assignment_3

August 17, 2020

0.0.1 Assignment 3: Python for Analytics, Summer

- covers lectures 7-9
- due: August 17th by 5pm.
- Points will be deducted if:
    - Problems are not completed.
    - Portions of problems are not completed.
    - Third party modules are used when the question specified not to do so.
    - The problem was solved in a very inefficient manner. For instance, copying and pasting the same block of code 10 times instead of using a for loop or using a for loop when a comprehension would work.

0.0.2 Question 1 (10 points)

Run the count vectorizer on the below and use the matrix to create a Kmeans clustering model.

Print the feature matrix and the cluster assignments.

[1]:

```
import pandas as pd
import numpy as np
import os
import seaborn as sns
import matplotlib.pyplot as plt

%matplotlib inline
```

[2]:

```
corpus = [
    'This is the first document.',
    'This document is the second document.',
    'And this is the third one.',
    'Is this the first document?',
]
```

[3]:

```
# CountVectorizer will convert a collection of strings to a vector of term/
 ↪token counts

from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()
```

```
cvect_data = vectorizer.fit_transform(corpus).toarray()

cvect_data
```

```
[3]: array([[0, 1, 1, 1, 0, 0, 1, 0, 1],
            [0, 2, 0, 1, 0, 1, 1, 0, 1],
            [1, 0, 0, 1, 1, 0, 1, 1, 1],
            [0, 1, 1, 1, 0, 0, 1, 0, 1]])
```

```
[4]: cvect_df = pd.DataFrame(cvect_data, columns = vectorizer.get_feature_names())
     cvect_df
```

```
[4]:    and  document  first  is  one  second  the  third  this
     0    0         1      1   1    0       0    1      0     1
     1    0         2      0   1    0       1    1      0     1
     2    1         0      0   1    1       0    1      1     1
     3    0         1      1   1    0       0    1      0     1
```

```
[5]: from sklearn.cluster import KMeans

     corpus_kmeans = KMeans(3)
     corpus_kmeans.fit(cvect_df)
```

```
[5]: KMeans(n_clusters=3)
```

```
[6]: yhat = corpus_kmeans.predict(cvect_df)
```

```
[7]: yhat[0:4]
```

```
[7]: array([0, 2, 1, 0], dtype=int32)
```

```
[8]: from collections import Counter

     cluster_counts = Counter(yhat)
     cluster_counts
```

```
[8]: Counter({0: 2, 2: 1, 1: 1})
```

There are 3 clusters returned. The first one has two sentences, and the remaining two have 1 sentence each.

The first and fourth sentence in the corpus belong to the first cluster, the second sentence belongs to the third cluster, and the third sentence belongs to the second cluster.

0.0.3 Question 2 (10 points)

Make a UDF that takes a matrix of data and normalizes the data using either StandardScaler, MinMax or scale. Include a param called "how" that indicates which type of normalization to perform.

Use an assert to check that the “how” param is one of the 3 options (minmax, standardscaler, mean centering).

Test this using the Iris data for one of the normalization types. Print the describe of the resulting dataframe.

[9]:

```
def normalize(df, how = "standardscaler"): # default to StandardScaler if no␣
 ↪value is given to how
    """
    this UDF will take a dataframe object and
    normalize the data according to the provided input parameter "how"
    """
    assert how in ["standardscaler", "minmax", "meancentered"], "how param must␣
 ↪be one of: standardscaler, minmax, meancentered"

    from sklearn.preprocessing import StandardScaler, MinMaxScaler, scale

    if how == "standardscaler":
        scaler = StandardScaler()
        df_scaler = scaler.fit_transform(df)
        normalized_df = pd.DataFrame(df_scaler, columns = df.columns)
    elif how == "minmax":
        minmax = MinMaxScaler()
        df_minmax = minmax.fit_transform(df)
        normalized_df = pd.DataFrame(df_minmax, columns = df.columns)
    else:
        df_mean_centered = scale(df)
        normalized_df = pd.DataFrame(df_mean_centered, columns = df.columns)
    return normalized_df
```

