Capstone Project

Hannah Moran

December 18, 2018

I. Definition

Project Overview

We've all read statistics about the ever-increasing rate at which we generate data. Much of this data is structured and therefore relatively easy to analyze, but what about the data that is unstructured, particularly natural language data? Analyzing natural language data, especially data in the form of online reviews, continues to be an important part of how we acquire information and make decisions on a daily basis. Humans are good at doing this - we are typically pretty successful in extracting information from natural language data produced by another human. Many sites are available to aggregate this information for us, but as more and more of it accumulates, our ability to read and understand it doesn't scale.

To that end, many natural language processing (NLP) techniques have been developed. The objective of this project was to find ways of applying a couple of NLP techniques to extract the distinct topics present in Amazon reviews. Specifically, I am using a modeling approach known as Latent Dirichlet Allocation (LDA) along with some sentiment analysis tools.

The idea for this project came from my personal experience as a Yelp user. When I use Yelp, I'm personally more interested in information about the quality of the food, vs. service. I started to notice that many restaurants where I live had reviews with a relatively low star rating (3 or fewer stars) with the only content in the review being complaints about service (waiting in line, slow service, allegedly rude servers, etc.). For restaurants in the 3.5-4 star range, I started to wonder how much of this was driven by low ratings that were exclusively related to service, and wished that I could see separate star ratings for service and food, and this was the initial inspiration for the project. As soon as I did a little research I discovered that someone else had already worked on this exact problem, and done so very successfully - Jack Linshi won Yelp's own data challenge in 2014 by applying LDA to the dataset they provided for the challenge[1]. His innovation was to incorporate some sentiment analysis into the topic modeling by inserting "codewords" according to the sentiment assigned to individual words present in the reviews.

[1] Jack Linshi. 2014. Personalizing Yelp Star Ratings: a Semantic Topic Modeling Approach. https://www.yelp.com/html/pdf/YelpDatasetChallengeWinner_PersonalizingRatings.pdf

Problem Statement

The problem statement for this project is to develop a modeling technique that will extract distinct topics from within a set of online product reviews that is too large to read in its entirety, and present them in a way that helps a reader (either a consumer or a business owner) understand what aspects of the product reviewers are evaluating (topics) and if any of those topics are specifically good or bad. A successful solution would help a consumer in their decision making process by telling them whether others have evaluated the product as good or bad with respect to specific criteria (topics), and would help a business owner get specific feedback on those same criteria (topics).

The tasks involved in creating such a solution are:

1. Obtain a set of online product reviews, including raw human-generated review text and a quantitative score
2. Prepare the raw text so that it can be ingested by an LDA model
    a. This involves a set of steps standard in NLP analysis - removing stop words, lemmatizing text, removing punctuation, lowercasing, etc.
3. Insert additional “codewords” into the text to encourage the emergence of topics that are specifically positive or negative in nature
    a. I will test Jack Linshi’s method, inserting “GOODREVIEW” and “BADREVIEW” after each positive or negative word in a review, respectively
    b. I will also extend his method by adding “valence codewords,” inserting the additional codewords “VGOODREVIEW” and “VBADREVIEW” for words that exceed a threshold of sentiment intensity
4. Develop topic models for each product using various versions of the review text
    a. I will use gensim’s implementation of LDA modeling, and tune the models to achieve the highest coherence measure as discussed below
5. Evaluate the topic models to determine which produces the most useful outputs
    a. I will use several quantitative methods to compare the models, as well as presenting qualitative analysis
6. Create a format for presenting the outputs of the best topic modeling approach to a user in a way that will be useful to them

Metrics

Establishing a metric to measure how “good” the topics that come out of a topic modelling exercise is challenging. There are a variety of approaches to doing so, but often these measures still don’t show very strong correlation with human assessments of topic quality.

Coherence

The primary metric I have chosen to look at in this project, and the one for which I will optimize the topic models themselves during model tuning, is coherence.

Coherence is defined as the extent to which items in a set support each other - in this case, the extent to which words in a topic support each other. There are many variations to computing this measure, but the method can generally be described as a pipeline that includes segmentation, probability estimation, confirmation measure, and finally aggregation. The below is a summary of the more thorough discussion of this pipeline in the AKSW paper "Exploring the Space of Topic Coherence Measures."[2]

1. Segmentation: first, segment a word set **W** into a set of pairs of subsets (for example, subsets of groups of words or individual words). The first half of the subset the primary set **W'** and the second is the conditioning set **W*.**
2. Probability estimation: estimate the probability of occurrence of each half of the subset **W'** and **W***. For example, estimate the probability of a single word as the number of documents in which the word appears divided by the total number of documents. "Documents" here refers to the set of words that fall within a sliding window of a specified word size that moves over **W.**
3. Confirmation measure: take a single subset pair and use the previously calculated probabilities to estimate how strongly the second half of the pair (the conditioning set **W***) supports the first half of the pair (the primary set **W'**). There are many ways to compute this, but they can be divided into direct and indirect measures of confirmation. There are direct confirmation measures, such as the difference or ratio measures. There are also indirect confirmation measures - these can help when there are words in the corpus that don't have high joint probability (they don't occur often together in documents), but support each other semantically. Indirect confirmation essentially takes a transform of the word vectors for **W'** and **W*** and computes their similarity according to a variety of similarity measures such as cosine, dice, or jaccard.
4. Aggregation: the confirmation measures from Step 3 for all the subset pairs are aggregated into a single coherence score. This is often done by using the average but other summary measures such as median, maximum, or minimum might be used depending on the context. This score expresses the "coherence" of the word set **W**, that is, how much the words in the set support each other.

Gensim has available a coherence pipeline like the one above that allows for several customizations within the various steps. I've chosen to use a coherence measure dubbed "C_v"

[2] M. Röder, A. Both & A. Hinneburg. 2015. Exploring the Space of Topic Coherence Measures. *Proceedings of the Eighth International Conference on Web Search and Data Mining, Shanghai, February 2-6.* http://svn.aksw.org/papers/2015/WSDM_Topic_Evaluation/public.pdf

by the AKSW paper authors, which comes from a coherence pipeline newly defined by them in their work. This coherence measure uses, in each step:

1. NPMI - segmentation that takes larger subsets into account
2. Boolean sliding window, with a window size of at least 50 words
3. Cosine similarity as indirect confirmation method
4. Mean or median as the aggregation method

I chose C_v for its high performance amongst the various pipelines compared in the AKSW paper as well as for its ease of interpretability - cosine similarity scores (and by extension the mean or median of the cosine similarity scores) are always between 0 and 1, with 0 indicating that the two word vectors have nothing in common (they are orthogonal) and 1 indicating perfect similarity. In addition this is a normalized score, so it can be used to compare different models and assist in model tuning in a way that some other measures cannot.

II. Analysis

Data Exploration

(See “Data Cleansing & Prep” Jupyter notebook)

The dataset consists of user reviews for 74,258 food products from Amazon.com produced between 1999 and 2012. There are a total of 568,454 reviews from 256,059 unique users in the dataset, but a large number of these are duplicates that had to be removed. I obtained the data from Kaggle[3] but it was originally published on SNAP[4]. The dataset was produced by Julian McAuley and Jure Leskovec at Stanford University; they collected the publicly available data from Amazon’s website using a web crawler[5].

Each item in the dataset is a review, with a corresponding unique review ID, a ProductID, a user ID and username, title and review text, an overall rating for the product as well helpfulness ratings for the review, and some metadata such as datetime information. I will primarily use the ProductID, Score, and Text fields for my analysis.

In addition to the product data, I am also relying on a couple of lexicons for the sentiment analysis and code word insertion. For general positive vs. negative sentiment, I am using the Hu-Liu lexicon[6] to insert the tag “GOODREVIEW” wherever a positive word is found and

---

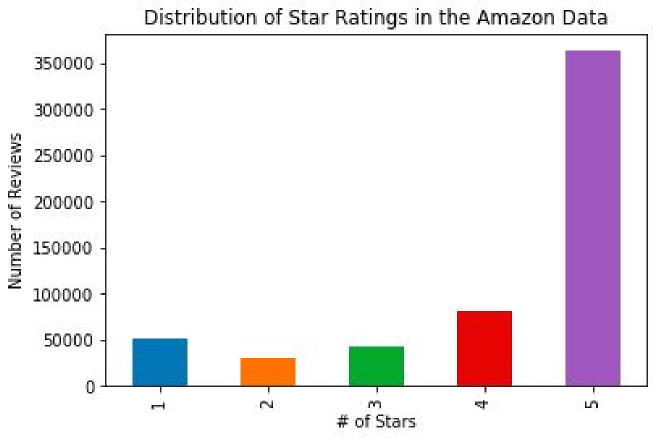
[3] https://www.kaggle.com/snap/amazon-fine-food-reviews
[4] http://snap.stanford.edu/data/web-FineFoods.html
[5] Julian John McAuley and Jure Leskovec. 2013. From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews. In Proceedings of the 22nd international conference on World Wide Web (WWW '13). ACM, New York, NY, USA, 897-908. http://i.stanford.edu/~julian/pdfs/www13.pdf
[6] Minqing Hu and Bing Liu. 2004. Mining and summarizing customer reviews. In Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '04). ACM, New York, NY, USA, 168-177. https://www.cs.uic.edu/~liub/publications/kdd04-revSummary.pdf

"BADREVIEW" wherever a negative word is found. For the valence-aware sentiment, I am using the VADER lexicon[7] to insert additional codewords "VGOODREVIEW" and "VBADREVIEW" wherever a word above a threshold of intensity is found.

Exploratory Visualization

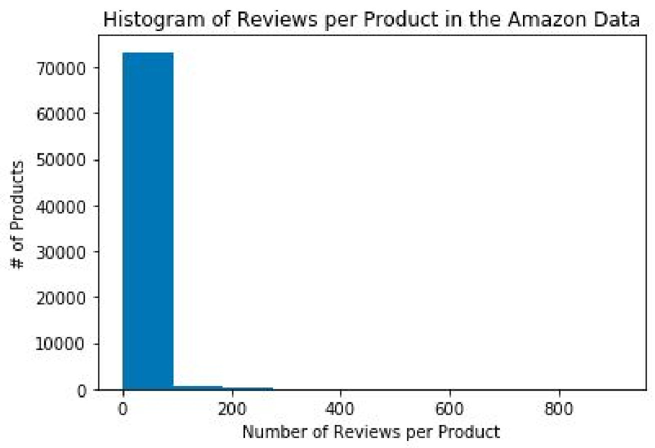
(See "Data Cleansing & Prep" Jupyter notebook)

As is common with online product reviews, there is heavy class imbalance in favor of positive ratings vs. negative ratings. This is obvious in a bar chart of the original full review set. This will mean that some upsampling or other approach to address the class imbalance would be needed if classification modeling or similar is to take place.

