Assignment_1 Report

1. Visiting Prediction

The methodology I used in this prediction is to use Cosine score to see if the item category in each prediction pairs matches the category pattern of the reviewer's past purchase category; if they matches well, the predictor will predict True, otherwise No.

To improve the overall accuracy of the prediction, I implemented the follwoing strategies:

- Creating a more reasonable category list to use in the model:
- The basis of picking the category is: first, I flattened all the category and made a dictionary consisting the frequncy count for each of them; then, from the 50 most frequent categories I chose the 22 categories that are either extreamely popular or necessary for classifying the pattern; finally, I built a dictionary with the 22 categories to use in the model.
- Helpful (ideas):
  - Opposites: ('Women', 198548), ('Men', 80021),

    ('Petite', 20100), ('Big & Tall', 12217), ('Plus-Size', 11704),

    ('Girls', 3046), ('Boys', 2509),

    ('Baby Girls', 2120), ('Baby Boys', 2069),
  - Broad categories: ('Intimates', 28040), ('Jewelry', 23468), ('Sports & Outdoors', 17663) ('Novelty', 28807) ('Fashion', 6629) ('Shoes', 61746)

```
def flatten(items, seqtypes=(list, tuple)):
    for i, x in enumerate(items):
        while i < len(items) and isinstance(items[i], seqtypes):
            items[i:i+1] = items[i]
    return items
```

```
catCuratedList = [
    'Women',
    'Men',
    'Petite',
    'Big & Tall',
    'Plus-Size',
    'Girls',
    'Boys',
    'Baby Girls',
    'Baby Boys',
    'Intimates',
    'Jewelry',
    'Sports & Outdoors',
    'Novelty',
    'Fashion',
    'Shoes',
    'Bras',
    'Accessories',
    'Jeans',
    'Tops & Tees',
    'Socks',
    'Underwear',
    'Baby'
]
```

- Creating a popular item list for the model based on the the frequncy count of each item and used different threshold to further restrict the predicter making predictions.

```
total_items = 0
for k,v in sorted_hist_bis:
    total_items += v
def popular_sorter(cutoff):
    print(total_items)
    popular_items = set()
    count = 0
    for k,v in sorted_hist_bis:
        popular_items.add(k)
        count += v
        if (count >= cutoff * total_items):
            break
    return popular_items.copy()

popular_items_10 = popular_sorter(0.1)
popular_items_20 = popular_sorter(0.2)
popular_items_50 = popular_sorter(0.5)
popular_items_80 = popular_sorter(0.8)
popular_items_90 = popular_sorter(0.9)
```

- Creating the count weighted reviewer/category arrary and count weighted item/category array which are used to calculate cosine similarities.

```
weighted_categories = np.zeros((num_users, len(unique_category)))
weighted_categories.fill(0)
for d in train:
    uId = reviewerIds[d['reviewerID']]
    cats = getCatIds(d, unique_category)
    for c in cats:
        weighted_categories[uId][c] += 1

weighted_catToitem = np.zeros((num_business, len(unique_category)))
for d in train:
    bId = businessIds[d['itemID']]
    cats = getCatIds(d, unique_category)
    for c in cats:
        weighted_catToitem[bId][c] += 1
```

- Developing the final prediction model:

```
def predict_purchase_baseline(uid, bid):

    if bid not in unique_business:
        return 0
    if bid in popular_items_10:
        return 1
    elif uid not in unique_reviewer:
        if bid in popular_items_50:
            return 1
        return 0
    else:
        b = businessIds[bid]
        u = reviewerIds[uid]

        bc = np.atleast_2d(weighted_catToitem[b])
        uc = np.atleast_2d(weighted_categories[u])
        cos = cosine_similarity(bc, uc)
        if bid in popular_items_20 and cos > 0:
            return 1
        elif cos >= 0.40 and bid in popular_items_50:

            return 1
        elif bid in popular_items_80 and cos >= 0.60:
            return 1
        elif cos >= 0.80:

            return 1

    return 0
```

