Yarnatron: Predicting Unwanted Attributes of Yarn using Text Mining, Document Clustering, and Classification Models on Web-Mined Social Data

CSC 478 Final Project

Cara Bowen-Goldberg and David Liao

Introduction

Ravelry.com is a website for fiber artists, especially those who knit and crochet. With over 6 million user accounts, many of them active monthly users, it is the largest and most active topic-specific social network.

The website is maintained by a single developer, who maintains the site's database and has made an API available for app developers. Included in the data are about over 100,000 yarns, and up to two dozen different searchable attributes for each, such as weight (thickness), construction (plied, chained, etc), fiber makeup (wool, cotton, etc), MSRP, popularity (how often they have been used) and star ratings.

However, there remain some attributes of yarn that are not currently available on the site. Most of these attributes describe a negative characteristic of yarn that knitters typically wish to avoid, or at least be forewarned of, and it would be of great utility to Ravelry's users if they could filter out yarns with such attributes when seeking the perfect yarn for their next project. We suspect these attributes aren't currently available as filters on Ravelry's yarn search because the website stays running with the help of advertisers, mainly yarn companies, and it wouldn't be good business for Ravelry to have negative sentiment in their database of yarns.

Goals

We polled members of the Ravelry community to find out which negative attributes of yarn they'd most like to avoid. The results of our poll revealed the two most unwanted yarn attributes to be pilling and splitting. Knitters generally consider themselves as "product knitters" -- those who knit for the sake of the outcome (the finished object) -- or as "process knitters" -- those who knit for the fun and meditative benefits of the activity itself. Likewise, pilling is a 'product' problem, while splitting is a 'process' problem. Pilling is the fuzzing up of little balls of wool and fibers that occurs over time with wear, especially in high-friction areas like the armpits of sweaters. Splitting occurs with plied yarns when the yarn is too loosely plied, or the fibers are too long, and the strands/plies of the yarn lose their twist or are easily separated by pointy knitting needles. Splitting is even more problematic for crocheters, who use dull hooks.

Overview

Each yarn in Ravelry's database has a page of text comments, similar to reviews. We seek to gain information from these comments and use that information, in addition to the features already available in Ravelry's database, to predict how likely a yarn is to pill or split, relative to the set of all yarns that have comments. After building our prediction model, we will make a basic app that lets a user search yarns by a small set of important parameters, lets the user choose whether they want to sort by a pilliness score or by a splittiness score, and then displays the search results along with a 'score' and a link back to the yarn's webpage on Ravelry.

NOTE: Ravelry is only accessible with a username and password. We created an account for this project/report and its reader: username: yarnatron | password: scrapeit

Methodology

Narrowing Scope

First, we chose a relevant subset of yarns to analyze. We chose yarns that are not discontinued, that have comments, and that are not 100% acrylic. Acrylic yarn is basically plastic, is considered the McDonalds of yarn, and would not be of interest to the discerning target audience of our app. However, acrylic can sometimes be helpful as part of a blend, so yarns that are less than 100% acrylic were included in our data.

Data Collection

Next, we scraped data from the Ravelry API and from Ravelry.com. It was most efficient to retrieve attributes available in the database from the API but comments and various popularity measures were available only from the website itself. The scraped data was munged and stored in a CSV file. We considered using a database, but due to the relatively small size of the data it was actually faster to store and save data locally.

Data Munging

Once we had retrieved all the data, it was extensively munged and normalized. There were many messy entries or missing values in the database which were relatively easy to impute. For example, many yarns were missing a Boolean value for the "Machine Washable" attribute, but this information could sometimes be found by searching the "Yarn Notes" (like a short optional product description) or scanning the comments for relevant keywords. A similar approach helped us fill a handful of other missing values. We also used OpenRefine to turn the 'texture' attribute into several dummy variables containing information about the texture and construction of yarn (eg, smooth, plied, lofty). Details about the munging can be found in the Appendix Table of Contents sections 1f and 1g, and in the comments of the yarn_preprocessing.py (appendix item 1g).

Text Mining and Document Clustering

After scraping comments from the web, we started with a set of 33,752 comment rows for 3,743 yarns. Each comment was retrieved into a separate row, and indexed by its yarn ID. Some exploratory testing revealed that it was best to come up with our own unique list of stop words customized for the yarn comments dataset. We also explored keeping 2 special characters, '!', and '?' but decided they were not adding value as identifying sentiment. The Snowball Stemmer was also preferred over the popular Porter Stemmer. This came from analyzing simple word frequencies that resulted from exploration passes at simple clustering using K Means.