[10]:

```
parent_path = os.path.dirname(os.getcwd())
print(parent_path)

data_path = parent_path + "/iris.csv"
print(data_path)
```

```
/Users/rohitsatishchandra/Desktop/MScA/Summer_2020/Python/Assignments
/Users/rohitsatishchandra/Desktop/MScA/Summer_2020/Python/Assignments/iris.csv
```

[11]:

```
iris = pd.read_csv(data_path)
iris_features = iris.drop(["Species", "Id"], 1)
iris_features.describe()
```

[11]:

```
       SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm
count     150.000000    150.000000     150.000000    150.000000
mean        5.843333      3.054000       3.758667      1.198667
std         0.828066      0.433594       1.764420      0.763161
min         4.300000      2.000000       1.000000      0.100000
```

```
25%          5.100000      2.800000       1.600000      0.300000
50%          5.800000      3.000000       4.350000      1.300000
75%          6.400000      3.300000       5.100000      1.800000
max          7.900000      4.400000       6.900000      2.500000
```

[12]:

```
iris_centered = normalize(iris_features, "minmax")
iris_centered.describe()
```

[12]:

```
       SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm
count     150.000000    150.000000     150.000000    150.000000
mean        0.428704      0.439167       0.467571      0.457778
std         0.230018      0.180664       0.299054      0.317984
min         0.000000      0.000000       0.000000      0.000000
25%         0.222222      0.333333       0.101695      0.083333
50%         0.416667      0.416667       0.567797      0.500000
75%         0.583333      0.541667       0.694915      0.708333
max         1.000000      1.000000       1.000000      1.000000
```

0.0.4 Question 3 (10 points)

Make a train test split on the Boston Housing dataset. Run a regression. Plot the y and yhat as a scatter plot. Record the train and test MSE and RMSE.

Print the MSE and RMSE for the train and test.

[13]:

```
# load the data
from sklearn.datasets import load_boston
boston_dataset = load_boston()
boston = pd.DataFrame(boston_dataset.data, columns=boston_dataset.feature_names)
boston["MEDV"] = boston_dataset.target # median value of owner-occupied homes,␣
 ↪in $1000's

boston.head()
```

[13]:

```
      CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX  \
0  0.00632  18.0   2.31   0.0  0.538  6.575  65.2  4.0900  1.0  296.0
1  0.02731   0.0   7.07   0.0  0.469  6.421  78.9  4.9671  2.0  242.0
2  0.02729   0.0   7.07   0.0  0.469  7.185  61.1  4.9671  2.0  242.0
3  0.03237   0.0   2.18   0.0  0.458  6.998  45.8  6.0622  3.0  222.0
4  0.06905   0.0   2.18   0.0  0.458  7.147  54.2  6.0622  3.0  222.0

   PTRATIO       B  LSTAT  MEDV
0     15.3  396.90   4.98  24.0
1     17.8  396.90   9.14  21.6
2     17.8  392.83   4.03  34.7
3     18.7  394.63   2.94  33.4
4     18.7  396.90   5.33  36.2
```

[14]:
```
boston_x = boston.loc[:, boston.columns != "MEDV"] # note: "CHAS" is a␣
 ↪categorical predictor (dummy encoded)
boston_y = boston[["MEDV"]]
```

[15]:
```
from sklearn.model_selection import train_test_split

b_x_train, b_x_test, b_y_train, b_y_test = train_test_split(boston_x,
                                                            boston_y,
                                                            test_size=0.3,
                                                            random_state=42)
```

[16]:
```
from sklearn.linear_model import LinearRegression

lr = LinearRegression(fit_intercept=True, normalize=False)
lr.fit(b_x_train, b_y_train) # fit the model on the training data
```

