Stochastic gradient descent on beer reviews

I collaborated with Ruben Abbou.

In [69]:
```
import numpy as np
import json
import re
import matplotlib.pyplot as plt
from scipy.sparse import csr_matrix
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
import time
import random
from math import exp
```

In [70]:
```
with open('/project2/cmsc25025/beer_review/labeled.json', 'r') as f:
    brv = json.loads(f.read())
```

Part 1: Data inspection.

To warm up, check the mean, median and standard deviation of the overall ratings for each beer and brewer. Do you think people have similar taste?

In [71]:
```
beers = set([br['beer_name'] for br in brv])
brewers = set([br['brewer'] for br in brv])
```

In [72]:
```
beer_values = {}
for b in beers:
    beer_values[b] = []
for b in brv:
    beer_values[b['beer_name']].append(b['overall'])
beer_statistics = {}
for k,v in beer_values.items():
    beer_statistics[k] = {'sd': np.std(v), 'median': np.median(v), 'mea
n':np.mean(v)}
```

In [73]:
```
brewer_values = {}
for b in brewers:
    brewer_values[b] = []
for b in brv:
    brewer_values[b['brewer']].append(b['overall'])
brewer_statistics = {}
for k,v in brewer_values.items():
    brewer_statistics[k] = {'sd': np.std(v), 'median': np.median(v), 'me
an':np.mean(v)}
```

