2 Importing Bunch of libraries.

3 Gethering data information.

[2]:
```
from google.colab import drive
drive.mount("/content/gdrive")
```

```
Mounted at /content/gdrive
```

[ ]:
```
# Importing bunch of libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import os
import warnings
warnings.filterwarnings("ignore")
import re
from nltk.stem import PorterStemmer
from wordcloud import WordCloud , STOPWORDS
import nltk
nltk.download("stopwords")
from bs4 import BeautifulSoup
from nltk.corpus import stopwords
! pip install distance
import distance
! pip install fuzzywuzzy
from fuzzywuzzy import fuzz
from os import path
! pip install wordcloud
from wordcloud import WordCloud, STOPWORDS
from sklearn.preprocessing import MinMaxScaler
from sklearn.manifold import TSNE
import plotly.graph_objects as go
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: distance in /usr/local/lib/python3.7/dist-
packages (0.1.3)
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: fuzzywuzzy in /usr/local/lib/python3.7/dist-
packages (0.18.0)
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: wordcloud in /usr/local/lib/python3.7/dist-
```

```
packages (1.5.0)
Requirement already satisfied: numpy>=1.6.1 in /usr/local/lib/python3.7/dist-
packages (from wordcloud) (1.21.6)
Requirement already satisfied: pillow in /usr/local/lib/python3.7/dist-packages
(from wordcloud) (7.1.2)
```

1.1 Reading the data and analysing it.

[ ]:
```
df = pd.read_csv("/content/gdrive/My Drive/Quora dataset/train.csv")
df.head()
```

[ ]:
```
   id  qid1  qid2                                                question1  \
0   0     1     2  What is the step by step guide to invest in sh…
1   1     3     4  What is the story of Kohinoor (Koh-i-Noor) Dia…
2   2     5     6  How can I increase the speed of my internet co…
3   3     7     8  Why am I mentally very lonely? How can I solve…
4   4     9    10  Which one dissolve in water quikly sugar, salt…

                                                question2  is_duplicate
0  What is the step by step guide to invest in sh…             0
1  What would happen if the Indian government sto…             0
2  How can Internet speed be increased by hacking…             0
3  Find the remainder when [math]23^{24}[/math] i…             0
4            Which fish would survive in salt water?              0
```

[ ]:
```
df.describe
```

[ ]:
```
<bound method NDFrame.describe of             id    qid1    qid2  \
0            0       1       2
1            1       3       4
2            2       5       6
3            3       7       8
4            4       9      10
…          …     …     …
404285  404285  433578  379845
404286  404286   18840  155606
404287  404287  537928  537929
404288  404288  537930  537931
404289  404289  537932  537933

                                                question1  \
0       What is the step by step guide to invest in sh…
1       What is the story of Kohinoor (Koh-i-Noor) Dia…
2       How can I increase the speed of my internet co…
3       Why am I mentally very lonely? How can I solve…
4       Which one dissolve in water quikly sugar, salt…
…                                                     …
404285  How many keywords are there in the Racket prog…
```

```
404286          Do you believe there is life after death?
404287                                  What is one coin?
404288  What is the approx annual cost of living while…
404289              What is like to have sex with cousin?

                                                question2  is_duplicate
0       What is the step by step guide to invest in sh…              0
1       What would happen if the Indian government sto…              0
2       How can Internet speed be increased by hacking…              0
3       Find the remainder when [math]23^{24}[/math] i…              0
4                  Which fish would survive in salt water?             0
…                                                     …           …
404285  How many keywords are there in PERL Programmin…              0
404286         Is it true that there is life after death?             1
404287                                  What's this coin?             0
404288  I am having little hairfall problem but I want…              0
404289      What is it like to have sex with your cousin?             0

[404290 rows x 6 columns]>
```

[ ]:
```
df.describe()
```

[ ]:
```
                 id           qid1           qid2   is_duplicate
count  404290.000000  404290.000000  404290.000000  404290.000000
mean   202144.500000  217243.942418  220955.655337       0.369198
std    116708.614502  157751.700002  159903.182629       0.482588
min         0.000000       1.000000       2.000000       0.000000
25%    101072.250000   74437.500000   74727.000000       0.000000
50%    202144.500000  192182.000000  197052.000000       0.000000
75%    303216.750000  346573.500000  354692.500000       1.000000
max    404289.000000  537932.000000  537933.000000       1.000000
```

[ ]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 404290 entries, 0 to 404289
Data columns (total 6 columns):
 #   Column        Non-Null Count   Dtype
---  ------        --------------   -----
 0   id            404290 non-null  int64
 1   qid1          404290 non-null  int64
 2   qid2          404290 non-null  int64
 3   question1     404289 non-null  object
 4   question2     404288 non-null  object
 5   is_duplicate  404290 non-null  int64
dtypes: int64(4), object(2)
memory usage: 18.5+ MB
```

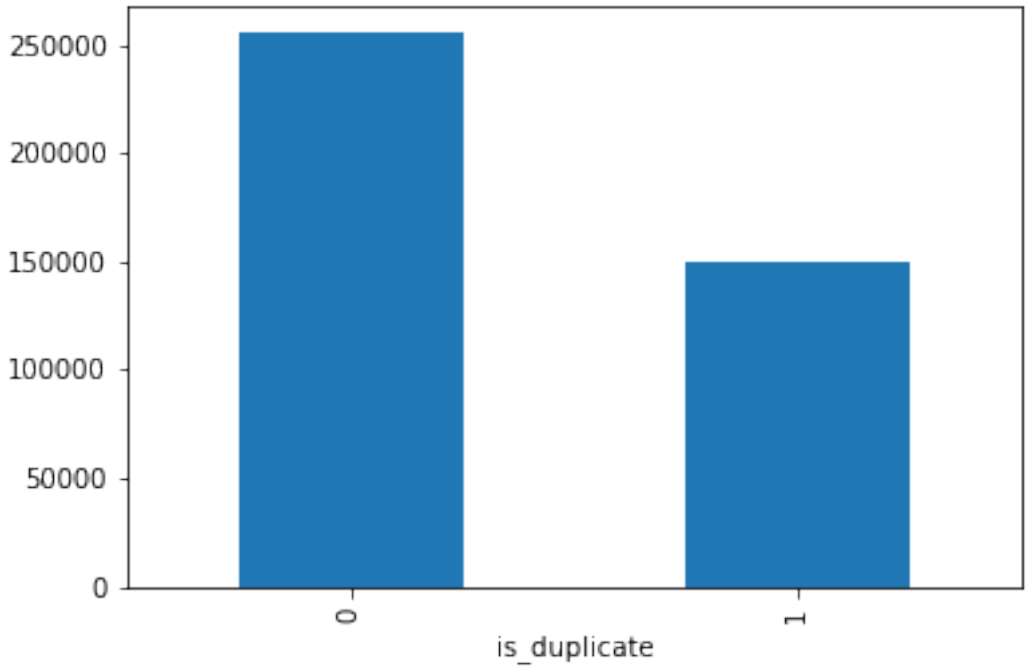
[ ]:
```
df.columns
```

[ ]:
```
Index(['id', 'qid1', 'qid2', 'question1', 'question2', 'is_duplicate'],
dtype='object')
```

1.2 Plotting the duplicate(1) and non duplicate(0) questions id.

[ ]:
```
df.groupby("is_duplicate")["id"].count().plot.bar()
```

[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f1180cd6650>
```



[ ]:
```
print(" {}% of data points is non-duplicate (not Similar).".format(100 -␣
↪round(df['is_duplicate'].mean() *100,2)))
```

```
63.08% of data points is non-duplicate (not Similar).
```

[ ]:
```
print("{}% of data points is duplicate (Similar)." .
↪format(round(df["is_duplicate"].mean() *100 , 2)))
```

```
36.92% of data points is duplicate (Similar).
```

[ ]:
```
# the code aims to analyze the values in the 'qid1' and 'qid2' columns of the␣
↪DataFrame and provide insights about the uniqueness
# and frequency of the values present in these columns.
qids = pd.Series(df['qid1'].tolist() + df['qid2'].tolist())
```

```
unq_qstn = len(np.unique(qids))
qstn_more_than_one = np.sum(qids.value_counts() > 1)
print(qids.value_counts())
```

```
2559      157
30782     120
4044      111
2561       88
14376      79
          …
416446      1
416444      1
416441      1
416439      1
537933      1
Length: 537933, dtype: int64
```

[ ]:
```
print("Number of unique question that appear more than once are {} {}%".
  ↪format(qstn_more_than_one , qstn_more_than_one/unq_qstn*100))
```

```
Number of unique question that appear more than once are 111780
20.77953945937505%
```

[ ]:
```
np.where(qids==2559)
# Question id 2559 that repeated max 157 times.
```

[ ]:
```
(array([ 14712,  38200,  56239,  81363,  81973,  82016,  86631,  89295,
        106632, 113625, 115228, 115816, 132320, 134629, 140355, 161485,
        202883, 213954, 216562, 228052, 228265, 253672, 263505, 268883,
        273689, 277652, 288565, 306135, 326227, 327551, 339152, 345086,
        346570, 360010, 377925, 379679, 381257, 387610, 390048, 390396,
        390423, 402909, 405573, 406580, 413055, 413291, 416537, 423174,
        424601, 430411, 437518, 441079, 443432, 454554, 460749, 461301,
        464299, 464481, 477254, 481244, 483415, 485065, 489063, 489338,
        495118, 500642, 508987, 511594, 512300, 523253, 524870, 529593,
        531767, 535880, 541727, 542669, 549361, 551582, 553411, 554847,
        555661, 558792, 569210, 572732, 574029, 574958, 576006, 576139,
        582954, 583724, 595202, 595902, 598104, 600816, 604358, 618444,
        618709, 619190, 623634, 623878, 634487, 636050, 638455, 640661,
        641556, 643098, 643585, 643874, 644415, 644517, 647931, 651401,
        651844, 655912, 656443, 670105, 670449, 672320, 674205, 675331,
        675826, 683168, 683538, 684803, 684920, 690001, 690112, 698005,
        704534, 708032, 709302, 709648, 719759, 728488, 730950, 736108,
        740937, 754889, 754986, 756201, 764848, 765572, 767892, 768339,
        773928, 782048, 783430, 784552, 789666, 789758, 790177, 790756,
        795340, 795608, 802874, 803258, 805535]),)
```

