ORIE 5355/INFO 5370 HW 2: Recommendation systems

- Name: Nikhil Pereira
- Net-id:nmp54
- Date: 10/5/2021
- Late days used for this assignment:0
- Total late days used (counting this assignment):1
- People with whom you discussed this assignment: Julia Hoffman

After you finish the homework, please complete the following (short, anonymous) post-homework survey: https://forms.gle/5AG69LXM4yV3TL7o6 (https://forms.gle/5AG69LXM4yV3TL7o6) and include the survey completion code below.

Question 0 [2 points]

Survey completion code: hw22021abc

```
We have marked questions in <font color='blue'> blue </font>. Please put answers in black (do not change colors). You'll want
to write text answers in "markdown" mode instead of code. In Jupyter notebook, you can go to Cell > Cell Type > Markdown, from
the menu. Please carefully read the late days policy and grading procedure [here]
(https://orie5355.github.io/Fall_2021/assignments/).
```

Conceptual component

Go through the "Algorithms tour" here (https://algorithms-tour.stitchfix.com/). It's a great view of the combination of algorithms used by a modern e-commerce company.

1) How do they use a combination of "latent" factors and explicit features to gain the benefits of collaborative filtering (matrix factorization) while not being susceptible to cold start issues?

They use explicit data from client's style profile and supplier's clothing attributes to fill in gaps for cold start users. Latent factors such as size and style are inferred from previous purchase history. Then they employ a "mixed-effects" approach to track client's preferences over time using both the explicit data and "latent" factors with the hope of keeping customers overtime.

2) How do they match clients with human stylists who make the final decision? Does it remind you of anything we learned in class?

According to Stitch Fix, "We first calculate a match score between each available stylist and each client. This score is a complex function of the history between that client and stylist (if any), and the affinities between the client's stated and latent style preferences and those of the stylist. " This match score is very similar to the "Index" policy we learned in class where we need a score between each user i and stylist j and this time the capacity variable is representative of the queue of the stylists work.

3) How do they manage their inventory to ensure that they have enough items that future customers will want?

To manager inventory for future customers they allocate inventory appropriately to different warehouses, and occasionally donate old inventory to make room for new styles. Using a model of robust optimization for system dynamics, they fit it to historical data and quantify uncertainties from forecasts.

Programming component

Helper code

In [1]:
```python
import numpy as np
import pandas as pd
import os, sys, math
import matplotlib.pyplot as plt
import pickle
def load_pickle(filename):
    with open(filename, "rb") as f:
        data = pickle.load(f)
    return data

def load_ratings_and_factors(type_name = 'interaction'):
    ratings = load_pickle('data/{}_ratings'.format(type_name))
    book_vectors = load_pickle('data/{}_dict_book_factor'.format(type_name))
    user_vectors = load_pickle('data/{}_dict_reader_factor'.format(type_name))
    return ratings, book_vectors, user_vectors
```

In this homework, we are giving you trained user and book item vectors using a GoodReads dataset. Goodreads is a social cataloging website that allows individuals to search its database of books, annotations, quotes, and reviews. There are multiple types of interactions that a user can have with a book: add books to a list of books they intend to read ("short-list" the book), indicate they have read books before, and review books they have read.

Here, we work with multiple types of interactions as training data for a recommendation system. For each "type" of rating data, we give you the raw ratings data, as well as user and item vectors trained using a Python package (https://berkeley-reclab.github.io/ (https://berkeley-reclab.github.io/)) that implements matrix factorization in cases where there are missing entries in a matrix. The "ratings" data is in a "sparse matrix"/dictionary format, meaning that the dictionary keys are of the kind (user, item), and the dictionary value is the corresponding value. Not all pairs are in the matrix, indicating that that value is missing or at its default value.

There are 4 types of rating/interaction data:

- `Interaction` : a "1" indicates the user has interacted with the book at some point in the past, either by saying that they intend to read it, have read it, or have given it a rating. If it is missing, that means the user has not interacted with the book.

- `Explicit Rating` : explicit ratings. Numeric values indicate the ratings given. If it is missing, that means the user has not rated the book.
- `Rating_all_zero` : explicit ratings. Numeric values more than 0 indicate the ratings given. Now, we replace missing values from above with "zeros," so that there are no missing ratings.
- `Rating_interaction_zero` : explicit ratings. Numeric values more than 0 indicate the ratings given. Now, we replace missing values from above with "zeros," only if the user interacted with that book in the past.

In [2]:
```
ratings_interactions, book_vectors_interactions, user_vectors_interactions = load_ratings_and_factors(type_name = 'interactio
ratings_explicit, book_vectors_explicit, user_vectors_explicit = load_ratings_and_factors(type_name = 'rating')
ratings_allmissing0, book_vectors_allmissing0, user_vectors_allmissing0 = load_ratings_and_factors(type_name = 'rating_all_ze
ratings_interact0, book_vectors_interact0, user_vectors_interact0 = load_ratings_and_factors(type_name = 'rating_interaction_

```

In [3]:
```
def get_shapes_and_ranges(ratings, book_vectors, item_vectors):
    print(len(ratings), np.shape(book_vectors), np.shape(item_vectors), min(ratings.values()), max(ratings.values()))
```

In [4]:
```
get_shapes_and_ranges(ratings_interactions, book_vectors_interactions, user_vectors_interactions)
get_shapes_and_ranges(ratings_explicit, book_vectors_explicit, user_vectors_explicit)
get_shapes_and_ranges(ratings_allmissing0, book_vectors_allmissing0, user_vectors_allmissing0)
get_shapes_and_ranges(ratings_interact0, book_vectors_interact0, user_vectors_interact0)
```

```
12238 (200, 10) (1000, 10) 1 1
8324 (200, 10) (1000, 10) 1 5
200000 (200, 10) (1000, 10) 0 5
12238 (200, 10) (1000, 10) 0 5
```

Problem 1: Predictions and recommendations with different data types

1a) What do different data types mean?

What is `Rating_interaction_zero` trying to capture -- why would we fill in books that someone interacted with but did not rate as a 0? (Hint: connect to conceptual reading from HW1). Answer in no more than 3 sentences.

We want to add the factor of non-response to postively skewed ratings. If someone did not leave a negative rating because its "costly" we need to balance this opinion. In the last reading, the authors developed the EPP effective positive proposensity rating which had a wider distribution and proved to be statistically significant for determining buyer experience. In our case adding the rating of 0, will counter-balance positvely skewed ratings.

What are some potential problems you see with using `rating_all_zero` for recommendations? Answer in no more than 3 sentences.

Numeric ratings are not always accurate because they can be heavily skewed or even inadvertently flipped (Someone thinks 1 is best, 5 is worst). The numeric rating of 0 might not directly relate to the user having a bad experience. Since we are making this assumption, it may bias our recommendation system.

1b) Generating predictions

Fill in the following function that takes in a user matrix (where each row is 1 user vector) and an item matrix (where each row is 1 item vector), and returns a matrix of predicted ratings for each user and item, where each entry is associated with the corresponding user (row number) and item (column number)

In [5]:
```
def get_predictions(user_vectors, book_vectors):
    return np.dot(user_vectors,book_vectors.T) #formula to calculate the prediction matrix
```

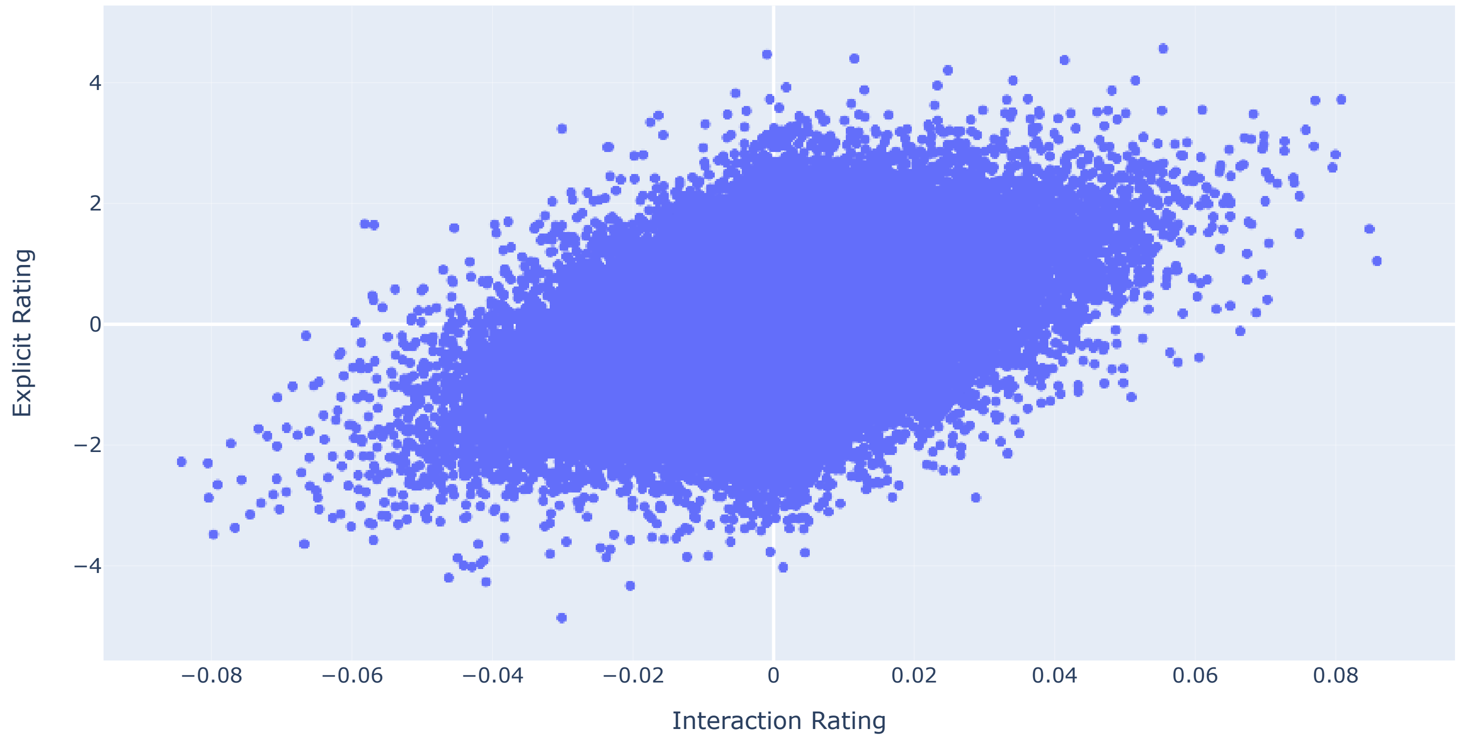
Output the predictions for first 10 items for the first user, using each of the 4 data types