In [78]:
```
#For the sake of not printing too much stuff, I only print the first 100
items.
for k,v in list(beer_statistics.items())[:100]:
    print("Beer: " + str(k) +"\n"
          + "mean: " + str(v['mean'])
          + " median: " + str(v['median'])
          + " sd: " + str(v['sd']))
print("\n")
for k,v in list(brewer_statistics.items())[:100]:
    print("Brewer: " + str(k) +"\n"
          + "mean: " + str(v['mean'])
          + " median: " + str(v['median'])
          + " sd: " + str(v['sd']))
```

```
Beer: Schoune La Trip des Schoune
mean: 12.17910447761194 median: 13.0 sd: 2.849065692019515
Beer: Full Circle Euro-Fuggle
mean: 12.285714285714286 median: 11.0 sd: 3.1036515689143473
Beer: Blue Frog 10th Anniversary Ale
mean: 15.166666666666666 median: 16.0 sd: 2.266911751455907
Beer: Quay Street Blue Water Pale Ale
mean: 12.0 median: 11.5 sd: 1.632993161855452
Beer: Captain Lawrence Barrel Select Cherry
mean: 14.0 median: 14.5 sd: 1.632993161855452
Beer: Lost Abbey Gift of the Magi
mean: 15.01126126126126 median: 15.0 sd: 1.7313638935078186
Beer: John Harvards Brewers Gold
mean: 10.0 median: 10.0 sd: 0.0
Beer: Kelham Island Island
mean: 13.4 median: 13.0 sd: 0.48989794855663565
Beer: Hidden Fantasy
mean: 12.0625 median: 12.0 sd: 2.0757152381769517
Beer: Red Rock Czech Pilsner
mean: 9.5 median: 9.0 sd: 2.362907813126304
Beer: Bull Falls March Madness Ale
mean: 15.0 median: 15.0 sd: 0.0
Beer: Freisinger Schwarzbier
mean: 12.307692307692308 median: 12.0 sd: 1.6817854699288806
Beer: BryggeriKAIA Bayer
mean: 14.0 median: 14.0 sd: 0.0
Beer: Bayhawk Stout
mean: 16.0 median: 16.0 sd: 1.0
Beer: Claymore Lager
mean: 9.0 median: 9.0 sd: 0.0
Beer: Ritter Schwarz
mean: 14.555555555555555 median: 15.0 sd: 2.408831487630978
Beer: Three Boys Porter
mean: 13.36 median: 13.0 sd: 1.8736061485808588
Beer: Welde Remix
mean: 5.833333333333333 median: 6.0 sd: 2.074983266331455
Beer: Wye Valley Rapid Ale
mean: 12.426229508196721 median: 13.0 sd: 1.928721791914188
Beer: Cottage Planet Ale
mean: 13.0 median: 13.0 sd: 0.0
Beer: Earth We Heavy Yo
mean: 15.133333333333333 median: 15.0 sd: 1.024152766382481
Beer: Kenbach Beer
mean: 6.0 median: 6.0 sd: 0.0
Beer: Little Valley Yorkshire Hemp
mean: 9.5 median: 9.5 sd: 3.5
Beer: Dark Star Summer Meltdown
mean: 13.0 median: 13.5 sd: 2.700762419587999
Beer: North Country Abbey Holiday Ale
mean: 9.0 median: 9.0 sd: 0.0
Beer: Kronen Gold-Export
mean: 12.166666666666666 median: 12.5 sd: 1.674979270186815
Beer: Funfair Candy Floss
mean: 15.0 median: 15.0 sd: 0.0
Beer: Delafield Leisure Beer
mean: 11.5 median: 11.5 sd: 2.5
Beer: Coronado Seasons Best Winter Brew
```

```
mean: 14.0 median: 14.0 sd: 1.591644851508443
Beer: Papa Murphys Irish Amber Ale
mean: 13.6 median: 14.0 sd: 1.4966629547095764
Beer: Empire Royal Mead
mean: 15.56 median: 16.0 sd: 1.820183140968696
Beer: Elegancia Clasico
mean: 16.0 median: 16.0 sd: 0.0
Beer: Victory CBC Saison
mean: 14.0 median: 14.0 sd: 1.0289915108550531
Beer: Brau Brothers Extra Special Bitter
mean: 13.473684210526315 median: 14.0 sd: 2.962367848309534
Beer: La Jolla Brewhouse Pilsner
mean: 14.0 median: 14.0 sd: 0.0
Beer: Dempseys Petaluma Strong Ale
mean: 13.8 median: 14.0 sd: 1.3266499161421599
Beer: Wabash Pilsner
mean: 10.0 median: 10.0 sd: 0.0
Beer: Shepherd Neame Tapping The Admiral &#40;Bottle&#41;
mean: 11.555555555555555 median: 12.0 sd: 1.0657403385139377
Beer: Weasel Boy Feisty Fisher Amber Ale
mean: 13.0 median: 13.0 sd: 0.816496580927726
Beer: Bristol Beer Factory Ultimate Stout
mean: 14.74074074074074 median: 15.0 sd: 1.4035033571478825
Beer: The Civil Life Black Ale
mean: 15.0 median: 15.0 sd: 0.816496580927726
Beer: Il Vicino Sweet Sanderine Porter
mean: 14.555555555555555 median: 14.0 sd: 0.9558139185602919
Beer: Silverado Scottish Ale with Heather Tips
mean: 10.0 median: 10.0 sd: 0.0
Beer: Wormtown Beer Goggles Barleywine
mean: 15.0 median: 15.0 sd: 0.0
Beer: Saint-Bock 666
mean: 13.7 median: 13.0 sd: 1.3453624047073711
Beer: Barhop Judge Porter
mean: 13.333333333333334 median: 14.0 sd: 1.3333333333333333
Beer: Mallinsons Lynx
mean: 13.0 median: 13.0 sd: 0.0
Beer: 5 Seasons North AK-47 Mild Ale
mean: 13.6 median: 13.0 sd: 1.2
Beer: Goose Island Extremely Naughty Goose
mean: 15.066666666666666 median: 15.0 sd: 1.4360439485692011
Beer: Wedge Derailed iHemp Ale
mean: 14.333333333333334 median: 14.5 sd: 1.9720265943665385
Beer: Casco Bay Carrabassett IPA
mean: 11.88888888888889 median: 11.0 sd: 2.5579698740491863
Beer: Michelbacher Schwarzer Adler Dunkles Export
mean: 12.0 median: 11.5 sd: 1.224744871391589
Beer: North By Northwest Guldenbiere &#40;2005, 2007 and later&#41;
mean: 15.074074074074074 median: 15.0 sd: 1.8242385712629259
Beer: Shenandoah Blueberry Blonde
mean: 10.5 median: 10.5 sd: 2.5
Beer: Durham St Cuthbert
mean: 13.909090909090908 median: 14.0 sd: 1.888780853605748
Beer: Pittsburgh Gold Crown Premium Beer
mean: 1.0 median: 1.0 sd: 0.0
Beer: Sierra Madre Brewing Regio Light
mean: 12.75 median: 11.5 sd: 4.437059837324712
```

```