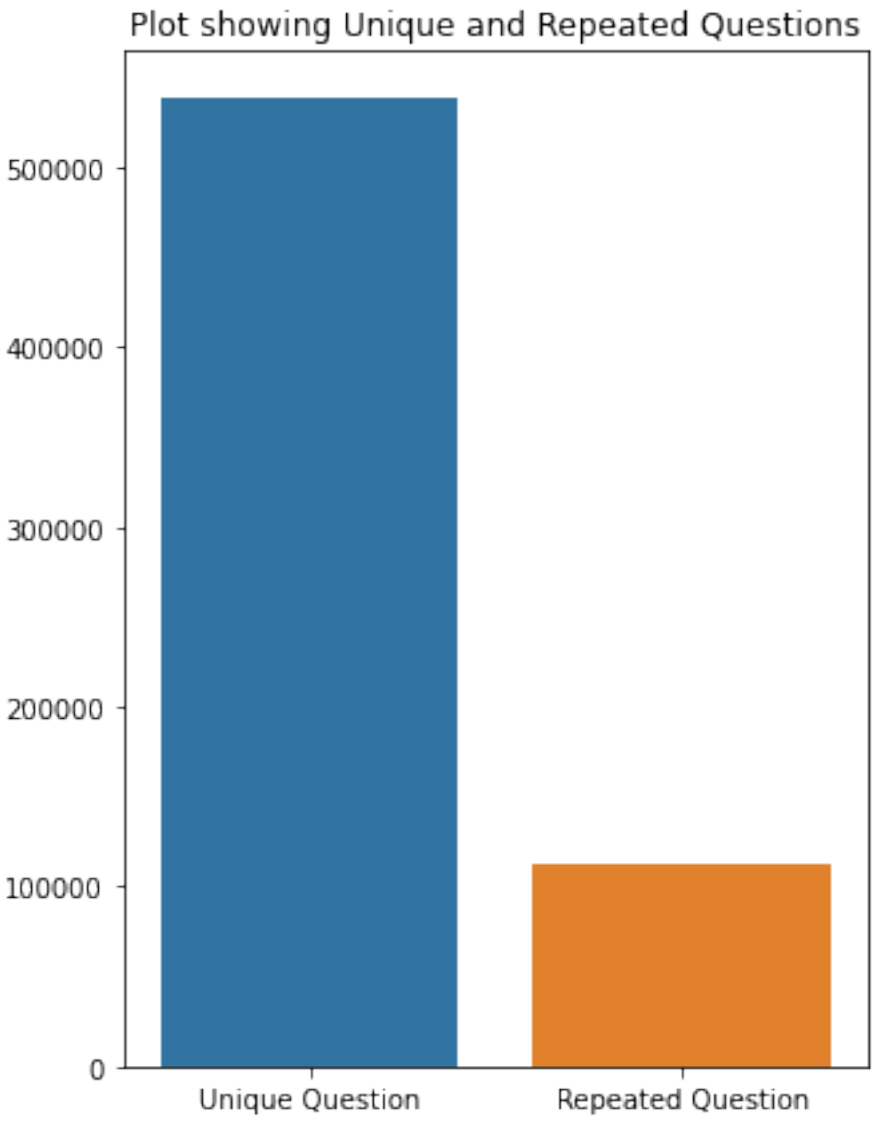
[ ]:
```
print(df.loc[38200])
```

```
id                                                    38200
qid1                                                   2559
qid2                                                   2711
question1          What are the best ways to lose weight?
question2       What is the best method of losing weight?
is_duplicate                                               1
Name: 38200, dtype: object
```

Question with **qid1= *2559*** repeated max **157** times and the question is " **What are the best way to lose weight?** "

1.3 Plotting Unique and Repeated Questions

[ ]:
```
x = ['Unique Question' , 'Repeated Question']
y = [unq_qstn , qstn_more_than_one]
plt.figure(figsize=(5,7))
sns.barplot(x,y)
plt.title("Plot showing Unique and Repeated Questions")
plt.show()
```



–> *Checking the duplicate Pair of Questions*

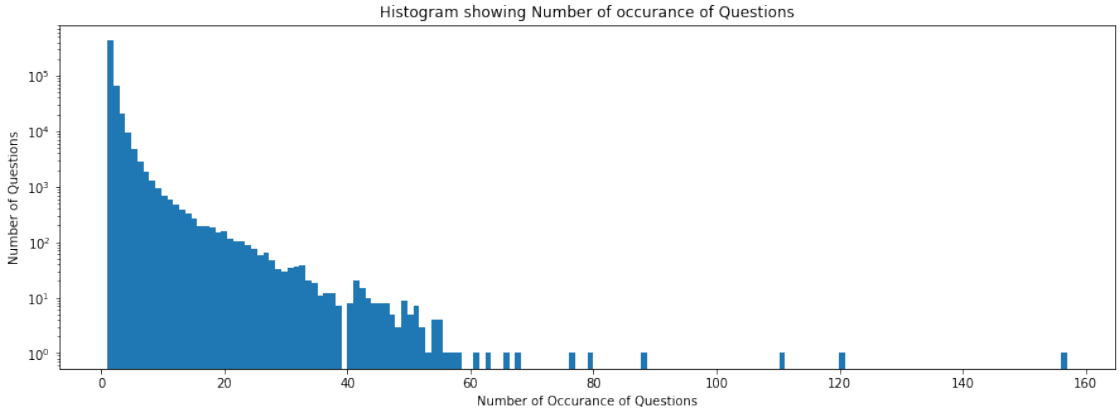
[ ]:
```
pair_duplicates = df[["qid1","qid2","is_duplicate"]].groupby(["qid1" ,"qid2" ]).
↪count().reset_index()
print("Number of Duplicate Questions",(pair_duplicates).shape[0] - df.shape[0])
```

```
Number of Duplicate Questions 0
```

1.4 Plotting number of occurance of questions

[ ]:
```
plt.figure(figsize=(15,5))
plt.hist(qids.value_counts() , bins=160)
plt.yscale('log',nonposy='clip')
```

```
plt.xlabel("Number of Occurance of Questions")
plt.ylabel("Number of Questions")
plt.title("Histogram showing Number of occurance of Questions")
plt.show()
print("Maximum number of time a single question occure is {}".format(max(qids.
  ↪value_counts())))
```




```
Maximum number of time a single question occure is 157
```

–> Checking the Null Values

[ ]:
```
narow = df[df.isnull().any(1)]
narow
```

[ ]:
```
            id    qid1    qid2                         question1  \
105780  105780  174363  174364    How can I develop android app?
201841  201841  303951  174364  How can I create an Android app?
363362  363362  493340  493341                               NaN

                                                 question2  is_duplicate
105780                                                 NaN             0
201841                                                 NaN             0
363362  My Chinese name is Haichao Yu. What English na…                0
```

OBSERVATION: Column question2 has **Two** null value and column question1 has **One** null value.

[ ]:
```
# removing the null values.
df = df.fillna('')
na_row = df[df.isnull().any(1)]
print(na_row)
```

```
Empty DataFrame
Columns: [id, qid1, qid2, question1, question2, is_duplicate]
Index: []
```

1.5 Lets try some basic **Feature Extraction** before cleaning the data

- _____freq_qid1_____
- _____freq_qid2_____
- _____q1_len_____
- _____q2_len_____
- _____q1_n_words_____
- _____q2_n_words_____
- _____word_common_____
- _____word_total_____
- _____word_share_____
- _____freq_qid1 + freq_qid2_____
- _____freq_qid1 - freq_qid2_____

1.5.1 The below code snippet is essentially extracting various features from the 'df' DataFrame and saving the modified DataFrame to the CSV file for further analysis or usage.

[ ]:
```
if os.path.isfile("/content/gdrive/My Drive/Quora dataset/
↪basic_feature_extraction.csv"): # if file.csv in my gdrive then open else␣
↪create new file.csv
  df = pd.read_csv("/content/gdrive/My Drive/Quora dataset/
↪basic_feature_extraction.csv" , encoding="latin-1")
else:
  df["freq_qid1"] = df.groupby('qid1')['qid1'].transform('count') # checking␣
↪repeating frequency of question1
  df["freq_qid2"] = df.groupby('qid2')['qid2'].transform('count') # ---do----
  df["q1_len"]    = df['question1'].str.len()   # creating length of string eg:
↪- shubham balgotra ----o/p= 16(including space as character)
  df["q2_len"]    = df['question2'].str.len()   #----do---
  df["q1_n_words"]= df['question1'].apply(lambda row: len(row.split(" ")))   #␣
↪apply() use to apply function, lambda keyword is used to define an anonymous␣
↪function in Python.
  df["q2_n_words"]= df['question2'].apply(lambda row: len(row.split(" ")))  ␣
↪#----do----

  def normalized_word_common(row):
    w1 = set(map(lambda word: word.lower().strip() , row['question1'].split("␣
↪")))   # set() arrange in ascending order, map() allows you to process and␣
↪transform all the items in an iterable without using an explicit for loop
    w2 = set(map(lambda word: word.lower().strip() , row['question2'].split("␣
↪")))   #strip() remove whitespaces.
    return 1.0 * len(w1 & w2)   # question words split with space, perform␣
↪lower() and strip() function on it and store to w1 and w2 respect. and then␣
↪using AND on w1 and w2.
  df["word_common"] = df.apply(normalized_word_common , axis=1)

  def normalized_word_total(row):
```

```
    w1 = set(map(lambda word: word.lower().strip() , row['question1'].split(" "
↪")))
    w2 = set(map(lambda word: word.lower().strip() , row['question2'].split(" "
↪")))
    return 1.0 * (len(w1) + len(w2)) # question words split with space, perform
↪lower() and strip() function on it and store to w1 and w2 respect. and then
↪using OR on w1 and w2.
  df["word_total"] = df.apply(normalized_word_total , axis=1)

  def normalized_word_share(row):
    w1 = set(map(lambda word: word.lower().strip() , row['question1'].split(" "
↪")))
    w2 = set(map(lambda word: word.lower().strip() , row['question2'].split(" "
↪")))
    return 1.0 * len(w1 & w2)/(len(w1)+len(w2)) # question words split with
↪space, perform lower() and strip() function on it and store to w1 and w2
↪respect. and then using AND/OR on w1 and w2.
  df["word_share"] = df.apply(normalized_word_share , axis=1)

  df["freq_qid1+qid2"] = df["freq_qid1"] + df["freq_qid2"]   # Adding occurance
↪of both questions
  df["freq_qid1-qid2"] = abs(df["freq_qid1"] - df["freq_qid2"]) # Subtracting
↪occurance of questions

  df.to_csv("/content/gdrive/My Drive/Quora dataset/basic_feature_extraction.
↪csv", index=False)

df.head(2)
```

