In [1]:
```
import gzip
import math
import matplotlib.pyplot as plt
import random
import scipy
import tensorflow as tf
from collections import defaultdict
from implicit import bpr
from scipy.spatial import distance
```

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/"
```

Basic fairness measurements - interaction and recommendation distributions

Goodreads graphic novel data. Train a standard recommender, and compare generated recommendations to historical interactions, in terms of frequency distributions.

In [3]:
```
def parse(path):
    g = gzip.open(path, 'r')
    for l in g:
        yield eval(l)
```

In [4]:
```
data = []
for x in parse(dataDir + "goodreads_reviews_comics_graphic.json.gz"):
    del x['review_text']
    data.append(x)
```

In [5]:
```
random.shuffle(data)
```

In [6]:
```
userIDs,itemIDs = {},{}
revUIDs,revIIDs = {},{}
for d in data:
    u,i = d['user_id'],d['book_id']
    if not u in userIDs:
        userIDs[u] = len(userIDs)
        revUIDs[userIDs[u]] = u
    if not i in itemIDs:
        itemIDs[i] = len(itemIDs)
        revIIDs[itemIDs[i]] = i
```

In [7]:
```
nUsers,nItems = len(userIDs),len(itemIDs)
nUsers,nItems
```

Out[7]:
```
(59347, 89311)
```

Problem setup follows the BPR library from "implicit" (see Chapter 5), but could be completed with any recommender

In [8]:
```
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)
```

In [9]:
```
model = bpr.BayesianPersonalizedRanking(factors = 5)
```

Fit the model

In [10]:
```
model.fit(Xiu)
```

Extract recommendations (example)

In [11]:
```
model.recommend(0, Xui)
```

Out[11]:
```
[(3056, 5.0516043),
 (24859, 5.0356407),
 (10783, 5.030104),
 (5767, 4.985171),
 (769, 4.9555626),
 (2193, 4.9438295),
 (3191, 4.919392),
 (1388, 4.907593),
 (14141, 4.9072804),
 (3324, 4.898348)]
```

Next we extract recommendations for all users, and compare these to their interaction histories. First collect histories from historical trends:

In [12]:
```
interactionTuples = []
itemsPerUser = defaultdict(list)
```

In [13]:
```
for d in data:
    u,i = d['user_id'],d['book_id']
    interactionTuples.append((userIDs[u],itemIDs[i]))
    itemsPerUser[userIDs[u]].append(itemIDs[i])
```

Next build similar data structures containing recommendations for each user

In [14]:
```
recommendationTuples = []
```

For each user, generate a set of recommendations equivalent in size to their number of interactions used for training

In [15]:
```
for u in range(len(userIDs)):
    A = model.recommend(u, Xui, N = len(itemsPerUser[u]))
    for i, sc in A:
        recommendationTuples.append((u,i))
```

So far our data structures just contain lists of historical interactions and recommendations for each user. Convert these into counts for each item. This is done both for interactions (I) and recommendations (R).

In [16]:
```
countsPerItemI = defaultdict(int)
countsPerItemR = defaultdict(int)
for u,i in interactionTuples:
    countsPerItemI[i] += 1

