



Data Science Fundamentals
(CS 890ES)

Instructor: Dr. Alireza Manashty
Department of Computer Science
University of Regina
Winter 2020

User Guide for Engineer
on
"Games Description Analysis & Keyword Suggestions"

Submitted by:

Name	ID	Email	Role
Sai Keerthi Mettu	200416252	smj102@uregina.ca	BI Analyst Data Engineer Data Scientist Communicator
Somi Deepthi Nalamalpu	200412879	sny899@uregina.ca	

USER GUIDE FOR ENGINEER

INTRODUCTION:

This user guide provides a detailed description and implementation guide to executing the project without any obstacles. The Game description analysis and keyword prediction as the title say facilitates providing its users with context-oriented keywords that enable a readable and lucid game description. In this project, we focused on analyzing the game description, obtaining important keywords from each game description and applying the machine learning model to identify the patterns between these keywords thereby grouping into a cluster. The user can provide a new game description and get specific important phrases that can be embedded in their game description to obtain a more contextual view of the description that is appropriate and concerning the game.

To attain the above-stated functionality, at first, the data scraped from the play store needs to be stored in Google Drive. We used Beautiful soup and play store scrapper to get the data from the play store.

- At first, using beautiful soup we scrapped the games from different genres using the subcategory section from the Google play store.

```
pip install google-play-scraper

Requirement already satisfied: google-play-scraper in /usr/local/lib/python3.6/dist-packages (0.0.2.0)

import requests
from bs4 import BeautifulSoup

import pandas as pd

from google_play_scraper import app

action_url='https://play.google.com/store/apps/category/GAME_TRIVIA'

# 'https://play.google.com/store/apps/category/GAME_STRATEGY'
# 'https://play.google.com/store/apps/category/GAME_SPORTS'
# 'https://play.google.com/store/apps/category/GAME_SIMULATION'
# 'https://play.google.com/store/apps/category/GAME_ROLE_PLAYING'
# 'https://play.google.com/store/apps/category/GAME_RACING'
# 'https://play.google.com/store/apps/category/GAME_PUZZLE'
# 'https://play.google.com/store/apps/category/GAME_MUSIC'
```

Once all the genres details were extracted, we requesting the application access for those games listed in them using,

```
app_page = requests.get(action_url)
```

```
# Provide the app page content to BeautifulSoup parser
soup = BeautifulSoup(app_page.content, 'html.parser')
```

Once the records were fetched by using the app id's automatically using the beautiful soup scrapper, we saved all the records fetched into a csv file.

- The data obtained is around 1200 records and we saved it as **originalgaming.csv** file.
- Then we used the app id's of existing data and scraped more games from the archived list using play store scraper.

```
[ ] def getSimilarAppRecords(simApps):
    rs=pd.DataFrame()
    id=[]
    for x in simApps:
        url=x['url']
        id=url.split("id=")[1]
        each_game_result=app(id)
        eachrow=add_details_as_row(each_game_result)
        # print(eachrow)
        rs = rs.append([eachrow])

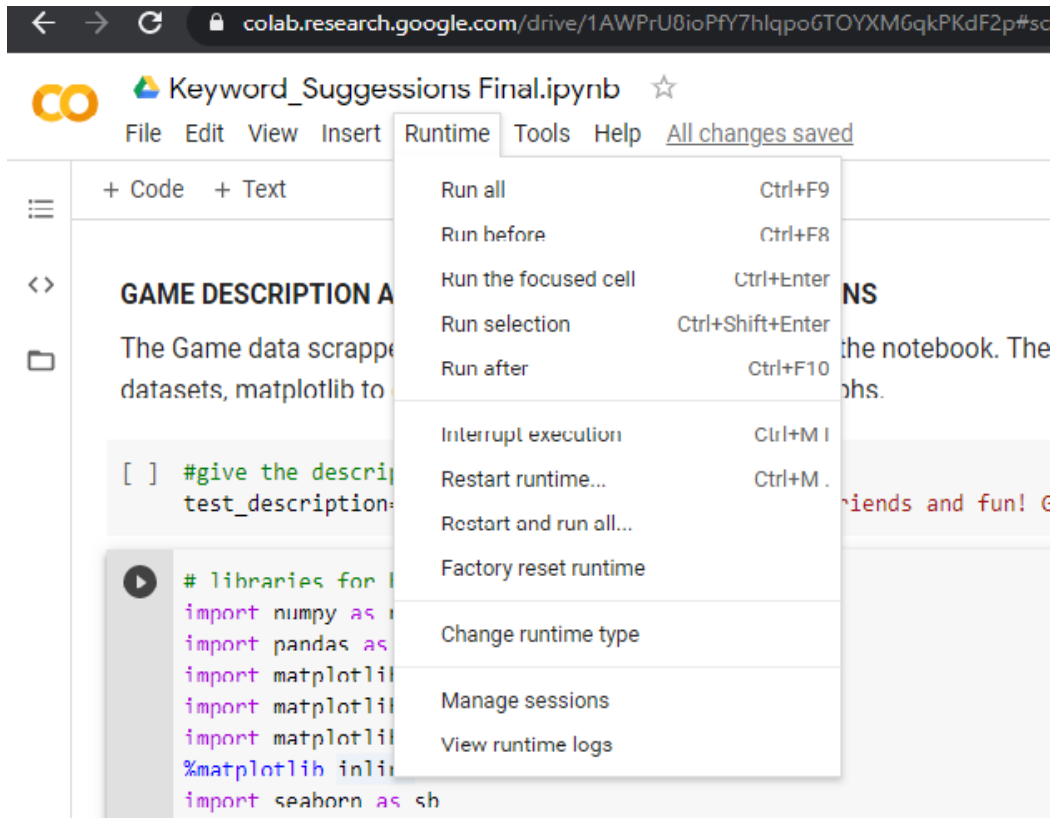
    return rs;
```

```
[ ] for i in PartAppId:
    try:
        similarApps= ps.similar(i,detailed=False)
        apprecs=getSimilarAppRecords(similarApps)
        recordsDF=recordsDF.append(apprecs)
    except(HTTPEror, RuntimeError) as e:
        print(e)
        print(i)
        pass
```