[ ]:

	id	qid1	qid2	question1	question2	is_duplicate	freq_qid1	freq_qid2	q1_len	q2_len	q1_n_words	q2_n_words	word_common	word_total	word_share	freq_qid1+qid2	freq_qid1-qid2
0	0	1	2	What is the step by step guide to invest in sh…	What is the step by step guide to invest in sh…	0	1	1	66	57	14	12	10.0	23.0	0.434783	2	0
1	1	3	4	What is the story of Kohinoor (Koh-i-Noor) Dia…	What would happen if the Indian government sto…	0	4	1	51	88	8	13	4.0	20.0	0.200000	5	3

[ ]:
```
df.shape
```

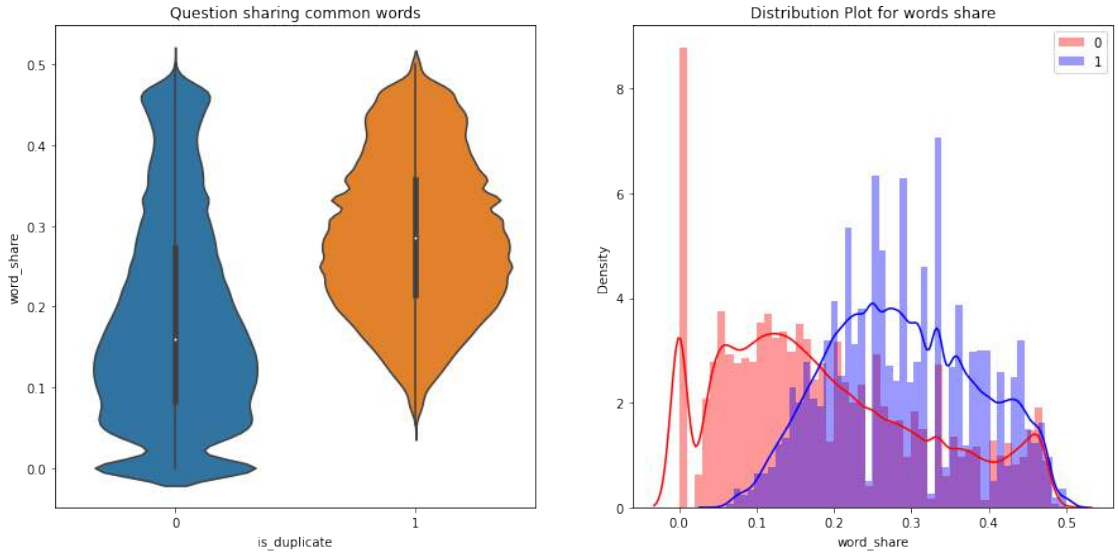
[ ]: (404290, 17)

[ ]:
```
print("Minimum length of words in question1 {}".format(min(df['q1_n_words'])))
print("Minimum length of words in question2 {}\n".format(min(df['q2_n_words'])))

print("Maximum length of words in question1 {}".format(max(df['q1_n_words'])))
print("Maximum length of words in question2 {}\n".format(max(df['q2_n_words'])))
print("Number of questions in question1 containing ONE word only = {}".
  ↪format(len(df[df['q1_n_words']==1])))
print("Number of questions in question2 containing ONE word only = {}".
  ↪format(len(df[df['q2_n_words']==1])))
```

```
Minimum length of words in question1 1
Minimum length of words in question2 1

Maximum length of words in question1 125
Maximum length of words in question2 237

Number of questions in question1 containing ONE word only = 67
Number of questions in question2 containing ONE word only = 24
```

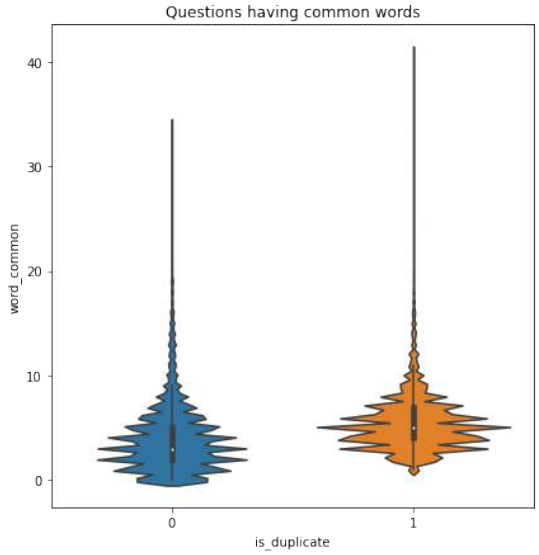
1.5.2 Analysing feature Word_share.

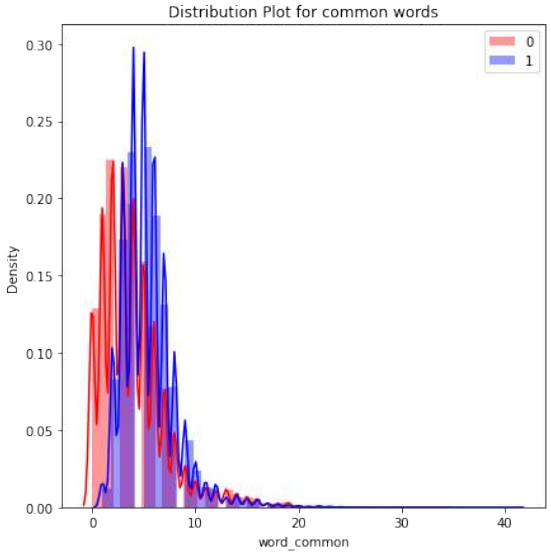
[ ]:
```
plt.figure(figsize=(15,7))
plt.subplot(1,2,1)
plt.title("Question sharing common words")
sns.violinplot(x="is_duplicate", y="word_share", data=df[0:])

plt.subplot(1,2,2)
sns.distplot(df[df["is_duplicate"] == 0.0]["word_share"][0:], label='0',␣
  ↪color='red')
sns.distplot(df[df['is_duplicate'] == 1.0]["word_share"][0:], label='1',␣
  ↪color='blue')
plt.legend()
plt.title("Distribution Plot for words share")
plt.show()
```



1.5.3 Analysing feature Word_common.

[ ]:
```
plt.figure(figsize=(15,7))
plt.subplot(1,2,1)
plt.title("Questions having common words")
sns.violinplot(x='is_duplicate', y='word_common', data=df)

plt.subplot(1,2,2)
plt.title("Distribution Plot for common words")
sns.distplot(df[df['is_duplicate'] == 0.0]['word_common'][0:], label='0',␣
 ↪color='red')
sns.distplot(df[df['is_duplicate'] == 1.0]['word_common'][0:], label='1',␣
 ↪color='blue')
plt.legend()
plt.show()
```





1.6 Preprocessing Text

1.6.1 The preprocess() function applies various text cleaning and normalization techniques to the input text, such as converting to lowercase, replacing specific patterns or words, stemming, and removing HTML tags. These steps help standardize and clean the text data for further analysis or processing.

[ ]:
```
STOP_WORDS = stopwords.words("english")
SAFE_DIV = 0.00001

def preprocess(x):
  x = str(x).lower()
  x = x.replace(',000,000', 'm').replace(',000', 'k').replace(" ", "'").
 ↪replace("’", "'")\
                           .replace("won't", "will not").replace("cannot", "can␣
 ↪not").replace("can't", "can not")\
                           .replace("n't", " not").replace("what's", "what is").
 ↪replace("it's", "it is")\
                           .replace("'ve", " have").replace("i'm", "i am").
 ↪replace("'re", " are")\
                           .replace("he's", "he is").replace("she's", "she is").
 ↪replace("'s", " own")\
                           .replace("%", " percent ").replace(" ", " rupee ").
 ↪replace("$", " dollar ")\
                           .replace("€", " euro ").replace("'ll", " will")
  x = re.sub(r"([0-9]+)000000", r"\1m",x)
  x = re.sub(r"([0-9]+)000" , r"\1k",x)

  porter = PorterStemmer()
```

```
    pattern = re.compile("\W")

    if type(x) == type(''):
        x = re.sub(pattern, " ",x)

    if type(x) ==  type(''):
        x = porter.stem(x)
        example = BeautifulSoup(x)
        x = example.get_text()

    return x
```

1.6.2 The token_feature list, containing calculated features, is returned by the function. These features can be useful for further analysis or modeling tasks involving natural language processing and question similarity. Also creating the new datafield with these new features.

[ ]:
```
def get_token_feature(q1,q2):
    token_feature = [0.0]*10
    q1_token = q1.split()
    q2_token = q2.split()
    if(len(q1_token) == 0 or len(q2_token) == 0):
        return token_feature
    q1_stop = set([word for word in q1_token if word in STOPWORDS])
    q2_stop = set([word for word in q2_token if word in STOPWORDS])

    q1_word = set([word for word in q1_token if word not in STOPWORDS])
    q2_word = set([word for word in q2_token if word not in STOPWORDS])

    common_word_count  = len(q1_word.intersection(q2_word))
    common_stop_count  = len(q2_stop.intersection(q2_stop))
    common_token_count = len(set(q1_token).intersection(set(q2_token)))

    token_feature[0] = common_word_count / (min(len(q1_word) , len(q2_word)) +␣
↪SAFE_DIV)
    token_feature[1] = common_word_count / (max(len(q1_word) , len(q2_word)) +␣
↪SAFE_DIV)
    token_feature[2] = common_stop_count / (min(len(q1_stop) , len(q2_word)) +␣
↪SAFE_DIV)
    token_feature[3] = common_stop_count / (max(len(q1_stop) , len(q2_stop)) +␣
↪SAFE_DIV)
    token_feature[4] = common_token_count / (min(len(q1_token) , len(q2_token)) +␣
↪SAFE_DIV)
    token_feature[5] = common_token_count / (max(len(q1_token) , len(q2_token)) +␣
↪SAFE_DIV)
    token_feature[6] = int(q1_token[-1] == q2_token[-1])
    token_feature[7] = int(q1_token[0] == q2_token[0])
    token_feature[8] = abs(len(q1_token) - len(q2_token))
```

```
  token_feature[9] = (len(q1_token) + len(q2_token)) / 2
  return token_feature

# Getting longest common sub string in question1 and question2
def get_longest_common_substring(a,b):
  string = list(distance.lcsubstrings(a,b))
  if len(string) == 0:
    return 0
  else:
    return len(string[0]) / (min(len(a) , len(b)) +1)

def extract_features(df):

  df['question1'] = df['question1'].fillna('').apply(preprocess)
  df['question2'] = df['question2'].fillna('').apply(preprocess)

  token_feature  = df.apply(lambda x: get_token_feature (x['question1'],␣
 ↪x['question2']), axis = 1)

  df['cwc_min']                   = list(map(lambda x: x[0] , token_feature))
  df['cwc_max']                   = list(map(lambda x: x[1] , token_feature))
  df['csc_min']                   = list(map(lambda x: x[2] , token_feature))
  df['csc_max']                   = list(map(lambda x: x[3] , token_feature))
  df['ctc_min']                   = list(map(lambda x: x[4] , token_feature))
  df['ctc_max']                   = list(map(lambda x: x[5] , token_feature))
  df['last_word_common']          = list(map(lambda x: x[6] , token_feature))
  df['first_word_common']         = list(map(lambda x: x[7] , token_feature))
  df['abs_len_diff']              = list(map(lambda x: x[8] , token_feature))
  df['mean_ratio']                = list(map(lambda x: x[9] , token_feature))

  df['fuzz_ratio']                = df.apply(lambda x: fuzz.
 ↪QRatio(x['question1'] , x['question2']), axis=1)
  df['fuzz_partial_ratio']        = df.apply(lambda x: fuzz.
 ↪partial_ratio(x['question1'] , x['question2']), axis=1)
  df['token_set_ratio']           = df.apply(lambda x: fuzz.
 ↪token_set_ratio(x['question1'] , x['question2']), axis=1)
  df['token_sort_ratio']          = df.apply(lambda x: fuzz.
 ↪token_sort_ratio(x['question1'] , x['question2']), axis=1)
  df['longest_common_substring']  = df.apply(lambda x:␣
 ↪get_longest_common_substring(x['question1'] , x['question2']), axis=1)
  return df
```