for u,i in recommendationTuples:
    countsPerItemR[i] += 1
```

Sort counts by popularity to generate plots

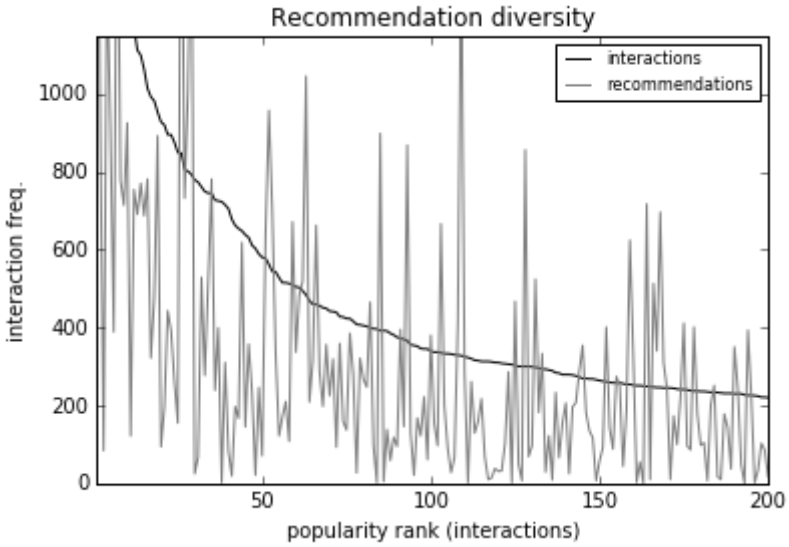
In [17]:
```
sortPopularI = [(countsPerItemI[i], i) for i in countsPerItemI]
sortPopularR = [(countsPerItemR[i], i) for i in countsPerItemR]
```

In [18]:
```
sortPopularI.sort(reverse=True)
sortPopularR.sort(reverse=True)
```

Collect the information for interactions and recommendations for plotting

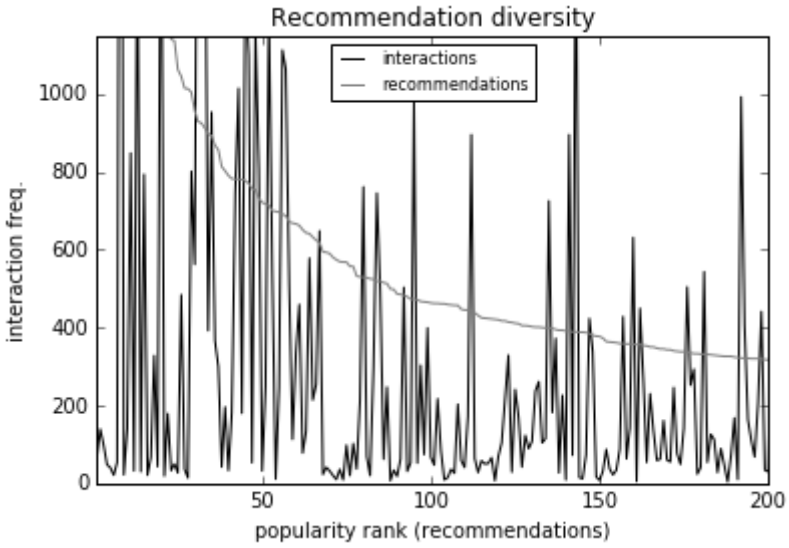
In [19]:
```
YI = [x[0] for x in sortPopularI[:300]] # Interaction frequency
Ys = [x[1] for x in sortPopularI[:300]] # Associated items
YR = [countsPerItemR[x] for x in Ys] # Recommendation frequency for those items
```

In [20]:
```
plt.plot(range(1,301),YI, color='k', label = "interactions")
plt.xlim(1,200)
plt.ylim(0,1150)
plt.plot(range(1,301),YR, color='grey', label = "recommendations")
plt.xlabel("popularity rank (interactions)")
plt.ylabel("interaction freq.")
plt.legend(loc="best",  prop={'size': 8})
plt.title("Recommendation diversity")
plt.show()
```



In [21]:
```
YR = [x[0] for x in sortPopularR[:300]]
Ys = [x[1] for x in sortPopularR[:300]]
YI = [countsPerItemI[x] for x in Ys]
```

In [22]:
```
plt.plot(range(1,301),YI, color='k', label = "interactions")
plt.xlim(1,200)
plt.ylim(0,1150)
plt.plot(range(1,301),YR, color='grey', label = "recommendations")
plt.xlabel("popularity rank (recommendations)")
plt.ylabel("interaction freq.")
plt.legend(loc="best",  prop={'size': 8})
plt.title("Recommendation diversity")
plt.show()
```



Gini goefficient

The two implementations below compute the gini coefficient either by comparing all pairs, or by doing so for a given number of samples

In [23]:
```
def gini(z, samples=1000000):
    m = sum(z) / len(z)
    denom = 2 * samples * m
    numer = 0
    for _ in range(samples):
        i = random.choice(z)
        j = random.choice(z)
        numer += math.fabs(i - j)
    return numer / denom