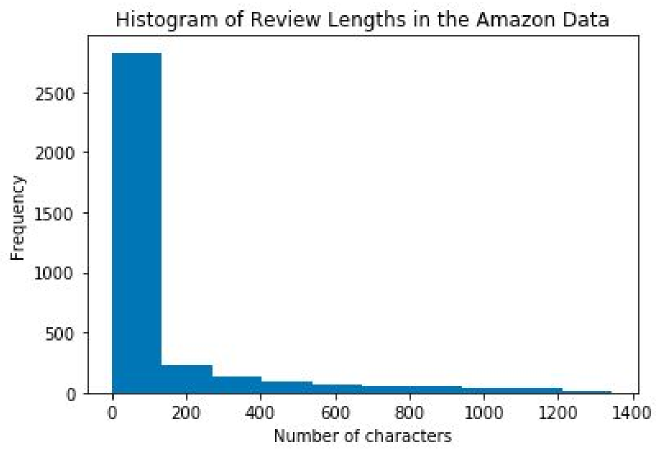


Additionally, there were a large number of products in the dataset that had very few reviews - more than 85% of the original product set had fewer than 10 reviews each. The number of reviews served as the primary filter for getting to the subset of products actually used for this analysis - only the top ten products in terms of review counts were used.

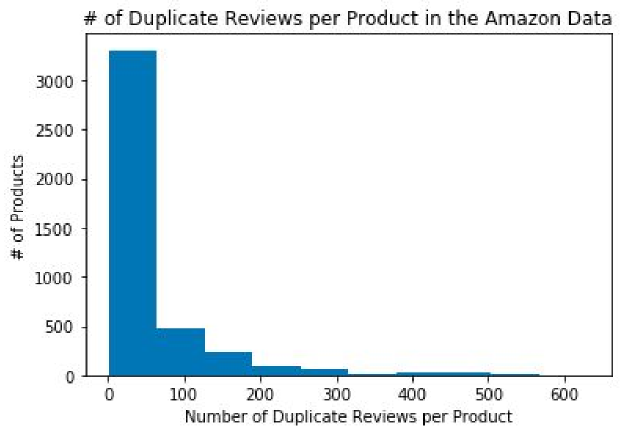
[7] C.J. Hutto. E.E. Gilbert. 2014. VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text. Eighth International Conference on Weblogs and Social Media (ICWSM-14). Ann Arbor, MI, June 2014. http://comp.social.gatech.edu/papers/icwsm14.vader.hutto.pdf



Length of reviews could also be important - reviews that are too short won't offer much information for the topic modeling algorithms to use. Most reviews in the dataset had fewer than 150 characters, but there were more than 300 products with at least 200 reviews above this threshold.



The final significant issue I uncovered during exploratory analysis was the presence of a large number of duplicates in the dataset.



It turned out that there were a number of products that seemed to be present in the dataset under different product IDs, such that there were two (or more) fully identical sets of reviews present in some cases. I identified these products and removed them entirely, and then skinnied the final dataset down to the ten products with the highest number of reviews. This left a final dataset of 5,392 reviews for 10 products:

Product ID	**Review Count**
B0026RQTGE	629
B001EO5Q64	567
B000KV61FC	556
B000NMJWZO	542
B0013A0QXC	542
B005K4Q34S	541
B003GTR8IO	530
B005ZBZLT4	506
B006MONQMC	491
B002IEZJMA	487

Algorithms and Techniques

Latent Dirichlet Allocation

Background and the Generative Model

Latent Dirichlet Allocation: Latent Dirichlet Allocation (LDA) is an unsupervised method for identifying "topics," or co-occurring groups of words, in a corpus of texts. To set out some common terminology, aligned with one of the primary papers outlining the LDA approach[8]:

- A ***word*** is discrete piece of data, also sometimes called a token, and belongs to a collection of words (a vocabulary) indexed from {1,...., V}
- A ***document*** **w** is a sequence of *N* words, written as $\mathbf{w} = (w_1, w_2, \ldots, w_N)$ where w_n is the *n*th word in the document
- A ***corpus D*** is a collection of *M* documents, written as $\boldsymbol{D} = (\mathbf{w_1}, \mathbf{w_2}, \ldots, \mathbf{w_M})$
- A ***topic k*** is a probability distribution over the vocabulary ***V*** of words. I will refer to this throughout as the *topic term distribution* or *term distribution*. There are ***K*** topics in all (this is a number we fix)

The motivating idea of LDA is that we believe the documents in our corpus were created based by some set of ***K*** underlying "topics." "Created" is a key word here - this is a generative model - meaning that we assume that an underlying generative process has created the documents we have in our corpus. LDA says that each document is generated (i.e., written) in the following way:

1. Decide how many words will be in the document
2. Decide on a *topic mixture* from which to create the document - i.e., the document will draw ⅓ of its words from topic A, and ⅔ of its words from topic B
3. Generate each word of the document by
    a. Choosing a topic to draw from, according to the topic mixture decided in step 2 (i.e., draw from topic A with ⅓ probability, and from topic B with ⅔ probability), and then
    b. Using the topic term distribution to generate the word itself - as stated above, the topic is just a distribution over a vocabulary of words

This might not be the process you use as a human when writing a document, but by assuming this process we can learn the underlying topics that might have generated a collection of documents, using LDA. To understand it, first break down the name of the algorithm. "Latent" refers to the fact that there are hidden (latent) variables generating our documents that we cannot directly observe, in this case the topics themselves. "Dirichlet" refers to the particular type of multivariate distribution by which we believe the topic mixtures (step 2) for each document are actually generated. I'll restate the generative process with a little more notation:

[8] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent dirichlet allocation. J. Mach. Learn. Res. 3 (March 2003), 993-1022. http://www.jmlr.org/papers/volume3/blei03a/blei03a.pdf

1. Choose *N* as the number of words for your document, according to some probability distribution (like a Poisson)
2. Choose θ, the *k*-dimensional topic mixture for the document, which is drawn from a Dirichlet distribution over α, where α is a *k*-dimensional Dirichlet random variable. Essentially drawing a variable from a Dirichlet distribution lets us dictate that all the values contained within it are between 0 and 1, and also that they sum to 1 - so it basically lets us express the relative likelihoods of *k* different events (or topics in this case).
3. Generate each word of the document by
    a. Choosing the topic identity for each word z_n, chosen according to the multinomial distribution θ, from which we will draw each word, and then
    b. Choosing each word w_n from $p(w_n \mid z_n, \beta)$, which is a multinomial probability conditioned on z_n and β, where β is the term distribution, a *k* x *V* matrix that tells us the probability of each word *w* in *V* appearing in each topic *k.*

So when we start from a collection of documents and an assumption about the number of topics *k*, what we want to do is find the α (topic mixtures) and β (term distributions) most likely to have generated our corpus of documents. From here, the details of the algorithm get very complicated, so I will give a general explanation and point to further reading for a more in-depth discussion.

Bayesian Inference

Bayes' Theorem allows us to understand the probability of something we can't observe directly (like a hidden or latent variable), based on information we have about the probability of other related variables that we can directly observe. In Bayesian language, we want to know the *posterior distribution* of the term distributions β, the topic mixtures θ, and the per-word topic identities *z*, conditioned on (given) the documents we have in our corpus. In plain English, given the documents in the corpus, what topics are present in each document, and of what words (with what frequency) do the topics consist? This is the underlying structure of the corpus.

Unfortunately we can't calculate this posterior distribution directly because it is too complex, so it has to be approximated. This can be done in several ways; a common approach is to use something called variational inference. Variational inference approximates a complex distribution like the one we are trying to find by using a family of simpler distributions and a set of free parameters. The learning process is then organized around minimizing the KL (Kullback-Leibler) divergence between those two distributions - essentially, finding the settings of the free parameters that make a simpler distribution that is as close to the true posterior distribution as possible. For much more mathematical detail on this, see the excellent overview of variational inference on Eric Jang's blog[9].

[9] Eric Jang. A Beginner's Guide to Variational Methods: Mean-Field Approximation. August 2016. http://blog.evjang.com/2016/08/variational-bayes.html

The online variational Bayes algorithm and gensim implementation

I will be using the gensim implementation of LDA, which is based on a python script by Matthew Hoffman that implements the algorithm described in Hoffman's paper outlining an "online variational Bayes algorithm for LDA[10]." What does this name mean, briefly?

- *"Online"* refers to the fact that it is designed to work with chunks of documents, updating itself after passing over each chunk. This is in contrast to the standard batch algorithm which must iterate through the entire corpus of documents before completing its update steps. This makes it useful for situations where a corpus might be expanded over time, so that you could deploy the algorithm to update periodically as new documents are produced from an online source (e.g., a ratings/reviews website). However even with a fixed corpus such as I am using it has some advantages including faster convergence than the batch algorithm.
- *"Variational Bayes"* refers to the way we are approximating the density function of a posterior probability, by using a simpler distribution as explained above. Recall that the "Dirichlet" in LDA refers to the use of a Dirichlet distribution to model the prior probability of topics.

A plain English outline of the basic steps necessary to do LDA topic modeling using the gensim implementation is as follows:

1. Starting with the collection of documents ***D***, create what gensim refers to as a *Dictionary*, which assigns an integer to each unique word, or token, in the corpus and creates a mapping to access the tokens and their integer IDs.
2. Count the number of occurrences of each unique token in the dictionary, and store the results in a sparse matrix of tuples where each tuple contains the integer ID of a token and its number of occurrences in the collection of documents. Be aware that gensim refers to this matrix as a *corpus* (and it is indeed a representation of what I called a corpus above, the collection of documents being studied).
3. Take an optional step of removing any tokens that aren't wanted for the analysis - this could be removing tokens that appear extremely frequently or infrequently, for example. Here you could also specify tokens that are not to be dropped (such as the codewords I am inserting).
4. Set an expectation about the number of topics *K* that organize the corpus.
5. Begin the following learning process, which is repeated until convergence (i.e., KL divergence is no longer decreasing):
    a. Randomly initialize term distributions for each topic
    b. For each document in the corpus (or the chunk of the corpus, if using the online algorithm):

[10] M Hoffman, DM Blei, F Bach. 2010. Online learning for latent dirichlet allocation. Advances in Neural Information Processing Systems 23, 856-864.
http://papers.nips.cc/paper/3902-online-learning-for-latent-dirichlet-allocation.pdf

    i. Set the values of the topic mixture vector for that document to an arbitrary constant value (in the gensim implementation, it is 1)
    ii. Repeat the following two steps until the value of the topic mixtures for the document are no longer changing
        1. Update the free parameters representing the likelihood of per-word topic assignments based on the expected value (under the variational distribution that is approximating the true distribution) of the current (initially random) term distribution values and the current (initially fixed) topic mixture values
        2. Update the topic mixture vector for this document based on the values set in the previous step

c. After each document's topic mixture has stabilized in the above steps, update the (initially random) term distributions for each topic, by summing over the documents the product of each word's frequency of appearance in the document and its likelihood of of assignment to that topic.

