Kaggle Username: mollyyyyy

Name on leaderboard (both): Molly

Purchase Prediction

Train Dataset: the whole set

Validation Dataset: the second half of set

Final Approach:

1. Store the categories that users have bought into cateU dictionary and the category of each item into cateI dictionary. Build a category classifier that if the pair contains a new item or new user, randomly return 0 or 1; else if the item's category has been bought by the user, return 1; else if the item is not in the user's category list, return 0. Combine the category classifier with the most popular classifier with threshold 50%.
2. Reason: As the most popular model has a high true negative rate at its best accuracy, I tend to trust the most popular classifier when these two classifiers has different results and the most popular model predict 0. Conversely, since the most popular model has a moderate true positive rate i.e.<0.5 at its best accuracy, I tend to trust the categories classifier when these two classifiers has different results and the most popular model predict 1.
3. Code:

```
# Construct dict of the whole train data (200000) of users-categories or items-categories
cateU={}
for d in data:
    if d['reviewerID'] in cateU and (d['categoryID'] not in cateU[d['reviewerID']]):
        cateU[d['reviewerID']].append(d['categoryID'])
    elif d['reviewerID'] not in cateU:
        cateU[d['reviewerID']] = [d['categoryID']]

cateI={}
#As the length of mul is 0, we are sure that each item only corresponds to one category
mul = []
for d in data:
    if d['itemID'] in cateI and d['categoryID'] != cateI[d['itemID']]:
        mul.append(d['itemID'])
    elif d['itemID'] not in cateI:
        cateI[d['itemID']] = d['categoryID']
print(len(mul))
```

```
0
```

```
def predictions_Purchase_categories(user,item):
    # If this item has never been purchased or the user is new, we randomly
    # true or false. Assume the buy and not buy probability are equal in this case.
    if item not in cateI or user not in cateU:
        return np.random.randint(2)
    elif cateI[item] in cateU[user]:
        return True
    else:
        return False
```

```
predictions = open("predictions_Purchase.txt", 'w')
for l in open("pairs_Purchase.txt"):
    if l.startswith("reviewerID"):
        predictions.write(l)
        continue
    u,i = l.strip().split('-')
    # predict 1 if both models predict 1
    if predictions_Purchase_categories(u,i) == True and i in return1:
        predictions.write(u + '-' + i +",1\n")
    # predict 0 if both models predict 0
    elif predictions_Purchase_categories(u,i) == False and i not in return1:
        predictions.write(u + '-' + i + ",0\n")
    else:
        predictions.write(u + '-' + i + ",0\n")
predictions.close()
```

Trial:

This approach has been tried but the accuracy is high on validation but not high on test set.

