Modern Data Analysis HomeWork 2020

Edgar Zakharyan, Jacob Malyshev, Fedor Zhukov, Ivan Suchkov

1 Home work 1: Report writing

A dataset used in our work is a subset with 900 rows of one which contains 7000 Apple iOS mobile application details. The data was extracted from the iTunes Search API at the Apple Inc website. Data collection date (from API) is July 2017. This dataset was chosen because of several reasons. First of all, mobile apps market is one of the fastest growing markets in IT and iOS holds about 43% of it. So, with million of apps around nowadays, the following dataset has a huge value, as well as, results of it's analysis with machine learning algorithms.

Here is a content of this dataset:

- *size_bytes*: Size (in Bytes) of app
- *currency*: Currency Type
- *price*: Price amount
- *user_rating* : Average User Rating value (for all version). There are ratings in App Store
- *cont_rating*: Content Rating (4+, 12+, 16+,. . . .)
- *prime_genre*: Primary Genre (Games, Finance, Travel,. . .)
- *sup_devices.num*: Number of supporting devices
- *lang.num*: Number of supported languages

Source address: `https://www.kaggle.com/ramamet4/app-store-apple-data-set-10k-apps`

Examples of problems:

- Finding out how does the App details contribute the user ratings
- Comparing app statistics for different groups in dataset

2 Home Work Assignment 2: K-means

It is very valuable to know how parameters of those apps coincide with each other. To find out what kind of clusters there exist, we are going to use partitional clustering method: K-Means.

We applied K-Means with 10 random initializations with 5 and 9 clusters

For K-Means optimization, we used this cost-function, that k-means is minimizing:

$\frac{1}{m}\sum_{i=1}^{m}||x^{(i)} - \mu_{c^{(i)}}||^2$

where $c^{(i)}$ is index of cluster to which example $x^{(i)}$ is currently assigned. μ_k is a cluster centroid k, $\mu_{c^{(i)}}$ is cluster centroid of cluster to which example $x^{(i)}$ has been assigned.

We have chosen these 4 parameters for K-means clusterization:

- *size_bytes*: Size (in Bytes) of app
- *user_rating* : Average User Rating value (for all version). There are ratings in App Store
- *sup_devices.num*: Number of supporting devices
- *lang.num*: Number of supported languages

Because, using pairplots with colorization according to primary genre showed that these parameters play a big role in separating apps into some types, genres and subgenres.

Our Kmeans algorithms from *sklearn.cluster* was constructed with those arguments:

- n_clusters
- init
- max_iter
- random_state

For dataset standardtization (centring and normalizing) we used *sklearn.preprocessing StandardScaler()* function.

The unexplained part of our data's scatter is in inertia_ attribute of Kmeans algorithm. It is used in code to identify the best partition.

Here is a code on Python3 which brings us centroids and clusters labels over them:

```
def apply_kmeans(x_org, n_clusters, n_repeats):


    tmp_inertia = 0
    clusters, best_clusters = {}, {}
    indices, best_indices = {}, {}
    cluster_means, best_cluster_means = {}, {}
```

```
    differences, best_differences = {}, {}
    rel_differences, best_rel_differences = {}, {}
    inertia, best_inertia = {}, {}
    g_mean = np.mean(x_org, axis=0)

    x_zscr_std = scaler.fit_transform(x_org)

    for i in range(n_repeats):
        clusters[i] = {}
        cluster_means[i] = {}
        differences[i] = {}
        rel_differences[i] = {}
        indices[i] = {}
        inertia[i] = {}

        # instantiate KMeans Alg. object
        km = KMeans(n_clusters=n_clusters, init='random', n_init=10, max_iter
            =500,
                    tol=1e-4, random_state=i, algorithm='full', )  # verbose
                        =1
        km.fit(x_zscr_std)  # Compute k-means by calling fit method

        # Store the computation results per each initilization
        for k in range(n_clusters):
            clusters[i][k] = x_org[np.where(km.labels_==k)]
            indices[i][k] = np.where(km.labels_==k)[0]

        inertia[i] = km.inertia_

        for k in range(n_clusters):
            cluster_means[i][k] = np.mean(clusters[i][k], axis=0)
            differences[i][k] = np.subtract(cluster_means[i][k], g_mean)
            rel_differences[i][k] = 100*(np.divide(
                np.subtract(cluster_means[i][k], g_mean), g_mean)
                                         )
        # to chose the best clustering results regarding the inertia
        if i == 0 :
            tmp_inertia = km.inertia_
            delta = 0
        if i != 0:
            delta = tmp_inertia - km.inertia_
        if delta >= 0:
            tmp_inertia = km.inertia_
            for k in range(n_clusters):
                best_clusters[k] = x_org[np.where(km.labels_==k)]
                best_indices[k] = np.where(km.labels_==k)[0]
            for k in range(n_clusters):
                best_cluster_means[k] = np.mean(best_clusters[k], axis=0)
                best_differences[k] = np.subtract(best_cluster_means[k],
                    g_mean)
                best_rel_differences[k] = 100*(np.divide(
                    np.subtract(best_cluster_means[k], g_mean), g_mean)
                                               )
            best_inertia = km.inertia_

    return  clusters, best_clusters, indices, \
best_indices, cluster_means, best_cluster_means, \
differences, rel_differences, best_differences, \
best_rel_differences , inertia, best_inertia

n_clusters_1 =9

clusters_1, best_clusters_1, indices_1, best_indices_1,\
cluster_means_1, best_cluster_means_1,\
differences_1, rel_differences_1, \
best_differences_1, best_rel_differences_1, \
inertia_1, best_intertia_1 = apply_kmeans(x_org=df_hw1_scaled,
                                          n_clusters=n_clusters_1,
```

```
                                                    n_repeats=10)
```

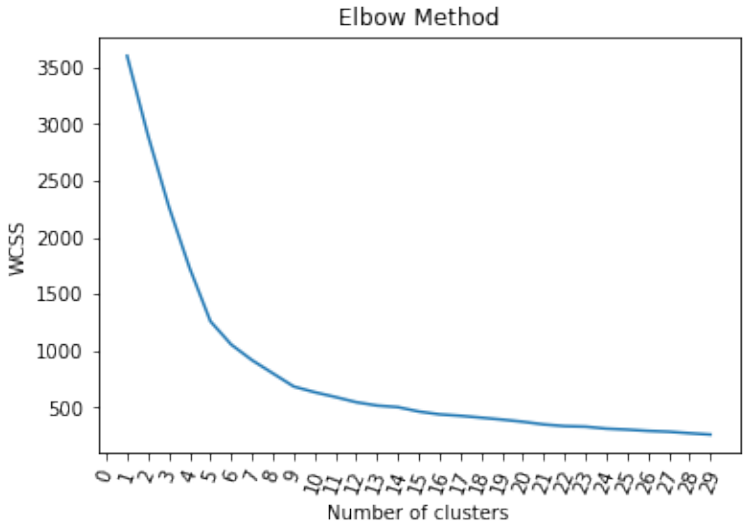
And here is a code to get table with best clusterization results:

```
# creating tables with results in latex format
def demonstrate_best_results(x, features, clusters, indices,
                             cluster_means, differences, rel_differences):
    dataframes = []
    for cluster, result in clusters.items():
        print( )
        print("cluster number  : " + str(cluster+1),
              "Number of el.", len(indices[cluster] ))
        df = pd.DataFrame({"grand mean      : ": np.mean(x, axis=0),
                           "cluster mean    : ": cluster_means[cluster],
                           "differences     : ": differences[cluster],
                           "rel. differences: ": rel_differences[cluster]})
        new_data = ["size_bytes", "user_rating", "sup_devices.num", "lang.num
            "]
        df.insert(0, "feature ", new_data)
        df.insert(0, "cluster: " + str(cluster+1), "")
        dataframes.append(pd.DataFrame(df))

    return dataframes

dataframes = demonstrate_best_results(x=df_hw1.to_numpy(), features=df_hw1.
    columns,
                          clusters=best_clusters_1,
                          indices=best_indices_1,
                          cluster_means=best_cluster_means_1,
                          differences=best_differences_1,
                          rel_differences=best_rel_differences_1)
latex_tables = []
for i in range(len(dataframes)):
    latex_tables.append(dataframes[i].to_latex(index=False))
```

Elbow method shows us, that the best clusterization results might be achieved with 5 and 9 clusters:



For elbow method we graph the relationship between the number of clusters and Within Cluster Sum of Squares (WCSS) then we select the number of

clusters where the change in WCSS begins to level off (elbow method). WCSS is defined as the sum of the squared distance between each member of the cluster and its centroid.

The main parameter for our clusterization result's interpretation was **the relative difference**.