[ ]:
```
if os.path.isfile("/content/gdrive/My Drive/Quora dataset/nlp_feature_train.
 ↪csv"):
  df2 = pd.read_csv("/content/gdrive/My Drive/Quora dataset/nlp_feature_train.
 ↪csv")
```

```
else:
  df2 = pd.read_csv("/content/gdrive/My Drive/Quora dataset/train.csv")
  df2 = extract_features(df2)
  df2.to_csv("/content/gdrive/My Drive/Quora dataset/nlp_feature_train.csv" ,␣
 ↪index=False)
```

[ ]:
```
df2.head(2)
```

[ ]:
```
   id  qid1  qid2                                                question1  \
0   0     1     2  what is the step by step guide to invest in sh…
1   1     3     4  what is the story of kohinoor  koh i noor  dia…

                                            question2  is_duplicate   cwc_min  \
0  what is the step by step guide to invest in sh…              0  0.999998
1  what would happen if the indian government sto…              0  0.799998

    cwc_max   csc_min   csc_max  …   ctc_max  last_word_common  \
0  0.833332  1.199998  0.999998  …  0.785714               0.0
1  0.444444  0.999998  0.999998  …  0.466666               0.0

   first_word_common  abs_len_diff  mean_ratio  fuzz_ratio  \
0                1.0           2.0        13.0          93
1                1.0           5.0        12.5          66

   fuzz_partial_ratio  token_set_ratio  token_sort_ratio  \
0                 100              100                93
1                  75               86                63

   longest_common_substring
0                  0.982759
1                  0.596154

[2 rows x 21 columns]
```

```
Warning: Total number of columns (21) exceeds max_columns (20) limiting to first
(20) columns.
```

1.7 Analysing Extracting Features

[ ]:
```
# Creating and Saving positive and negative questios.
df_duplicate    = df2[df2['is_duplicate'] == 1]
df_nonduplicate = df2[df2['is_duplicate'] == 0]

p = np.dstack([df_duplicate["question1"] , df_duplicate["question2"]]).flatten()
n = np.dstack([df_nonduplicate["question1"] , df_nonduplicate["question2"]]).
 ↪flatten()

print("Number of data points in duplicate questions are {}".format(len(p)))
```

```
print("Number of data points in non_duplicate questions are {}".format(len(n)))

np.savetxt("/content/gdrive/My Drive/Quora dataset/train_p.txt", p , fmt='%s',␣
 ↪delimiter=' ')
np.savetxt("/content/gdrive/My Drive/Quora dataset/train_n.txt", n , fmt='%s',␣
 ↪delimiter=' ')
```

```
Number of data points in duplicate questions are 298526
Number of data points in non_duplicate questions are 510054
```

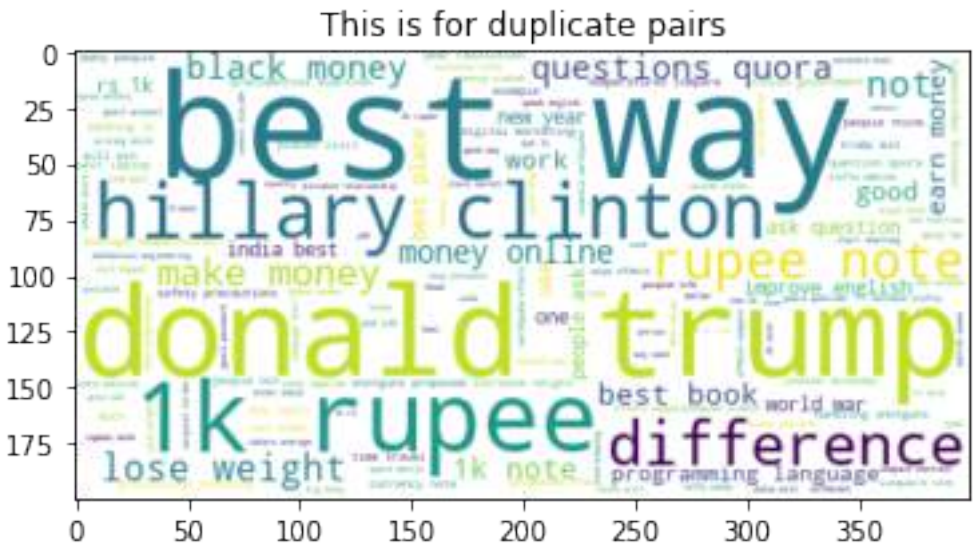
[ ]:
```
link = path.dirname("/content/gdrive/My Drive/Quora dataset/")
textp_w = open(path.join(link, "train_p.txt")).read()
textn_w = open(path.join(link, "train_n.txt")).read()
```

[ ]:
```
stopwords = set(STOPWORDS)
stopwords.add(" ")
stopwords.remove("not")
stopwords.remove("no")
print("Number of words in duplicate pair of questions :",format(len(textp_w)))
print("Number of words in non duplicate pair of questions :
 ↪",format(len(textn_w)))
```

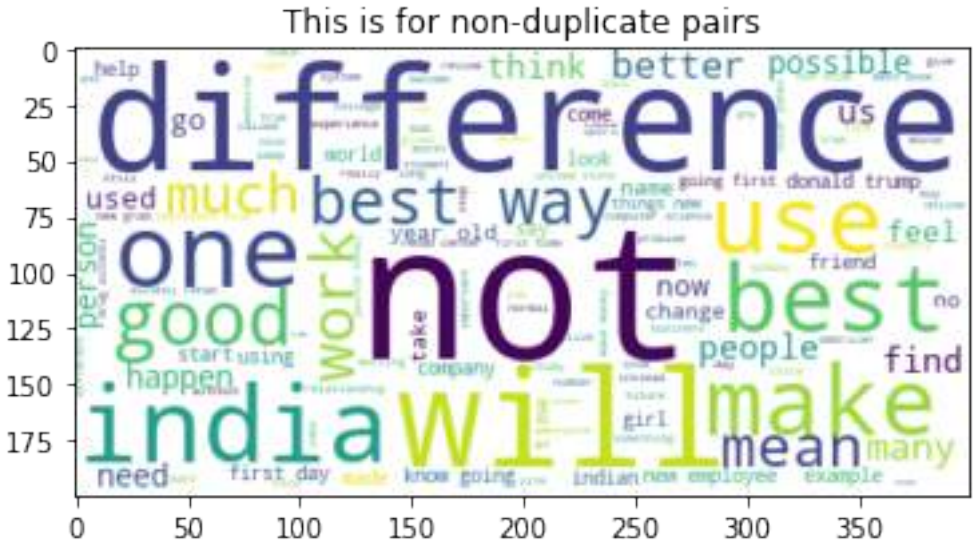
```
Number of words in duplicate pair of questions : 16109886
Number of words in non duplicate pair of questions : 33193130
```

