equals the number of nodes in self loops  assert T.number\_of\_selfloops() == len(find\_selfloop\_nodes(T)) |
| Matrix plots |  | # Import nxviz  import nxviz as nv  # Create the MatrixPlot object: m  m = nv.MatrixPlot(T)  # Draw m to the screen  m.draw()  # Display the plot  plt.show()  # Convert T to a matrix format: A  A = nx.to\_numpy\_matrix(T)  # Convert A back to the NetworkX form as a directed graph: T\_conv  T\_conv = nx.from\_numpy\_matrix(A, create\_using=nx.DiGraph())  # Check that the `category` metadata field is lost from each node  for n, d in T\_conv.nodes(data=True):  assert 'category' not in d.keys()  # Import necessary modules  from nxviz import MatrixPlot  import matplotlib.pyplot as plt  # Calculate the largest connected component subgraph: largest\_ccs  largest\_ccs = sorted(nx.connected\_component\_subgraphs(G), key=lambda x: len(x))[-1]  # Create the customized MatrixPlot object: h  h = MatrixPlot(largest\_ccs, 'grouping')  # Draw the MatrixPlot to the screen  h.draw()  plt.show() |
| Circos plots |  | # Import necessary modules  import matplotlib.pyplot as plt  from nxviz import CircosPlot  # Create the CircosPlot object: c  c = CircosPlot(T)  # Draw c to the screen  c.draw()  # Display the plot  plt.show()  # Import necessary modules  from nxviz import CircosPlot  import matplotlib.pyplot as plt  # Iterate over all the nodes, including the metadata  for n, d in G.nodes(data=True):  # Calculate the degree of each node: G.node[n]['degree']  G.node[n]['degree'] = nx.degree(G, n)  # Create the CircosPlot object: c  c = CircosPlot(G, node\_order='degree', node\_grouping='grouping', node\_color='grouping')  # Draw the CircosPlot object to the screen  c.draw()  plt.show() |
| Arc Plot |  | # Import necessary modules  import matplotlib.pyplot as plt  from nxviz import ArcPlot  # Create the un-customized ArcPlot object: a  a = ArcPlot(T)  # Draw a to the screen  a.draw()  # Display the plot  plt.show()  # Create the customized ArcPlot object: a2  a2 = ArcPlot(T, node\_order='category', node\_color='category')  # Draw a2 to the screen  a2.draw()  # Display the plot  plt.show()  # Import necessary modules  from nxviz.plots import ArcPlot  import matplotlib.pyplot as plt  # Iterate over all the nodes in G, including the metadata  for n, d in G.nodes(data=True):  # Calculate the degree of each node: G.node[n]['degree']  G.node[n]['degree'] = nx.degree(G, n)  # Create the ArcPlot object: a  a = ArcPlot(G, node\_order='degree')  # Draw the ArcPlot to the screen  a.draw()  plt.show() |
| Compute number of neighbours for each node |  | # Define nodes\_with\_m\_nbrs()  def nodes\_with\_m\_nbrs(G, m):  """  Returns all nodes in graph G that have m neighbors.  """  nodes = set()  # Iterate over all nodes in G  for n in G.nodes():  # Check if the number of neighbors of n matches m  if len(list(G.neighbors(n))) == m:  # Add the node n to the set  nodes.add(n)  # Return the nodes with m neighbors  return nodes  # Compute and print all nodes in T that have 6 neighbors  six\_nbrs = nodes\_with\_m\_nbrs(T, 6)  print(six\_nbrs) |