def giniExact(z):
    m = sum(z) / len(z)
    denom = 2 * len(z)**2 * m
    numer = 0
    for i in range(len(z)):
        for j in range(len(z)):
            numer += math.fabs(z[i] - z[j])
    return numer / denom
```

Compute the gini coefficients of interactions versus distributions for the two distributions computed in the experiments above

In [24]:
```
gini([x[0] for x in sortPopularI])
```

Out[24]:
```
0.7190767454225961
```

In [25]:
```
gini([x[0] for x in sortPopularR])
```

Out[25]:
```
0.7722158419767746
```

Average cosine similarity between interactions versus recommendations

Given a set of items, measure the average cosine distance between them (by taking a sample of pairs, similar to our implementation of the gini coefficient). This can be used as a rough measure of the diversity of a set of recommendations.

In [26]:
```
def avCosine(z, samples = 100):
    av = []
    while len(av) < samples:
        i = random.choice(z)
        j = random.choice(z)
        d = 1 - distance.cosine(i, j)
        if not math.isnan(d) and d > 0:
            av.append(d)
    return sum(av) / len(av)
```

In [27]:
```
itemFactors = model.item_factors
```

Compute the average cosine similarity among interactions from a particular user

In [28]:
```
avCosine([itemFactors[i] for i in itemsPerUser[0]])
```

Out[28]:
```
0.6911979604884982
```

Compute the same quantity across a large sample of recommendations and interactions for several users, as an aggregate measure of recommendation versus interaction diversity

In [29]:
```
avavI = []
avavR = []

while len(avavI) < 1000:
    u = random.choice(range(len(userIDs)))
    if len(itemsPerUser[u]) < 10:
        continue
    aI = avCosine([itemFactors[i] for i in itemsPerUser[u]])
    A = model.recommend(u, Xui, N = len(itemsPerUser[u]))
    aR = avCosine([itemFactors[i[0]] for i in A])
    avavI.append(aI)
    avavR.append(aR)
```

A lower average cosine similarity among interactions indicates that they are more diverse compared to recommendations

In [30]:
```
sum(avavI) / len(avavI)
```

Out[30]:
```
0.6248067230512387
```

In [31]:
```
sum(avavR) / len(avavR)
```

Out[31]:
```
0.8639710014653086
```

Fair recommendation

First, implement a latent factor model. Our basic implementation follows the Tensorflow latent factor model from Chapter 5.

In [32]:
```
userIDs = {}
itemIDs = {}
itemsPerUser = defaultdict(set)
styles = {}
beerNames = {}
interactions = []

for d in parse(dataDir + "beer.json.gz"):
    if not 'user/gender' in d: continue
    g = d['user/gender'] == 'Male'
    u = d['user/profileName']
    i = d['beer/beerId']
    r = d['review/overall']
    styles[i] = d['beer/style']
    beerNames[i] = d['beer/name']
    if not u in userIDs: userIDs[u] = len(userIDs)
    if not i in itemIDs: itemIDs[i] = len(itemIDs)
```

```
    itemsPerUser[u].add(i)
    interactions.append((g,u,i,r))
```

In [33]:
```
len(interactions)
```

Out[33]:
```
637221
```

Data structures to organize interactions by gender

In [34]:
```
interactionsPerItemG = defaultdict(list) # Male interactions
interactionsPerItemGneg = defaultdict(list) # Other interactions
```

In [35]:
```
for g,u,i,r in interactions:
    if g: interactionsPerItemG[i].append((u,r)) # Male interactions for this ite
    else: interactionsPerItemGneg[i].append((u,r))
```

In [36]:
```
itemsG = set(interactionsPerItemG.keys()) # Set of items associated with male in
itemsGneg = set(interactionsPerItemGneg.keys())
itemsBoth = itemsG.intersection(itemsGneg)
```

In [37]:
```
mu = sum([r for _,_,_,r in interactions]) / len(interactions) # For initializati
```

In [38]:
```
optimizer = tf.keras.optimizers.Adam(0.1)
```

Latent factor model. The "absoluteFairness" function is new; others are equivalent to our model from Chapter 5.

In [39]:
```
class LatentFactorModel(tf.keras.Model):
    def __init__(self, mu, K, lamb, lambFair):
        super(LatentFactorModel, self).__init__()
        self.alpha = tf.Variable(mu)
        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
        self.lambFair = lambFair

    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

    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))