1.8 Building Wordloud

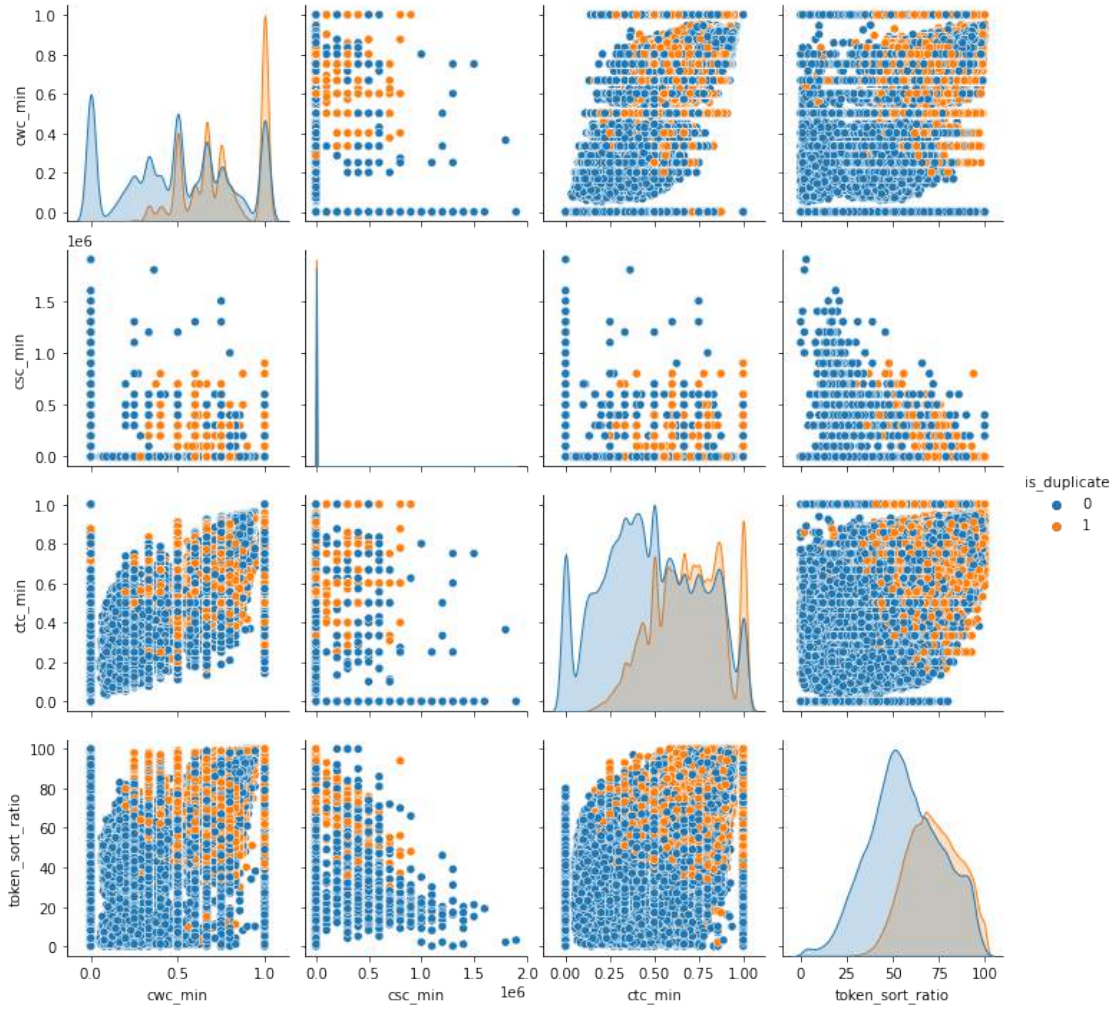
[ ]:
```
wc = WordCloud(stopwords=stopwords , background_color='white' ,␣
 ↪max_words=len(textp_w))
wc.generate(textp_w)
plt.title("This is for duplicate pairs")
plt.imshow(wc , interpolation='bilinear',)
#plt.axis('off')
plt.show()
```



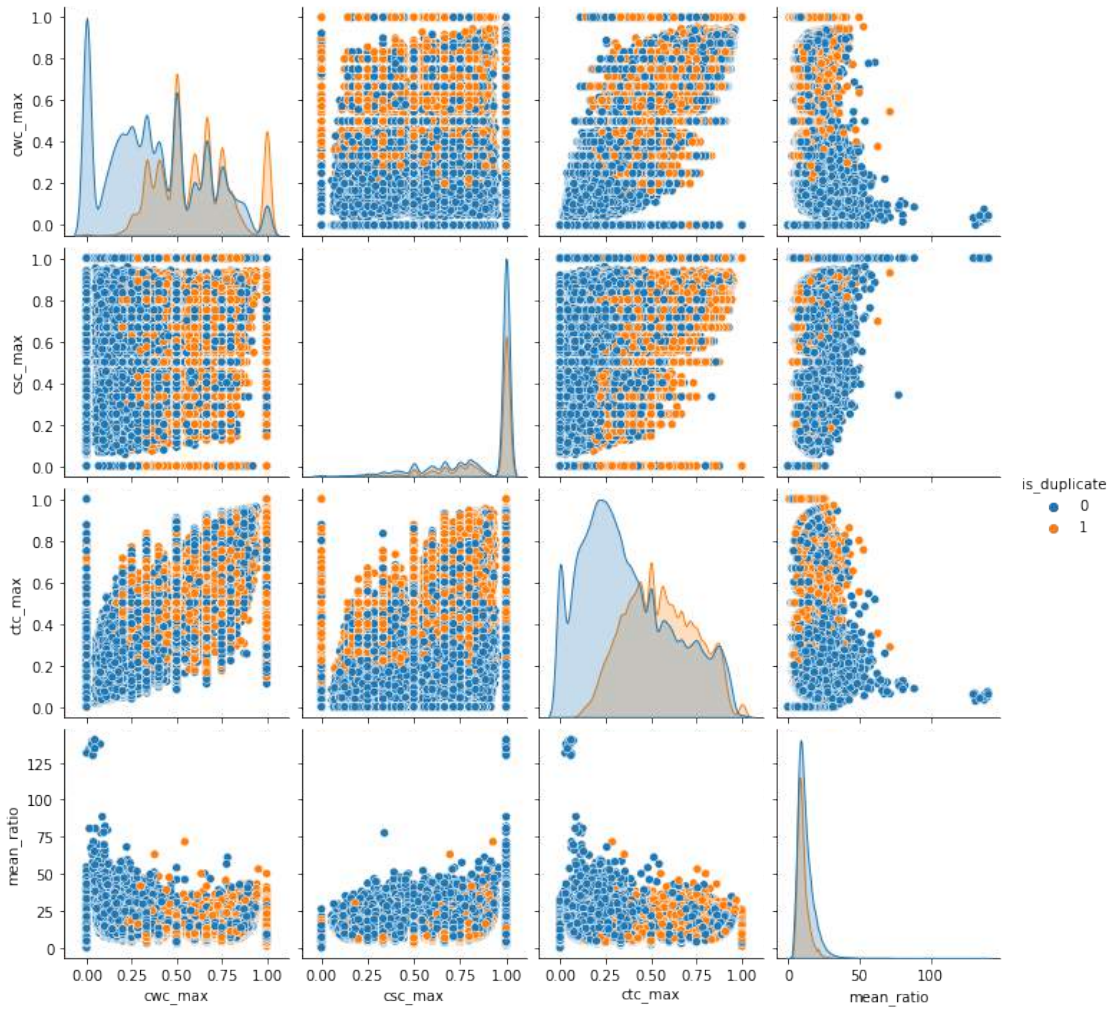
[ ]:
```
wc = WordCloud(stopwords=stopwords , background_color='white' ,␣
↪max_words=len(textn_w))
wc.generate(textn_w)
plt.title("This is for non-duplicate pairs")
plt.imshow(wc , interpolation='bilinear')
#plt.axis("off")
plt.show()
```



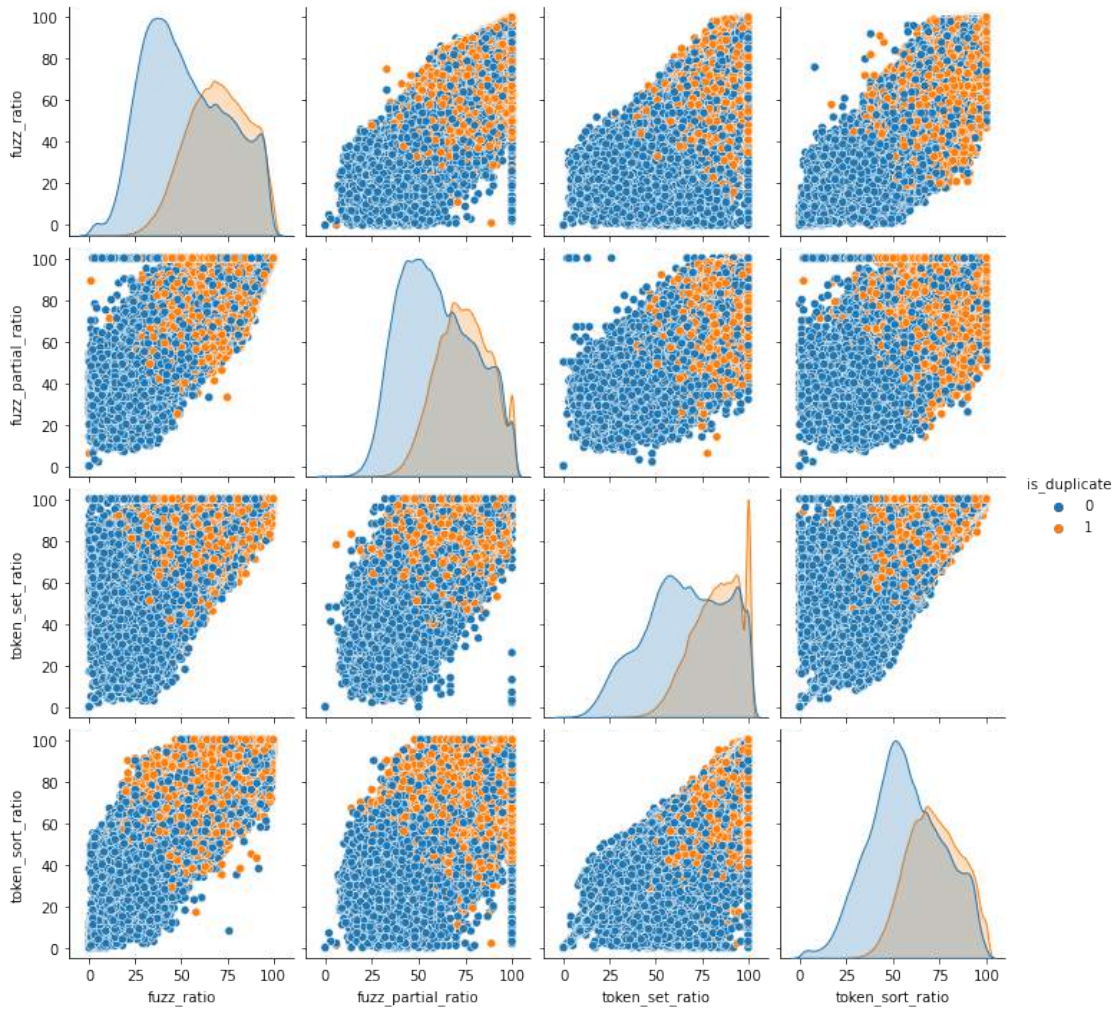
[ ]:
```
df2.shape[0]
sns.pairplot(df2 , hue='is_duplicate' , vars = ['cwc_min' , 'csc_min',␣
 ↪'ctc_min' , 'token_sort_ratio'])
plt.show()
```



[ ]:
```
df2.shape[0]
sns.pairplot(df2[['cwc_max' , 'csc_max' , 'ctc_max' ,'mean_ratio',␣
 ↪'is_duplicate']] , hue='is_duplicate' , vars = ['cwc_max' , 'csc_max' ,␣
 ↪'ctc_max' ,'mean_ratio'])
plt.show()
```