Here is a table with values for 5 clusters:

cluster: 1, N.el.:553	feature	grand mean :	cluster mean :	differences :	rel. differences:
	size_bytes	1.148171e+08	9.458228e+07	-2.023484e+07	-17.623538
	user_rating	3.859444e+00	4.163653e+00	3.042084e-01	7.882180
	sup_devices.num	3.688889e+01	3.849186e+01	1.602974e+00	4.345411
	lang.num	6.594444e+00	3.674503e+00	-2.919942e+00	-44.278813
cluster: 2, N.el.:127	feature	grand mean :	cluster mean :	differences :	rel. differences:
	size_bytes	1.148171e+08	7.231301e+07	-4.250410e+07	-37.018963
	user_rating	3.859444e+00	2.307087e+00	-1.552358e+00	-40.222313
	sup_devices.num	3.688889e+01	3.796850e+01	1.079615e+00	2.926667
	lang.num	6.594444e+00	2.724409e+00	-3.870035e+00	-58.686293
cluster: 3, N.el.:115	feature	grand mean :	cluster mean :	differences :	rel. differences:
	size_bytes	1.148171e+08	1.098441e+08	-4.973061e+06	-4.331289
	user_rating	3.859444e+00	4.039130e+00	1.796860e-01	4.655748
	sup_devices.num	3.688889e+01	3.720000e+01	3.111111e-01	0.843373
	lang.num	6.594444e+00	2.566087e+01	1.906643e+01	289.128603
cluster: 4, N.el.:22	feature	grand mean :	cluster mean :	differences :	rel. differences:
	size_bytes	1.148171e+08	9.814566e+08	8.666395e+08	754.799959
	user_rating	3.859444e+00	3.863636e+00	4.191919e-03	0.108615
	sup_devices.num	3.688889e+01	3.645455e+01	-4.343434e-01	-1.177437
	lang.num	6.594444e+00	6.000000e+00	-5.944444e-01	-9.014322
cluster: 5, N.el.:83	feature	grand mean :	cluster mean :	differences :	rel. differences:
	size_bytes	1.148171e+08	9.184987e+07	-2.296725e+07	-20.003329
	user_rating	3.859444e+00	3.957831e+00	9.838688e-02	2.549250
	sup_devices.num	3.688889e+01	2.424096e+01	-1.264793e+01	-34.286544
	lang.num	6.594444e+00	5.710843e+00	-8.836011e-01	-13.399174

Lets pck up those features and categories which values of relative difference are far from 0: **greater than 20%** and **less than 20%**.

We can see that **first cluster** has very low number of supported languages with relative difference -44.278813 and little low size (in Bytes) of app with relative difference -17.623538. These can be small applications not from large studios for some local things necessary for a specific countries (so they support just one language).

Second cluster has specimens just about the average for all features, besides of number of supported languages. It has very high relative difference 289.128603. Those might be apps for studying languages, translators, dictionaries and so on.

Third cluster has specimens just about the average for user rating and number of supported languages, but low relative differences for size and number of supported devices. So these might be apps with no integrations with smart watches, laptops and so on. Nowadays, most of the apps has some kind of integrations with those devices, and these might be those which still don't.

Fourth cluster has very low relative difference of size of app, user rating and number of supported languages. These can be bad, underdeveloped applications with little functionality and little language support. They probably crash often and don't work well, "trash apps"basically.

Fifth cluster has specimens just about the average for all features, besides of size of app. It has relative difference 754.799959, what is very very high. These can be heavy games with a big storyline adapted for the iPhone / iPad. A few years ago, this trend began when games like "GTA"and "Civilization"were adapted for IOS.

And here is a table with values for 9 clusters:

cluster: 1, N.el.:145	feature	grand mean :	cluster mean :	differences :	rel. differences:
	size_bytes	1.148171e+08	9.500082e+07	-1.981630e+07	-17.259011
	user_rating	3.859444e+00	4.279310e+00	4.198659e-01	10.878921
	sup_devices.num	3.688889e+01	3.748276e+01	5.938697e-01	1.609888
	lang.num	6.594444e+00	1.318621e+01	6.591762e+00	99.959330
cluster: 2, N.el.:26	feature	grand mean :	cluster mean :	differences :	rel. differences:
	size_bytes	1.148171e+08	8.324764e+07	-3.156948e+07	-27.495444
	user_rating	3.859444e+00	3.076923e-01	-3.551752e+00	-92.027549
	sup_devices.num	3.688889e+01	3.846154e+01	1.572650e+00	4.263207
	lang.num	6.594444e+00	2.461538e+00	-4.132906e+00	-62.672542
cluster: 3, N.el.:30	feature	grand mean :	cluster mean :	differences :	rel. differences:
	size_bytes	1.148171e+08	5.871922e+08	4.723751e+08	411.415235
	user_rating	3.859444e+00	4.183333e+00	3.238889e-01	8.392112
	sup_devices.num	3.688889e+01	3.786667e+01	9.777778e-01	2.650602
	lang.num	6.594444e+00	4.066667e+00	-2.527778e+00	-38.331929
cluster: 4, N.el.:280	feature	grand mean :	cluster mean :	differences :	rel. differences:
	size_bytes	1.148171e+08	8.568944e+07	-2.912768e+07	-25.368761
	user_rating	3.859444e+00	4.303571e+00	4.441270e-01	11.507537
	sup_devices.num	3.688889e+01	3.762857e+01	7.396825e-01	2.005164
	lang.num	6.594444e+00	1.989286e+00	-4.605159e+00	-69.833915
cluster: 5, N.el.:7	feature	grand mean :	cluster mean :	differences :	rel. differences:
	size_bytes	1.148171e+08	1.460312e+09	1.345495e+09	1171.858971
	user_rating	3.859444e+00	4.071429e+00	2.119841e-01	5.492607
	sup_devices.num	3.688889e+01	3.957143e+01	2.682540e+00	7.271945
	lang.num	6.594444e+00	9.428571e+00	2.834127e+00	42.977494
cluster: 6, N.el.:184	feature	grand mean :	cluster mean :	differences :	rel. differences:
	size_bytes	1.148171e+08	8.848036e+07	-2.633675e+07	-22.938003
	user_rating	3.859444e+00	3.116848e+00	-7.425966e-01	-19.241024
	sup_devices.num	3.688889e+01	3.742391e+01	5.350242e-01	1.450367
	lang.num	6.594444e+00	2.570652e+00	-4.023792e+00	-61.017911

cluster: 7, N.el.:75	feature	grand mean :	cluster mean :	differences :	rel. differences:
	size_bytes	1.148171e+08	5.205001e+07	-6.276711e+07	-54.667029
	user_rating	3.859444e+00	4.086667e+00	2.272222e-01	5.887433
	sup_devices.num	3.688889e+01	4.430667e+01	7.417778e+00	20.108434
	lang.num	6.594444e+00	1.533333e+00	-5.061111e+00	-76.748104
cluster: 8, N.el.:70	feature	grand mean :	cluster mean :	differences :	rel. differences:
	size_bytes	1.148171e+08	1.108168e+08	-4.000272e+06	-3.484038
	user_rating	3.859444e+00	3.964286e+00	1.048413e-01	2.716486
	sup_devices.num	3.688889e+01	3.707143e+01	1.825397e-01	0.494836
	lang.num	6.594444e+00	3.074286e+01	2.414841e+01	366.193284
cluster: 9, N.el.:83	feature	grand mean :	cluster mean :	differences :	rel. differences:
	size_bytes	1.148171e+08	9.184987e+07	-2.296725e+07	-20.003329
	user_rating	3.859444e+00	3.957831e+00	9.838688e-02	2.549250
	sup_devices.num	3.688889e+01	2.424096e+01	-1.264793e+01	-34.286544
	lang.num	6.594444e+00	5.710843e+00	-8.836011e-01	-13.399174

9 clusters clusterization showed very poor results, especially compared to 5 clusters one. There is a cluster with only 7 elements out of 900, it is not a good value for criteria of clusters fullness. So we are not going to consider those ones.

3 Home Work Assignment 3: Bootstrap

In case of 4 clusters partition, we can compare first and second clusters from there. They are pretty similar in each parameter, besides of users rating. So it is interesting to compare the within-cluster means for number of supported languages between these two clusters by using bootstrap and central limit theorem. After that we will compare the grand mean with the within cluster mean for that feature in first cluster also by using bootstrap and central limit theorem.