We also explored both processing unigrams and bigrams and examined word frequencies of the dictionaries produced but decided that unigrams were already producing enough features for clustering. Bigrams explodes the number of features to use exponentially and was not required in the first round of clustering. Round 1 clustering on unigrams returns a word frequency dictionary of 20K unique terms. Using the 'CountVectorizer' function on the raw text cleaned yields 3418 features on the 33572 rows of comments. This is with applying a filter of Minimum Doc Frequency=10. This allows a creation of a smaller word frequency dictionary for easier analysis. We use the 3418 features to create a term freq inverse doc frequency, TFIDF, array to start our Round 1 clustering. After several attempts of examining clustering sizes and results we decide that K=8 appeared optimal in Round 1.

Before clustering began, we had suspected there would be one cluster that is likely much larger than the rest as it would be a 'junk' cluster where the K Means algorithm throws in comments that are not providing alignment with yarn attributes we seek to find. This is indeed what happens in Round 1 clustering where there is always 1 cluster with about 20K rows out of 33572 total rows, as we can see in Figure 1.

```
print km_clusters.shape
cluster_tab = pd.crosstab(index=km_clusters,  # Make a crosstab
                          columns="count")      # Name the count column

print 'KMeans 8 Cluster Results', '\n', cluster_tab
#kmeans.cluster_centers_
```

```
(33752L,)
KMeans 8 Cluster Results
col_0  count
row_0
0       1874
1       2377
2       1708
3       1318
4       1761
5       1709
6       1479
7      21526
```

Figure 1 Distribution after first clustering attempt with K=8

We decide to remove this junk cluster and re-run clustering again without these comments that are of no value. This leaves about 12K rows left for Round 2 clustering. We get about 1800-1900 features from CountVectorizer in Round 2. This is a much more manageable sized dataset now for detailed clustering where we hope to mine the comments into distinct attributes on yarn that can later be used as extra features in classification with other yarn data.

Round 2 Clustering – Varying K & Measuring Cluster Quality

We iterate through values and K and experiment with Silhouette Coefficient and other calculated clustering indices like Dunn Index and Davies Bouldin (found from a custom github library: https://github.com/jqmviegas/jqm_cvi). The Silhouette values are best when values are closer to 1. This measure combines both intra-cluster and inter-cluster distances. We plot the Silhouette values as area charts and draw a dotted vertical line showing the Average Silhouette value for each round of K tested to show the number of points on either side of the Silhouette Average. We vary K from 3 to 10 clusters. We decide on using the Davies Bouldin clustering index measure as a second clustering criteria which seeks to minimize values to find good clusters.

In Figure 2 below, we see that the Average Silhouette Coefficient finds its maximum at K=8 with a value of 0.019579 and then the Silhouette Average starts to decrease at K=9 and K =10.

