Assignment 1 Report
Zerong Li – A15689664
Kaggle User Name: zerongli, Kaggle Diaply Name: Zerong Li

Part 1 – Read Prediction:
Since all (user,book) pairs in the test data are balanced, meaning that we can roughly predict half of each user's books as read and half of them as non-read. By doing this, we need to score each pair in the test set. The way I score each (user,book) pair is by calculating the popularity of the book and the Jaccard similarity. To make the prediction more accurate, I also add cosine similarity.

Parsing Data as usual:

In [2]:
```
path = "Downloads/assignment1/train_Interactions.csv.gz"
f = gzip.open(path, 'rt')
header = f.readline()
header = header.strip().split(',')
header
```

Out[2]:
```
['userID', 'bookID', 'rating']
```

In [3]:
```
dataset = []
for line in f:
    fields = line.strip().split(',')
    d = dict(zip(header, fields))
    d['rating'] = int(d['rating'])
    dataset.append(d)
```

In [4]:
```
dataset[0]
```

Out[4]:
```
{'userID': 'u79354815', 'bookID': 'b14275065', 'rating': 4}
```

Parsing Data

In [5]:
```
usersPerBook = defaultdict(set)
BooksPerUser = defaultdict(set)
books = set()
for d in dataset:
    u,b = d['userID'], d['bookID']
    usersPerBook[b].add(u)
    BooksPerUser[u].add(b)
    if (b not in books):
        books.add(b)
```

In [6]:
```
X_train = dataset[:190000]
X_valid = dataset[190000:]
```

Popularity:

In [8]:
```
def popularity(book):
    count = 0
    for b in mostPopular:
        if b[1] != book:
            count = count + b[0]
        else:
            count = count + b[0]
            break
    #print(count/totalRead)
    return (count/totalRead)
```

Jaccard similarity:

In [9]:
```
def Jaccard(s1, s2):
    numer = len(s1.intersection(s2))
    denom = len(s1.union(s2))
    if denom > 0:
        return numer/denom
    return 0
```

In [10]:
```
def Jaccard_score(u, b):
    similarities = []
    users = set(usersPerBook[b])
    for b2 in BooksPerUser[u]:
        sim = Jaccard(users,set(usersPerBook[b2]))
        if sim > 0:
            similarities.append(sim)
    similarities.sort()
    if len(similarities) == 0: return 0
    average = np.mean(similarities)
    return average
```

Cosine similarity:

In [12]:
```
def cos_sim(a,b):
    numer = len(a.intersection(b))
    denom = len(a) * len(b)
    if denom > 0:
        return numer/denom
    return 0
```

In [13]:
```
def cos_score(u,b):
    similarities = []
    uprime = usersPerBook[b]
    for i in uprime:
        book = BooksPerUser[i]
        candidateItems = u
        sim = cos_sim(book, BooksPerUser[candidateItems])
        if sim > 0:
            similarities.append(sim)
    similarities.sort()
    #print(len(similarities))
    if len(similarities) == 0: return 0
    return max(similarities)
```

Make prediction:

In [14]:
```
test_data = []
for l in open("Downloads/assignment1/pairs_Read.txt"):
    if l.startswith("userID"):
        continue
    u,b = l.strip().split('-')
    test_data.append((u,b))
```

In [15]:
```
BooksPerUserTest = defaultdict(set)
for d in test_data:
    u,b = d[0], d[1]
    BooksPerUserTest[u].add(b)
```

In [17]:
```
BooksPerUserScore = defaultdict(set)
for u in BooksPerUserTest:
    for b in BooksPerUserTest[u]:
        pop = popularity(b)
        jac = Jaccard_score(u,b)
        cos = cos_score(u,b)
        #jacc2 = Jaccard_score_book(u,b)
        dp = (jac * cos) / pop
        BooksPerUserScore[u].add((dp, b))
```

In [ ]:
```

```

In [19]:
```
rank = []
for u in BooksPerUserScore:
    BooksPerUserScore[u] = sorted(BooksPerUserScore[u], reverse = True)
    length = math.ceil(len(BooksPerUserScore[u])/2)
    count = 0
    for pair in BooksPerUserScore[u]:
        if count != length:
            rank.append((u,pair[1]))
            count = count + 1
        else:
            break
```

In [22]:
```
predictions = open("Downloads/assignment1/predictions_Read.txt", 'w')
for l in open("Downloads/assignment1/pairs_Read.txt"):
    if l.startswith("userID"):
    #header
        predictions.write(l)
        continue
    u,b = l.strip().split('-')
    if (u,b) in rank:
        predictions.write(u + '-' + b + ",1\n")
    else:
        predictions.write(u + '-' + b + ",0\n")

predictions.close()
```

Similar to Homework 3, I need to generate some negative samples to test the accuracy of my prediction.

Generate negative sample from the 10000-validation set:

In [32]:
```
validateSet = []
trueSet = []

for i in mix:
    if ((i['userID'], i['bookID']) in rank_valid):
        validateSet.append(True)
    else:
        validateSet.append(False)

    if (i['read'] == 1):
        trueSet.append(True)
    else:
        trueSet.append(False)
```

In [33]:
```
zipping = list(zip(validateSet, trueSet))
acc = [i[0] == i[1] for i in zipping]
accuracy = sum(acc)/len(acc)
print("accuracy = " + str(accuracy))
```

```
accuracy = 0.7079
```

Example the negative sample to the whole dataset, meaning we will have 200,000 positive samples and 200,000 negative samples.

