**Comparison between R and Python codes**

**(To be completed)**

|  |  |  |
| --- | --- | --- |
| **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 |
| **Read file** | | |
| Read file | read.csv('#file') | with open('#file') as f:  print(f.read())  pd.read\_csv('#file',  sep = "#",  encoding = 'latin-1',  header = None,  names = ['#col1', '#col2'],  parse\_dates=['#col']) |
| Read head | head(#df, #num) | #df.head(#num) |
| Read tail | tail(#df, #num) | #df.tail(#num) |
| **Summary** | | |
| Describe | summary(#df)  glimpse(#df)  str(#df) | #df.describe() |
| Shape of dataframe |  | #df.shape()  #df.shape[0] #rows |
| Length of list |  | len(#list) |
| Count values |  | #Show proportion  #df.#col.value\_counts(normalize = True)  #df.#col.unique().dropna() |
| Count NA |  | #df.isnull().sum() |
| Number of columns |  | len(#df.columns) |
| Rename column | #df %>%  rename(#new = #old) | #df.rename(  columns = {#old': '#new'},  inplace = True  ) |
| Insert new column |  | #df.insert(#n\_cols, '#col', #col\_value) |
| DataFrame to list |  | #df.values.tolist() |
| Print |  | print("%s " % (#var))  print("{}".format(#df[0])) |
| **Data Manipulation** | | |
| Filter | #df %>%  dplyr::filter(#condition) | #col\_list = ['#col1', '#col2']  # Filter for rows containing these metrics  #df\_f = #df[#df.#colname.isin(#col\_list)]  # See the result |
| Locate value |  | #df['#col'].iloc[#row\_num] |
| Group |  | #df\_crosstab = #df.groupby(['#col1', '#col2']).mean()  # Convert the GroupBy object to a DataFrame  #new\_df = #df\_crosstab.reset\_index()  #df.groupby(pd.Grouper(key = '#col', freq = 'AS #for year')).size() |
| Sort |  | #df. sort\_values(by = '#col', ascending = False) |
| Replace string |  | #df.replace('#string', np.NaN) |
| Impute numeric value |  | #df.fillna(#df.mean(), inplace=True) |
| Impute categorical value |  | for col in #df.columns:  # Check if the column is of object type  if #df [col].dtypes == 'object':  # Impute with the most frequent value  #print(#df[col].value\_counts().index[0])  #df = #df.replace(np.NaN, #df [col].value\_counts().index[0])  # Count the number of NaNs in the dataset and print the counts to verify  #df.isnull().sum() |
|  |  | #df\_dummy = pd.get\_dummies(data=#df['#col'], drop\_first=True) |
| **Date manipulation** | | |
| Date extraction (Datetime) |  | 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) |
| 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 |
| Loop  iteritems |  | # 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) |
| **Encoding / Recoding** | | |
| Encoding (Convert non-numeric to numeric) |  | # Import LabelEncoder  import sklearn  from sklearn.preprocessing import LabelEncoder  # Instantiate LabelEncoder  le = LabelEncoder()  # Iterate over all the values of each column and extract their dtypes  for col in #df.columns:  # Compare if the dtype is object  if #df[col].dtypes == 'object':  # Use LabelEncoder to do the numeric transformation  le.fit(#df[col])  #df[col]=le.transform(#df[col]) |
| Min-Max Scaling |  | # Import MinMaxScaler  from sklearn.preprocessing import MinMaxScaler  # Instantiate MinMaxScaler and use it to rescale X\_train and X\_test  scaler = MinMaxScaler(feature\_range = (0, 1))  rescaledX\_train = scaler.fit\_transform(X\_train)  rescaledX\_test = scaler. fit\_transform(X\_test) |
| **Linear Regression** | | |