First of all, we need to find the 95% confidence interval for number of supported languages feature grand mean by using bootstrap in both, pivotal and non-pivotal, versions. Here is a code in Python 3 to get left and right boundaries of 95% confidence interval of our feature in our sample, first cluster and second cluster. Also we get new data of that feature in main dataframe and clusters. We use 5000 trials means for that:

```
def compute_confidence(resampled_means):
    """
```

```
        Compute the confidence interval
        Arguments:
            resampled_means data, numpy array.
        Returns:
            bootstrap means, bootstrap std, and pivotal and non pivotal
                boundries
    """

    # pivotal method:
    boots_mean = np.mean(resampled_means)
    boots_std = np.std(resampled_means)

    # 95\% confidence, thats why 1.96
    lbp = boots_mean - 1.96*boots_std  # left bound pivotal
    rbp = boots_mean + 1.96*boots_std  # right bound pivotal

    # Non-pivotal:
    lower_bound_index = int((len(resampled_means))*.025)
    higher_bound_index = int((len(resampled_means))*.975)
    resampled_mean_sorted = sorted(resampled_means,)

    lbn = resampled_mean_sorted[lower_bound_index]  # left bound non pivotal
    rbn = resampled_mean_sorted[higher_bound_index]  # right bound non
        pivotal

    return boots_mean, boots_std, lbp, rbp, lbn, rbn
```

```
def bootstrapping(x, x_idx_1, x_idx_2, n_resample, n_bootstrap):

    """
        x, a 1-D array, representing the random samples.
        x_idx, a 1-D array, representing the random samples indices:
            e.g 1st cluster indices (membership vector) or
            a range from 0 to number of random samples.
        n_resample, an int., representing the random sampling size.
        n_bootstrap, an int, representing the number of
            repeats in the bootstrapping procedure.

        Returns x_resampled_means, x_resampled_stds,
                boots_mean, boots_std,
                lbp, rbp, lbn, rbn,
        where:
            x_resampled_means: list of means of resampled data(x) of the
                length n_resample;
            x_resampled_stds: list of standard deviations of resampled data(x
                );
                of the length n_resample,
            boots_mean: floating number, representing bootstrap mean --i.e
                mean of means)
            boots_std: floating number, representing bootstrap std --i.e std
                of stds.
    """

    np.random.seed(43)  # for the sake of reproducibility
    x_resampled_means, x_resampled_stds = [], []  # grand mean
    x_resampled_means_1, x_resampled_stds_1 = [], []  # 1st subset
    x_resampled_means_2, x_resampled_stds_2 = [], []  # 2nd subset

    for _ in range(n_bootstrap):

        # Grand mean:
        list_to_choose = range(0, n_resample, 1)
        resampled_idx = np.random.choice(list_to_choose,
                                         size=n_resample,
                                         replace=True,)
```

```
        x_resampled = x[resampled_idx]
        x_resampled_means.append(np.mean(x_resampled))
        x_resampled_stds.append(np.std(x_resampled))  # /np.sqrt(len(x)))

        # 1st subset:
        idx_1 = [i for i in resampled_idx if i in x_idx_1]
        x_resampled_1 = x[idx_1]
        x_resampled_means_1.append(np.mean(x_resampled_1))
        x_resampled_stds_1.append(np.std(x_resampled_1))  # /np.sqrt(len(x)))

        # 2nd subset:
        idx_2 = [i for i in resampled_idx if i in x_idx_2]
        x_resampled_2 = x[idx_2]
        x_resampled_means_2.append(np.mean(x_resampled_2))
        x_resampled_stds_2.append(np.std(x_resampled_2))  # /np.sqrt(len(x)))

    x_resampled_means = np.asarray(x_resampled_means)
    x_resampled_means_1 = np.asarray(x_resampled_means_1)
    x_resampled_means_2 = np.asarray(x_resampled_means_2)

    # Grand mean:
    boots_mean, boots_std, lbp, rbp, lbn, rbn = compute_confidence(
        resampled_means=x_resampled_means)

    # Subtraction of grand mean from the 1st cluster:
    d1g = np.subtract(x_resampled_means_1, x_resampled_means)
    boots_mean_1g, boots_std_1g, lbp_1g, rbp_1g, lbn_1g, rbn_1g =
         compute_confidence(
        resampled_means=d1g)

    # Subtraction of 1st cluster from 2nd cluster:
    d12 = np.subtract(x_resampled_means_1, x_resampled_means_2)
    boots_mean_12, boots_std_12, lbp_12, rbp_12, lbn_12, rbn_12 =
         compute_confidence(
        resampled_means=d12)

    # x_resampled_means, x_resampled_stds, boots_mean, boots_std, lbp, rbp,
        lbn, rbn

    return ((x_resampled_means, x_resampled_means_1,  x_resampled_means_2),
        (lbp, rbp, lbn, rbn),
            (lbp_1g, rbp_1g, lbn_1g, rbn_1g),
            (lbp_12, rbp_12, lbn_12, rbn_12))

n_resample = df_hw1.to_numpy().shape[0]
n_bootstrap = 5000
feature_n = 3

((x_resampled_means, x_resampled_means_1,  x_resampled_means_2),
 (lbp, rbp, lbn, rbn), \
 (lbp_1g, rbp_1g, lbn_1g, rbn_1g), \
 (lbp_12, rbp_12, lbn_12, rbn_12)) = bootstrapping(x=df_hw1.to_numpy()[:,
     feature_n],
                                                   x_idx_1=best_indices_1[0],
                                                   x_idx_2=best_indices_1[1],
                                                   n_resample=n_resample,
                                                   n_bootstrap=n_bootstrap)
```

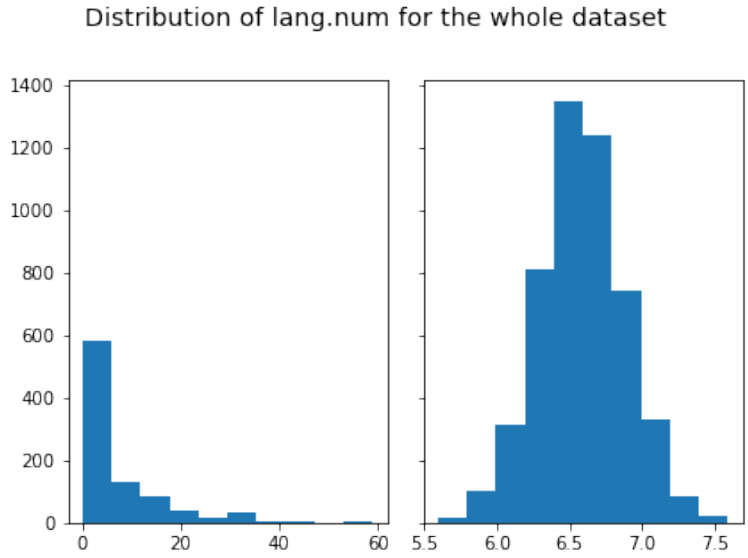
Here we did bootstrapping by creating random samples from a number of supported languages feature and calculation of their expected values. We got 5000 means of random samples and by them we got right and left boundaries for 95% confidence intervals.

Here is 95% confidence intervals for number of supported languages feature

for the whole dataset with 5000 trials:

	Left Bound	Right Bound
Pivotal	6.0160	7.162
Nonpivotal	6.01	7.1767

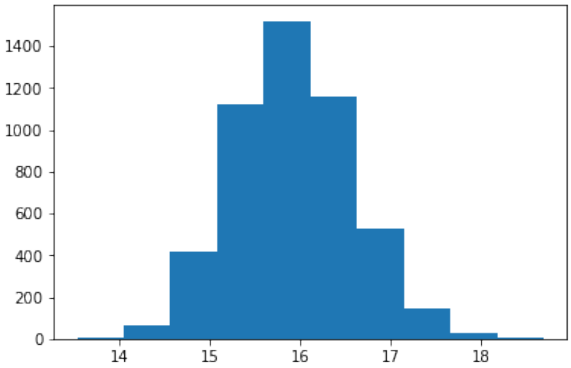
As we can see, after bootstrapping, distribution become Gaussian:



Here is 95% confidence intervals for the difference between expected values of number of supported languages feature for first cluster and the whole dataset with 5000 trials:

	Left Bound	Right Bound
Pivotal	14.6308	17.1932
Nonpivotal	14.7058	17.2538

Distribution of for the diff. between lang.num and the whole dataset

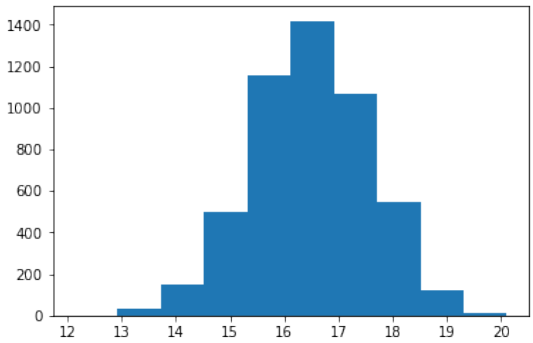


As we can see zero is not in interval, it means that Mean in first cluster is greater than in the whole dataset for both Pivotal and Nonpivotal methods.