1. Store the sub-categories (i.e. strings under 'categories') of each item into a dictionary. Store the average time and average rating of each item. Build a classifier that compare the similarity of this item with items which have been bought by the user in these three dimensions. If this is a new item or new user, randomly return 1 or 0; else if the item has been bought by the user or the maximum Jaccard similarity between the subcategories of this item and those of items bought by the user is greater than 0.8, return 1; else if the intersection of sets of the top 50% cosine similarity of rating and time is not empty, return 1; else return 0.
2. Reason: If the subcategories of an item has a large portion of overlapping with subcategories of items that the user has bought, then the user is very likely to buy this item. Moreover, if the average time and rating of an item is close to those of an item has been bought, then the user is more likely to buy this item. As the user might buy frequently in a specific range of time or tend to buy products with around a level of average rating.
3. Failure reason: Probably due to overfitting. Also average time and rating probably are not strong features to distinguish items, as the standard deviation probably is large. Planned to use latent factor model but have another midterm & homework due so didn't have enough time to achieve and run it.
4. Code:

```
ratU={}
for d in data:
    if d['reviewerID'] in ratU:
        if d['categoryID'] in ratU[d['reviewerID']]:
            r,c = ratU[d['reviewerID']][d['categoryID']]
            c+=1
            ratU[d['reviewerID']][d['categoryID']] = (r+d['rating'],c)
        else:
            ratU[d['reviewerID']][d['categoryID']] = (d['rating'],1)
    else:
        ratU[d['reviewerID']] = {}
        ratU[d['reviewerID']][d['categoryID']] = (d['rating'],1)
```

```
ratI={}
for d in data:
    if d['itemID'] in ratI:
        r,c = ratI[d['itemID']]
        c+=1
        ratI[d['itemID']] = (r+d['rating'],c)
    else:
        ratI[d['itemID']] = (d['rating'],1)

ratIavg = defaultdict(float)
for ra in item:
    ratIavg[ra] = ratI[ra][0]/ratI[ra][1]
```

```
itemsub = defaultdict(list)
for d in data:
    catitem = d['categories']
    for dd in catitem:
        for ddd in dd:
            itemsub[d['itemID']].append(ddd)
for m in item:
    itemsub[m] = list(set(itemsub[m]))
```

```
timeI={}
for d in data:
    if d['itemID'] in timeI:
        time,c = timeI[d['itemID']]
        c+=1
        timeI[d['itemID']] = (time+d['unixReviewTime'],c)
    else:
        timeI[d['itemID']] = (d['unixReviewTime'],1)
timeIavg = defaultdict(int)
for ite in item:
    timeIavg[ite] = int(timeI[ite][0]/timeI[ite][1])
```

```
predictions = open("predictions_Purchase.txt", 'w')
for l in open("pairs_Purchase.txt"):
    if l.startswith("reviewerID"):
        predictions.write(l)
        continue
    u,it = l.strip().split('-')
    if u not in user or it not in item:
        predictions.write(u + '-' + it +","+str(np.random.randint(2))+"\n")
    else:
        tisim = list()
        catsim = list()
        ratesim = list()
        ittime = timeIavg[it]
        itrate = ratIavg[it]
        itcub = itemsub[it]
        for k in range(len(buypair[u])):
            items = buypair[u][k]
            tisim.append(cosine_similarity([timeIavg[items]],[ittime]))
            ratesim.append(cosine_similarity([ratIavg[items]],[itrate]))
            catsim.append(jaccard_similarity(itemsub[items],itcub))
        tisim.sort()
        ratesim.sort()
        tisim.reverse()
        ratesim.reverse()
        if len(tisim)>5:
            ktime = [tisim.index(tisim[ind]) for ind in range(int(len(tisim)/2))]
            krate = [ratesim.index(ratesim[ind]) for ind in range(int(len(ratesim)/2))]
        else:
            ktime = [tisim.index(tisim[0])]
            krate = [ratesim.index(ratesim[0])]
        if max(catsim)>0.8 or it in buypair[u]:
            predictions.write(u + '-' + it +",1\n")
        elif len(set(ktime)&set(krate))>0:
            predictions.write(u + '-' + it +",1\n")
        else:
            predictions.write(u + '-' + it +",0\n")
predictions.close()
```

Category Prediction

Train Dataset: the whole set

Validation Dataset: the second half of dataset

Final Approach:

1. Use linear SVC trained by the frequent words of Men to predict if the category is Men or not, if it is (i.e. the result of decision function is greater than 0), return 1, else if the item or user is new, return 0; else return the most frequent categories bought by the user.
2. Reason: If I only use the most frequent categories classifier, the classifier will tend to predict Women and the accuracy of other categories thus will be low. As Men has the second most portion in the dataset and data for Girls, Boys and Baby are insufficient to train, I used bag of words model to enhance the accuracy of Men. To reduce the noise of Women, I filtered the intersection of Men and popular Women words to construct the words bag for feature vectors. I've tried Linear, RBF kernel SVC and neural network (both under TensorFlow and MLPClassifier of sklearn) and LinearSVC is faster and more accurate. In the C value selection of LinearSVC, I have tried twenty C values from 0.01 to 150 and tends out that they are very close, I finally used 0.1 for the speed and accuracy consideration.
3. Code:

```
# Dictionary store most frequently purchased categories
sortdict = defaultdict(str)
for usr, calist in usrcat.items():
    # If the user just bought one category, this is the most popular one
    if len(calist) == 1:
        sortdict[usr] = calist[0][0]
    else:
        # Sort the category by count in descending order
        calist = sorted(calist, key=lambda x: x[1],reverse=True)
        tie = search_tuple(calist, calist[0][1])
        # If there is no tie, choose the category with the most count
        if len(tie) == 1:
            sortdict[usr] = calist[0][0]
        else:
            cat = (-1,-1)
            # Compare global popularity in the tie case and choose the most globally popular
            for tupca,tupco in tie:
                if catdict[tupca] >  catdict[cat[0]]:
                    cat = list(cat)
                    cat[0] = tupca
                    cat[1] = tupco
                    cat = tuple(cat)
            sortdict[usr] = cat[0]
```

```
Women=[d[1] for d in popw[0:500]]
Men=[d[1] for d in popm[0:1100]]
Girls=[d[1] for d in popg[0:500]]
Boys=[d[1] for d in popb[0:500]]
Baby=[d[1] for d in popba[0:500]]
'''
Women = [d for d in Women if (d not in Men and d not in Girls and d not in Boys and d not in Baby)]
Girls = [d for d in Girls if d not in Boys]
Boys = [d for d in Boys if (d not in Girls)]
Baby = [d for d in Baby if (d not in Men and d not in Girls and d not in Boys and d not in Women)]
Women = Women[:500]
Men = Men[:50]
Girls = Girls[:50]
Boys = Boys[:50]
Baby = Baby[:50]
'''
Men = [d for d in Men if (d not in Women)]
Men = Men[:950]
```

```
classifier = MLPClassifier(activation = 'logistic',shuffle=False,alpha=0.01)
classifier.fit(X, Y)
predm=classifier.predict(dataa)
print(acc(predm,validY))
```

```
learning_rate = 0.001
training_epochs = 120
batch_size = 100
display_step = 20
n_hidden_1 = len(wordstotal)
n_hidden_2 = len(wordstotal)
n_input = len(wordstotal)
n_classes = 2
x = tf.placeholder("float", [None, n_input])
y = tf.placeholder("float", [None, n_classes])
def multilayer_perceptron(x, weights, biases):
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_1 = tf.nn.relu(layer_1)
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    layer_2 = tf.nn.relu(layer_2)
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
pred = multilayer_perceptron(x, weights, biases)
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=pred, labels=y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)
init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