| Linear Regression |  | # Import LinearRegression  from sklearn.linear\_model import LinearRegression  # Build a linear regression model  regr = LinearRegression()  # Fit regr to the dataset  regr.fit(X = #df\_dummy, y = #df['#col'])  # Get estimated intercept and coefficient values  #col1 = regr.intercept\_  #col2 = regr.coef\_[[0]][0]  # Inspect the estimated intercept and coefficient values  print((#col1, #col2)) |
| Confidence interval |  | # Import a module  import numpy as np  # Create an array of indices to sample from  #inds = np.arange(len(#df['#col']))  # Initialize 500 replicate arrays  size = 500  #sample1 = np.empty(size)  #sample2 = np.empty(size)  # Generate replicates  for i in range(size):  # Resample the indices  #bs\_inds = np.random.choice(#inds, len(#inds))    # Get the sampled genre and sampled adjusted gross  #bs\_X = #df['#col'][bs\_inds]  #bs\_Y = #df['#col'][bs\_inds]    # Convert sampled genre to dummy variables  #bs\_dummies = pd.get\_dummies(data=#df['#col'], drop\_first=True)    # Build and fit a regression model  regr = LinearRegression().fit(#bs\_dummies, #bs\_Y)    # Compute replicates of estimated intercept and coefficient  #X0[i] = regr.intercept\_  #X1[i] = regr.coef\_[[0]][0]  # Compute 95% confidence intervals for intercept and coefficient values  #confidence\_interval\_1 = np.percentile(#X0, [2.5, 97.5])  #confidence\_interval\_2 = np.percentile(#X1, [2.5, 97.5])    # Inspect the confidence intervals  print(#confidence\_interval\_1)  print(#confidence\_interval\_2) |
| **Logistic Regression** | | |
| Train Test Split |  | # Import train\_test\_split  from sklearn.model\_selection import train\_test\_split  # Drop the features 11 and 13 and convert the DataFrame to a NumPy array  #df = #df.drop([11, 13], axis=1).values  # Segregate features and labels into separate variables  X,y = #df[:,0:12] , #df[:,13]  # Split into train and test sets  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.33, random\_state=42) |
| Logistic Regression |  | # Import LogisticRegression  from sklearn.linear\_model import LogisticRegression  # Instantiate a LogisticRegression classifier with default parameter values  logreg = LogisticRegression()  # Fit logreg to the train set  logreg.fit(rescaledX\_train, y\_train) |
| Evaluation for Logistic Regression |  | # 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) |
| 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)) |
| **Classification** | | |
|  |  | # Import train\_test\_split method  from sklearn.model\_selection import train\_test\_split  # Split transfusion DataFrame into  # X\_train, X\_test, y\_train and y\_test datasets,  # stratifying on the `target` column  X\_train, X\_test, y\_train, y\_test = train\_test\_split(  #df.drop(columns='#colname'),  #df.target,  test\_size=0.25,  random\_state=42,  stratify=#df.#colname  )  # Print out the first 2 rows of X\_train  X\_train.head() |
| **TPOT Classification** | | |
| TPOT Classification |  | # Import TPOTClassifier and roc\_auc\_score  from tpot import TPOTClassifier  from sklearn.metrics import roc\_auc\_score  # Instantiate TPOTClassifier  tpot = TPOTClassifier(  generations=5,  population\_size=20,  verbosity=2,  scoring='roc\_auc',  random\_state=42,  disable\_update\_check=True,  config\_dict='TPOT light'  )  tpot.fit(X\_train, y\_train)  # AUC score for tpot model  tpot\_auc\_score = roc\_auc\_score(y\_test, tpot.predict\_proba(X\_test)[:, 1])  print(f'\nAUC score: {tpot\_auc\_score:.4f}')  # Print best pipeline steps  print('\nBest pipeline steps:', end='\n')  for idx, (name, transform) in enumerate(tpot.fitted\_pipeline\_.steps, start=1):  # Print idx and transform  print(f'{idx}. {transform}') |