    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

    # For a single item. This score should be averaged over several items
    def absoluteUnfairness(self, i):
        G = interactionsPerItemG[i]
        Gneg = interactionsPerItemGneg[i]
        # interactions take the form (u,r)
        rG    = tf.reduce_mean(tf.convert_to_tensor([r for _,r in G]))
        rGneg = tf.reduce_mean(tf.convert_to_tensor([r for _,r in Gneg]))
        pG    = tf.reduce_mean(
            self.predictSample([userIDs[u] for u,_ in G], [itemIDs[i]]*len(G)))
        pGneg = tf.reduce_mean(
            self.predictSample([userIDs[u] for u,_ in Gneg], [itemIDs[i]]*len(Gn
        Uabs = tf.abs(tf.abs(pG - rG) - tf.abs(pGneg - rGneg))
        return self.lambFair * Uabs

    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 [40]:

```
modelFair = LatentFactorModel(mu, 10, 0.000001, 0.000001)
```

In [41]:

```
def trainingStep(model, interactions):
    Nsamples = 5000
    Nfair = 50
    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 += sum([model.absoluteUnfairness(i)
                     for i in random.sample(itemsBoth,Nfair)])/Nfair
        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()
```

In [42]:

```
for i in range(100):
    obj = trainingStep(modelFair, interactions)
    if (i % 50 == 49): print("iteration " + str(i+1) + ", objective = " + str(ob
```

```
iteration 50, objective = 0.2573108
iteration 100, objective = 0.27134913
```

Same model, but not including the fairness terms

In [43]:
```
modelUnfair = LatentFactorModel(mu, 10, 0.000001, 0.000001)
```

In [44]:
```
def trainingStepUnfair(model, interactions):
    Nsamples = 5000
    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()
```

In [45]:
```
for i in range(100):
    obj = trainingStepUnfair(modelUnfair, interactions)
    if (i % 50 == 49): print("iteration " + str(i+1) + ", objective = " + str(ob
```

```
iteration 50, objective = 0.573847
iteration 100, objective = 0.419454
```

Maximal marginal relevance (MMR)

Similarity between items (in this case cosine)

In [46]:
```
def cosine(v,w):
    return float(tf.tensordot(v, w, 1) / (tf.norm(v) * tf.norm(w)))
```

In [47]:
```
def sim(itemEmbeddings,i,j):
    gamma_i = itemEmbeddings[i]
    gamma_j = itemEmbeddings[j]
    return cosine(gamma_i, gamma_j)
```

Find the most similar item among a candidate set

In [48]:
```
def maxSim(itemEmbeddings,i,seq):
    if len(seq) == 0: return 0
    return max([sim(itemEmbeddings,i,j) for j in seq])
```

Select a random user to receive recommendations

In [49]:
```
u = random.choice(list(userIDs.keys()))
```

In [50]:
```
itemSet = set(itemIDs.keys())
```

Define a function to get the next recommendation given an initial list, i.e., the mamimal marginally relevant item. Lambda (lamb) controls the tradeoff between compatibility an diversity.

In [51]:
```
def getNextRec(model,compatScores,itemEmbeddings,seq,lamb):
    scores = [(lamb * s - (1 - lamb) * maxSim(itemEmbeddings,i,seq), i)
        for (s,i) in compatScores if not i in seq]
    (maxScore,maxItem) = max(scores)
    return maxItem
```

Before re-ranking, generate a list of compatibility scores, i.e., a ranked list of items for a particular user

In [52]:
```
candidates = list(itemSet.difference(itemsPerUser[u]))
compatScores = list(zip([float(f)
    for f in modelUnfair.predictSample([userIDs[u]]*len(candidates),
        [itemIDs[i] for i in candidates])], candidates))
itemEmbeddings = dict(zip(candidates,
    tf.nn.embedding_lookup(modelUnfair.gammaI, [itemIDs[i] for i in candidates]))

compatScores.sort(reverse=True)
```

Generate a list of recommendations for the user by repeatedly calling the retrieving the maximal marginally relevant recommendation. First, just get the most relevant items (lambda=1) without encouraging diversity.

Note that this implementation is not particularly optimized, and takes several seconds to generate a list of recommendations.