In [6]:
```
#Calling get_predictions for all data
predicted_interactions = get_predictions(user_vectors_interactions,book_vectors_interactions)
predicted_explicit = get_predictions(user_vectors_explicit,book_vectors_explicit)
predicted_allmissing0 = get_predictions(user_vectors_allmissing0,book_vectors_allmissing0)
predicted_interact0 = get_predictions(user_vectors_interact0,book_vectors_interact0)
print("-----------ratings_interactions predictions for first user--------------- \n", get_predictions(user_vectors_interactio
print("-----------ratings_explicit predictions for first user------------------ \n", get_predictions(user_vectors_explicit,boo
print("-----------ratings_allmissing0 predictions for first user--------------- \n", get_predictions(user_vectors_allmissing0
print("-----------ratings_interact0 predictions for first user----------------- \n", get_predictions(user_vectors_interact0,bo
```

```
-----------ratings_interactions predictions for first user---------------
 [-0.00268685  0.01007861  0.00173543 -0.0009836   0.0029315   0.00733778
 -0.01030278  0.00722029  0.00110368  0.00335202]

-----------ratings_explicit predictions for first user------------------
 [ 0.07256422  0.40674721  0.07614967 -0.0750923   0.17636483  0.95914661
 -0.23892582 -0.00221766 -0.91204988  0.65839933]

-----------ratings_allmissing0 predictions for first user---------------
 [ 0.08031172  0.28975416  0.06345163  1.5695302  -0.18567229  0.05502522
  0.01107722 -0.08822294 -0.89503553 -0.01175208]

-----------ratings_interact0 predictions for first user-----------------
 [-1.00879311  4.16663033 -0.91856862 -0.30194806 -2.52606751 -0.18905701
 -3.08103904 -0.53718459 -1.18618829 -1.21838417]
```

Do a scatterplot of the predicted rating for the "interaction" and "explicit ratings" types. (Each dot represents one user and one book, with X axis being predicted ratings using interaction data and Y axis being predicted rating using explicit ratings). Describe what you see in no more than 2 sentences.

In [7]:
```python
import plotly.express as px
import pandas as pd
d = {'Interaction Rating':predicted_interactions.flatten(),'Explicit Rating':predicted_explicit.flatten()}
df = pd.DataFrame(d, columns=['Interaction Rating','Explicit Rating'])
fig = px.scatter(df, x='Interaction Rating', y='Explicit Rating')
fig.show()
```



In this scatter plot we are comparing the interaction rating to the explicit rating for each (user, item) pair. Interaction and Explicit ratings are positively correlated indicating those who responded usually left higher ratings versus those who did not interact left lower ratings.

1c) From predictions to recommendations (without capacity constraints)

Fill in the following function that takes in the matrix predicted ratings for each user and item, returns a dictioanry where the keys are the user indices and the values are a list of length "number_top_items" indicating the recommendations given to that user

In [8]:

```python
def get_recommendations_for_each_user(predictions, number_top_items = 10):
  dict_top10 = {}
  userIx = 0
  for x in predictions: # for each row in the prediction matrix
      lst = list(np.argsort(x)[::-1][:number_top_items]) #sort the row greatest to least, keep the index,and top n items
      dict_top10.update({userIx:lst}) #append the user and its top n recommendations to dict
      userIx += 1
  return dict_top10 #return dict
```

Output the recommendations for the first user, using each of the 4 data types

In [9]:

```python
#Call the get_recommendations for each user for all data sets
rec_interactions = get_recommendations_for_each_user(predicted_interactions)
rec_explicit = get_recommendations_for_each_user(predicted_explicit)
rec_allmissing0 = get_recommendations_for_each_user(predicted_allmissing0)
rec_interact0 = get_recommendations_for_each_user(predicted_interact0)
print("-----------ratings_interactions Top 10 recommendations for first user--------------- \n", list(get_recommendations_for_
print("-----------ratings__explicit Top 10 recommendations for first user--------------- \n", list(get_recommendations_for_ea
print("-----------ratings_allmissing0 Top 10 recommendations for first user--------------- \n", list(get_recommendations_for_
print("-----------ratings_interact0 Top 10 recommendations for first user--------------- \n", list(get_recommendations_for_ea
```

```
-----------ratings_interactions Top 10 recommendations for first user---------------
 (0, [74, 164, 199, 30, 73, 104, 100, 19, 198, 182])

-----------ratings__explicit Top 10 recommendations for first user---------------
 (0, [30, 164, 121, 87, 199, 116, 108, 124, 5, 93])

-----------ratings_allmissing0 Top 10 recommendations for first user---------------
 (0, [57, 55, 56, 81, 50, 78, 58, 86, 77, 96])

-----------ratings_interact0 Top 10 recommendations for first user---------------
 (0, [166, 53, 1, 111, 170, 74, 182, 37, 73, 52])
```

Fill in the following function that takes in the recommendations for each user and item, and outputs a histogram for how often each item is to be recommended. For example, if there are 18 items, and 10 of them were never recommended, 5 of them were recommended once each, and 3 of them were recommended five times each, then you would have bars at 0, 1, and 5, of height 10, 5, and 3, respectively.

In [10]:

```python
def show_frequency_histograms(recommendations): #gets frequency of the recommendations and creates a new dictionary
    a = list(range(0,200))
    d_items = {el:0 for el in a} #dict of items with a frequency set to 0
    for item in d_items.keys():
        count = 0
        for user in recommendations.keys():
            if item in recommendations[user]: #if item was recomended update the d_items dictionary with a counter
                count +=1
                d_items.update({item:count})
    return d_items #return the dict of items with a frequency of each item
```

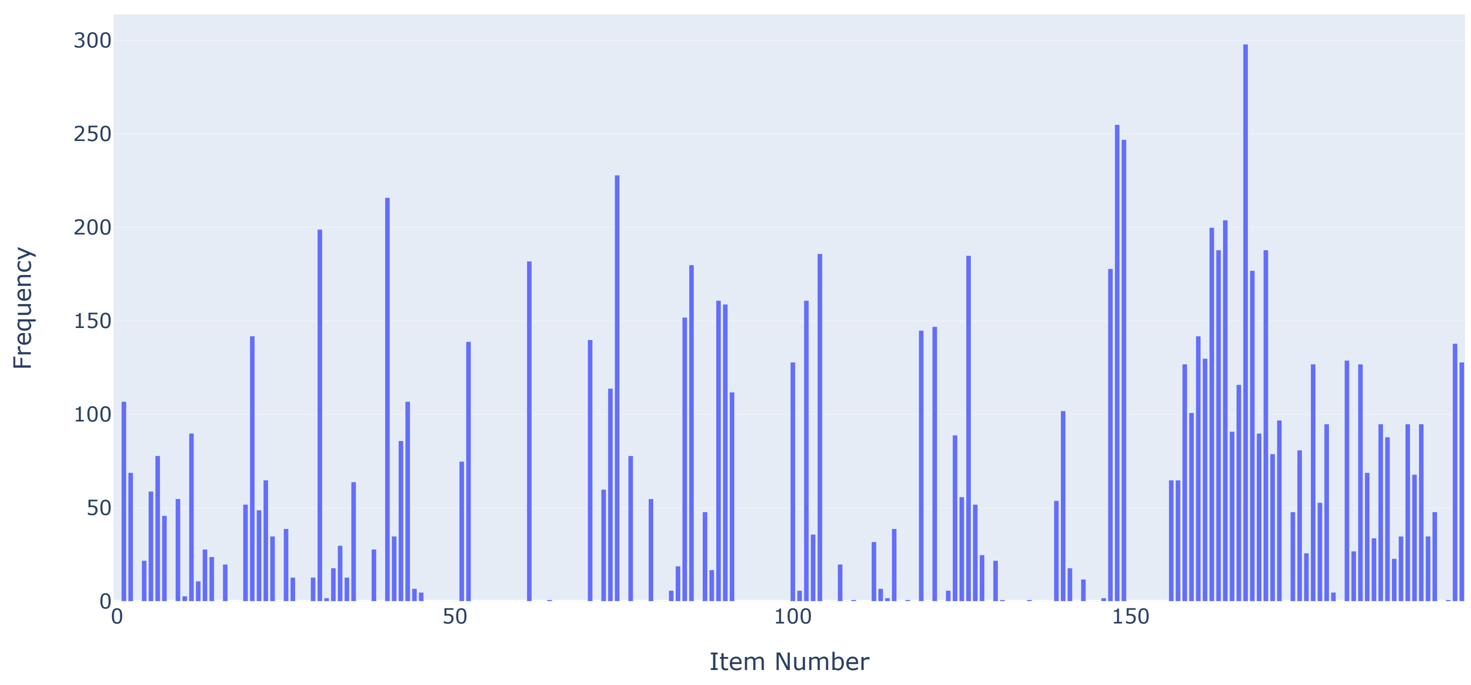
Show the histograms for the "interact" and "explicit" data types. Describe what you observe in no more than 3 sentences. For example, discuss how often is the most recommended item recommended, how that compares to the least recommended items, and what that could mean for recommendations in various contexts.