At the conclusion of the learning process we have two key items of information:

- The topic term distributions, a set of *k* vectors with *V* values each that describe the probability of each word being generated by each topic - this is *β* from step 3b of the generative process
- The topic mixtures for each document, a set of *D* vectors with *k* values each that describe the likelihood of each successive word in the document being drawn from topic *k* - this is θ from step 2 of the generative process

Using LDA, we can obtain each of these pieces of information using only the word frequencies for each document, and an assumption about how many topics there were.

Benchmark

Given that topic modeling is unsupervised, it is more difficult to judge its performance than with a supervised learning model where performance can be measured in terms of accuracy on a test set with known labels. There are a variety of metrics including perplexity, topic difference, and coherence that are used to evaluate topic models. "Quality" of the topics in terms of how well they reflect what the user was discussing in their review is much harder to quantify and usually requires some level of manual review by a human in this type of research.

However, since we are interested in getting topics that have the same function as a star rating (in that they give a user quick insight into the general consensus of other users), we can use a supervised learning method to look at the relationship between the two. Specifically, if we treat the probability distribution over the sentiment-aware topics as features for the classification task, can we predict the overall star ratings? Jack Linshi took this approach in his Yelp research and found that the topics from his modified LDA approach outperformed the predictive power of topics generated by traditional LDA in a regression analysis against star ratings. So, if the additional modification with sentiment-aware codewords is an effective one, I would expect the topics generated from that input to have more predictive power than the simple codeword topics.

Natural language processing often involves some level of human intervention, whether it is an extensive labelling exercise or careful inspection of outputs to determine their quality. Additionally, it is more difficult to benchmark NLP ML models because their performance and tuning is so context-specific. I will be benchmarking my analysis internally by comparing the outputs of the three variants of LDA topic modeling (‘vanilla’ text inputs, coded inputs, and valence coded inputs) in several ways:

1. Compare the coherence of the topics generated using each of the three text input types to determine whether encoding the reviews produces topics with a higher coherence score
2. Compare the power of codeword counts in each review to predict that review’s star rating (codewords vs. valence codewords only)
    a. Count the number of codewords and valence codewords appearing in each review
    b. Simple linear regression of codeword and valence codeword counts against star rating to determine which has more predictive power - measured by r-squared values as well as by the average difference between true and predicted star rating
3. Compare the power of topic assignment distributions for each review to predict that review’s star rating (compares vanilla, coded, and valence-coded inputs)
    a. For each review, acquire the probability distribution across topics produced for each product (i.e., the likelihood that each topic term distribution produced the review)
    b. Simple linear regression of the distributions against the star rating to determine which input (vanilla, coded, valence) has more predictive power - measured by r-squared values as well as by the average difference between true and predicted star rating
4. Qualitative assessment: review the topics generated for each product on a more qualitative basis by looking at the term distributions for each and evaluating their interpretability and usefulness to a person

III. Methodology

Data Preprocessing

(See “Data Cleansing & Prep” Jupyter notebook)

Removing Duplicates

As part of the exploratory data analysis, I checked the dataset for duplicate reviews. In fact there were a huge number of duplicates - reviews that were identical in their text content and came from the same username and user ID, but belonged to different product IDs. It's not clear exactly where the duplicates came from, but it may be that some products have more than one product ID on the Amazon marketplace. To deal with the duplicates I simply dropped out any product whose entire body of reviews was duplicated by another product.

Review Counts

I dropped out all the products that had fewer than 10 reviews - this would not be a sufficient amount of data to get anything of quality from the topic modelling. Next, I worked to remove duplicates from the data - there were a large number of fully duplicated rows in the data. Additionally, in some cases there were products that seemed to have multiple product IDs (i.e., two product ID numbers corresponded to the same product). Where the full review set was identical for each of the product IDs, I have also dropped those products. Where there was overlap but not full duplication, I have not removed ay rows so as not to lose any of the unique review data.

Text Pre-Processing

To prepare the text so that it can be used by other algorithms, I've taken a number of typical steps. I used NLTK libraries and worked with a pandas dataframe to take advantage of vectorized operations as much as possible.

1. **Removing HTML markup:** some reviews contain links to Amazon product pages or other webpages, or have other bits of markup. I removed these using regular expressions.
2. **Expanding contractions and lower-casing:** due to splitting and later removal of punctuation, I want to have as few contractions in the reviews as possible. Many lists of English-language contractions can be found online; I have chosen to make a custom dictionary that makes as few assumptions about the context as possible. For example, expanding the contraction "I'd" requires a choice about whether to expand to "I had" or "I would." In particular, it is important to expand negative contractions like "didn't" so that handling of negation (discussed below) can work properly. This step also lower-cases the whole dataset and is a very typical way to pre-process text for natural language analysis.
3. **Inserting sentiment codewords**: this is a step unique to my analysis. It's covered below in more detail, but essentially a "codeword" like "GOODREVIEW" or "BADREVIEW" is inserted after each positive or negative word in the review.
4. **Lemmatizing the text**: Lemmatization is another very standard NLP step. It returns each word to its "root", so that so that "trying," "tried," and "tries" would all become "try,"

for example. This makes topic modeling more effective by reducing the number of unique words with similar meaning present in the text. We don’t want the topic model to consider “cookie” and “cookies” as two different things, nor do we want the conjugation of a verb to affect what we know about how often it is used in a review. Lemmatization requires three steps that can be accomplished using NLTK libraries and regular expressions.

   a. *Tokenization*: Tokenization refers to splitting a text string document up into its individual elements; these individual items (typically words) are referred to as "tokens" in natural language processing.
   b. *Tagging*: this involves using a Part-of-Speech (POS) tagger to identify the grammatical role that each word plays in a sentence. I have used the WordNet POS tagger available through NLTK. The tagger is context-aware, so generally it is successful in determining whether the word "trip" is a noun or a verb, for example.Once the words are tagged (noun, verb, etc.), they can be reduced to their lemma, or the root version of the word. For example, ‘ran’ and ‘running’ become ‘run,’ pluralization is removed, etc. This allows the topic modelling algorithms to pick out common themes more easily.
   c. *Lemmatization*: The last step, the word can now be returned to its root given that we know its POS tag - nouns can be returned to singular form, verbs can be returned to infinitive, etc.

5. Removing stop words, punctuation, and numbers: The last step is to remove anything left that isn’t a word (like punctuation and numbers), and stop words. Punctuation and numbers are easily removed through regular expressions. Stop words are common words that don't carry much information, like “the,” “and,” etc. There are many pre-made lists of stop words available on the internet. For the purposes of this analysis, I started with a common list and added some words to it - these are words that might appear too frequently across reviews and end up influencing how a topic is formed when I don't really want them to. Removing stop words gives the topic modelling algorithm better data to work with, and allows topical words to rise to the top more easily. I also don't want common negating words (like “not”) influencing the topics too much, so I'll create a version of the review without those words, for topic modeling, and a version that leaves them in, for readability of later n-gram analysis.

Codeword Insertion

To do the analysis in this project, I enriched the reviews in several ways, mostly related to the insertion of what I am calling “codewords.” I’ve inserted two different sets of codewords - simple codewords and “valence-aware” codewords. The basic approach is to check the sentiment of each word in the review by referencing a lexicon of word sentiment values, and insert the appropriate codeword.

This is a technique inspired by Jack Linshi’s Yelp dataset paper and should service to make topics that are more common in positive and negative reviews easier to identify. For example, we have many words in the English language that mean “good” or “bad.” Reviews will contain a whole range of these words - a human reader is quickly able to identify them as communicating a common sentiment, but without some additional information a computer cannot. Especially in this case where the descriptive word is usually being applied to a subject (like “taste,” “packaging,” etc.), these codewords essentially replace the various “good” or “bad” words. In a collocation analysis, that means we can more easily pick up on comments about what aspect of the experience was positive or negative, because in addition to a descriptive word, the codeword will appear as well.

One element of complexity in this step is negation. If a review states, “The taste was not great,” we don’t want to assign a positive codeword to the word “great.” I modified an NLTK utility so that if a word appears between a negation and a punctuation mark, its sentiment will be reversed, based on techniques used by the NLTK authors and others.

To insert the simple codewords “GOODREVIEW” and “BADREVIEW,” I used the Liu-Hu lexicon made available through NLTK, and removed or added a few words based on the context of this project.

To insert the valence-aware codewords “VGOODREVIEW” and “VBADREVIEW,” I used a lexicon from VADER (for Valence Aware Dictionary for sEntiment Reasoning). The VADER lexicon assigns a numeric value to each word in the lexicon. For words that exceed a certain threshold (either positive or negative), they are assigned the appropriate codeword “VGOODREVIEW” or “VBADREVIEW,” while words below those thresholds receive the basic “GOODREVIEW” or “BADREVIEW” tag.

At this point the dataset contains three different sets of clean reviews for analysis - “vanilla” text with no codewords, coded text with the basic codewords only, and valence coded text that includes the valence-aware codewords.

Other Dataset Enrichment

I created a new feature in the dataset, Sentiment. This is a binary feature that indicates whether the Score for the review was a 4 or 5 (which I am calling positive sentiment).

Finally, for each review, I’ve added a count of all the codewords present in the review.

Implementation

(See “Topic Modeling” Jupyter notebook)

Data pre-processing has produced three corpuses for analysis - “vanilla” text, text with codewords, and text with valence-aware codewords. The next step is to run and tune LDA models for each model across each of these corpuses.

