Amazon reviews multilingual UK dataset

This is divided into 4 tasks:

1. **Data Processing**
2. **Classification**
3. **Regression**
4. **Recommender Sytstems**
    A. Similarity matching
    B. Predictions
    C. Recommendations on Test set

1: Data Processing

The Data

For this project I will be doing on amazon reviews dataset. The list of such dataset repository can be found here. (https://s3.amazonaws.com/amazon-reviews-pds/readme.html) This dataset is a set of multiple Product reviews bought in UK on amazon. This dataset is of size ~333 MB, so its a mid-range dataset.

DATA COLUMNS:

```
marketplace       - 2 letter country code of the marketplace where the review was writte
n.
customer_id       - Random identifier that can be used to aggregate reviews written by a
single author.
review_id         - The unique ID of the review.
product_id        - The unique Product ID the review pertains to. In the multilingual da
taset the reviews
                for the same product in different countries can be grouped by the same p
roduct_id.
product_parent    - Random identifier that can be used to aggregate reviews for the same
product.
product_title     - Title of the product.
product_category  - Broad product category that can be used to group reviews
                (also used to group the dataset into coherent parts).
star_rating       - The 1-5 star rating of the review.
helpful_votes     - Number of helpful votes.
total_votes       - Number of total votes the review received.
vine              - Review was written as part of the Vine program.
verified_purchase - The review is on a verified purchase.
review_headline   - The title of the review.
review_body       - The review text.
review_date       - The date the review was written.
```

DATA FORMAT

```
Tab ('\t') separated text file, without quote or escape characters.
First line in each file is header; 1 line corresponds to 1 record.
```

First Step: Imports

Importing all necessary libraries needed in this project.

In [1]:
```python
import gzip
from collections import defaultdict
import random
import numpy as np
import scipy.optimize
import string
import nltk
from sklearn import linear_model
from nltk.stem.porter import PorterStemmer # Stemming
```

1: Read the data and Fill your dataset

1. Type Casting some of the features.
2. Converting any boolean responses to True/False.

In [2]:
```python
path = "amazon_reviews_multilingual_UK_v1_00.tsv.gz"

f = gzip.open(path, 'rt', encoding="utf8")

header = f.readline()
header = header.strip().split('\t')

# print(header)
dataset = []

for line in f:
    fields = line.strip().split('\t')
    d = dict(zip(header, fields))
    d['star_rating'] = int(d['star_rating'])
    d['helpful_votes'] = int(d['helpful_votes'])
    d['total_votes'] = int(d['total_votes'])
    for field in ['verified_purchase','vine']:
        if d[field] == 'Y':
            d[field]=True
        else:
            d[field]=False
    dataset.append(d)
```

In [45]:
```
dataset[10]
```

Out[45]:
```
{'marketplace': 'UK',
 'customer_id': '28026896',
 'review_id': 'R4CP7B77ADSJ3',
 'product_id': 'B003TML0VO',
 'product_parent': '838418618',
 'product_title': 'Guitar Heaven: Santana Performs The Greatest Guitar Classics Of
All Time',
 'product_category': 'Music',
 'star_rating': 2,
 'helpful_votes': 0,
 'total_votes': 0,
 'vine': False,
 'verified_purchase': True,
 'review_headline': 'Ok',
 'review_body': 'Ok have bought better.',
 'review_date': '2015-01-18'}
```

2: Split the data into a Training and Testing set

Have Training be the first 80%, and testing be the remaining 20%.

In [3]:
```
#2107824 526957
# Lengths should be: 2107824 526957
random.shuffle(dataset)

N = len(dataset)
trainingSet = dataset[:4*N//5]
testingSet = dataset[4*N//5:]

print("Training Set: ",len(trainingSet), "Test Set: ",len(testingSet), "Total no.of
rows",N)
# print("Lengths should be: 2107824 526957")
```

```
Training Set:  1365995 Test Set:  341499 Total no.of rows 1707494
```

3: Extracting Basic Statistics

Next calculate the answer to some statistic questions all based on the **Training Set:**

1. What is the **average rating**?
2. What fraction of reviews are from **verified purchases**?
3. How many **total users** are there?
4. How many **total items** are there?
5. What fraction of reviews have **5-star ratings**?