In [53]:
```
recs = []
while len(recs) < 10:
    i = getNextRec(modelUnfair,compatScores[:1000],itemEmbeddings,recs,1.0)
    recs.append(i)
[beerNames[i] for i in recs]
```

Out[53]:
```
['Moonflower ESB',
 'Augustiner BrÃ¤u MÃ¤rzen Bier',
 'Perseguidor (Batch 5)',
 'Saint Arnold Divine Reserve #3',
 'Beer Hunter Brown Porter',
 'Kuhnhenn Vintage Ale',
 'Oktoberfest',
 'Witte Noire',
 'Golden Nugget',
 'Spring Sour']
```

More Recommendations for different relevance/diversity tradeoffs. Note that the tradeoff parameter is quite sensitive to the specific scale of the model parameters.

In [54]:
```
recs = []
while len(recs) < 10:
    i = getNextRec(model,compatScores[:1000],itemEmbeddings,recs,0.25)
    recs.append(i)
[beerNames[i] for i in recs]
```

Out[54]:
```
['Moonflower ESB',
 "He'Brew Jewbelation Bar Mitzvah (13) Barrel-Aged On Rye",
 'Perseguidor (Batch 5)',
 'Houblon Noir',
 'Bourbon Barrel Imperial Stout',
 'Pumpkin Ale',
 'The Great Dismal Black IPA',
 'Leireken Wild Berries Belgian Ale',
 'Saint Arnold Divine Reserve #3',
 'California Rollin Plum Brew']
```

In [55]:
```
recs = []
while len(recs) < 10:
    i = getNextRec(model,compatScores[:1000],itemEmbeddings,recs,0.1)
    recs.append(i)
[beerNames[i] for i in recs]
```

Out[55]:
```
['Moonflower ESB',
 "He'Brew Jewbelation Bar Mitzvah (13) Barrel-Aged On Rye",
 "MacLean's Pale Ale",
 'Leireken Wild Berries Belgian Ale',
 'Pumpkin Ale',
 'Allagash FOUR - Bourbon Barrel Aged',
 'Mount Desert Island Ginger',
 'Brooklyn High Line Elevated Wheat',
 'Oak Barrel Reverend',
 'Saison De Lente, 100% Brett']
```

In [ ]:
```

```

Exercises

10.1

First just try out a different similarity function (based on the inner product)

In [56]:
```
def sim(itemEmbeddings,i,j):
    gamma_i = itemEmbeddings[i]
    gamma_j = itemEmbeddings[j]
    return tf.sigmoid(-tf.tensordot(gamma_i, gamma_j, 1)).numpy()
```

In [57]:
```
def maxSim(itemEmbeddings,i,seq):
    if len(seq) == 0: return 0
    return max([sim(itemEmbeddings,i,j) for j in seq])
```

In [58]:
```
def getNextRec(model,compatScores,itemEmbeddings,seq,lamb):
    rels = dict([(i,s) for (s,i) in compatScores if not i in seq])
```

```
    divs = dict([(i,maxSim(itemEmbeddings,i,seq)) for (s,i) in compatScores if n
    scores = [(lamb * rels[i] - (1 - lamb) * divs[i], i) for (s,i) in compatScor
    (maxScore,maxItem) = max(scores)
    return rels[maxItem],divs[maxItem],maxItem
```