In this project, I am doing topic modeling on a per-product basis. This means that instead of using all the reviews as one big input set, I am splitting them up by product and developing an individual topic model for each product and each input text type. There are a couple of reasons for this - first, I do believe that there might be overarching topics that are present across products, such as topics related to shipping speed and packaging, or taste generally since these are mostly food products. However, I'm really more interested in the unique topics that might come up just for an individual product, related to some aspect of the user experience that is unique to that product (e.g., ingredients). Even in the case of themes that are present across products, I'd want to know about the manifestation of that topic for an individual product. The tradeoff is that the input set of reviews is obviously much smaller for each product.

Another reason for building the topic models this way is that there are some words that appear very frequently but aren't meaningful in the context of topics - such as the brand name of a product. Leaving them in might lead to the generation of topics that are basically just mentioning a particular product. There are lots of ways to remove words like this that appear frequently but aren't meaningful (these might be thought of as context-specific stop words), but it would be hard to do it consistently for all the products in the set when treating the reviews as one big corpus.

Additionally, I'm letting the hyperparameters, most importantly the number of topics, vary across the different input text sets. This means that for the same product, I might end up with a topic model that has 12 topics and requires 80 passes for the vanilla (unencoded) text, but has 8 topics and takes 100 passes with one of the encoded text sets. This is because the intent of the codeword insertion is to get topics that contain a lot of positive or negative sentiment to coalesce - so I don't want to hold that number of topics constant.

Initial Grid Search - LDA Model

Using gensim's Latent Dirichlet Allocation (LDA) model library and some of my own code, I implemented a grid search to tune the some key hyperparameters - passes, number of topics, and two metrics involving how many and which words should be removed from the frequency input dictionary.

- **Passes** is a hyperparameter particular to gensim's implementation of LDA - it allows the model to see the corpus of documents multiple times during training, which is very useful for smaller corpuses such as mine (none of the products have more than 500 or so reviews).
- **Number of topics** is how many topics I am requesting the model to produce. As mentioned above this number is not fixed for any topic or input text set.
- **Top *n* words (*top_n)*** is a parameter indicating which very frequently appearing words should be removed. Though the stopwords list does a good job getting universally common words out of the topic modeling process, there are other words that are very common in the context of a review set - for example, words that describe the product

(“coconut,” “oil”) or its brand (“nutiva”). These words aren’t actually adding a lot of information in terms of our quest to understand the topics contained in the review set.

- **Review threshold *(no_above)*** is a parameter that sets another threshold for removing common words. Expressed as a float, it represents the percentage of reviews that a word appears in. Above this threshold, the word will be removed - this aims to get words that appear in almost all reviews out of the topic modeling, for the same reasons described above.

The metric I am looking to maximize is average topic coherence, as described in the Metrics section. I hoped to achieve a value of 0.5 average coherence for each product (out of possible values between 0 and 1). Recall that I’m using C_v coherence, a cosine similarity measure, where 0 indicates that the vector representations of the topics are orthogonal (no similarity) and 1 indicates that they are perfectly similar.

For each corpus (vanilla, coded, valence coded), I performed an initial grid search allowing for all combinations of [6, 8, 12] topics and [50, 80] passes over the corpus, with *top_n* in [2,10] and *no_above* in [0.5, 0.1].

In the first runs of these grid searches, I was not including the *top_n* and *no_above* metrics, and a lot of the results were really quite poor (in the 0.3 topic coherence range). Including those metrics added to the training time, but resulted in significantly better initial grid search results. Additionally, the models tuned with these additional metrics were much more robust as well - they were actually quite difficult to improve on significantly in the further manual tuning step.

The steps for the grid search implementation can be reviewed in detail in the notebook “Topic Modeling” and in the source code in lda_funcs.py.

LDA Input Comparisons - Initial Results

In general, the coded reviews (both basic and valence) had much higher coherence values right off the bat, versus the vanilla reviews. The comparison between coded and valence coded reviews is a little more mixed on the first pass of grid search.

	Vanilla Reviews			**Coded Reviews**			**Valence Coded Reviews**		
product	**coherence**	**n_topics**	**passes**	**coherence**	**n_topics**	**passes**	**coherence**	**n_topics**	**passes**
B002IEZJMA	0.456494	6	50	0.480028	12	50	0.479089	8	80
B006MONQMC	0.411036	6	50	0.488278	12	50	0.492218	10	50
B005ZBZLT4	0.398368	12	80	0.535866	10	50	0.451033	10	50
B003GTR8IO	0.389339	10	80	0.438479	10	80	0.488773	12	50
B005K4Q34S	0.424359	12	80	0.472876	12	80	0.494551	12	80
B0013A0QXC	0.333889	10	50	0.451919	10	50	0.453215	10	80
B000NMJWZO	0.429569	12	80	0.511190	12	80	0.494670	6	80

B000KV61FC	0.419217	12	80	0.475092	12	80	0.490179	12	80
B001EO5Q64	0.396260	6	50	0.489757	10	80	0.518175	6	80
B0026RQTGE	0.397528	6	50	0.434202	6	80	0.463676	6	50

Table 1: Results of initial LDA grid search analysis over possible parameters n_topics=[6,8,10,12] and passes=[50,80]

Refinement

Further LDA Tuning

After the initial grid search, I did a little more manual tuning for each product to see if the coherence scores could be at all improved. This simply involved eyeballing the best result from grid search and tweaking the parameters up or down slightly. For example, if the best model took 80 passes and 12 topics, I would run a second mini grid search with [80, 100] passes and [11, 12, 13] topics to see if average coherence could be improved.

In the end, the models using the basic coded reviews consistently perform best in terms of coherence. In just two cases, the vanilla reviews performed best, and the valence coded reviews didn't produce the highest coherence for any product. The best coherence value for each product is shown in bold italics in the table below.

	Vanilla Reviews					Coded Reviews					Valence Coded Reviews				
product	coh	t	p	top_n	no_ab	coh	t	p	top_n	no_ab	coh	t	p	top_n	no_ab
B002IEZJMA	***0.511191***	12	80	2	0.5	0.489860	12	80	10	0.5	0.479089	8	80	2	0.5
B006MONQMC	0.439355	12	50	2	0.5	***0.532982***	8	80	10	1.0	0.492218	10	50	2	1.0
B005ZBZLT4	0.479682	6	80	2	0.5	***0.535866***	10	50	10	0.5	0.470474	9	50	2	1.0
B003GTR8IO	0.413433	8	80	2	1.0	***0.507346***	11	80	2	1.0	0.488773	12	50	2	1.0
B005K4Q34S	0.446017	12	80	2	0.5	***0.499075***	5	50	2	0.5	0.498201	11	80	2	0.5
B0013A0QXC	0.473835	6	80	2	1.0	***0.476632***	11	80	2	0.5	0.453760	10	100	10	0.5
B000NMJWZO	0.497153	6	50	2	0.5	***0.546554***	9	80	2	0.5	0.494670	6	80	2	0.5
B000KV61FC	***0.496802***	10	80	2	1.0	0.494435	10	50	2	0.5	0.490179	12	80	10	0.5
B001EO5Q64	0.458964	9	80	2	1.0	***0.550463***	8	80	10	0.5	0.518175	6	80	2	1.0
B0026RQTGE	0.461583	7	80	2	1.0	***0.509347***	6	80	2	1.0	0.463676	6	50	2	1.0

Table 2: Results of refined LDA grid search analysis over possible parameters number of topics n_topics=*[6, 8, 10, 12], number of training passes* passes=*[50, 80], number of top frequent tokens to remove* top_n=*[2, 10], threshold of appearance frequency for removal* no_ab=*[0.5, 1.0]*

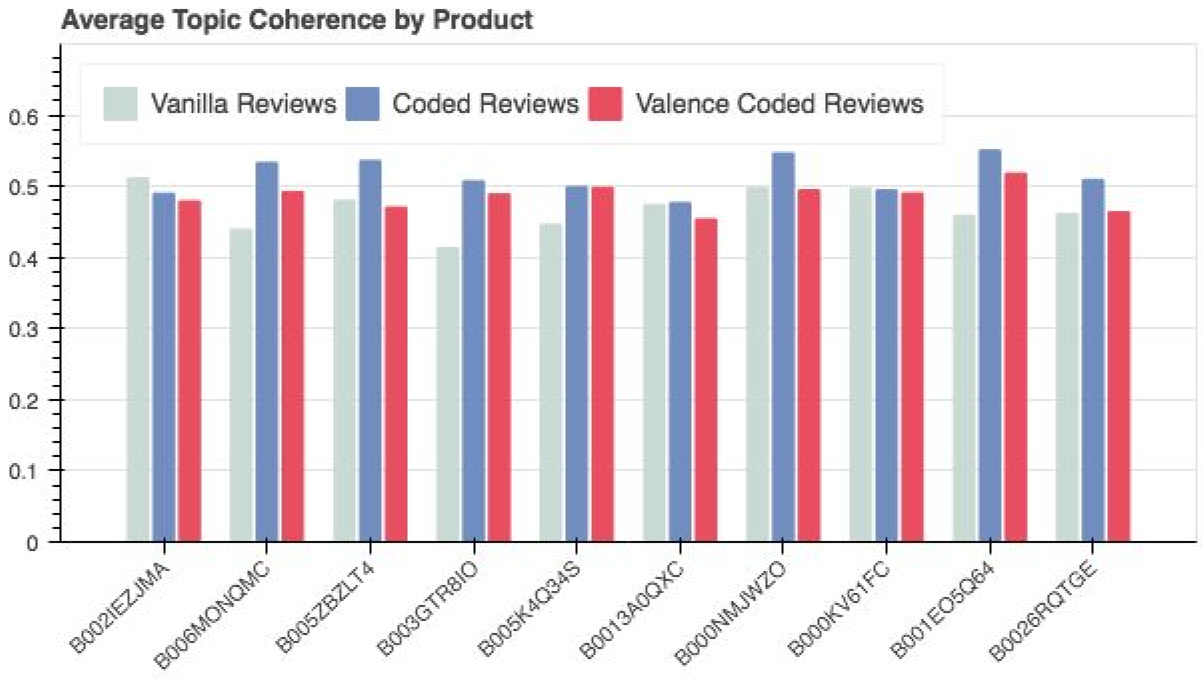
IV. Results

Model Evaluation and Validation

(See "Topic Model Visualization & Evaluation" Jupyter notebook)

Average Topic Coherence

As noted above, after running grid search and manually tuning the best models, the topic models using the simple codewords achieved the highest average topic coherence out of the three input types for all but two products.



Star Rating Predictive Power

Analysis of any online reviews with a score component typically includes an attempt to predict the score using whatever model is being developed. On Amazon, a user writing a review must give the product between 1 and 5 stars. So the first and simplest way to assess the value of the topic models is to see if they have any predictive power over star ratings.