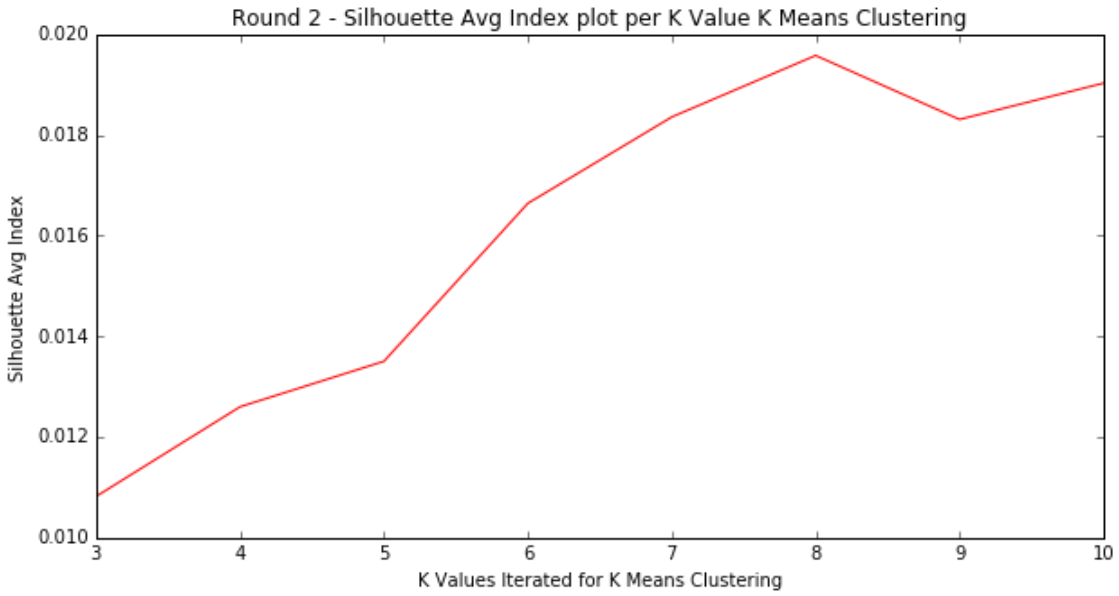

Figure 2 Average Silhouette index for different K

In Figure 3 below, we see that the Davies Bouldin finds it's first minimum at K=8 with a value of 6.69803 and then it stays relatively flat and does not decrease much in relative value for K=9 and K =10. The information from these two graphs and examination of the Silhouette Plots for various K leads us to conclude that K=8 is the optimal number of clusters for Round 2. A low K value like K =3 or 4 shows a larger number of silhouette coefficient scores that are negative and indicates less than ideal clustering quality. This can be seen in Appendix 2a where the python notebook with full clustering results can be seen. When K is 9 or 10, there are small clusters of sizes of only 400-500 or 750 that look a bit small relative to the total row size of about 12K rows. We are believe that a minimum number of comments > 1000 is likely required to indicate a real yarn attribute has been clustered together correctly. We apply the findings of word frequencies from Round 2 on unigrams to corroborate this and decide on K=8 is again the optimal clustering in Round 2.