In [59]:
```
u = random.choice(list(userIDs.keys()))
```

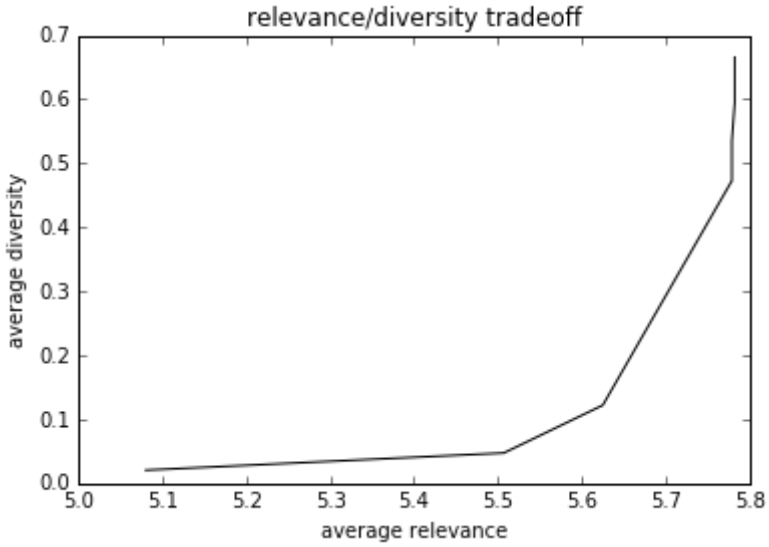
In [60]:
```
candidates = list(itemSet.difference(itemsPerUser[u]))
compatScores = list(zip([float(f)
    for f in modelUnfair.predictSample([userIDs[u]]*len(candidates),
        [itemIDs[i] for i in candidates])], candidates))
itemEmbeddings = dict(zip(candidates,
    tf.nn.embedding_lookup(modelUnfair.gammaI, [itemIDs[i] for i in candidates])

compatScores.sort(reverse=True)
```

Experiment with different relevance/diversity tradeoffs

In [61]:
```
xs = []
ys = []
for lamb in [1.0,0.99,0.9,0.75,0.5,0.25,0.1,0.01,0.001]:
    rels = []
    divs = []
    recs = []
    while len(recs) < 5:
        r,d,i = getNextRec(modelUnfair,compatScores[:1000],itemEmbeddings,recs,l
        rels.append(r)
        divs.append(d)
        recs.append(i)
    xs.append(sum(rels)/len(rels))
    ys.append(sum(divs)/len(divs))
```

And plot the results

In [62]:
```
plt.plot(xs,ys,color='k')
plt.xlabel("average relevance")
plt.ylabel("average diversity")
plt.title("relevance/diversity tradeoff")
plt.show()
```



10.2

Concentration effects

In [63]:
```
data = []
for x in parse(dataDir + "goodreads_reviews_comics_graphic.json.gz"):
    del x['review_text']
    data.append(x)
```

In [64]:
```
random.shuffle(data)
```

In [65]:
```
userIDs,itemIDs = {},{}
revUIDs,revIIDs = {},{}
for d in data:
    u,i = d['user_id'],d['book_id']
    if not u in userIDs:
        userIDs[u] = len(userIDs)
        revUIDs[userIDs[u]] = u
    if not i in itemIDs:
        itemIDs[i] = len(itemIDs)
        revIIDs[itemIDs[i]] = i
```

In [66]:
```
nUsers,nItems = len(userIDs),len(itemIDs)
nUsers,nItems
```

Out[66]:
```
(59347, 89311)
```

In [67]:
```
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)
```

In [68]:
```
model = bpr.BayesianPersonalizedRanking(factors = 5)
```

```
model.fit(Xiu)
```

Measure concentration in terms of the Gini coefficient

In [69]:
```
interactionTuples = []
itemsPerUser = defaultdict(list)
usersPerItem = defaultdict(list)

for d in data:
    u,i = d['user_id'],d['book_id']
    interactionTuples.append((userIDs[u],itemIDs[i]))
    itemsPerUser[userIDs[u]].append(itemIDs[i])
    usersPerItem[itemIDs[i]].append(userIDs[u])
```

In [70]:
```
recommendationTuples1 = []
for u in range(len(userIDs)):
    A = model.recommend(u, Xui, N = len(itemsPerUser[u]))
    for i, sc in A:
        recommendationTuples1.append((u,i))
```

In [71]:
```
countsPerItemI = defaultdict(int)
countsPerItemR1 = defaultdict(int)

for u,i in interactionTuples:
    countsPerItemI[i] += 1

for u,i in recommendationTuples1:
    countsPerItemR1[i] += 1
```

Aplly a small penalty for items that are highly recommended (resulting a reduction of concentration). This is essentially just a simple re-ranking strategy.

