In [1]:
```
import gzip
import random
import scipy
import tensorflow as tf
from collections import defaultdict
from implicit import bpr
from surprise import SVD, Reader, Dataset
from surprise.model_selection import train_test_split
```

Data is available at http://cseweb.ucsd.edu/~jmcauley/pml/data/. Download and save to your own directory

In [2]:
```
dataDir = "/home/jmcauley/pml_data/"
```

Latent factor model (Surprise)

Using the library's inbuilt data reader, extract tsv-formatted data

In [3]:
```
reader = Reader(line_format='user item rating', sep='\t')
data = Dataset.load_from_file(dataDir + "goodreads_fantasy.tsv", reader=reader)
```

Standard latent-factor model

In [4]:
```
model = SVD()
```

Inbuilt functions to split into training and test fractions

In [5]:
```
trainset, testset = train_test_split(data, test_size=.25)
```

Fit the model and extract predictions

In [6]:
```
model.fit(trainset)
predictions = model.test(testset)
```

Estimate for a single (test) rating

In [7]:
```
predictions[0].est
```

Out[7]:
```
3.6334479463688463
```

MSE for model predictions (test set)

In [8]:
```
sse = 0
for p in predictions:
    sse += (p.r_ui - p.est)**2

print(sse / len(predictions))
```

```
1.1883531641648757
```

Bayesian Personalized Ranking (Implicit)

In [9]:

```
def parseData(fname):
    for l in gzip.open(fname):
        d = eval(l)
        del d['review_text'] # Discard the reviews, to save memory when we don't
        yield d
```

Full dataset of Goodreads fantasy reviews (fairly memory-hungry, could be replaced by something smaller)

In [10]:

```
data = list(parseData(dataDir + "goodreads_reviews_fantasy_paranormal.json.gz"))
```

In [11]:

```
random.shuffle(data)
```

Example from the dataset

In [12]:

```
data[0]
```

Out[12]:

```
{'book_id': '13451182',
 'date_added': 'Sun Sep 09 18:58:45 -0700 2012',
 'date_updated': 'Sun Oct 07 15:13:32 -0700 2012',
 'n_comments': 1,
 'n_votes': 0,
 'rating': 1,
 'read_at': 'Sun Sep 09 00:00:00 -0700 2012',
 'review_id': 'fec617f0bd2947c641189b01e2433ec9',
 'started_at': 'Sun Sep 09 00:00:00 -0700 2012',
 'user_id': '30d8035cfe8ee0fb007fa896b5a3ba54'}
```

Build a few utility data structures. Since we'll be converting the data to a sparse interaction matrix, the main structure here is to assign each user/item to an ID from 0 to nUsers/nItems.

In [13]:

```
userIDs,itemIDs = {},{}

for d in data:
    u,i = d['user_id'],d['book_id']
    if not u in userIDs: userIDs[u] = len(userIDs)
    if not i in itemIDs: itemIDs[i] = len(itemIDs)

nUsers,nItems = len(userIDs),len(itemIDs)
```

In [14]:

```
nUsers,nItems
```

Out[14]:

```
(256088, 258212)
```

Convert dataset to sparse matrix. Only storing positive feedback instances (i.e., rated items).

In [15]:

```
Xiu = scipy.sparse.lil_matrix((nItems, nUsers))
for d in data:
    Xiu[itemIDs[d['book_id']],userIDs[d['user_id']]] = 1

Xui = scipy.sparse.csr_matrix(Xiu.T)
```

Bayesian Personalized Ranking model with 5 latent factors

In [16]:
```
model = bpr.BayesianPersonalizedRanking(factors = 5)
```

Fit the model

In [17]:
```
model.fit(Xiu)
```

Get recommendations for a particular user (the first one) and to get items related to (similar latent factors) to a particular item

In [18]:
```
recommended = model.recommend(0, Xui)
related = model.similar_items(0)
```

In [19]:
```
related
```

Out[19]:
```
[(0, 1.0),
 (42098, 0.9885355),
 (142964, 0.9845209),
 (150861, 0.98274595),
 (231639, 0.9826295),
 (182330, 0.9813926),
 (240868, 0.98134804),
 (226720, 0.9796706),
 (84748, 0.9783791),
 (140340, 0.97788805)]
```