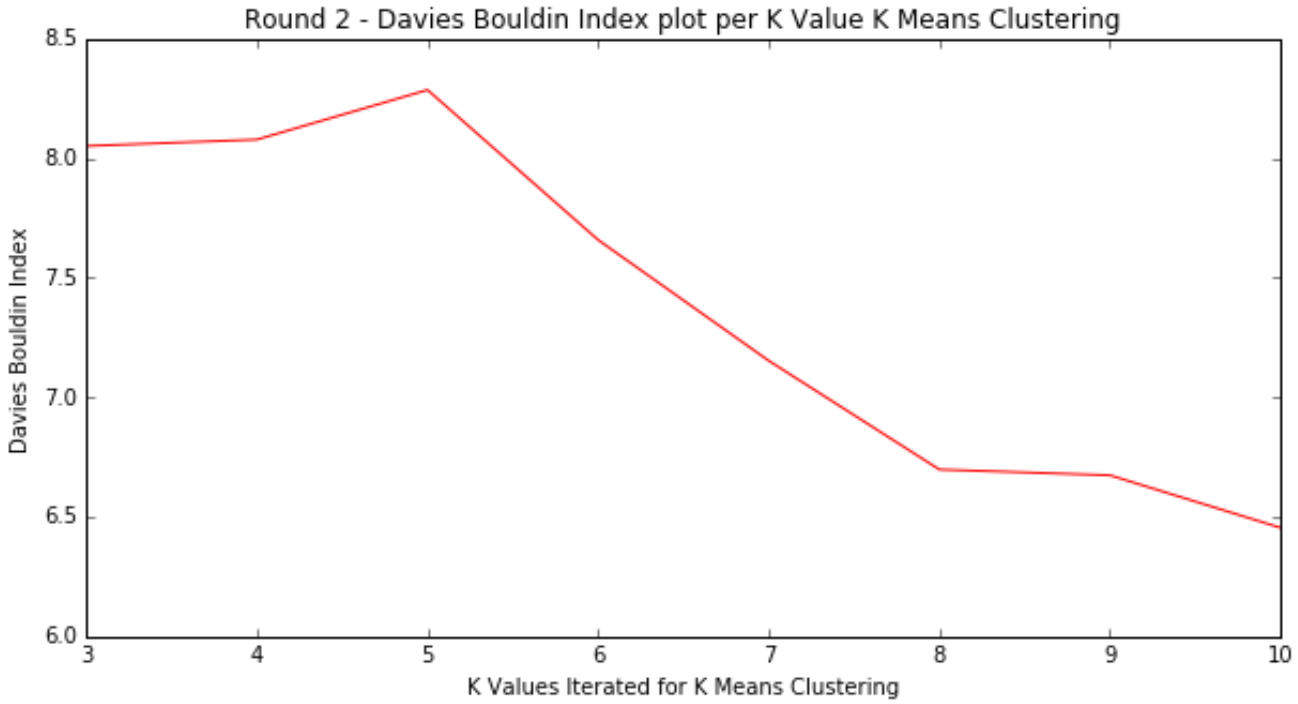

Figure 3 Davies Bouldin Index for different K

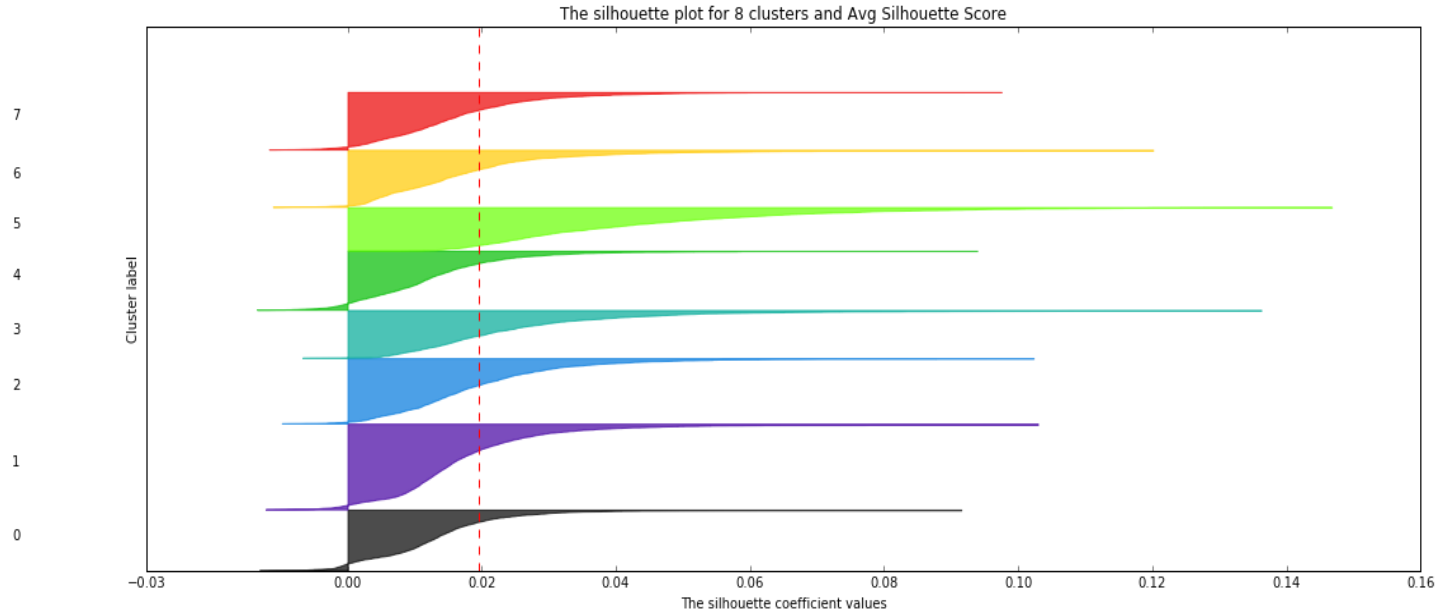

Figure 4 Silhouette Plots for final clustering (round 2, K=8)

As a final clustering check, since unsupervised learning has subjective qualities in the entire process of deciding how many clusters are optimally found in the data with the algorithm used, we print out the Top 20 and Top 50 word terms of each 8 clusters to verify if each cluster is indeed grouping together user comments that describe a common yarn attribute of interest. Because this is Round 2 clustering and the noise in the 'junk' cluster of Round 1 has been removed, the top word frequency in each cluster is actually of great interest and value as a domain knowledge verification of clustering results.

Indeed we found that in the 8 clusters, the top word often has word frequency of close to 90-100% in that cluster. We find that Yarn material, Wool vs Cotton, has separated into 2 distinct clusters. There's also a cluster with the top word = 'babi', which is the word baby after stemming is applied. There's also a cluster with 'split' as the top word term with 100% frequency and whether a yarn is 'splitty' or not is of great interest when choosing a yarn. There's also a cluster where 'knot' is the top word.

Transforming Cluster Assignments per Comment to a Per-Yarn Attribute

We feel confident that Round 2 clustering is accurate enough so that these cluster labels can be used as extra yarn features to be used to help classify each unique yarn id in the supervised learning phase of this project.

Because a yarn can have as many as 200 comments (though most yarns have much less), we decided to take the comment assignments per cluster as a sort of "vote" for each yarn's attributes. So for we created 8 new attributes for each yarn, one for each cluster, the value of which is a membership percentage per cluster per yarn. In other words, if yarn X has 10 comments, with 3 of those comments in Cluster One, 1 of those comments in Cluster Four, and 6 of those comments in Cluster Eight, then yarn X will have a Cluster One value of 0.3, a Cluster Four value of 0.1, a Cluster Eight value of 0.6, and a value of 0 for Cluster Two, Cluster Three, Cluster Five, Cluster Six, and Cluster Seven. This approach not only retains the most information but also provides a normalized continuous 0-1 vector for the modeling in our next steps.

Yarn Classifier Model Testing and Evaluation

Training Data Distribution

Because the yarn data has been hand scraped from a web site, and we had to hand label data rows for training, we have a small dataset for training relative to the final dataset yarn we will predict on. We know that we have an unbalanced training data set for 230 rows total as seen below:

- Training data for yarn that is 'Pilly' = 153 / (153 + 77) = 66.5%
- Training data for yarn that is 'Not Pilly' = 77 / (153 + 77) = 33.5%

This tells us that we are more interested in measures like Precision, Recall and False Positive Rates from the confusion matrix.