- The scraping and downloading of the data are done in batches and once we got the required data we merged all the batch files into a single one named **batchfile1.csv**.
- User needs to store those 2 datasets in their **Google Drive** to read the data and execute the model.
- Once the user stores the 2 datasets in his Google Drive, the further functionality like importing the data into **Google Colab IDE (Google Collaboratory)** and getting the data on the environment where are set up in the main implementation notebook called **Keyword_Suggestion Final.ipynb**.



The above image shows tools available in Google colab and it can be observed that there is a memory allotted to your notebook- as RAM and DISK spaces bars.



This runtime feature helps to execute your code in the IDE as a whole or after an intended code block. We can even start new run tie assign new memory of your environment.

The notebook has the implementation methods and functionality that can guide the user to reach the result. Important libraries like

- Pandas - to operate on data frames, data analysis and manipulation
- Numpy - for statistical operations
- Matplotlib, Seaborn - for data visualization

```
# libraries for basic operations and plots
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib as mpl
import matplotlib.cm as cm
%matplotlib inline
import seaborn as sb
```

- Scikit Learn - open source machine learning library, providing tools for data preparation, preprocessing, fitting, model selection, evaluation and etc features.
- KMeans for SKlearn - for implementing KMeans model
- Mini Batch KMeans - for analyzing the clusters.
- PCA and TSNE for elbow method and graph representation

```
[ ] #libraries used for modelling
    from sklearn.cluster import MiniBatchKMeans
    from sklearn.cluster import KMeans

    from sklearn.decomposition import PCA
    from sklearn.manifold import TSNE
```

- Countvectorizer - for the word to vector
- Tfidfvectorizer - to extract important keywords
- TF-IDF Transformer- to obtain the TF-IDF score representing the importance of a phrase in the text.

```
[ ] from sklearn.feature_extraction.text import CountVectorizer
    from sklearn.feature_extraction.text import TfidfVectorizer
    from sklearn.feature_extraction.text import TfidfTransformer
```

- NLP (natural language processing), NLTK (natural language took kit) - to operate and analyze the text data.

```
[ ] #libraries for NLP operations
import re
from bs4 import BeautifulSoup

import string
string.punctuation

import nltk
from nltk.tokenize import word_tokenize
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('words')
stopword=nltk.corpus.stopwords.words('english')
all_words = set(nltk.corpus.words.words())
```

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

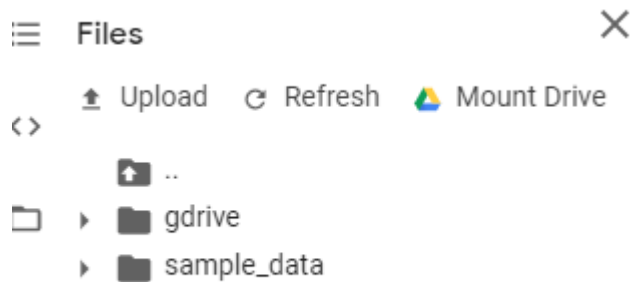
- Google Drive - to import data set from the Google drive and grant access permission to the colab.
-
- When importing drive access to colab, user need to execute the following commands and click on the url generated. The url pops up to your Google account and requests permission to access to account and drive to get all the requirements needed. A password will be generated once you provide the permission, which has to be copied and pasted in the empty box and press enter stating to give the permission.

```
from google.colab import drive
drive.mount('/content/gdrive/')
```

... Go to this URL in a browser: <https://accounts.google.com/o/oauth2/au>

Enter your authorization code:

Once executed you can find the gdrive mounted on to your runtime to save your working notebook and also get the essential files from it.



User need to execute following commands so that they can go to the right folder in their drive to get the datasets.

```
[7] cd /content/gdrive
```

```
/content/gdrive
```

```
[8] ls
```

```
'My Drive'/'
```

The above libraries would conveniently help the user to execute the methods and functions available in the notebook without any hassle. It is important to import the libraries in the correct format to avoid any missing functionality.