Extract user and item factors

In [20]:
```
itemFactors = model.item_factors
userFactors = model.user_factors
```

In [21]:
```
itemFactors[0]
```

Out[21]:
```
array([-0.74582803, -0.10878776,  0.32922822,  0.16516064,  0.38874012,
        0.7460656 ], dtype=float32)
```

Latent factor model (Tensorflow)

In [22]:
```
def parse(path):
    g = gzip.open(path, 'r')
    for l in g:
        yield eval(l)
```

Goodreads comic book data

In [23]:
```
userIDs = {}
itemIDs = {}
interactions = []

for d in parse(dataDir + "goodreads_reviews_comics_graphic.json.gz"):
    u = d['user_id']
    i = d['book_id']
    r = d['rating']
    if not u in userIDs: userIDs[u] = len(userIDs)
    if not i in itemIDs: itemIDs[i] = len(itemIDs)
    interactions.append((u,i,r))
```

In [24]:
```
random.shuffle(interactions)
len(interactions)
```

Out[24]:
```
542338
```

Split into train and test sets

In [25]:
```
nTrain = int(len(interactions) * 0.9)
nTest = len(interactions) - nTrain
interactionsTrain = interactions[:nTrain]
interactionsTest = interactions[nTrain:]
```

In [26]:
```
itemsPerUser = defaultdict(list)
usersPerItem = defaultdict(list)
for u,i,r in interactionsTrain:
    itemsPerUser[u].append(i)
    usersPerItem[i].append(u)
```

Mean rating, just for initialization

In [27]:
```
mu = sum([r for _,_,r in interactionsTrain]) / len(interactionsTrain)
```

Gradient descent optimizer, could experiment with learning rate

In [28]:
```
optimizer = tf.keras.optimizers.Adam(0.1)
```

Latent factor model tensorflow class

In [29]:
```
class LatentFactorModel(tf.keras.Model):
    def __init__(self, mu, K, lamb):
        super(LatentFactorModel, self).__init__()
        # Initialize to average
        self.alpha = tf.Variable(mu)
        # Initialize to small random values
        self.betaU = tf.Variable(tf.random.normal([len(userIDs)],stddev=0.001))
        self.betaI = tf.Variable(tf.random.normal([len(itemIDs)],stddev=0.001))
        self.gammaU = tf.Variable(tf.random.normal([len(userIDs),K],stddev=0.001
```

```
        self.gammaI = tf.Variable(tf.random.normal([len(itemIDs),K],stddev=0.001
        self.lamb = lamb

    # Prediction for a single instance (useful for evaluation)
    def predict(self, u, i):
        p = self.alpha + self.betaU[u] + self.betaI[i] +\
            tf.tensordot(self.gammaU[u], self.gammaI[i], 1)
        return p

    # Regularizer
    def reg(self):
        return self.lamb * (tf.reduce_sum(self.betaU**2) +\
                            tf.reduce_sum(self.betaI**2) +\
                            tf.reduce_sum(self.gammaU**2) +\
                            tf.reduce_sum(self.gammaI**2))

    # Prediction for a sample of instances
    def predictSample(self, sampleU, sampleI):
        u = tf.convert_to_tensor(sampleU, dtype=tf.int32)
        i = tf.convert_to_tensor(sampleI, dtype=tf.int32)
        beta_u = tf.nn.embedding_lookup(self.betaU, u)
        beta_i = tf.nn.embedding_lookup(self.betaI, i)
        gamma_u = tf.nn.embedding_lookup(self.gammaU, u)
        gamma_i = tf.nn.embedding_lookup(self.gammaI, i)
        pred = self.alpha + beta_u + beta_i +\
               tf.reduce_sum(tf.multiply(gamma_u, gamma_i), 1)
        return pred

    # Loss
    def call(self, sampleU, sampleI, sampleR):
        pred = self.predictSample(sampleU, sampleI)
        r = tf.convert_to_tensor(sampleR, dtype=tf.float32)
        return tf.nn.l2_loss(pred - r) / len(sampleR)
```

Initialize the model. Could experiment with number of factors and regularization rate.