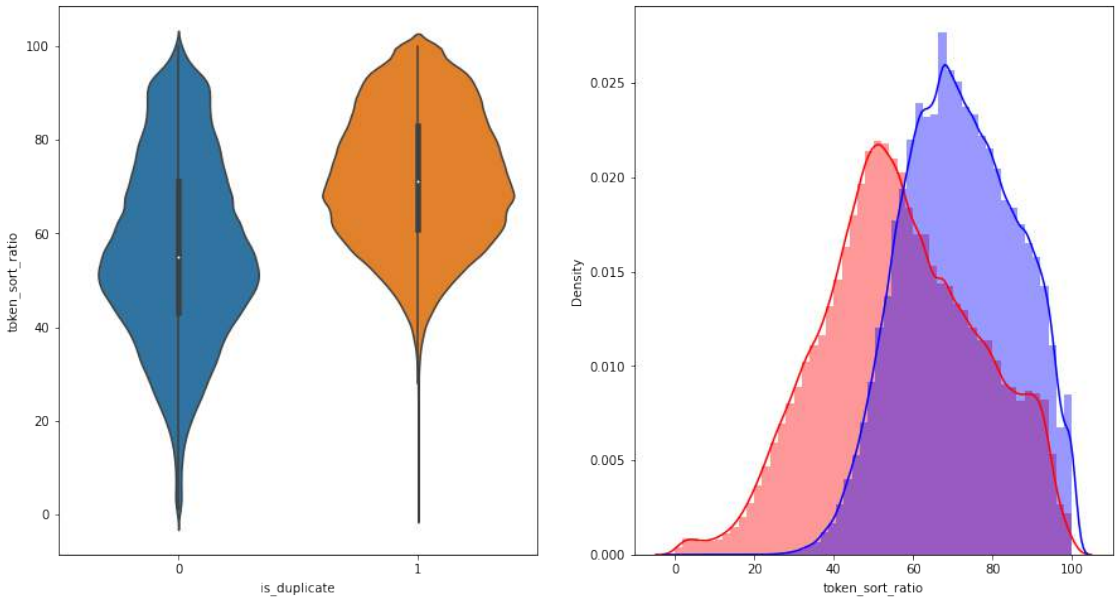


[ ]:
```
df2.shape[0]
sns.pairplot(df2[['fuzz_ratio' , 'fuzz_partial_ratio' , 'token_set_ratio' ,␣
 ↪'token_sort_ratio', 'is_duplicate']] , hue='is_duplicate' ,␣
 ↪vars=['fuzz_ratio' , 'fuzz_partial_ratio' , 'token_set_ratio' ,␣
 ↪'token_sort_ratio', ])
plt.show()
```



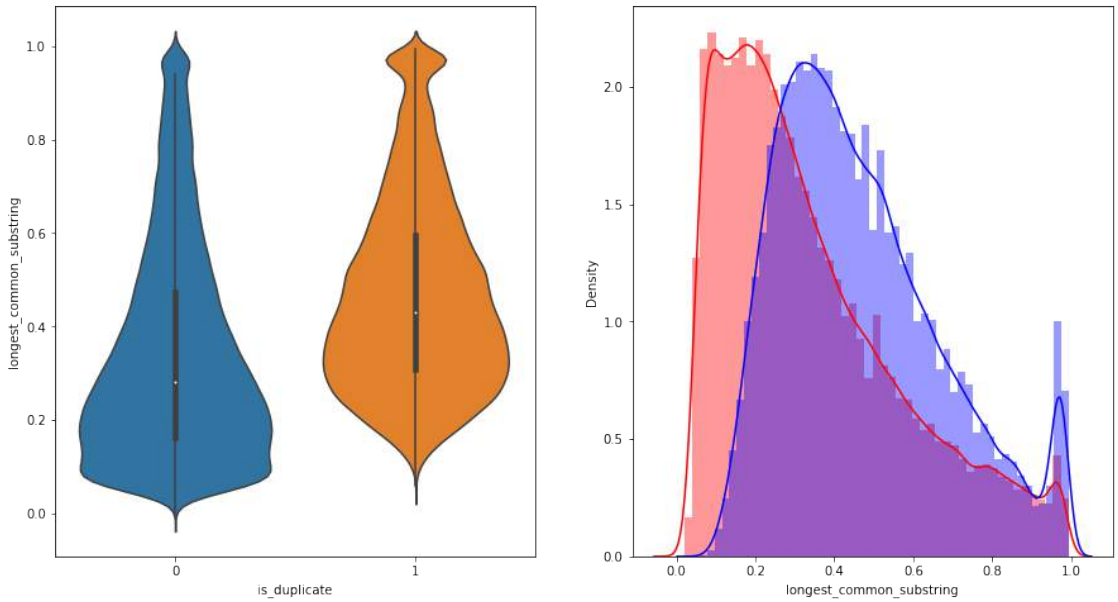
1.9 Plotting Violin Plot and Distributed Plot for feature Token sort ratio.

[ ]:
```
plt.figure(figsize=(15,8))
plt.subplot(1,2,1)
sns.violinplot(x='is_duplicate' , y='token_sort_ratio' , data = df2)

plt.subplot(1,2,2)
sns.distplot(df2[df2['is_duplicate'] == 0.0] ['token_sort_ratio'] , label = '0',
↪, color = 'red' )
sns.distplot(df2[df2['is_duplicate'] == 1.0] ['token_sort_ratio'] , label = '1',
↪, color = 'blue')
plt.show()
```



1.10 Plotting Violin Plot and Distributed Plot for feature Longest common substring.

[ ]:
```
plt.figure(figsize=(15,8))
plt.subplot(1,2,1)
sns.violinplot(x='is_duplicate' , y='longest_common_substring' , data = df2)

plt.subplot(1,2,2)
sns.distplot(df2[df2['is_duplicate'] == 0.0] ['longest_common_substring'] ,␣
 ↪label = '0' , color = 'red' )
sns.distplot(df2[df2['is_duplicate'] == 1.0] ['longest_common_substring'] ,␣
 ↪label = '1' , color = 'blue')
plt.show()
```



1.11 Scale the data between 0 and 1. For this we use MinMaxScaler

[ ]:
```
df.columns
```

[ ]:
```
Index(['id', 'qid1', 'qid2', 'question1', 'question2', 'is_duplicate',
       'freq_qid1', 'freq_qid2', 'q1_len', 'q2_len', 'q1_n_words',
       'q2_n_words', 'word_common', 'word_total', 'word_share',
       'freq_qid1+qid2', 'freq_qid1-qid2'],
      dtype='object')
```

[ ]:
```
df2.columns
```

[ ]:
```
Index(['id', 'qid1', 'qid2', 'question1', 'question2', 'is_duplicate',
       'cwc_min', 'cwc_max', 'csc_min', 'csc_max', 'ctc_min', 'ctc_max',
       'last_word_common', 'first_word_common', 'abs_len_diff', 'mean_ratio',
       'fuzz_ratio', 'fuzz_partial_ratio', 'token_set_ratio',
       'token_sort_ratio', 'longest_common_substring'],
      dtype='object')
```

[ ]:
```
# ust taking sample size.
df2_sample = df2[0:10000]
x = MinMaxScaler().fit_transform(df2_sample[['cwc_min', 'cwc_max', 'csc_min',␣
 ↪'csc_max', 'ctc_min', 'ctc_max','last_word_common', 'first_word_common',␣
 ↪'abs_len_diff', 'mean_ratio',
       'fuzz_ratio', 'fuzz_partial_ratio',␣
 ↪'token_set_ratio','token_sort_ratio', 'longest_common_substring']])
y = df2_sample['is_duplicate']
```

1.12 Plotting using TSNE

1.12.1 t-SNE (t-Distributed Stochastic Neighbor Embedding) is a dimensionality reduction technique commonly used for visualizing high-dimensional data in a lower-dimensional space. It is particularly useful for exploring and understanding complex patterns and structures in data.

The main idea behind t-SNE is to represent each data point as a two- or three-dimensional point on a scatter plot while preserving the local structure and relationships between data points from the original high-dimensional space. It accomplishes this by modeling the similarity between data points in both the high-dimensional and low-dimensional spaces.