Moving on, the main notebook (**Keyword_Suggestion_Final.ipynb**) itself provides a detailed explanation of what the method does and why we are using that method at that particular point for the users to understand the context and operationality of the project. So following the notebook and the user guides can help in understanding and implementing the project without any difficulties.

The **Google Collaboratory (Google colab)** provides the user with enough runtime and the user can even switch to the GPU version from CPU to have fast processing. The runtime is initialized by the server when a user connects the colab with their Google Account. The work and executions done in a notebook will be stored in the **Google Drive** connected to the appropriate Google Accounts by the name of **Colab Notebooks**. Users can open and access his notebooks at any time and from anywhere and even work on them as Google colab provides the runtime and initializes the IDE as soon user connects to their **Google Account**.

```
[9] # /content/gdrive/My Drive/Batchfile1.csv  
df1=pd.read_csv('My Drive/Batchfile1.csv')  
df1.head()
```

The above commands are used to retrieve the data set Batchfile1.csv from the drive. The path provided may differ based on the folder the datasets where stored in your cases. Make sure to provide the correct path.

```
[10] df2=pd.read_csv('My Drive/originalgaming.csv')
     fea=df2.columns
     fea

Index(['Title', 'AppID', 'URL', 'Description', 'Summary', 'Installs',
      'MinInstalls', 'Score', 'Rating', 'Reviews', 'Price', 'Free',
      'Currency', 'OffersIAP', 'Size', 'Android Version', 'Developer',
      'Developer ID', 'Developer Email', 'Developer Website',
      'Privacy policy', 'DeveloperInternal ID', 'Genre', 'Genre ID', 'Icon',
      'Header Image', 'Content Rating', 'Content Rating description',
      'adSupported', 'ContainsAds', 'released date', 'Updated', 'Version',
      'Recent Changes', 'Comments'],
      dtype='object')
```

As the first dataframe, df1 doesnt posses column names, the feature names of the gamin d

```
[11] df1.columns=fea
```

The above command gets the second dataset originalgames.csv. In our datasets the batchfile1.csv is a merge of lots of batch files extracted which doesn't possess any feature names. so we are assigning the features names to the batchfile.csv by using the originalgame.csv features by directing to their data frames.

Using concat function we are merging the two datasets by using their respective data frames.

```
[13] rawset= pd.concat([df1,df2],ignore_index=True,sort=False)
```

Once the required libraries and data are imported on to the notebooks, the actual processing of the data is initiated by visualizing and cleaning the dataset. As the dataset is scraped from the website there might be duplicate data points that need to be removed. So anomalies, missing values along duplicate data points were removed and the dataset is prepared for visualization.


```
[17] data.dtypes
```

```
↳ Title           object
   AppID           object
   Description      object
   Genre           object
   Score           float64
   MinInstalls     float64
   Reviews         float64
   Free           object
   Currency        object
   Size           object
   Android Version object
   Genre ID        object
   Version         object
   dtype: object
```

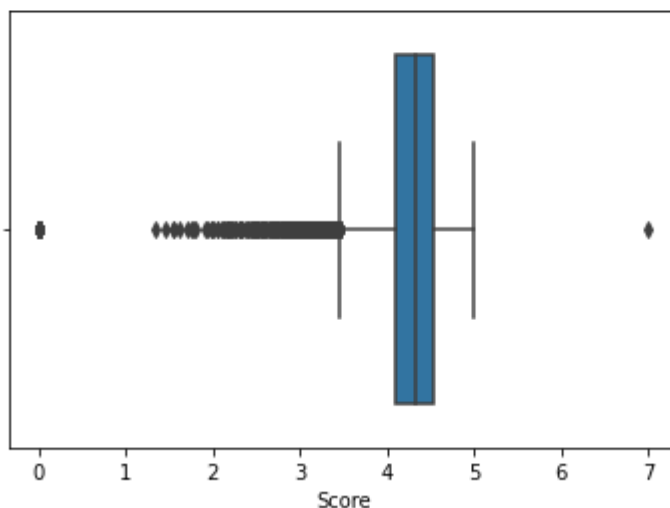
Once all the duplicates and essential features were gathered, the data available is displayed above along with their datatypes.

The visualization phase considers providing a semantic view of the data available and available relationships between the features of those data. Two or three features were drafted on a graph to view the effect of each feature on the other feature. In our project, we plotted a box graph on the rating score feature to check the variations and distribution of ratings over the games in the dataset. By doing that, we found out that there exists a game that has a rating of 7.0 which is not how the rating for a particular game is distributed. So we extracted the data that has ratings more than 5.0 and deleted them as the data it has may not be appropriate as the rating score. ... As said the data point with score =7.0 doesn't provide details about the game and game description.

```
[18] #checking the anamolies in Score column
```

```
sb.boxplot(x=data['Score'])
```

```
↳ <matplotlib.axes._subplots.AxesSubplot at 0x7f699017b160>
```