As noted in the data exploration section, all the products in the dataset have a severe class imbalance problem as far as low-scored vs. high-scored reviews. To deal with this, I have treated each star rating value (1 through 5) as a class and have upsampled all minority classes to match the number of reviews in the majority class for each product.

First, I used a simple linear regression to see how well a simple count of the number of codewords in a review can predict its star rating. For each product, the table below displays the average difference between the regression model's predicted star rating and the actual star rating (over all reviews for that product), as well as the R-squared value for the regression model. Finally, results for a test of whether there is a significant difference between the average star rating gap using simple codewords vs. valence codewords.

Product	Avg Score Gap - Simple Codewords	Avg Score Gap - Valence Codewords	R-sq Coded	R-sq Valence
B002IEZJMA	1.022222	0.987037	0.239499	0.292767
B006MONQMC	1.087983	1.019241	0.189106	0.262805
B005ZBZLT4	1.016389	0.969103	0.256098	0.310939
B003GTR8IO	0.958230	0.915746	0.323831	0.365044
B005K4Q34S	1.058872	0.949178	0.213636	0.325040
B0013A0QXC	1.093235	1.028903	0.146099	0.223652
B000NMJWZO	1.109480	1.105521	0.158164	0.165802
B000KV61FC	1.120367	1.076863	0.126999	0.183014
B001EO5Q64	0.962651	0.887011	0.312179	0.396951
B0026RQTGE	1.042733	0.974923	0.261694	0.332914

Table 3: Power of regression over codeword counts to predict score rating

The results are pretty clear - in every case, the regression over valence codeword counts does a better job predicting the star rating of a review. Jack Linshi reported that his codeword counts were able to predict Yelp ratings within 0.91 stars. Performance for most of these products isn't quite that good, but as he points out even human reviewers don't achieve much better consensus when they try to complete a prediction task like this - it's not like image recognition where labeling by a group of human reviewers would have a very high degree of consensus.

Another way to assess the comparative predictive power of the three models is to use the topic assignment distribution for each review. More concretely: associated with each review there is a vector of values of dimension *t,* where *t* is the number of topics in the model for that product, and each value in the vector represents the likelihood that the topic generated this particular review. The values must sum to 1. Below are the results of a regression over the topic mixtures for each review against each review's star rating - again, a comparison of the average gap between predicted score and actual score, and the r-squared values.

Product	Avg Score Gap - Vanilla	Avg Score Gap - Coded	Avg Score Gap - Valence	R-sq Vanilla	R-sq Coded	R-sq Valence

B002IEZJMA	***1.091576***	1.107250	1.130436	0.171116	0.167198	0.106630
B006MONQMC	***1.073818***	1.097333	1.080866	0.210549	0.158421	0.185915
B005ZBZLT4	1.152803	1.146767	***1.093914***	0.089364	0.089285	0.135693
B003GTR8IO	1.196013	1.149682	***1.033393***	0.028426	0.101799	0.240125
B005K4Q34S	1.175577	1.205587	***1.009096***	0.060092	0.024306	0.263057
B0013A0QXC	1.195537	1.123160	***1.073270***	0.024710	0.117057	0.179977
B000NMJWZO	***0.903799***	1.016606	0.987132	0.364043	0.252699	0.267302
B000KV61FC	1.178548	***1.093550***	1.143541	0.065918	0.154792	0.097736
B001EO5Q64	***1.021393***	1.054449	1.124795	0.241496	0.185179	0.128967
B0026RQTGE	1.110817	***0.889268***	0.965197	0.117995	0.387711	0.312246

Table 3: Power of regression over topic assignment distribution values to predict score rating

These results are a lot more mixed. For ease of reading, the lowest score gap from among the three inputs is bolded and italicized. In four of ten cases, the best predictive power actually came from the topic assignment distributions over the vanilla topics. Valence topic assignment distributions had the best performance for four more products, and in just two cases did the best results come from the coded topic assignment distributions.

What does this suggest? For topic distribution values to be strongly linked to star ratings, the topics would need to be differentiated from each other in terms of their overall sentiment. That way, a higher probability value for a "positive" or "negative" topic would correspond to an increase or a decrease in the review's star rating. Theoretically, the insertion of these positive and negative codewords should be encouraging the topics emerging from the LDA modeling to be sentiment-aligned, but perhaps they are not.

In this section, the final model and any supporting qualities should be evaluated in detail. It should be clear how the final model was derived and why this model was chosen. In addition, some type of analysis should be used to validate the robustness of this model and its solution, such as manipulating the input data or environment to see how the model's solution is affected (this is called sensitivity analysis). Questions to ask yourself when writing this section:

- *Is the final model reasonable and aligning with solution expectations? Are the final parameters of the model appropriate?*
- *Has the final model been tested with various inputs to evaluate whether the model generalizes well to unseen data?*

- *Is the model robust enough for the problem? Do small perturbations (changes) in training data or the input space greatly affect the results?*
- *Can results found from the model be trusted?*

Justification

The two quantitative approaches I used to evaluate the different input text sets seem to show that judging which is best is a complicated question. In this section I will dig a little deeper into the results and give an explanation for how best to interpret them.

Aside from a high coherence metric, indicating that the topics would “make sense” to a person, and strong predictive power over the review star ratings, there are other things we would look for in a topic model. In particular for this application I would like to see:

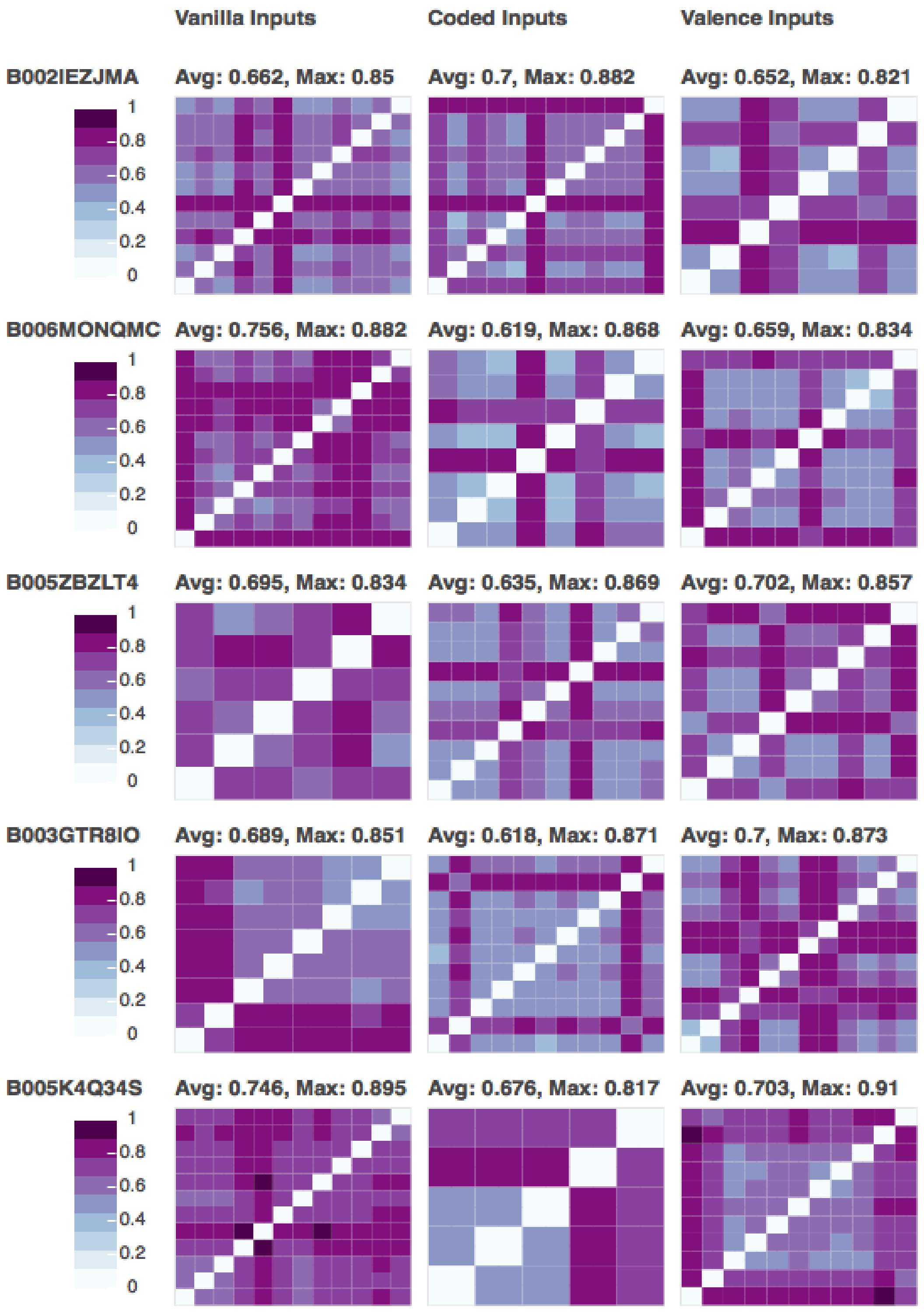
- Distinct topics, in terms of the topic term distribution itself. I don’t want to see a group of topics that are very similar, and hardly distinguishable to a human reader. That result doesn’t provide much of use or interest to a consumer or a business owner. However, such a result could also just indicate that the inputs on which the topic modeling was done were really very homogenous.
- Diverse topic assignments. Similar to the above, ideally I don’t want to see all reviews being assigned to the same one or two topics (i.e., a couple of topics have a high likelihood of having generated all the reviews). As stated above, such an outcome could reflect the fact that the reviews are actually very homogenous in content, but could also be a poor topic model that has failed on the above count.