And here is 95% confidence intervals for the difference between expected values of number of supported languages feature for first and second clusters with 5000 trials:

	Left Bound	Right Bound
Pivotal	14.3694	18.6084
Nonpivotal	14.3817	18.5515

Distribution of for the diff. between lang.num for first and second clusters



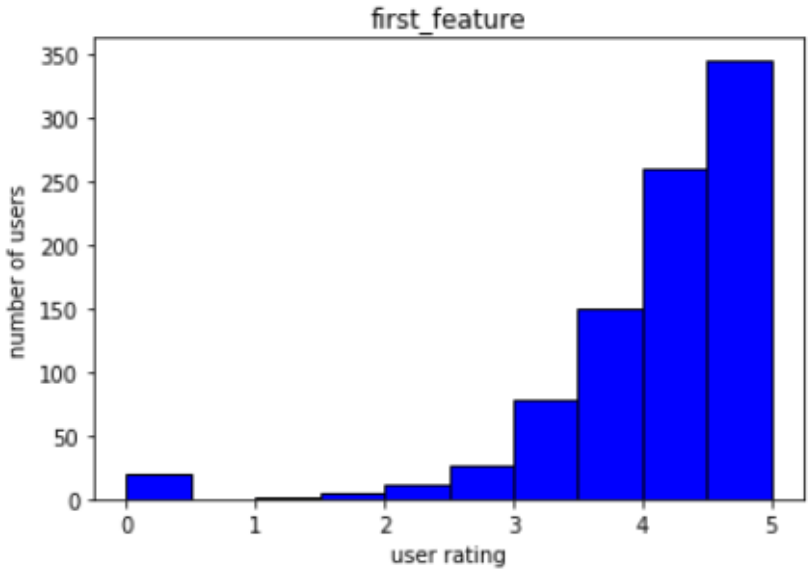
As we can see zero is not in interval, it means that with 95% confidence we can say that Mean in first cluster is greater than in the second for both Pivotal and Nonpivotal methods. So we can conclude that first and second cluster strongly differ as app types in terms of number of supported language.

Homework assignment 4: Contingency table.

1. Consider three nominal features.

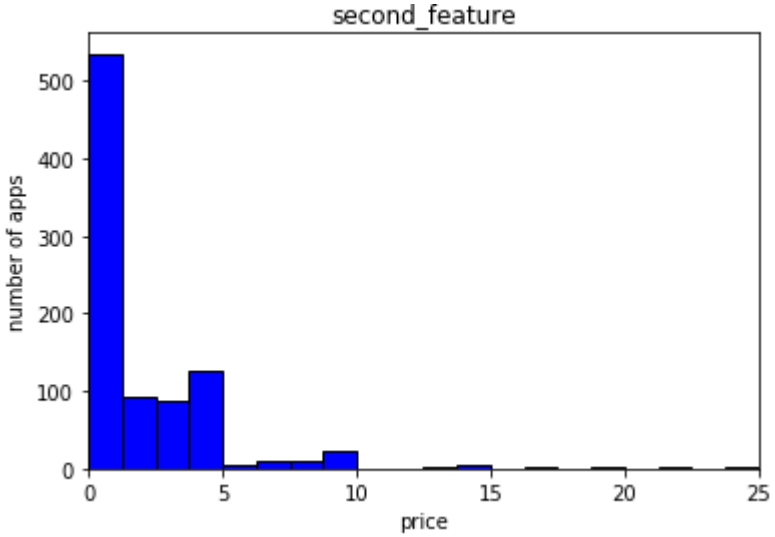
In order to get some insights about the dependencies of some variables. We decided to take the rating of the app and try to find some dependencies between this feature and two other features: the price of the app (price) and the content rating (cont_rating).

To check it through contingency tables we need to bin these these features. First we need to take a look at distribution of this features. First feature is user_rating:



This feature is distributed from 0 to 5 and we decided to make three categories according to the values of rating of the app and the boundaries for them are [0, 3.5, 4.5, 5.0]. So, we received categories if rating is less or equal 3.5, if it is equal to 4 and if it is more or equal 4.5.

Let's take a look at the distribution of the second feature - price of the app.



From distribution of the second feature we can observe that almost the half of the apps are free for users, more precisely – the number of free apps is 404, the number of apps with some price is 497 and we decided to make this feature binary for this part – is free or not.

The third feature is content rating and contains four categories – 4+, 9+, 12+, 17+.

2. Build two contingency tables over them: present a conditional frequency table and Quetelet relative index tables.

Let us present a conditional frequency table and quetelet index table of user rating and price of app:

Frequency table					Quetelet index			
Price/rating	0-3.5	4	4.5-5		Price/rating	0-3.5	4	4.5-5
0	0.379	0.636	0.636		0	-0.313	0.151	0.151
1	0.621	0.364	0.364		1	0.387	-0.187	-0.186

Rows represents categories of rating and columns represents price category. From conditional frequency table we can observe that apps with rating less than 3.5 are more probable to be free and for apps with rating 4 and higher it is more probable to be non-free as the probability is twice more. As for the quetelet index table we can only say that there are no relatively big numbers.

Let's check contingency tables over the rating and the second feature:

Frequency table					Quetelet index				
Rtg/age	12+	17+	4+	9+	Rtg/age	12+	17+	4+	9+
0-3.5	0.338	0.363	0.324	0.253	0-3.5	0.038	0.116	-0.003	-0.22
4	0.239	0.383	0.275	0.388	4	-0.174	0.323	-0.05	0.338
4.5-5	0.422	0.252	0.4	0.358	4.5-5	0.099	-0.343	0.041	-0.068

We also cannot see high dependency between rating and age limit which is logical, but also it is more probable for 4+ and 12+ limits to have rating 4.5-5.0 and for 9+, 17+ to have 4.0. In this case we also do not observe relatively big numbers in quetelet index table.

Contingency tables were built with the help of the function:

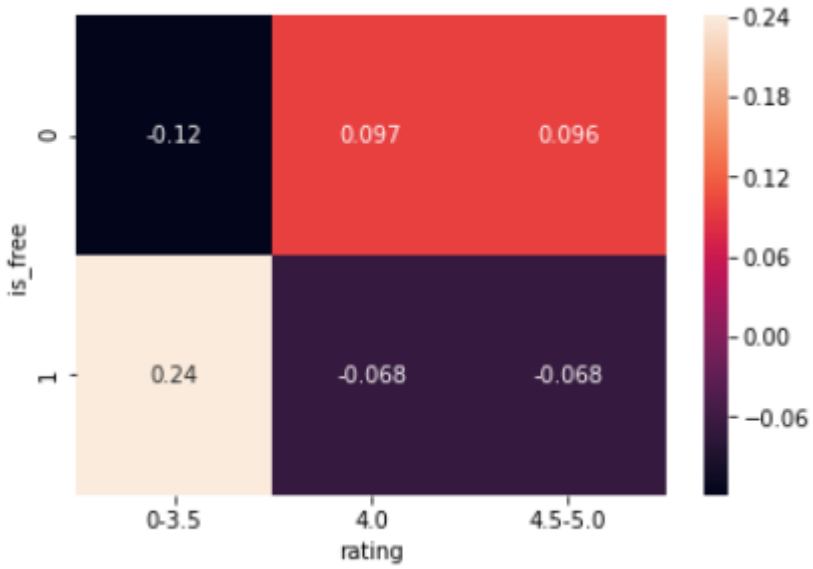
```python
def contingency_table(df, first_feature, second_feature):
    conditional_frequency_table = pd.crosstab(df[first_feature], df[second_feature], normalize='columns')
    quetelet_table = pd.crosstab(df[first_feature], df[second_feature])

    n = quetelet_table.to_numpy().sum()
    quetelet_table_copy = quetelet_table.copy()
    for i in range(len(quetelet_table)):
        for j in range(len(quetelet_table.columns)):
            pi = sum(quetelet_table_copy.iloc[i, :])
            pj = sum(quetelet_table_copy.iloc[:, j])
            quetelet_table.iloc[i, j] = n * quetelet_table.iloc[i, j] / (pi * pj) - 1

    return conditional_frequency_table, quetelet_table
```

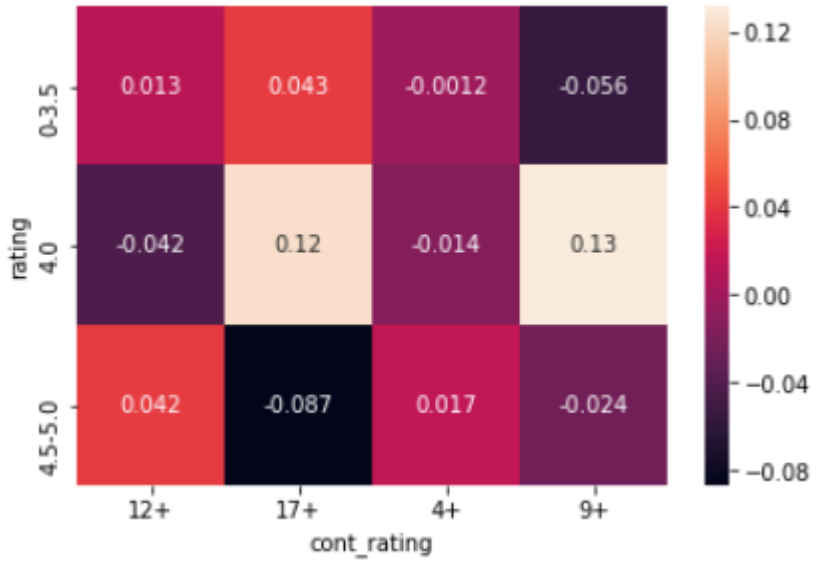
3. Compute and visualize the chi-square-average-Quetelet index over both tables.

