In [15]:

```
import gzip
from collections import defaultdict
from sklearn import linear_model
import csv
import random
import numpy as np
dataDir = "/Users/Judy-Ccino412/Desktop/cookdata"

##### Would Cook Prediction #####
def readGz(path):
    for l in gzip.open(path, 'rt'):
        yield eval(l)

def readCSV(path):
    f = gzip.open(path, 'rt')
    c = csv.reader(f)
    header = next(c)
    for l in c:
        d = dict(zip(header,l))
        yield d['user_id'],d['recipe_id'],d

# Jaccard Similarity
def Jaccard(s1, s2):
    numer = len(s1.intersection(s2))
    denom = len(s1.union(s2))
    if denom == 0:
        return 0
    return numer / denom

# few utility features
allRatings = []
userRatings = defaultdict(list)
data = []
for user,recipe,d in readCSV(dataDir + "trainInteractions.csv.gz"):
    data.append(d)
    r = int(d['rating'])
    allRatings.append(r)
    userRatings[user].append(r)

r_data = {}
mins_data = {}
steps_data = {}
for d in readGz("trainRecipes.json.gz"):
    r = d['recipe_id']
    i = d['ingredients']
    s = d['steps']
    mi = d['minutes']
    r_data[r] = i
    mins_data[r] = mi
    steps_data[r] = s
```

```
# Reviews 1-400,000 for training
# Reviews 400,000-500,000 for validation
training = data[:400000]
validation = data[400000:]

# Extract a few utility data structures from validation set
usersPerRecipe = defaultdict(set) # Maps a recipe to the users who cooked it
recipesPerUser = defaultdict(set) # Maps a user to the recipe that they cooke
dates = {}
ratingDict = {} # To retrieve a rating for a specific user/recipe pair

for d in validation:
    user, recipe = d['user_id'], d['recipe_id']
    usersPerRecipe[recipe].add(user)
    recipesPerUser[user].add(recipe)
    ratingDict[(user,recipe)] = d['rating']

# positive pairs
new_validation1 = []
for v in validation:
    new_validation1.append((v['user_id'], v['recipe_id']))

# all recipes in dataset
all_recipes = list(set([d['recipe_id'] for d in data]))

# negative pairs
new_validation2 = []
for pair in new_validation1:
    u = pair[0]
    neg = random.sample(all_recipes,1)[0]
    while neg in recipesPerUser[u]:
        neg = random.sample(all_recipes,1)[0]
    neg_pair = (u, neg)
    new_validation2.append(neg_pair)

# generate new validation set
new_validation = new_validation1 + new_validation2

# define popularity
recipeCount = defaultdict(int)
totalCooked = 0

for user,recipe,_ in readCSV(dataDir + "trainInteractions.csv.gz"):
    recipeCount[recipe] += 1
    totalCooked += 1

mostPopular = [(recipeCount[x], x) for x in recipeCount]
mostPopular.sort()
mostPopular.reverse()

new_return1 = set()
count = 0
for ic, i in mostPopular:
```

```
    count += ic
    new_return1.add(i)
    if count > totalCooked * 0.69:  # a better threshold for popularity
        break
```