Plotting Results of the Top 10 Recommended Items

In [11]:

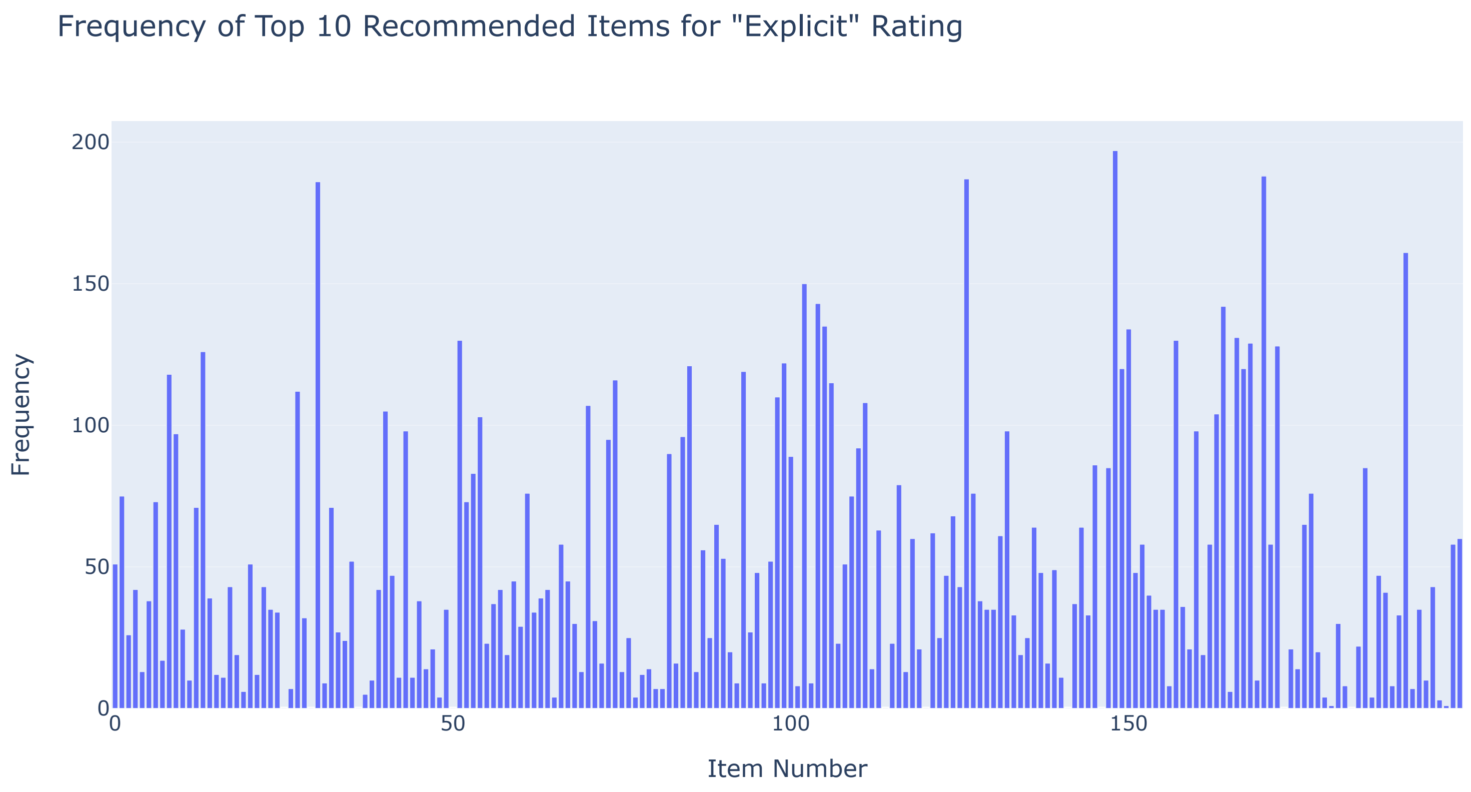
```
d_interact = show_frequency_histograms(rec_interactions)
d = {'Item Number':list(d_interact.keys()),'Frequency':list(d_interact.values())}
df = pd.DataFrame(d, columns=['Item Number','Frequency'])
fig = px.bar(df, x='Item Number', y='Frequency', title='Frequency of Top 10 Recommended Items for "Interact" Rating')
fig.show()
```

Frequency of Top 10 Recommended Items for "Interact" Rating



The most recommended item from the "interact" ratings was item # 167, 148, 149. The least recommended items are item # 180, 64, 146. The difference in frequency between the highest and lowest recommended items is 296 and if this system is for physical books there might be capacity issues wheras in a digital media context, the difference does not matter when there are no capacity constraints.

In [12]:
```python
d_explicit = show_frequency_histograms(rec_explicit)
d = {'Item Number':list(d_explicit.keys()),'Frequency':list(d_explicit.values())}
df = pd.DataFrame(d, columns=['Item Number','Frequency'])
fig = px.bar(df, x='Item Number', y='Frequency', title='Frequency of Top 10 Recommended Items for "Explicit" Rating')
fig.show()
```



The most recommended item from the "Explicit" ratings predictions was item # 126, 148, 170. The least recommended items are item # 77, 26, 179. The difference in frequency between the highest and lowest recommended items is 193 and if this system is for physical books there might be capacity issues wheras in a digital media context, the difference does not matter when there are no capacity constraints.

Problem 2: Cold start -- recommendations for new users

In this part of the assignment, we are going to ask you to tackle the "cold-start" problem with matrix-factorization based recommendation systems. The above recommendation techniques worked when you had access to past data for reach user, such as interactions or explicit ratings. However, it doesn't work as well when a new user has just joined the platform and so the platform doesn't have any data.

You should also see a comma-separated values file (user_demographics.csv) that contains basic demographic information on each user. Each row describes one user, and have four attributes: 'User ID', 'Wealth', 'Age group' and 'Location'.

User ID is the unique identifier associated with each user, and it is in the same order as the user_vectors, and in the same indexing as the ratings (be careful about 0 and 1 indexing in Python).

Wealth is a non-negative, normalized value indicating the average wealth of the neighborhood in which the user is, where we normalized it such that each Location has similar wealth distributions. Age group describes the age of the user. Location describes the region that the user is from.

In [13]:
```
demographics = pd.read_csv("data/user_demographics.csv")
demographics.head()
```

Out[13]:

	User ID	Wealth	Age group	Location
0	1	1.833101	50 to 64	America
1	2	2.194996	18 to 34	America
2	3	2.216195	18 to 34	Europe
3	4	0.838690	50 to 64	Asia Pacific
4	5	2.109313	18 to 34	America

We are now going to pretend that we don't have the personalized ratings/interactions history for the last 100 users, and thus don't have their user vectors. Rather, let's pretend that these are new users to the platform, and you are able to get the above demographics from their browswer cookies/IP address. Now, we're going to try to recommend items for them anyway. For this part, we'll exclusively use the "ratings with interaction0" data.

In [53]:
```
existing_user_vectors = user_vectors_interact0[0:900,:]
existing_user_demographics = demographics.iloc[0:900,:]
new_user_demographics = demographics.iloc[900:,:]
```

2a) Predictions for new users [Simple]

Fill in the following function that takes in: the demographics of a single new user, the demographics of all the existing users in your platform, and the user vectors of all the existing users, and outputs a 'predicted' user vector for the new user to use until we get enough data for that user.

For this question, we ask you to use the following simple method to construct the vector for the new user. Each user is classified as "Low" or "High" wealth based on whether their Wealth score is below or above the median of 1.70. Then, we simply construct a mean user vector for "Low" and "High" wealth, based on the 900 users (take the average vector among users with "Low" and "High" Wealth, respectively.). The correpsonding mean vector is then used for each new user.

In [15]:
```python
existing_user_demographics.Wealth.median()
```

Out[15]:
```
1.7026180771992308
```

In [16]:
```python
#Helper function to calculate the prediction vector for a new user
#check is a boolean to distinguish between high and low
def get_predicted_vect(check,existing_user_demographics, existing_user_vectors):
    #if check is set to true, type is wealthy
    if check:
        #filter the existing user demographic frame where the Wealth is greater than the median
        df = existing_user_demographics[existing_user_demographics.Wealth > existing_user_demographics.Wealth.median()]
    else:
        #filter the existing user demographic frame where the Wealth is less than the median
        df = existing_user_demographics[existing_user_demographics.Wealth < existing_user_demographics.Wealth.median()]
    #get the User IDs for this df but subtract 1 to make sure its python indexing
    filter_indices=list(df['User ID'] - 1)
    #filter the existing user vector by these indices
    filteredArray = np.take(existing_user_vectors, filter_indices, axis=0)
    #average the filtered array over each column
    pred_vict = np.mean(filteredArray, axis=0)
    return pred_vict #return the predicted vector for that type of user
```

In [17]:
```python
# function that takes in the new_users and adds the predicted vector for that user calls get predicted vect helper function
def get_user_vector_for_new_user(new_user, existing_user_demographics, existing_user_vectors):
    #get the low vect and high vect by calling the helper functions
    low_vect = get_predicted_vect(False,existing_user_demographics, existing_user_vectors)
    high_vect = get_predicted_vect(True,existing_user_demographics, existing_user_vectors)
    #make a copy of the new_user frame to avoid python warnings
    new_user1 = new_user.copy()
    #Adding in a column to indicate if new user wealth is high or low
    new_user1['Wealth Score'] = np.where(new_user['Wealth'] > existing_user_demographics.Wealth.median(), 'High', 'Low')
    #Adding the low or high vect depending on Wealth attribute High or low
    new_user1['Predicted Wealth Vector'] = new_user1['Wealth Score'].map({'High':high_vect,'Low':low_vect})
    #Return the Data Frame
    return new_user1
```

Output the mean vector predicted for the first user in `new_user_demographics`.

In [18]:
```python
new_user2 = get_user_vector_for_new_user(new_user_demographics,existing_user_demographics,existing_user_vectors)
print('User 1 Predicted Vector \n',new_user2.iloc[0,4],list(new_user2.iloc[0,5]))
```

```
User 1 Predicted Vector
 Low [0.08780943150811217, -0.20709913353749781, -0.17762736410299737, -0.13094589164366768, -0.08942902403642827, -0.12702041629
128474, -0.12242307839294987, -0.4268105867406346, -0.057752848850433, -0.10569089870005266]
```

2b) [Bonus, 3 points] Predictions for new users [Using KNN or another model]

Fill in the following function that takes in: the demographics of a single new user, the demographics of all the existing users in your platform, and the user vectors of all the existing users, and outputs a 'predicted' user vector for the new user to use until we get enough data for that user.

Now, use K nearest neighbors or some other machine learning method.

Feel free to prepare data/train a model outside this function, and then use your trained model within the function.