In [30]:

```
modelLFM = LatentFactorModel(mu, 5, 0.00001)
```

Training step (for the batch-based model from Chapter 5)

In [31]:

```
def trainingStep(model, interactions):
    Nsamples = 50000
    with tf.GradientTape() as tape:
        sampleU, sampleI, sampleR = [], [], []
        for _ in range(Nsamples):
            u,i,r = random.choice(interactions)
            sampleU.append(userIDs[u])
            sampleI.append(itemIDs[i])
            sampleR.append(r)

        loss = model(sampleU,sampleI,sampleR)
        loss += model.reg()
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients((grad, var) for
                              (grad, var) in zip(gradients, model.trainable_vari
                              if grad is not None)
    return loss.numpy()
```

Run 100 iterations (really 100 batches) of gradient descent

In [32]:
```
for i in range(100):
    obj = trainingStep(modelLFM, interactionsTrain)
    if (i % 10 == 9): print("iteration " + str(i+1) + ", objective = " + str(obj
```

```
iteration 10, objective = 0.53868735
iteration 20, objective = 0.5198331
iteration 30, objective = 0.5232154
iteration 40, objective = 0.52439743
iteration 50, objective = 0.5135148
iteration 60, objective = 0.49897566
iteration 70, objective = 0.50508446
iteration 80, objective = 0.5124023
iteration 90, objective = 0.51079476
iteration 100, objective = 0.5071494
```

Prediction for a particular user/item pair

In [33]:
```
u,i,r = interactionsTest[0]
```

In [34]:
```
modelLFM.predict(userIDs[u], itemIDs[i]).numpy()
```

Out[34]:
```
3.3664043
```

Bayesian personalized ranking (Tensorflow)

In [35]:
```
items = list(itemIDs.keys())
```

Batch-based version from Chapter 5

In [36]:
```
class BPRbatch(tf.keras.Model):
    def __init__(self, K, lamb):
        super(BPRbatch, self).__init__()
        # Initialize variables
        self.betaI = tf.Variable(tf.random.normal([len(itemIDs)],stddev=0.001))
        self.gammaU = tf.Variable(tf.random.normal([len(userIDs),K],stddev=0.001
        self.gammaI = tf.Variable(tf.random.normal([len(itemIDs),K],stddev=0.001
        # Regularization coefficient
        self.lamb = lamb

    # Prediction for a single instance
    def predict(self, u, i):
        p = self.betaI[i] + tf.tensordot(self.gammaU[u], self.gammaI[i], 1)
        return p

    # Regularizer
    def reg(self):
        return self.lamb * (tf.nn.l2_loss(self.betaI) +\
                            tf.nn.l2_loss(self.gammaU) +\
                            tf.nn.l2_loss(self.gammaI))

    def score(self, sampleU, sampleI):
```

```
        u = tf.convert_to_tensor(sampleU, dtype=tf.int32)
        i = tf.convert_to_tensor(sampleI, dtype=tf.int32)
        beta_i = tf.nn.embedding_lookup(self.betaI, i)
        gamma_u = tf.nn.embedding_lookup(self.gammaU, u)
        gamma_i = tf.nn.embedding_lookup(self.gammaI, i)
        x_ui = beta_i + tf.reduce_sum(tf.multiply(gamma_u, gamma_i), 1)
        return x_ui

    def call(self, sampleU, sampleI, sampleJ):
        x_ui = self.score(sampleU, sampleI)
        x_uj = self.score(sampleU, sampleJ)
        return -tf.reduce_mean(tf.math.log(tf.math.sigmoid(x_ui - x_uj)))
```

In [37]:

```
optimizer = tf.keras.optimizers.Adam(0.1)
```

In [38]:

```
modelBPR = BPRbatch(5, 0.00001)
```

In [39]:

```
def trainingStepBPR(model, interactions):
    Nsamples = 50000
    with tf.GradientTape() as tape:
        sampleU, sampleI, sampleJ = [], [], []
        for _ in range(Nsamples):
            u,i,_ = random.choice(interactions) # positive sample
            j = random.choice(items) # negative sample
            while j in itemsPerUser[u]:
                j = random.choice(items)
            sampleU.append(userIDs[u])
            sampleI.append(itemIDs[i])
            sampleJ.append(itemIDs[j])

        loss = model(sampleU,sampleI,sampleJ)
        loss += model.reg()
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients((grad, var) for
                              (grad, var) in zip(gradients, model.trainable_vari
                              if grad is not None)
    return loss.numpy()
```