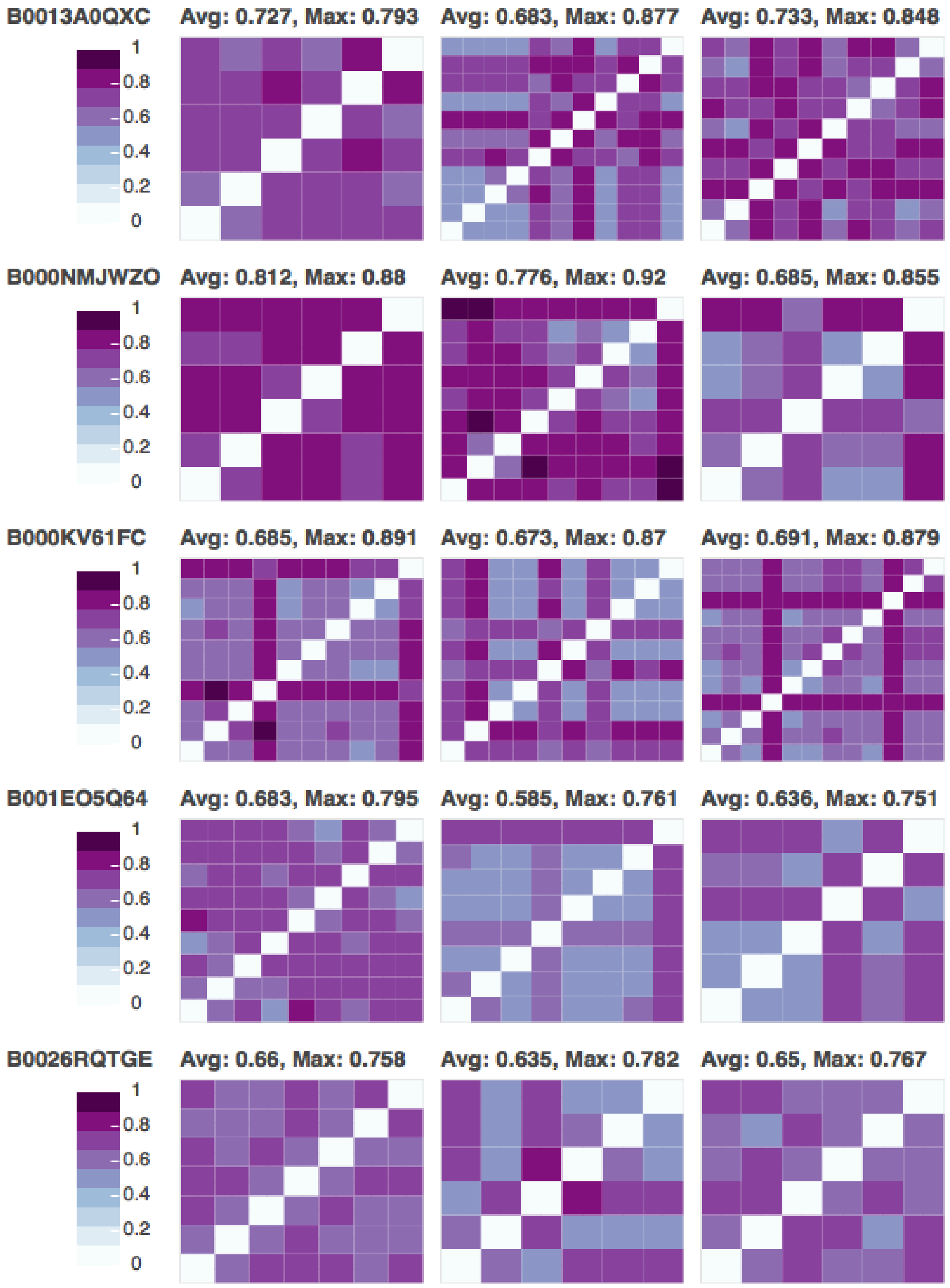
Distinct Topics

To assess how distinct the topics are in the final topic models for each of the products, I’ll be using a metric called the Hellinger Distance, which is a metric for quantifying the similarity between two probability distributions. The metric takes values between 0 and 1, with values closer to 1 indicating that there is less similarity (more distance) between the two distributions. A value of 0 would indicate that the two distributions are the same. I’m using a confusion matrix-style heatmap to visualize the Hellinger distances between topics for each product and input text set, below.

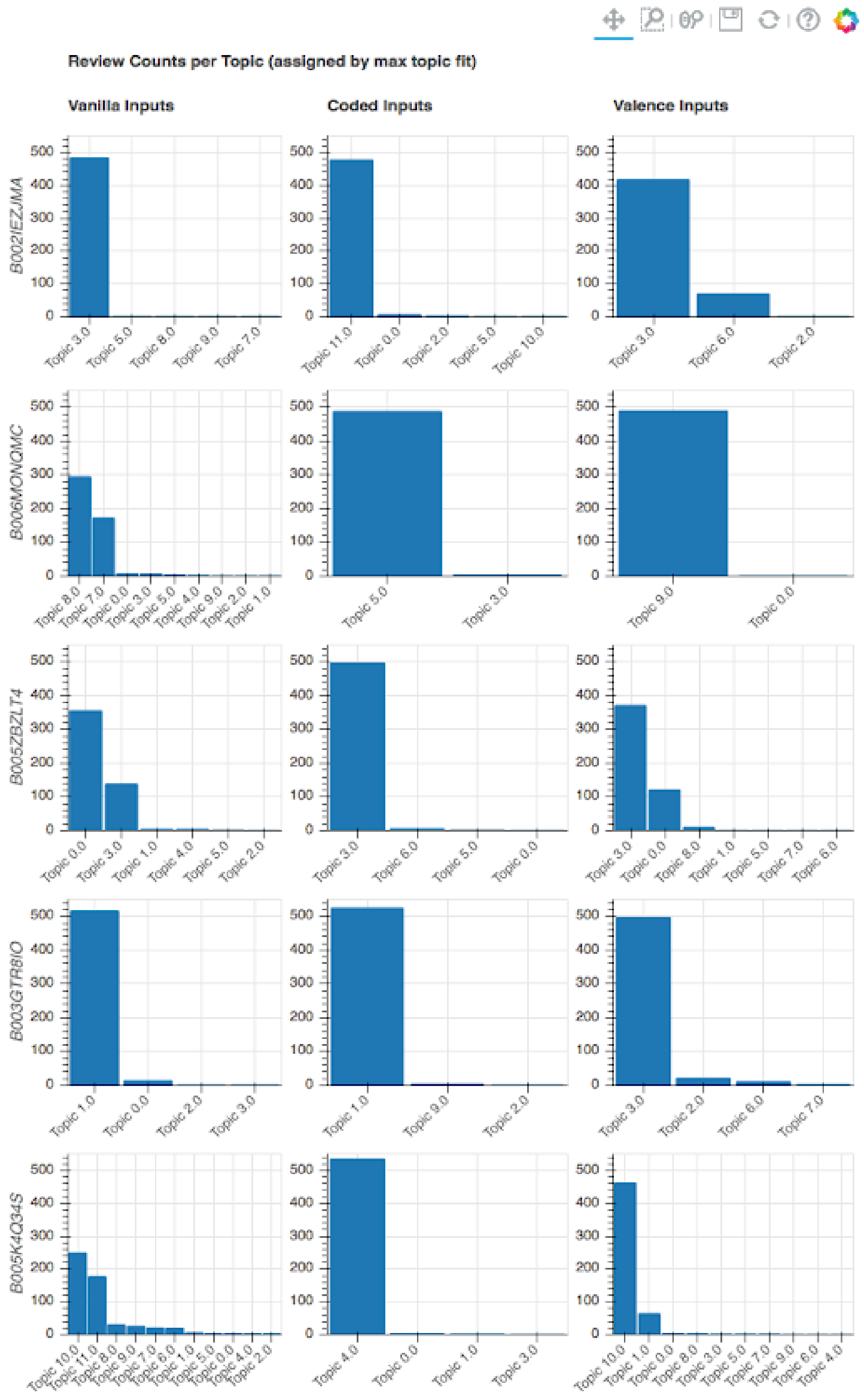
Overall, the results are good - there are few heatmaps that are mostly blue, which would indicate values closer to 0 and thus a set of very similar topics. Looking at the average Hellinger distance values, it appears that the topic models built from simple vanilla text may have a slight edge, as those models have higher average distance values six out of ten times, with the valence codeword models performing best in the other four cases. This is borne out visually; the simple codeword models seem to have large areas of blue in the heatmaps than the other two model varieties. This could mean that the codewords are overly simplistic and that the topic models using these text input sets are obscuring some of the topical diversity in the reviews.

