Beer Reviews

October 3, 2014

1 Introduction

In this notebook, I'll demonstrate some Data Science applications on a fun dataset. The dataset consists of a series of beer ratings sampled from the Beer Advocate database. (Note that this data *used to be* available here, but has recently been removed, by request of Beer Advocate). The dataset has approx. 1.5 million reviews of tens of thousands of beers from tens of thousands of users. Each beer review is from a single user and contains ratings of the beer's palate, taste, ABV, aroma, appearance, and overall rating (all from 0-5).

On this dataset, I'll demonstrate two recommendation system examples, one using content-based filtering, and another using collaborative filtering. The third example will demonstrate building predictive models of overall beer ratings. The analysis (obviously, given that I'm using IPython notebook) will be in python. The main python modules that I will use are Pandas (for data munging), Numpy (for number crunching), Matplotlib (for visualization), scikits-learn (for modeling) and pyrsvd (for fancy matrix completion used in the collaborative filtering example).

First, some setup and loading the data...

```
In [1]: # PREAMBLE/SETUP
        import pandas as pd
        import numpy as np
        from matplotlib import pyplot as plt

        # DISPLAY FONTS
        font = {'family' : 'arial',
                'weight' : 'bold',
                'size'   : 16}
        matplotlib.rc('font', **font)

        # PATH TO DATA...
        df = pd.read_csv('/home/dustin/data/toy_datasets/beer_reviews/beer_reviews.csv')
```

2 Example I: Content-based Filtering

In most cases, content-based filtering compares products based on similarity metrics, suggesting products that are similar to one that was chosen by a user. Here, I'll demonstrate content-based filtering on the beer dataset.

2.1 Let's compare some beers

As mentioned above, the database contains around 1.5 million beer reviews for thousands of beers and reviewers. To keep the demonstration simple and properly scaled for a laptop, I'll choose a popular subset of all the available beers. (For larger scale problems, like analyzing the original or bigger databases, we would parallelize a lot of these calculations.)

We'll investigate the following 11 popular beers:

In [2]:
```
# DEFINE A SET OF BEERS TO COMPARE
# (FOR ALL BEERS IN DATASET, WE WOULD WANT TO PARALLELIZE)
beers = ["Dale's Pale Ale",
         "Sierra Nevada Pale Ale",
         "Michelob Ultra",
         "Two Hearted Ale",
         "Natural Light",
         "Bud Light",
         "Fat Tire Amber Ale",
         "Coors Light",
         "Blue Moon Belgian White",
         "90 Minute IPA",
         "Guinness Draught"]
```

In order to compare any two of these beers, we need an objective definition of similarity. One way of defining similarity is by calculating the (Euclidian) distance between features (e.g. ABV or palate ratings) of the two beers. To do so, I define a function **feature_distances** that calculates the distance between two beers' palate, aroma, taste, and ABV. For each feature, the distance is calculated across all reviewers that reviewed both beers. (The function also uses helper function for extracting beer reviews)

In [3]:
```
def get_beer_reviews(beer, reviewers):
    '''Extract review given a list of overlapping reviewers'''
    mask = (df.review_profilename.isin(reviewers)) & (df.beer_name == beer)
    reviews = df[mask].sort('review_profilename')
    reviews = reviews[reviews.review_profilename.duplicated() == False] # GET RID OF REPLICATED
    return reviews

def feature_distances(beer1,beer2):
    '''Calculate similarities between a set of features for beer1 and beer2'''
    from sklearn.metrics.pairwise import euclidean_distances

    # GET REVIEWERS FOR EACH BEER
    beer_1_reviewers = df[df.beer_name == beer1].review_profilename.unique()
    beer_2_reviewers = df[df.beer_name == beer2].review_profilename.unique()

    # TAKE THE INTERSECTION
    common_reviewers = set(beer_1_reviewers).intersection(beer_2_reviewers)

    # EXTRACT REVIEWS
    revs1 = get_beer_reviews(beer1, common_reviewers)
    revs2 = get_beer_reviews(beer2, common_reviewers)

    # CALCULATE THE FEATURE DISTANCES
    dist = []
    features = ['review_palate', 'review_aroma', 'review_taste','beer_abv']
    for f in features:
        dist.append(euclidean_distances(revs1[f],revs2[f])[0][0])

    return dist
```

Now that we have a way of calculating the distance between two beers, we can compare each of the common beers defined above...

In [4]:
```
# CALCULATE SIMILARITIES BETWEEN EACH OF THE COMMON BEERS
sims = []
```

```
        for ii, b1 in enumerate(beers):
            print 'Calculating similarities for beer (%d / %d): %s' % (ii+1, len(beers), b1)
            for b2 in beers:
                if b1 != b2:
                    row = [b1, b2] + feature_distances(b1,b2)
                    sims.append(row)
```

```
Calculating similarities for beer (1 / 11): Dale's Pale Ale
Calculating similarities for beer (2 / 11): Sierra Nevada Pale Ale
Calculating similarities for beer (3 / 11): Michelob Ultra
Calculating similarities for beer (4 / 11): Two Hearted Ale
Calculating similarities for beer (5 / 11): Natural Light
Calculating similarities for beer (6 / 11): Bud Light
Calculating similarities for beer (7 / 11): Fat Tire Amber Ale
Calculating similarities for beer (8 / 11): Coors Light
Calculating similarities for beer (9 / 11): Blue Moon Belgian White
Calculating similarities for beer (10 / 11): 90 Minute IPA
Calculating similarities for beer (11 / 11): Guinness Draught
```

Now, I create a new pandas dataframe to hold the similarities/distances.

```
In [5]: # CREATE NEW DATAFRAME FROM SIMILARITIES
        dists = pd.DataFrame(sims,columns=['Beer 1', 'Beer 2', 'Palate','Aroma','Taste','ABV'])
        dists.head(10)
```

Out[5]:

	Beer 1	Beer 2	Palate	Aroma	Taste	ABV
0	Dale's Pale Ale	Sierra Nevada Pale Ale	15.524175	15.564382	16.124515	23.658191
1	Dale's Pale Ale	Michelob Ultra	29.832868	30.565503	31.626729	30.685827
2	Dale's Pale Ale	Two Hearted Ale	17.248188	19.519221	18.621224	13.756816
3	Dale's Pale Ale	Natural Light	23.622024	25.787594	26.162951	25.611716
4	Dale's Pale Ale	Bud Light	38.108398	40.540104	41.542147	42.284986
5	Dale's Pale Ale	Fat Tire Amber Ale	16.598193	17.392527	17.663522	29.039800
6	Dale's Pale Ale	Coors Light	35.902646	38.301436	38.755645	40.495679
7	Dale's Pale Ale	Blue Moon Belgian White	18.980253	18.907670	20.868637	22.408704
8	Dale's Pale Ale	90 Minute IPA	19.855730	20.724382	21.230874	74.035464
9	Dale's Pale Ale	Guinness Draught	24.382371	24.844516	25.258662	51.120055

3 Find Closest Beer

Now let's say that I'm at a beer store that has all of these common beers, except 90 Minute IPA. BOO! I really like 90 Minute IPA! But since it isn't available, I'd like to find the next closest thing. The code below finds the overall closest beer using the mean distance of each of the beer features. The beer that is nearest to 90 Minute IPA will have the lowest average distance.

First we define a function to calculate the average distance between two beers based on their palate, aroma, taste, and abv.

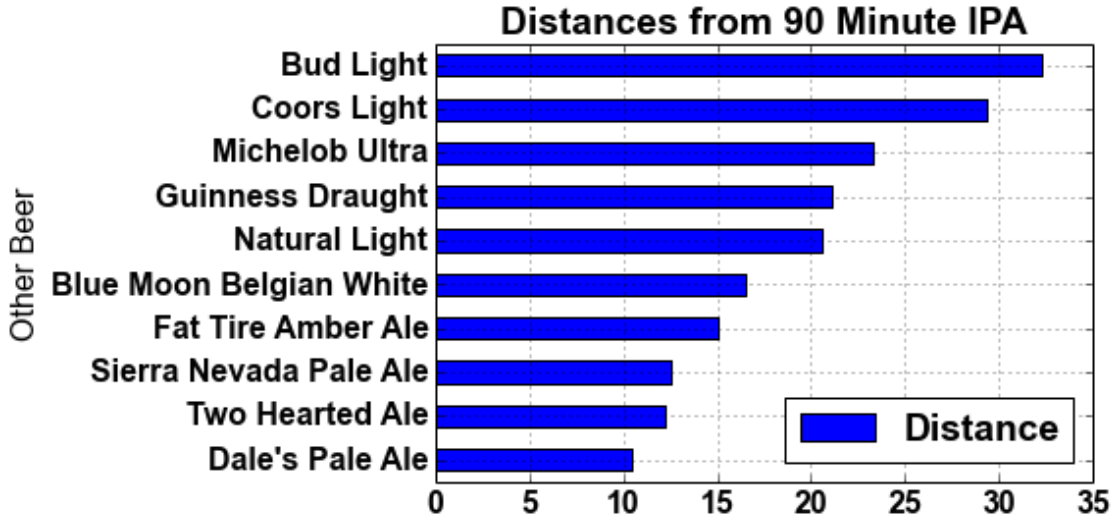
```
In [6]: # CALCULATE OVERALL DISTANCES BETWEEN BEERS BASED ON FEATURE SIMILARITIES
        wfeatures = ['Palate','Aroma','Taste','ABV']
        def mean_distance(dists, beer1, beer2, w=[1,1,1,1]):
            '''Calculate average (weighted) distance between two beers
            takes DataFrame <dists> and two beer names as required input'''
            mask = (dists['Beer 1'] == beer1) & (dists['Beer 2'] == beer2)
            row = dists[mask]
            return (row[wfeatures]*(pow(np.array(w),-1))).mean(1).tolist()[0]
```

Note that the function also allows us to weight each of the features, giving it more importance in the distance calculation. For example, perhaps, I really care about palate and ABV above all other things; I could capture this by giving ABV and palate larger weights.

Now we calculate the average distance between 90 Minute IPA and all the other beers in the store.

```
In [7]: beer_test = "90 Minute IPA" # (ONE OF MY FAVORITES!)
        weights = [2,1,1,2] # I PREFER PALATE AND ABV

        results = []
        for b in beers:
            if b != beer_test:
                results.append((beer_test, b, mean_distance(dists,beer_test,b, weights)))

        results = sorted(results, key=lambda x: x[2])

        # CREATE NEW DATAFRAME HOLDING RANKINGS
        rankings = pd.DataFrame(results,columns=['Test Beer','Other Beer','Distance'])
        srankings = rankings.sort('Distance')[['Other Beer','Distance']]

        # PLOT RESULTS
        srankings.set_index('Other Beer').plot(kind='barh');
        title('Distances from %s' % beer_test);
        print 'Closest beer to %s is %s' % (beer_test, rankings.ix[0]['Other Beer'])
```

```
Closest beer to 90 Minute IPA is Dale's Pale Ale
```



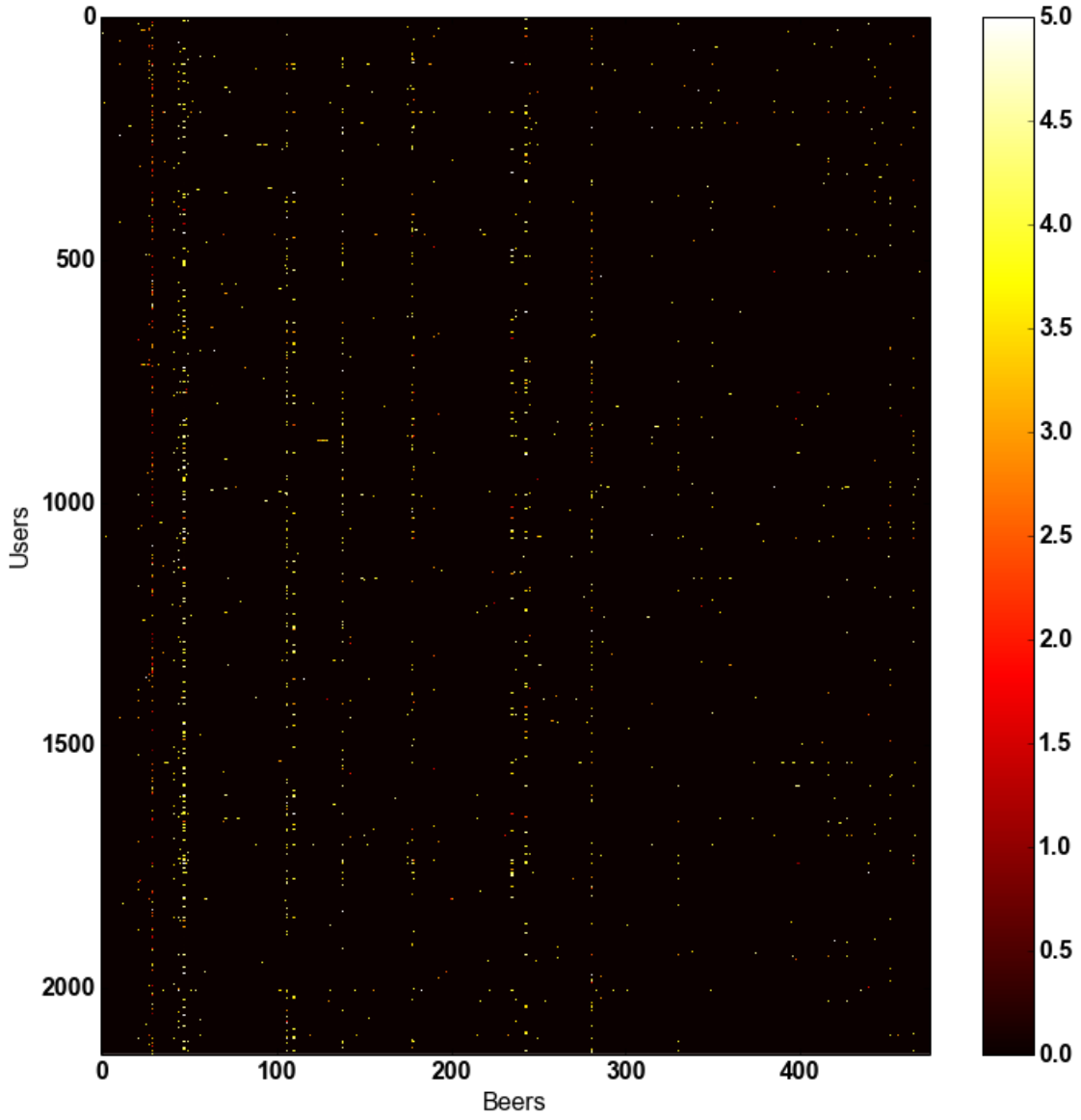
OK, the results suggest that Dale's Pale Ale is closest to 90 Minute IPA, followed closesly by Two Hearted Ale (another one of my favorites!). Not so close is Bud Light and Coors Light. I'd have to say that I agree with these suggested rankings!

4 Example II: Collaborative Filtering

In the content-based filtering example above we chose beers that had overlapping reviews from multiple reviewers. However, this generally isn't going to be the case. Many beers are rated by only a few reviewers, and it is generally unlikely that these reviewers will overlap. To better demonstrate this, let's take a look at the first 5000 reviews in the data set:

In [8]:
```
# DISPLAY A SUBSET OF THE RATINGS MATRIX
df2 = df[:5000].copy()

# GET RID OF REPLICATED REVIEWS FOR PIVOT
df2 = df2.drop_duplicates(['review_profilename','beer_name'])
df2 = df2.reindex().pivot('review_profilename','beer_name','review_overall')
df2 = df2.replace(NaN,0)
df2.head()
feature_matrix = df2.values
print feature_matrix.shape
# PLOT
plt.figure(figsize=(12,12));
plt.imshow(feature_matrix,interpolation='none');
plt.colorbar();
plt.set_cmap('hot');
plt.axis('tight');
plt.xlabel('Beers');
plt.ylabel('Users');
```

```
(2137, 475)
```



We see above that the ratings are very sparsely distributed, with lots of zero (black) values. These zero values indicate beers that a beer drinker may not have rated, but could potentially like (or dislike). What we would like is a way of "filling in" these zero values. A common way of performing this filling in is what is called low-rank matrix factorization (LRMF).

In LRFM we assume that the ratings matrix $R \in \mathbb{R}^{D \times B}$ (D is the number of users/drinkers B is the number of beers) can be approximated by the product of two component matrices $U \in \mathbb{R}^{D \times F}$ and $V \in \mathbb{R}^{B \times F}$, both of which have low rank F (i.e. both are tall):

$R = VU^T$

The component matrices can be calculated using an algorithm known as Regularized Singular Value Decomposition (RSVD). Because the ratings matrices for real-world problems are generally very large (millions of rows and columns) and sparse (only 1-2% of nonzero values), the RSVD algorithm often uses a sparse data representation.

To perform the remainder of the collaborative filtering analysis, I select the beers that have atleast 500 reviews (to scale down the problem size for a laptop), and transform the data into a sparse vector representation that can be used with the RSVD python module.

```
In [9]: # REMOVE BEERS WITH FEWER THAN 500 REVIEWS
        # (MAKES THE PROBLEM SMALLER ON MY LAPTOP)
        n_reviews_min = 500
        df = df.groupby('beer_name').filter(lambda x: x.count() >= n_reviews_min)

In [10]: # CREATE UNIQUE USER AND BEER MAPPING
         user_mapping = {j:i for i,j in enumerate(df.review_profilename.unique())}
         beerid_mapping = {j:i+1 for i,j in enumerate(df.beer_name.unique())}

         # CREATE SPARSE DATA REPRESENTATION
         # (beerID,userID,rating)
         records = []
         for ii in range(df.shape[0]):
             tmp = df[ii:ii+1]
             records.append((beerid_mapping[tmp.beer_name.values[0]],
                             user_mapping[tmp.review_profilename.values[0]],
                             tmp.review_overall.values[0]))

In [11]: from rsvd import rating_t, RSVD

         # CONVERT INTO PTHON RECARRAY FOR REGULARIZED SVD
         records = np.array(records, dtype=rating_t)
         n_reviewers = len(df.review_profilename.unique())
         n_beers = len(df.beer_name.unique())
         print '%d distinct reviewers' % n_reviewers
         print '%d distinct beers' % n_beers
         print '%d total reviews' % len(records)

25316 distinct reviewers
613 distinct beers
593932 total reviews
```

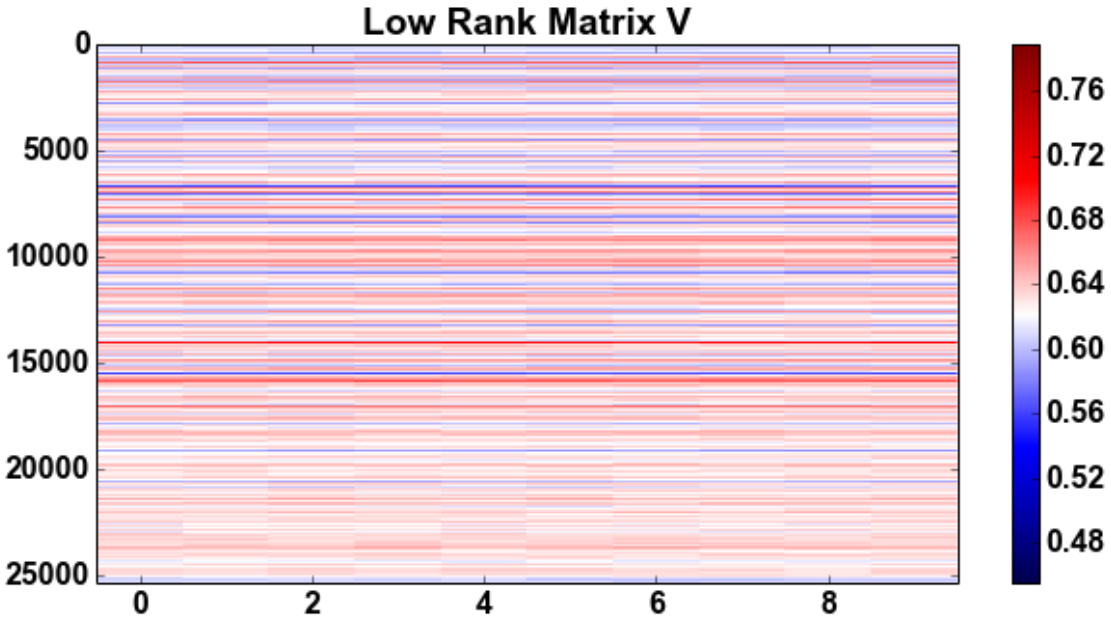
Thus, after removing beers with less than 500 reviews, we still have nearly 600k reviews from 25316 reviewers and 613 beers. Below, I run RSVD on the sparse representation of the ratings matrix, assuming a rank of $F = 10$

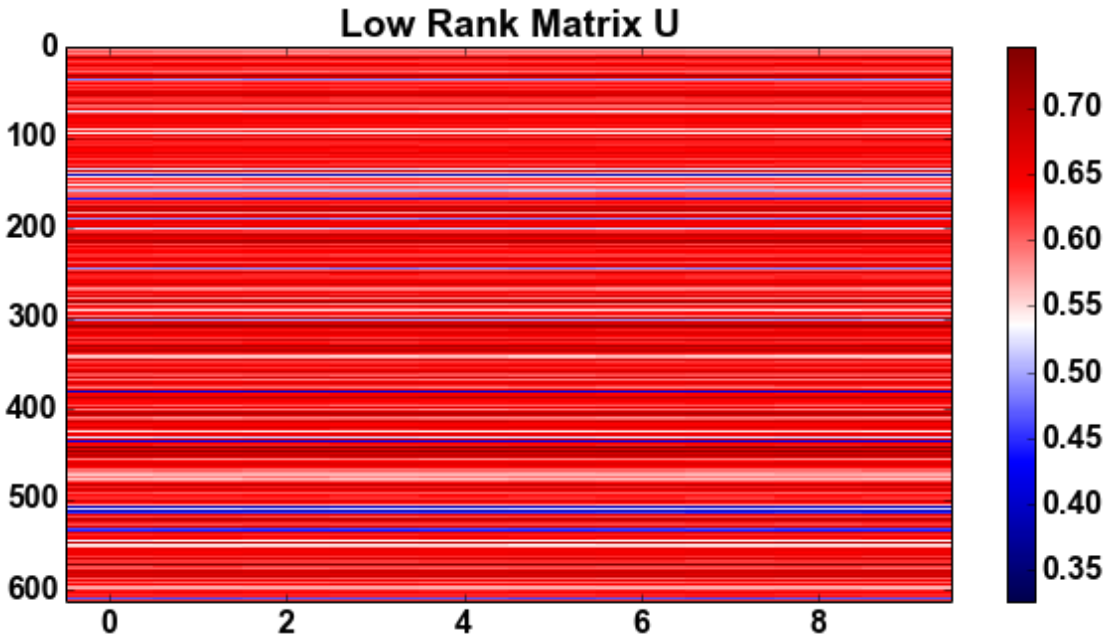
```
In [12]: F = 10 # ASSUMED RANK OF COMPONENT MATRICES

         # RUN LOW-RANK MATRIX FACTORIZATION
         model = RSVD.train(F, records, (n_beers,n_reviewers), maxEpochs=20)

########################################
            Factorizing
########################################
factors=10, epochs=20, lr=0.001000, reg=0.011000, n=593932
Init TRMSE: 0.683222
----------------------------------------
epoche        train err        probe err        elapsed time
0        0.606077        0.000000        0.031238
1        0.587010        0.000000        0.031169
2        0.584024        0.000000        0.030205
3        0.582278        0.000000        0.033183
4        0.581051        0.000000        0.030448
5        0.580104        0.000000        0.036744
6        0.579332        0.000000        0.027876
```

```
7         0.578677          0.000000          0.027437
8         0.578107          0.000000          0.027492
9         0.577601          0.000000          0.028200
10         0.577145          0.000000          0.032120
11         0.576730          0.000000          0.027384
12         0.576348          0.000000          0.028080
13         0.575995          0.000000          0.027306
14         0.575665          0.000000          0.027512
15         0.575357          0.000000          0.027448
16         0.575066          0.000000          0.027276
17         0.574792          0.000000          0.028764
18         0.574531          0.000000          0.029337
19         0.574284          0.000000          0.029109
```

Now, let's take a look at the resulting component matrices calculated using RSVD.

```
In [13]: ## DISPLAY LOW-RANK COMPONENT MATRICES
         # V
         plt.figure(figsize=(10,5))
         plt.imshow(model.v,interpolation='none');
         plt.set_cmap('seismic')
         plt.colorbar()
         plt.axis('tight');
         plt.title('Low Rank Matrix V');

         # U
         plt.figure(figsize=(10,5))
         plt.imshow(model.u,interpolation='none');
         plt.colorbar()
         plt.axis('tight');
         plt.title('Low Rank Matrix U');
```

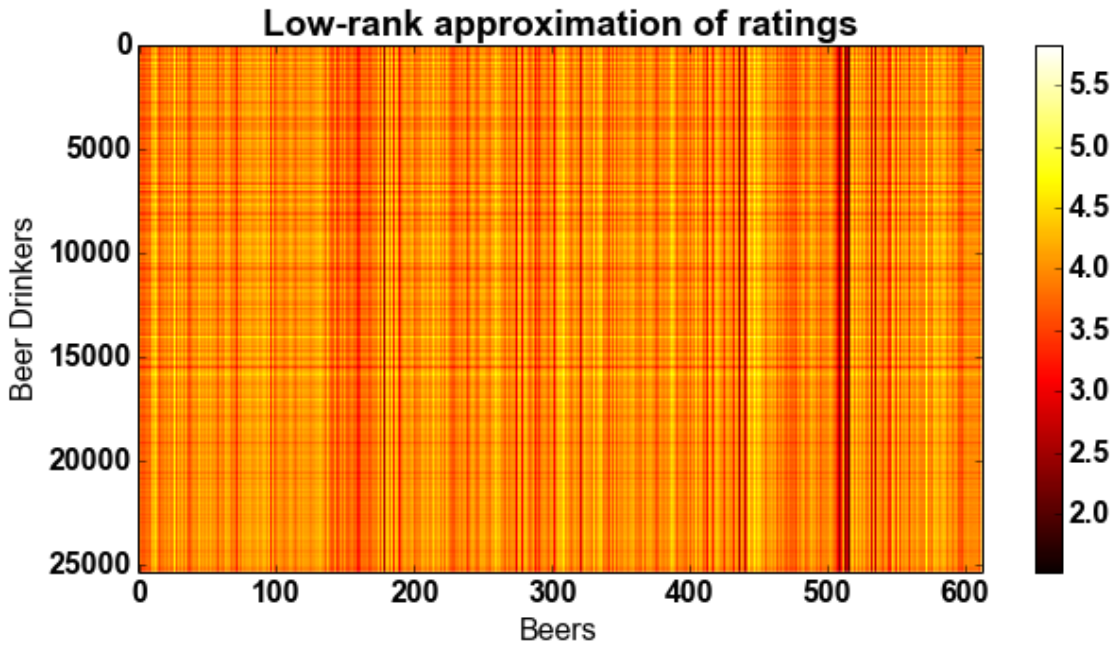




We see that each matrix has only $F = 10$ columns. Now, let's calculate the approximated ratings matrix as $R = VU^T$

and take a look at its structure.

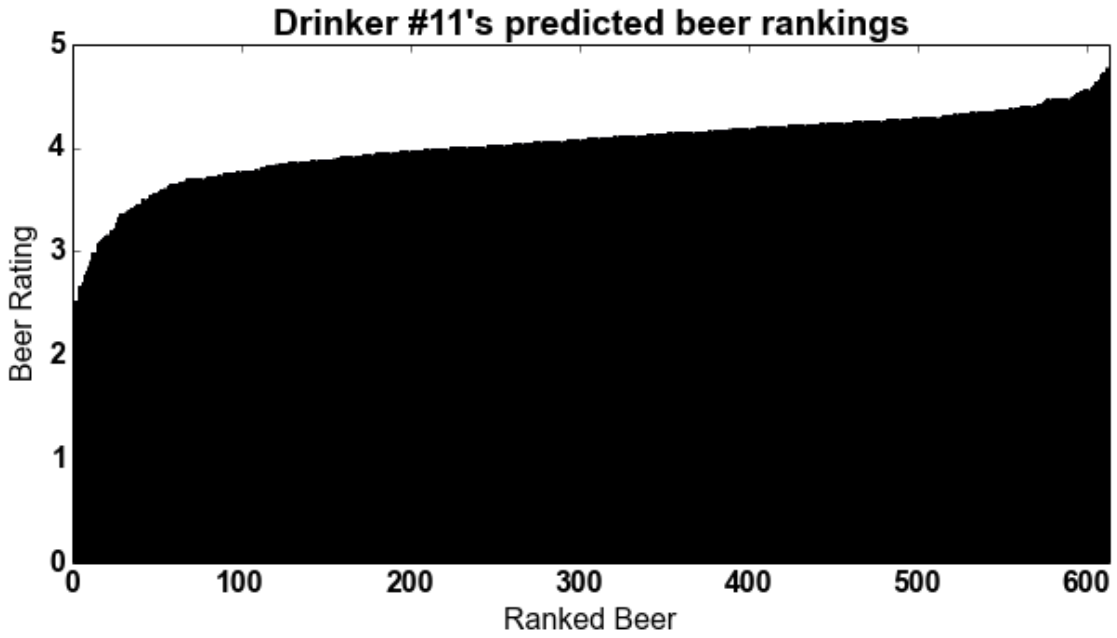
```
In [14]: # DISPLAY LOW-RANK APPROXIMATION OF RATINGS
         # £VU^T£
         all_rankings = model.v.dot(model.u.T)
         plt.figure(figsize=(10,5))
         plt.set_cmap('hot')
         plt.imshow(all_rankings,interpolation='none');
         plt.colorbar();
         plt.axis('tight');
         plt.ylabel('Beer Drinkers');
         plt.xlabel('Beers');
         plt.title('Low-rank approximation of ratings');
```



We see that most of the empty reviews have now been filled in. Now, given this filled-in matrix, we can look at the predicted ratings that a reviewer would give to all beers, despited the fact that only a few reviews may have been provided. Specifically, the predicted reviews for a beer reviewer will correspond to a row of VU^T

Below we'll choose a random reviewer (say, reviewer number 11) look at their predicted beer ratings, and suggest the top 10 and bottom 10 beers for that particular beer drinker.

```
In [15]: # CREATE REVERSED BEER-ID MAPPING (i.e. ID --> BEER NAME)
         beerid_mapping = {i:j for i,j in enumerate(df.beer_name.unique())}

In [16]: # PREDICT AN ARBITRARY USER'S PREFERENCES
         user_num = 10
         user_rankings = all_rankings[user_num,:]

         # SORT THE PREDICTED RATINGS
         sort_idx = sorted(range(len(user_rankings)),key=user_rankings.__getitem__)

         # DISPLAY
         plt.figure(figsize=(10,5))
         plt.bar(arange(len(sort_idx)),user_rankings[sort_idx]);
         plt.title("Drinker #%d's predicted beer rankings" % (user_num + 1))
         plt.xlim([0,n_beers]);
         plt.xlabel('Ranked Beer');
         plt.ylabel('Beer Rating');
```




```
In [25]: # GET TOP AND BOTTOM RATED BEERS
         n_suggest = 10
         ranked_beers = [beerid_mapping[idx] for idx in sort_idx]
         print "Drinker # %d's likely %d most preferred beers" % (user_num + 1, n_suggest)
         print ranked_beers[:-n_suggest:-1]
         print ''
         print "Drinker # %d's likely %d least preferred beers" % (user_num + 1, n_suggest)
         print ranked_beers[:n_suggest]
```

```
Drinker # 11's likely 10 most preferred beers
['Trappist Westvleteren 12', 'Pliny The Elder', 'Pliny The Younger', 'Founders CBS Imperial Stout', 'We

Drinker # 11's likely 10 least preferred beers
['Michelob Ultra', 'Samuel Adams Triple Bock', 'Natural Light', 'Bud Light', 'Corona Extra', 'Coors Lig
```

The model suggests that this beer drinker really likes IPAs (and good ones too from Russian River, like the Pliny's) and some stouts. The reviewer however likely dislikes lighter, more large scale production beers such as Bud/Michelobe/Coors Light. Sounds like my kind of beer drinker!

5 Example III: Regression & Beer Rating Prediction

Another useful task in Data Science is being able to predict the outome of some target variable (e.g. product rating) contingent on some product features. Here, I'll devise a simple regression model to predict overall beer rating from the features aroma, appearance, palate, taste, and ABV. I'll also assess which features are most important for predicting overall rating.

First let's extract our features and perform some transformation that will be useful for modeling. First we define a new data frame filled with the model features, and use the dataframe to imput (fill in) NaN values with the median of each respective feature.

```
In [18]: # LOAD DATA
         df = pd.read_csv('/home/dustin/data/toy_datasets/beer_reviews/beer_reviews.csv')
```

```
# EXTRACT INDEPENDENT VARIABLES/FEATURES
features = ['review_aroma','review_appearance','review_palate','review_taste','beer_abv']
dfmodel = df[features];
dfmodel = dfmodel.fillna(dfmodel.median(0)) # IMPUTE MISSING VALUES WITH COLUMN MEDIAN
dfmodel.describe()
```

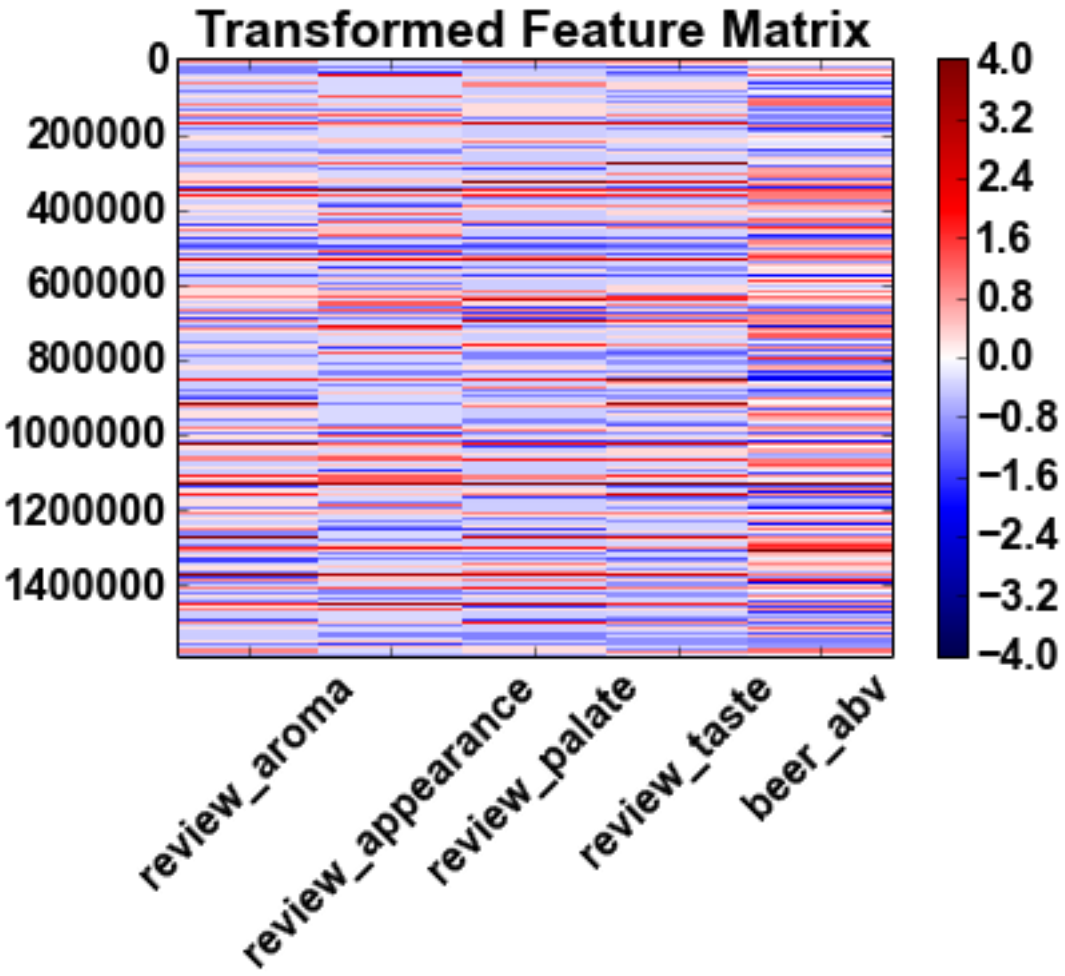
Out[18]:

	review_aroma	review_appearance	review_palate	review_taste	beer_abv
count	1586614.000000	1586614.000000	1586614.000000	1586614.00000	1586614.000000
mean	3.735636	3.841642	3.743701	3.79286	7.019214
std	0.697617	0.616093	0.682218	0.73197	2.275018
min	1.000000	0.000000	1.000000	1.00000	0.010000
25%	3.500000	3.500000	3.500000	3.50000	5.300000
50%	4.000000	4.000000	4.000000	4.00000	6.500000
75%	4.000000	4.000000	4.000000	4.50000	8.400000
max	5.000000	5.000000	5.000000	5.00000	57.700000

Now we Gaussianize theses strictly-positive features by using a compressive nonlinearity (i.e. negative logarithm). Furthermore, we standardize the model features so that each has zero mean and unit variance. (Note we keep around the feature means and standard deviations around, as I will use them to reverse the feature transformation durin model interpretation).

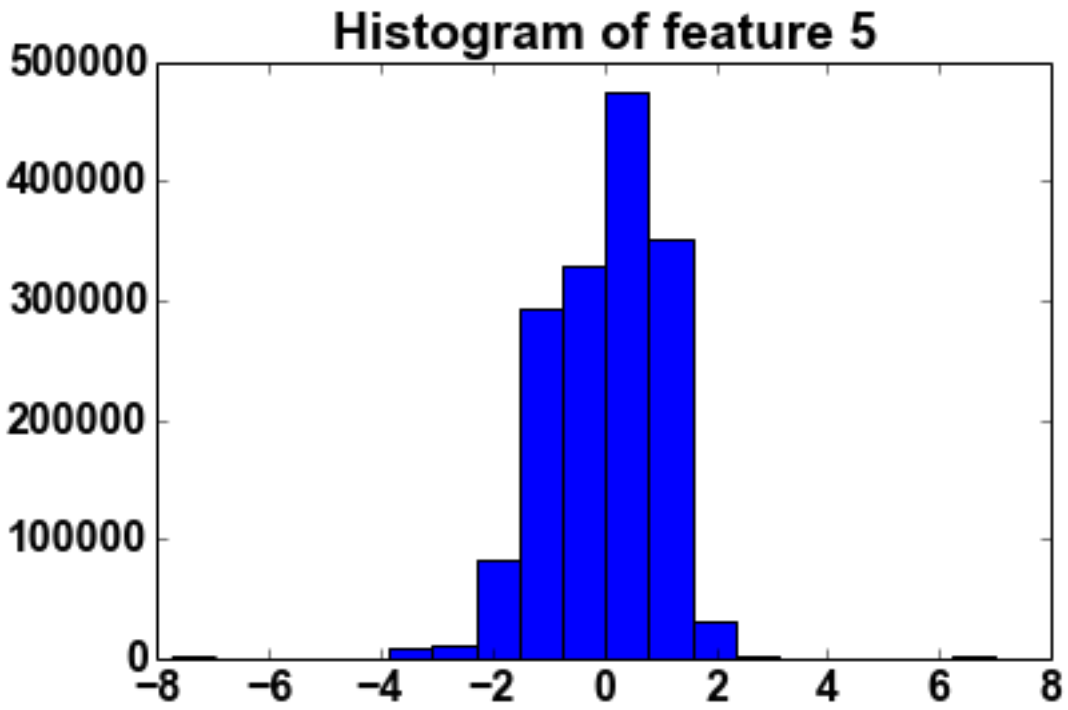
Below we calculate the tranformed features and visualize them.

In [19]:
```
# MODEL FEATURES
model_features = dfmodel.values;

# GAUSSIANIZE
model_features = -np.log(1 + model_features);

# KEEP MEANS AND STDS AROUND FOR INTERPRETATION LATERE
feature_means = model_features.mean(0)
feature_std = model_features.std(0)

# STANDARDIZE VARIABLES
model_features = (model_features - feature_means ) / feature_std

# DISPLAY THE FEATURE MATRIX
plt.imshow(model_features,interpolation='none');
plt.set_cmap('seismic')
plt.clim([-4, 4])
plt.axis('tight');
plt.colorbar();
plt.title('Transformed Feature Matrix');
plt.xticks(arange(len(features)),features, rotation=45);
```




```
In [20]: # SANITY CHECK TO MAKE SURE FEATURES ARE CENTERD AND SCALED
         idx = 4
         plt.hist(model_features[:,idx],bins=20);
         title('Histogram of feature %d' % (idx + 1))
         print 'Features Means: ' + str(model_features.mean(0))
         print 'Features Variances: ' + str(model_features.var(0))
```

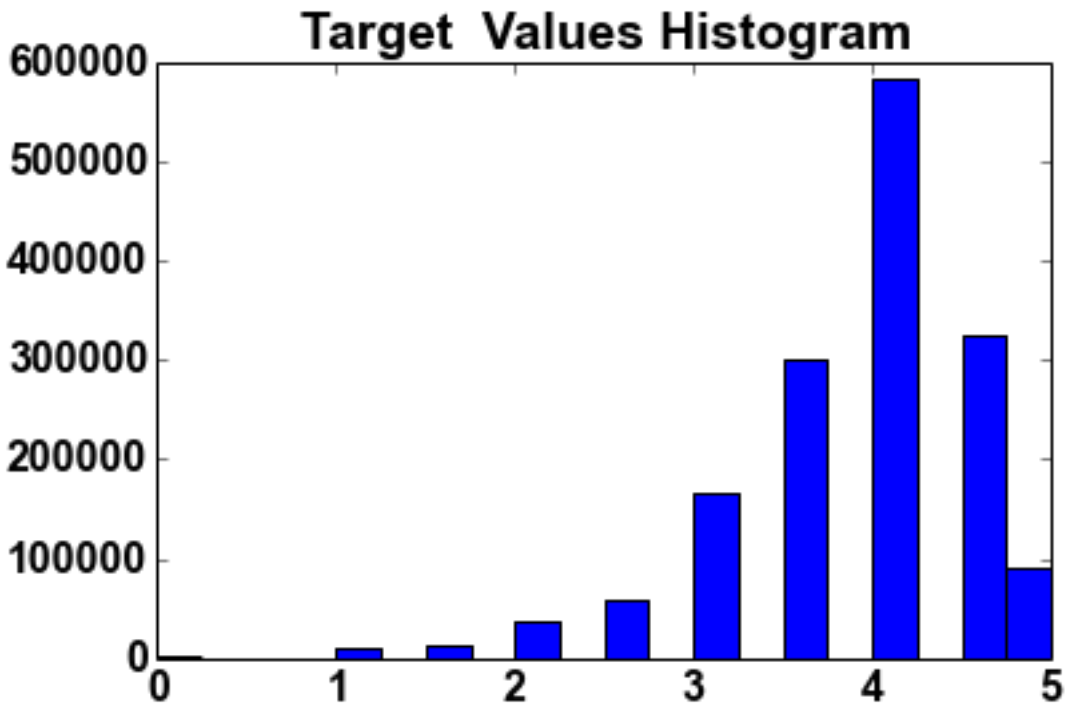
```
Features Means: [  5.09083994e-11   3.27769103e-11   4.92104533e-11   3.24433521e-11
  -9.66557237e-12]
Features Variances: [ 1.  1.  1.  1.  1.]
```



Ok, the features looks good; they all have zero mean and unit variance. Now let's extract the target values. (Once and perform similar preprocessing sucha as centering and scaling, but I found that it makes little difference in model accuracy).

```
In [21]: # EXTRACT TARGET VARIABLES
         dftargets = df['review_overall']
         dftargets.fillna(dftargets.median(0)) # IMPUTE WITH MEDIAN
         targets = dftargets.values;

         # STANDARDIZE TARGET VARIABLES
         #z_score = lambda x: (x - x.mean(0)) / x.std(0);
         #targets = z_score(targets)
         plt.hist(targets,bins=20);
         plt.title('Target  Values Histogram')

         print 'Target Mean: ' + str(targets.mean())
         print 'Target Variance: ' + str(targets.var())
```

```
Target Mean: 3.81558085331
Target Variance: 0.519295549419
```



Alright! Now that we have our model features and targets, let's construct a regression model. This is a pretty ideal modeling situation, given that there are so many observations and so few features. Therefore I'll likely not need a super fancy model. I'll choose to just use Linear Regression for now.

One thing that is helpful for interpreting Linear Regression models is to add some form of regularizer penalty. In principle regularizers such as the Lasso penalty perform feature selection by influencing the distribution of model parameters to be sparse (i.e. many equal to zero). However, regularizers generally require setting a hyperparameter that determines the strength of the regularizer's influence. Since we don't know a priori the correct hyperparameter setting, we'll try out a bunch, validating each one using cross validation. Specifically, we'll hold out random portions of the training data, and use it as a cross-validation set. We'll then estimate the model for a given hyperparameter setting, then assess the model accuracy on the cross-valiation set. This process is repeated many times, and the hyperparameter that offers the most accurate model is kept.

Scikits-learn offers some nice functionality for performing such hyperparameter search. In particular, I'll use the GridSearchCV module, which automatically selects hyperparameters to test from a provided grid of possible values.

Before model fitting, I'll shuffle the observation order remove redundant local structure in the data; this will keep the optimization algorithm used to estimate the coefficients from getting stuck in the wrong location in parameter space. I'll also remove the first 1000 observations and use them as as a testing set used to asses the best cross-validated model's performance. The testing set is important as it is an indicator of how the final model will perform on arbitrary data.

The entire model fitting analysis is below:

```
In [22]: from time import time
         from operator import itemgetter
         from sklearn.linear_model import Lasso
         from sklearn.grid_search import GridSearchCV
         from sklearn.utils import shuffle
         from sklearn.metrics import r2_score
```

```
        def report(grid_scores, n_top=3):
            """For reporting best models"""
            top_scores = sorted(grid_scores, key=itemgetter(1), reverse=True)[:n_top]
            for i, score in enumerate(top_scores):
                print("Model with rank: {0}".format(i + 1))
                print("Mean validation score: {0:.3f} (std: {1:.3f})".format(
                      score.mean_validation_score,
                      np.std(score.cv_validation_scores)))
                print("Parameters: {0}".format(score.parameters))
                print("")

        # SHUFFLE DATA
        model_features, targets = shuffle(model_features, targets)

        train_features = model_features[1000:]
        train_targets = targets[1000:]

        # TEST ON FIRST 1000 OBSERVATIONS
        test_features = model_features[:1000]
        test_targets = targets[:1000]

        ### INIT LINEAR REGRESSION MODEL
        print '_'*20 + ' Linear Regression Model ' + '_'*20 + '\n'
        regr = Lasso()

        # PARAMETER GRID FOR REGULARIZED LINEAR REGRESSION
        param_grid_lr = {"alpha": np.logspace(-5,1,10)}

        # RUN PARAMETER ESTIMATION
        n_iter_search = 10

        # GRID SEARCH OBJECT
        random_search_lr = GridSearchCV(regr, param_grid=param_grid_lr)

        # FIT MODELS
        start = time()
        random_search_lr.fit(train_features, train_targets)

        print("Linear Regression Estimation using RandomizedSearchCV took %.2f seconds for %d candidat
              "\nBest Parameter settings:" % ((time() - start), n_iter_search))
        print '_'*25 + '\n'

        report(random_search_lr.grid_scores_)
        model = random_search_lr.best_estimator_
```

```
-------------------- Linear Regression Model --------------------

Linear Regression Estimation using RandomizedSearchCV took 18.63 seconds for 10 candidates models
Best Parameter settings:
-----------------------

Model with rank: 1
Mean validation score: 0.672 (std: 0.000)
```

```
Parameters: {'alpha': 1.0000000000000001e-05}

Model with rank: 2
Mean validation score: 0.672 (std: 0.000)
Parameters: {'alpha': 4.6415888336127818e-05}

Model with rank: 3
Mean validation score: 0.672 (std: 0.000)
Parameters: {'alpha': 0.00021544346900318823}
```

We see that the grid search determines a low amount of regularization for this problem (i.e. performing little feature selection). This is not at all surprising given the fact that there are so few features. We also see that the best models generally predict 67 percent of the variance in the validation set responses. This corresponds to a correlation coefficient of 0.81, not too shabby for such a simple model.

But what we really care about is how well the model generalizes to completely new data (i.e. the testing set). Let's see how it does on the testing set:

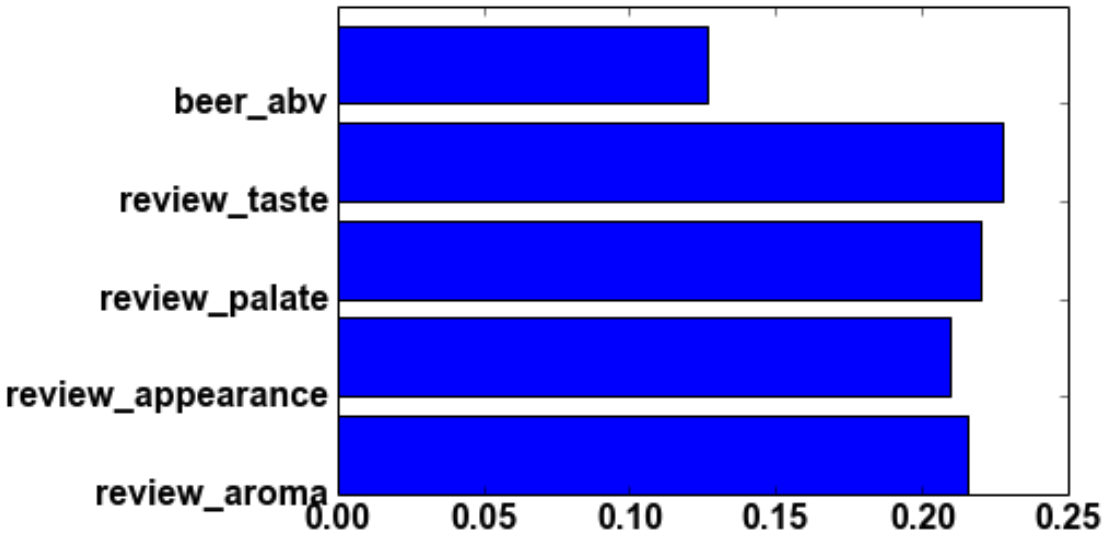
```
In [23]: r2 = r2_score(model.predict(test_features),test_targets)
         cc = np.sqrt(r2)
         print 'Percent of ratings variance explained on testing set is %1.2f (correlation %1.2f)' % (r
```

```
Percent of ratings variance explained on testing set is 0.57 (correlation 0.76)
```

Over fifty percent of variance explained by 5 predictor variables isnt too bad. In fact, the chance of getting this prediction accuracy on the 1000 testing observations is basically zero (pvalue $<$ 1e-12).

Now, let's look at what the model represents, in terms of the beer features. To do so, I'll project the model coefficients back into the original space by applying a reverse of the standardization and Gaussianization transform applied to the features. The results are plotted below:

```
In [24]: # RETREIVE MODEL COEFFICIENTS AND REVERSE NORMALIZATION AND
         # COMPRESSION
         yy = range(len(model.coef_))
         coeff = exp(-(model.coef_*feature_std) + feature_means)
         plt.barh(yy,coeff);
         plt.yticks(yy,features);
```



The inverse-transformed coefficients suggest that overall beer ratings are most related to a beer's rated taste, followed closely by the beer's palate and aroma. It turns out that ABV is less related to the overall rating than the remaining four features. Cool!

6 Other things to try

The regression analyses could be extended in a number of ways including: - Hand-designed features. Perhaps we could hang-engineer better features for predicing overall rating. For example, including interaction terms or the time of review. - Other, more nonlinear model. Perphaps the nonlinear transformed provided by logarithm is incorrect. Other models such as a neural network or decision trees could determine a more fitting (pun intended) nonlinear transformation of the input features. - Feature learning. It seems that a lot of the features used are redundant (i.e. similar model weights). Perhaps the model can be improved by using features derived using some factor analysis or feature learning approach such as Principal or Independent Components analysis (PCA or ICA).