Preprocessing the Data

- Maps the Ages and Locations to values [1,2,3,4] for both the existing user demographics and new user demographics
- Averages the Age and Location into a new column
- Drops all non informative columns (User Id, Age Group, Location)

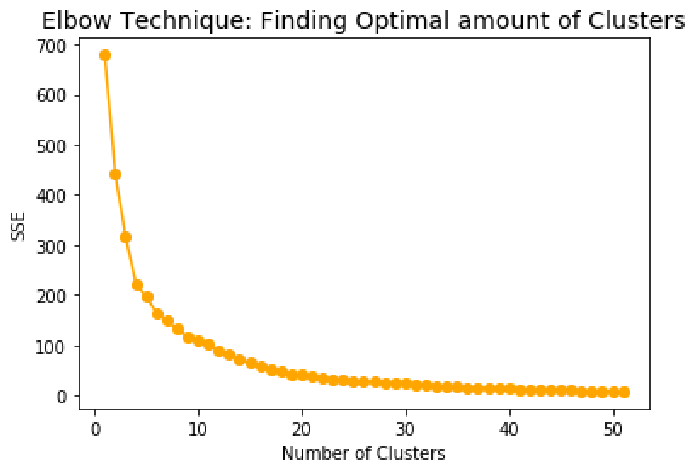
In [41]:
```python
#creating dictionary mappings
keysAge = ['18 to 34', '35 to 49', '50 to 64', '65 and older']
dictionaryAge = dict(zip(keysAge, [1,2,3,4]))
keyslocation = ['America', 'Europe', 'Asia Pacific', 'Africa']
dictionaryLocation = dict(zip(keyslocation, [1,2,3,4]))
#applying mappings to the existing user demographics
df = existing_user_demographics.copy()
df['Age group'] = df['Age group'].map(dictionaryAge)
df['Location'] = df['Location'].map(dictionaryLocation)
df['Age group + Location'] = (df['Age group'] + df['Location']) / 2
df = df.drop(['User ID','Age group','Location'],axis=1)
#applying mappings to the new user demographics
df2 = new_user_demographics.copy()
df2['Age group'] = df2['Age group'].map(dictionaryAge)
df2['Location'] = df2['Location'].map(dictionaryLocation)
df2['Age group + Location'] = (df2['Age group'] + df2['Location']) / 2
df2 = df2.drop(['User ID','Age group','Location'],axis=1)
```

Finding Optimal Amount of Clusters with Elbow Method

- Using K Means we want to find the optimal amount of clusters that minimize the SSE. Here I wrote code to iteratively calculate the best K. Unfortuanetly K > 10 is good but makes future code harder so I stuck with K = 5

In [42]:

```
import sklearn.cluster
import sklearn.manifold
import matplotlib
import warnings
warnings.filterwarnings("ignore")

K_df = df.iloc[:,:2]
sse = {} #Create an empty dictionary
KVect = pd.DataFrame(columns=['K','Inertia']) #Create a new DataFrame
for k in range(1, 52): #clusters between 1 and 52 to see optimal cluster amount in elbow technique
    kmeansNew = sklearn.cluster.KMeans(n_clusters=k,n_init=3, init='k-means++',max_iter=20).fit(K_df) #Run the Kmeans functio
    K_df["clusters"] = kmeansNew.labels_ #Grab the cluster label and store in the dataframe
    sse[k] = kmeansNew.inertia_ # Inertia: Sum of distances of samples to their closest cluster center
    KVect = KVect.append({'K':k,'Inertia':sse[k]},ignore_index=True) #Append each amount of cluster K with the Inertia

#Plotting Results for Visual Purposes
matplotlib.pyplot.figure()
matplotlib.pyplot.plot(list(sse.keys()), list(sse.values()),marker='o',color='orange')
matplotlib.pyplot.title('Elbow Technique: Finding Optimal amount of Clusters',fontsize=14) #Put a big title
matplotlib.pyplot.xlabel("Number of Clusters") #Set the xlabel
matplotlib.pyplot.ylabel("SSE") #Set y label
matplotlib.pyplot.show() #Show plot
KVect.head(10)
```



Out[42]:

	K	Inertia
0	1.0	679.438841
1	2.0	442.503746
2	3.0	317.651121

	K	Inertia
3	4.0	222.337246
4	5.0	196.681456
5	6.0	163.625757
6	7.0	148.884584
7	8.0	131.228480
8	9.0	117.025966
9	10.0	108.453600

We can see the Inertia goes down for K = 10 but for simplicity we will go with K = 5

Running K Means Clustering for K = 5

In [43]:

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=5) #fit the existing df with 5 clusters
kmeans.fit(df.iloc[:,:2])
labels = kmeans.predict(df.iloc[:,:2]) #create the labels
centroids = kmeans.cluster_centers_
# color data points based on cluster labels
colmap = {0:'r', 1:'orange', 2: 'b',3:'c',4:'g',5:'k',6:'y'} #Create a color map dictionary
df['Cluster'] = list(labels)
df['color'] = df.Cluster.map(colmap)
```

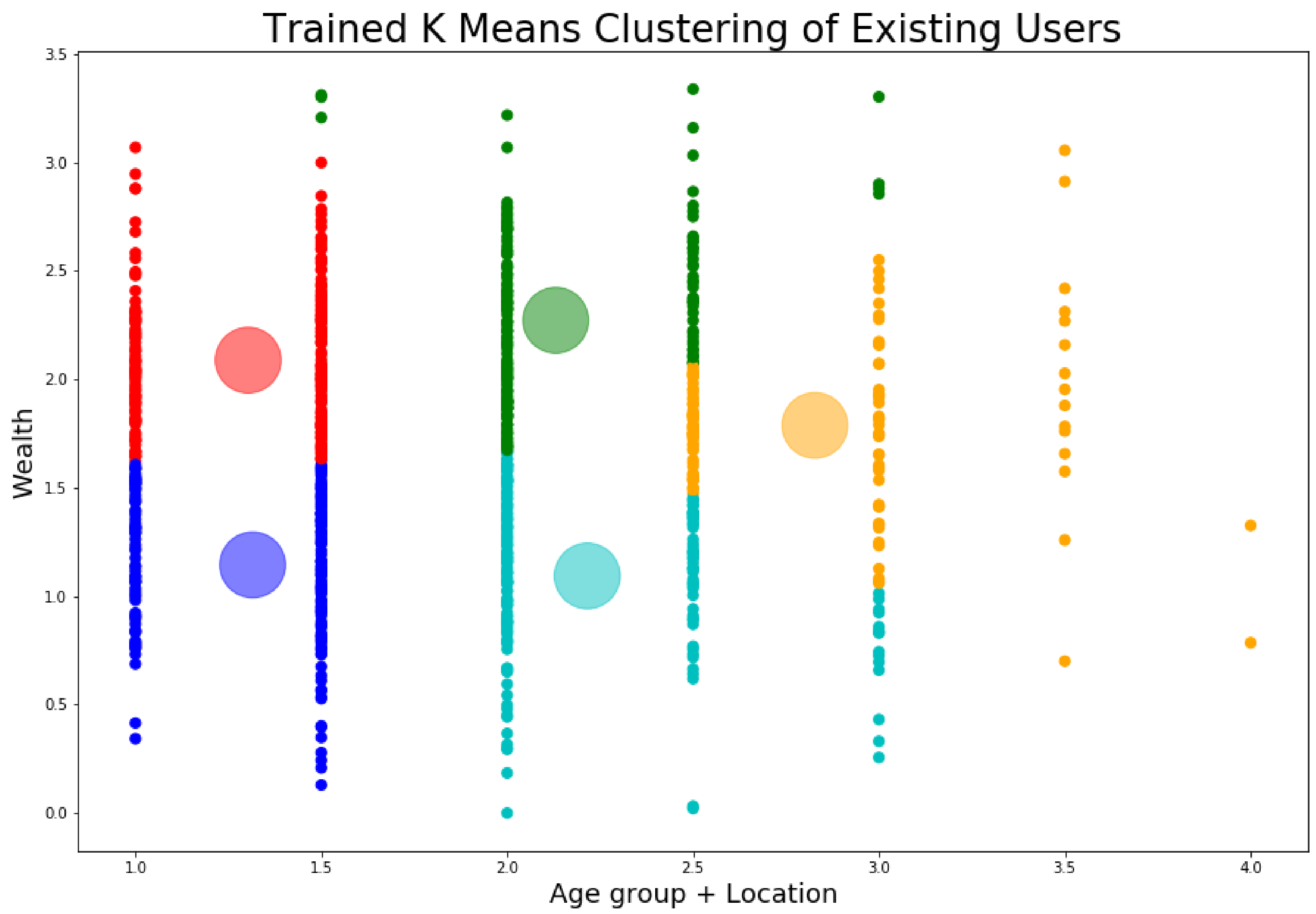
Visualizing the Clusters and their Centroids on the Training Set

In [44]:

```python
import matplotlib
p = df.plot.scatter(x='Age group + Location', y='Wealth', s=50, color=df['color'],figsize=(15,10),sharex=False) #Plot the fig

clusterDF = pd.DataFrame(columns=['Age group + Location', 'Wealth'])

for cluster in set(df.Cluster): #Find the clusters centroids based of the average of each cluster 0-4
    clusterDF = clusterDF.append(df[df.Cluster == cluster].mean(),ignore_index=True)
clusterDF.Cluster = clusterDF.Cluster.map(int) #Convert the clusters to int
clusterDF['color'] = clusterDF.Cluster.map(colmap) #Add the colors
matplotlib.pyplot.scatter(x=clusterDF['Age group + Location'],y=clusterDF['Wealth'],s=2000,color=clusterDF['color'],alpha=0.5
p.set_title('Trained K Means Clustering of Existing Users',fontsize=26) #Put a big title
p.set_xlabel("Age group + Location", fontsize=18) #Put a x label
p.set_ylabel("Wealth", fontsize=18) #Put a y label
```

Out[44]:

```
Text(0, 0.5, 'Wealth')
```



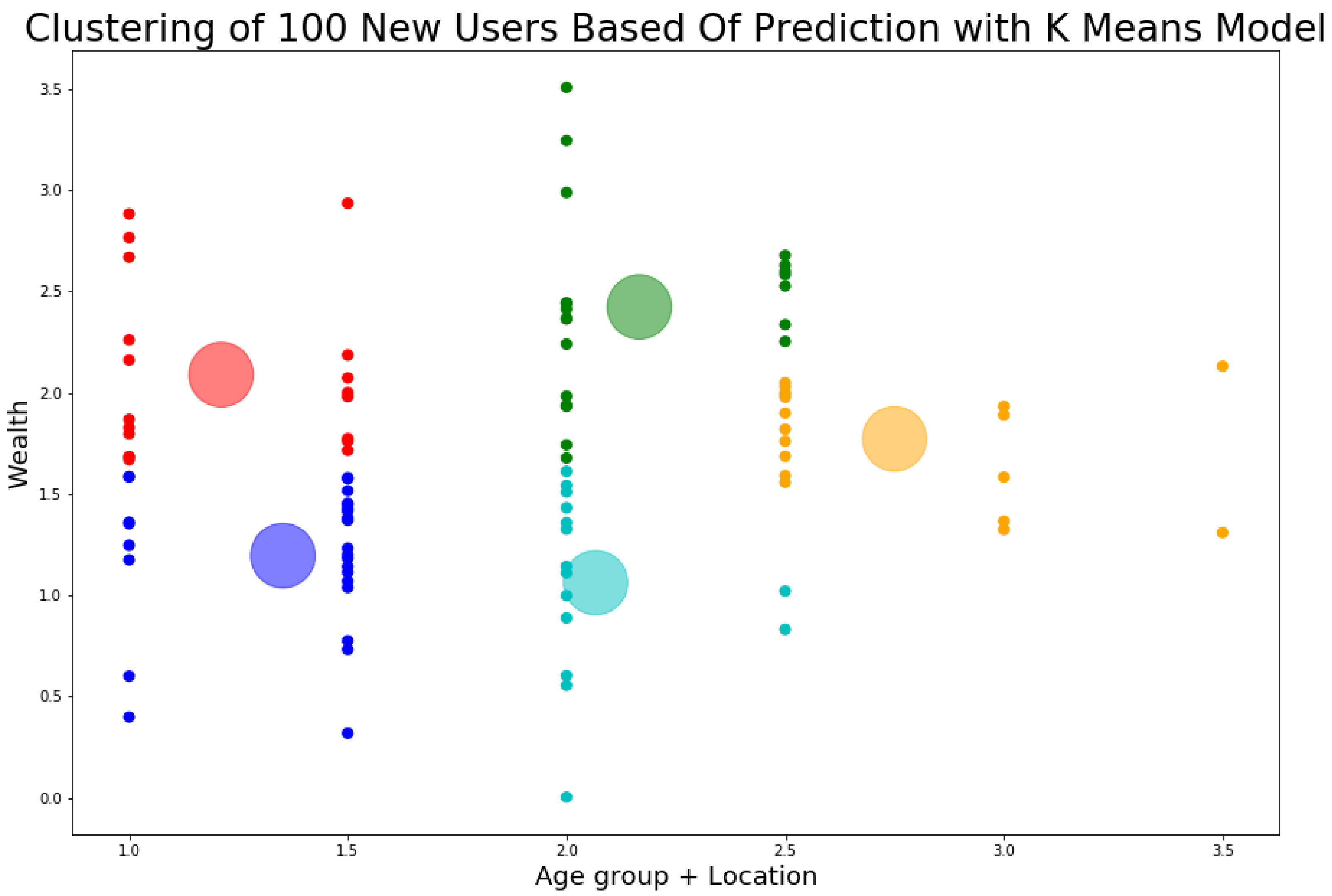
Applying K Means Clustering on the Test Set (100 New Users)

In [45]:

```python
labels = kmeans.predict(df2.iloc[:,:2]) #create the labels for New Users
centroids = kmeans.cluster_centers_ #retrieve centroids
# color data points based on cluster labels
colmap = {0:'r', 1:'orange', 2: 'b',3:'c',4:'g',5:'k',6:'y'} #Create a color map dictionary
df2['Cluster'] = list(labels) #assign labels to df2 users
df2['color'] = df2.Cluster.map(colmap) #assign colors for plotting
import matplotlib
p = df2.plot.scatter(x='Age group + Location', y='Wealth', s=50, color=df2['color'],figsize=(15,10),sharex=False) #Plot the f
clusterDF = pd.DataFrame(columns=['Age group + Location', 'Wealth'])

for cluster in set(df2.Cluster): #Find the clusters centroids based of the average of each cluster 0-4
    clusterDF = clusterDF.append(df2[df2.Cluster == cluster].mean(),ignore_index=True)
clusterDF.Cluster = clusterDF.Cluster.map(int) #Convert the clusters to int
clusterDF['color'] = clusterDF.Cluster.map(colmap) #Add the colors
clusterDF
matplotlib.pyplot.scatter(x=clusterDF['Age group + Location'],y=clusterDF['Wealth'],s=2000,color=clusterDF['color'],alpha=0.5
p.set_title('Clustering of 100 New Users Based Of Prediction with K Means Model',fontsize=26) #Put a big title
p.set_xlabel("Age group + Location", fontsize=18) #Put a x label
p.set_ylabel("Wealth", fontsize=18) #Put a y label
```

Out[45]:

```
Text(0, 0.5, 'Wealth')
```



Specifying a Helper function to calculate Predicted Vector Based on Cluster Assignment

In [46]:
```
#Helper function to calculate the prediction vector for a new user
def get_predicted_vect_kmeans(check,existing_df, existing_user_vectors):
    #if check is set to true
    df = existing_df[existing_df.Cluster == check]
    #get the User IDs for this df but subtract 1 to make sure its python indexing
    filter_indices=list(df['User ID'] - 1)
    #print(filter_indices)
    #filter the existing user vector by these indices
    filteredArray = np.take(existing_user_vectors, filter_indices, axis=0)
    #average the filtered array over each column
    pred_vict = np.mean(filteredArray, axis=0)
    return pred_vict
```

Getting the User Vector for New Users based on K Means Cluster Assignment

In [47]:
```
def get_user_vector_for_new_user_kmeans(new_user_df, existing_df, existing_user_vectors):
    #get the cluster vect by calling the helper functions
    Zero_vect = get_predicted_vect_kmeans(0,existing_df, existing_user_vectors)
    One_vect = get_predicted_vect_kmeans(1,existing_df, existing_user_vectors)
    Two_vect = get_predicted_vect_kmeans(2,existing_df, existing_user_vectors)
    Three_vect = get_predicted_vect_kmeans(3,existing_df, existing_user_vectors)
    Four_vect = get_predicted_vect_kmeans(4,existing_df, existing_user_vectors)

    #make a copy of the new_user frame
    new_user1 = new_user_df.copy()
    #Adding the low or high vect depending on Wealth attribute High or low
    new_user1['Predicted K Means Vector'] = new_user1['Cluster'].map({0:Zero_vect,1:One_vect,2:Two_vect,3:Three_vect,4:Four_v
    #Return the Data Frame
    return new_user1
```

Output the mean vector predicted for the first user in `new_user_demographics` .

In [48]:
```
dfnew = existing_user_demographics.copy()
dfnew['Cluster'] = df['Cluster'] #adding cluster assignments to existing demographics
usernew = new_user_demographics.copy()
usernew['Cluster'] = df2['Cluster'] #adding cluster assignments to the new user dempgrahics
new_user3 = get_user_vector_for_new_user_kmeans(usernew,dfnew,existing_user_vectors)
print('User 1 Predicted Vector \n','Assignment Cluster ', new_user3.iloc[0,4], '\n',list(new_user2.iloc[0,5]))
```

```
User 1 Predicted Vector
 Assignment Cluster  2
 [0.08780943150811217, -0.20709913353749781, -0.17762736410299737, -0.13094589164366768, -0.08942902403642827, -0.127020416291284
74, -0.12242307839294987, -0.4268105867406346, -0.057752848850433, -0.10569089870005266]
```

In [51]:
```
#showing the new user vectors from K Means and their Cluster Assignment
new_user3.head(10)
```

Out[51]:

	User ID	Wealth	Age group	Location	Cluster	Predicted K Means Vector
900	901	0.731783	35 to 49	America	2	[0.1566815768233348, -0.1312700821333831, -0.1...
901	902	2.439378	50 to 64	America	4	[-0.22794095547114254, -0.13064465113444274, -...
902	903	1.760649	50 to 64	Europe	1	[-0.0871719401474109, -0.208499273219674, -0.1...
903	904	2.582320	50 to 64	Europe	4	[-0.22794095547114254, -0.13064465113444274, -...
904	905	1.888599	65 and older	Europe	1	[-0.0871719401474109, -0.208499273219674, -0.1...
905	906	2.000803	50 to 64	Europe	1	[-0.0871719401474109, -0.208499273219674, -0.1...
906	907	1.382386	18 to 34	Europe	2	[0.1566815768233348, -0.1312700821333831, -0.1...
907	908	1.113202	18 to 34	Europe	2	[0.1566815768233348, -0.1312700821333831, -0.1...
908	909	1.557470	35 to 49	Asia Pacific	1	[-0.0871719401474109, -0.208499273219674, -0.1...
909	910	2.677586	50 to 64	Europe	4	[-0.22794095547114254, -0.13064465113444274, -...

Justify your choice of model. If you used K nearest neighbors, then how did you decide upon your distance function? If you used another model, how does that model weight the different demographics in importance (either implicitly or explicitly)?

I used the K Means model with K = 5 Clusters. The model learned clusters based on the ordered numeric values of (Age + Location) / 2 and corresponding Wealth value. It explicitly created clusters when the Wealth above or below ~1.7 (Median). It also created a division based on the younger people in America (mean of 1) and older people in Africa mean of 4.

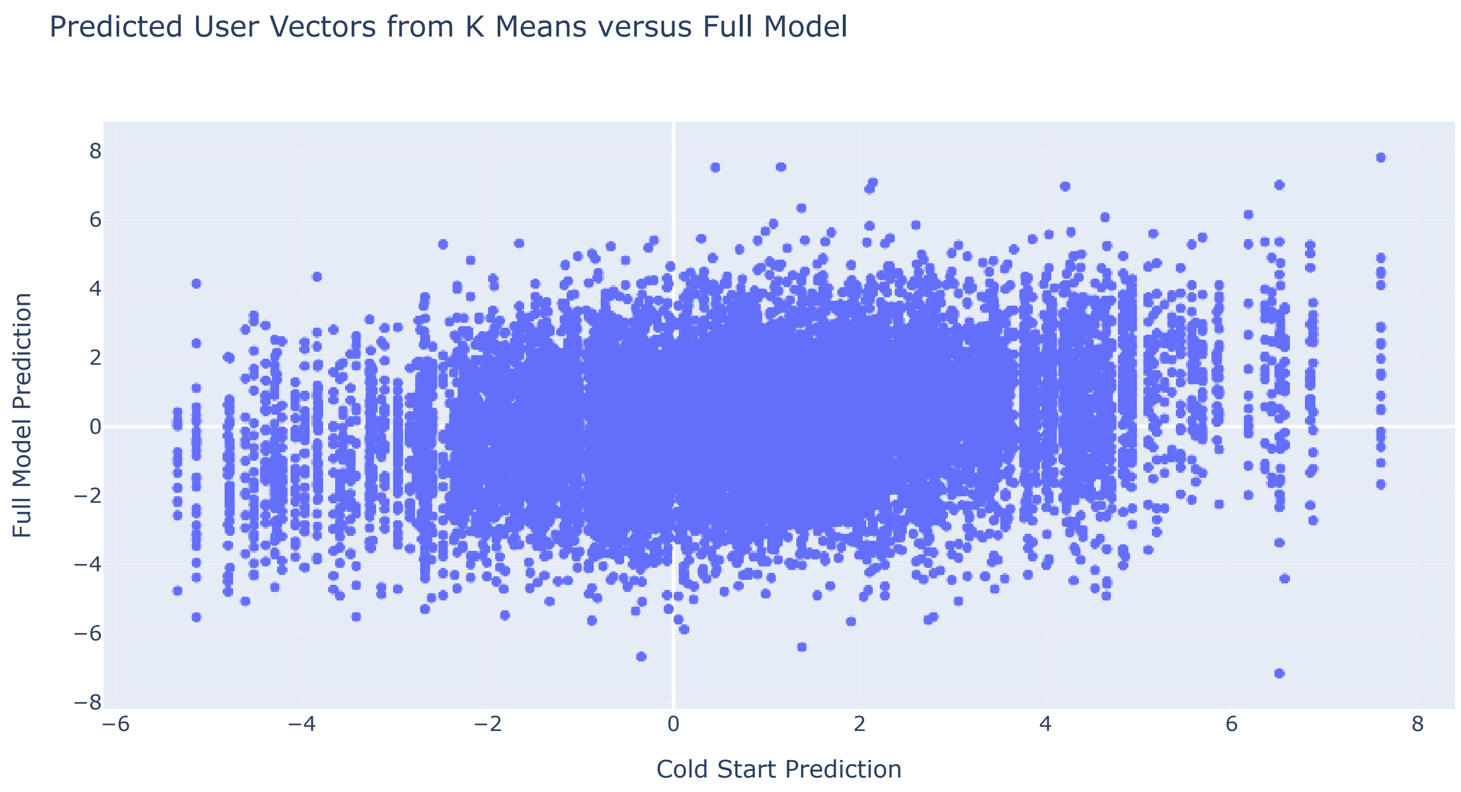
2c) Comparing predictions from from "true" user vector and from above

For each of the 100 "new" users, use either your model from 2a or 2b ("demographic model") to retrieve a user vector for that user, and then your functions from Problem 1 to get predicted ratings and top-10 recommendations. First, plot a scatterplot between the ratings predicted by the demographic model and the ratings predicted by the full model from Problem 1. Each point in the scatter plot should correspond to one user and one item, and so your scatterplot should have 100*200 points.

Now for each user, calculate the mean rating (according to the "full" model in Problem 1) for the 10 items recommended to that user, by each of the demopgraphic and "full" models. Output a scatterplot for the two mean ratings, where each point correpsonds to 1 user.

In [102]:

```

cold_start_users = np.stack(new_user3['Predicted K Means Vector'].values * 5)
cold_start_users
#Unstack the Predicted Vectors from K Means
last100 = user_vectors_interact0[-100:] #Filter the original users to last 100
predicted_full_model = get_predictions(last100,book_vectors_interact0) #get the predicted model based on original user vector
cold_start_predicted_interact0 = get_predictions(cold_start_users,book_vectors_interact0) #get predicted model based on K Mea
#Plot Results
d = {'Cold Start Prediction':cold_start_predicted_interact0.flatten(),'Full Model Prediction':predicted_full_model.flatten()}
df = pd.DataFrame(d, columns=['Cold Start Prediction','Full Model Prediction'])
fig = px.scatter(df, x='Cold Start Prediction', y='Full Model Prediction', title='Predicted User Vectors from K Means versus
fig.show()
```



Now for each user, calculate the mean rating (according to the "full" model in Problem 1) for the 10 items recommended

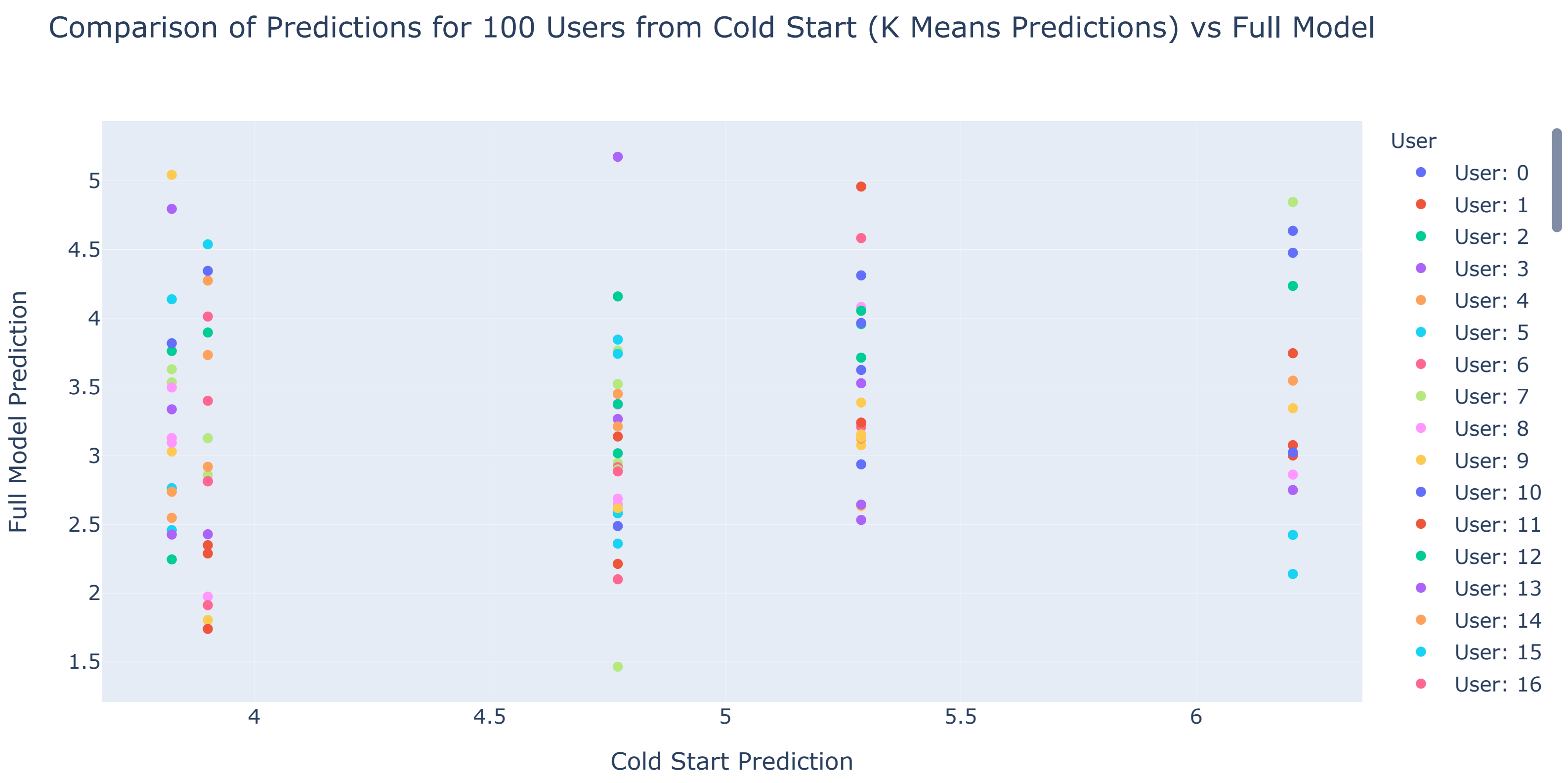
to that user, by each of the demopgraphic and "full" models. Output a scatterplot for the two mean ratings, where each point correpsonds to 1 user.

In [103]:

```python
#Function that calculates mean ratings for each user's top 10 recommendations
def get_recommendations_for_each_user2(predictions, number_top_items = 10):
    dict_top10 = {}
    userIx = 0
    for x in predictions:
        lst2 = list(np.sort(x)[::-1][:10]) #sort the list by the top 10 recommendations

        #calculates the mean rating for the top 10 recommendations
        dict_top10.update({'User: '+str(userIx):sum(lst2)/10})
        userIx += 1
    return dict_top10
```

In [105]:

```python
rec_full = get_recommendations_for_each_user2(predicted_full_model) #getting the mean rating for the full model
rec_cold = get_recommendations_for_each_user2(cold_start_predicted_interact0) #getting the mean ratings for the demographic m
d = {'Cold Start Prediction':list(rec_cold.values()),'Full Model Prediction':list(rec_full.values()), 'User':list(rec_full.ke
df = pd.DataFrame(d, columns=['Cold Start Prediction','Full Model Prediction','User'])
fig = px.scatter(df, x='Cold Start Prediction', y='Full Model Prediction', title='Comparison of Predictions for 100 Users fro
fig.show()
```



Comment on the above. What is the "loss" from using demogprahics since we do not have access to the full data?

Since we do not have access to the full data we notice that the predictions for cold start are grouping people into groups of ratings and does not distinguish whether or not they will have high or low ratings. It assigns a mean rating for each cluster instead of each person so when the person gets more data they will have their independent predicted rating. The loss is that cold start will not be as accurate as having the true user vectors and would want to see a positively correlated graph (Ideally r = 1).

Problem 3: Predictions under capacity constraints

Above, you should have observed that if we just recommend the top items for each user, some items get recommended quite a bit, and many items do not get recommended at all. Here, we are going to ask you to implement recommendations under capacity constraints.

Throughout this part, assume that you only have 5 copies of each item that you recommend, and that you will only recommend 1 item to each user. In other words, you cannot recommend the same item more than 5 times, and so there are exactly 1000 items in stock (representing 200 unique books) for your 1000 users.

We'll continue exclusively using the "ratings with interaction0" data.

Now, let's assume that users are entering the platform sequentially in order of index. So the index 0 user comes first, index 1 user comes second, etc.

3a) Naive recommendations under capacity constraints

First, let's pretend that we were naively recommending the predicted favorite item to each user. Of course, with unlimited capacity, each user would be recommended their predicted favorite. With capacity constraints, the favorite items of the users who come in later might already have reached their capacity, and so they have to be recommended an item further down their list.

Do the following: simulate users coming in sequentially, in order of index. For each user, recommend to them their predicted favorite item that is still available. So the first user will get their favorite item, but the last few users will almost certainly not receive any of their top few predicted items. For each user, keep track of what the rank of the item that they were ultimately recommended was, according to the predicting ranking over items for that user.

Plot the resulting rankings in 2 ways: 1) A line plot, where the X axis is the user index and the Y axis is the rank of the item that they were recommended. and 2) A histogram of how often each rank shows up. (the X axis is the (binned) rank, and the Y axis is the count of that bin).

Generate Simulation of Users Arriving

In [106]:

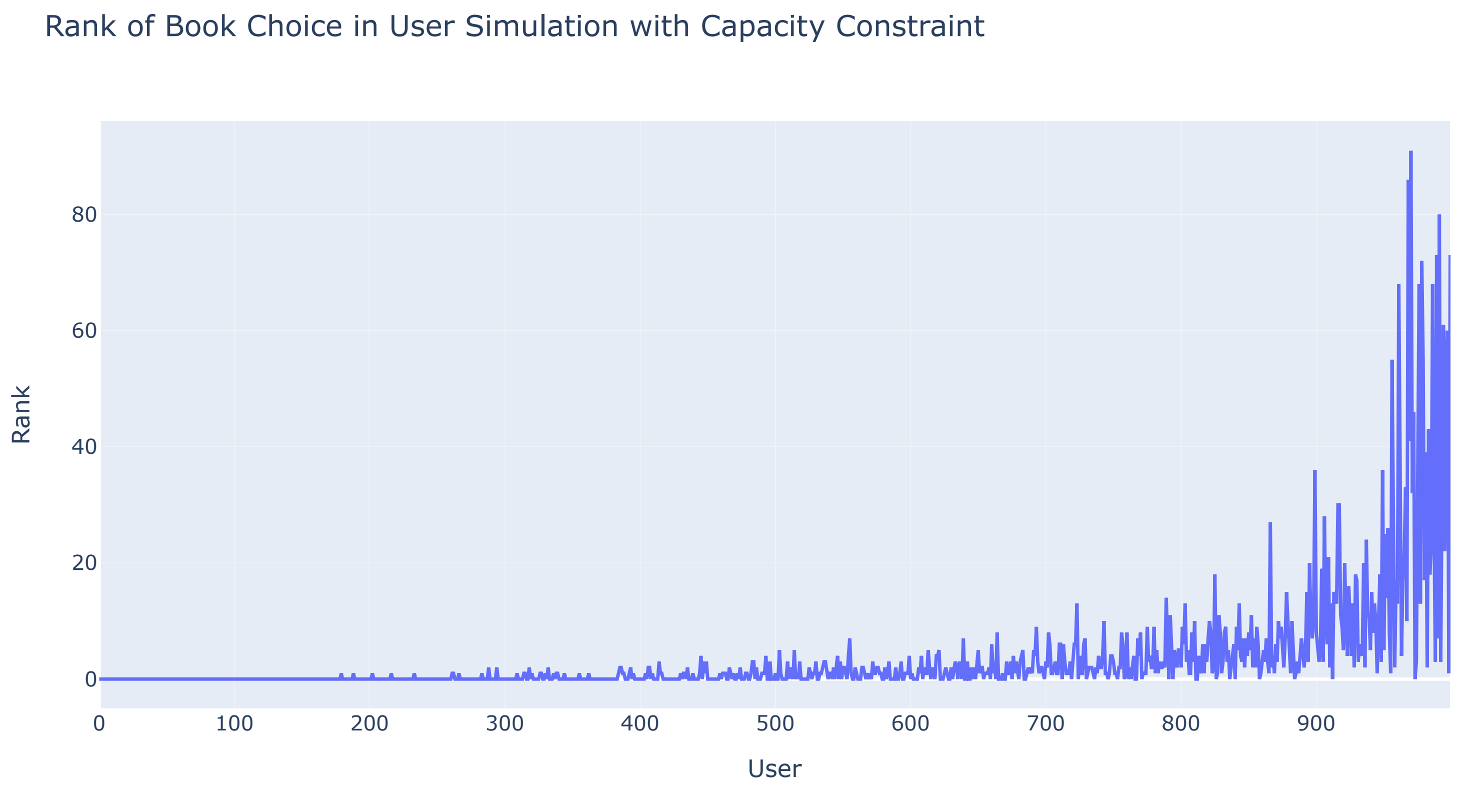
```
#Create a list to simulate users with index 0 - 999
sequence_user = list(range(1000))

#Set Up of Book Capacity
bookCapacity = {}
for x in range(200):
    bookCapacity.update({x:5})
```

In [107]:

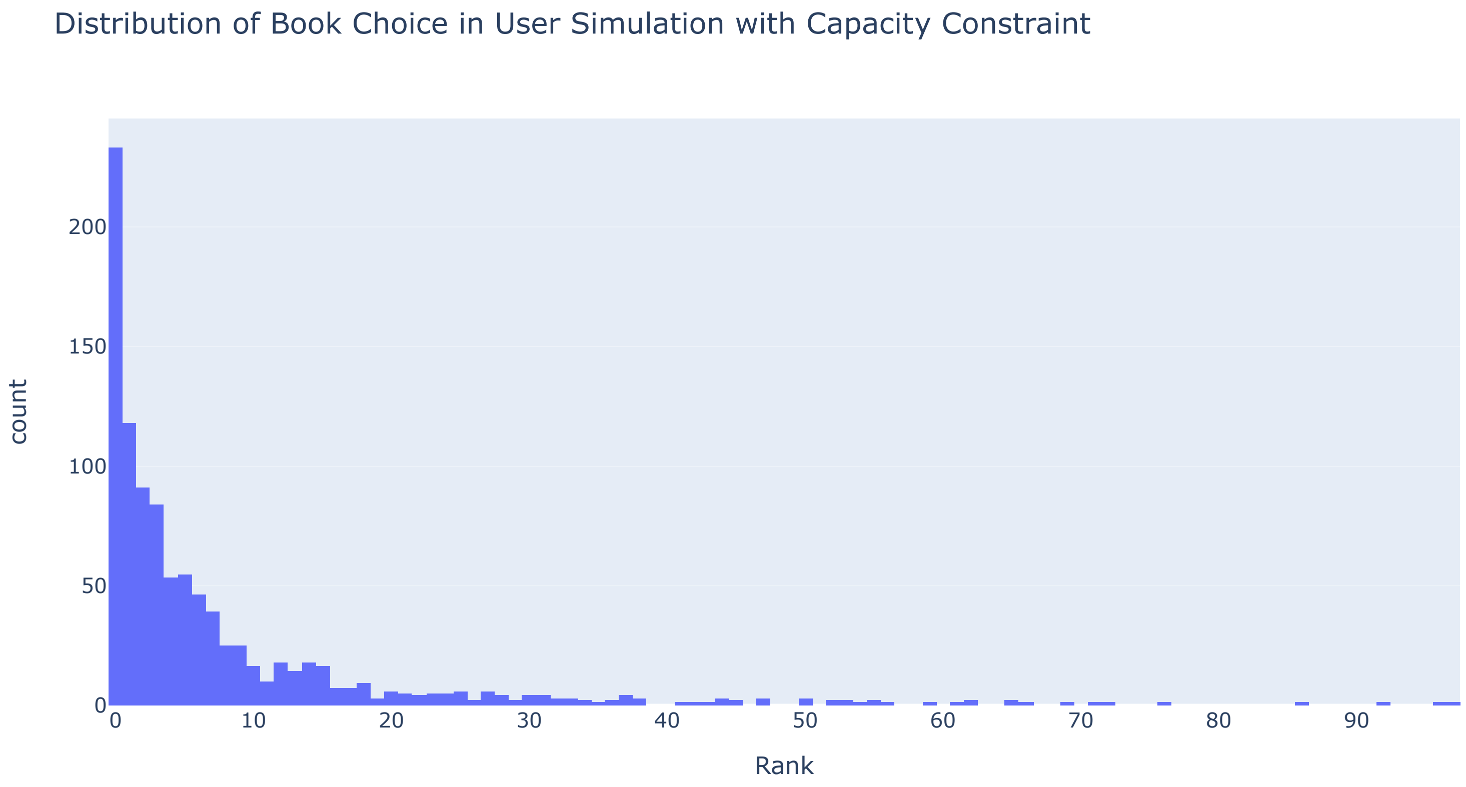
```python
#Get the Recommendations for each user
rec_interact0 = get_recommendations_for_each_user(predicted_interact0,100)
#Create an empty dict to keep track of ranks
user_rank = {}
for user in sequence_user: #for each user in the sequence
        index = 0 #init index of rank to be 0
        for book in rec_interact0[user]: #iterate through its book list
            if bookCapacity[book] > 0: #if the book is available
                bookCapacity.update({book:bookCapacity[book] - 1}) #subtract the book from inventory
                user_rank.update({user:index}) #give the book to the user
                break #break inner for loop
            else:
                index = index + 1 #update the index to be the next rank
```

In [109]:

```
#Plot Results
d = {'User':list(user_rank.keys()),'Rank':list(user_rank.values())}
df = pd.DataFrame(d, columns=['User','Rank'])
fig = px.line(df, x="User", y="Rank", title='Rank of Book Choice in User Simulation with Capacity Constraint')
fig.show()
```



In [34]:
```python
fig = px.histogram(df, x="Rank",nbins=100, title='Distribution of Book Choice in User Simulation with Capacity Constraint')
fig.show()
```



3b) [Bonus -- 6 pts] Optimal recommendations under capacity constraints -- maximum weight matching

[3 points] Now let's do "optimal" recommendations with capacity, using maximum weight matching. Create the same two plots as above. Describe what you observe compared to the naive recommendations above.

We suggest you use the `scipy.optimize.linear_sum_assignment` function. In that case, `np.tile` might also come in handy to create 5 copies of each items.

In [ ]:
```

```

[3 points] Of course, in reality you don't observe all the users at the same time -- they come in one by one, and you need to create a recommendation for the first user before the 50th user shows up. Here's let's pretend that users show up in batches of 100. So the first 100 users at the same time, next 100, etc. In this case, you can do "batched maximum weight matching," where you run maximum weight matching for the first 100 together to determine recommendations. Then, you do the same thing for the next 100 users with the items that are remaining, etc.

Implement the above, show the same two plots as above, and describe what you observe. Note that this part requires careful attention for how many of each item remain after each round.

In [ ]:
```

```

3c) Score functions for recommendations under capacity constraints

Here, we are working with just 200 items and 1000 users, and so batched maximum weight matching is feasible to run. In practice, with millions of items, that might not be an effective strategy. Now, we ask you to implement the score function approach from class.

You should normalize the predicted ratings between 0 and 1 so that you are not dividing by a negative or close to 0 average rating before proceeding.

Implement the above and run the same simulation as part 3c, show the same two plots, and describe what you observe.

For this part, use the following score function:

$$\frac{r_{ij}}{\bar{r}_{ij}}\sqrt{C_j}$$

Creating Functions to normalize the prediction ratings and compute score formula

In [35]:
```python
#Get the Predicted Interact 0 Ratings
predicted_interact0 = get_predictions(user_vectors_interact0,book_vectors_interact0)
#Intialize the df to be predicted ratings
df = pd.DataFrame(predicted_interact0, columns = list(range(200)))
#Functions to normalize columns between 0 and 1
def normalize(x):
    df[x] = (df[x] - min(df[x])) / (max(df[x]) - min(df[x]))
    return df[x]
#Function to compute score sij
def score(rdf,x,C=5):
    rdf[x] = (rdf[x]/rdf[x].mean())*math.sqrt(C)
    return rdf[x]