In [4]:
```
d_star = [d['star_rating'] for d in trainingSet]
avg_rating = np.average(d_star)
print("1. ",avg_rating)

d_ver = [d['verified_purchase'] for d in trainingSet if d['verified_purchase'] ==Tr
ue ]
frac_reviews = (len(d_ver)/len(trainingSet))*100
print("2. ",frac_reviews)

# This way it takes unique customer_id and product_id
users = set()
for d in trainingSet:
    users.add(d['customer_id'])
print("3. ",len(users))
items = set()
for d in trainingSet:
    items.add(d['product_id'])
print("4. ",len(items))

d_five = [d['star_rating'] for d in trainingSet if d['star_rating'] ==5 ]
frac_five = (len(d_five)/len(trainingSet))*100
print("5. ",frac_five)
```

```
1.  4.379938433156783
2.  76.2219481037632
3.  797681
4.  54954
5.  67.1279177449405
```

2: Classification

Perform classification to extract features and make predictions based on them. Here I will be using a Logistic Regression Model.

1: Define the feature function

This implementation will be based on the **star rating** and the ***length*** of the **review body**.

In [10]:
```
#GIVEN for 1.
# wordCount = defaultdict(int)
# punctuation = set(string.punctuation)

# #GIVEN for 2.
# # wordCountStem = defaultdict(int)
#  print(len(wordCount))

# counts = [(wordCount[w],w) for w in wordCount]
# words = [x[1] for x in counts]
# wordid = dict(zip(words,range(len(words))))
# for d in dataset:
#     f = ''.join([c for c in d['text'].lower() if not c in punctuation])
#     for w in r.split():
#         w = stemmer.stem(w) # with stemming
#         wordCount[w] += 1
# stemmer.stem()
#     features = [0]*len(words)
#     global f
#     for w in f.split():
#         if w in words:
#             features[wordid[w]]+=1
#     features.append(1)

wordCount = defaultdict(int)
stemmer = PorterStemmer() #use stemmer.stem(stuff)
for d in trainingSet:
    f = ''.join([x for x in d['review_body'].lower() if not x in string.punctuation
])
for w in f.split():
    w = stemmer.stem(w) # with stemming
    wordCount[w]+=1


def feature(dat):
    feat = [1, dat['star_rating'], len(wordCount)]
    return feat
```

2: Fit your model

1. Creating a **Feature Vector** based on the feature function defined above.
2. Creating a **Label Vector** based on the "verified purchase" column from the training set.
3. Defining a model i.e; **Logistic Regression** model.
4. Fitting the model.

In [11]:
```python
X = [feature(d) for d in trainingSet]
y = [d['verified_purchase'] for d in trainingSet]

# print("Label: ", y[:100], "\nFeatures:", X[:10])
model = linear_model.LogisticRegression()
model.fit(X, y)
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/logistic.py:432: Futur
eWarning: Default solver will be changed to 'lbfgs' in 0.22. Specify a solver to s
ilence this warning.
  FutureWarning)
```

Out[11]:
```
LogisticRegression(C=1.0, 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)
```

3: Compute Accuracy of Your Model

1. Make **Predictions** based on the model.
2. Compute the **Accuracy** of the model.

In [12]:
```python
#YOUR CODE HERE without stemming: 0.7619478841430606  with stemming: 0.761947884143
0606
predictions = model.predict(X)
# predictions
correctPredictions = predictions == y
accuracy = sum(correctPredictions) / len(correctPredictions)
accuracy
```

Out[12]:
```
0.761759742898034
```

4: Finding the Balanced Error Rate

1. Compute **True** and **False Positives**
2. Compute **True** and **False Negatives**
3. Compute **Balanced Error Rate** based on the above defined variables.

In [13]:
```
#YOUR CODE HERE
TP = sum([(p and l) for (p,l) in zip(predictions, y)])
FP = sum([(p and not l) for (p,l) in zip(predictions, y)])
TN = sum([(not p and not l) for (p,l) in zip(predictions, y)])
FN = sum([(not p and l) for (p,l) in zip(predictions, y)])

TFaccuracy = (TP + TN) / (TP + FP + TN + FN)
TPR = TP / (TP + FN)
TNR = TN / (TN + FP)
BER = 1 - 1/2 * (TPR + TNR)
print("TP:",TP,"\nFP:",FP,"\nTN:",TN,"\nFN:",FN,"\nTF accuracy(should be equal to a
bove accuracy):",TFaccuracy,"\nBalanced Error rate:",BER)
```

```
TP: 1040560
FP: 325435
TN: 0
FN: 0
TF accuracy(should be equal to above accuracy): 0.761759742898034
Balanced Error rate: 0.5
```

3: Regression

Alter the features to differentiate.