Beer: Blue Mountain Blue Reserve 2010
mean: 14.6 median: 15.0 sd: 1.8547236990991407
Beer: Big Dogs Red Hydrant Ale
mean: 11.777777777777779 median: 12.0 sd: 1.6850834320114556
Beer: Turtle Mountain Heather Scotch
mean: 15.5 median: 15.5 sd: 0.5
Beer: Saxer JackFrost Winter Doppelbock
mean: 13.044117647058824 median: 13.0 sd: 2.3975098817769225
Beer: North Peak Berserker
mean: 17.0 median: 17.0 sd: 0.0
Beer: Old Testy Stout
mean: 13.2 median: 15.0 sd: 3.655133376499413
Beer: Sweetwater Tavern Black IPA
mean: 13.0 median: 13.0 sd: 0.0
Beer: Amalgamated Imperial Oktoberfest &#40;Marzen&#41;
mean: 15.0 median: 15.0 sd: 0.0
Beer: Bull & Bush Ice Cream Clone Stout
mean: 15.6 median: 16.0 sd: 1.624807680927192
Beer: Acorn Simcoe IPA
mean: 14.0 median: 14.0 sd: 0.816496580927726
Beer: Maine Mead Works HoneyMaker Dry Hopped Mead
mean: 11.714285714285714 median: 11.0 sd: 3.6140316116210047
Beer: Great Waters Capitol ESB
mean: 14.5 median: 14.5 sd: 0.5
Beer: Brutopia Honey Brown
mean: 10.75 median: 11.0 sd: 2.4585855399279595
Beer: Ithaca Double IPA
mean: 14.492537313432836 median: 15.0 sd: 1.7005103213698292
Beer: Pocono Lager
mean: 10.204545454545455 median: 10.0 sd: 3.0119424829603165
Beer: Hartwall Karjala Terva &#40;6.3% version&#41;
mean: 10.363636363636363 median: 10.0 sd: 2.6148547884611664
Beer: Hartlands Medium Sweet Perry
mean: 14.2 median: 15.0 sd: 1.1661903789690602
Beer: Sarasota Brewing Coriander Wheat
mean: 12.0 median: 12.0 sd: 1.0
Beer: New Old Lompoc Crystal Missile Saison
mean: 11.0 median: 11.0 sd: 0.0
Beer: Samuel Adams Oatmeal Stout
mean: 12.727272727272727 median: 14.0 sd: 3.3053925386723346
Beer: Arcadia Starboard Stout
mean: 12.868312757201647 median: 13.0 sd: 2.125473569337544
Beer: Oggis HoliDazed Ale
mean: 11.0 median: 11.0 sd: 1.0
Beer: Big Dogs Mango Madness
mean: 4.0 median: 4.0 sd: 0.0
Beer: Bushwakker Northern Lights Lager
mean: 11.8 median: 12.0 sd: 2.7129319932501073
Beer: Smuttynose Oak-aged Terminator G-Bock
mean: 14.857142857142858 median: 14.0 sd: 1.8070158058105024
Beer: The Lucky Monk Tritica Wheat
mean: 14.0 median: 13.5 sd: 1.8708286933869707
Beer: Buntingford Honeyway
mean: 13.0 median: 13.0 sd: 0.0
Beer: Moab Brewery Desert Select Scotch Ale
mean: 14.333333333333334 median: 15.0 sd: 1.699673171197595
Beer: Brewers Art 10th Anniversary Ale
```

```
mean: 15.285714285714286 median: 15.0 sd: 0.45175395145262565
Beer: Clarks Code Red
mean: 11.0 median: 11.0 sd: 1.0
Beer: Lambrate Beccamort
mean: 12.962962962962964 median: 13.0 sd: 1.5748364167014295
Beer: Camerons Pomegranate Cream
mean: 14.11111111111111 median: 15.0 sd: 1.3698697784375502
Beer: Ryburn Best Mild
mean: 13.4 median: 13.0 sd: 1.8547236990991407
Beer: Bergadler Premium Pils 3.5
mean: 11.0 median: 11.0 sd: 0.816496580927726
Beer: Bullfrog eSTEAMed Beer
mean: 13.2 median: 13.0 sd: 1.9390719429665317
Beer: Lake Placid Pulpit Rock Dopplebock
mean: 15.0 median: 15.0 sd: 0.0
Beer: Sly Fox Slacker Eisbock
mean: 14.166666666666666 median: 14.5 sd: 0.8975274678557508
Beer: Flying Bison Bird of Prey IPA
mean: 12.692307692307692 median: 13.0 sd: 1.3234346564680965
Beer: Revolution El Bastardo
mean: 13.5 median: 14.0 sd: 2.29128784747792
Beer: Kaiser Wilhelm II Premium Pils
mean: 11.0 median: 11.0 sd: 0.0
Beer: Flying Fish Love Fish
mean: 14.518518518518519 median: 15.0 sd: 1.4998856838012287
Beer: 21st Amendment Belgian Strong Ale
mean: 15.0 median: 15.0 sd: 0.0
Beer: Cascade Amber Wheat
mean: 13.5 median: 13.5 sd: 0.5


Brewer: 0
mean: 12.836223506743737 median: 13.0 sd: 2.5432971325766625
Brewer: 1
mean: 13.251677852348994 median: 14.0 sd: 2.7974845850519783
Brewer: 2
mean: 14.822445170321979 median: 15.0 sd: 2.243541088466132
Brewer: 3
mean: 13.8 median: 14.0 sd: 2.6381811916545836
Brewer: 4
mean: 11.64 median: 12.0 sd: 2.4799999999999995
Brewer: 5
mean: 10.0 median: 10.0 sd: 0.0
Brewer: 6
mean: 9.861111111111111 median: 11.0 sd: 4.619720877678035
Brewer: 7
mean: 9.206106870229007 median: 9.0 sd: 3.174607163897181
Brewer: 8
mean: 12.442857142857143 median: 13.0 sd: 2.168842444950856
Brewer: 9
mean: 9.533333333333333 median: 10.0 sd: 3.630733014450694
Brewer: 10
mean: 14.141904761904762 median: 14.0 sd: 2.2235084939517855
Brewer: 11
mean: 13.867924528301886 median: 14.0 sd: 1.6372749547106897
Brewer: 12
mean: 14.57345971563981 median: 15.0 sd: 2.7332237455074773
```

```
Brewer: 13
mean: 11.0 median: 11.0 sd: 2.0
Brewer: 14
mean: 13.333333333333334 median: 13.0 sd: 0.4714045207910317
Brewer: 15
mean: 6.123473282442748 median: 5.0 sd: 4.4324147087168555
Brewer: 16
mean: 11.9 median: 13.0 sd: 3.3301651610693423
Brewer: 17
mean: 12.831683168316832 median: 13.0 sd: 3.0222583744272926
Brewer: 18
mean: 14.2109337860781 median: 14.0 sd: 2.455233690037572
Brewer: 19
mean: 13.358490566037736 median: 13.0 sd: 1.7385131320436895
Brewer: 20
mean: 13.95 median: 14.0 sd: 2.423324163210527
Brewer: 21
mean: 12.414429278536073 median: 13.0 sd: 2.57923577104781
Brewer: 22
mean: 14.584699453551913 median: 15.0 sd: 2.2917097265342896
Brewer: 23
mean: 5.0 median: 5.0 sd: 0.0