Average quetelet index is the inner product of two matrices: observe relative contingency table and the table of quetelet indeces. Let's visualize it for the first table:



Sum of elements of table above gives us average Quetelet index, that tells us how much frequency knowing what category ratings "adds" to the probability of to be free or non-free for users. In our case Avg_Q = 0.178. It means that knowing category rating gives us 17.8%.

Visualization for the second table:



For this case average index is equal to 0.145. It means that knowledge of content rating (cont_rating) adds 14.5% to the frequency of rating categories.

Both indicies are pretty small and we cannot infere about strong dependence between the variables. It looks like the truth as rating of the app ususally does not depend on content_rating but first index is higher and it can be explained by the lack of one's willingness to pay for poorly rated apps.

4. Tell: what numbers of observations would suffice to see the features as associated at 95% confidence level; at 99% confidence level

To compute X-square for this part we will use scipy.stats library. The function:

```
def number_of_observations(average_quetelet_index, confidence_level):
    n = len(average_quetelet_index)
    k = len(average_quetelet_index.columns)
    dataframe = (n - 1)*(k - 1)

    x = average_quetelet_index.to_numpy().sum()
    return ceil(chi2.ppf(confidence_level, dataframe) / x)
```

The results for the first two features:

"rating" and "is_free" are associted at confidence level 95% if N > 34
"rating" and "is_free" are associted at confidence level 99% if N > 52

The results for the second features:
"cont_rating" and "rating" are associted at confidence level 95% if N > 87
"cont_rating" and "rating" are associted at confidence level 99% if N > 116

Home Work Assignment 5: PCA/SVD

We want to estimate mobile apps and find the least and the most successful of them. We can do it using hidden factor ranking via SVD. First of all, we have to choose the most representative features for our analysis. We choose the following list:

1. user_rating;
2. price;
3. size_bytes;
4. sup_devices.num;
5. lang.num.

The same features were used in K-means clustering, so it's interesting to see how they work for PCA. Someome can say, that main factor of success is user's rating. However, this feature doesn't take account of availability of app. We can create some very cool app, however if price is too high, only a small part of users can use it. Therefore, such factors like price, number of languages supported and etc. are also important.

First step of our analysis is data standardization. We use three approaches:

a) Z-scoring $y_i = \frac{x_i - \bar{x}}{s}$

b) Range standardization $y_i = \frac{x_i - \bar{x}}{\max - \min}$

c) Rank method $y_i = \frac{x_i - \min}{\max - \min}$

We define *standardizer(x)* function, where x is our input matrix.

```
def standardizer(x):
    x_ave = np.mean(x, axis=0)
    x_min = np.min(x, axis=0)
    x_rng = np.ptp(x, axis=0)
    x_std = np.std(x, axis=0)
    x_zscore = np.divide(np.subtract(x, x_ave), x_std)
    x_range = np.divide(np.subtract(x, x_ave), x_rng)
    x_rank = np.divide(np.subtract(x, x_min), x_rng)
    return x_zscore, x_range, x_rank
```

Listing 1: Data standartization

To this moment, our data is ready and we can implement SVD. We define *singular_decomposition(x)* function. This function takes as input the matrix and returns SVD decomposition, data scatter, contributions of all principal components (naturally and per cent) and index of maximum singular value. The function *singular_decomposition(x)*:

```
def singular_decomposition(x):

    z, mu, c = np.linalg.svd(x, full_matrices=True)
    z = -z
    c = -c
    mu_arg_max = np.argmax(mu)
    n_contributions = np.power(mu, 2)
    ds = np.sum(np.power(x, 2))
    p_contributions = np.divide(n_contributions, ds)

    return z, mu, c, ds, n_contributions, p_contributions,
    mu_arg_max
```

Listing 2: Singular value decomposition

We use this function for our standartized data. Results are presented in table 1 and table 2.

Table 1: Contribution of all principal components (naturally)

Natural contributions	**size_bytes**	**price**	**user_rating**	**sup_devices.num**	**lang.num**
Original data	34750818.51	889060.103	75154.181	70147.931	856.134
Ranked std data	1027.89	23.667	18.308	6.661	1.185
Ranged std data	28.275	19.844	16.086	6.648	1.184
Z-Scored std data	1105.518	1003.402	889.617	774.168	727.295

Table 2: Contribution of all principal components (per cent)

Percent contributions	**size_bytes**	**price**	**user_rating**	**sup_devices.num**	**lang.num**
Original data	97.107	2.484	0.21	0.196	0.002
Ranked std data	95.377	2.196	1.699	0.618	0.11
Ranged std data	39.25	27.54	22.331	9.229	1.644
Z-Scored std data	24.567	22.298	19.769	17.204	16.162

Then we can estimate success of mobile apps using hidden factor model:

$$x_{iv} = z_i c_v + e_{iv}$$

. We have to rescale z_1 to convert it to 0-100 scale. To do it, we have to find α using the following equation:

$$z_1 = \alpha \times (100 * c_{1,1} + ... + 100 \times c_{1,v})$$

By assigning 100 to z_1 we have:

$$\alpha = \frac{100}{\sum_{i=1}^{V} c_{1,i}}$$

Where V is the number of features, *ranking(x, z, c, mu, argmax)* function helps us to solve this task with arguments:our input matrix, SVD decomposition, index of maximum singular value.

```
def ranking(x, z, c, mu, argmax):
    z1 = z[:, argmax]
    c1 = c[argmax, :]
    mu1 = mu[argmax]
    alpha = 1/np.sum(c1)
    loading = c1*alpha
    ranking_factors = 100*x@(loading)
    return ranking_factors
```

Listing 3: Estimate hidden ranking factor

And we finally can see the top and the bottom ten apps ranking by their hidden ranking factor. Top 10 apps are in table 3 and worst ten apps are in table 4.

Table 3: Top 10 successeful mobile apps, according to hidden ranking factor

size_mbytes	price	user_rating	cont_rating	prime_genre	sup_devices.num	lang.num	HRF
11.359	0.99	5.0	9+	Games	47	1	89.326
595.196	0.99	5.0	12+	Games	43	13	86.918
347.605	9.99	5.0	12+	Games	43	1	85.103
25.796	2.99	4.5	4+	Games	47	1	84.689
22.361	2.99	4.5	4+	Games	47	1	84.684
1886.449	6.99	4.5	17+	Games	43	12	84.517
1735.875	6.99	4.5	17+	Games	43	12	84.240
40.080	0.00	5.0	4+	Utilities	40	22	83.290
483.998	13.99	4.5	9+	Games	45	1	83.124
33.371	3.99	4.5	4+	Games	45	1	82.277

We see, that top 10 apps have user_rating from 4.5 to 5.0, almost all are paid. It also worths to say, that all best apps are games except one.
Speaking about 10 least succesuful apps, we see, that for all 10 apps user rating is 0.0, which shows user frustration. It's interesting, that there is no any game in this list of apps. Finally, almost all apps are paid again. We can conclude, that even paid apps can be awful according to hidden ranking factor.

Table 4: Bottom 10 successeful mobile apps, according to hidden ranking factor

size_mbytes	price	user_rating	cont_rating	prime_genre	sup_devices.num	lang.num	HRF
7.783	1.99	0.0	4+	News	38	1	31.719
196.137	47.99	0.0	9+	Reference	37	2	31.069
3.219	3.99	0.0	4+	Finance	37	3	30.731
119.844	0.00	0.0	4+	Navigation	37	1	30.705
41.591	0.00	0.0	4+	Shopping	37	2	30.677
4.124	1.99	0.0	4+	Photo & Video	37	2	30.614
46.452	0.00	0.0	4+	Navigation	37	1	30.571
23.312	3.99	0.0	4+	Education	37	1	30.538
10.422	2.99	0.0	4+	Education	37	1	30.512
7.667	3.99	0.0	9+	Book	37	1	30.509

Let's visualize first two principal components at the standardization with two versions of normalization: (a) range normalization and (b) z-scoring and using Conventional PCA with centered data (c).