## Doing the normalization and scoring for the inital frame
for x in list(df.columns):
    df[x] = normalize(x)
    df[x] = score(df,x)
```

Setting the Sequence of Users and Book Inventory

In [110]:
```python
sequence_user = list(range(1000))
#Set Up of Book Capacity
bookCapacity = {}
for x in range(200):
    bookCapacity.update({x:5})
```

Running the Simulation where recommendations are dynamically updated based on Score function

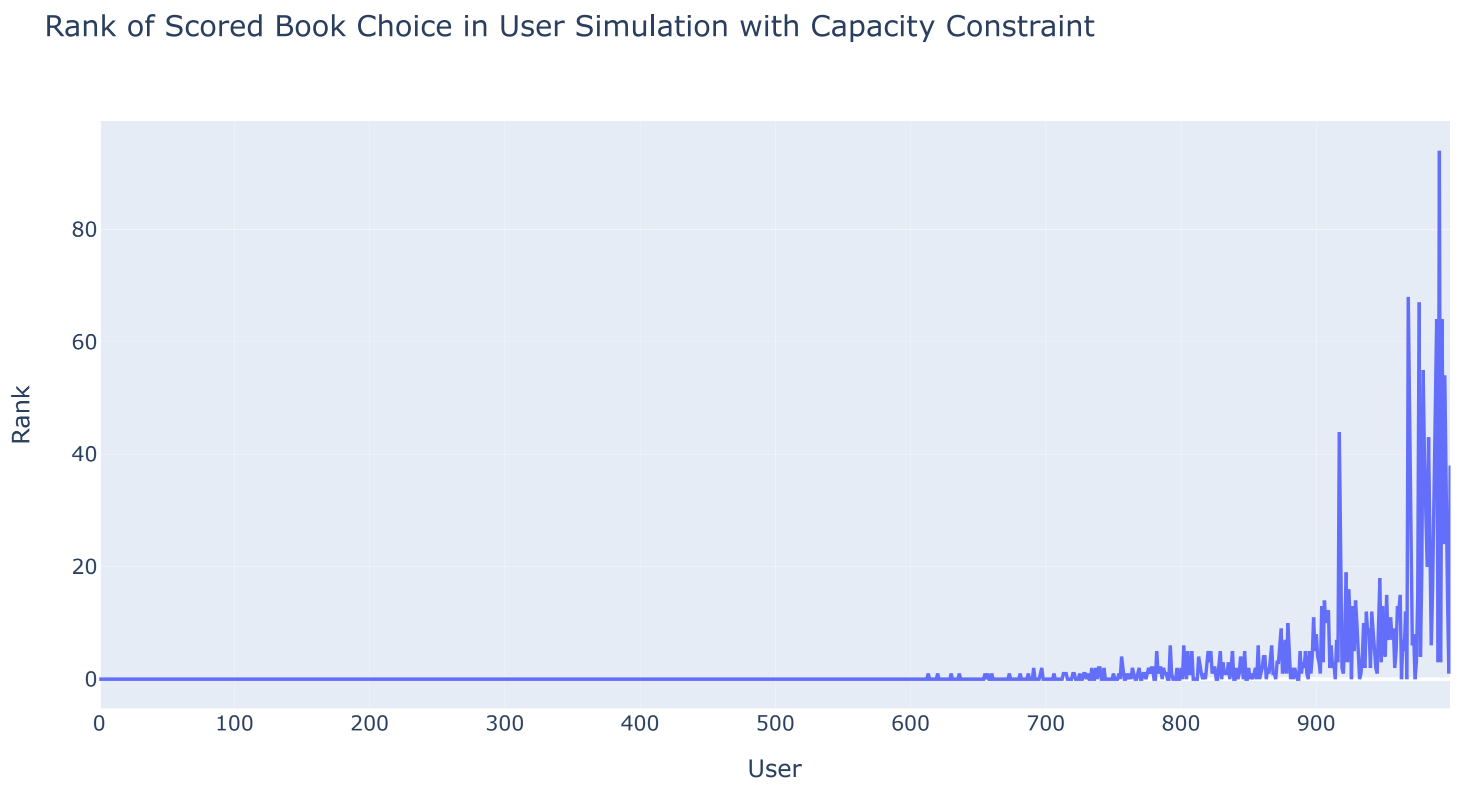
In [37]:

```
runningdf = df.copy() #Intialize a running df of predicted ratings that will update throughout the simulation
scored_interact0 = df.to_numpy() #get the scores in array form
rec_interact0 = get_recommendations_for_each_user(scored_interact0, 100) #get initial recommendations
user_rank2 = {}
for user in sequence_user: #for each user in the sequence
        index = 0 #set the rank to 0
        temp_rec = rec_interact0 #get temporary rec
        for book in temp_rec[user]: #iterate through its book list
            if bookCapacity[book] > 0: #if the book is available
                bookCapacity.update({book:bookCapacity[book] - 1}) #subtract the book from inventory
                user_rank2.update({user:index}) #give the book to the user

                ## if the item is selected re compute the prediction matrix
                ### run for loop again

                C = bookCapacity[book] - 1 #calculate how many are left
                ### recompute the recommendation
                if C > 0:
                    runningdf[book] = score(runningdf,book,C) #overwriting that column's prediction to be recomputed with new
                    scored_interact0 = runningdf.to_numpy() #update scored predictions
                    rec_interact0 = get_recommendations_for_each_user(scored_interact0, 100) #update recommendations
                break
            else:
                index = index + 1 #iterate rank
```

Plotting Results

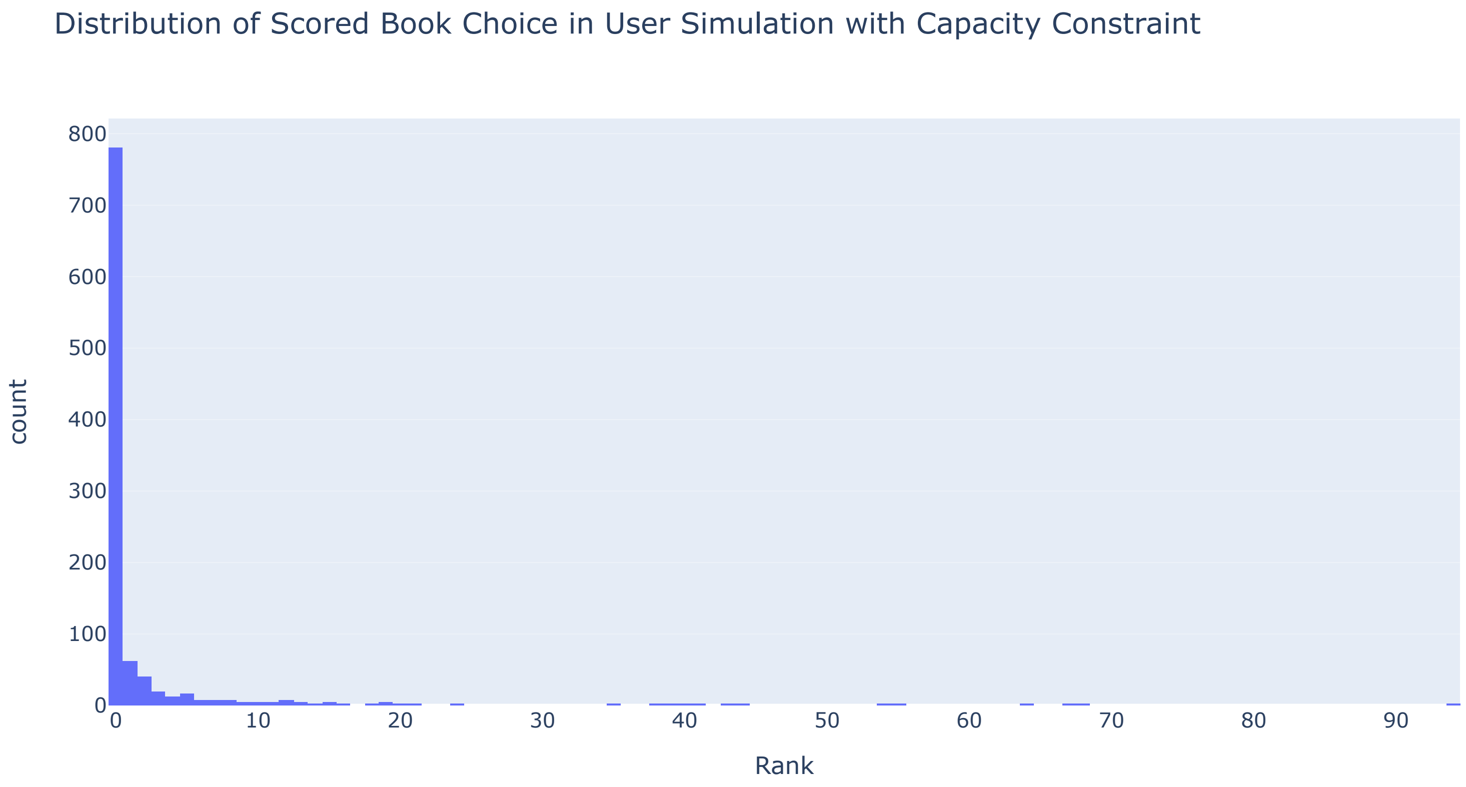
In [38]:

```python
d = {'User':list(user_rank2.keys()),'Rank':list(user_rank2.values())}
df = pd.DataFrame(d, columns=['User','Rank'])
fig = px.line(df, x="User", y="Rank", title='Rank of Scored Book Choice in User Simulation with Capacity Constraint')
fig.show()
```



In [39]:

```
fig = px.histogram(df, x="Rank",nbins=100, title='Distribution of Scored Book Choice in User Simulation with Capacity Constra
fig.show()
```



Observation of Scored Plots

In the scored plots you can see that the first 780 users gets their first choice wheras in the non-scored version only 532 people got their first choice. Scoring allows the recommendations to be dynamic based on what capacity is left hence giving more people thier first choice.

Comment for entire homework: In this homework, we haven't been careful with what is "training" data and what is "test" data. For example, in 3c, you're using average ratings from customers who haven't shown up yet in your simulation. In Problem 2, when training the user/book vectors we used data from customers that we are then pretending we haven't seen data from. In practice, and for the class project, you should be more careful. Such train/test/validation pipelines should be a core part of what you learn in machine learning classes.