[16]:
```
LinearRegression()
```

[17]:
```
y_hat = lr.predict(b_x_test) # fitted values, using test set
```

[18]:
```
from sklearn.metrics import mean_squared_error, mean_absolute_error

y_test = b_y_test["MEDV"].values

mse = round(mean_squared_error(y_test, y_hat), 3)
rmse = round(np.sqrt(mse), 3)

print("The mean squared error of the model is: " + str(mse))
print("The root mean squared error of the model is: " + str(rmse))
```

```
The mean squared error of the model is: 21.517
The root mean squared error of the model is: 4.639
```

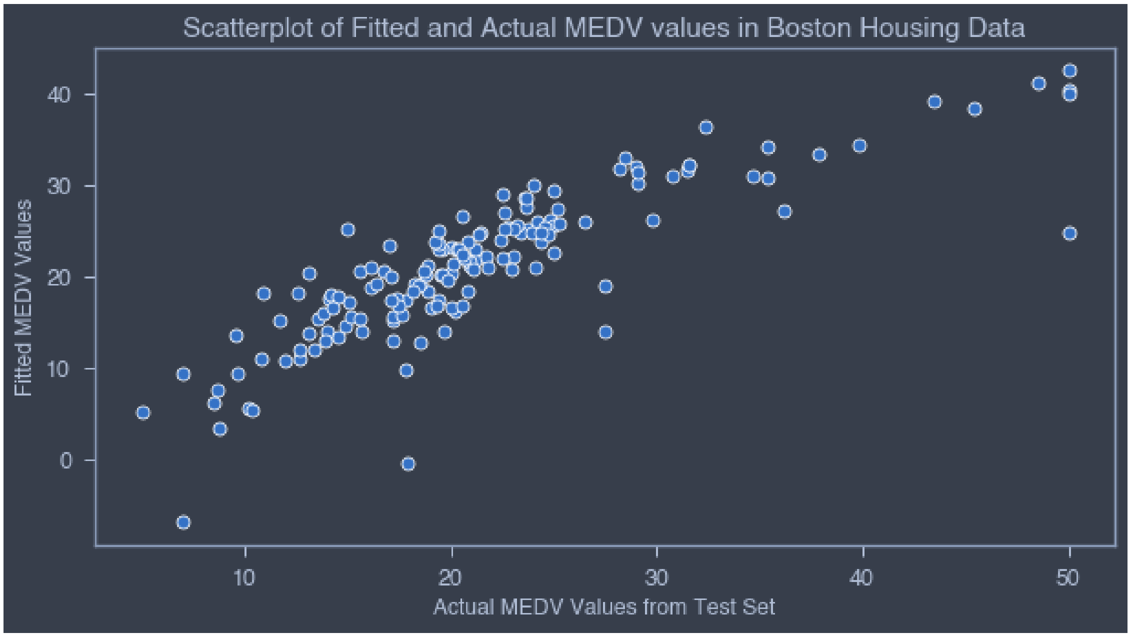
[19]:
```
boston_predictions = pd.DataFrame(y_hat, columns = ["y_fitted"])
boston_predictions["y_actual"] = y_test
boston_predictions.head()
```

[19]:
```
    y_fitted  y_actual
0  28.648960      23.6
1  36.495014      32.4
2  15.411193      13.6
3  25.403213      22.8
4  18.855280      16.1
```

[20]:
```
# scatter plot of fitted values (y_hat) and actual MEDV testset values
from jupyterthemes import jtplot
jtplot.style(theme="onedork",
             context="notebook",
             ticks=True,
             grid=False)
```

[21]:
```
plt.figure(figsize=(10,5))
ax = sns.scatterplot(x ="y_actual", y="y_fitted", data = boston_predictions)
ax.set_xlabel("Actual MEDV Values from Test Set", fontsize = 12)
ax.set_ylabel("Fitted MEDV Values", fontsize = 12)
ax.set_title("Scatterplot of Fitted and Actual MEDV values in Boston Housing␣
 ↪Data", fontsize = 15)
```

[21]:
```
Text(0.5, 1.0, 'Scatterplot of Fitted and Actual MEDV values in Boston Housing
Data')
```



0.0.5 Question 4 (15 points)

Run a Grid Search cross validation on a decision tree using the Iris dataset. Pick 3 params and have 2 values for each param. Use 3 fold cross validation. Plot the validation metrics returned from gridsearch (how the model performed) as a bar graph.