Testing Predictions on All Yarns without 'Ground Truth'

We do not have any kind of 'ground truth' label for entire dataset of 3657 rows of unique yarn ids to compare our prediction results for each model, so we compute the probabilities on each prediction when a fitted model is used to predict on the entire yarn dataset. A cutoff probability, like >= 0.70, is then used to estimate how many rows were correctly predicting a class label like 'pilly'. We expect that the fitted model will predict a relatively low percentage of the entire 3657 unique yarn ids would have any attribute like 'pilly'. It would only make sense for a low percentage of all yarns to have any combination of attributes.

Cross-Validation, Stratified K Fold vs Training & Test with 35% Test Holdout

With such a small training set relative to the total yarn dataset we want to predict on, Cross Validation did not perform as well as using a 35% testing holdout sample and saving 65% to train on. This is due to the smallness of our training set

and lack of ground truth labels on the larger 3657 unique yarn ids. Because the Pilly vs Not Pilly dataset was uneven, we did find Stratified K Fold testing, which preserves original distribution frequency of class labels in each fold, to be better than traditional K Fold testing.

But using Optimized Log Regression parameters of (penalty=l1, C=10) did not always result in better performance for precision, recall and AUC. Log Regression with grid search optimized parameters only yielded improvements in these metrics when the complete training data fitted model is applied on the entire yarn dataset. In cross validation or stratified k-fold, optimized Log Regression parameters fared worse in Recall and was no better in Precision or AUC.

Logistic Regression vs Naïve Bayes, LDA and Decision Trees

We spent considerably more effort on testing Logistic Regression as we have a binary outcome and we were immediately suspicious of any classifier that performs "too well" immediately on a small training set like we have. As seen in the summary of performance metrics in Appendix item 5, LDA and Decision Trees and Random Forests all overfitted right away to the training data with high scores for precision and recall but cross validation immediately brings those scores back to reality.

We also did not want to see lopsided results between precision and recall and this is what Naïve Bayes exhibited right away. If the final classifier is to accurately predict if a yarn will have a 'pilly' or 'splitty' attribute, we need a balance between precision and recall.

Amongst this group of classifiers, only Logistic Regression was deemed worthy enough to try to use it to predict values on the entire yarn dataset. As there are no Y prediction labels to use as ground truth for entire yarn dataset, so we compute probabilities and use a cutoff probability value to estimate how many predictions were correct for the 'Pilly' label 1. We want to use a stricter definition on probability cutoff to ensure that a yarn id was predicted to have an attribute class like 'pilly'. We expect relatively low %s when predicting on entire yarn dataset (3657 rows) as only a minority of yarns will truly possess any particular attribute like 'pilly' or 'scratchy'. In the Appendix ### excel table summarizing performance metrics, the column named "Est. Prediction Accuracy All Yarns" is the computed estimate of Accuracy when the training fit model is used to make predictions on the entire 3657 yarn id dataset.

Logistic Regression gave consistently modest performance metrics that remained stable through Cross Validation and Stratified K Fold, which we used as a replacement for Boot Strap Sampling since Boot Strap functions were no longer available from Scikit Learn libraries.

We used Grid Search to optimize parameters on Logistic Regression and also used it as an attempt see how many features it would choose and which ones. We found that at least 100 features were needed to not have lopsided performance metrics and since we did not have any performance issues with our dataset of only 3567 rows, there did not seem a need to reduce 151 features down to 100. But the optimized Logistic Regression parameters found of C=10 and penalty ='l1' did make an improvement when the Log Regression model was fitted using the entire training dataset.

Support Vector Machine

We initially had high hopes that SVM would work well on a small training set that was imbalanced. We used grid search on three kernel types (linear, rbf and polynomial) and loop through soft/hard margin C and gamma values but we end up with an overfitted model on Training data as the Precision vs Recall was very lopsided in favor of Precision and with 0 False Positives found, was further confirmation that SVM was not a balanced model for this yarn dataset. We also tried using SVM to make predictions on the entire yarn dataset and the differences between SVM and Log Regression in column "Est. Prediction Accuracy All Yarns" tends to confirm that SVM was not an appropriate classifier on the yarn data.

Rocchio Method as Alternative Classifier

We decide to explore the use of the Rocchio method as a similarity classifier since we have a small training set that was hand labelled where we are quite sure the yarns in the training set was positively 'pilly' or 'not pilly' and 'splitty' or 'not splitty'. We can easily build a prototype vector that we would have high confidence in and thus be able to compare each unique yarn id to the prototype vector and calculate a Cosine Similarity to see how good a match any yarn is to a prototype vector for 'pilly' or 'splitty' or Not.