```
predicttf=[]
with tf.Session() as sess:
    sess.run(init)
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(len(X)/batch_size)
        X_batches = np.array_split(X, total_batch)
        Y_batches = np.array_split(Y, total_batch)
        for i in range(total_batch):
            batch_x, batch_y = X_batches[i], Y_batches[i]
            _, c = sess.run([optimizer, cost], feed_dict={x: batch_x,y: batch_y})
            avg_cost += c / total_batch
        if epoch % display_step == 0:
            print("Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".format(avg_cost))
    print("Optimization Finished!")
    correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    predicttf= sess.run(tf.argmax(pred, 1), feed_dict={x: validd})
    save_path = saver.save(sess, "/tmp/model.ckpt")
    print("Model saved in file: %s" % save_path)
```

```
Epoch: 0001 cost= 65.576773573
Epoch: 0021 cost= 1.132750048
Epoch: 0041 cost= 0.338878933
Epoch: 0061 cost= 0.189288672
Epoch: 0081 cost= 0.138321377
Optimization Finished!
Model saved in file: /tmp/model.ckpt
```

```
clf = LinearSVC(C=0.1,max_iter=10000)
clf.fit(X, Y)
#red=clf.predict(validd)
#int(acc(pred,validY))
```

```
LinearSVC(C=0.1, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=10000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)
```

```
pred = [1 if d>0 else 0 for d in clf.decision_function(validd)]
#print(acc(pred,validY))
```

```
dataa = list(readGz("test_Category.json.gz"))
sortpred = [0]*len(dataa)
for i in range(len(dataa)):
    d = dataa[i]
    r = d['reviewerID']
    if r not in sortdict:
        sortpred[i]=0
    else:
        sortpred[i] = sortdict[r]
```

```
0
```

```
predictions = open("predictions_Categories.txt", 'w')
predictions.write("reviewerID-reviewHash,category\n")
for i in range(len(dataa)):
    d = dataa[i]
    r = d['reviewerID']
    h = d['reviewHash']
    #if predicttf[i] == 0:
    if pred[i] == 1:
        predictions.write(r + '-' +h+  ","+str(1)+"\n")
    else:
        predictions.write(r + '-' +h+  ","+str(sortpred[i])+"\n")
predictions.close()
```

```
# Measure the performance of large classifiers with
#different c values to choose the optimised c
from operator import truediv
resp = [0]*5
respt = [0]*5
predictions = open("predictions_Categories.txt", 'w')
arr = [confarr,confarr2,confarr3,confarr4,confarr5,confarr6,confarr7,
       confarr8,confarr9,confarr10,confarr11,confarr12,confarr13,confarr14,
       confarr15,confarr16,confarr17,confarr18,confarr19,confarr20,confarr21]
acc = np.zeros(len(arr))
for i in range(len(validcat)):
    l = validcat[i]
    respt[l['categoryID']] +=1
    for k in range(len(arr)):
        conf = arr[k]
        precat = -1
        maxconf = -100
        for j in range(5):
            if conf[j][i] > maxconf:
                maxconf = confarr[j][i]
                precat = j
        if int(precat) == int(l['categoryID']):
            resp[int(precat)]+=1
            acc[k]+=1
acc = acc/(1.0*len(validcat))
print(acc)
print(resp,respt, map(truediv, resp, respt))
```