| Degree, Betweenness centrality |  | # Define find\_nodes\_with\_highest\_deg\_cent()  def find\_nodes\_with\_highest\_deg\_cent(G):  # Compute the degree centrality of G: deg\_cent  deg\_cent = nx.degree\_centrality(G)  # Compute the maximum degree centrality: max\_dc  max\_dc = max(list(deg\_cent.values()))  nodes = set()  # Iterate over the degree centrality dictionary  for k, v in deg\_cent.items():  # Check if the current value has the maximum degree centrality  if v == max\_dc:  # Add the current node to the set of nodes  nodes.add(k)  return nodes  # Find the node(s) that has the highest degree centrality in T: top\_dc  top\_dc = find\_nodes\_with\_highest\_deg\_cent(T)  print(top\_dc)  # Write the assertion statement  for node in top\_dc:  assert nx.degree\_centrality(T)[node] == max(nx.degree\_centrality(T).values())  # Define find\_node\_with\_highest\_bet\_cent()  def find\_node\_with\_highest\_bet\_cent(G):  # Compute betweenness centrality: bet\_cent  bet\_cent = nx.betweenness\_centrality(G)  # Compute maximum betweenness centrality: max\_bc  max\_bc = max(list(bet\_cent.values()))  nodes = set()  # Iterate over the betweenness centrality dictionary  for k, v in bet\_cent.items():  # Check if the current value has the maximum betweenness centrality  if v == max\_bc:  # Add the current node to the set of nodes  nodes.add(k)  return nodes  # Use that function to find the node(s) that has the highest betweenness centrality in the network: top\_bc  top\_bc = find\_node\_with\_highest\_bet\_cent(T)  print(top\_bc)  # Write an assertion statement that checks that the node(s) is/are correctly identified.  for node in top\_bc:  assert nx.betweenness\_centrality(T)[node] == max(nx.betweenness\_centrality(T).values()) |
| Centrality |  | # 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) |
| Degree distribution |  | # Compute the degree of every node: degrees  degrees = [len(list(T.neighbors(n))) for n in T.nodes()]  # Print the degrees  print(degrees)  # Import matplotlib.pyplot  import matplotlib.pyplot as plt  # Compute the degree centrality of the Twitter network: deg\_cent  deg\_cent = nx.degree\_centrality(T)  # Plot a histogram of the degree centrality distribution of the graph.  plt.figure()  plt.hist(list(deg\_cent.values()))  plt.show()  # Plot a histogram of the degree distribution of the graph  plt.figure()  plt.hist(degrees)  plt.show()  # Plot a scatter plot of the centrality distribution and the degree distribution  plt.figure()  plt.scatter(x=degrees, y=list(deg\_cent.values())  plt.show() |
| Shortest Path Algorithm |  | # Define path\_exists()  def path\_exists(G, node1, node2):  """  This function checks whether a path exists between two nodes (node1, node2) in graph G.  """  visited\_nodes = set()  queue = [node1]  for node in queue:  neighbors = G.neighbors(node)  if node2 in neighbors:  print('Path exists between nodes {0} and {1}'.format(node1, node2))  return True    else:  # Add current node to visited nodes  visited\_nodes=visited\_nodes.add(node)  # Add neighbors of current node that have not yet been visited  queue.extend([n for n in neighbors if n not in visited\_nodes])  def path\_exists(G, node1, node2):  """  This function checks whether a path exists between two nodes (node1, node2) in graph G.  """  visited\_nodes = set()  queue = [node1]  for node in queue:  neighbors = G.neighbors(node)  if node2 in neighbors:  print('Path exists between nodes {0} and {1}'.format(node1, node2))  return True  break  else:  visited\_nodes.add(node)  queue.extend([n for n in neighbors if n not in visited\_nodes])  # Check to see if the final element of the queue has been reached  if node == queue[-1]:  print('Path does not exist between nodes {0} and {1}'.format(node1, node2))  # Place the appropriate return statement  return False |