Problem and difficulties of building the purchase prediction recommender system:

I think the greatest limitation when building the model was the computing time. In fact, the initial model was based on jaccard similarity score comparing user/category pattens; however, the model could only run on a very small dataset and never finishing running on the test set. That is why I decided to compare similarity only based the user's past category purchase pattern and the item's belonging categories. Similarly, I think the precition accuracy can be greatly improved if using KNN to compare similarities instead of the other methods. By examing the dataset, it is quite obviouse that the reviewers are highly clustered based on past purchase category patterns.

The potential improvements that are possible to implement will be selecting and creating better category list, extracting more meaningful features to get similarity scores and making comparison between similar reviewers rather than between the single reviewer's past histories along.

---

2. Rating prediction

The model I selected is collaborative filtering, with the interaction term of reviewer's rating to items and items' rating from reviewers.

$$min \sum_{u,i \in U,I} (R_{u,i} - \alpha + \beta_u + \beta_i + \gamma_u \gamma_i^T)^2 + \lambda(\sum_{i \in I} \beta_i^2 + \sum_{u \in U} \beta_u^2 \sum_{u \in U} ||\gamma_u||2^2 + \sum i \in I||\gamma_i||2^2)$$

where 'Lambda' is the regularization constant, 'alpha' is the default bias, 'beta_u', 'beta_i' is the bias for the user and item, and 'gamma_u', 'gamma_i' is a dimensionality reduction of the Ru, i matrix. Optimizing this can be done by finding converg in a gradient descent process.

```
def update_alpha_beta(alpha, bU, bI, Ru, Rb, train_data, lam):
    a = 0.0
    for uid in list(range(len(bU))):
        for bid, rating in Ru[uid].items():
            a += (rating - (bU[uid] + bI[bid]))/len(train_data)
    alpha = a
    bU_new = [0.0]*len(bU)
    for uid in list(range(len(bU))):
        for bid, rating in Ru[uid].items():
                bU_new[uid] += (rating - (alpha + bI[bid]))/(lam + len(Ru[uid]))
    bU = bU_new

    bI_new = [0.0]*len(bI)
    for bid in list(range(len(bI))):
        for uid, rating in Rb[bid].items():
            bI_new[bid] += (rating - (alpha + bU[uid]))/(lam + len(Rb[bid]))
    bI = bI_new
    return a, bU_new, bI_new
def find_cost(alpha, bU, bI, Ru, lam):
    bUsq = np.sum(np.square(bU))
    bIsq = np.sum(np.square(bI))

    term1 = 0.0
    for uid in list(range(len(bU))):
        for bid, r in Ru[uid].items():
            term1 += (alpha + bU[uid] + bI[bid] - r)**2
    return term1 + lam*(bUsq + bIsq)
```

The regulization constant I tried to optimize the model:

```
lambdas = [0.1, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 20.0, 50.0, 100.0, 1000.0]
```

and it turned out that when lambda is 7.0, the model has the minimum MSE.

Final prediction on rating:

```
def pred_rating(uid, bid, a, bU, bI, ru, rb):
    rating = 0.0
    if (uid not in reviewerIds and bid in businessIds):
        for u in bU:
            rating += a + u + bI[businessIds[bid]]
        rating = rating/len(bU)
    elif (uid in reviewerIds and bid not in businessIds):
        for i in bI:
            rating += a + bU[reviewerIds[uid]] + i
        rating = rating/len(bI)
    elif (uid not in reviewerIds and bid not in businessIds):
        rating = a
    elif (uid in reviewerIds and bid in businessIds):
        rating += a + bU[reviewerIds[uid]] + bI[businessIds[bid]]
    return rating
```

Improvements that are possible to implement in order to get a much better accuracy would be to incroporate reviewers' rating towards category as a bias term.