Our training results in Figure 5 show modest accuracy scores using our small training set with hand label Y class labels for comparison.

Label	Accuracy- Training
Pilly	0.718954
Not Pilly	0.623377
Splitty	0.69403
Not Splitty	0.729323

Figure 5 Accuracy of Rocchio method on Training data

As there is no 'ground truth' to compare the computed Cosine Similarities for each yarn and determine accuracy of prediction the traditional way, we looked at the distribution of computed cosine values and use the 75% IQR value as a cutoff to estimate that any yarn with a cosine value > 75% IQR is likely to be a match for the prototype vector compared to. This matches our domain knowledge that there is no yarn attribute, like 'pilly' or 'splitty', that in reality would exceed 25% of all the yarns listed on the Ravelry website.

Below in Figure 6 is our computed True Positives when we apply the Rocchio method on all 3657 unique Yarn Ids and using the cosine 75% IQR value cutoff to rigorously be sure the Rocchio vectors found a real match of a yarn with a class label.

Label	True Positives on 3657 Yarn Ids
Pilly	0.1586
Splitty	0.250752

Figure 6 True Positives of Rocchio Method on All Data

These results are in line with the domain knowledge developed when we computed word frequency dictionaries to help identify clusters found using yarn comments. Spot checking of the Rocchio method results with yarn ids identified as 'pilly' or 'splitty' gives us confidence that Rocchio method of using prototype vectors is the best classifier for our yarn dataset with our hand created training labels that is rather small compared to total dataset of 3657 unique yarn ids.

Future Improvements on Text Clustering

We have chiefly used the Bag of Words approach to process our short yarn comments and converted the word terms to a term frequency/inverse document frequency (TFIDF) format as a way to convert text into numerical features for data mining. Though we looked at bigrams as word frequency to understand the data, we did not have time to try TFIDF processing of bigrams. Turning bigrams into numeric features causes an explosion of column features and techniques like PCA would likely be required to bring the number of features down so the overall resulting dataset is more manageable for clustering on a desktop PC's limited resources of CPU and RAM.

In addition to bigrams, using Parts of Speech tagging would also create more sophisticated features to be available. This would also require exploration of new rules to implement as some POS tags would likely need to be dropped and some kept for different types of analysis. Sentiment analysis is different than purely document classification and what we have done is chiefly grouped comments into yarn attributes being discussed in the comments and is similar to document

classification. It would be interesting to be able to rate overall quality of a yarn based on sentiment classification on user comments and maybe even find a relationship between yarn quality from text comments to a yarn's list price.

Another example of feature detection is Word Sense Disambiguation where key target words are examined for word(s) immediately to the left and right of the key target word. Thus neighboring word contexts become features to be used in clustering or classification.

Support Vector Machine and Singular Value Decomposition algorithms are now commonly used in large text mining software applications. The SAS Enterprise Miner module for text data mining uses SVD exclusively to generate features from text to be numerical inputs for supervised and unsupervised learning. SVM can handle a large number of features as inputs and is often a preferred algorithm for sparse datasets. It would be interesting future work to use SVM and SVD as a base algorithms for clustering or classification on text data.

Future Improvements on Yarn Classification

There were many approaches we did not have an opportunity to try and insights that came to us in hindsight as we progressed through our classification attempts. Probably the most helpful thing would be to do a stratified random sample of yarn before labeling, which took into account as many attributes as possible so that the labeled data would be more valuable as such. Manual labels were very time consuming but we would probably improve our results if we were able to label at least 10% of the ~3500 yarns for each target attribute of pilly and splitty (in other words, we would need to generate at least 700 manual labels instead of the ~400 we generated this time). And it would be best if those labels themselves were more balanced as well.

We might also go back and iteratively refine our clusterings based on our classification attempts. We were satisfied using just unigrams at the document clustering stage, but going back and creating a new clustering that uses bigrams might give us more information as to sentiment. Basically, many yarns will have the word "pill" or "split" in almost all of their comments because users are having a back and forth about whether or not it pilled for them, and we may have understood our initial clustering to be indicating that all of these yarns *will* pill, when actually the clustering is just indicating the main topic of discussion.

Given a larger labeled training set, we might also have more success with feature selection. We did attempt many methods of feature selection, but each one tended to overfit the training data. Still, it seems sensible that a different subset of attributes would be best at predicting each of the target variables "pilly" and "splitty". We could also explore an ensemble of classifiers which would probably have a better balance of overfit/underfit than any one classifier alone. We could also try using the information gained by clustering as a 'majority vote' instead of as a member percentage. We did try document clustering on merged comment sets (e.g. treating all of one yarn's comments as a single document) and didn't see any noticeable differences, but this might change if we incorporate bigrams.

