Google Play Store: User Story 99 Report

User would like to see which apps are similar by rating.

KenzieM

1. Introduction

For this report, I used data on apps from the Google Play Store to answer the user story: “User would like to see which apps are similar by rating.” Specifically, the user wanted a clustering algorithm performed that can show similar apps rated between 3.8 and 4.8. The data contains 10,841 rows and 13 columns. The columns of this data set include information such as name, category, genre, rating, size, number of installations, price, content rating and more. Every row of the set represents an individual application available on the Google Play Store. In the following report, I will discuss how I transformed the data in order to answer the question, and provide some visual aids to better understand that answer.

1. Data

As stated in the introduction, I worked with one CSV from the Google Play Store which contained 10,841 rows and 13 columns. I used Python’s Pandas library to parse the csv into a dataframe, from which I could inspect the rows and columns for irregularities. First, I checked the data for duplicate rows, of which there were about 1,500, and removed these and reset the indices. Next I inspected the columns for nulls and incorrect data types and found quite a few. Specifically four columns containing numerical information that were still listed as string types, one column with many null values, and several columns with only 1. After drilling down into the columns with only one null value, I found a row that had several entry errors and decided to delete this row outright since the dataset is relatively large. Of the columns with incorrect data types, the Reviews column was the most straightforward to fix. Two columns, however, contained symbols like commas, plus signs, and dollar signs, which make a direct type conversion a little trickier. I used the following code to remove the dollar signs from the price column:

|  |
| --- |
| df['Price'] = df['Price'].apply(lambda x: x.replace('$', '')) |

The column containing information about the size of the application was the most difficult to fix, as it had two different units, k for kilobytes and M for Megabytes. To do this, I wrote the following function, which removes the symbol and converts the data type to a numeric, before applying the correct multiplication:

|  |
| --- |
| def new\_size(x):  if 'k' in x:  return (float(x.replace('k', ''))) \* 1000  if 'M' in x:  return (float(x.replace('M', ''))) \* 1000000   df['Size'] = df['Size'].apply(new\_size) |

Once I completed the data type conversions, I saved the final cleaned data frame into a new CSV. I recognize that most of this cleaning was not necessary for this user story in particular, but I wanted to have a clean csv to work with on all future stories.

For this project, I only needed the information about category and rating to feed into the machine learning model, but I wanted to find similar apps, so I created a dataframe with only the columns containing names, rating and category. I then did a search for null values and found 1450 null values in the ratings column. I did not want to delete this many rows of data, so I filled in those values with the mean. Next, I selected only rows whose rating scored between 3.8 and 4.8. I stored this dataframe in a variable to save for the end so I could append the cluster labels from each machine learning model used. Then, I created a new data frame with a different name for just the features, and applied one-hot encoding for the categories. One-hot encoding is a way I can convert categorical data to numerical data by creating a column for each category and representing an apps’ membership to that category with the value of 1. I then used a min-max scaling to bring all the ratings to a scale of 0 to 1 to reflect the same scale of the categorical values. After scaling, I had the data ready for machine learning

1. Method

I chose this story because it provided a significant challenge for my current skill set, so I had to do a lot of research on clustering algorithms before starting. It’s a brave new world out there with regards to data science and machine learning and I got mixed answers as to what the best algorithm to use for my data would be. I decided to focus on three different clustering algorithms and use the silhouette score from these algorithms to judge how well they fit the data. I chose KMeans, DBSCAN and Meanshift. I chose KMeans because it is one of the most commonly used and has a lot of documentation online, but it perhaps isn’t the best choice for categorical data. I chose DBSCAN and Meanshift because these algorithms do not require an initial setting of the number of clusters, and the data had many dimensions due to the one-hot encoding that made visualization and prediction of clusters difficult. I chose the silhouette score as my measure because I was working with unlabeled data.

To get an initial look at the performance of these algorithms, I constructed a helper function that would run the algorithm and return the silhouette score, number of clusters, and amount of noise produced by that model. Then, for each algorithm I built another function that would run it with the given hyperparameters and feed that information to the helper function.Below, I’ve included a copy of the general helper function and the builder function for kmeans.

|  |
| --- |
| def build\_model(clustering\_model, data):  model = clustering\_model(data)  n\_noise = list(model.labels\_).count(-1)  print('silhouette score: ',   metrics.silhouette\_score(data, model.labels\_))  print('n\_clusters: ', model.labels\_.max() + 1)  print('n\_noise: ', n\_noise) |

|  |
| --- |
| def k\_means(data, n\_clusters=40, max\_iter=1000):  model = KMeans(n\_clusters=n\_clusters, max\_iter=max\_iter).fit(data)  return model |

I used these functions to play around with some different hyperparameters and see what kinds of silhouette scores I received. The models had silhouette scores ranging from .7 to .8 which indicates a well fitting model. My data is bunched rather close together however, so this shouldn’t have surprised me.

The next step in this process was to use a more precise method to tune my hyperparameters. I struggled to find resources that could provide me with a way to set my initial parameter ranges, but luckily one of the examples used in my research used the iris data set, which has values not too far away from the ones in my data set. I decided to use those parameters as a starting place and then use some tuning techniques to hone in on the best ones. First, I used the ParameterGrid method from scikit learn and the following code to create a list of parameter dictionaries.

|  |
| --- |
| parameters = {'n\_clusters' : [20, 25, 30, 35, 40, 45]} parameter\_grid = ParameterGrid(parameters) list(parameter\_grid) |

Output:

|  |
| --- |
| [{'n\_clusters': 20},  {'n\_clusters': 25},  {'n\_clusters': 30},  {'n\_clusters': 35},  {'n\_clusters': 40},  {'n\_clusters': 45}] |