In [254]:
```
validateSet = []
trueSet = []

for i in tqdm(mix):
    if ((i['userID'], i['bookID']) in rank_valid):
        validateSet.append(True)
    else:
        validateSet.append(False)

    if (i['read'] == 1):
        trueSet.append(True)
    else:
        trueSet.append(False)
```

```
HBox(children=(IntProgress(value=0, max=400000), HTML(value='')))
```

In [255]:
```
zipping = list(zip(validateSet, trueSet))
acc = [i[0] == i[1] for i in zipping]
accuracy = sum(acc)/len(acc)
print("accuracy = " + str(accuracy))
```

```
accuracy = 0.7570675
```

Part 2 – Category Prediction

For this part, I am using TF-IDF to create a vector for each review in the dataset. I treat each review as a document and basically loop through every single word in each review and create a TF-IDF vector for each review. The TF-IDF vector should help us distinguish some unique words in reviews among all documents. Sklearn has a method called TfidfVectorizer. I fit and transform the vector containing all reviews and it generates a feature vector.

Parsing data:

```
In [2]: path = "Downloads/assignment1/train_Category.json"

In [3]: def parseDataFromFile(fname):
            for l in open(fname):
                yield ast.literal_eval(l)
        def readGz(path):
            for l in gzip.open(path, 'rt'):
                yield eval(l)

In [4]: data = list(parseDataFromFile(path))

In [ ]:

In [5]: data[0]

Out[5]: {'n_votes': 0,
 'review_id': 'r99763621',
 'user_id': 'u17334941',
 'review_text': "Genuinely enthralling. If Collins or Bernard did invent this out of whole cloth, they deserve a meda
l for imagination. Lets leave the veracity aside for a moment - always a touchy subject when it comes to real life st
ories of the occult - and talk about the contents. \n The Black Alchemist covers a period of two years in which Colli
ns, a magician, and Bernard, a psychic, undertook a series of psychic quests that put them in opposition with the tit
ular Black Alchemist. As entertainment goes, the combination of harrowing discoveries, ancient lore, and going down t
he pub for a cigarette and a Guinness, trying to make sense of it all while a hen party screams at each other, is a w
inner. It is simultaneously down to earth and out of this world. \n It reads fast, both because of the curiousity and
because Collins has a very clear writing style. Sometimes its a little clunky or over repetitive and there's a few me
etings that get underreported, but I am very much quibbling here. Mostly important, he captures his own and Bernard's
sense of wonder, awe and occasionally revulsion enough that I shared them.",
 'rating': 5,
 'genreID': 2,
 'genre': 'fantasy_paranormal'}

In [ ]:

In [6]: X_train_raw = data[:190000]
        y_train_raw = data[:190000]
        X_valid_raw = data[190000:]
        y_valid_raw = data[190000:]

In [7]: all_reviews = [d['review_text'] for d in tqdm(data)]

100% 200000/200000 [00:00<00:00, 1404120.33it/s]
```

y is just a vector containing all genreID of each review

```
In [8]: vectorizer = TfidfVectorizer()
        X = vectorizer.fit_transform(tqdm(all_reviews))
        y = [d['genreID'] for d in tqdm(data)]
        y_train = [d['genreID'] for d in tqdm(y_train_raw)]
        y_valid = [d['genreID'] for d in tqdm(y_valid_raw)]
        pickle.dump(vectorizer.vocabulary_,open("feature.pkl","wb"))

100% 200000/200000 [00:18<00:00, 10981.29it/s]

100% 200000/200000 [00:00<00:00, 731120.03it/s]

100% 190000/190000 [00:00<00:00, 1515986.58it/s]

100% 10000/10000 [00:00<00:00, 325119.68it/s]
```

After I have the feature vector, I fit X and y into Logistic Regression model and SVM.

Accuracy of Logistic Regression locally:

```
In [9]: mod = LogisticRegression(C=1)
        X_train = X[:190000]

In [10]: mod.fit(X_train, y_train)

/Users/zawinglee/anaconda3/lib/python3.7/site-packages/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
/Users/zawinglee/anaconda3/lib/python3.7/site-packages/sklearn/linear_model/logistic.py:469: FutureWarning: Default multi_class will be changed to 'auto' in 0.22. Specify the multi_class option to silence this warning.
  "this warning.", FutureWarning)

Out[10]: LogisticRegression(C=1, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)

In [11]: X_valid = X[190000:]
         pred_valid = mod.predict(X_valid)
         accuracy_score(y_valid, pred_valid)

Out[11]: 0.7785
```

Accuracy of Linear SVM locally:

```
In [17]: clf = svm.LinearSVC(random_state=0, tol=1e-5)

In [18]: clf.fit(X_train, y_train)

Out[18]: LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
          intercept_scaling=1, loss='squared_hinge', max_iter=1000,
          multi_class='ovr', penalty='l2', random_state=0, tol=1e-05,
          verbose=0)

In [19]: pred_valid = clf.predict(X_valid)
         accuracy_score(y_valid, pred_valid)

Out[19]: 0.7935
```

And fit the test data with Logistic Regression or SVM. I chose SVM since it has higher accuracy.

```
In [28]: pred_svc = clf.predict(tfidf)

In [29]: index = 0
         predictions = open("Downloads/assignment1/predictions_Category_svc.txt", 'w
         predictions.write("userID-reviewID,prediction\n")
         for l in readGz("Downloads/assignment1/test_Category.json.gz"):
             predictions.write(l['user_id'] + '-' + l['review_id'] + "," + str(pred_
             index += 1
         predictions.close()
```