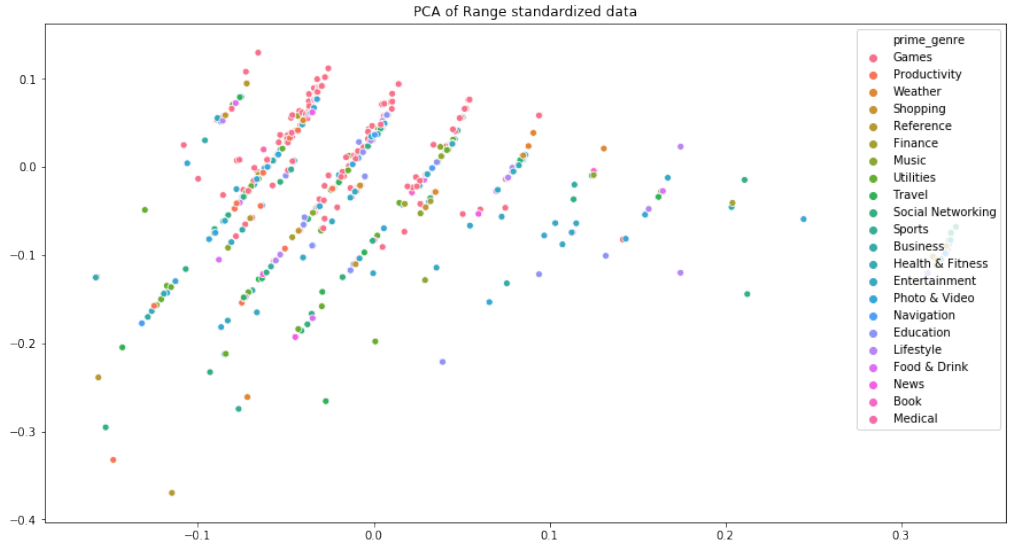


Figure 1: Visualisation of first two principal components with colour-encoded *prime_genre* feature for Range standartized data

Figure 2: Visualisation of first two principal components with colour-encoded *prime_genre* feature for Z-scoring standartized data

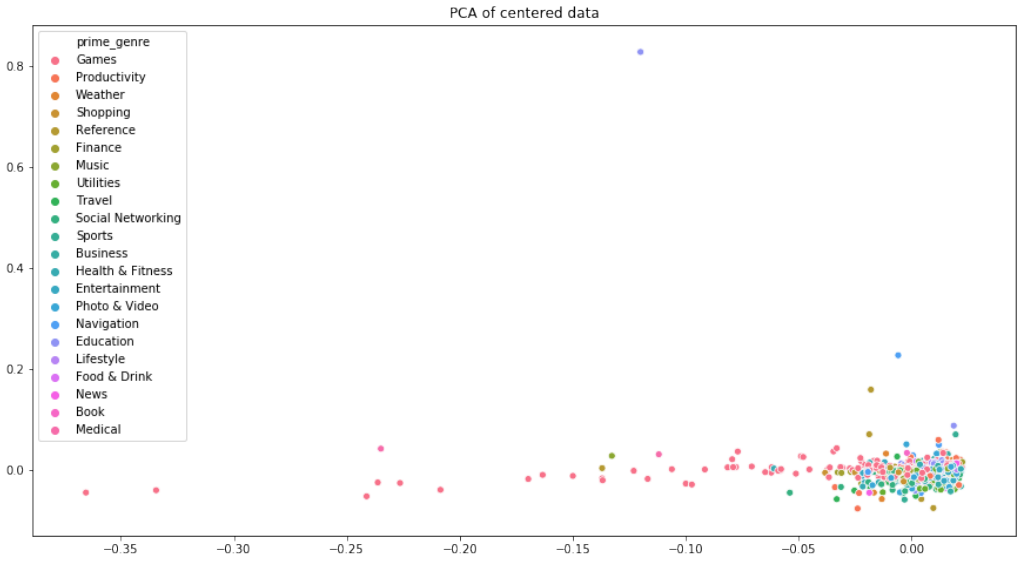


Figure 3: Visualisation of first two principal components with colour-encoded *prime_genre* feature for centered data using Conventional PCA

It seems, that 1 is the most representative. It is worse to say, that for Conventional PCA we use *conventional_pca(X)* function.

```
def conventional_pca(X):
    Y = X - np.mean(X,axis=0)
    B = (Y.T@Y)/Y.shape[0]
    L, C = np.linalg.eig(B)
    eigenvalue = L.max()
    eigenvector = C[:,L.argmax()]
    pc1 = Y@eigenvector / np.sqrt(Y.shape[0]*eigenvalue)
    B -= eigenvalue*(eigenvector[:,None] @ eigenvector[None,:])
    L, C = np.linalg.eig(B)
    eigenvalue = L.max()
    eigenvector = C[:,L.argmax()]
    pc2 = Y@eigenvector / np.sqrt(Y.shape[0]*eigenvalue)

return pc1, pc2
```

Listing 4: Estimate hidden ranking factor

Finally, I present all code for this task.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import linalg

np.set_printoptions(suppress=True, precision=3, linewidth=120)

df = pd.read_csv('Data/apple_store_dt_fin.csv', sep=',')

df = df.drop('Unnamed: 0', axis =1)

X = df.drop(['cont_rating','prime_genre','currency'], axis =1)

def standardizer(x):

    """
    standardize entity-to-feature data matrix by
    applying Z-scoring, Range standardization and Rank methods

    Arguments:
    x, numpy array, entity-to-feature data matrix
    Returns:
    Z-scored and Range standardized data matrices, x_zscore,
    x_range, x_rank.
    """

    x_ave = np.mean(x, axis=0)
    x_min = np.min(x, axis=0)
    x_rng = np.ptp(x, axis=0)
    x_std = np.std(x, axis=0)
    x_zscore = np.divide(np.subtract(x, x_ave), x_std)   # Z-
    scoring standardization
    x_range = np.divide(np.subtract(x, x_ave), x_rng)  # Range
    standardization
    x_rank = np.divide(np.subtract(x, x_min), x_rng) # Rank
    standardization
```

```
    return x_zscore, x_range, x_rank


X_np = X.to_numpy()

X_zscore, X_range, X_rank = standardizer(x=X_np)

def singular_decomposition(x):

    z, mu, c = np.linalg.svd(x, full_matrices=True)
    z = -z
    c = -c
    mu_arg_max = np.argmax(mu)
    n_contributions = np.power(mu, 2)
    ds = np.sum(np.power(x, 2))
    p_contributions = np.divide(n_contributions, ds)

    return z, mu, c, ds, n_contributions, p_contributions,
    mu_arg_max

# Singular decomposition on original data
Z_X, MU_X, C_X, ds_X, n_contributions_x, p_contributions_x,
    mu_argmax_x = \
singular_decomposition(x=X_np)

# Singular decomposition on Z-Scored standardized data
Z_X_zscore, MU_X_zscore, C_X_zscore, ds_x_zscore,\
n_contributions_x_zscore, p_contributions_x_zscore,
    mu_argmax_x_zscore  = \
singular_decomposition(x=X_zscore)

# Singular decomposition on Range standardized data
Z_X_range, MU_X_range, C_X_range, ds_x_range,\
n_contributions_x_range, p_contributions_x_range,mu_argmax_x_range
    = \
singular_decomposition(x=X_range)

# Singular decomposition on Rank standardized data
Z_X_rank, MU_X_rank, C_X_rank, ds_x_rank,\
n_contributions_x_rank, p_contributions_x_rank, mu_argmax_x_rank =
    \
singular_decomposition(x=X_rank)

print("Natural contributions of \n",
"Original data       :", n_contributions_x, "\n",
"Ranked std data     :", n_contributions_x_rank, "\n",
"Ranged std data     :", n_contributions_x_range, "\n",
"Z-Scored std data   :", n_contributions_x_zscore,)

print("Percent contributions of \n",
"Original data       :", 100*p_contributions_x, "\n",
"Ranked std data     :", 100*p_contributions_x_rank, "\n",
"Ranged std data     :", 100*p_contributions_x_range, "\n",
"Z-Scored std data   :", 100*p_contributions_x_zscore,)

def ranking(x, z, c, mu, argmax):
    z1 = z[:, argmax]  # hiden score
```

```
    c1 = c[argmax, :]  # loading
    mu1 = mu[argmax]
    print("c1              :", c1)
    alpha = 1/np.sum(c1)
    loading = c1*alpha
    print("Loading         :", loading)
    ranking_factors = 100*x@(loading)  # ranking factors
    return ranking_factors

rank_fact = ranking(x=X_rank, z=Z_X_rank, c=C_X_rank, mu=MU_X_rank,
     argmax=mu_argmax_x_rank)
df_rank = df.copy()
df_rank['HidRankFac'] = rank_fact
df_rank = df_rank.sort_values('HidRankFac', ascending=False)

df_rank_test.head(10)
df_rank_test.tail(10)