I created another helper function (shown below) that would take a list of parameters, run the model for each one, and then print the parameter along with the silhouette score, number of clusters and amount of noise.

|  |
| --- |
| best\_score = -1 model = KMeans() for g in parameter\_grid:  model.set\_params(\*\*g)  model.fit(cluster\_df)  ss = metrics.silhouette\_score(cluster\_df, model.labels\_)  print('Parameter: ', g, 'Score: ', ss)  if ss > best\_score:  best\_score = ss  best\_grid = g |

Output:

|  |
| --- |
| Parameter: {'n\_clusters': 20} Score: 0.7146894784906 Parameter: {'n\_clusters': 25} Score: 0.7724299504896364 Parameter: {'n\_clusters': 30} Score: 0.8045479274935803 Parameter: {'n\_clusters': 35} Score: 0.7566386774660472 Parameter: {'n\_clusters': 40} Score: 0.7261422306661404 Parameter: {'n\_clusters': 45} Score: 0.7039987734312707 |

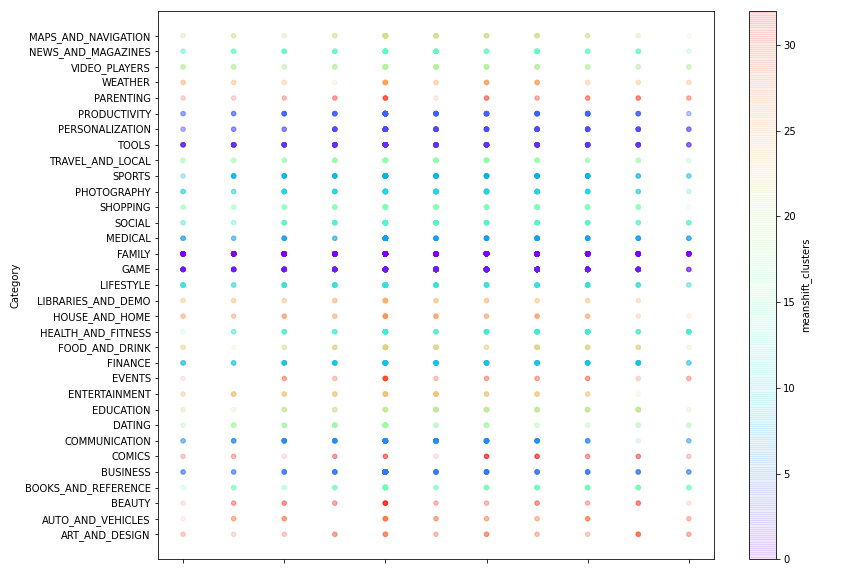
Next, I chose the parameters with the best silhouette scores and appended the assigned labels back to the data frame I saved with the app names still attached. I attempted to make some graphs plotting the app rating by original category and using colors to differentiate the assigned clusters, but I had some trouble with this, and I was running out of time. The final data frame can be referenced to see which apps were clustered together by the different algorithms

1. Results

Taking a look at the final data frame by using: final\_cluster\_df.sample(50).sort\_values('kmeans\_clusters')

I can see that each algorithm clustered the apps almost the same with some slight variations. The silhouette scores from DBSCAN and meanshift were the same at about 0.820, and the best kmeans score I got was slightly lower at 0.768. These silhouette scores indicate good clustering, meaning points within a cluster are close together and the points between clusters are farther apart. 32 clusters produced the best kmeans silhouette score and 33 clusters were labeled by the meanshift and DBSCAN. This is basically the amount of categories present (33), and the labels correspond almost exactly to the category label. From what I understand, this tells me that the greatest similarity of these apps with regards to category and rating, is the category. This also tells me that the models either do a great job with the data because they work well, or that the model may need more columns of data to find other meaningful groupings.

I struggled with the best way to present this information. I created a groupby object that can be queried by cluster label to find all the apps in that cluster. Perhaps I could create an interactive widget that will produce the correct dataframe based on the type of clustering and the label. It would also be great to rename the cluster labels such that they are shared by the different algorithms (where appropriate). Visualizing this data also proved tricky for me. I couldn’t get seaborn or matplotlib to produce any jitter for me, so it’s difficult to really see the density of points in the graph. Here’s an ugly graph I made while struggling to get matplotlib to cooperate and the x-labels are cut off (they should read 3.8-4.8) :D



1. Discussion

If I had more time to work on this story, there are several things I would either try to do differently or improve upon. The first thing I would try is running the algorithms without normalizing the data. The scales of the data were not drastically different, and in some of the examples I learned from, the instructor did not scale data that was on much different scales than mine. This could have been for ease of instruction, but I also believe that the category distinctions ended up too heavily weighted based on how they were originally categorized to give the rating differences any meaningful weight.

Another avenue for exploration would be trying different distance measurements. Using one-hot encoding for categorical columns meant I had data that was all very close together, but in a very high-dimensional space. The theory behind these different measurements is a bit over my head, but I think it would be interesting to create an additional distance parameter in the parameter grid and see what results are found with different types of distance measures. The default Euclidean distance was used for all these trials.

Finally, I think adding in some additional columns from the data set could be used to cluster the data. For example, using the number of installs and the price could reveal interesting groupings. The algorithms I used really only told me what category an app belonged to, and unless we want to use the algorithm for prediction, it doesn’t tell us much else. I also struggled with some of my data presentation such as visuals. I dedicated a lot of my time on this story learning about clustering algorithms and how to prepare data for machine learning, but I still have a lot to learn in regards to visualizing large datasets.

1. Conclusion

This was a really interesting project for me to take on, and I grew my skill set in the process. I have a lot more to learn regarding machine learning, but this has opened me to many opportunities for further exploration. With more time and research, I could provide some useful groupings of the apps, either for prediction or recommendation.