| Triangles |  | from itertools import combinations  # Define is\_in\_triangle()  def is\_in\_triangle(G, n):  """  Checks whether a node `n` in graph `G` is in a triangle relationship or not.  Returns a boolean.  """  in\_triangle = False  # Iterate over all possible triangle relationship combinations  for n1, n2 in combinations(G.neighbors(n), 2):  # Check if n1 and n2 have an edge between them  if G.has\_edge(n1, n2):  # Add n1 to triangle\_nodes  triangle\_nodes.add(n1)  # Add n2 to triangle\_nodes  triangle\_nodes.add(n2)  return triangle\_nodes  # Write the assertion statement  assert len(nodes\_in\_triangle(T, 1)) == 35  from itertools import combinations  # Define node\_in\_open\_triangle()  def node\_in\_open\_triangle(G, n):  """  Checks whether pairs of neighbors of node `n` in graph `G` are in an 'open triangle' relationship with node `n`.  """  in\_open\_triangle = False  # Iterate over all possible triangle relationship combinations  for n1, n2 in combinations(G.neighbors(n), 2):  # Check if n1 and n2 do NOT have an edge between them  if not G.has\_edge(n1, n2):  in\_open\_triangle = True  break  return in\_open\_triangle  # Compute the number of open triangles in T  num\_open\_triangles = 0  # Iterate over all the nodes in T  for n in T.nodes():  # Check if the current node is in an open triangle  if node\_in\_open\_triangle(T, n):  # Increment num\_open\_triangles  num\_open\_triangles += 1  print(num\_open\_triangles) |
| Cliques |  | # Define maximal\_cliques()  def maximal\_cliques(G, size):  """  Finds all maximal cliques in graph `G` that are of size `size`.  """  mcs = []  for clique in nx.find\_cliques(G):  if len(clique) == size:  mcs.append(clique)  return mcs  # Check that there are 33 maximal cliques of size 3 in the graph T  assert len(maximal\_cliques(T, 3)) == 33 |
| Subgraphs |  | nodes\_of\_interest = [29, 38, 42]  # Define get\_nodes\_and\_nbrs()  def get\_nodes\_and\_nbrs(G, nodes\_of\_interest):  """  Returns a subgraph of the graph `G` with only the `nodes\_of\_interest` and their neighbors.  """  nodes\_to\_draw = []  # Iterate over the nodes of interest  for n in nodes\_of\_interest:  # Append the nodes of interest to nodes\_to\_draw  nodes\_to\_draw.append(n)  # Iterate over all the neighbors of node n  for nbr in G.neighbors(n):  # Append the neighbors of n to nodes\_to\_draw  nodes\_to\_draw.append(nbr)  return G.subgraph(nodes\_to\_draw)  # Extract the subgraph with the nodes of interest: T\_draw  T\_draw = get\_nodes\_and\_nbrs(T, nodes\_of\_interest)  # Draw the subgraph to the screen  nx.draw(T\_draw)  plt.show()  # Extract the nodes of interest: nodes  nodes = [n for n, d in T.nodes(data=True) if d['occupation'] == 'celebrity']  # Create the set of nodes: nodeset  nodeset = set(nodes)  # Iterate over nodes  for n in nodes:  # Compute the neighbors of n: nbrs  nbrs = T.neighbors(n)  # Compute the union of nodeset and nbrs: nodeset  nodeset = nodeset.union(nbrs)  # Compute the subgraph using nodeset: T\_sub  T\_sub = T.subgraph(nodeset)  # Draw T\_sub to the screen  nx.draw(T\_sub)  plt.show() |
| Recommender System |  | # Import necessary modules  from itertools import combinations  from collections import defaultdict  # Initialize the defaultdict: recommended  recommended = defaultdict(int)  # Iterate over all the nodes in G  for n, d in G.nodes(data=True):  # Iterate over all possible triangle relationship combinations  for n1, n2 in combinations(G.neighbors(n), 2):  # Check whether n1 and n2 do not have an edge  if not G.has\_edge(n1, n2):  # Increment recommended  recommended[(n1, n2)] += 1  # Identify the top 10 pairs of users  all\_counts = sorted(recommended.values())  top10\_pairs = [pair for pair, count in recommended.items() if count > 10]  print(top10\_pairs) |
| 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') |