| Log Normalization |  | # Import numpy  import numpy as np  # Copy X\_train and X\_test into X\_train\_normed and X\_test\_normed  X\_train\_normed, X\_test\_normed = X\_train.copy(), X\_test.copy()  # Specify which column to normalize  col\_to\_normalize = '#col'  # Log normalization  for df\_ in [X\_train\_normed, X\_test\_normed]:  # Add log normalized column  df\_['#col'] = np.log(df\_[col\_to\_normalize])  # Drop the original column  df\_.drop(columns=col\_to\_normalize, inplace=True)  # Check the variance for X\_train\_normed  X\_train\_normed.var().round(3) |
| Logistic Regression |  | # Importing modules  from sklearn import linear\_model  # Instantiate LogisticRegression  logreg = linear\_model.LogisticRegression(  solver='liblinear',  random\_state=42  )  # Train the model  logreg.fit(X\_train\_normed, y\_train)  # AUC score for tpot model  logreg\_auc\_score = roc\_auc\_score(y\_test, logreg.predict\_proba(X\_test\_normed)[:, 1])  print(f'\nAUC score: {logreg\_auc\_score:.4f}') |
| Model Comparison |  | # Importing itemgetter  from operator import itemgetter  # Sort models based on their AUC score from highest to lowest  sorted(  [('tpot', tpot\_auc\_score), ('logreg', logreg\_auc\_score)],  key=itemgetter(1),  reverse=True  ) |
| **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 |
| **Visualization** | | |
| Dual axis |  | #ax = #df.plot(x = '#col1', y = '#col1', label = '#col')  #df.plot(x = '#col1', y = '#col1', label = '#col', ax = #ax)  #ax.set\_ylabel('#label') |
| **Seaborn Visualization** | | |
|  |  | # Import seaborn library  import seaborn as sns  # Plot the data  sns.relplot(data = #df, x = '#col', y = '#col', kind = 'line / scatter', hue='#col') |
| **Bokeh visualization** | | |
|  |  | import bokeh  from bokeh.plotting import output\_notebook, figure, show  output\_notebook(bokeh.resources.INLINE)  # Set up figure  p = figure(plot\_width=900, plot\_height=450, x\_axis\_type='datetime', tools='lasso\_select, box\_zoom, save, reset, wheel\_zoom',  toolbar\_location='above', x\_axis\_label='Date', y\_axis\_label='#col',  title='#title')  # Plot on figure  p.circle(#df['#col1'], #df['#col2'], color='black', nonselection\_fill\_alpha=0.2, nonselection\_fill\_color='grey')  p.line(#df['#col1'], #df['#col2'], color='black', alpha=1, line\_width=2, legend='#legend')  show(p) |
| **Network Analysis** | | |
| Networkx |  | # Importing modules  import networkx as nx  #Open multiple files  #g = [#list1]  #file\_list = ['#list2', '#list3']  for #file in #file\_list:  #df = pd.read\_csv(#file)  #g1 = nx.Graph()  for \_, edge in #df.iterrows():  #g1.add\_edge(edge['Source'], edge['Target'], weight=edge['weight'])  #g.append(#g1)  #g\_deg = nx.degree\_centrality(#g)  #g\_deg\_sorted = sorted(#g\_deg.items(), key=lambda x: x[1], reverse=True)  #dc = [nx.degree\_centrality(#graph) for #graph in #g]  #dc\_df = pd.DataFrame.from\_records(#dc)  #dc\_df[['#col']].plot()  # Creating a list of pagerank, betweenness centrality, degree centrality  #measures = [nx.pagerank(#graph),  nx.betweenness\_centrality(#graph, weight='weight'),  nx.degree\_centrality(#graph)]  # Creating the correlation DataFrame  #cor = pd.DataFrame.from\_records(#measures)  # Calculating the correlation  #cor.T.corr()  #p\_rank, #b\_cent, #d\_cent = #cor.idxmax(axis=1)  print(#p\_rank, #b\_cent, #d\_cent) |
| **Save file** | | |
| Save file | write.csv(#df, ‘#file’) | #df.to\_csv('#file') |