Print the validation metrics and the best params out, along with the bar graph.

[22]:
```
from sklearn.datasets import load_iris
iris_dataset = load_iris()
iris = pd.DataFrame(iris_dataset.data, columns = iris_dataset.feature_names)
iris['iris_species'] = iris_dataset.target
```

[23]:
```
iris.head()
```

[23]:
```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

   iris_species
0             0
1             0
2             0
3             0
4             0
```

[24]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier

iris_features = iris.drop(["iris_species"], 1)
iris_y = iris["iris_species"]

iris_features.head()
```

[24]:
```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2
```

[25]:
```
print(iris_y.shape)
print(iris_features.shape)
```

```
(150,)
(150, 4)
```

[26]:
```
iris_y.value_counts() # the classes are balanced
```

[26]:
```
2    50
1    50
0    50
Name: iris_species, dtype: int64
```

[27]:
```
param_grid = {
    "criterion":("gini", "entropy"),
    "max_depth":[4,6],
    "min_samples_leaf":[35, 50]
}
```

[28]:

```
from sklearn.metrics import make_scorer, accuracy_score, roc_auc_score

scoring = ["accuracy", "roc_auc_ovr_weighted"]
clf = DecisionTreeClassifier()

grid_search = GridSearchCV(clf,
                           param_grid,
                           scoring=scoring,
                           cv=3,
                           refit="accuracy",
                           return_train_score=True,
                           verbose=1,
                           n_jobs=1)

grid_search.fit(iris_features, iris_y)
```

```
Fitting 3 folds for each of 8 candidates, totalling 24 fits
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  24 out of  24 | elapsed:    0.3s finished
```

[28]:

```
GridSearchCV(cv=3, estimator=DecisionTreeClassifier(), n_jobs=1,
             param_grid={'criterion': ('gini', 'entropy'), 'max_depth': [4, 6],
                         'min_samples_leaf': [35, 50]},
             refit='accuracy', return_train_score=True,
             scoring=['accuracy', 'roc_auc_ovr_weighted'], verbose=1)
```

[29]:

```
grid_search.best_index_
```

[29]:

```
0
```

[30]:

```
# which is the best parameter combination?
grid_search.best_params_
```

[30]:

```
{'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 35}
```

[31]:

```
best_tree = grid_search.best_estimator_
iris_tree_predclass = pd.DataFrame(best_tree.predict(iris_features), columns =␣
 ↪["predicted_species"])
iris_tree_predclass.value_counts()
```

[31]:

```
predicted_species
1                    54
0                    50
2                    46
```

```
dtype: int64
```

[32]:

```
cv_results_df = pd.DataFrame.from_dict(grid_search.cv_results_)
cv_results_df.shape
```

[32]:

```
(8, 30)
```

[33]:

```
param_combos = ["Param_Combo_" + str(i) for i in range(1,9)]
print(param_combos)
cv_results_df['Param_Combo_num'] = param_combos
```

```
['Param_Combo_1', 'Param_Combo_2', 'Param_Combo_3', 'Param_Combo_4',
'Param_Combo_5', 'Param_Combo_6', 'Param_Combo_7', 'Param_Combo_8']
```

[34]:

```
cv_results_df.loc[:, ["Param_Combo_num", "param_criterion", "param_max_depth",␣
 ↪"param_min_samples_leaf", "mean_test_accuracy"]]
```

[34]:

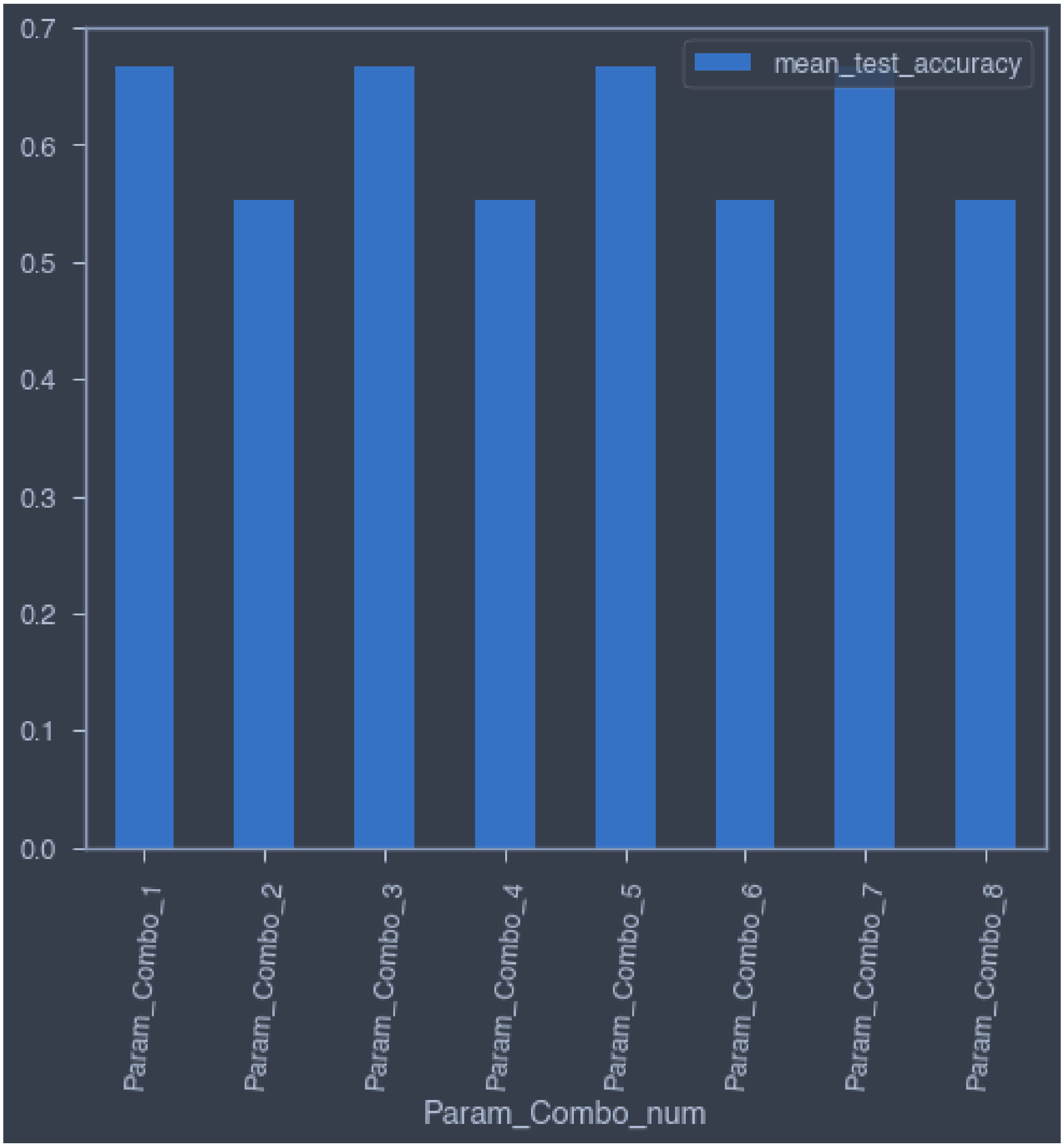
```
  Param_Combo_num param_criterion param_max_depth param_min_samples_leaf  \
0   Param_Combo_1            gini               4                     35
1   Param_Combo_2            gini               4                     50
2   Param_Combo_3            gini               6                     35
3   Param_Combo_4            gini               6                     50
4   Param_Combo_5         entropy               4                     35
5   Param_Combo_6         entropy               4                     50
6   Param_Combo_7         entropy               6                     35
7   Param_Combo_8         entropy               6                     50

   mean_test_accuracy
0            0.666667
1            0.553333
2            0.666667
3            0.553333
4            0.666667
5            0.553333
6            0.666667
7            0.553333
```

[35]:

```
cv_results_df.plot.bar(x="Param_Combo_num", y="mean_test_accuracy", rot=85)
```

[35]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff27b438810>
```



0.0.6 Question 5 (15 points)

Make a generator that parses the list of transaction data below. * Put the contents in a dictionary, where the key is the user id and the value is the list of items the user has purchased * Note the transactions are pipe (|) delimtied, meaning you'll have to use some string manipulations to get the values from each transaction, such as split("|"), which will split the string on the pipe and return a list. * Make sure to skip the header somehow and to iterate the generator, not the list.

Print the dictionary out.

[36]:
```
transactions = [
    ["user_id|item_id"],
    ["A|item_a"],
    ["B|item_a"],
    ["C|item_a"],
    ["C|item_b"],
    ["C|item_c"],
    ["B|item_c"],
    ["D|item_b"],
    ["D|item_b"]
]
```

[37]:
```
from itertools import chain

def transaction_gen(transaction_list):
    for transaction in transaction_list:
        if transaction == transaction_list[0]:
            pass
        else:
            transaction[0] = transaction[0].split("|") # will create a list␣
 ↪within each list
            transaction = list(chain.from_iterable(transaction)) # flatten
            yield transaction

trans_gen = transaction_gen(transactions)
```

[38]:
```
transactions_dict = {}

for transaction in trans_gen:
    if transaction[0] not in transactions_dict:
        transactions_dict[transaction[0]] = [transaction[1]]
    elif transaction[0] in transactions_dict:
        transactions_dict[transaction[0]].append(transaction[1])

transactions_dict
```

[38]:
```
{'A': ['item_a'],
 'B': ['item_a', 'item_c'],
 'C': ['item_a', 'item_b', 'item_c'],
 'D': ['item_b', 'item_b']}
```

0.0.7 Question 6 (20 points)

The below snippets of code will scrape the paragraph content from the below Wikipedia URL.

[39]:
```
import urllib
import requests
```

```
from bs4 import BeautifulSoup
import io
```

[40]:

```
sample_url = "https://en.wikipedia.org/wiki/Illinois"
```

[41]:

```
mybytes = urllib.request.urlopen(sample_url)
mybytes = mybytes.read().decode("utf8")

parsed_html = BeautifulSoup(mybytes, features="lxml")
```

[42]:

```
paragraph_data = [i.text for i in parsed_html.find_all("p")] # returns a list␣
 ↪of paragraphs
paragraph_data = " ".join(paragraph_data).strip()
paragraph_data[0:500]
```

[42]:

```
'Illinois (/ lə n / (listen) IL-ə-NOY) is a state in the Midwestern and Great
Lakes regions of the United States. It has the fifth largest gross domestic
product (GDP),\nthe sixth largest population, and the 25th largest land area of
all U.S. states. Illinois has been noted as a microcosm of the entire United
States.[7] With Chicago in northeastern Illinois, small industrial cities and
immense agricultural productivity in the north and center of the state, and
natural resources such as coal, tim'
```

Create a udf that returns the pargraph data from a given Wikipedia url. Map the udf to a list of 5 wiki (your choice of which wikis, though I'm sad to say Brian Craft doesn't have his own wiki) urls using a thread pool executor. Have the function return a tuple where the first element is the url and the second element is the paragraph data.

For one of the wikis, print out the first 500 characters of the paragraph data.