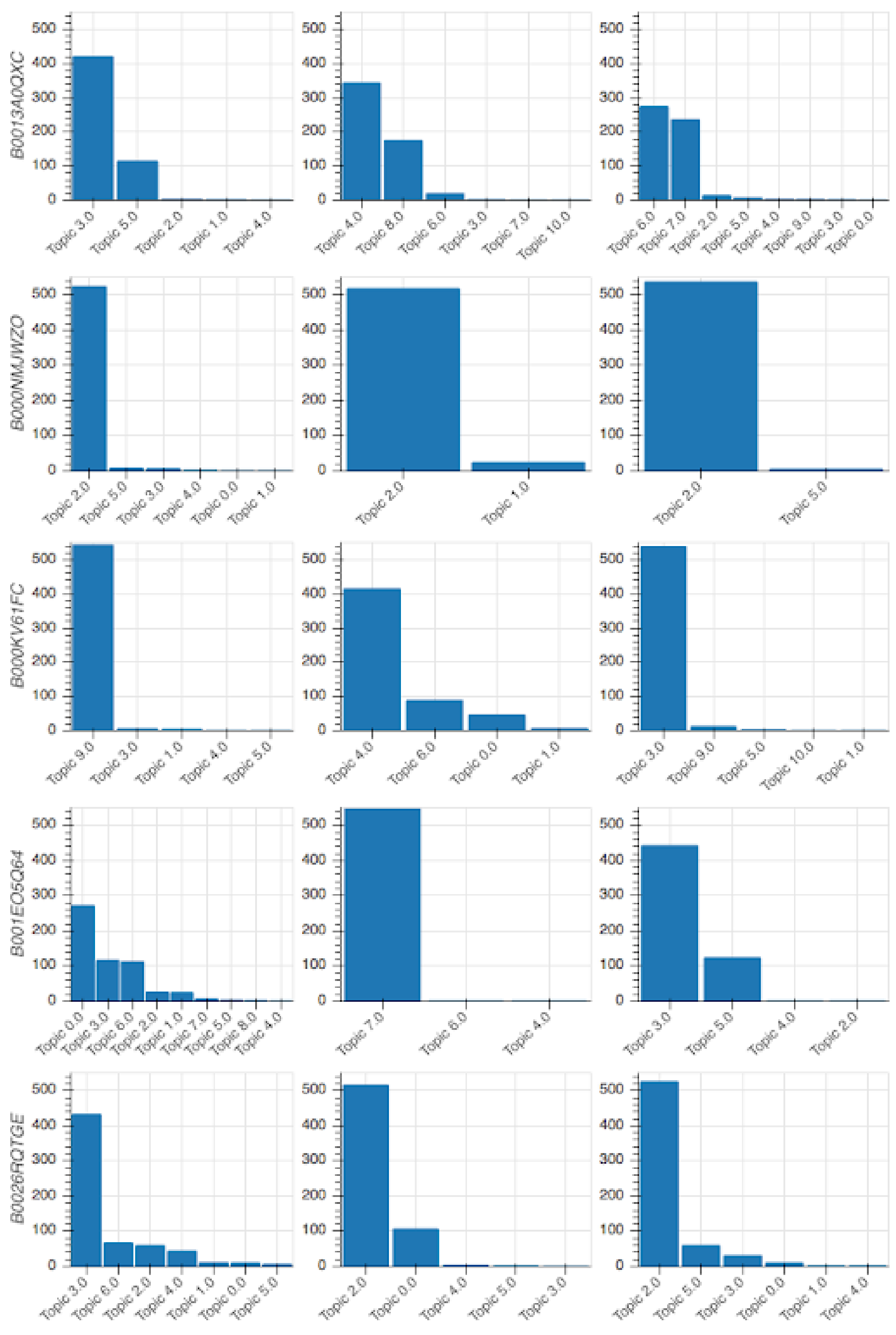
Per-Topic Hellinger Distances




B0013A0QXC
Avg: 0.727, Max: 0.793
Avg: 0.683, Max: 0.877
Avg: 0.733, Max: 0.848
B000NMJWZO
Avg: 0.812, Max: 0.88
Avg: 0.776, Max: 0.92
Avg: 0.685, Max: 0.855
B000KV61FC
Avg: 0.685, Max: 0.891
Avg: 0.673, Max: 0.87
Avg: 0.691, Max: 0.879
B001EO5Q64
Avg: 0.683, Max: 0.795
Avg: 0.585, Max: 0.761
Avg: 0.636, Max: 0.751
B0026RQTGE
Avg: 0.66, Max: 0.758
Avg: 0.635, Max: 0.782
Avg: 0.65, Max: 0.767
1
0.8
0.6
0.4
0.2
0

Diverse Topic Assignments





The bar charts above show how many reviews had the highest likelihood of having been generated by each topic in the topic model, by product and by input text type. That is to say, in the topic mixture for each document, which topic had the largest probability value? It's pretty clear that there are just one or two dominant topics for each product, no matter which input text type we are looking at. Note that only topics that had at least one review match are shown in these bar graphs, so although the topics at the left end of each x-axis may appear to have zero reviews, they all have at least one.

Additionally, the review assignment counts here seem to match up pretty well with the narrative about oversimplification by the simple codeword review sets - for the products where the heatmap showed a lot of blue (smaller Hellinger distances, more similar topics), the topic assignment is exceptionally concentrated. For example, look at the results for products B005ZBZLT4, B003GTR8IO, and B001EO5Q64, where the codeword topic models underperformed the vanilla and valence codeword topic models, and notice how concentrated the topic assignment counts are.

In this section, your model's final solution and its results should be compared to the benchmark you established earlier in the project using some type of statistical analysis. You should also justify whether these results and the solution are significant enough to have solved the problem posed in the project. Questions to ask yourself when writing this section:

- *Are the final results found stronger than the benchmark result reported earlier?*
- *Have you thoroughly analyzed and discussed the final solution?*
- *Is the final solution significant enough to have solved the problem?*

V. Conclusion

Free-Form Visualization

(See "Topic Model Visualization & Evaluation" Jupyter notebook)

In this section I'll give a couple examples of products for which the text encoding worked really well, and some examples of where it didn't work so well. I'm going to illustrate this first with a series of wordcloud visualizations for each product. A wordcloud is visualization of common words that appear in a given chunk of text, where the size of the words is usually proportional to how often the words appear. It's a nice visual way to get a sense of what is being said in the text and is a popular way to visualize data from social media posts, for example.

The wordclouds below are built in a very specific way, for a specific reason. I am building them on a per-product, per-topic basis - meaning a wordcloud for each topic that came out of the topic model for each product (and for each input text set).

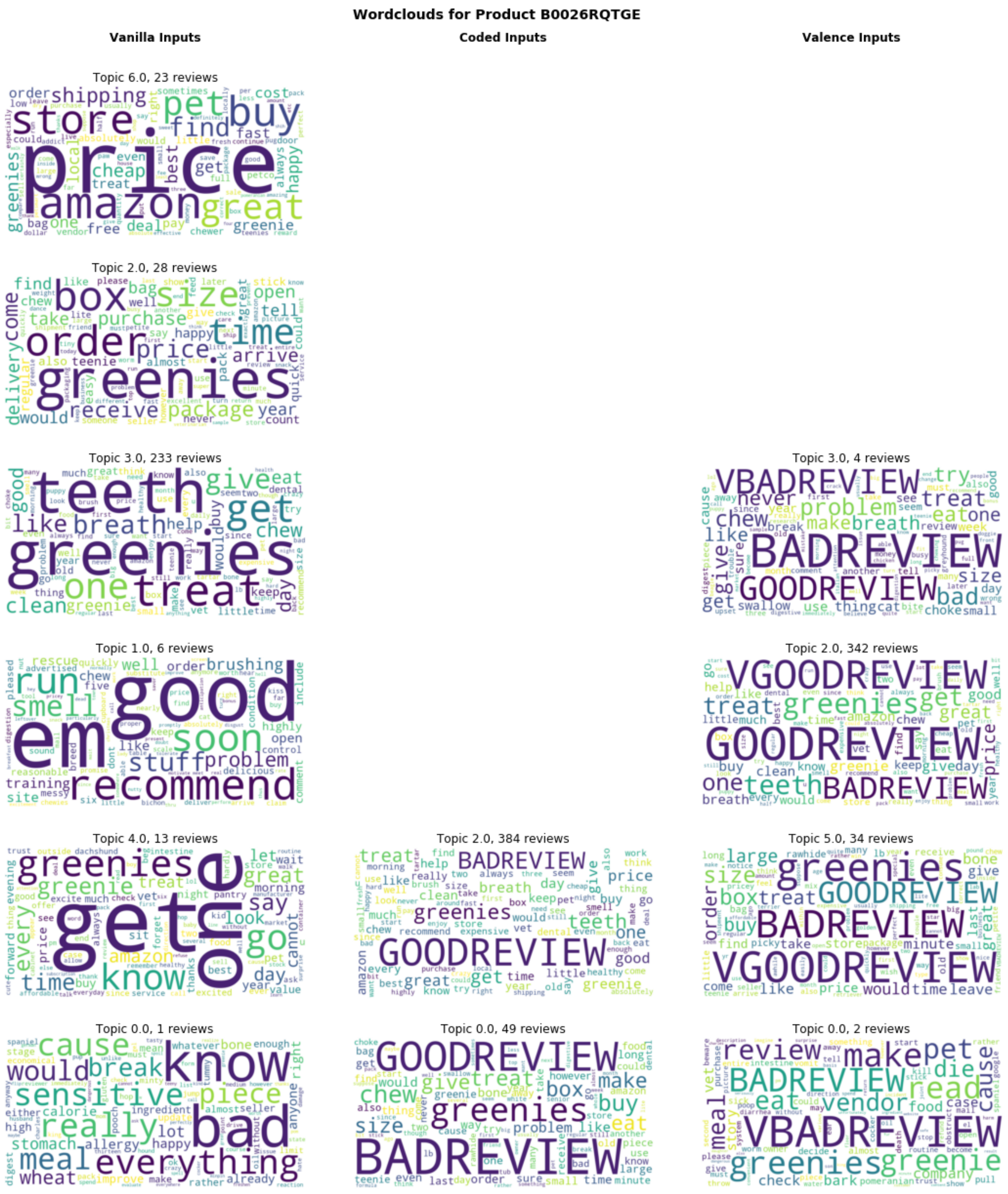
Recall that having developed a topic model for each product, meaning a group of topics and the topic term distributions corresponding to each, I assigned every review for that product to the topic most likely to have generated it.

The key input driving the visual is the frequency dictionary, which determines what words appear and how big they are in the wordcloud. For this project, there would be two ways to construct the visual:

1. Topic term distribution wordclouds - this would be a visualization using the topic term distribution as an input. That is, the vector of probabilities that tells us how likely the word is to appear in a document generated by the topic. These visualizations would have nothing to do with the actual text of the reviews tagged to the topics - they would purely show the makeup of the topic.
2. Review term frequency wordclouds - these visualizations would visualize what words actually appeared in the reviews assigned to a given topic, using the frequency counts of those words to drive the visual.

Because this project was meant to get at practical applications of the topic modeling, we are really interested in topics that actually seem to have generated a lot of reviews. So, including the review counts for each topic is an important component of the visualization, and of deciding how to use the outputs of the model. The way I've chosen to construct the wordcloud visuals is to incorporate those review counts, but to use the frequencies of the words that actually appeared in the reviews to drive the visual, not the topic term distributions from the model. One additional part of this visualization is that I applied a threshold of 0.7 probability in order to tag a review to a topic - that is, only reviews that have a probability of at least 0.7 of having been generated by a topic are counted in the frequency dictionaries. This excludes a large number of reviews that seemed to have a more mixed identity, with two or three topics coming in around 0.3, 0.4 in their topic mixture values.

Here's an example of the text encoding working really well:



This product is a dog treat called Greenies. We have a wordcloud for each topic in the topic model created for each of the three input text types (vanilla, coded, and valence coded). Only wordclouds for topics that had at least one review tagged to them are shown, and the number of reviews tagged to each topic is also displayed in the chart.