Brewer: 24
mean: 12.75 median: 13.0 sd: 2.165063509461097
Brewer: 25
mean: 9.135135135135135 median: 9.0 sd: 4.094572295006621
Brewer: 26
mean: 12.615384615384615 median: 13.0 sd: 1.4163040491939975
Brewer: 27
mean: 11.550617283950617 median: 12.0 sd: 2.1649351175970537
Brewer: 28
mean: 13.2 median: 13.0 sd: 0.7483314773547882
Brewer: 29
mean: 11.0 median: 11.0 sd: 0.0
Brewer: 30
mean: 9.333333333333334 median: 9.0 sd: 3.822875761121112
Brewer: 31
mean: 11.4 median: 12.0 sd: 2.5508168626278653
Brewer: 32
mean: 11.958041958041958 median: 12.0 sd: 2.31103804931299
Brewer: 33
mean: 14.018181818181818 median: 14.0 sd: 2.3470748455425348
Brewer: 34
mean: 11.666666666666666 median: 12.0 sd: 1.6996731711975948
Brewer: 35
mean: 12.909090909090908 median: 14.0 sd: 1.928473039599675
Brewer: 36
mean: 12.565014031805426 median: 13.0 sd: 2.5881523842441694
Brewer: 37
mean: 14.78 median: 15.0 sd: 1.446236495183274
Brewer: 38
mean: 12.901639344262295 median: 13.0 sd: 2.0098521112936893
Brewer: 39
mean: 14.913769123783032 median: 15.0 sd: 2.2997410768192283
Brewer: 40
mean: 14.853696456334102 median: 15.0 sd: 2.4378154506466116
Brewer: 41
```

```
mean: 12.166666666666666 median: 13.0 sd: 2.3746344747958346
Brewer: 42
mean: 13.529411764705882 median: 14.0 sd: 2.0802803466470343
Brewer: 43
mean: 14.0 median: 14.0 sd: 0.816496580927726
Brewer: 44
mean: 12.371308016877638 median: 13.0 sd: 2.6268051012868656
Brewer: 45
mean: 12.04109589041096 median: 13.0 sd: 2.951379320743882
Brewer: 46
mean: 11.9 median: 12.5 sd: 1.8411952639521967
Brewer: 47
mean: 11.1875 median: 12.0 sd: 2.9201615965559165
Brewer: 48
mean: 12.707207207207206 median: 13.0 sd: 2.268039265581248
Brewer: 49
mean: 13.459694989106755 median: 13.0 sd: 2.34032768560976
Brewer: 50
mean: 17.333333333333332 median: 17.0 sd: 1.247219128924647
Brewer: 51
mean: 11.5 median: 11.5 sd: 3.5
Brewer: 52
mean: 14.294621026894866 median: 15.0 sd: 2.2118806600771648
Brewer: 53
mean: 12.545454545454545 median: 13.0 sd: 1.6713433009863852
Brewer: 54
mean: 12.5625 median: 13.0 sd: 0.7043392293490403
Brewer: 55
mean: 12.952380952380953 median: 13.0 sd: 1.8381199110113127
Brewer: 56
mean: 12.976027397260275 median: 13.0 sd: 2.562126813307368
Brewer: 57
mean: 13.0 median: 13.0 sd: 1.632993161855452
Brewer: 58
mean: 12.76923076923077 median: 14.0 sd: 2.832608098387995
Brewer: 59
mean: 12.211678832116789 median: 12.0 sd: 2.27146904757196
Brewer: 60
mean: 11.88888888888889 median: 12.0 sd: 2.469567863432541
Brewer: 61
mean: 11.112903225806452 median: 11.5 sd: 2.103030368912974
Brewer: 62
mean: 11.76923076923077 median: 11.0 sd: 1.7166087388016462
Brewer: 63
mean: 13.256410256410257 median: 13.0 sd: 1.3722141702830413
Brewer: 64
mean: 11.325892857142858 median: 12.0 sd: 3.4841540666206163
Brewer: 65
mean: 20.0 median: 20.0 sd: 0.0
Brewer: 66
mean: 13.666666666666666 median: 14.0 sd: 1.247219128924647
Brewer: 67
mean: 13.024896265560166 median: 14.0 sd: 3.2907834469368167
Brewer: 68
mean: 13.5 median: 13.5 sd: 0.5
Brewer: 69
mean: 12.56 median: 13.0 sd: 2.786826151736057
```

```
Brewer: 70
mean: 15.386691484411354 median: 16.0 sd: 1.9307050162261605
Brewer: 71
mean: 11.0 median: 11.0 sd: 0.0
Brewer: 72
mean: 13.015094339622642 median: 13.0 sd: 2.293501584782954
Brewer: 73
mean: 12.6 median: 11.0 sd: 2.4166091947189146
Brewer: 74
mean: 11.10344827586207 median: 12.0 sd: 2.3758095184619217
Brewer: 75
mean: 12.924679487179487 median: 13.0 sd: 2.3789474401324124
Brewer: 76
mean: 12.5 median: 13.0 sd: 2.909676307807488
Brewer: 77
mean: 6.3253588516746415 median: 6.0 sd: 3.341686533827527
Brewer: 78
mean: 15.529032258064516 median: 16.0 sd: 2.147195252084058
Brewer: 79
mean: 10.0 median: 10.0 sd: 0.0
Brewer: 80
mean: 13.698863636363637 median: 14.0 sd: 2.341619210473706
Brewer: 81
mean: 13.875 median: 13.0 sd: 2.1469455046647083
Brewer: 82
mean: 12.8 median: 12.5 sd: 0.9797958971132712
Brewer: 83
mean: 14.114543114543114 median: 14.0 sd: 2.180056338876711
Brewer: 84
mean: 12.19047619047619 median: 14.0 sd: 3.7999880653828573
Brewer: 85
mean: 10.901960784313726 median: 11.0 sd: 1.6716189400742834
Brewer: 86
mean: 11.333333333333334 median: 12.0 sd: 3.2180049029725786
Brewer: 87
mean: 8.431034482758621 median: 9.0 sd: 2.7044794831303505
Brewer: 88
mean: 13.872865275142315 median: 14.0 sd: 2.3093251983625622
Brewer: 89
mean: 14.966666666666667 median: 15.0 sd: 1.32874209519965
Brewer: 90
mean: 10.75 median: 11.5 sd: 1.6393596310755
Brewer: 91
mean: 13.008849557522124 median: 13.0 sd: 2.462086226757002
Brewer: 92
mean: 12.485714285714286 median: 13.0 sd: 1.8878775470493474
Brewer: 93
mean: 14.0 median: 14.5 sd: 1.5491933384829668
Brewer: 94
mean: 10.966666666666667 median: 11.0 sd: 2.726210230745645
Brewer: 95
mean: 13.466908861159375 median: 14.0 sd: 2.4414100283035616
Brewer: 96
mean: 15.0 median: 15.0 sd: 2.0
Brewer: 97
mean: 11.571428571428571 median: 13.0 sd: 4.135461378894322
Brewer: 98
```

```
mean: 13.041666666666666 median: 13.0 sd: 1.135751097536584
Brewer: 99
mean: 7.5 median: 7.0 sd: 1.5
```