Run 100 batches of gradient descent

In [40]:

```
for i in range(100):
    obj = trainingStepBPR(modelBPR, interactions)
    if (i % 10 == 9): print("iteration " + str(i+1) + ", objective = " + str(obj
```

```
iteration 10, objective = 0.53062546
iteration 20, objective = 0.4767778
iteration 30, objective = 0.47098666
iteration 40, objective = 0.47340673
iteration 50, objective = 0.4744774
iteration 60, objective = 0.47567236
iteration 70, objective = 0.47382742
iteration 80, objective = 0.47544688
iteration 90, objective = 0.47159576
iteration 100, objective = 0.4722678
```

Prediction for a particular user/item pair. Note that this is an unnormalized score (which can be used for ranking)

In [41]:
```
u,i,_ = interactionsTest[0]
```

In [42]:
```
# In this case just a score (that can be used for ranking), rather than a predic
modelBPR.predict(userIDs[u], itemIDs[i]).numpy()
```

Out[42]:
```
2.2043138
```

In [ ]:
```

```

Exercises

5.1

Adapt the latent factor model above, simply deleting any terms associated with latent factors

In [43]:
```
class LatentFactorModelBiasOnly(tf.keras.Model):
    def __init__(self, mu, lamb):
        super(LatentFactorModelBiasOnly, self).__init__()
        # Initialize to average
        self.alpha = tf.Variable(mu)
        # Initialize to small random values
        self.betaU = tf.Variable(tf.random.normal([len(userIDs)],stddev=0.001))
        self.betaI = tf.Variable(tf.random.normal([len(itemIDs)],stddev=0.001))
        self.lamb = lamb

    # Prediction for a single instance (useful for evaluation)
    def predict(self, u, i):
        p = self.alpha + self.betaU[u] + self.betaI[i]
        return p

    # Regularizer
    def reg(self):
        return self.lamb * (tf.reduce_sum(self.betaU**2) +\
                            tf.reduce_sum(self.betaI**2))

    # Prediction for a sample of instances
    def predictSample(self, sampleU, sampleI):
        u = tf.convert_to_tensor(sampleU, dtype=tf.int32)
        i = tf.convert_to_tensor(sampleI, dtype=tf.int32)
        beta_u = tf.nn.embedding_lookup(self.betaU, u)
        beta_i = tf.nn.embedding_lookup(self.betaI, i)
        pred = self.alpha + beta_u + beta_i
        return pred

    # Loss
    def call(self, sampleU, sampleI, sampleR):
        pred = self.predictSample(sampleU, sampleI)
        r = tf.convert_to_tensor(sampleR, dtype=tf.float32)
        return tf.nn.l2_loss(pred - r) / len(sampleR)
```

In [44]:
```
modelBiasOnly = LatentFactorModelBiasOnly(mu, 0.00001)
```

In [45]:
```
def trainingStepBiasOnly(model, interactions):
    Nsamples = 50000
    with tf.GradientTape() as tape:
        sampleU, sampleI, sampleR = [], [], []
        for _ in range(Nsamples):
            u,i,r = random.choice(interactions)
            sampleU.append(userIDs[u])
            sampleI.append(itemIDs[i])
            sampleR.append(r)

        loss = model(sampleU,sampleI,sampleR)
        loss += model.reg()
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients((grad, var) for
        (grad, var) in zip(gradients, model.trainable_variables)
        if grad is not None)
    return loss.numpy()
```

In [46]:
```
for i in range(50):
    obj = trainingStepBiasOnly(modelBiasOnly, interactionsTrain)
    if (i % 10 == 9): print("iteration " + str(i+1) + ", objective = " + str(obj
```

```
iteration 10, objective = 0.5712534
iteration 20, objective = 0.53842133
iteration 30, objective = 0.52470183
iteration 40, objective = 0.5304408
iteration 50, objective = 0.52147734
```