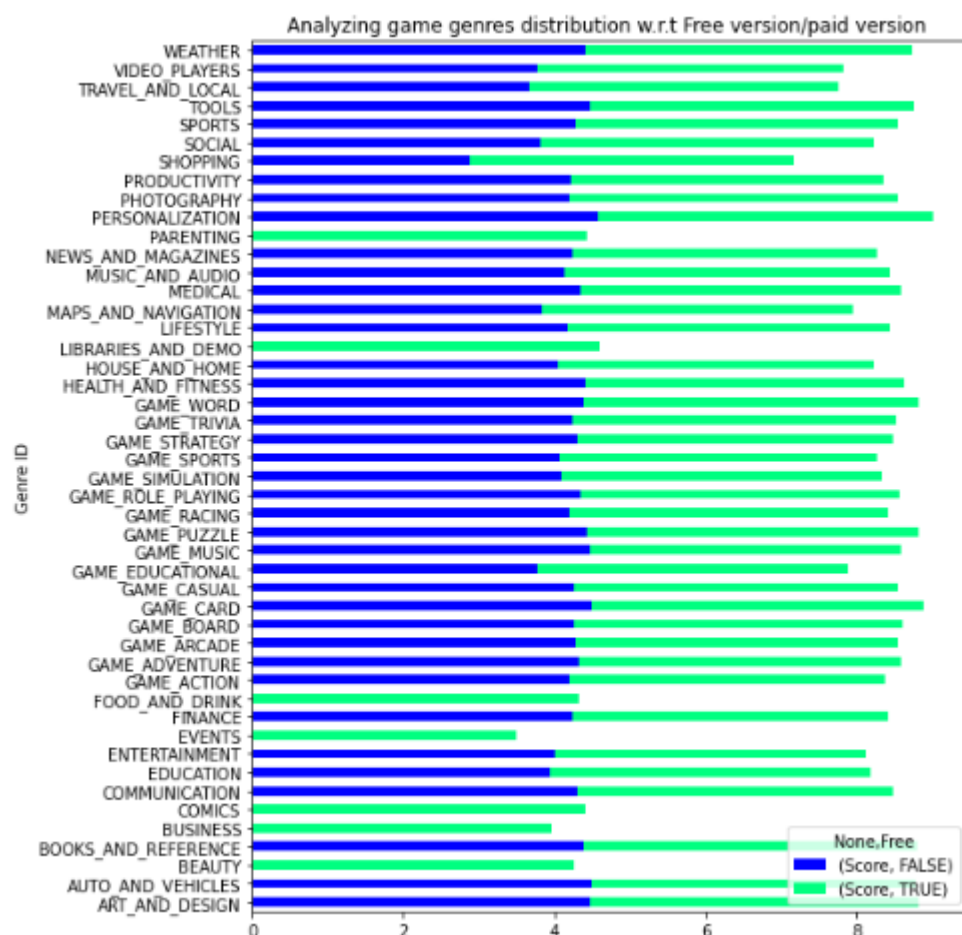
When the anomalies have been removed the distribution of score feature is meaningful and accurate

	Title	AppID	URL	Description	Summary	Installs	MinInstalls	Score	Rating	Reviews	Price	Free	Currency	OffersIAP	
11299		0	1	2	3	4	5	6.0	7.0	8.0	9.0	10.0	11	12	13

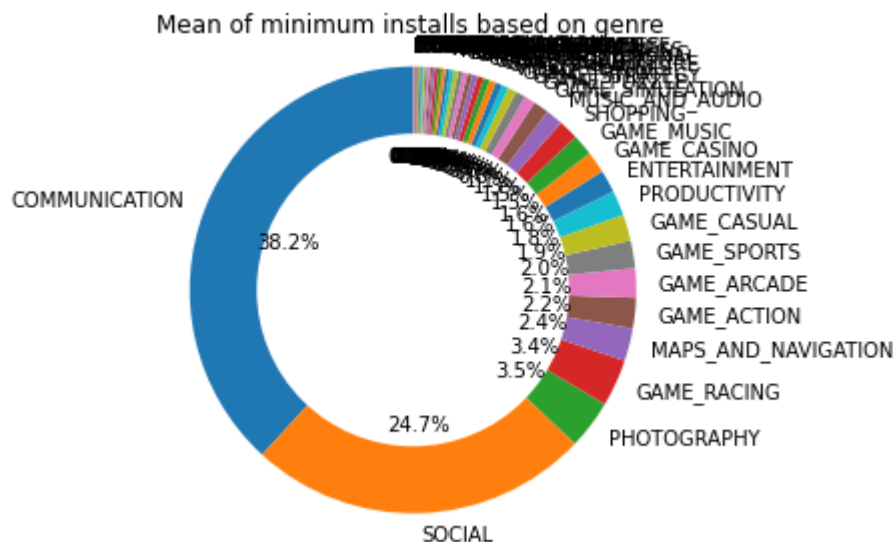
Moving on, the features were analyzed by plotting them with respect to other features as of the genre with respect to games to have a view on how many games are available as per the genre

```
{'GAME_ACTION': 866, 'GAME_SPORTS': 506, 'GAME_SIMULATION': 668, 'GAME_ROLE_PLAYING': 1122, 'GAME_RACING': 352, 'GAME_ADVENTURE': 783, 'GAME_
```

The games available as free versions and paid versions were checked and plotted. Similarly, the minimum installs of games with respect to their genre and other features were plotted



As the remaining features of the dataset were visualized,



Now it is the turn of game description to be structured and cleaned so that it can be fed to the model as training data.