It seems like people have different tastes, but there are definitely beers and brewer that are generally more liked. This is apparrent because the standard deviations are not too high, but the beers and brewers with a significant amount of reviews do have a decent standard deviation. which shows that there is definitely some differentiation in taste. There is also an extreme range of means, which shows that there is some form of objective ranking.

Part 2: Sentiment analysis.

In [79]:
```
with open('/project/cmsc25025/beer_review/vocab_30.json', 'r') as f:
        vocab = json.load(f)
vocab_set = set(vocab)
```

In [80]:
```
ratings = [1 if b['overall'] >=14 else 0 for b in brv]

demo_brv = brv[:10000]
demo_ratings = ratings[:10000]
```

(a) Generating features.

You need to represent text reviews in terms of a vector of features (covariates). One simple but effective representation is to use membership in a fixed vocab- ulary. Suppose the vocabulary contains p words. For a given review, you normalize the text, and separate it into space-delimited tokens. For each of the tokens, if it is in the dictionary you have a one for the corresponding word in the feature vector, and you ignore it otherwise.

In [110]:
```
def makeCSRandLabels(words, ratings):
    vocab_words = [list(vocab_set & set(re.sub("[^\w]", " ",  br['revie
w'].lower()).split())) for br in words]
    vocab_words_ratings = [(x, ratings[i]) for i,x in enumerate(vocab_wo
rds) if x != []]
    vocab_words = [x for x,i in vocab_words_ratings]
    csr_ratings = [i for x,i in vocab_words_ratings]
    indptr = [0]
    indices = []
    data = []
    vocabulary = {}
    for v in vocab:
        index = vocabulary.setdefault(v, len(vocabulary))
    for d in vocab_words:
        for term in d:
            index = vocabulary.setdefault(term, len(vocabulary))
            indices.append(index)
            data.append(1)
        indptr.append(len(indices))
    return csr_matrix((data, indices, indptr)), csr_ratings
```

In [111]:
```
csr_vocab, csr_labels = makeCSRandLabels(demo_brv, demo_ratings)
```

(b) Logistic regression using Newton's method

Logistic regression using Newton's method. Train an l2-regularized logistic regression classifier using the sklearn.linear model.LogisticRegression class. To select the regularization parameter C = 1/λ, you should try different values on the validation set. Pick the best. How long does it take to train?