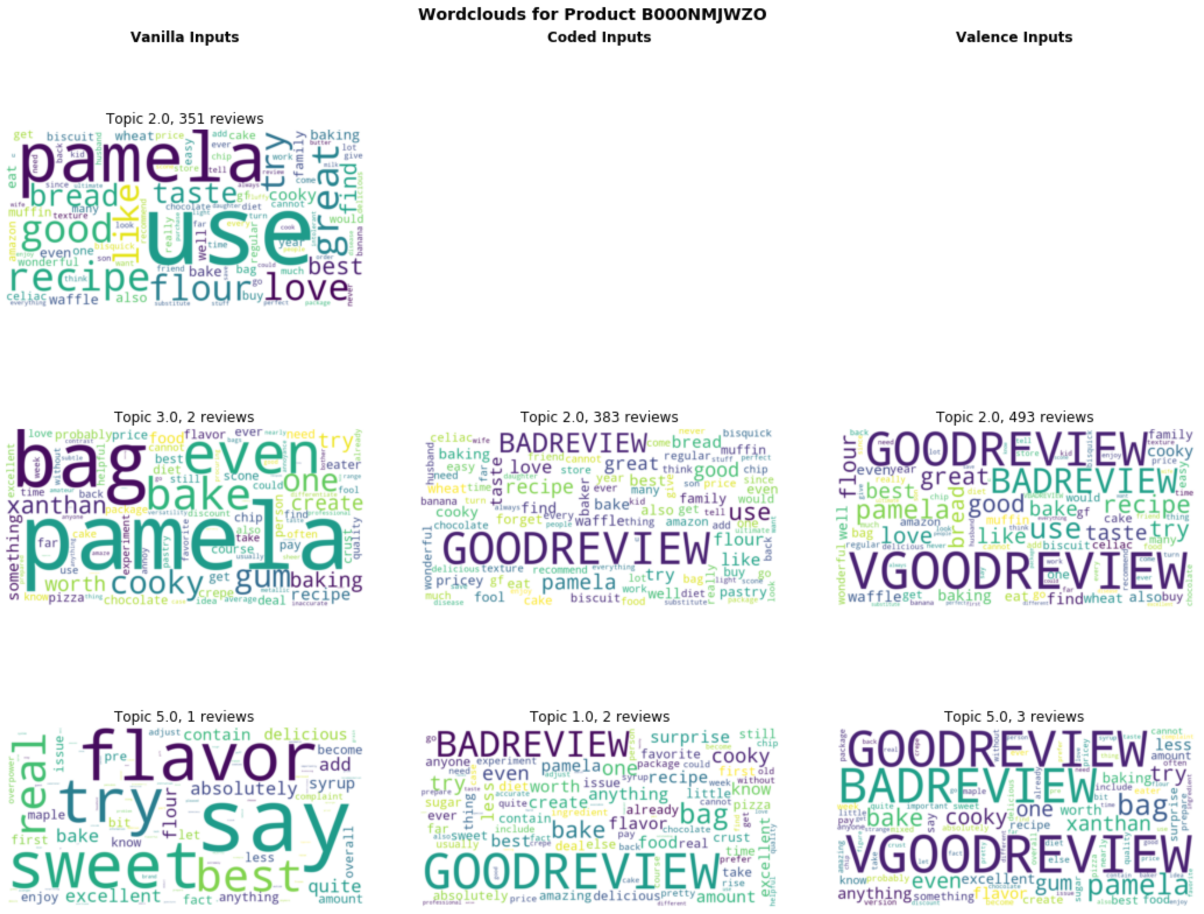
First, look at the vanilla wordclouds. Because these wordclouds aren’t overwhelmed by the codewords, they are a nice way to get a sense of the kinds of things people talked about in the reviews for this product. Note that by far the biggest variety in topics is shown for the vanilla reviews. This doesn’t necessarily mean that the topic models found fewer topics - it just means that there was more variety in what topic was most likely to have generated the reviews. This makes sense and is exactly the effect the codeword insertion would be expected to have - it was a common result across products. There is a clear set of reviews related about the price and how it compares on Amazon vs. at the pet store. Another set of reviews is obviously related about the purchasing experience, the packaging, how quickly the order arrived, etc. And finally there is a set of reviews (the largest) that is probably simply explaining why people bought the product and what their experience was using it with their dogs.

Looking the wordclouds for the coded inputs, it appears that the simple encoding may have effectively split reviews that were mostly positive from reviews that were more negative, but there’s not a whole lot more information to be gained from a glance at these wordclouds.

It’s the wordclouds for the topic model using the valence encoded input text that are really interesting and show exactly the kind of result I had hoped to achieve. The wordclouds for topics 2 and 5, which seem to have generated most of the reviews, show two of the broad topic themes from the vanilla wordclouds (user/dog experience and purchasing experience, respectively) and also give some insight into the overall sentiment of reviews related to those topics. However, there are two other topics that only generated a very small number of reviews each but are very striking in terms of their wordcloud content. Topic 3 is dominated by negative codewords, which is unusual, and includes the words "problem," "bad," "choke," and "swallow." This sounds like reviews where the customer is reporting a bad outcome with their pet and the treat. Topic 0, with just two reviews, is even more unusual because it is dominated by the "VBADREVIEW" codeword. There aren't many of the very negative codewords that ever appear in the reviews so this stands out right away. Then you see the word "die" along with "vendor," "company," and "obstruct.” This is describing some very bad outcomes indeed, and note that none of these words showed up anywhere in the wordclouds from the other input text sets. In this setting, the valence codewords were really effective at reinforcing the co-occurrence of some very negative words, which produced some distinct topics that had a very high probability of having generated the reviews that contained those words.

The results for this product illustrate a really interesting potential use case for topic modeling - as an early warning or detection system for quality or safety issues with a product. The number of reviews for each of these topics is very small, but the content of them looks like it is quite serious - having ongoing topic modeling analysis of online reviews for a product could allow the vendor to detect and respond to these issues as they start to be reported by consumers and even understand if there is a widespread issue emerging.

There are a number of products where the results were pretty similar to the Greenies topic modeling, though none were as striking. Here's an example where the outputs of topic modeling were not very interesting at all.



This product is a gluten-free baking and pancake mix, brand name Pamela's. With all three of the input text types, there was one topic dominating the review set, and the wordclouds aren't very interesting at all. Pretty much the only thing we might guess from glancing at these wordclouds is that people are using the mix for more than just pancakes, and that they like it. However, given the fact that the results are consistent across the input types, this might be a reflection of the fact that the reviews really are just very consistent and homogenous, and not a reflection of the model performing poorly in this setting.

Another thing to note is the fact that the word "pamela" looms pretty large here. Given that it's the brand name of the product, that's not a word that's going to give a lot of information, and to improve the topic model I would want to remove it. For whatever reason, the token removal parameters in grid search didn't remove this word (perhaps the coherence of the model was higher when it was included, which is pretty likely).

Reflection

There were many aspects of this project that were challenging and enjoyable. First, I had not previously done any work with NLP techniques or taken any classes that covered these approaches, so I had to first spend a lot of time learning about the methods that are out there. Additionally, there aren't as many resources available for independent learning that are made for learners without a deep academic background in this area. I would say this is in contrast to other popular applications of machine learning like computer vision and fraud detection where beginner-friendly, light-on-math MOOC videos and blog posts or tutorial Kaggle notebooks abound. That's not to say they don't exist - there just aren't as many, and I found I had to really dig into the academic papers to feel like I was getting a grasp on the LDA algorithm and the subtle differences between different coherence metrics, for example.

The other big challenge was determining how to benchmark and evaluate model performance when using an unsupervised learning technique. I will admit that when I started the project I was more excited about learning how to analyze text data and see what I could get to come out of the models than I was in finding ways to judge how successful those models were, but reading about the different performance metrics that researchers have developed was an interesting piece of the research process. The further I got into the project, the more I thought about practical ways the outputs could be used and how important it would be to have a way to evaluate whether the model was performing well. This led me to think not just about the ways model performance is measured in academic research (e.g., perplexity, coherence, etc.) but also about how to think about the criteria for determining when and how to use the outputs of the model. For example, early in my research I read a blog post[11] from a data scientist at Square talking about how she had used topic modeling on the data captured in the free-text section of their customer feedback form, and how the outputs of that work could be used to give businesses a summary of customer feedback without them needing to read every single free-text response. This helped me think about would be needed in order to decide what information to pass back to a business, especially if the feedback is negative.

Overall, the valence encoding didn't make a difference in an striking or consistent a way as I had originally hoped, although there were a few great examples of it working exactly the way I had expected, and that was really fun to see.

My conclusion is that this kind of approach could be very useful and effective in the settings I imagined it being applied, but it requires a lot of tweaking and tuning for each individual application, down to the individual product that is being reviewed, in my case. General settings might not produce anything very interesting or useful, and if there is a lot of effort required to develop an effective model for a single product or business, it might not be practical to do.

[11] Alyssa Wisdom. 2017. "Topic Modeling: Optimizing for Human Interpretability." https://medium.com/square-corner-blog/topic-modeling-optimizing-for-human-interpretability-48a81f6ce0ed

Improvement

In thinking about how to improve the implementation in this project, most of the ideas I have are related to making it more specific to the context of the text data being analyzed. From what I have read and my own experience in doing this project, making NLP models more general doesn't improve their performance.

That being said, here are some things I would try if I were actually trying to implement this for a business:

- Conduct a full review of the sentiment and valence-aware sentiment dictionaries and customize them for the application. I observed in several cases that there were words getting a sentiment tag that was not really appropriate for the context - such as "cold" and "bite," both of which were tagged as negative sentiment words. When you are looking at reviews of dog toys and of pre-brewed coffee products, not only are these words not necessarily (and indeed probably not) negative, they are also very common. I did remove a few of these words that I noticed in examining the results but I think that creating sentiment dictionaries fully customized to the context could improve the results a lot. Specifically, this would mean making sure that words appear or don't appear as appropriate in the sentiment dictionary, and that they have the appropriate value if they appear in the valence-aware sentiment dictionary so that they only receive a "VBADREVIEW" or "VGOODREVIEW" codeword if it's really appropriate.
- Try using even more specialized codewords for key topics that could emerge in reviews. For example, the vendor selling the dog treats, seeing that some customers had complained about the treats being dangerous to their dogs' health, might be interested in keeping a close eye on that topic. I would like to try creating a dictionary that contained key words related to safety and health (e.g., "choke," "die," "blockage," "dangerous," etc.) and using the same methodology I used to insert the sentiment codewords to insert a codeword about safety into a review whenever one of these. As another example, a clothing retailer could implement word encoding related to fit vs. quality of fabric vs. customer service.
- Do topic modeling separately over reviews that are grouped by their star rating. If the objective is to understand what people are saying when they liked the product vs. what they are saying when they were unhappy, this is could be a very effective way to do it. The problem is that in the context of online reviews there are usually relatively few negative examples to work with - it might not be until a product has a body of thousands of reviews that you would have enough negative reviews to work with, and until then it would probably be more effective just to read through them!
- Manually review and tweak the tokens to be removed before topic modeling. As seen in the example of the Pamela's baking mix, sometimes a word was left in after the grid search process, and through manual review we would recognize that although this word

occurs frequently, it’s essentially a context-specific stop word in that it’s not adding any information that would help form a meaningful topic.