Here I will be using word ID's and star rating as feature vectors.

In [14]:
```
y = [d['star_rating'] for d in trainingSet]
```

1: Unique Words in a Sample Set

I will take a smaller Sample Set here, as stemming on the normal training set will take a very long time.

1. Count the number of unique words found within the 'review body' portion of the sample set defined below, making sure to **Ignore Punctuation and Capitalization**.
2. Count the number of unique words found within the 'review body' portion of the sample set defined below, this time with use of **Stemming, Ignoring Puctuation, *and* Capitalization**.

In [15]:
```
#GIVEN for 1.
wordCount = defaultdict(int)
punctuation = set(string.punctuation)

#GIVEN for 2.
wordCountStem = defaultdict(int)
stemmer = PorterStemmer() #use stemmer.stem(stuff)
```

In [16]:
```
sampleSet = trainingSet[:2*len(trainingSet)//10]
```

In [17]:
```python
for d in sampleSet:
    f = ''.join([x for x in d['review_body'].lower() if not x in punctuation])
for w in f.split():
    w = stemmer.stem(w) # with stemming
    wordCountStem[w]+=1
for w in f.split():
#     w = stemmer.stem(w) # with stemming
    wordCount[w]+=1

counts = [(wordCount[w],w) for w in wordCount]
counts_stem = [(wordCountStem[w],w) for w in wordCountStem]

words = [x[1] for x in counts]
words_stem = [x[1] for x in counts_stem]
print("wordCount:",words,len(words),"\nwordStem Count:",words_stem,len(words_stem))
```

```
wordCount: ['great', 'cd'] 2
wordStem Count: ['great', 'cd'] 2
```

2: Evaluating Classifiers

1. Given the feature function and counts vector, **Define** a X vector.
2. **Fit** the model using a **Ridge Model** with (alpha = 1.0, fit_intercept = True).
3. Using the model, **Make your Predictions**.
4. Find the **MSE** between resulted predictions and y vector.

In [18]:
```python
#GIVEN FUNCTIONS
def feature_reg(datum):
    feat = [0]*len(words)
    r = ''.join([c for c in datum['review_body'].lower() if not c in punctuation])
    for w in r.split():
        if w in wordSet:
            feat[wordId[w]] += 1
    feat.append(1) #offset
    return feat

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

In [19]:
```python
#GIVEN COUNTS AND SETS
counts = [(wordCount[w], w) for w in wordCount]
counts.sort()
counts.reverse()

#Note: increasing the size of the dictionary may require a lot of memory
words = [x[1] for x in counts[:100]]

wordId = dict(zip(words, range(len(words))))
wordSet = set(words)
```

In [20]:
```
random.shuffle(trainingSet)
X = [feature_reg(d) for d in trainingSet]

model = linear_model.Ridge(alpha = 1.0, fit_intercept = True)
model.fit(X, y)
predictions = model.predict(X)

def MSE(model, X, y):
    predictions = model.predict(X)
    differences = [(a-b)**2 for (a,b) in zip(predictions, y)]
    return sum(differences) / len(differences)

MSE(model, X, y)
```

Out[20]: 1.1852399967940734

In [21]:
```
# If you would like to work with this example more in your free time, here are some tips to improve your solution:
# 1. Implement a validation pipeline and tune the regularization parameter
# 2. Alter the word features (e.g. dictionary size, punctuation, capitalization, stemming, etc.)
# 3. Incorporate features other than word features
```

4: Recommendation Systems

For this final task, you will see a simple latent factor-based recommender systems to make predictions. Then evaluating the performance of this predictions.

In [5]:
```
#Create and fill our default dictionaries for our dataset
reviewsPerUser = defaultdict(list)
reviewsPerItem = defaultdict(list)

for d in trainingSet:
    user,item = d['customer_id'], d['product_id']
    reviewsPerUser[user].append(d)
    reviewsPerItem[item].append(d)

#Create two dictionaries that will be filled with our rating prediction values
userBiases = defaultdict(float)
itemBiases = defaultdict(float)

#Getting the respective lengths of our dataset and dictionaries
N = len(trainingSet)
nUsers = len(reviewsPerUser)
nItems = len(reviewsPerItem)