[43]:

```
def get_paragraph_data(url):
    mybytes = urllib.request.urlopen(url)
    mybytes = mybytes.read().decode("utf8")

    parsed_html = BeautifulSoup(mybytes, features="lxml")
    paragraph_data = [i.text for i in parsed_html.find_all("p")] # returns a␣
 ↪list of paragraphs
    paragraph_data = " ".join(paragraph_data).strip()
    return (url, paragraph_data)
```

[44]:

```
import time
from concurrent.futures import ThreadPoolExecutor

wiki_list = ["https://en.wikipedia.org/wiki/Liverpool_F.C.",
            "https://en.wikipedia.org/wiki/The_Dark_Knight_(film)",
            "https://en.wikipedia.org/wiki/Chernobyl_disaster",
            "https://en.wikipedia.org/wiki/Roberto_Firmino",
            "https://en.wikipedia.org/wiki/Roger_Federer"]
```

```
start = time.time()

with ThreadPoolExecutor(max_workers = 4) as executor:
    results_gen = executor.map(get_paragraph_data, wiki_list)
    results_list = [result for result in results_gen]

print("Runtime:{}".format(time.time() - start))
```

```
Runtime:4.381223917007446
```

[45]:
```
results_list[3][1][0:500]
```

[45]:
```
"Roberto Firmino Barbosa de Oliveira (Brazilian Portuguese:\xa0[ u b tu
fi mĩnu];[5] born 2 October 1991) is a Brazilian professional footballer who
plays as a forward, attacking midfielder or winger for Premier League club
Liverpool and the Brazil national team.\n After starting his career with
Figueirense in 2009, he spent four-and-a-half seasons at Hoffenheim. His 16
goals in 33 games for the 2013-14 Bundesliga season earned him the award for the
league's Breakthrough Player. In July 2015, he si"
```

0.0.8 Question 7 (20 points)

Find all the possible combinations of the below user list. To the resulting list of tuples, use multiprocessing to map a function that finds the distances of each combination. Put the results in a Pandas dataframe and find the 2 most similar users. Do not include comparisons against a user to themselves. Remove these from the list of tuples prior to mapping the function.

Print out the top 5 rows of the dataframe, sorting from closest to least similar.

[46]:
```
users = ["a", "b", "c", "d"]

vectors = {
    "a": [1,2,2,1],
    "b": [2,4,2,1],
    "c": [5,4,2,4],
    "d": [5,3,2,1]
}
```

[47]:
```
from itertools import combinations

user_combos = list(combinations(vectors, 2))
user_combos
```

[47]:
```
[('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'c'), ('b', 'd'), ('c', 'd')]
```

[48]:
```
def get_arrays(vector_dict):
    for key, value in vector_dict.items():
        vector_dict[key] = np.array(value)
    return vector_dict

get_arrays(vectors) # convert the existing values (lists) to numpy arrays
```

[48]:
```
{'a': array([1, 2, 2, 1]),
 'b': array([2, 4, 2, 1]),
 'c': array([5, 4, 2, 4]),
 'd': array([5, 3, 2, 1])}
```

[49]:
```
from scipy.spatial import distance

def dict_get_distance(key_tup):
    key1 = key_tup[0]
    key2 = key_tup[1]
    array1 = vectors[key1]
    array2 = vectors[key2]

    dist = distance.euclidean(array1, array2)
    return dist
```

[50]:
```
import multiprocessing as mp

start = time.time()

pool = mp.Pool(4)
pairwise_distances = pool.map(dict_get_distance, user_combos)

end = time.time()

print("Runtime:", end - start)
```

```
Runtime: 0.03563380241394043
```

[51]:
```
 # user A and user B are the most similar, since the distance is smallest

list(zip(user_combos, pairwise_distances))
```

[51]:
```
[(('a', 'b'), 2.23606797749979),
 (('a', 'c'), 5.385164807134504),
 (('a', 'd'), 4.123105625617661),
 (('b', 'c'), 4.242640687119285),
 (('b', 'd'), 3.1622776601683795),
 (('c', 'd'), 3.1622776601683795)]
```

[52]:
```
combos = ["a_b", "a_c", "a_d", "b_c", "b_d", "c_d"]
```

```
dist_df = pd.DataFrame(data=np.array(pairwise_distances),
                       index=combos,
                       columns=["euclidian_distance"])

dist_df.sort_values(by="euclidian_distance", ascending=True)
```

```
[52]:      euclidian_distance
      a_b            2.236068
      b_d            3.162278
      c_d            3.162278
      a_d            4.123106
      b_c            4.242641
      a_c            5.385165
```