A better classifier and resulting feature selection might improve Rocchio classification or lend itself to good results using other similarity measures. Alternately, with enough training data, we might feel confident enough to use the predicted probabilities output from a classification model as the 'score' instead of using RoccHio vector similarity

All that said, upon running our app our domain expert does find the results and scoring method implemented by the app to be excellent and giving results in line with what she knows about yarn. So we do consider our outcome to be successful and are excited by the possibilities of further refining it, and of trying the same approach on other unwanted features of yarn such as shedding, felting, knotting, and being scratchy/itchy against one's skin.

Welcome to Yarnatron!

After web scraping, data munging, text mining, document clustering, and classification modeling, we exported the cosine similarity distances of each yarn for the "pilly" Rocchio prototype and the "splitty" Rocchio protoype. We cleaned up our other yarn statistics a bit more, keeping only the columns we need to process a user's query and return an output. We also normalized each vector of cosine similarity distances to a 0-1 min-max scale (each distance vector separately normalized, each over all yarns).

Yarnatron comes from a single .py file and a single .csv file.

Yarnatron.py imports *pandas*, *easygui*, *tabulate*, and *sys*. Yarnatron contains four custom functions.

First, we read in 'yarn_search.csv', which will be our search space and also contains the data we need for display and output.

Then we create three global variables: a list of fiber names (*fiber_names*), a list of yarn weights (*yarn_weights*), and a string (*title*) to display our program name.

Next we define the four custom functions.

1. ***get_search_params(****title****)*** takes a string and returns a single list of 6 user inputs.
   The function calls up six *easygui* input boxes and saves the input from each; the user can choose not to fill any of the boxes and will simply be passed on to the next. The global variables *fiber_names* and *yarn_weights* are used by this function to display selection choices.

2. ***return_results(****search_df, inputs****)*** takes the search space dataframe and the input list returned by *get_search_params(),* and returns the input list as-is, as well as a dataframe of results.
   a. First it expands the list of inputs back into separate variables.
   b. Then it creates a copy of the search dataframe. The copy becomes the results dataframe and it will be modified by the user's search parameters.
   c. For each input, the function checks if the user set a filter or not. If there's no filter, the function moves on to the next input. If there is a filter, the function modifies the results dataframe in place.
   d. After all inputs have been checked, the function checks to make sure the results dataframe is not empty. If it is empty, the user is notified that their search returned no results, and is given the option to search again. If they choose to search again, the function calls *one_run()* **[SEE CUSTOM FUNCTION #4 BELOW]** Otherwise, *exit(*0*)* is called from the *sys* module and the user exits the program.
   e. If the dataframe passes validation, the function puts together a paragraph in the form of a string that will display the user's query.
   f. Then the *textbox()* of *easygui* is called to display the paragraph showing the user's query.
   g. The expanded input variables are packed back up into a list before being returned along with the results dataframe.

3. ***print_results(****results_df, inputs****)*** takes the results_df and input list returned by *return_results(),* and returns nothing.
   a. First it expands the list of inputs back into separate variables.
   b. Then it checks the length of results. If there are more than 50 results, only the top 50 will be displayed.
   c. The stem string needed for displaying the yarn URL is stored as a local variable
   d. An empty list of lists (*table*) is initialized to store the results

e. The function checks whether the user chose “less pilly” or “less splitty” as the sort parameter. It then sorts the results and assigns the scoring choice, and stores a score-customized message in a local string variable
f. Then the function loops over the dataframe (either the top 50 rows or, if the results are less than 50, the whole dataframe) and extracts the data from the results dataframe by its index location. Floats are rounded to 2 decimal places. The list of fibers is stripped of brackets for prettier display. The yarn permalink is appended to the yarn_url stem to get the full string of the yarn’s URL on Ravelry.com. The row is stored as a list and appended to the list of lists.
g. Headers are assigned to a local list variable.
h. The *tabulate* module is called to fancy-format the list of lists as a results table.
i. *textbox()* is called from the *easygui* module to display the results table.
j. The user is given the option to search again, or to exit. If the user decides to search again, *one_run()* is called **[SEE CUSTOM FUNCTION #4 BELOW].** Otherwise, *exit(*0*)* is called from the *sys* module and the user exits the program.

4. ***one_run(****results_df****)*** takes the results dataframe and returns nothing.
    a. ***return_results****(search_df, get_search_params(*title*))* is called and the results are stored as *results_df, inputs* **[SEE CUSTOM FUNCTIONS #1 and #2 ABOVE]**
    b. ***print_results(****results_df, inputs*)* is called **[SEE CUSTOM FUNCTION #3 ABOVE]**

After the functions are defined, the *msgbox()* module of *easygui* is called to welcome the user and initiate the program.