In [112]:
```
def getDualDataRange(data, ratings,l,r):
    return csr_matrix(data[int(l*len(data)):int(r*len(data))]), ratings[
int(l*len(ratings)):int(r*len(ratings))]

train_csr_words, train_ratings = getDualDataRange(csr_vocab.toarray(), c
sr_labels, 0, 0.7)
valid_csr_words, valid_ratings = getDualDataRange(csr_vocab.toarray(), c
sr_labels, 0.7, 0.85)
test_csr_words, test_ratings = getDualDataRange(csr_vocab.toarray(), csr
_labels, 0.85, 1)
```

In [109]:
```
# Testing for best lambda

ltrain_csr_words, ltrain_ratings = getDualDataRange(csr_vocab.toarray(),
csr_labels, 0, 0.07)
lvalid_csr_words, lvalid_ratings = getDualDataRange(csr_vocab.toarray(),
csr_labels, 0.07, 0.085)

ls = [5,10,20,30,40,50,75,100,200,300,400,500]
errors = []
for l in ls:
    lg=LogisticRegression(fit_intercept=True, C=l, penalty='l2',
                multi_class='multinomial',solver='newton-cg')
    model = lg.fit(ltrain_csr_words, ltrain_ratings)
    predicted_valid_ratings = lg.predict(lvalid_csr_words)
    error = np.mean(lvalid_ratings != predicted_valid_ratings)
    errors.append(error)
```

In [85]:
```
for i in range(len(ls)):
    print("The error rate for 1/lambda = " + str(ls[i]) + " is " + str(e
rrors[i]))
error_pairs = [(ls[i], errors[i]) for i in range(len(ls))]
min_pair = min(error_pairs, key = lambda x: x[1])
print("The minimum 1/lambda value is: " + str(min_pair[0]))
l = min_pair[0]
```

```
The error rate for 1/lambda = 5 is 0.36
The error rate for 1/lambda = 10 is 0.36
The error rate for 1/lambda = 20 is 0.35333333333333333
The error rate for 1/lambda = 30 is 0.36
The error rate for 1/lambda = 40 is 0.36
The error rate for 1/lambda = 50 is 0.36
The error rate for 1/lambda = 75 is 0.36666666666666664
The error rate for 1/lambda = 100 is 0.36666666666666664
The error rate for 1/lambda = 200 is 0.36
The error rate for 1/lambda = 300 is 0.36666666666666664
The error rate for 1/lambda = 400 is 0.36666666666666664
The error rate for 1/lambda = 500 is 0.36666666666666664
The minimum 1/lambda value is: 20
```

In [86]:
```
#Fitting the Regression
start_time = time.time()
lg=LogisticRegression(fit_intercept=True, C=l, penalty='l2',
                multi_class='multinomial',solver='newton-cg')
model = lg.fit(train_csr_words, train_ratings)
end_time = time.time()
print("Traning the model took %s seconds." % (end_time - start_time))
```

```
Traning the model took 11.313294887542725 seconds.
```

In [87]:
```
# Testing the Training
predicted_test_ratings = lg.predict(test_csr_words)
error = np.mean(test_ratings != predicted_test_ratings)
print("The error rate is:", error)
```

```
The error rate is: 0.19786096256684493
```

Do the same thing using the LinearSVC class in sklearn.svm. Use loss='hinge'. Compare the results of the logistic loss to the hinge loss. Is there a difference?

In [88]:
```
ltrain_csr_words, ltrain_ratings = getDualDataRange(csr_vocab.toarray(),
csr_labels, 0, 0.07)
lvalid_csr_words, lvalid_ratings = getDualDataRange(csr_vocab.toarray(),
csr_labels, 0.07, 0.085)

ls = [5,10,20,30,40,50,75,100,200,300,400,500]
errors = []
for l in ls:
    hinge=LinearSVC(loss='hinge', penalty='l2',dual=True, tol=.001, C =
l, max_iter = 100000)
    model = hinge.fit(ltrain_csr_words, ltrain_ratings)
    predicted_valid_ratings = hinge.predict(lvalid_csr_words)
    error = np.mean(lvalid_ratings != predicted_valid_ratings)
    errors.append(error)
```

In [89]:
```
error_pairs = [(ls[i], errors[i]) for i in range(len(ls))]
min_pair = min(error_pairs, key = lambda x: x[1])
lhinge = min_pair[0]
```

In [113]:
```
#Fitting the Regression

start_time = time.time()
hinge=LinearSVC(loss='hinge', penalty='l2',dual=True, tol=.001, C = l, m
ax_iter = 100000)
model = hinge.fit(train_csr_words, train_ratings)
end_time = time.time()
print("Traning the model took %s seconds." % (end_time - start_time))
```

```
Traning the model took 8.50538682937622 seconds.
```

In [91]:
```
# Testing the Training
predicted_test_ratings = hinge.predict(test_csr_words)
error = np.mean(test_ratings != predicted_test_ratings)
print("The error rate is:", error)
```

```
The error rate is: 0.2520053475935829
```

The hinge model was extremely faster, but does get a slightly higher error rate. This is important to realize as using hinge is probably better when in an instance where you are willing to sacrifice accuracy for a faster runtime. For instance, if your dataset is extremely large and you are only testing whether the model is good and thus can sacrifice some accuracay.

(c) Stochastic gradient descent

Your next job is to train an l2-regularized logistic regression classifier using stochastic gradient descent. Recall the SGD framework that was covered in class using minibatches.

i. Initialize the model with θ = 0 (uniform).

ii. Randomly split the training data into mini-batches. Make one pass of the data, processing one mini-batch in every iteration. This is called one training epoch.

In [92]:
```
def getYhat(x, theta):
    return 1/(1+exp(-theta.dot(x)))

def getNewThetas(x, y, yhat, thetas, alpha, lamb):
    return thetas + alpha*((y - yhat) * yhat *(1 - yhat) * x - 2*lamb*th
etas)

def getBatch(b, x, y):
    indices = random.sample(range(len(y)), b)
    xb = [x.getrow(i).toarray()[0] for i in indices]
    yb = [y[i] for i in indices]
    return xb,yb

def SGDstep(x,y, theta, alpha, lamb, b):
    xb, yb = getBatch(b, x, y)
    for i in range(b):
        yhat = getYhat(xb[i],theta)
        theta = getNewThetas(xb[i], yb[i], yhat, theta, alpha, lamb)
    return theta

def predictSGD(X,theta):
    yhats = []
    for i in range(X.shape[0]):
        x = X.getrow(i).toarray()[0]
        yhats.append(getYhat(x, theta))
    return yhats


def runSGDsteps(x, y, theta, alpha, lamb, b, n):
    thetas = []
    for i in range(n):
        theta = SGDstep(x, y, theta, alpha, lamb, b)
        thetas.append(theta)

    return theta, thetas
```

In [93]:
```
theta = SGDstep(train_csr_words, train_ratings, np.zeros(len(vocab)), 0.
001, 1/l, 10)
yhats = predictSGD(test_csr_words, theta)
predicted_test_ratings = [1 if yhat >= 0.5 else 0 for yhat in yhats]
wrong = 0
for i in range(len(test_ratings)):
    if test_ratings[i] != predicted_test_ratings[i]:
        wrong += 1
error = wrong/len(test_ratings)
print("The error rate is:", error)
```

```
The error rate is: 0.15641711229946523
```

iii. Repeat the last step a few times.

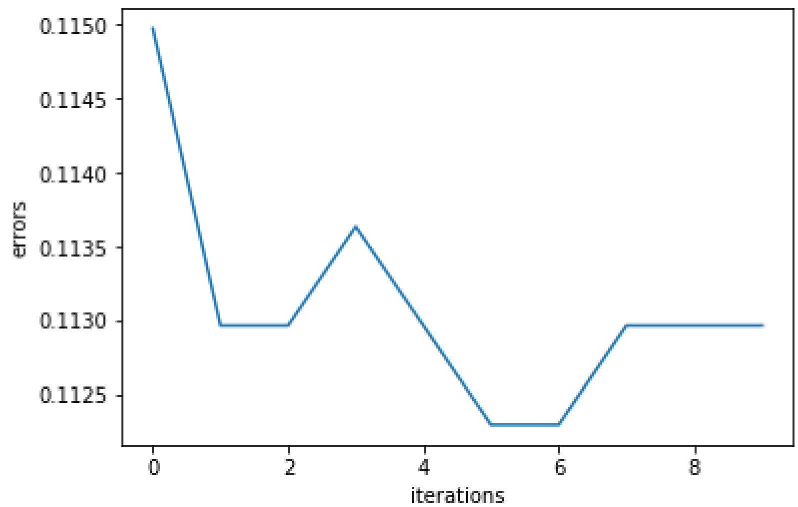
In [94]:
```
start_time = time.time()
theta, thetas = runSGDsteps(train_csr_words, train_ratings, np.zeros(len
(vocab)), 0.001, 1/l, 100, 10)
end_time = time.time()
print("Traning the model took %s seconds." % (end_time - start_time))
```

```
Traning the model took 2.682340621948242 seconds.
```

In [95]:
```
def log_likelihood(x, y, theta):
    scores = np.dot(x, theta)
    ll = np.sum( y*scores - np.log(1 + np.exp(y)) )
    return ll

errors = []
ll = []
for theta in thetas:
    yhats = predictSGD(test_csr_words, theta)
    predicted_test_ratings = [1 if yhat >= 0.5 else 0 for yhat in yhats]
    wrong = 0
    for i in range(len(test_ratings)):
        if test_ratings[i] != predicted_test_ratings[i]:
            wrong += 1
    errors.append(wrong/len(test_ratings))
    #Note that computing the log likelihood kills the kernal and uses a
 lot of memory, so I commented it out. If you want to try to compute it,
uncomment this.
    # ll.append(log_likelihood(test_csr_words,test_ratings,theta))
```

In [96]:
```
plt.plot(errors)
plt.ylabel('errors')
plt.xlabel('iterations')
plt.show()
if (len(ll)>0):
    plt.plot(ll)
    plt.ylabel('log-likelihood')
    plt.xlabel('iterations')
    plt.show()
```



In [97]:
```
error = min(errors)
print("The best error rate is:", error)
```

```
The best error rate is: 0.11229946524064172
```

As you can see, the error rate generally shrinks at first, but once it reaches a minimum, it begins to increase. This the behavior that we expect with SGD. As it is stochastic, we cannot expect the same behavior for every run and in some cases, the error rate may even increase.It appears that the best results happen around 3-5 iteraitons. Overall, the change in error rate is extremely small after the first iteration and thus if we run too many or too little iterations, it is not too big of a deal.

The error rate for SGD was significantly better than the error rate for the logistic and hinge regression. It runs slower than the hinge regression, but still runs fairly fast. This is definitely the best prediction function as it runs fast enough and has the best error rate.

Part 3: Scores versus text.

In addition to text reviews, the users also scored appearance, aroma, palate, style, taste of a beer. In this problem, you will check whether those scores could reflect people's opinion better than text. Train another logistic regression model using those features. You should use the same SGD algorithm as before. Compare the model using score features with that using review text. Again, use the validation set to tune the regularization parameters, and retrain the model on the union of training and validation set. Finally, compute the prediction error on the testing set

In [98]:
```
scores = [[b["appearance"], b["aroma"], b["palate"], b["style"], b["tast
e"]] for b in brv]
```

In [99]:
```
demo_scores = scores[:10000]
demo_ratings = ratings[:10000]

def getDataRange(data, l, r):
    return data[int(l*len(data)):int(r*len(data))]


train_scores = getDataRange(demo_scores, 0, 0.7)
valid_scores = getDataRange(demo_scores, 0.7, 0.85)
test_scores = getDataRange(demo_scores, 0.85, 1)

train_ratings = getDataRange(demo_ratings, 0, 0.7)
valid_ratings = getDataRange(demo_ratings, 0.7, 0.85)
test_ratings = getDataRange(demo_ratings, 0.85, 1)
```

In [100]:
```
# Testing for best lambda

ltrain_scores = getDataRange(demo_scores, 0, 0.7)
lvalid_scores = getDataRange(demo_scores, 0.7, 0.85)
ltrain_ratings = getDataRange(demo_ratings, 0, 0.7)
lvalid_ratings = getDataRange(demo_ratings, 0.7, 0.85)

ls = [5, 10, 20, 30, 40, 50, 75, 100, 200, 300, 400,500, 1000, 2000, 500
0, 10000]
errors = []
for l in ls:
    lg=LogisticRegression(fit_intercept=True, C=l, penalty='l2',
                multi_class='multinomial',solver='newton-cg')
    model = lg.fit(ltrain_scores, ltrain_ratings)
    predicted_valid_ratings = lg.predict(lvalid_scores)
    error = np.mean(lvalid_ratings != predicted_valid_ratings)
    errors.append(error)
```

In [101]:
```
error_pairs = [(ls[i], errors[i]) for i in range(len(ls))]
min_pair = min(error_pairs, key = lambda x: x[1])
l = min_pair[0]
```

In [102]:
```
#Fitting the Regression
start_time = time.time()
lg=LogisticRegression(fit_intercept=True, C=1, penalty='l2',
                multi_class='multinomial',solver='newton-cg')
model = lg.fit(train_scores, train_ratings)
end_time = time.time()
print("Traning the model took %s seconds." % (end_time - start_time))
```

```
Traning the model took 0.2636234760284424 seconds.
```

In [103]:
```
# Testing the Training
predicted_test_ratings = lg.predict(test_scores)
error = np.mean(test_ratings != predicted_test_ratings)
print("The error rate is:", error)
```

```
The error rate is: 0.09
```

In [104]:
```
#Running the SGD
def getScoreBatch(b, x, y):
    indices = random.sample(range(len(y)), b)
    xb = [x[i] for i in indices]
    yb = [y[i] for i in indices]
    return np.array(xb),np.array(yb)

def SGDScoreStep(x,y, theta, alpha, lamb, b):
    xb, yb = getScoreBatch(b, x, y)
    for i in range(b):
        yhat = getYhat(xb[i],theta)
        theta = getNewThetas(xb[i], yb[i], yhat, theta, alpha, lamb)
    return theta

def runSGDScoreSteps(x, y, theta, alpha, lamb, b, n):
    for i in range(n):
        theta = SGDScoreStep(x, y, theta, alpha, lamb, b)
    return theta

start_time = time.time()
theta = runSGDScoreSteps(train_scores, train_ratings, np.zeros(len(test_
scores[0])), 0.001, 1/1, 100, 5)
end_time = time.time()
print("Traning the model took %s seconds." % (end_time - start_time))
```

```
Traning the model took 0.008984565734863281 seconds.
```

In [105]:
```
def predictScoresSGD(X,theta):
    yhats = []
    for i in range(len(X)):
        x = X[i]
        yhats.append(getYhat(x, theta))
    return yhats

yhats = predictScoresSGD(test_scores, theta)

predicted_test_ratings = [1 if yhat >= 0.5 else 0 for yhat in yhats]
```

In [106]:
```
wrong = 0
for i in range(len(test_ratings)):
    if test_ratings[i] != predicted_test_ratings[i]:
        wrong += 1
error = wrong/len(test_ratings)
print("The error rate is:", error)
```

```
The error rate is: 0.11133333333333334
```

Which model predicts better? Is the representation you constructed for text more powerful, or are the scores? Why? Comment on your findings and discuss your thinking.

The scores model predicts way better than the test model in every case, but SGD. This makes sense as sentiment analysis is an extremely conplex field and thus our model does not do a fantastic job of performing it. In addition, there is a much more objective relationship between the scores and the ratings. The text in the review is informative, but it is far less deterministic of the actual rating.

In the case of SGD, they interestingly get about the same error rate. This is a significant note as it shows that the power of the scores and text are about the same in determining the actual rating. I was really suprised to see this as I imagined that it would be better for the scores. However, with the scores, there are far more dimensions and thus it makes sense that SGD would perform well on the text.