#Getting the list of keys
users = list(reviewsPerUser.keys())
items = list(reviewsPerItem.keys())

labels = [d['star_rating'] for d in trainingSet]
```

1: Calculate the ratingMean

1. Find the **average rating** of the training set.
2. Calculate a **baseline MSE value** from the actual ratings to the average ratings.

In [6]:
```
alpha = sum([d['star_rating'] for d in trainingSet]) / len(trainingSet)
# alpha = np.reshape(-1,1)
alwaysPredictMean = [alpha for d in dataset]

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

# labels = [d['star_rating'] for d in trainingSet]
# print(labels[:100])
print("Rating mean: ",alpha)
print("MSE: ",MSE(alwaysPredictMean, labels))
```

```
Rating mean:  4.379938433156783
MSE:  1.1841831286457427
```

In [7]:
```
userGamma = {}
itemGamma = {}
K = 2 #Dimensionality of gamma

for u in reviewsPerUser:
    userGamma[u] = [random.random() * 0.1 - 0.05 for k in range(K)]

for i in reviewsPerItem:
    itemGamma[i] = [random.random() * 0.1 - 0.05 for k in range(K)]
```

Here are some functions defined to optimize the above MSE value.

In [8]:

```
# alpha = ratingMean
def unpack(theta):
    global alpha
    global userBiases
    global itemBiases
    global userGamma
    global itemGamma
    index = 0
    alpha = theta[index]
    index += 1
    userBiases = dict(zip(users, theta[index:index+nUsers]))
    index += nUsers
    itemBiases = dict(zip(items, theta[index:index+nItems]))
    index += nItems
    for u in users:
        userGamma[u] = theta[index:index+K]
        index += K
    for i in items:
        itemGamma[i] = theta[index:index+K]
        index += K

def inner(x, y):
    return sum([a*b for a,b in zip(x,y)])


def prediction(user, item):
    return alpha + userBiases[user] + itemBiases[item] + inner(userGamma[user], ite
mGamma[item])


def cost(theta, labels, lamb):
    unpack(theta)
    predictions = [prediction(d['customer_id'], d['product_id']) for d in trainingS
et]
    cost = MSE(predictions, labels)
    print("MSE = " + str(cost))
    for u in users:
        cost += lamb*userBiases[u]**2
        for k in range(K):
            cost += lamb*userGamma[u][k]**2
    for i in items:
        cost += lamb*itemBiases[i]**2
        for k in range(K):
            cost += lamb*itemGamma[i][k]**2
    return cost


def derivative(theta, labels, lamb):
    unpack(theta)
    N = len(trainingSet)
    dalpha = 0
    dUserBiases = defaultdict(float)
    dItemBiases = defaultdict(float)
    dUserGamma = {}
    dItemGamma = {}
    for u in reviewsPerUser:
        dUserGamma[u] = [0.0 for k in range(K)]
    for i in reviewsPerItem:
        dItemGamma[i] = [0.0 for k in range(K)]
```

```
    for d in trainingSet:
        u,i = d['customer_id'], d['product_id']
        pred = prediction(u, i)
        diff = pred - d['star_rating']
        dalpha += 2/N*diff
        dUserBiases[u] += 2/N*diff
        dItemBiases[i] += 2/N*diff
        for k in range(K):
            dUserGamma[u][k] += 2/N*itemGamma[i][k]*diff
            dItemGamma[i][k] += 2/N*userGamma[u][k]*diff
    for u in userBiases:
        dUserBiases[u] += 2*lamb*userBiases[u]
        for k in range(K):
            dUserGamma[u][k] += 2*lamb*userGamma[u][k]
    for i in itemBiases:
        dItemBiases[i] += 2*lamb*itemBiases[i]
        for k in range(K):
            dItemGamma[i][k] += 2*lamb*itemGamma[i][k]
    dtheta = [dalpha] + [dUserBiases[u] for u in users] + [dItemBiases[i] for i in
items]
    for u in users:
        dtheta += dUserGamma[u]
    for i in items:
        dtheta += dItemGamma[i]
    return np.array(dtheta)
```

2: Optimize

1. **Optimize** the above MSE using the scipy.optimize.fmin_1_bfgs_b("arguments") functions.

In [9]:
```
scipy.optimize.fmin_l_bfgs_b(cost, [alpha] + # Initialize alpha
                                   [0.0]*(nUsers+nItems) + # Initialize beta
                                   [random.random() * 0.1 - 0.05 for k in range(K*(
nUsers+nItems))], derivative,
                             args = (labels, 0.001))
```