[ ]:
```
# 2D TSNE
tsne_2d_per50 = TSNE(n_components = 2, verbose=2, init='random', perplexity=␣
 ↪50, n_iter=2000, random_state=100).fit_transform(x)
```

```
[t-SNE] Computing 151 nearest neighbors…
[t-SNE] Indexed 10000 samples in 0.033s…
[t-SNE] Computed neighbors for 10000 samples in 1.797s…
[t-SNE] Computed conditional probabilities for sample 1000 / 10000
[t-SNE] Computed conditional probabilities for sample 2000 / 10000
[t-SNE] Computed conditional probabilities for sample 3000 / 10000
[t-SNE] Computed conditional probabilities for sample 4000 / 10000
[t-SNE] Computed conditional probabilities for sample 5000 / 10000
[t-SNE] Computed conditional probabilities for sample 6000 / 10000
[t-SNE] Computed conditional probabilities for sample 7000 / 10000
[t-SNE] Computed conditional probabilities for sample 8000 / 10000
[t-SNE] Computed conditional probabilities for sample 9000 / 10000
[t-SNE] Computed conditional probabilities for sample 10000 / 10000
[t-SNE] Mean sigma: 0.120331
[t-SNE] Computed conditional probabilities in 1.618s
[t-SNE] Iteration 50: error = 89.1238556, gradient norm = 0.0224051 (50
iterations in 11.737s)
[t-SNE] Iteration 100: error = 73.3824539, gradient norm = 0.0036441 (50
iterations in 10.646s)
[t-SNE] Iteration 150: error = 71.4300690, gradient norm = 0.0020819 (50
iterations in 10.613s)
[t-SNE] Iteration 200: error = 70.7280579, gradient norm = 0.0014269 (50
iterations in 9.176s)
[t-SNE] Iteration 250: error = 70.3386993, gradient norm = 0.0011325 (50
iterations in 8.497s)
[t-SNE] KL divergence after 250 iterations with early exaggeration: 70.338699
[t-SNE] Iteration 300: error = 2.2458773, gradient norm = 0.0012729 (50
iterations in 11.790s)
[t-SNE] Iteration 350: error = 1.8016474, gradient norm = 0.0005447 (50
iterations in 6.332s)
[t-SNE] Iteration 400: error = 1.5921911, gradient norm = 0.0003203 (50
iterations in 4.705s)
[t-SNE] Iteration 450: error = 1.4726726, gradient norm = 0.0002147 (50
iterations in 4.769s)
```

```
[t-SNE] Iteration 500: error = 1.3957461, gradient norm = 0.0001593 (50
iterations in 4.734s)
[t-SNE] Iteration 550: error = 1.3422916, gradient norm = 0.0001239 (50
iterations in 4.610s)
[t-SNE] Iteration 600: error = 1.3034012, gradient norm = 0.0001000 (50
iterations in 6.661s)
[t-SNE] Iteration 650: error = 1.2737918, gradient norm = 0.0000841 (50
iterations in 5.725s)
[t-SNE] Iteration 700: error = 1.2509595, gradient norm = 0.0000745 (50
iterations in 4.664s)
[t-SNE] Iteration 750: error = 1.2333169, gradient norm = 0.0000676 (50
iterations in 4.609s)
[t-SNE] Iteration 800: error = 1.2200935, gradient norm = 0.0000630 (50
iterations in 4.559s)
[t-SNE] Iteration 850: error = 1.2100791, gradient norm = 0.0000590 (50
iterations in 4.602s)
[t-SNE] Iteration 900: error = 1.2020028, gradient norm = 0.0000573 (50
iterations in 4.632s)
[t-SNE] Iteration 950: error = 1.1953907, gradient norm = 0.0000541 (50
iterations in 4.614s)
[t-SNE] Iteration 1000: error = 1.1899886, gradient norm = 0.0000521 (50
iterations in 4.658s)
[t-SNE] Iteration 1050: error = 1.1855805, gradient norm = 0.0000486 (50
iterations in 4.645s)
[t-SNE] Iteration 1100: error = 1.1819538, gradient norm = 0.0000486 (50
iterations in 4.707s)
[t-SNE] Iteration 1150: error = 1.1789588, gradient norm = 0.0000444 (50
iterations in 4.625s)
[t-SNE] Iteration 1200: error = 1.1759641, gradient norm = 0.0000432 (50
iterations in 4.636s)
[t-SNE] Iteration 1250: error = 1.1733139, gradient norm = 0.0000418 (50
iterations in 4.654s)
[t-SNE] Iteration 1300: error = 1.1708270, gradient norm = 0.0000405 (50
iterations in 4.635s)
[t-SNE] Iteration 1350: error = 1.1684444, gradient norm = 0.0000402 (50
iterations in 4.634s)
[t-SNE] Iteration 1400: error = 1.1662292, gradient norm = 0.0000388 (50
iterations in 4.709s)
[t-SNE] Iteration 1450: error = 1.1642499, gradient norm = 0.0000379 (50
iterations in 5.646s)
[t-SNE] Iteration 1500: error = 1.1624308, gradient norm = 0.0000357 (50
iterations in 4.621s)
[t-SNE] Iteration 1550: error = 1.1605508, gradient norm = 0.0000364 (50
iterations in 4.646s)
[t-SNE] Iteration 1600: error = 1.1588383, gradient norm = 0.0000336 (50
iterations in 4.711s)
[t-SNE] Iteration 1650: error = 1.1572727, gradient norm = 0.0000327 (50
iterations in 4.665s)
```

```
[t-SNE] Iteration 1700: error = 1.1557095, gradient norm = 0.0000328 (50
iterations in 4.673s)
[t-SNE] Iteration 1750: error = 1.1542506, gradient norm = 0.0000321 (50
iterations in 5.627s)
[t-SNE] Iteration 1800: error = 1.1529474, gradient norm = 0.0000320 (50
iterations in 4.676s)
[t-SNE] Iteration 1850: error = 1.1517563, gradient norm = 0.0000315 (50
iterations in 4.702s)
[t-SNE] Iteration 1900: error = 1.1505815, gradient norm = 0.0000305 (50
iterations in 4.631s)
[t-SNE] Iteration 1950: error = 1.1494670, gradient norm = 0.0000299 (50
iterations in 4.670s)
[t-SNE] Iteration 2000: error = 1.1484246, gradient norm = 0.0000297 (50
iterations in 4.678s)
[t-SNE] KL divergence after 2000 iterations: 1.148425
```

[ ]:
```
tsne_2d_per30 = TSNE(n_components = 2, verbose=2, init='random', perplexity=␣
↪30, n_iter=2000, random_state=100).fit_transform(x)
```

```
[t-SNE] Computing 91 nearest neighbors…
[t-SNE] Indexed 10000 samples in 0.024s…
[t-SNE] Computed neighbors for 10000 samples in 1.053s…
[t-SNE] Computed conditional probabilities for sample 1000 / 10000
[t-SNE] Computed conditional probabilities for sample 2000 / 10000
[t-SNE] Computed conditional probabilities for sample 3000 / 10000
[t-SNE] Computed conditional probabilities for sample 4000 / 10000
[t-SNE] Computed conditional probabilities for sample 5000 / 10000
[t-SNE] Computed conditional probabilities for sample 6000 / 10000
[t-SNE] Computed conditional probabilities for sample 7000 / 10000
[t-SNE] Computed conditional probabilities for sample 8000 / 10000
[t-SNE] Computed conditional probabilities for sample 9000 / 10000
[t-SNE] Computed conditional probabilities for sample 10000 / 10000
[t-SNE] Mean sigma: 0.105405
[t-SNE] Computed conditional probabilities in 0.522s
[t-SNE] Iteration 50: error = 96.1028442, gradient norm = 0.0203163 (50
iterations in 5.270s)
[t-SNE] Iteration 100: error = 78.7991257, gradient norm = 0.0044192 (50
iterations in 4.169s)
[t-SNE] Iteration 150: error = 76.3004913, gradient norm = 0.0023867 (50
iterations in 3.803s)
[t-SNE] Iteration 200: error = 75.2546768, gradient norm = 0.0017121 (50
iterations in 4.123s)
[t-SNE] Iteration 250: error = 74.7221146, gradient norm = 0.0013129 (50
iterations in 10.719s)
[t-SNE] KL divergence after 250 iterations with early exaggeration: 74.722115
[t-SNE] Iteration 300: error = 2.5981379, gradient norm = 0.0012748 (50
iterations in 10.986s)
[t-SNE] Iteration 350: error = 2.0869210, gradient norm = 0.0005828 (50
```

```
iterations in 9.815s)
[t-SNE] Iteration 400: error = 1.8334410, gradient norm = 0.0003520 (50
iterations in 8.536s)
[t-SNE] Iteration 450: error = 1.6827555, gradient norm = 0.0002429 (50
iterations in 9.323s)
[t-SNE] Iteration 500: error = 1.5837038, gradient norm = 0.0001791 (50
iterations in 6.773s)
[t-SNE] Iteration 550: error = 1.5135000, gradient norm = 0.0001401 (50
iterations in 10.275s)
[t-SNE] Iteration 600: error = 1.4612315, gradient norm = 0.0001157 (50
iterations in 12.370s)
[t-SNE] Iteration 650: error = 1.4213074, gradient norm = 0.0000974 (50
iterations in 10.552s)
[t-SNE] Iteration 700: error = 1.3901994, gradient norm = 0.0000851 (50
iterations in 10.193s)
[t-SNE] Iteration 750: error = 1.3660527, gradient norm = 0.0000771 (50
iterations in 9.125s)
[t-SNE] Iteration 800: error = 1.3474269, gradient norm = 0.0000706 (50
iterations in 7.495s)
[t-SNE] Iteration 850: error = 1.3332534, gradient norm = 0.0000678 (50
iterations in 3.960s)
[t-SNE] Iteration 900: error = 1.3220465, gradient norm = 0.0000661 (50
iterations in 3.967s)
[t-SNE] Iteration 950: error = 1.3134146, gradient norm = 0.0000612 (50
iterations in 3.974s)
[t-SNE] Iteration 1000: error = 1.3062085, gradient norm = 0.0000596 (50
iterations in 3.990s)
[t-SNE] Iteration 1050: error = 1.3001913, gradient norm = 0.0000560 (50
iterations in 6.030s)
[t-SNE] Iteration 1100: error = 1.2948552, gradient norm = 0.0000545 (50
iterations in 3.954s)
[t-SNE] Iteration 1150: error = 1.2902144, gradient norm = 0.0000533 (50
iterations in 4.015s)
[t-SNE] Iteration 1200: error = 1.2860074, gradient norm = 0.0000510 (50
iterations in 4.947s)
[t-SNE] Iteration 1250: error = 1.2820382, gradient norm = 0.0000479 (50
iterations in 3.971s)
[t-SNE] Iteration 1300: error = 1.2783041, gradient norm = 0.0000461 (50
iterations in 4.045s)
[t-SNE] Iteration 1350: error = 1.2748585, gradient norm = 0.0000448 (50
iterations in 4.058s)
[t-SNE] Iteration 1400: error = 1.2714806, gradient norm = 0.0000429 (50
iterations in 4.084s)
[t-SNE] Iteration 1450: error = 1.2682548, gradient norm = 0.0000422 (50
iterations in 3.998s)
[t-SNE] Iteration 1500: error = 1.2653996, gradient norm = 0.0000422 (50
iterations in 4.023s)
[t-SNE] Iteration 1550: error = 1.2627816, gradient norm = 0.0000412 (50
```

```
iterations in 4.103s)
[t-SNE] Iteration 1600: error = 1.2605631, gradient norm = 0.0000391 (50
iterations in 4.010s)
[t-SNE] Iteration 1650: error = 1.2585409, gradient norm = 0.0000383 (50
iterations in 4.021s)
[t-SNE] Iteration 1700: error = 1.2565433, gradient norm = 0.0000367 (50
iterations in 3.949s)
[t-SNE] Iteration 1750: error = 1.2545985, gradient norm = 0.0000360 (50
iterations in 4.719s)
[t-SNE] Iteration 1800: error = 1.2527514, gradient norm = 0.0000353 (50
iterations in 3.943s)
[t-SNE] Iteration 1850: error = 1.2507801, gradient norm = 0.0000352 (50
iterations in 3.985s)
[t-SNE] Iteration 1900: error = 1.2490935, gradient norm = 0.0000357 (50
iterations in 4.008s)
[t-SNE] Iteration 1950: error = 1.2476227, gradient norm = 0.0000349 (50
iterations in 3.961s)
[t-SNE] Iteration 2000: error = 1.2462083, gradient norm = 0.0000341 (50
iterations in 4.019s)
[t-SNE] KL divergence after 2000 iterations: 1.246208
```