In [72]:
```
recommendationTuples2 = []
for u in range(len(userIDs)):
    N = len(itemsPerUser[u])
    A = model.recommend(u, Xui, max(N,100))
    Aadj = []
    for i, sc in A:
        Aadj.append((sc - 0.1*countsPerItemR1[i], i)) # Rerank by penalizing hig
    Aadj.sort(reverse=True)
    for sc, i in Aadj[:N]:
        recommendationTuples2.append((u,i))
```

In [73]:
```
countsPerItemR2 = defaultdict(int)

for u,i in recommendationTuples2:
    countsPerItemR2[i] += 1
```

Compare interaction data, recommendations, and "corrected" recommendations in terms of concentration

In [74]:
```
sortPopularI = [(countsPerItemI[i], i) for i in countsPerItemI]
sortPopularR1 = [(countsPerItemR1[i], i) for i in countsPerItemR1]
sortPopularR2 = [(countsPerItemR2[i], i) for i in countsPerItemR2]
```

In [75]:
```
sortPopularI.sort(reverse=True)
sortPopularR1.sort(reverse=True)
sortPopularR2.sort(reverse=True)
```

In [76]:
```
gini([x[0] for x in sortPopularI])
```

Out[76]:
```
0.7249546705882677
```

In [77]:
```
gini([x[0] for x in sortPopularR1])
```

Out[77]:
```
0.7873034048204256
```

In [78]:
```
gini([x[0] for x in sortPopularR2])
```

Out[78]:
```
0.6873886143871535
```

Ultimately we got a model with lower concentration than the original data. Could adjust the penalty term to control the concentration amount.

10.3

Measure parity in terms of beer ABV (alcohol level). Do high- (or low-) alcohol items tend to get recommended more than we would expect from interaction data?

In [79]:
```
userIDs = {}
itemIDs = {}
itemsPerUser = defaultdict(list)
usersPerItem = defaultdict(list)
styles = {}
beerNames = {}
interactions = []

atr = dict()

for d in parse(dataDir + "beer.json.gz"):
    if not 'user/gender' in d: continue
    try:
        a = d['beer/ABV'] > 7.5
    except Exception as e:
        continue
    u = d['user/profileName']
    i = d['beer/beerId']
    r = d['review/overall']
    styles[i] = d['beer/style']
    beerNames[i] = d['beer/name']
    if not u in userIDs: userIDs[u] = len(userIDs)
    if not i in itemIDs: itemIDs[i] = len(itemIDs)
    itemsPerUser[userIDs[u]].append(i)
```

```
    usersPerItem[itemIDs[i]].append(u)
    interactions.append((u,i,r))
    atr[itemIDs[i]] = a
```

In [80]:
```
nUsers,nItems = len(userIDs),len(itemIDs)
nUsers,nItems
```

Out[80]:
```
(8402, 34866)
```

Start by training a BPR model (using the implicit library)

In [81]:
```
Xiu = scipy.sparse.lil_matrix((nItems, nUsers))
for (u,i,r) in interactions:
    Xiu[itemIDs[i],userIDs[u]] = 1

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

In [82]:
```
model = bpr.BayesianPersonalizedRanking(factors = 5)
model.fit(Xiu)
```

Frequency of positive (high alcohol) versus negative (low alcohol) among interactions

In [83]:
```
apos,aneg = 0,0

for (u,i,r) in interactions:
    if atr[itemIDs[i]]:
        apos += 1
    else:
        aneg += 1

apos,aneg
```

Out[83]:
```
(216825, 395700)
```

Frequency among recommendations (low alcohol items end up being slightly over-recommended)

In [84]:
```
apos,aneg = 0,0

for u in range(len(userIDs)):
    A = model.recommend(u, Xui, N = len(itemsPerUser[u]))
    for i, sc in A:
        if atr[i]:
            apos += 1
        else:
            aneg += 1

apos,aneg
```

Out[84]:
```
(197201, 415324)
```

As in 10.2, correct using a re-ranking strategy with a simple penalty term (in this case encouraging high-alcohol items to be recommended). Again this could be adjusted to achieve the desired calibration.

In [85]:
```
apos,aneg = 0,0

for u in range(len(userIDs)):
    N = len(itemsPerUser[u])
    A = model.recommend(u, Xui, N*2)
    Aadj = []
    for i, sc in A:
        Aadj.append((sc + 0.1*atr[i], i))
    Aadj.sort(reverse=True)
    for sc, i in Aadj[:N]:
        if atr[i]:
            apos += 1
        else:
            aneg += 1

apos,aneg
```

Out[85]: (233451, 379074)

In [ ]:
```
```