- At first, the data is converted into lower case letters so that when it can be easy for the lemmatizer to group the data to its root when needed using the lower method.
- The two lettered words (we, if and other) were extracted along with the top ten most frequent words appearing in each game description. The descriptions were added to the counter for each game and the extracted words were removed as they might not be useful when we are looking for important phrases that determine the game. After removing the frequent words and converting to lower case the game description of each game looks as of below output-of first five data points.

```
0 welcome world sniper assassin. hold back achie...
1 duty is lead battle become best shooter sniper...
2 welcome shooting world, an incredible 3d fps m...
3 aim shoot! modern sniper is #1 first person sh...
4 time collect hunting gear, load gun, sharpen s...
Name: Description, dtype: object
```

An emoji in the text data can be observed which needs to be removed.

```
10 keep defense up shoot zombies one best first-p...
11 get ready for one best fps sniper shooting gam...
12 join most exciting multiplayer archery game ev...
13 join best online multiplayer shooting range ex...
14 like killing games? if so sniper 3d shooter is...
15 death race ® killer car shooting games, offici...
16 😊 ready roll dice on life crime? then city veg...
17 become #1 most lethal sniper world play downlo...
18 bike racing 3d is no.1 xtreme bmx game insane ...
19 it's about time take on role fearless modern s...
Name: Description, dtype: object
```

- The next step is to identify if any URLs, web addresses and emojis or emoticons were hidden in the text data. As we are focusing on phrases it would be easier if we remove those types of texts to improve the integrity of the text. To achieve that different methods were implemented and also along with the above methods, the top twenty rarely appeared words in each game description were also exempted.
- The clean description function is defined that takes a game description and removes punctuations in the text using string. punctuation method, divides the sentence into tokens, checks if the word is alphabetical and then pass on to obtain the stopwords. If punctuations are not removed the system might understand the words as we" ll as we and ll. And stop words are the words that are repeatedly used to form a meaningful English sentence like really, when and etc. These types of words are in no use to the problem we are addressing.

```
51] data['Cleaned_Text']=data['Description'].apply(lambda x:clean_description(x))
```

```
52] data['Cleaned_Text'][:10]
```

```
0    [welcome, world, sniper, assassin, hold, back,...
1    [duty, lead, battle, become, best, shooter, sn...
2    [welcome, shooting, world, incredible, 3d, mob...
3    [aim, shoot, modern, sniper, 1, first, person,...
4    [time, collect, hunting, gear, load, gun, shar...
5    [fight, honor, pick, cool, sniper, hand, exper...
6    [feel, action, survive, apocalypse, horror, on...
7    [get, ready, epic, sniper, 3d, free, shooting,...
8    [welcome, death, shooter, special, force, rece...
9    [modern, strike, cant, get, enough, action, lo...
Name: Cleaned_Text, dtype: object
```

- Once the clean description function is applied the data looks like the above output.
- The NLTK is imported to obtain the functionality of the lemmatizer using wordnet. lemmatize method. This method groups words into their root word based on their context and meaning. Lemmatizer always returns a word that belongs to the dictionary. By condensing similar context words to their root words, we can get better results of phrases without any phrases that have the same meaning. The model can be trained with valid data instead of a load of identical data.

```
[53] wn=nlk.WordNetLemmatizer()
      nltk.download('wordnet')
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data] Unzipping corpora/wordnet.zip.
True
```

```
[54] def lemmatizing(tokenized_text):
      text=[wn.lemmatize(word) for word in tokenized_text]
      return text
```


- The TF-IDF vectorizer is applied to the cleaned data to obtain important phrases from the game descriptions. The vectorizer fits and transforms the data as a feature and identifies all the important phrases while transforming them. The functionality of TF-IDF will be explained by the diagram below

Formula of TF-IDF vectorizer:

$$w_{i,j} = \text{tf}_{i,j} \times \log\left(\frac{N}{\text{df}_i}\right)$$

$\text{tf}_{i,j}$ = number of times i occurs in j divided by total number of terms in j
 df_i = number of documents containing i
 N = total number of documents

Example of the vectorizer:



$$\text{tf}_{\text{NLP},j} = \frac{\text{\# of occurrences of NLP}}{\text{number of words in text message}} = \frac{1}{3} = 0.\overline{33}$$

$$N = 20$$

$$\text{df}_{\text{NLP}} = 1$$

$$w_{i,j} = \text{tf}_{i,j} \times \log\left(\frac{N}{\text{df}_i}\right)$$

$$w_{i,j} = 0.\overline{33} \times \log\left(\frac{20}{1}\right)$$

$$w_{i,j} = 0.\overline{33} \times 1.301$$

$$w_{i,j} = 0.43$$

Implementation:

```
# analyzer=clean_description
tfidf_vect=TfidfVectorizer()
tf_counts=tfidf_vect.fit_transform(data['Cleaned'])
print(tf_counts.shape)
print(tfidf_vect.vocabulary_.keys())
```

(15133, 26350)

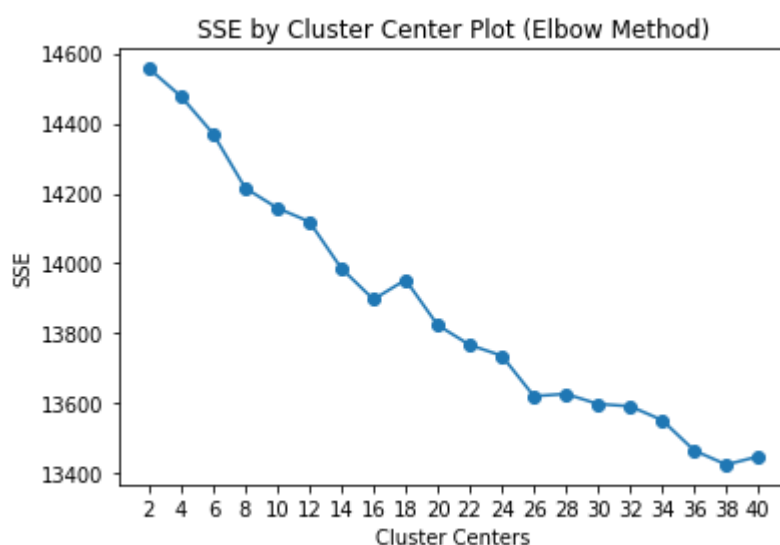
dict_keys(['welcome', 'world', 'sniper', 'assassin', 'hold', 'back', 'achieve', 'perfect', 'shot', 'target', 'play', 'amazing', ...])

Once it is achieved gathered keywords were fed to the KMeans clustering model for the model to be trained and identify the clusters based on the words vectorized by the TF-IDF vectorizer.

- At first, before getting the prediction the foremost step in the KMeans algorithm is to get a valid number of clusters. Clusters are formed by recognizing the groups of items that tend to be similar.

```
sse = []
for k in range(2, 40):
    sse.append(MiniBatchKMeans(n_clusters=k, init_size=1024, batch_size=2048, random_state=20).fit(data).inertia_)
print('Fit {} clusters'.format(k))
```

- The above image takes the data fit it with respect to the model and obtains the optimal number of cluster groups based on the clusters count given.
- To visualize the clusters formed elbow method is used which plots the SSE (The centroid of each cluster and distance between the data points from the cluster is measured using SSE-sum of squared error) and clusters count generated. The elbow method is applied to see if the data is compatible enough and provides a deep curve or not which helps in finalizing the number of clusters to be taken for the data available.



- As represented above we considered 16 clusters and the KMeans model is applied to the data as of training it with 100 iterations. The KMeans algorithm iterates until the

cluster stops changing or max iterations occurred. Each time cluster keeps changing and when the model recognizes the pattern with respect to those training data it stops and the cluster remains intact.

```
true_k = 16
k_model = KMeans(n_clusters=true_k, init='k-means++', max_iter=100, n_init=1)
%time k_model.fit(tf_counts)
```

```
↳ CPU times: user 48.5 s, sys: 4.1 s, total: 52.6 s
Wall time: 47.8 s
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=100,
       n_clusters=16, n_init=1, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=0.0001, verbose=0)
```

- We are gathering the clusters and the phrases available in those clusters so that we can test the model with unseen data and check, in which cluster the test data falls into and what are the phrases that the model has predicted that can be added to the test game description.

```
order_centroids = k_model.cluster_centers_.argsort()[:, :-1]
terms = tfidf_vect.get_feature_names()
```

```
def display_top_keywords(true_k, words_count):
    for i in range(true_k):
        print("\nCluster %d: " % i),
        for indexword in order_centroids[i, :words_count]:
            print("%s" % terms[indexword])
```

- The above function gets the keywords that are predicted by the model which are not in the user given game description. As there may exist certain words that the model predicted to be available in the test description already. We are ignoring those phrases and providing the words that can be added to the description from among the predictions.

```
#give the description that has to be predicted
test_description="Escape to the world of farming, friends and fun! Go on farm adventures to collect rare goods and craft new recipes. Raise animals a
```

For the above test game description (different example) we get the prediction as below

```

x,y,z,w=get_cluster_predictions(test_description2)
print(x)
print(y)
print(z)
print(w)

```

```

[15]
['make', 'time', 'use']
['game', 'new', 'play', 'get', 'world', 'free', 'fun', 'help', 'like', 'one', 'find', 'city', 'different', 'best', '
['learning', 'learn', 'educational', 'child', 'coloring', 'math', 'color', 'fun', 'alphabet', 'baby']

```

- The above picture shows it clearly that the test game description falls in the 13th cluster predicted by the model. Below are the phrases predicted by the model those are already present in the test description, obtained by actually subtracting the list of words from the top keywords by the words that are already present in the given description.

```

## list1 - list2
suggest_words=[i for i in test_tokens + cl_keywords if i not in test_tokens ]
suggest_words

```

```

['level', 'character', 'adventure', 'story', 'get', 'friend']

```

The next set shows the new words that can be added to the test description to improve it. And the next set is the words that shouldn't be attached to this particular test description as these words are not relevant to the game context in accordance with the model.

- The most, unlike words, was gathered by taking the top rare or words in the same cluster that suffices that these words may be relevant to the test data given.

```

def display_cluster_rare_words(cluster_number,words_count):
    cluster_keywords=[]
    rare_words_centroids=order_centroids[::-1] # reverse the array

```

- And then, all the results accumulated were displayed by calling the function of **all suggestions**. This is done for the user to check or grasp all the results that are needed for their game description at an instance at a single place.


```
def all_suggestions(text):
    a,b,c,d=get_cluster_predictions(text)
    e,f=get_tf_idf_predictions(text)
    g,h=get_genre_based_predictions(text)
    print("Predicted Cluster : ",a)
    print("\nWords that are must to be kept in description : ",b)
    print("\nWords to be Added (Suggestion from Clustering model) :",c)
    print("\nWords not to be Added (Suggestion from Clustering model) :",d)
    print("\nTF-IDF score determinations from trained Game Descriptions")
    #KEYWORDS that are likely to be expected from this given type of game
    print("Keywords in the given description that are more likely interested to keep: \n")
    print_function(e)
    #KEYWORDS that are likely to be expected from this given type of game
    print("\nKeywords in the given description that are not very likely interested to keep: \n")
    print_function(f)

    print("\nTF-IDF score determinations from trained Game Descriptions+Genre+Title+Score/Rating")
    print("Keywords in the given description that are more likely interested to keep based on the Genre and Score Ratings: \n")
    print_function(g)
    print("\nKeywords in the given description that are not very likely interested to keep based on the Genre and Score Ratings:")
    print_function(h)
```

The results were represented or printed as:

```
all_suggestions(test_description)

Predicted Cluster : [6]

Words that are must to be kept in description : ['game', 'play', 'free', 'fun', 'new']

Words to be Added (Suggestion from Clustering model) : ['learn', 'learning', 'music', 'color', 'time', 'use', 'help', 'make', 'application',

Words not to be Added (Suggestion from Clustering model) : ['zombie', 'shooting', 'sniper', 'dead', 'game', 'survival', 'shooter', 'gun', 'a

TF-IDF score determinations from trained Game Descriptions
Keywords in the given description that are more likely interested to keep:

friends: 46.6
farm: 40.7
farming: 32.5
anytime: 26.2
animals: 26.2
co: 25.0
adventures: 22.4
anonymous: 19.9
raise: 13.9
play: 13.9
connected: 13.8
trade: 13.3

Keywords in the given description that are not very likely interested to keep:

game: 7.5
```

It can be noticed that there exist the results of most likely words to be kept and the most unlikely words that can be removed, are also displayed.

- The TF-IDF transformer is used to obtain this functionality. The cleaned text is passed to the TF-IDF transformer and the features were extracted which is a bit less than the features generated by the TF-IDF vectorizer.

```
#applying Tf-Idf for the acquired words
tfidf_transformer=TfidfTransformer(smooth_idf=True,use_idf=True)
tfidf_transformer.fit(word_count_vector)
```

```
TfidfTransformer(norm='l2', smooth_idf=True, sublinear_tf=False, use_idf=True)
```

```
feature_names=cv.get_feature_names()
len(cv.vocabulary_.keys())
```

26257

- A score is allotted for each word that is provided as training data and its importance is recognized by allotting a percentage value to it.
- If the score is higher than that phrase is most likely to be kept and in the same way, if the score is lower than it can be deleted or changed.
- The most unlikely words to be kept in the description were extracted by taking out the least scored phrases from the bottom of the feature list generated by the TF-IDF transformer when applied on a test data.

```
X=data['Cleaned']+str(data['Score'])+data['Titlle']+str(data['Genre ID'])

cv2=CountVectorizer(max_df=0.75,stop_words=stopword)
word_count_vector2=cv2.fit_transform(X)

tfidf_transformer2=TfidfTransformer(smooth_idf=True,use_idf=True)
tfidf_transformer2.fit(word_count_vector2)
all_features=cv2.get_feature_names()
len(cv2.vocabulary_.keys())
```

39836

There is a increase in the quantity of features obtained when genre, score and title were embedded and TF-IDF transformer is applied.

- Other features like genre, rating score were added along with cleaned text to check the functionality of the TF-IDF transformer and compare its results with those ones obtained (when only cleaned description was given) for the score comparison.

TF-IDF score determinations from trained Game Descriptions

Keywords in the given description that are more likely interested to keep:

friends: 46.6
farm: 40.7
farming: 32.5
anytime: 26.2
animals: 26.2
co: 25.0
adventures: 22.4
anonymous: 19.9
raise: 13.9
play: 13.9
connected: 13.8
trade: 13.3

Keywords in the given description that are not very likely interested to keep:

even: 7.5
mode: 7.7
collect: 7.7
go: 7.9
join: 8.1
share: 9.1
popular: 10.1
anywhere: 10.2

TF-IDF score determinations from trained Game Descriptions+Genre+Title+Score/Rating

Keywords in the given description that are more likely interested to keep based on the Genre and Score Ratings:

farm: 38.6
friends: 36.6
farming: 30.7
op: 26.6
internet: 24.8
anytime: 24.8
recipes: 23.7
co: 23.3
animals: 21.2
adventures: 19.6
anonymous: 18.8
raise: 13.2
play: 13.1
connected: 13.1
trade: 12.6

Keywords in the given description that are not very likely interested to keep based on the Genre and Score Ratings:

even: 7.1
collect: 7.3
mode: 7.3
go: 7.5
join: 7.7
share: 8.7
popular: 9.6
anywhere: 9.7

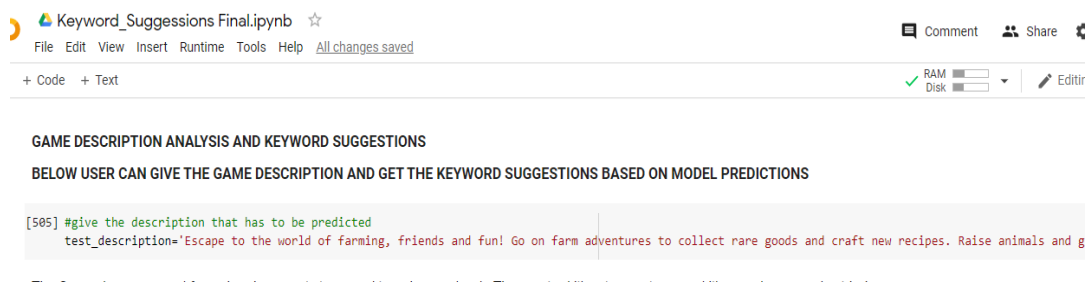
As we are working to deploy the whole analysis and its prediction operation on the web to allow it to interact with the users globally, these predictions (like the words that can be added to the description, words that should be kept without changing based on genre and rating score) will give good results when the user provides his game description, the genre of their game, title and ratings obtained to it. The predictions and suggestions obtained can be added

to their existing description to improve its readability over the game stores. It can even help the new developers to get some ideas by looking at these phrases as, how they can improve their game description before launching it over the internet so that it can be reachable to more number of audiences.

A user can execute the whole project by understanding the whole working aspects of the model implemented and executing it in the way mentioned above. The document provides the whole working and implementation scenario from scratch and the requirements and libraries needed were also listed above for user convenience.

Below are the steps to be performed to implement the project and obtain results without concerning the methodology

- We added the execution code, for the user to provide their game description, at the top of the notebook.



```
[505] #give the description that has to be predicted
      test_description='Escape to the world of farming, friends and fun! Go on farm adventures to collect rare goods and craft new recipes. Raise animals and g
```

- User can give the description use the option runtime on the top tool bar.
- select the "run all" option and wait for the all code blocks to get executed.
- Once executed, user can find the predictions given by the model at the bottom of the notebook. All the outputs/results gathered and displayed at a time.
- This can help users that doesn't concern on going through code block by block to check the result, for those who just want the prediction to analyze his/her description ignoring all the programming stuff.

OVERALL SETUP STEPS:

STEPS TO BE PERFORMED:

- User needs to download the Gaming.csv and BatchFile1.csv and store it their Google Drive.
- The Keyword suggestions.ipynb available in the GitHub can be opened with Colab (Google Collaboratory).
- The user needs to sign in his Google account to get the runtime and working environment in Google colab or create an account if it doesn't exist.
- Once the Keyword suggestions.ipynb is imported on to your colab environment allot the runtime and check if the drive path is correct. The path specified should be the same as the one where the dataset is store. Make sure it is accurate.

- There exist a text description variable where user can give their game description within the inverted commas. The variable takes the user input and provides it to the model as test data.
- Then the user can execute the code by opting for "Run ALL" at runtime to attain the suggestions ignoring the remaining methodology or can run each cell to view and analyze the output of each technique and operation performed on the data.

Git Hub Link to access the project files:

<https://github.com/KeerthiMettu/Games-Analysis>

<https://github.com/somideepthi/Game-description-analysis-and-keyword-suggestion>

Access links for our 2 datasets from Google drive:

1. BatchFile1.csv:
<https://drive.google.com/open?id=18cRi8kB54mGyih2ufc1pPr6hHvxPDbJT>
2. originalgaming.csv:
<https://drive.google.com/file/d/10-E7sLMq1P2E8Ubv0IrlCugyXrsyfoHw/view>