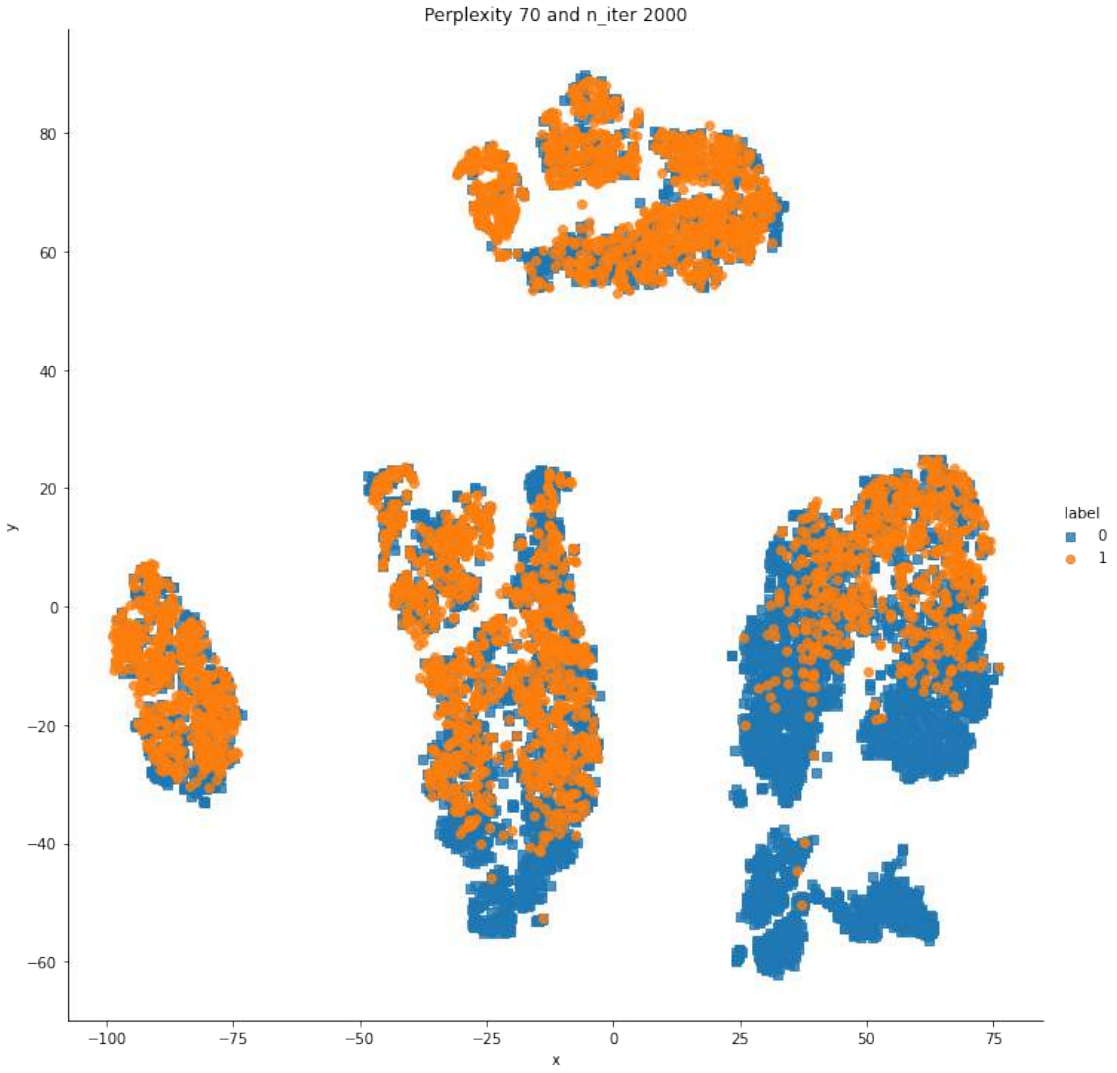
[ ]:
```
tsne_2d_per70 = TSNE(n_components = 2, verbose=2, init='random', perplexity=␣
↪70, n_iter=2000, random_state=100).fit_transform(x)
```

```
[t-SNE] Computing 211 nearest neighbors…
[t-SNE] Indexed 10000 samples in 0.024s…
[t-SNE] Computed neighbors for 10000 samples in 1.834s…
[t-SNE] Computed conditional probabilities for sample 1000 / 10000
[t-SNE] Computed conditional probabilities for sample 2000 / 10000
[t-SNE] Computed conditional probabilities for sample 3000 / 10000
[t-SNE] Computed conditional probabilities for sample 4000 / 10000
[t-SNE] Computed conditional probabilities for sample 5000 / 10000
[t-SNE] Computed conditional probabilities for sample 6000 / 10000
[t-SNE] Computed conditional probabilities for sample 7000 / 10000
[t-SNE] Computed conditional probabilities for sample 8000 / 10000
[t-SNE] Computed conditional probabilities for sample 9000 / 10000
[t-SNE] Computed conditional probabilities for sample 10000 / 10000
[t-SNE] Mean sigma: 0.131321
[t-SNE] Computed conditional probabilities in 1.569s
[t-SNE] Iteration 50: error = 84.5861282, gradient norm = 0.0241215 (50
iterations in 7.496s)
[t-SNE] Iteration 100: error = 69.8257141, gradient norm = 0.0033340 (50
iterations in 12.070s)
[t-SNE] Iteration 150: error = 68.1703415, gradient norm = 0.0018469 (50
iterations in 15.108s)
[t-SNE] Iteration 200: error = 67.5222168, gradient norm = 0.0012891 (50
iterations in 18.361s)
[t-SNE] Iteration 250: error = 67.1721954, gradient norm = 0.0010387 (50
```

```
iterations in 15.077s)
[t-SNE] KL divergence after 250 iterations with early exaggeration: 67.172195
[t-SNE] Iteration 300: error = 2.0192904, gradient norm = 0.0012786 (50
iterations in 13.368s)
[t-SNE] Iteration 350: error = 1.6211202, gradient norm = 0.0005205 (50
iterations in 11.505s)
[t-SNE] Iteration 400: error = 1.4376500, gradient norm = 0.0002977 (50
iterations in 11.703s)
[t-SNE] Iteration 450: error = 1.3356879, gradient norm = 0.0001969 (50
iterations in 10.593s)
[t-SNE] Iteration 500: error = 1.2715051, gradient norm = 0.0001435 (50
iterations in 11.804s)
[t-SNE] Iteration 550: error = 1.2276862, gradient norm = 0.0001112 (50
iterations in 9.759s)
[t-SNE] Iteration 600: error = 1.1959743, gradient norm = 0.0000907 (50
iterations in 10.014s)
[t-SNE] Iteration 650: error = 1.1722977, gradient norm = 0.0000785 (50
iterations in 9.602s)
[t-SNE] Iteration 700: error = 1.1543826, gradient norm = 0.0000697 (50
iterations in 10.540s)
[t-SNE] Iteration 750: error = 1.1405582, gradient norm = 0.0000615 (50
iterations in 12.859s)
[t-SNE] Iteration 800: error = 1.1298046, gradient norm = 0.0000566 (50
iterations in 6.729s)
[t-SNE] Iteration 850: error = 1.1213685, gradient norm = 0.0000548 (50
iterations in 5.089s)
[t-SNE] Iteration 900: error = 1.1148310, gradient norm = 0.0000518 (50
iterations in 5.030s)
[t-SNE] Iteration 950: error = 1.1097231, gradient norm = 0.0000483 (50
iterations in 4.981s)
[t-SNE] Iteration 1000: error = 1.1057383, gradient norm = 0.0000460 (50
iterations in 5.044s)
[t-SNE] Iteration 1050: error = 1.1024643, gradient norm = 0.0000455 (50
iterations in 5.029s)
[t-SNE] Iteration 1100: error = 1.0994843, gradient norm = 0.0000430 (50
iterations in 5.107s)
[t-SNE] Iteration 1150: error = 1.0970118, gradient norm = 0.0000427 (50
iterations in 5.045s)
[t-SNE] Iteration 1200: error = 1.0947062, gradient norm = 0.0000406 (50
iterations in 7.461s)
[t-SNE] Iteration 1250: error = 1.0927430, gradient norm = 0.0000376 (50
iterations in 5.499s)
[t-SNE] Iteration 1300: error = 1.0908074, gradient norm = 0.0000363 (50
iterations in 5.098s)
[t-SNE] Iteration 1350: error = 1.0889087, gradient norm = 0.0000347 (50
iterations in 5.038s)
[t-SNE] Iteration 1400: error = 1.0869468, gradient norm = 0.0000332 (50
iterations in 5.063s)
```

```
[t-SNE] Iteration 1450: error = 1.0851626, gradient norm = 0.0000326 (50
iterations in 5.075s)
[t-SNE] Iteration 1500: error = 1.0834668, gradient norm = 0.0000307 (50
iterations in 5.076s)
[t-SNE] Iteration 1550: error = 1.0816903, gradient norm = 0.0000297 (50
iterations in 5.166s)
[t-SNE] Iteration 1600: error = 1.0800558, gradient norm = 0.0000289 (50
iterations in 5.113s)
[t-SNE] Iteration 1650: error = 1.0784963, gradient norm = 0.0000282 (50
iterations in 5.179s)
[t-SNE] Iteration 1700: error = 1.0771290, gradient norm = 0.0000280 (50
iterations in 6.122s)
[t-SNE] Iteration 1750: error = 1.0758779, gradient norm = 0.0000272 (50
iterations in 6.062s)
[t-SNE] Iteration 1800: error = 1.0746903, gradient norm = 0.0000269 (50
iterations in 5.091s)
[t-SNE] Iteration 1850: error = 1.0736756, gradient norm = 0.0000275 (50
iterations in 5.101s)
[t-SNE] Iteration 1900: error = 1.0728042, gradient norm = 0.0000258 (50
iterations in 5.220s)
[t-SNE] Iteration 1950: error = 1.0719571, gradient norm = 0.0000257 (50
iterations in 5.211s)
[t-SNE] Iteration 2000: error = 1.0712305, gradient norm = 0.0000260 (50
iterations in 5.161s)
[t-SNE] KL divergence after 2000 iterations: 1.071231
```

[ ]:
```
df = pd.DataFrame({'x': tsne_2d_per70[ :,0], 'y': tsne_2d_per70[ :,1], 'label':
  ↪y})
sns.lmplot(data=df, x='x', y='y', hue='label', markers=['s','o'],␣
  ↪fit_reg=False, size=10)
plt.title("Perplexity {} and n_iter {}".format(70,2000))
plt.show()
```



[ ]:
```
tsne_3d_per70 = TSNE(n_components = 3, verbose=2, init='random', perplexity=␣
↪70, n_iter=2000, random_state=100).fit_transform(x)
tsne_3d = go.Scatter3d(x=df_sample[:,0], y=df_sample[:,1], z=df_sample[:,2])
```

[ ]:
```

```