Z_range, MU_range, C_range = np.linalg.svd(X_range, full_matrices=
    True)

Z0_range = Z_range[:,0] * np.sqrt(MU_range[0])
Z1_range = Z_range[:,1] * np.sqrt(MU_range[1])

plt.figure(figsize=(15,8))
sns.scatterplot(x=Z0_range,y=Z1_range,hue=df['prime_genre'])
plt.title("PCA of Range standardized data")
plt.show()

Z_zscore, MU_zscore, C_zscore = np.linalg.svd(X_zscore,
    full_matrices=True)

Z0_zscore = Z_zscore[:,0] * np.sqrt(MU_zscore[0])
Z1_zscore = Z_zscore[:,1] * np.sqrt(MU_zscore[1])

plt.figure(figsize=(15,8))
sns.scatterplot(x=Z0_zscore,y=Z1_zscore,hue=df['prime_genre'])
plt.title("PCA of Z-score standardized data")
plt.show()


def conventional_pca(X):
    Y = X - np.mean(X,axis=0)
    B = (Y.T@Y)/Y.shape[0]
    L, C = np.linalg.eig(B)
    eigenvalue = L.max()
    eigenvector = C[:,L.argmax()]
    pc1 = Y@eigenvector / np.sqrt(Y.shape[0]*eigenvalue)
    B -= eigenvalue*(eigenvector[:,None] @ eigenvector[None,:])
    L, C = np.linalg.eig(B)
    eigenvalue = L.max()
    eigenvector = C[:,L.argmax()]
    pc2 = Y@eigenvector / np.sqrt(Y.shape[0]*eigenvalue)


    return pc1, pc2

```

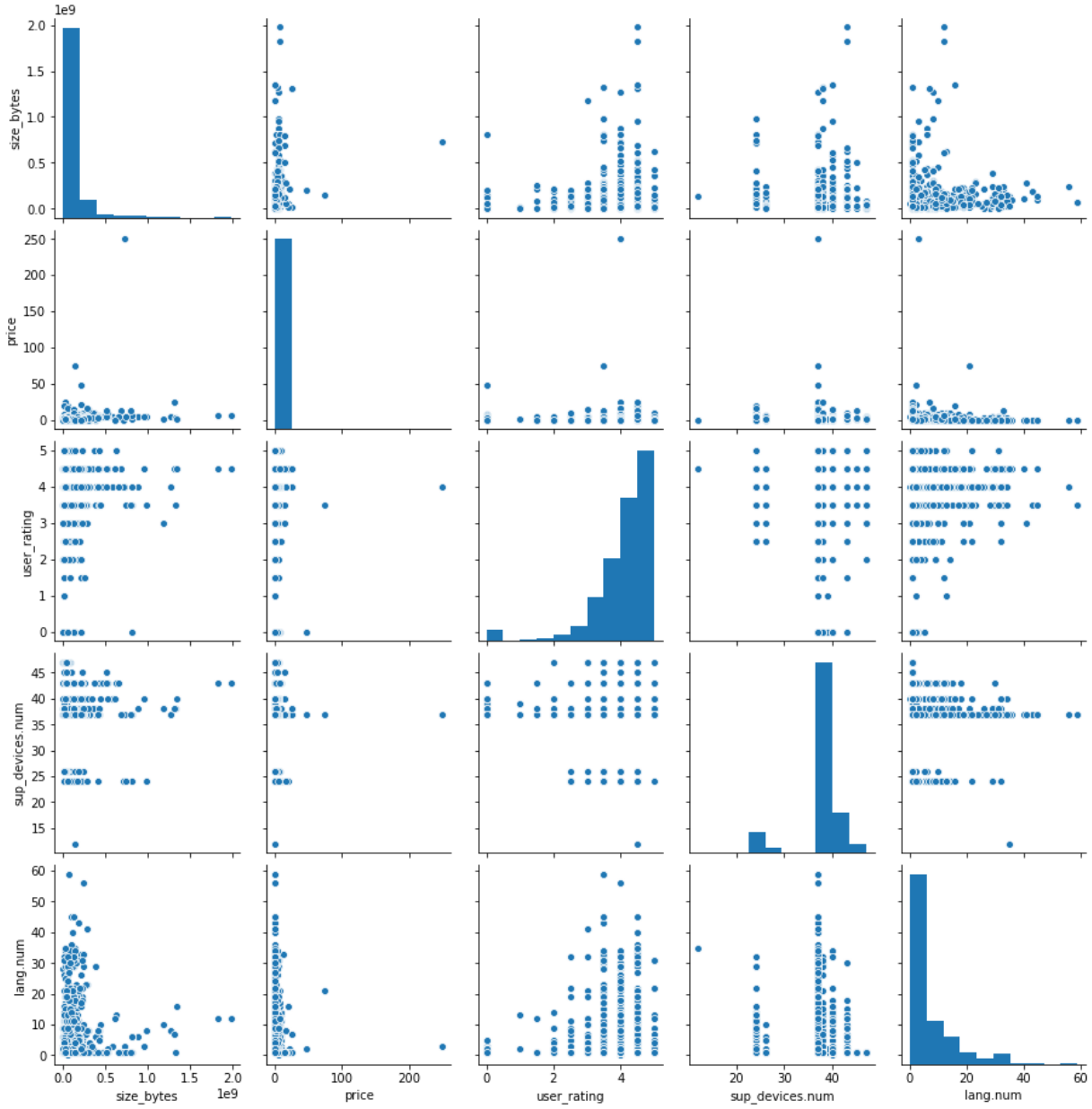
```
Z0_conventional,Z1_conventional = conventional_pca(X_np)

plt.figure(figsize=(15,8))
sns.scatterplot(x=Z0_conventional,y=Z1_conventional,hue=df['
    prime_genre'])
plt.title("PCA of centered data")
plt.show()
```

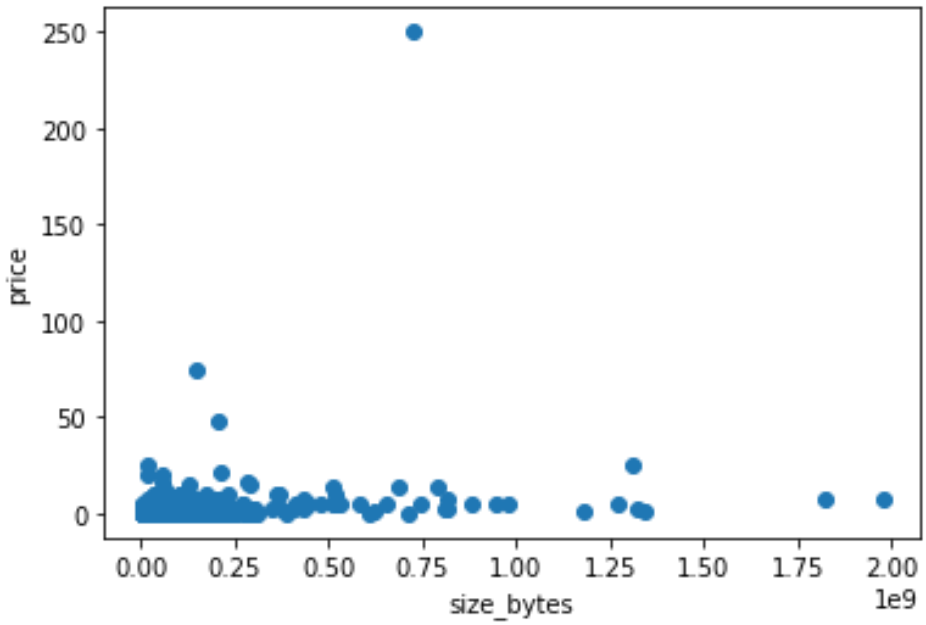
Listing 5: Full code for PCA/SVD

Home Work Assignment 6: Correlation coefficient

To learn more about how the characteristics of mobile apps are related to each other, let's look at the scatter plots that are presented below.



As we can see, there are no obvious linear patterns, but the only thing that may be interesting and has more linear-like scatterplot is the size bytes - price pair, shown below.



As we can see, most of the values are concentrated very close to zero. This is due to the fact that mobile applications are either free or very cheap.

Linear regression

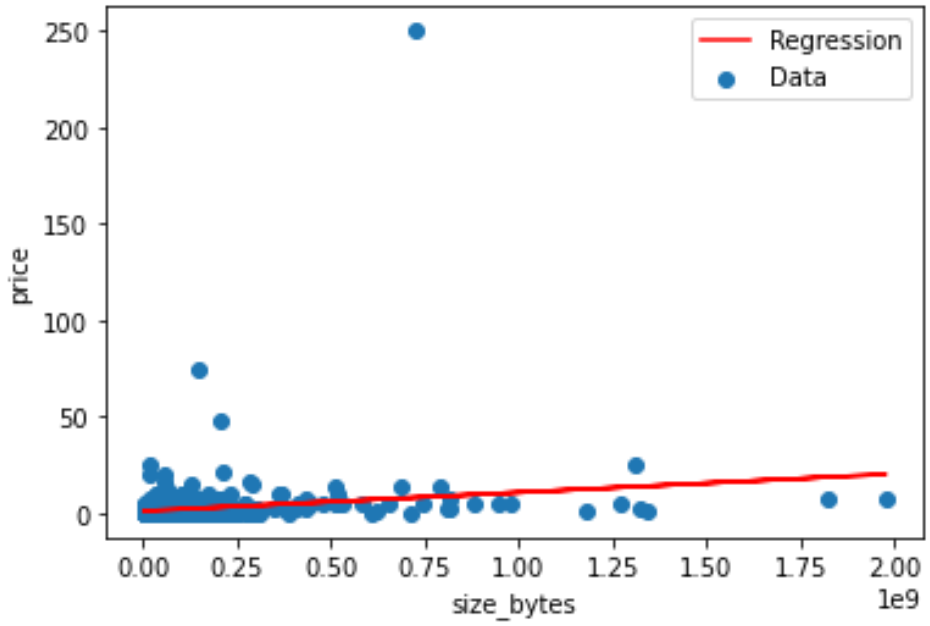
In order to build linear regression, we will use standard methods from the sklearn library.