In [16]:

```
########## train classifier ########
# classfier includes following features:
# Popularity (> 0.69 percentile) and Jaccard Similarity (> 0.9)
# number of ingredients, time, steps combined

# popularity vector
pop = []
for u, i in new_validation:
    if i in new_return1:
        pop.append(1)
    else:
        pop.append(0)

# n-ingredients vector, time vector, step vector
feat = []
for u, i in new_validation:
    n = len(r_data[i])
    mi = mins_data[i]
    st = len(steps_data[i])
    if n > 10 or mi > 200 or st > 300:
        pre = 0
    else:
        pre = 1
    feat.append(pre)

# sim vector
sims_bi = []
for u,g in new_validation:
    recipes_of_u = recipesPerUser[u] # all training items g' that user u has

    sim_list = [0]
    for g1 in recipes_of_u: # for each, compute the Jaccard similarity betwee
        s1 = usersPerRecipe[g] # users (in the training set) who have made g
        s2 = usersPerRecipe[g1]
        sim = Jaccard(s1, s2)
        sim_list.append(sim)

    if max(sim_list) > 0.9 or g in new_return1:
        popsim = 1
    else: popsim = 0
    sims_bi.append(popsim)

# feature vectors
X = np.matrix([[1,p,f] for p,f in zip(sims_bi,feat)])
mod = linear_model.LogisticRegression(C=1.0, class_weight='balanced')
validation_labels = [1 for i in range(100000)] + [0 for i in range(100000)]
mod.fit(X,validation_labels)
```

```
####### predict on test #########
res = []
predictions = open(dataDir + "predictions_Made.txt", 'w')
for l in open(dataDir + "stub_Made.txt"):
    if l.startswith("user_id"):
        #header
        predictions.write(l)
        continue

    u,g = l.strip().split('-')
    recipes_of_u = recipesPerUser[u]

    # popularity
    if g in new_return1:
        pop = 1
    else: pop = 0

    # jaccard
    sim_list = [0]
    for g1 in recipes_of_u: # for each, compute the Jaccard similarity betwee
        s1 = usersPerRecipe[g] # users (in the training set) who have made g
        s2 = usersPerRecipe[g1]
        sim = Jaccard(s1, s2)
        sim_list.append(sim)

    if pop == 1 or max(sim_list) > 0.9:
        popsim = 1
    else: popsim = 0

    # time
    t = mins_data[g]

    # n-ingred
    n = len(r_data[g])

    # steps
    s = len(steps_data[g])

    if n > 10 or t > 200 or s > 300:
        pre = 0
    else:
        pre = 1

    X = np.matrix([1,popsim,pre])
    pred = mod.predict(X)[0]
    res.append(pred)
    predictions.write(u + '-' + g + "," + str(pred) + "\n")

predictions.close()
```

Out[16]: LogisticRegression(class_weight='balanced')

Cooktime Prediction

In [13]:

```
def readGz(path):
    for l in gzip.open(path, 'rt'):
        yield eval(l)

def readCSV(path):
    f = gzip.open(path, 'rt')
    c = csv.reader(f)
    header = next(c)
    for l in c:
        d = dict(zip(header,l))
        yield d['user_id'],d['recipe_id'],d

data = []
for d in readGz(dataDir + 'trainRecipes.json.gz'):
    data.append(d)

# Reviews 1-190,000 for training
training = data[:190000]

# Ignore capitalization and remove punctuation
wordCount = defaultdict(int)
punctuation = set(string.punctuation)
for d in training:
    r = ''.join([c for c in d['steps'].lower() if not c in punctuation])
    for w in r.split():
        wordCount[w] += 1

counts = [(wordCount[w], w) for w in wordCount]
counts.sort()
counts.reverse()

# 4,000 most common words in the training set
bigger_words = [x[1] for x in counts[:4000]]
wordId = dict(zip(bigger_words, range(len(bigger_words))))
wordSet = set(bigger_words)

# Build bag-of-words feature vectors by counting the instances of these 4,000
def feature(datum):
    feat = [0]*len(bigger_words)
    r = ''.join([c for c in datum['steps'].lower() if not c in punctuation])
    for w in r.split():
        if w in bigger_words:
            feat[wordId[w]] += 1
    feat.append(1) # offset
    return feat

# Extract bag-of-word features in training
X_train = [feature(d) for d in training]
y_train = [d['minutes'] for d in training]
```

In [42]:
```
pl = Pipeline([('regressor', linear_model.Ridge(alpha = 1.0,
                                                fit_intercept=False,
                                                normalize = False))])
parameters = {'regressor__alpha': [200, 230, 250, 280, 320, 400]}
# I used grid search to find the best alpha = 400 for Ridge regression
grids = GridSearchCV(pl, param_grid=parameters, cv=4, return_train_score=True
grids.fit(X_train, y_train);
grids.best_params_['regressor__alpha']
```

Out[42]: 400

In [43]:
```
#### fit regressor
# Regularized regression
clf = linear_model.Ridge(400, fit_intercept=False) # MSE + 400 l2
clf.fit(X_train, y_train)

#### predict on test set
predictions = open("predictions_Minutes.txt", 'w')
predictions.write("recipe_id,prediction\n")
for d in readGz("testRecipes.json.gz"):
    x = feature(d)
    pred = clf.predict([x])[0]
    # if there is a negative prediction, predict 30 min instead
    # (which is close to the mean of the cooktime)
    if pred < 0:
        pred = 30
        predictions.write(d['recipe_id'] + ',' + str(pred) + '\n')

predictions.close()
```

Out[43]: Ridge(alpha=400, fit_intercept=False)

Kaggle performace: 3006.67779