```
MSE = 1.1841849224291974
MSE = 1.181288667550596
MSE = 1.1710962556469446
MSE = 101.85112437814225
MSE = 1.1878040176514137
MSE = 1.1644660269237566
MSE = 1.1380740009422736
MSE = 1.137450892993197
MSE = 1.1383953066158647
MSE = 1.1402806036412134
MSE = 1.140799705830948
MSE = 1.1410790537243674
MSE = 1.1411118599148247
MSE = 1.1411078990215409
```

Out[9]:
```
(array([ 4.38379472e+00,  9.09613047e-04, -3.19293241e-04, ...,
         1.79226023e-07,  1.98776919e-07, -3.33638325e-07]),
 1.1574362235405544,
 {'grad': array([ 2.29508812e-06, -3.51997289e-10, -3.12451242e-10, ...,
          3.66167954e-10,  3.97248128e-10, -6.66952398e-10]),
  'task': b'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL',
  'funcalls': 14,
  'nit': 11,
  'warnflag': 0})
```

3: Recommending Products

Based on similarities in trainingSet Recommendations were made on TestingSet.

In [33]:
```python
usersPerItem = defaultdict(set)
itemsPerUser = defaultdict(set)
itemNames = {}

for d in trainingSet:
    user,item = d['customer_id'], d['product_id']
    usersPerItem[item].add(user)
    itemsPerUser[user].add(item)
    itemNames[item] = d['product_title']

def Jaccard(s1, s2):
    numer = len(s1.intersection(s2))
    denom = len(s1.union(s2))
    return numer / denom

def mostSimilar(iD, n):
    similarities = []
    id_list = []
    users = usersPerItem[iD]
    for i2 in usersPerItem:
        if i2 == iD: continue
        sim = Jaccard(users, usersPerItem[i2])
        similarities.append((sim,i2))
    similarities.sort(reverse=True)

    for i in similarities:
        id_list.append(i[1])
    print(id_list[:n])
    return similarities[:n]

# def predictRating(user,item):
#     ratings = []
#     similarities = []
#     for d in reviewsPerUser[user]:
#         i2 = d['product_id']
#         if i2 == item: continue
#         ratings.append(d['star_rating'])
#         similarities.append(Jaccard(usersPerItem[item],usersPerItem[i2]))
#     if (sum(similarities) > 0):
#         weightedRatings = [(x*y) for x,y in zip(ratings,similarities)]
#         return sum(weightedRatings) / sum(similarities)
#     else:
#         # User hasn't rated any similar items
#         return ratingMean
```

In [43]:
```python
query = testingSet[10]['product_id']
# query1 = testingSet['product_id']

print(query)
```

```
0330517945
```

In [39]:
```
mostSimilar(query, 10)
```

```
['0330517937', '0330517953', '0230748260', '0230748252', '0330419080', '033041907
2', '0330419099', 'B004XBOANU', '0230748236', '0857207539']
```

Out[39]:
```
[(0.15421686746987953, '0330517937'),
 (0.07736389684813753, '0330517953'),
 (0.019390581717451522, '0230748260'),
 (0.0160857908847185, '0230748252'),
 (0.012448132780082987, '0330419080'),
 (0.011811023622047244, '0330419072'),
 (0.008658008658008658, '0330419099'),
 (0.007782101167315175, 'B004XBOANU'),
 (0.006734006734006734, '0230748236'),
 (0.00641025641025641, '0857207539')]
```

In [40]:
```
[itemNames[x[1]] for x in mostSimilar(query, 10)]
```

```
['0330517937', '0330517953', '0230748260', '0230748252', '0330419080', '033041907
2', '0330419099', 'B004XBOANU', '0230748236', '0857207539']
```

Out[40]:
```
['The Sins of the Father (The Clifton Chronicles)',
 'Be Careful What You Wish For (The Clifton Chronicles)',
 'Mightier than the Sword (The Clifton Chronicles)',
 'Be Careful What You Wish For (The Clifton Chronicles)',
 'The Fourth Estate',
 'The Eleventh Commandment',
 'To Cut A Long Story Short',
 'Little Voice [DVD]',
 'The Sins of the Father (The Clifton Chronicles)',
 "The White Princess (COUSINS' WAR)"]
```

Conclusion:

1. MSE after optimizing has slightly better result than average rating.
2. Recommendations on products was done here; Predicting ratings can be done further.