```
from sklearn import linear_model
import numpy as np

x = np.array(df['size_bytes'])
y = np.array(df['price'])

model = linear_model.LinearRegression()
model.fit(x.reshape(-1, 1), y)
print("y = {:.4}x + {:.4}".format(model.coef_[0], model.intercept_))
model.coef_[0]
```

```
plt.plot(df[['size_bytes']], model.predict(df[['size_bytes']]), color='r', label="Regressio
n")
plt.scatter(df['size_bytes'], df['price'], label='Data')
plt.xlabel('size_bytes')
plt.ylabel('price')
plt.legend()
```



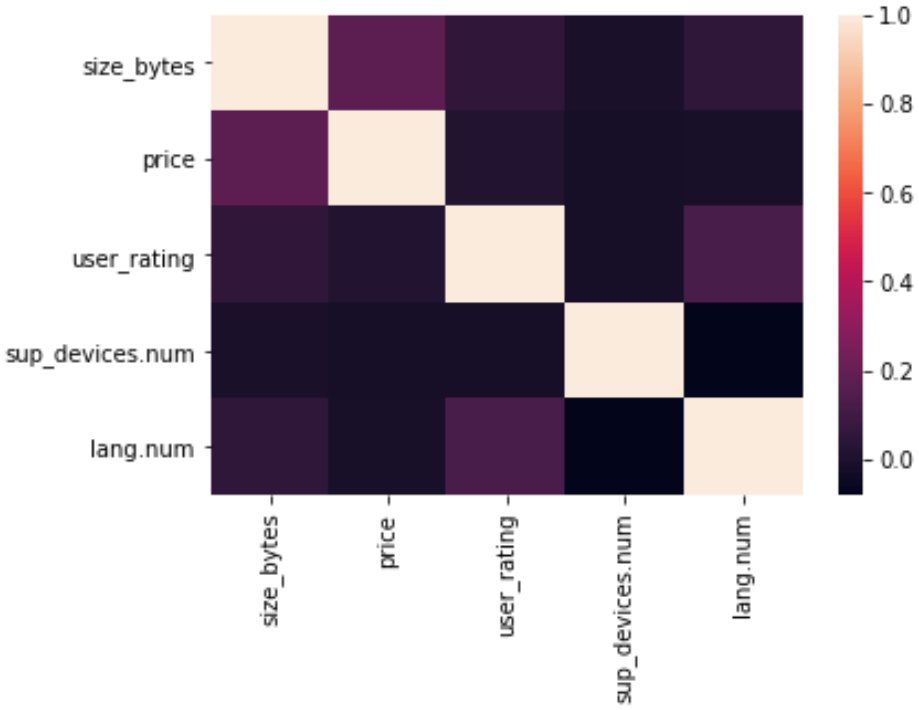
Y = 9.62x + 1.276

Slope	9.62
Intercept	1.276

The slope is positive, which means the more bytes the higher the price. The slope shows that for every increase in bytes by one, the price increases by 9,62 $.

Correlation and determinacy coefficients

The correlation matrix below shows the correlation between features



Coefficients for pair size bytes – price

Correlation coefficient	0.1767
Determinacy coefficient	0.03124

Correlation coefficient is a measure of degree of a linear relation between x and y. The closer it is to 1 or -1 the more linearly related are features. As we can see, the linear relation is very low.

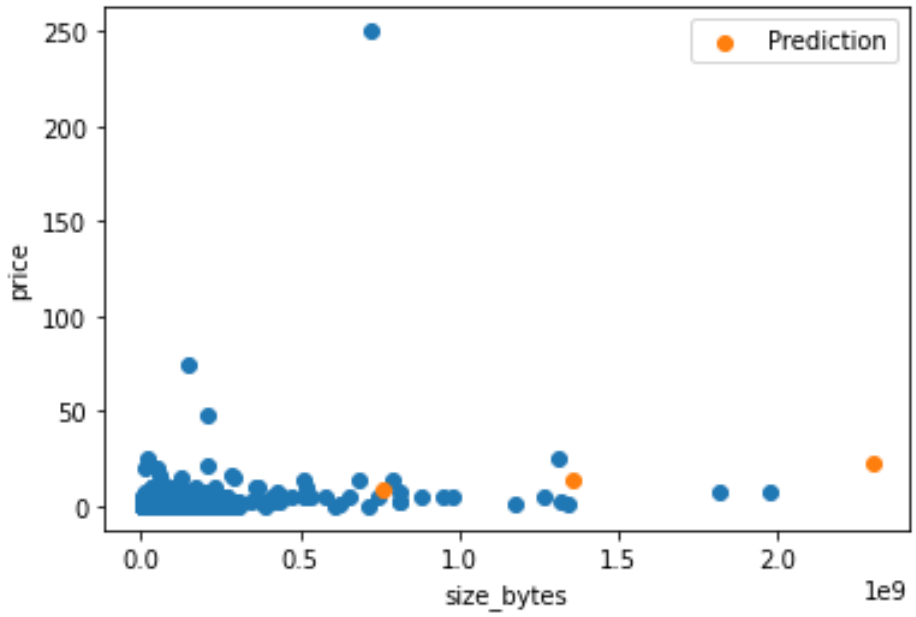
The determinacy coefficient is the proportion of the variance of y taken into account by the linear regression of y over x. It can be obtained by squaring the correlation coefficient. In our case the proportion is only 3%.

Prediction of the target values

Let's choose three target values and mark them on the graph.

```
import numpy as np
1
sets = np.array([759282132, 1359282132, 2299292132])

plt.scatter(df['size_bytes'], df['price'])
plt.scatter(sets, model.predict(sets.reshape(-1, 1)), label='Prediction')
plt.xlabel('size_bytes')
plt.ylabel('price')
plt.legend()
plt.show()
print('predict', model.predict(sets.reshape(-1, 1)))
```



predict [8.57962186 14.35146831 23.39412393]

The predicted values are quite close to target values up to 1.5. It is Because the price cannot increase indefinitely. There is a certain point after which the price will not rise at all.

Compare the MRAE and determinacy coefficient

```
new_df = pd.DataFrame()
new_df['price'] = df[df['price'] > 0.1] ['price']

y_true = new_df['price']
y_pred = model.predict(new_df[['price']])

mrae = 100 * np.mean(np.absolute(np.divide(y_true - y_pred, y_true)))

print("mean relative absolute error : %.3f" % mrae + "%")
print("Determinacy coefficients: {:.4}".
      format((cor['size_bytes']['price'] ** 2)))
```

mean relative absolute error : 53.190%

Determinacy coefficients: 0.03124

By squaring the correlation coefficient, we get the coefficient of determination. The square of this statistical measure shows how well the regression model describes the data. If it is close to one, then the function fits very well on all points. In our case, the MRAE is very high, and the coefficient is very low. This means that our function is very bad at describing the spread of data. This is directly related to the dataset. Many phone apps are free or very cheap. The price of apps can't be infinitely high. In other words, most of the data is concentrated close to zero, so it's not surprising that we got this result.