Compute the MSEs for a model which always predicts the mean, versus one which involves bias terms

In [47]:
```
def MSE(predictions, labels):
    differences = [(x-y)**2 for x,y in zip(predictions,labels)]
    return sum(differences) / len(differences)
```

In [48]:
```
alwaysPredictMean = [mu for _ in interactionsTest]
labels = [r for _,_,r in interactionsTest]
```

In [49]:
```
MSE(alwaysPredictMean, labels)
```

Out[49]:
```
1.3266520043138732
```

In [50]:
```
biasOnlyPredictions =\
    [modelBiasOnly.predict(userIDs[u],itemIDs[i]).numpy() for u,i,_ in interacti
```

In [51]:
```
biasOnlyPredictions[0]
```

Out[51]: 3.3093212

In [52]:
```
MSE(biasOnlyPredictions, labels)
```

Out[52]: 0.9999872631832556

5.2

Performance of a complete latent factor model (using the latent factor model implementation in the examples above)

In [53]:
```
optimizer = tf.keras.optimizers.Adam(0.1)
modelLFM = LatentFactorModel(mu, 10, 0.00001)
```

In [54]:
```
for i in range(50):
    obj = trainingStep(modelLFM, interactionsTrain)
    if (i % 10 == 9): print("iteration " + str(i+1) + ", objective = " + str(obj
```

```
iteration 10, objective = 0.530306
iteration 20, objective = 0.53029466
iteration 30, objective = 0.5466817
iteration 40, objective = 0.53111464
iteration 50, objective = 0.5286445
```

In [55]:
```
predictions = [modelLFM.predict(userIDs[u],itemIDs[i]).numpy() for u,i,_ in inte
```

In [56]:
```
MSE(predictions, labels)
```

Out[56]: 1.0094479508990533

(probably needs a little more tuning in terms of number of latent factors, learning rate, etc.)

5.3

Experiment with rounding the predictions

In [57]:
```
predictionsRounded = [int(p + 0.5) for p in predictions]
```

In [58]:
```
MSE(predictionsRounded, labels)
```

Out[58]: 1.094756057085961

Seems to result in worse performance. For a rough explanation, consider a random variable that takes a value of "1" half the time and "2" half the time; in terms of the MSE, always predicting 1.5 (and always incurring moderate errors) is preferable to always predicting either of 1 or 2 (and incurring a large error half the time).

5.4

Following the BPR code from examples above

In [59]:
```
optimizer = tf.keras.optimizers.Adam(0.1)
modelBPR = BPRbatch(10, 0.00001)
```

In [60]:
```
for i in range(50):
    obj = trainingStepBPR(modelBPR, interactionsTrain)
    if (i % 10 == 9): print("iteration " + str(i+1) + ", objective = " + str(obj
```

```
iteration 10, objective = 0.5274262
iteration 20, objective = 0.48642576
iteration 30, objective = 0.48361415
iteration 40, objective = 0.48916948
iteration 50, objective = 0.49432752
```

In [61]:
```
interactionsTestPerUser = defaultdict(set)
itemSet = set()
for u,i,_ in interactionsTest:
    interactionsTestPerUser[u].add(i)
    itemSet.add(i)
```

AUC implementation

In [62]:
```
def AUCu(u, N): # N samples per user
    win = 0
    if N > len(interactionsTestPerUser[u]):
        N = len(interactionsTestPerUser[u])
    positive = random.sample(interactionsTestPerUser[u],N)
    negative = random.sample(itemSet.difference(interactionsTestPerUser[u]),N)
    for i,j in zip(positive,negative):
        si = modelBPR.predict(userIDs[u], itemIDs[i]).numpy()
        sj = modelBPR.predict(userIDs[u], itemIDs[j]).numpy()
        if si > sj:
            win += 1
    return win/N
```

In [63]:
```
def AUC():
    av = []
    for u in interactionsTestPerUser:
        av.append(AUCu(u, 10))
    return sum(av) / len(av)
```

In [64]:
```
AUC()
```

Out[64]:
```
0.793553704111058
```

In [ ]: