# Predictive Beer Analytics

Joshua Weinstein, Jim Sundkvist, Marek Kühn

***Abstract*—This project aims to apply various concepts of Python programming as well as data mining and processing. Based on a large amount of user and beer data, the application uses varying techniques including data mining, natural language processing, and color analysis to predict user-defined beers' desirability in the world through a web application.**

## I. Introduction

The Predictive Beer Analytics project has been initiated in the Fall of 2014 as a part of "*Data Mining Using Python*" course at Denmark Technical University. Its purpose is not only to exercise the study material, but also to give users some useful, informative results. To satisfy this, we used our previous experience with web development and created a lightweight web-service with visual representations of the results. The users can design their own beer by defining its style, alcohol by volume, keywords and dominant label color. This information is then used to generate a desirability map.

The application to retrieve and analyse data along with our web application (without the database) can be found on our GitHub account. The web-service can be run locally with an installation of Django, MySQL, and a copy of the database.

For more information, the documentation is available on ReadTheDocs.org.

## II. Data mining

Being the main aspect of any data mining project, retrieving the data, and a lot of it, is the first course of action. All data used by Predictive Beer Analytics was gathered through the Untappd API, a beer review website which provides access to the data it collects. Unfortunately, the service limited our access to their data to 100 calls an hour. This significantly increased the amount of time required to gather data. The script, `untappd.py` provides data structures to store Untappd data as well as uses the *Requests* module to make calls to Untappd's servers. Due to the available requests through the API service, it was necessary to first retrieve a list of active users and then later use their username to gather information about beer they have reviewed in the past. Over the course of roughly two weeks of mining, a reasonable amount of information was stored. Seen in Figure 1, over 200.000 user reviews are used to predict the desirability of a beer based on location. To keep this information up-to-date, `dataQuantity.py` script has been included in the library.

After data has been gathered, updated, and normalized with methods in the main script `predictiveBeerAnalytics.py`, it is split up into users and beers where users contain a location, using the Google Geocoding API, and their ratings and beers are composed of descriptive keywords, alcohol content, label's

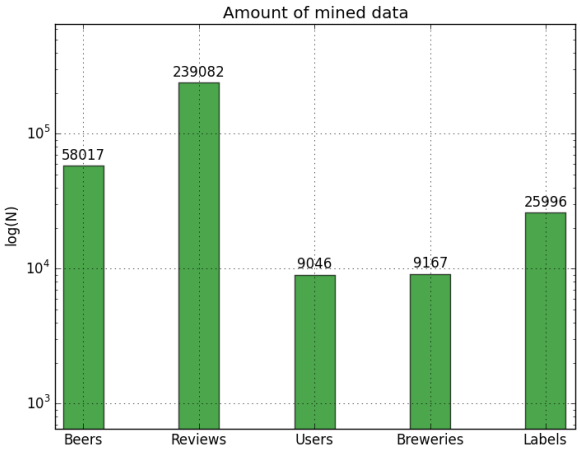


Fig. 1. Amount of used data (by the time of presentation)

url, style, and other unused information. These are then used in conjunction with other scripts we have written to analyse label colors, determine keywords from descriptions, find relationships between alcohol content and location, and determining relationships between a user's location and the style of beer they enjoy.

## III. Machine learning

There are two applications of machine learning:

- **Natural language processing**
- **Image color analysis**

With its extraordinary lexical resources and easy-to-use interface, **Natural Language Toolkit** (NLTK) was an easy choice. It is used to tokenize the sentence in order to classify the words according to its type. After various filtering (usage threshold, invalid characters etc.), we calculate the average rating of each keyword. The best, worst and the most used keywords are presented on the website. The relevance of the results is to be discussed later. The actual **keyword extraction** function is defined in `keywordExtractor.py`, making it easy to reuse in other projects. It is possible to introduce regular expressions in order to improve the keyword recognition, but the results are reasonable even without using it.

While thinking about all aspects that can possibly affect the beer desirability, we introduced also the visual point of view. We have no information about the color of the bottle. Nevertheless, more than half of the beers from untappd have a link to the beer label picture attached. All we had to

do is determine how the beers are rated in relation to the label color. The **dominant colors clustering** is done with sklearn.cluster module, built on NumPy, SciPy, and matplotlib, well-known open-source libraries. All of which are also used regularly in other context in this project. Currently, we use **K-means clustering** to determine five of the most used colors in each label. The class responsible for the calculations `labels.py:Image` is designed such as the parameters like number of clustered colors are easy to change in the future.

To check if the result is correct, we use the clustered colors to rebuild the picture as in the figure 2. Since we observed a lot of labels not having rectangular shape, we implemented a **custom masking algorithm** based on the color differences. It seeks a border at which certain threshold is overrode and generates numpy.mask. This is covered by `labels.py:Mask`.

Having these colors extracted, they need to be associated with ratings. Otherwise there would be no measurable impact on the total beer desirability. We couldn't match those color directly to the user input, because that way its rating would have to be calculated every time the request is made, which is not acceptable given the amount of data. So we come up with a solution by using **template color palette**. It serves as a **fixed categorizing root**. None of the available functions were suitable for comparing the RGB values of the colors, so we designed our own classification algorithm. It uses the well-known euclidean distance formula to determine which color from the template color palette is the closest. However, simple distance in RGB colorspace doesn't correspond to human perception. A change of one parameter results in much more different color than the same change distributed between multiple parameters. It was possible to overcome this by converting the colors to **YUV colorspace** (Fig. 3), which takes human perception into account, making the response more or less linear.

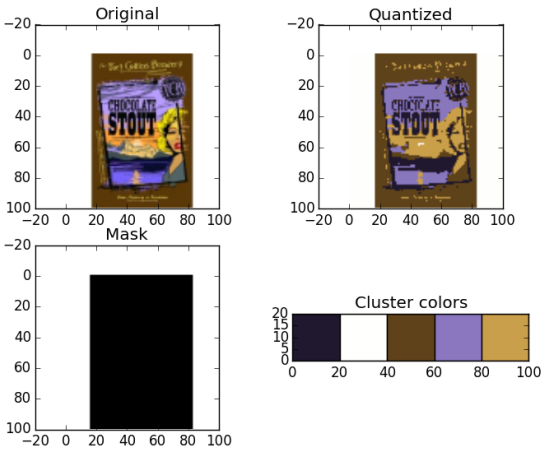


Fig. 2. Image color clustering

Fig. 3. YUV linear colorspace sample for Y = 0

## IV. Web application

The web-service uses one of the most popular Python frameworks - Django. This allowed us to create elegant web application while still being able to focus on the server-side python scripts rather than web-design.

The web-design and structure of the web application is pretty straight forward: A landing page that serves as a portal and first encounter information of the web applications functionality. A prediction page where the core of PBA is preformed. The user can fill in data that would reflect their beer and check where it would do well with a histogram. A Description page where a off the bat ranking of the processed words are presented. Here it is also possible to search for a specific word. The result will be a list of similar words and their ranking starting form the highest rating. A color page where the user can see the rankings off colors used in beer labels. An about page where we tell the story and information about Predictive Beer Analytics. And last a map page where we showcase some predefined results from the predictor.

Django uses the MVC architecture. So the Project has a data model, representing our processed data was created. A single view file that contains all (as the size of the Web application was not large enough to warrant separation of concerns in our eyes) application level functionality. And templates which uses Css and Javascript to make a smoother appearance. Bootstrap was used to speed up the styling in this regard. All templates extends the base.html template which acts as a master page.

The data that the web application uses are stored in a MySQL database. The data itself is generated from the data mining scripts and saved to the webapps database by importing them as a csv file. The web application uses **Django's database-abstraction API** to perform the basic CRUD functionality to the data model

SE FIGURE ?.

When it comes to security, Django has some great features that will bring about basic **security** with minimum effort. Django's database-abstraction API automatically sanitizes any input so **SQL-injection is not a problem**. Cross-site-scripting is dealt with by using **cross-site request forgery (CSFR) tokens**. Those are automatically attached to the server responses

so we need to add it only while doing POST.

Django also have some **predefined user authentication and authorization functionality**. In the web application we do not use any access control so this part is unnecessary until a time arrive when we do need it.

The decision to use these technologies is pretty straight forward: The technologies were familiar, widely used, well documented and easy to use. This lead to quite a fast development of the Predictive beer analytics GUI.

## V. Documentation

The documentation has been created with Sphinx tool. It allows us to generate the documentation in various formats including html, pdf, epub etc. from reStructuredText and python sources. For this to work, the `doctstring` convention is required. Apart from this, we also performed a brief code-style checking according to PEP conventions. The documentation is hosted on http://predictive-beer-analytics.readthedocs.org/, which integrates directly to the GitHub account.

## VI. Discussion

There is usually no simple way to determine the relevance of the results in these kind of projects. However, we tried to validate it as much as possible.

As we went through the list of keywords, we found a lot of invalid words which NLTK wasn't able to recognize. For example links, unknown abbreviations, misspelled word, html snippets etc. Those could be easily eliminated by defining usage threshold, that is to filter out words used less times than let's say 20. This is also to make the results more reliable, since the worst ones are used just once. There is still space for improvements here. For instance we can imagine using regular expressions for the rating of English phrases and idioms instead of just separate keywords. Altough after having the results generated, they seem reasonable enough for purposes of this project. In the end, it is only logical to expect words like "inexpensive" and "watery" to be much worse rated than "immeasurable" or "aged". Other language support has not been considered in this project.

It is problematic to assign ratings to the whole color palette. There was an idea of double-clustering the colors to form rated groups, but it made them look dull and actually quite distant from the original ones. This method is too radical. Consequently, we decided to use supervised classification to assign the colors to template palette that covers all the common colors. At this point the palette is hard-coded in the labels:Image class. It is shared with the web-service and also used as a prediction parameter. To use full color picker on the website (e.g. using jQuery), it is possible to use the same classification function defined in `labels.py`.

## VII. Further development

At this early stage, the quality of the results highly increases with getting new data. The scripts to gain information from UNTAPPD are finished, so the only limit is the time. As seen in the figure 4, there are still some areas with little or no user reviews. Those are rendered as zero rating blue. Another sources of beer information and reviews could also be implemented in the future. It would be interesting to relate brewery information to the beer popularity. The required data is already available.

Depending on the quantity of mined data, the density of the map grid can be increased. Current grid area varies between 30 000 $km^2$ and 200 000 $km^2$. Another idea to consider would be implementing different location distribution method. Now we use latitude/longitude classification, but Google API offers also to lookup the current state or city. This would all lead to more detailed results. Machine learning could be improved by introducing more sophisticated algorithm to extract keywords and phrases.

## VIII. Results

Using `PBAMap.py`, `keywordClassifier.py` we are able to graphically visualize the effects of specific words, alcohol content, and beer style on a user's rating. Figure 4 shows the consistent and seemingly global idea that people typically enjoy stronger beers. While there is a clear correlation between user reviews and and a beer's alcohol content, the type of beers an area likes are typically varied and relatively unpredictable. Though it does seem that the United States enjoys IPAs more than ciders and fruitier beers.

It was also interesting to see that certain words used in a beer's description had a large impact on a user's review of the beer. For example, beer descriptions containing words like "inexpensive", "alcohol-free", and "grandma" all had an average rating of below three out of five stars. Whereas "sought-after", "chocolate-covered", and "fabled" all had ratings above four stars.

## IX. Conclusion

At the beginning of Predictive Beer Analytics, our goal was to find relationships between the alcohol content or style of a beer and the location of a user's rating to determine what areas enjoy the most. As the project progressed, ambitions rose and we wanted to see if the color of the label or the description of beers made a significant impact on the desirability of a beer. Using techniques described in the data mining course, it was found that there were relationships between the beer properties and user ratings. There were also findings on the locations of certain preferred types of beer. It can be shown that the challenge of determining what makes a good beer was overcome.

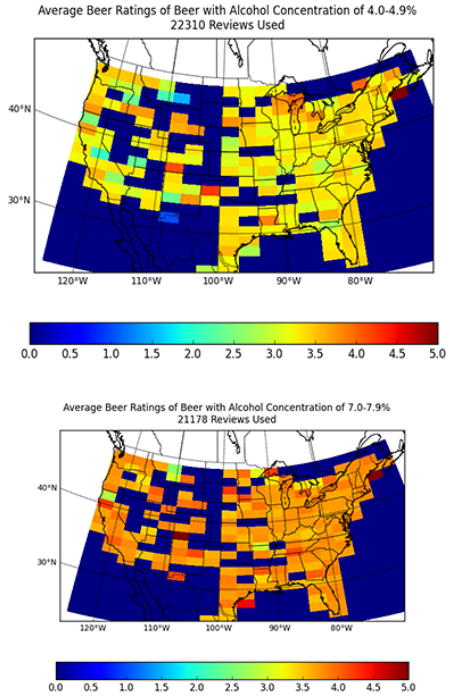


Fig. 4. Comparison of average beer ratings in US based on alcohol content

# Appendix A
# Code listings

## Listings

../machine_learning/lib/predictiveBeerAnalytics.py . . . 4
../machine_learning/lib/untappd.py . . . 9
../machine_learning/lib/keywordExtractor.py . . . 10
../machine_learning/lib/keywordClassifier.py . . . 11
../machine_learning/lib/dataPoints.py . . . 13
../machine_learning/lib/PBAMap.py . . . 13
../machine_learning/lib/labels.py . . . 16
../machine_learning/lib/fileReader.py . . . 20
../machine_learning/lib/dataQuantity.py . . . 21

```
1  """
2  This is the main script for Predictive Beer Analytics project.
3
4  Use it to mine and process new data.
5  """
6
7  import argparse
8  import jsonpickle as jpickle
9  import os
10 import operator
11 import csv
12 # import cPickle
13 from time import sleep
14 import untappd as UT
15 import PBAMap
16 import keywordExtractor as extract
17 import dataPoints as dp
18 import labels
19 import fileReader as files
20
21 parser = argparse.ArgumentParser(prog='PBA')
22 group = parser.add_mutually_exclusive_group(required=True)
23 group.add_argument('--users', action='store_true',
24                    help='Add to the list of users')
25 group.add_argument('--reviews', action='store_true',
26                    help='Add to the list of users, beers, and breweries')
27 group.add_argument('--normalizeData', action='store_true',
28                    help='Alter Untappd data for privacy')
29 group.add_argument('--keywords', action='store_true',
30                    help='Extract keywords from beer descriptions and attach to beer')
31 group.add_argument('--dataPoints', action='store_true',
32                    help='Create list of data points from user locations, ratings, \
33 ____________________beer alchol content, and beer style')
34 group.add_argument('--styles', action='store_true',
35                    help='Create csv file of allowable beer styles to make maps with')
36 group.add_argument('--abvMap', type=float,
37                    help='Create map of ratings using data points on maps \
38 ____________________provided alcohol level. Data is split into abv ranges: \
39 ____________________0, 0.1 - 3.9, 4.0 - 4.9, 5.0 - 5.9, 6.0 - 6.9, 7.0 - 7.9, 8.0 - 8.9, \
40 ____________________9.0 - 9.9, 10.0 - 10.9, 11.0+ Requires GEOS Library and \
41 ____________________mpl_toolkits.basemap')
42 group.add_argument('--styleMap', type=str,
43                    help='Create map of ratings using data points on maps \
44 ____________________provided beer style. Styles limited to those listed in \
45 ____________________styles.csv. Requires GEOS Library and mpl_toolkits.basemap')
46 group.add_argument('--colorPalette', action='store_true',
47                    help='Download label images, clusterize colors, \
48 ____________________generate global color rating palette of N colors.')
49 args = parser.parse_args()
50
51 # set the api settings and create an Untappd object
52 untappd = UT.Untappd()
53 untappd.settings('../apiConfig.ini')
54
55
56
57 def writeJSONFile(path, data):
58     """Write JSON file."""
59     with open(path, 'wb') as jsonFile:
60         json = jpickle.encode(data)
61         jsonFile.write(json)
62
63
64 def usersList():
65     """
66 ____Parse through data from /thepub to get unique usernames, user ids,
67 ____and locations. Stores this information in a csv file to be used in later api
```

```
    requests. Limited to 100 api calls per hour requiring sleep method.
    May be run multiple times to retrieve Continuously run until user stops script.
    """

    usersList = files.readUsers()
    apiCount = 0
    userNameCountAdditions = 0
    while (True):
        # get 25 most recent updates
        data = untappd.getPubFeed()
        apiCount += 1
        print 'apiCount: ' + str(apiCount)
        checkins = data['response']['checkins']['items']
        # each response has 25 items, each with a username
        for checkin in checkins:
            userId = checkin['user']['uid']
            username = checkin['user']['user_name']
            userLocation = checkin['user']['location']
            if hash(str(userId)) not in usersList:
                if userLocation != '':
                    userNameCountAdditions += 1
                    userAttribs = {'uid': str(userId), 'username': username,
                                   'location': {'name': unicode(userLocation).encode("utf-8")}, 'ratings': {}}
                    user = UT.UntappdUser(userAttribs)
                    usersList[hash(str(userId))] = user
        writeJSONFile('../data/users.json', usersList)
        userCount = len(usersList)
        print 'Total Users: ' + str(userCount)
        # Untappd only allows 100 api requests per hour. Sleep for 38
        # seconds between requests
        sleep(37)


def userReviews():
    """
    Parse through user reviews /user/beers/{username}
    Retrieves at most 50 reviews per user, retains review, beer, and
    brewery information. After querying the api, remove username to
    lessen privacy concerns with untappd data.
    """
    usersList = files.readUsers()
    beersList = files.readBeers()
    breweryList = files.readBreweries()
    breweryToBeers = files.readBreweryToBeers()

    total = 0
    totalUsersComplete = 0
    for userHash, user in usersList.iteritems():
        totalUsersComplete += 1
        # if the data has been normalized, old data will not
        # have usernames. Ignore older users which may have
        # already gotten reviews
        if user.username:
            userId = user.uid
            username = user.username
            user.username = None
            userReviewCount = 0
            offsetTotal = 0
            ratings = {}

            print 'Processing ' + str(userId) + ': ' + username
            # each response returns at most 25 reviews. To get more user
            # reviews, call again with an offset get at most 50 reviews
            # from the same user
            while (userReviewCount < 2):
                print username + ': ' + str(userReviewCount + 1)
                data = untappd.getUserReviewData(username, offsetTotal)
                offset = data['response']['beers']['count']
                offsetTotal += offset
                reviews = data['response']['beers']['items']
                for review in reviews:
                    userRating = review['rating_score']
                    if userRating > 0:
                        beerInfo = review['beer']
                        breweryInfo = review['brewery']
                        # fill in beer information
                        if hash(str(beerInfo['bid'])) not in beersList:
                            stylesList = []
                            style = unicode(beerInfo['beer_style']).encode("utf-8")
                            styles = style.lower().title().split('/')
                            for style in styles:
                                style = style.strip()
                                stylesList.append(style)
                            beerAttribs = {
                                'bid': str(beerInfo['bid']),
                                'name': unicode(beerInfo['beer_name']).encode("utf-8"),
```

```
                                'label': beerInfo['beer_label'],
                                'abv': beerInfo['beer_abv'],
                                'ibu': beerInfo['beer_ibu'],
                                'style': stylesList,
                                'description': unicode(beerInfo['beer_description']).encode("utf-8"),
                                'rating': beerInfo['rating_score'],
                                'numRatings': 1,
                                'brewery': str(breweryInfo['brewery_id'])
                            }
                            beer = UT.UntappdBeer(beerAttribs)
                            beersList[hash(beer.bid)] = beer
                        else:
                            beersList[hash(str(beerInfo['bid']))].numRatings += 1
                        # fill in brewery information
                        if hash(str(breweryInfo['brewery_id'])) not in breweryList:
                            breweryAttribs = {
                                'breweryId': str(breweryInfo['brewery_id']),
                                'name': unicode(breweryInfo['brewery_name']).encode("utf-8"),
                                'label': breweryInfo['brewery_label'],
                                'country': unicode(breweryInfo['country_name']).encode("utf-8"),
                                'location': unicode(breweryInfo['location']).encode("utf-8")
                            }
                            brewery = UT.UntappdBrewery(breweryAttribs)
                            breweryList[hash(brewery.breweryId)] = brewery

                        # map breweery_id to a list of beers produced there
                        if hash(str(breweryInfo['brewery_id'])) not in breweryToBeers:
                            # store the current beer in a list of beers of
                            # the brewery
                            breweryToBeers[hash(str(breweryInfo['brewery_id']))] = {str(breweryInfo['brewery_id']): [str(beerInfo['bid'
                        else:
                            # add current beer to brewery's list of beers
                            breweryToBeers[hash(str(breweryInfo['brewery_id']))][str(breweryInfo['brewery_id'])].append(str(beerInfo['b

                        # add list of beer ratings to user
                        ratings[str(beerInfo['bid'])] = userRating
                userReviewCount += 1
                user.ratings = ratings

                # store the dictionaries after new data so user doesn't kill process before writing
                # with open('../data/users.json', 'wb') as usersFile:
                #     json = jpickle.encode(usersList)
                #     usersFile.write(json)
                # with open('../data/beers.json', 'wb') as beersFile:
                #     json = jpickle.encode(beersList)
                #     beersFile.write(json)
                # with open('../data/breweries.json', 'wb') as breweriesFile:
                #     json = jpickle.encode(breweryList)
                #     breweriesFile.write(json)
                # with open('../data/breweryToBeers.json', 'wb') as breweryToBeersFile:
                #     json = jpickle.encode(breweryToBeers)
                #     breweryToBeersFile.write(json)

                # if the offset is less than 25, then there are no more reviews to retrieve
                if offset < 25:
                    break
            writeJSONFile('../data/users.json', usersList)
            writeJSONFile('../data/beers.json', beersList)
            writeJSONFile('../data/breweries.json', breweryList)
            writeJSONFile('../data/breweryToBeers.json', breweryToBeers)

            total += len(ratings)
            print str(userId) + ': ' + username + ', Processed: ' + str(len(ratings)) + ' reviews'
            print 'Total Reviews: ' + str(total)
            print 'Total Users Completed: ' + str(totalUsersComplete)
            sleep(37 * (userReviewCount))
        else:
            total += len(user.ratings)


def normalizeUsers():
    """
    Change the user ids so the information can be made public and
    use the googlemaps module to determine the user's location.
    """
    usersList = files.readUsers()
    newUsersList = {}

    i = 1
    newUid = 1
    for hashId, user in usersList.iteritems():
        uid = user.uid
        user.uid = str(newUid)
        location = user.location
        if location['name'] != "" and 'lat' not in location:
            if isinstance(location['name'], unicode):
```

```
240                 location = location['name'].encode('utf-8')
241             else:
242                 location = location['name']
243 
244             mapInfo = PBAMap.getLatLong(location, i)
245             i += 1
246             if mapInfo == 'apiLimit':
247                 print str(i) + " At daily API limit. Update script and repeat tomorrow"
248             elif mapInfo != '':
249                 user.location = {
250                     'name': location,
251                     'lat': mapInfo['lat'],
252                     'lng': mapInfo['lng'],
253                 }
254                 if 'country' in mapInfo:
255                     user.location['country'] = mapInfo['country']
256                 print str(i), user.location
257             else:
258                 print str(i), "checked: none"
259                 user.location = {'name': ''}
260         newUid += 1
261         newUsersList[hash(str(uid))] = user
262 
263     writeJSONFile('../data/users.json', newUsersList)
264     print "User ids, usernames, and locations updated\n"
265 
266 
267 def beerKeywords():
268     """Extract keywords from beer descriptions and rate it."""
269     beersList = files.readBeers()
270     print 'beers.json loaded...'
271 
272     # List of keywords generation
273     keywordsList = {}
274     position = 0
275 
276     for hashId, beer in beersList.iteritems():
277         beer.keywords = []
278         beer.keywords = extract.extractKeywords(beer.description)
279         for keyword in beer.keywords:
280             if keyword in keywordsList:
281                 keywordsList[keyword][0] += beer.rating
282                 keywordsList[keyword][1] += 1
283             else:
284                 keywordsList[keyword] = [beer.rating, 1]
285         position += 1
286         if (position % 100) == 0:
287             print 'Processed ' + str(position) + '/' + str(len(beersList)) + ' beers.'
288 
289     writeJSONFile('../data/beers.json', beersList)
290     writeJSONFile('../data/keywords.json', keywordsList)
291 
292 
293 def createDataPoints():
294     """Make the data points of user locations for the map generation."""
295     usersList = files.readUsers()
296     beersList = files.readBeers()
297     points = []
298     i = 1
299     for hashId, user in usersList.iteritems():
300         if 'lat' in user.location and user.ratings:
301             for bid, rating in user.ratings.iteritems():
302                 country = None
303                 if 'country' in user.location:
304                     country = user.location['country']
305                 pointAttribs = {'lat': user.location['lat'], 'lng': user.location['lng'],
306                 'country': country, 'abv': beersList[str(hash(bid))].abv, 'rating': rating,
307                 'style': beersList[str(hash(bid))].style}
308                 point = dp.dataPoint(pointAttribs)
309                 points.append(point)
310                 if i % 1000 == 0:
311                     print "Points added: " + str(i)
312                 i += 1
313     data = dp.dataPoints(points)
314     writeJSONFile('../data/dataPoints.json', data)
315 
316 
317 def createABVMap():
318     """Make a color map of specific alcohol by volume."""
319     print "Drawing user rating maps of beers with an alcohol concentration of " + str(args.abvMap) + '%'
320     dataPoints = files.readDataPoints()
321     if len(dataPoints) > 0:
322         abv = int(args.abvMap)
323         points = []
324         for point in dataPoints:
325             if int(point.abv) == abv:
```

```
326                 points.append(point)
327             elif (abv in [1, 2, 3]) and (int(point.abv) in [1, 2, 3]):
328                 points.append(point)
329             elif (abv > 11 and int(point.abv) > 11):
330                 points.append(point)
331         print str(len(points)) + " points of data"
332         PBAMap.drawMap(points, abv)
333     else:
334         print "No data points found"
335 
336 
337 def createStyleMap():
338     print "Drawing user rating maps of beers with a style of " + str(args.styleMap)
339     dataPoints = files.readDataPoints()
340     if len(dataPoints) > 0:
341         style = args.styleMap
342         points = []
343         for point in dataPoints:
344             if style in point.style:
345                 points.append(point)
346         print str(len(points)) + " points of data"
347         PBAMap.drawMap(points, style)
348     else:
349         print "No data points found"
350 
351 
352 def createCommonStyles():
353     """Generate common beer styles and save it to csv file."""
354     beersList = files.readBeers()
355     allStyles = {}
356     for hashId, beer in beersList.iteritems():
357         styles = beer.style
358         for style in styles:
359             numRatings = beer.numRatings if (hasattr(beer, 'numRatings')) else 0
360             if style in allStyles:
361                 allStyles[style] += numRatings
362             else:
363                 allStyles[style] = numRatings
364 
365     sorted_styles = sorted(allStyles.items(), key=operator.itemgetter(1))[-20:]
366     with open('../data/styles.csv', 'wb') as stylesCSV:
367         csvwriter = csv.writer(stylesCSV, delimiter=',',
368                                quotechar='"')
369         csvwriter.writerow(["id", "style", "numRatings"])
370         i = 1
371         for style in sorted_styles:
372             csvwriter.writerow([i, unicode(style[0]).encode("utf-8"), style[1]])
373             i += 1
374 
375 
376 def processLabels():
377     """
378     Prediction of how dominant label color affects the beer rating.
379 
380     Download beer bottle labels, extract n dominant colors,
381     make the color palette, flag each color and calculate
382     average rating of that color.
383     """
384     beersList = files.readBeers()
385     beerColorsDict = files.readBeerColors()
386 
387     # Path for saving the images
388     path = "../data/labels/"
389 
390     fileList = os.listdir(path)
391     fileList = [item for item in fileList
392                 if item.split(".")[-1] in ('jpeg', 'jpg', 'png')]
393 
394     # Download and save images
395     labels.download(beersList, path, fileList)
396 
397     # Number of label colors to cluster
398     nColors = 5
399     i = 0
400     # stop = 6  # Then use the whole list.
401 
402     # Loop over images in the folder
403     for file in fileList:
404         i += 1
405         bid = unicode(file.split('.')[0])
406         if (bid in beerColorsDict and
407                 len(beerColorsDict[bid].colorPaletteFlags) == nColors):
408             continue
409 
410         print ("Processing image " + file +
411                "[" + str(i - 1) + "/" + str(len(fileList)) + "]")
```

```
412             beerLabel = labels.Image(path + file)
413
414             beerLabel.preprocess()
415             beerColor = beerLabel.clusterize(nColors)
416             beerColorsDict[bid] = beerColor
417
418             # Only for presentation
419             # beerLabel.quantizeImage()
420             # beerLabel.showResults()
421
422         # Generate the color palette with ratings - Classification
423         colorPalette = labels.ColorPalette()
424         colorPalette.build(beerColorsDict, beersList)
425
426         # Write the colorsFile - dict{ 'bid': beerColor{RGB,intensity}}
427         writeJSONFile('../data/beerColors.json', beerColorsDict)
428         writeJSONFile('../data/colorPalette.json', colorPalette.palette)
429
430         print 'Color palette saved.'
431
432     if args.users:
433         usersList()
434     elif args.reviews:
435         userReviews()
436     elif args.normalizeData:
437         normalizeUsers()
438     elif args.keywords:
439         beerKeywords()
440     elif args.styles:
441         createCommonStyles()
442     elif args.colorPalette:
443         processLabels()
444     elif args.dataPoints:
445         createDataPoints()
446     elif args.abvMap >= 0:
447         createABVMap()
448     elif args.styleMap:
449         if args.styleMap in files.readBeerStyles():
450             createStyleMap()
451         else:
452             print "No support for style: " + args.styleMap
453             print "See styles.csv for supported styles."
```

---

```
1   import requests
2   import json
3   from ConfigParser import SafeConfigParser
4
5
6   class UntappdUser:
7       """
8       Representation of an untappd user
9       """
10      def __init__(self, attribs):
11          self.uid = attribs['uid']
12          self.username = attribs['username']
13          self.location = attribs['location']
14          self.ratings = attribs['ratings']
15
16
17  class UntappdBeer:
18      """
19      Representation of an untappd beer
20      """
21      def __init__(self, attribs):
22          self.bid = attribs['bid']
23          self.name = attribs['name']
24          self.label = attribs['label']
25          self.abv = attribs['abv']
26          self.ibu = attribs['ibu']
27          self.style = attribs['style']
28          self.description = attribs['description']
29          self.rating = attribs['rating']
30          self.numRatings = attribs['numRatings']
31          self.brewery = attribs['brewery']
32
33
34  class UntappdBrewery:
35      """
36      Representation of an untappd brewery
37      """
38      def __init__(self, attribs):
39          self.breweryId = attribs['breweryId']
40          self.name = attribs['name']
41          self.label = attribs['label']
42          self.country = attribs['country']
43          self.location = attribs['location']
```

```
44
45
46  class Untappd:
47      """
48  ____Untappd object which handles everything related to untappd
49  ____and the data obtained through untappd's api
50  ____"""
51      def __init__(self):
52          self.client_id = ''
53          self.client_secret = ''
54          self.endpoint = ''
55          self.request_header = {}
56          self.users = {}
57          self.beers = {}
58          self.breweries = {}
59
60      def settings(self, filename):
61          config = SafeConfigParser()
62          config.read(filename)
63          client_id = config.get('untappd', 'clientId')
64          client_secret = config.get('untappd', 'clientSecret')
65          endpoint = config.get('untappd', 'endpoint')
66          request_header = {'User-Agent': config.get('untappd', 'header')}
67
68          self.client_id = client_id
69          self.client_secret = client_secret
70          self.endpoint = endpoint
71          self.request_header = request_header
72
73      def createUrl(self, method):
74          """
75  ________Creates the api url for the GET request
76  ________"""
77          return self.endpoint + '/' + method
78
79      def getPubFeed(self):
80          """"
81  ________Retrieves information which includes the usernames of
82  ________active members of the site
83  ________"""
84          method = 'thepub'
85          url = self.createUrl(method)
86          parameters = {'client_id': self.client_id,
87                  'client_secret': self.client_secret}
88          response = requests.get(url, headers=self.request_header,
89                                  params=parameters)
90          data = json.loads(response.text)
91          return data
92
93      def getUserReviewData(self, username, offset):
94          method = 'user/beers/' + username
95          url = self.createUrl(method)
96          parameters = {'offset': offset, 'client_id': self.client_id,
97                  'client_secret': self.client_secret}
98          response = requests.get(url, headers=self.request_header,
99                                  params=parameters)
100         data = json.loads(response.text)
101         return data
```

---

```
1   '''
2   Extract keywords from beer descriptions obtained from Untappd.
3   Use NLTK natural language processing tool.
4   '''
5
6   import nltk
7   from nltk.corpus import wordnet as wn
8   import re
9   import jsonpickle as jpickle
10  import untappd as UT
11
12
13  def extractKeywords(text):
14      """Extract the keywords from the given the text."""
15      keywords = []
16      sentences = nltk.sent_tokenize(text)
17      words = [nltk.word_tokenize(sent) for sent in sentences]
18      words = [nltk.pos_tag(sent) for sent in words]
19
20      # Regex for more precise extraction
21      # grammar = "NP: {<DT>?<JJ>*<NN>}"
22      # cp = nltk.RegexpParser(grammar)
23
24      for w in words:
25          for ww in w:
26              # Pick conditions based on word types from nltk
27              if (ww[1] == 'NN' or ww[1] == 'JJ') and len(ww[0]) > 3:
```

```
                keywords.append(ww[0])
    return keywords
```

---

```
'''
Show the usage of keywords in various ways.
Write sorted csv and text files for export to database and graphics..
'''

import jsonpickle as jpickle
import sys
import numpy as np
import matplotlib.pyplot as plt
import csv


# Load keywords
try:
    keywordsFile = open('../data/keywords.json', 'rb')
except:
    print 'Keywords.json not found.'
    sys.exit()

try:
    f = keywordsFile.read()
    keywordsRawDict = jpickle.decode(f)
except:
    print 'Keywords list corrupted'
    sys.exit()
keywordsFile.close()

# Filters - minimum usage and lowercase conversion
votesThreshold = 50
keywordsDict = {k.lower(): v for (k, v) in
                keywordsRawDict.iteritems()
                if (v[1] >= votesThreshold)}

# Dictionary of sorted lists
SortedByRating = {}
SortedByRating['keywords'] = []
SortedByRating['ratings'] = []
SortedByRating['usage'] = []
for word in sorted(keywordsDict.items(),
                   key=lambda k: (k[1][0] / k[1][1]), reverse=True):
    ratingSum = word[1][0]
    usage = word[1][1]
    if usage > 2:
        SortedByRating['keywords'].append(word[0])
        SortedByRating['ratings'].append(ratingSum / usage)
        SortedByRating['usage'].append(usage)

# Sorting by keywords usage
SortedByUsage = {}
SortedByUsage['keywords'] = []
SortedByUsage['ratings'] = []
SortedByUsage['usage'] = []
for word in sorted(keywordsDict.items(),
                   key=lambda k: (k[1][1]), reverse=True):
    ratingSum = word[1][0]
    usage = word[1][1]
    SortedByUsage['keywords'].append(word[0])
    SortedByUsage['ratings'].append(ratingSum / usage)
    SortedByUsage['usage'].append(usage)


def plotBestKeywords(n=10):
    """
    Plot a graph of keywords associated with the best rated beers.
    :param n: Amount of keywords.
    """

    fig, ax = plt.subplots()
    index = np.arange(n)

    bar_width = 0.35
    opacity = 0.4
    error_config = {'ecolor': '0.3'}

    bars1 = plt.barh(index, SortedByRating['ratings'][0:n],
                     bar_width,
                     alpha=opacity,
                     color='b',
                     error_kw=error_config,
                     label='Rating avg.')

    bars2 = plt.barh(index + bar_width, SortedByRating['usage'][0:n],
                     bar_width,
```

```
84                      alpha=opacity,
85                      color='r',
86                      error_kw=error_config,
87                      label='Usage')
88
89      plt.title('Beer rating keywords')
90      plt.yticks(index + bar_width, SortedByRating['keywords'][0:n])
91      plt.legend()
92      plt.tight_layout()
93      plt.show()
94
95
96  def plotWorstKeywords(n=10):
97      """
98      Plot a graph of keywords associated with the worst rated beers.
99      :param n: Amount of keywords.
100     """
101
102     fig, ax = plt.subplots()
103     index = np.arange(n)
104
105     bar_width = 0.35
106     opacity = 0.4
107     error_config = {'ecolor': '0.3'}
108
109     bars1 = plt.barh(index, SortedByRating['ratings'][-n:],
110                      bar_width,
111                      alpha=opacity,
112                      color='b',
113                      error_kw=error_config,
114                      label='Rating avg.')
115
116     bars2 = plt.barh(index + bar_width,
117                      SortedByRating['usage'][-n:],
118                      bar_width,
119                      alpha=opacity,
120                      color='r',
121                      error_kw=error_config,
122                      label='Usage')
123
124     plt.title('Beer rating keywords')
125     plt.yticks(index + bar_width, SortedByRating['keywords'][-n:])
126     plt.legend()
127     plt.tight_layout()
128     plt.show()
129
130
131 def plotMostUsed(n=10):
132     """
133     Plot a graph of most used keywords in the beer description.
134     :param n: Amount of keywords.
135     """
136
137     fig, ax = plt.subplots()
138     index = np.arange(n)
139
140     bar_width = 0.35
141     opacity = 0.4
142     error_config = {'ecolor': '0.3'}
143
144     bars1 = plt.barh(index, [x*1000 for x in SortedByUsage['ratings'][0:n]],
145                      bar_width,
146                      alpha=opacity,
147                      color='b',
148                      error_kw=error_config,
149                      label='Rating avg. [x1000]')
150
151     bars2 = plt.barh(index + bar_width, SortedByUsage['usage'][0:n],
152                      bar_width,
153                      alpha=opacity,
154                      color='r',
155                      error_kw=error_config,
156                      label='Usage')
157
158     plt.title('Beer rating keywords')
159     plt.yticks(index + bar_width, SortedByUsage['keywords'][0:n])
160     plt.legend()
161     plt.tight_layout()
162     plt.show()
163
164
165 def writeFiles():
166     """Export CSV and text files export."""
167     f = open('../data/sortedByRating.csv', 'wt')
168     writer = csv.writer(f)
169     writer.writerow(('id', 'Keyword', 'Rating', 'Usage'))
```

```
    for i, word in enumerate(SortedByRating['keywords']):
        try:
            writer.writerow((str(i),
                             word, SortedByRating['ratings'][i],
                             SortedByRating['usage'][i]))
        except:
            pass
    f.close()

    f = open('../data/sortedByUsage.csv', 'wt')
    writer = csv.writer(f)
    writer.writerow(('id', 'Keyword', 'Rating', 'Usage'))
    for i, word in enumerate(SortedByUsage['keywords']):
        try:
            writer.writerow((str(i), word,
                             SortedByUsage['ratings'][i],
                             SortedByUsage['ratings'][i]))
        except:
            pass
    f.close()

    # Files for graphic export to http://www.wordle.net/
    f = open('../data/mostUsedKeywords.txt', 'wt')
    for i, word in enumerate(SortedByUsage['keywords'][0:100]):
        f.write(word + "␣:␣" + str(SortedByUsage['usage'][i]) + "\n")
    f.close()

    f = open('../data/bestKeywords.txt', 'wt')
    for i, word in enumerate(SortedByRating['keywords'][0:100]):
        f.write(word + "␣:␣" + str(SortedByRating['ratings'][i]*1000) + "\n")
    f.close()

    f = open('../data/worstKeywords.txt', 'wt')
    for i, word in enumerate(SortedByRating['keywords'][100:]):
        f.write(word + "␣:␣" + str(SortedByRating['ratings'][i]*1000) + "\n")
    f.close()

plotBestKeywords(10)
plotWorstKeywords(10)
plotMostUsed(40)

writeFiles()

print 'done'
```

---

```
class dataPoint:
    def __init__(self, attribs):
        self.lat = attribs['lat']
        self.lng = attribs['lng']
        self.country = attribs['country']
        self.abv = attribs['abv']
        self.rating = attribs['rating']
        self.style = attribs['style']


class dataPoints:
    def __init__(self, points):
        self.points = points
```

---

```
import urllib
import json
from ConfigParser import SafeConfigParser
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import numpy

config = SafeConfigParser()
config.read('../apiConfig.ini')
# set the api setting values
baseUrl = "https://maps.googleapis.com/maps/api/geocode/json?"
apiKey = config.get('googleMaps', 'apiKey')
apiKey2 = config.get('googleMaps', 'apiKey2')
apiKey3 = config.get('googleMaps', 'apiKey3')
apiKey4 = config.get('googleMaps', 'apiKey4')

abvMapName = {'0': '0', '1': '.1-3.9', '2': '.1-3.9', '3': '.1-3.9',
              '4': '4.0-4.9', '5': '5.0-5.9', '6': '6.0-6.9', '7': '7.0-7.9',
              '8': '8.0-8.9', '9': '9.0-9.9', '10': '10.0-10.9', '11': '11+'}


def getLatLong(address, calls):
    # Google Geocoding api only allows 2500 api calls a day
    # Update code to handle number of users and api keys
    url = ''
    if calls < 2450:
```

```
27          url = baseUrl + "address=%s&sensor=false&key=%s" % (urllib.quote(address.replace(' ', '+')), apiKey)
28      elif calls > 2450 and calls < 4950:
29          url = baseUrl + "address=%s&sensor=false&key=%s" % (urllib.quote(address.replace(' ', '+')), apiKey2)
30      elif calls > 4950 and calls < 7450:
31          url = baseUrl + "address=%s&sensor=false&key=%s" % (urllib.quote(address.replace(' ', '+')), apiKey3)
32      elif calls > 7450 and calls < 9950:
33          url = baseUrl + "address=%s&sensor=false&key=%s" % (urllib.quote(address.replace(' ', '+')), apiKey4)
34
35      if (url != ''):
36          data = urllib.urlopen(url).read()
37          info = json.loads(data).get("results")
38          if info:
39              location = info[0].get("geometry").get("location")
40              address_components = info[0].get("address_components")
41              for component in address_components:
42                  if "country" in component["types"]:
43                      country = component["short_name"]
44                      print country
45                      location["country"] = country
46          else:
47              location = ""
48          return location
49      else:
50          return "apiLimit"
51
52
53  def createMap(lats, lngs, parallels, meridians, states=False):
54      # create polar stereographic Basemap instance.
55      m = Basemap(projection='stere', lat_0=lats[0], lon_0=lngs[0],
56              llcrnrlat=lats[1], urcrnrlat=lats[2],
57              llcrnrlon=lngs[1], urcrnrlon=lngs[2],
58              rsphere=6371200, resolution='l', area_thresh=10000)
59      # draw coastlines, state and country boundaries, edge of map.
60      m.drawcoastlines()
61      m.drawcountries()
62      if states:
63          m.drawstates()
64
65      # draw parallels.
66      m.drawparallels(parallels, labels=[1, 0, 0, 0], fontsize=10)
67      # draw meridians
68      m.drawmeridians(meridians, labels=[0, 0, 0, 1], fontsize=10)
69
70      return m
71
72
73  def saveMap(param, numPoints, filePrefix):
74      if type(param) is int:
75          if str(param) in abvMapName:
76              abvRange = abvMapName[str(param)]
77          else:
78              abvRange += abvMapName['11']
79
80          title = 'Average Beer Ratings of Beer with Alcohol Concentration of ' + \
81              abvRange + '%' + '\n' + str(numPoints) + " Reviews Used"
82
83          filename = filePrefix + 'ABV' + abvRange + '.png'
84      else:
85          title = 'Average Beer Ratings of Beer with a Style of ' + param + '\n' + \
86              str(numPoints) + " Reviews Used"
87          filename = filePrefix + 'Style' + param.replace(' ', '') + '.png'
88      plt.suptitle(title)
89      plt.savefig(filename)
90      plt.close()
91
92
93  def drawUSMap(points, param):
94      print "Drawing US map with " + str(len(points)) + " data points."
95      lats = []
96      lngs = []
97      ratings = []
98
99      for point in points:
100         lats.append(point.lat)
101         lngs.append(point.lng)
102         ratings.append(point.rating)
103
104     # US middle, lower left, and upper right
105     # latitude and longitude coordinates
106     usLat = [38, 22, 48]
107     usLng = [-97, -125, -59]
108     parallels = [30, 40]
109     meridians = [280, 270, 260, 250, 240]
110
111     # create polar stereographic Basemap instance.
112     m = createMap(usLat, usLng, parallels, meridians, True)
```

```
113
114         xs, ys, average = createHistogram(m, lats, lngs, ratings, True)
115
116         # overlay the averages histogram over map
117         plt.pcolormesh(xs, ys, average, vmin=0, vmax=5)
118         plt.colorbar(orientation='horizontal')
119         saveMap(param, len(points), '../graphics/US')
120
121
122     def drawEUMap(points, param):
123         print "Drawing EU map with " + str(len(points)) + " data points."
124         lats = []
125         lngs = []
126         ratings = []
127
128         for point in points:
129             lats.append(point.lat)
130             lngs.append(point.lng)
131             ratings.append(point.rating)
132
133         # EU middle, lower left, and upper right
134         # latitude and longitude coordinates
135         euLat = [51, 27, 71]
136         euLng = [20, -16, 45]
137         parallels = [30, 40, 50, 60, 70]
138         meridians = [350, 0, 10, 20, 30, 40]
139
140         # create polar stereographic Basemap instance.
141         m = createMap(euLat, euLng, parallels, meridians)
142         xs, ys, average = createHistogram(m, lats, lngs, ratings)
143         ######################################################
144         # xs, ys, average data will be stored in db as lists. Need to
145         # change back to ndarray later
146         # xs = numpy.array(xs.tolist())
147         # ys = numpy.array(ys.tolist())
148         # average = numpy.array(average.tolist())
149         ######################################################
150
151         # overlay the averages histogram over map
152         plt.pcolormesh(xs, ys, average, vmin=0, vmax=5)
153         plt.colorbar(orientation='horizontal')
154         saveMap(param, len(points), '../graphics/EU')
155
156
157     def inEU(lat, lng):
158         return lat >= 27 and lat <= 71 and lng >= -16 and lng <= 45
159
160
161     def inUS(lat, lng):
162         return lat >= 22 and lat <= 48 and lng >= -125 and lng <= -59
163
164
165     def createHistogram(m, lats, lngs, ratings, us=False):
166         nx, ny = 10, 10
167         if us:
168             nx, ny = 20, 20
169         # compute appropriate bins to histogram the data into
170         lng_bins = numpy.linspace(min(lngs), max(lngs), nx + 1)
171         lat_bins = numpy.linspace(min(lats), max(lats), ny + 1)
172
173         # Histogram the lats and lons to produce an array of frequencies in each box.
174         frequency, _, _ = numpy.histogram2d(lats, lngs, [lat_bins, lng_bins])
175
176         # Histogram the lats and lons to produce an array of frequencies weighted by
177         # ratings in each box.
178         weighted, _, _ = numpy.histogram2d(lats, lngs, [lat_bins, lng_bins], weights=ratings)
179
180         # divide the weighted bins by the frequency bins to create bins of average
181         # beer ratings in each bin
182         with numpy.errstate(invalid='ignore'):
183             average = numpy.divide(weighted, frequency)
184             average = numpy.nan_to_num(average)
185
186         # Turn the lng/lat bins into 2 dimensional arrays ready
187         # for conversion into projected coordinates
188         lng_bins_2d, lat_bins_2d = numpy.meshgrid(lng_bins, lat_bins)
189
190         # convert the xs and ys to map coordinates
191         xs, ys = m(lng_bins_2d, lat_bins_2d)
192
193         return xs, ys, average
194
195
196     def drawMap(points, param):
197         euPoints = []
198         usPoints = []
```

```
199
200     for point in points:
201         if inUS(point.lat, point.lng):
202             usPoints.append(point)
203         elif inEU(point.lat, point.lng):
204             euPoints.append(point)
205     drawEUMap(euPoints, param)
206     drawUSMap(usPoints, param)
```

---

```
1   """
2   Download, process, label images for all beers.
3
4   * Assign N most used colors based on K-Means clustering algorithm
5   * Classify these colors to fit the palette
6   * Rate these palette colors based on previously downloaded UNTAPPD beer ratings.
7   """
8
9   import os
10  import numpy as np
11  import numpy.ma as ma  # Masked array
12  import matplotlib.pyplot as plt
13  import matplotlib.cm as cm  # Color map
14  from sklearn.cluster import KMeans
15  from scipy import misc
16  from math import sqrt
17  import requests
18  import jsonpickle as jpickle
19  import csv
20
21
22  class Image:
23
24      """Object to manipulate image data and cluster the colors."""
25
26      def __init__(self, imgFilePath):
27          self.filePath = imgFilePath
28          self.raw_data = misc.imread(imgFilePath)
29
30          # Normalizing RGB to be in the range [0-1]. Assuming 8bit integer coding.
31          self.original_data = np.array(self.raw_data, dtype=np.float64) / 255
32          #self.original_data = np.array(self.raw_data, dtype=np.float64)
33
34          # Transform to a 2D numpy array [100x100px - no need to crop].
35          self.w, self.h, self.d = self.shape = tuple(self.raw_data.shape)
36          self.shapeId = (self.w, self.h, 1)
37          assert self.d == 3  # [R,G,B] tuple
38
39          # Arrays for further manipulation
40          self.pixels = np.reshape(self.original_data, (self.w * self.h, self.d))
41          self.kmeans = None
42          self.nColors = 0
43          self.mask = Mask(self)
44
45          # Convert to grayscale for analysis (cropping etc.)
46          self.grayscale = np.zeros(self.shapeId)
47          for i, row in enumerate(self.original_data):
48              for j, color in enumerate(row):
49                  r, g, b = tuple(color)
50                  # Uses luminosity method - better match with human perception
51                  self.grayscale[i][j] = 0.21 * r + 0.72 * g + 0.07 * b
52
53      def preprocess(self):
54          """Prepare image for the clustering."""
55          # Check if the picture has white background [sample 5x5], otherwise do not crop
56          offsetFromWhite = sqrt(sum(np.array([x * x for x in (1 - self.original_data[0:5, 0:5])]).flatten()))
57          if offsetFromWhite > 0.05:
58              return
59
60          # Cropping based on luminiscence derivative
61          # looks for the first jump from the sides and makes flags for generating mask
62          lightDifference = np.zeros(self.shapeId)
63          for r, row in enumerate(self.grayscale):
64              borderDetected = False
65              for c, col in enumerate(row):
66                  if c < (self.w - 1):
67                      diff = self.grayscale[r][c + 1][0] - self.grayscale[r][c][0]
68                      lightDifference[r, c] = diff
69
70                      # Border detection from the left side
71                      if abs(diff) > 0.05 and borderDetected == False:
72                          self.mask.boundaries[r, c] = 1
73                          borderDetected = True
74
75              # Border detection from the right side
76              c = self.w - 1
77              borderDetected = False
```

```
78                  while c > 0:
79                      diff = lightDifference[r, c]
80                      if abs(diff) > 0.05 and borderDetected == False:
81                          self.mask.boundaries[r, c] = 1
82                          borderDetected = True
83                      c -= 1
84
85              self.mask.genMatrix()
86
87          def clusterize(self, n):
88              """Cluster the image into [n] of RGB Clusters using Scikit KMeans class."""
89              self.nColors = n
90
91              # Apply cropping mask if exists
92              try:
93                  pixels = np.reshape(ma.masked_array(self.pixels, mask=self.mask.matrix), (self.w * self.h, self.d))
94              except:
95                  pixels = self.pixels
96
97              # Calculate the clusters
98              self.kmeans = KMeans(init='k-means++', n_clusters=n, random_state=0).fit(pixels)
99
100             return BeerColor(self.kmeans)
101
102         def quantizeImage(self):
103             """Plot the image using only clustered colors."""
104             codeBook = self.kmeans.cluster_centers_
105
106             # Determine to which cluster the pixel belongs
107             labels = self.kmeans.predict(self.pixels)
108
109             newImage = np.zeros((self.w, self.h, self.d))
110             label_idx = 0
111             for i in range(self.w):
112                 for j in range(self.h):
113                     newImage[i][j] = codeBook[labels[label_idx]]
114                     label_idx += 1
115             self.pixels = newImage
116
117         def showResults(self):
118             """Plot the original picture, processed picture and clustered colors."""
119             plt.figure(1)
120             plt.clf()
121
122             plt.subplot(2, 2, 1)
123             plt.title('Original')
124
125             plt.imshow(self.original_data)
126             plt.axis('scaled')
127
128             plt.subplot(2, 2, 2)
129             plt.title('Quantized')
130             plt.imshow(self.pixels)
131             plt.axis('scaled')
132
133             plt.subplot(2, 2, 3)
134             plt.title('Mask')
135             plt.imshow(self.mask.matrix)
136             plt.axis('scaled')
137
138             plt.subplot(2, 2, 4)
139             plt.title('Cluster colors')
140             for i, color in enumerate(self.kmeans.cluster_centers_):
141                 rectangleHeight = self.h / self.nColors
142                 rectangleWidth = rectangleHeight
143                 rectangle = plt.Rectangle((i * rectangleWidth, 0), rectangleWidth, rectangleHeight, fc=color)
144                 plt.gca().add_patch(rectangle)
145             plt.axis('scaled')
146             plt.show()
147
148
149 class Mask:
150
151     """Object to handle non-rectangular color-difference based cropping."""
152
153     def __init__(self, img):
154         self.w, self.h, self.d = self.shape = img.shape
155         self.boundaries = np.zeros(self.shape)
156         self.matrix = np.zeros(self.shape)
157
158     def genMatrix(self):
159         """ Generate Numpy matrix mask matrix based on the boundaries."""
160         for r, row in enumerate(self.boundaries):
161             # From the left
162             c = 0
163             while row[c][0] == 0 and c < self.w - 1:
```

```
                self.matrix[r][c] = (1, 1, 1)
                c += 1
            # From the right
            c = self.w - 1
            while row[c][0] == 0 and c > 0:
                self.matrix[r][c] = (1, 1, 1)
                c -= 1


class ColorPalette:

    """Object to create color palette from already processed beer labels."""

    def __init__(self):
        # self.nColors = nPaletteColors
        self.palette = dict()

    def build(self, beerColorsDict, beersList):
        """
        Generate color rating palette based on beer ratings.

        Palette colors correspond to those on the web colorPicker.
        """
        print 'Generating the color palette... '

        webPaletteRGB = [[254, 82, 9],
                         [251, 153, 2],
                         [247, 189, 1],
                         [255, 254, 53],
                         [209, 233, 51],
                         [102, 177, 49],
                         [2, 145, 205],
                         [9, 68, 253],
                         [63, 1, 164],
                         [134, 2, 172],
                         [168, 24, 75],
                         [254, 38, 18],
                         [255, 255, 255],
                         [0, 0, 0]]

        # Normalize
        webPaletteRGB = (np.array(webPaletteRGB, dtype=np.float64)/255).tolist()

        # Construct the Palette
        for i in range(len(webPaletteRGB)):
            paletteColor = self.palette[i] = dict()
            paletteColor['RGB'] = webPaletteRGB[i]
            paletteColor['RatingSum'] = 0
            paletteColor['Votes'] = 0
            paletteColor['Rating'] = 0

        progress = Progress(max=len(beerColorsDict), msg="Rating the  colors from palette... ")
        for bid, beerColor in beerColorsDict.iteritems():
            beer = getBeer(bid, beersList)
            if beer:
                for colorId, beerColorValue in enumerate(beerColor.colors):
                    closestPaletteColorId =  self.classifyColor(beerColorValue)

                    # Update classification flags
                    beerColor.colorPaletteFlags[colorId] = closestPaletteColorId

                    # Update color rating
                    self.palette[closestPaletteColorId]['RatingSum'] += beer.rating
                    self.palette[closestPaletteColorId]['Votes'] += 1
            else:
                print 'Not found bid ' + bid

            # Show the classified color.
            # plt.figure(1)
            # plt.title('Palette colors')
            # color = [beerColor.colors[colorId], self.palette[closestPaletteColorId]['RGB']]
            # for i in range(2):
            #     rectangleHeight = 20
            #     rectangleWidth = 20
            #     rectangle = plt.Rectangle((i * rectangleWidth, 0), rectangleWidth, rectangleHeight, fc=color[i])
            #     plt.gca().add_patch(rectangle)
            # plt.axis('scaled')
            # plt.show()

            progress.tick()

        for color in self.palette.values():
            if color['Votes'] != 0:
                color['Rating'] = color['RatingSum']/color['Votes']
```

```
250     def classifyColor(self, inputColor):
251         """Custom classification of colors to fit the palette."""
252         closestDistance = sqrt(3)
253         inputColorYUV = RGBtoYUV(inputColor)
254         for paletteColorId, paletteColor in self.palette.iteritems():
255             # Find maximum Euclidean distance of linear colorspace (YUV) values
256             paletteColorYUV = RGBtoYUV(paletteColor['RGB'])
257             distance = sqrt((paletteColorYUV[0] - inputColorYUV[0])**2 +
258                             (paletteColorYUV[1] - inputColorYUV[1])**2 +
259                             (paletteColorYUV[2] - inputColorYUV[2])**2)
260             if distance < closestDistance:
261                 closestDistance = distance
262                 closestPaletteColorId = paletteColorId
263 
264         return closestPaletteColorId
265 
266     def readFromFile(self, path):
267         """Get the palette from file instead of building it again."""
268         try:
269             paletteFile = open(path, 'rb')
270         except IOError:
271             paletteFile = open(path, 'wb')
272 
273         try:
274             f = paletteFile.read()
275             loadedColorPalette = jpickle.decode(f)
276         except:
277             print "Palette file corrupted."
278             return 0
279         paletteFile.close()
280 
281         self.palette = loadedColorPalette
282         return 1
283 
284     def genCSV(self):
285         """Save the csv file of the color palette."""
286         f = open('../data/colorPalette.csv', 'wt')
287         writer = csv.writer(f)
288         writer.writerow(('id', 'HEX', 'Rating', 'Votes'))
289         for id, item in self.palette.iteritems():
290             try:
291                 writer.writerow((id, rgbToHex(tuple(np.array(item['RGB'])*255)), item['Rating'], item['Votes']))
292             except:
293                 pass
294         f.close()
295 
296 
297 class BeerColor:
298 
299     """Object to save dominant colors of beer along with the color palette flags."""
300 
301     def __init__(self, kmeans):
302         self.colors = kmeans.cluster_centers_.tolist()  # array of RGB values
303         self.presence = [(float(sum(kmeans.labels_ == i)) / kmeans.labels_.size)
304                          for i in range(len(kmeans.cluster_centers_))]
305         self.colorPaletteFlags = [0] * len(self.colors)
306 
307 
308 class BeerColorsDict(dict):
309 
310     """Container to hold colors of each beer."""
311 
312     def __init__(self, *arg, **kw):
313         super(BeerColorsDict, self).__init__(*arg, **kw)
314 
315     def getColors(self):
316         """Return non-nested color RGB values for K-Means processing."""
317         return np.concatenate([x for x in [i.colors for i in self.values()]])
318 
319 
320 class Progress:
321 
322     """Print percentage of done work."""
323 
324     def __init__(self, max, msg="Processing ... ", freq=100):
325         """Construct the object, determine frequency of printing."""
326         self.max = max
327         self.msg = msg
328         self.printFrequency = max/freq    # Each percent
329         self.i = 0
330 
331     def tick(self):
332         """Call in every cycle."""
333         if (self.i % self.printFrequency) == 0:
334             print self.msg + str(100*self.i/self.max) + "%"
335         if self.i == self.max:
```

```
            print "Done."
        self.i += 1


def download(beersList, imgPath, fileList):
    """Get the beer labels based on the Untapped beer list."""
    progress = Progress(max=len(beersList), msg="Downloading images... ")
    for hashId, beer in beersList.iteritems():
        url = beer.label
        if url and (url != 'https://d1c8v1qci5en44.cloudfront.net/site/assets/images/temp/badge-beer-default.png'):
            fileType = url.split("/")[-1].split(".")[-1]
            filePath = imgPath + str(beer.bid) + '.' + fileType
            fileName = str(beer.bid) + '.' + fileType
            if fileName not in fileList:
                r = requests.get(url, stream=True)
                if r.status_code == 200:
                    with open(filePath, 'wb') as f:
                        for chunk in r.iter_content(1024):
                            f.write(chunk)
        progress.tick()


def getBeer(bid, beersList):
    """Search for beer based on bid."""
    for beer in beersList.values():
        if beer.bid == bid:
            return beer
    return None


def RGBtoYUV(RGB):
    """Convert array of RGB values to YUV colorspace."""
    T = np.matrix('0.299 0.587 0.114;'
                  '-0.14713 -0.28886 0.436;'
                  '0.615 -0.51499 -0.10001')
    try:
        return T*np.transpose(np.matrix(RGB))
    except:
        print "Unable to convert RGB -> YUV."
        return 0

def rgbToHex(rgb):
    """Convert RGB values (in list) to its HEX equivalent."""
    return '#%02x%02x%02x' % rgb

# For database export
# pal = ColorPalette()
# pal.readFromFile('../data/colorPalette.json')
# pal.genCSV()
```

---

```
"""
Functions to load downloaded json data.

To do: Make this general.
"""

import jsonpickle as jpickle
import csv
import labels


def readUsers():
    """Load already processed users UntappdUser."""
    try:
        usersFile = open('../data/users.json', 'rb')
    except IOError:
        usersFile = open('../data/users.json', 'wb')

    try:
        f = usersFile.read()
        usersList = jpickle.decode(f)
    except:
        usersList = {}
    usersFile.close()
    return usersList


def readBeers():
    """Load already processed beers UntappdBeer."""
    try:
        beersFile = open('../data/beers.json', 'rb')
    except IOError:
        beersFile = open('../data/beers.json', 'wb')

    try:
        f = beersFile.read()
```

```
        beersList = jpickle.decode(f)
    except:
        beersList = {}
    beersFile.close()
    return beersList


def readBreweries():
    """Load already processed breweries UntappdBrewery."""
    try:
        breweriesFile = open('../data/breweries.json', 'rb')
    except IOError:
        breweriesFile = open('../data/breweries.json', 'wb')

    try:
        f = breweriesFile.read()
        breweryList = jpickle.decode(f)
    except:
        breweryList = {}
    breweriesFile.close()
    return breweryList


def readBreweryToBeers():
    """Load already processed breweries dictionary."""
    try:
        breweryToBeersFile = open('../data/breweryToBeers.json', 'rb')
    except IOError:
        breweryToBeersFile = open('../data/breweryToBeers.json', 'wb')

    try:
        f = breweryToBeersFile.read()
        breweryToBeers = jpickle.decode(f)
    except:
        breweryToBeers = {}
    breweryToBeersFile.close()
    return breweryToBeers


def readDataPoints():
    """Load dataPoints."""
    try:
        dataPointsFile = open('../data/dataPoints.json', 'rb')
    except IOError:
        dataPointsFile = open('../data/dataPoints.json', 'wb')

    try:
        f = dataPointsFile.read()
        dataPoints = jpickle.decode(f).points
    except:
        dataPoints = []
    dataPointsFile.close()

    return dataPoints


def readBeerStyles():
    """Load most rated beer styles."""
    styles = []
    with open('../data/styles.csv') as stylesFile:
        reader = csv.DictReader(stylesFile)
        for row in reader:
            if row['style'] not in styles:
                styles.append(row['style'])
    return styles


def readBeerColors():
    """Load the dominant label colors."""
    try:
        beerColorsFile = open('../data/beerColors.json', 'rb')
    except IOError:
        beerColorsFile = open('../data/beerColors.json', 'wb')

    try:
        f = beerColorsFile.read()
        beerColorsDict = jpickle.decode(f)
    except:
        beerColorsDict = labels.BeerColorsDict()
    beerColorsFile.close()
    return beerColorsDict
```

---

```
"""
Single-purpose script for easy monitoring of data quantity.

Load each json data file, find its size and generate
```

```
5   a plot for presentation.
6   """
7
8   import fileReader as files
9   import matplotlib.pyplot as plt
10  import os
11  import numpy as np
12
13  # Load files
14  print "Loading beers..."
15  beersList = files.readBeers()
16  print "Loading users..."
17  usersList = files.readUsers()
18  print "Loading breweries..."
19  breweriesList = files.readBreweries()
20
21  # Path for saving the images
22  path = "../data/labels/"
23  fileList = os.listdir(path)
24
25  # Data gathering
26  labels = ('Beers', 'Reviews', 'Users', 'Breweries', 'Labels')
27  index = np.arange(len(labels))
28  quantities = (len(beersList), sum([len(x.ratings) for x in usersList.values()]),
29                len(usersList),  len(breweriesList), len(fileList))
30
31  # Plot the quantities
32  plt.figure(1)
33  bar_width = 0.35
34  opacity = 0.7
35
36  bars = plt.bar(index, quantities,
37                 bar_width,
38                 alpha=opacity,
39                 color='g')
40
41  plt.xticks(index + bar_width/2, labels)
42  plt.title('Amount of mined data')
43  plt.ylabel('log(N)')
44  plt.yscale('log')
45  plt.grid()
46
47  def autoLabel(bars):
48      # attach some text labels
49      for rect in bars:
50          height = rect.get_height()
51          plt.text(rect.get_x()+rect.get_width()/2., 1.05*height, '%d'%int(height),
52                  ha='center', va='bottom')
53
54  autoLabel(bars)
55  plt.show()
```

# APPENDIX B
## AUTOMATIC GENERATION OF DOCUMENTATION

Demontration using epydoc:

```
epydoc --pdf -o /home/fnielsen/tmp/epydoc/ --name RBBase wikipedia/api.py
```

This example does not use `brede_str_nmf` but another more well-documented module called `api.py` that are used to download material from Wikipedia.