Then, *one_run(*search_df) is called.

That’s it!

Appendix Table of Contents

1. /appendix/data scraping and processing/
    a. yarn_scrape.py
        - various helper functions for scraping all data from the Ravelry API and from ravelry.com
    b. yarn_scrape_implementation.py
        - script to run functions from yarn_scrape.py and save all retrieved data to CSV
    c. yarn_stats_df.csv **3,743 rows of unique yarns**
        - output of non-comment data from yarn_scrape_implementation.py
    d. comments.csv
        - output of scraped comments, indexed by yarn_id (one row per yarn, each column is a comment)
    e. comments_with_yarn_ids.csv
        - output of scraped comments, indexed by yarn_id (one row per comment, multiple rows per yarn_id)
            33,753 rows of comments
    f. yarn_stats_manually_imputed.csv
        - yarn_stats_df.csv after semi-manual data munging of some missing values and fuzzy matching of messy database entries; OpenRefine was used for this task.
        - Manual labels were created with the help of OpenRefine and Excel by viewing yarn statistics and their comments side by side and using domain expertise as to the generally previously known characteristics of yarn content and construction (e.g., tightly twisted many-ply wool-nylon blend yarn does not generally pill, whereas single-ply, worsted weight, super soft, loosely spun, 100% merino is known to pill badly).
    g. yarn_preprocessing.py
        - script for all preprocessing of yarn data before classification modeling, including normalization of numeric variables and factorizing (getting dummy variables) of categorical variables, as well as reducing data by removing columns with very low frequency of values and dropping a small subset of rows missing most of their data (due to erroneous database entries not previously discovered in munging process).
        - Cluster assignments from text mining and kmeans clustering were merged with the rest of the data in this step
    h. all_reduced_reg_norm.csv, pilly_rows.csv, not_pilly_rows.csv, split_rows.csv, not_splitty_rows.csv
        - output data from yarn_preprocessing.py
    i. preprocessing_search_data.py
        - preparing data for app (search space, ranking info and data needed to print output)
    j. yarn_search.csv
        - output data from preprocessing_search_data.py, used by and packaged with app
2. /appendix/modeling /CSC478_FinalProj_KM_txt_clustering
    a. CSC478_FinalProj_KM_unigram_Clustering.ipynb (also in HTML format)
        - Jupyter notebook containing all code for K Means text clustering of unigrams (for both round 1 and 2 clustering).
        - Notebook references:
            - comments_with_yarn_ids.csv
            - cluster_helper_3.py (helper function for custom stop words)
            - jqmcvi_base.py (custom functions to calculate clustering quality indices,
                - source: https://github.com/jqmviegas/jqm_cvi
        - Notebook outputs the following images, all of which areincluded in project report:
            - Davies Bouldin Index vs K_3to10.png
            - Silhouette Avg vs K_3to10.png

- Silhouette Chart K-8.png

3. /appendix/modeling/CSC478_FinalProj_ModelTestingEval1
    a. CSC478-FinalProj-ModelTestingEval1.ipynb (also in HTML format)
        - Jupyter notebook containing all code for testing different classification models
    b. Rocchio1-Pilly.ipynb (also in HTML format)
    c. Rocchio2-Splitty.ipynb (also in HTML format)
    d. These notebooks reference the following data files (as described in part 1 of this Appendix):
        - all_reduced_reg_norm.csv, pilly_rows.csv, not_pilly_rows.csv, splitty_rows.csv, not_splitty_rows.csv
    e. These notebooks output splitty_distances.csv and pilly_distances.csv, which are used to build the app (see appendix items 1.i and 1.j)
4. /appendix/Sample Data and Description.xlsx
    a. The first tab of the spreadsheet contains a sample of the data used by the Yarnatron app, The second tab contains a Data Dictionary of the same data.
5. /appendix/Yarn Classifier Model Eval Metrics Summary.xlsx
    a. Summary of Performance metrics
6. /appendix/Sample Runs.docx
    a. Detailed description of app with screenshots
7. /appendix/yarnatron.zip **THE APPLICATION EXECUTABLE FILE**
    a. Compressed folder containing application file 'yarnatron.exe'.
    b. The only file we specifically asked the app to require on is packaged with the app:
        - /appendix/yarnatron/yarn_files/yarn_search.csv
8. /appendix/README.txt
    a. Instructions for running Yarnatron, with some notes on the nature of a py2exe package
9. /appendix/pyfiles used to build app/ The .py files from which yarnatron.exe was created
    a. yarnatron.py
    b. setup.py
    c. yarn_search.csv