Predicting Ratings of Amazon Products Using Review Texts

A. Agrafiotis, I. Chios, M. Z. Khairullah, G. Kyziridis, M. Tsiaousis

July 16, 2018

1 Introduction

Consumer reviews can be a valuable source to drive potential consumer's decisions and buying behaviors. The opinions can not only tell if a product is good or bad, but can also tell about the personal experience one had with a wide range of products. These reviews are accompanied with a rating score from 1 being the worst to 5 being the best. In this project, we use the Amazon US product reviews data provided by He and McAuley [2]. The assumed value and diversity of these opinions raises a question of interest to us: Can we predict the rating (1-5) of a product based on consumer's opinion (text)? To answer this question, we approach the problem from a Natural Language Processing (NLP) perspective, where we apply different essential text preprocessing techniques that are suitable to answer our question of interest. Furthermore, we convert the preprocessed text to Term Frequency - Inverse Document Frequency (TF-IDF) weights to be used in the prediction models. Given that NLP needs huge computation power, the preprocessing task is implemented in a parallel manner using `Dask`, a Python library for distributed and parallel computing, while the computation of TF-IDF weights is performed in a distributed fashion using the Distributed ASCII Supercomputer 3 (`DAS-3`) of Leiden University's Data Science Lab. The models used are Linear Regression, Logistic Regression, Naive Bayes, and Multilayer Perceptron (MLP) Neural Network. In the next section, we will discuss the data set in detail, along with the goal of this project and descriptive statistics.

2 Data

2.1 Amazon Data & Project Goal

The data set contains Amazon US product reviews (24 product categories) from 1996 to 2014. The size of the data is 18GB compressed and ≈ 58GB uncompressed in JSON format. The JSON file contains 9 fields, see figure Figure 1 for a review sample. These fields are:

1. `reviewerID`: unique consumer ID
2. `asin`: unique product ID
3. `reviewerName`: username of the consumer
4. `helpful`: an array of the count of helpful and non-helpful votes
5. `reviewText`: review text written by the consumer
6. `overall`: rating of the product in scale (1-5), as integer
7. `summary`: summary of the review
8. `unixReviewTime`: date of the review in unix format
9. `reviewTime`: date of the review in (MM/DD/YYYY) format

```
{
  "reviewerID": "A2SUAM1J3GNN3B",
  "asin": "0000013714",
  "reviewerName": "J. McDonald",
  "helpful": [2, 3],
  "reviewText": "I bought this for my husband who plays the
piano.  He is having a wonderful time playing these old hymns.
The music  is at times hard to read because we think the book
was published for singing from more than playing from.  Great
purchase though!",
  "overall": 5.0,
  "summary": "Heavenly Highway Hymns",
  "unixReviewTime": 1252800000,
  "reviewTime": "09 13, 2009"
}
```

Figure 1: Sample of a single review.

For the purpose of this project, we are only interested in two fields, `reviewText` and `overall`. Our main goal is to predict the rating of a product based on the text given in the review. In the following subsection, descriptive statistics of the data set are provided.

2.2 Descriptive Statistics

In this part of the project we conducted descriptive statistics and we marked that there are in total 82,677,140 ($\approx$ 82M) reviews. The ratings are represented as integers in the scale 1-5, where 1 is the worst and 5 is the best. Moreover, the mean value of the ratings is 4.16, with a standard deviation of 1.26. The data contained a total of 9,003,960 unique products and 20,896,479 unique consumers (reviewers). Furthermore, we investigated the mean value of the ratings for each month over the years, and the mean value of the ratings for each year, as illustrated on the left and right-hand side of Figure 2, respectively.

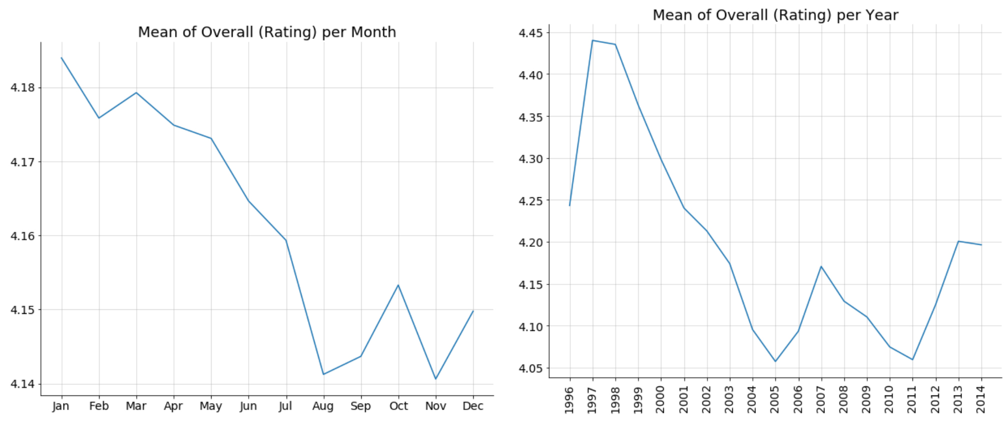

Figure 2: Left: Mean value of the ratings for each month from 1996 to 2014. Right: Mean value of the ratings for each year.

As we can see from Figure 2, the mean value of ratings experiences a moderate fluctuation between January and April and then keeps decreasing until September. Later, it peaks in October to fluctuate again between October and December. The minimum mean rating per month is $\approx$ 4.14, where the maximum is $\approx$ 4.19. Furthermore, the plot of the mean value of ratings per year faces an increase between 1996 and 1999 to drop between 2000 and 2005. Later on, it fluctuates between 2005 and 2014. The minimum mean rating per year is $\approx$ 4.05, and the maximum mean rating per year is $\approx$ 4.45. Note that the differences between mean values of ratings per month or per year are very small as all values are close to 4. Therefore, we do not consider these differences to be very important.

After inspecting mean value of rating, we are interested in knowing if the distribution of the ratings is balanced. Figure 3 shows that most of the ratings are 4 and 5, with approximately 40 million 5's, which is around 50% of all the reviews. In the next section, we discuss the data preprocessing techniques used.

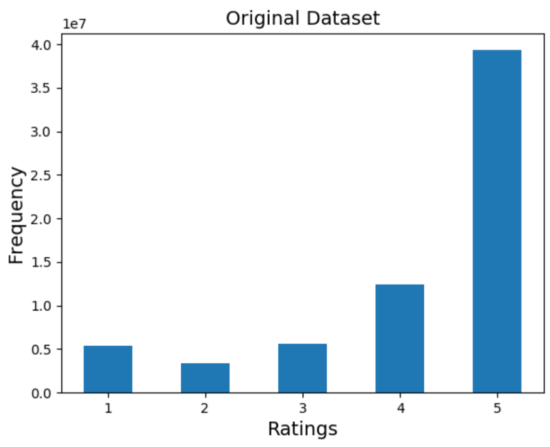

Figure 3: Distribution of Ratings.

3 Data Preprocessing

3.1 Natural Language Processing

This section concerns the initial and essential procedure of preprocessing linguistic data. NLP needs huge amount of computation power due to grammatical peculiarities. There are various methods and algorithms suitable for processing words and phrases according to a desirable goal. In this project, we use standard methods for linguistic data preprocessing in order to produce "clean" reviews from our dataset. The following steps describe the operation of text preprocessing:

Preprocessing Operation

1. Remove non-letters and numbers
2. Convert each word to lower case
3. Split sentences into words (Tokenization)
4. Remove "stop words"
5. Stemming

3.2 Preprocessing using Dask

`Dask`[1] is a relatively new open source Python library. By leveraging the already existing Python eco-system, `Dask` provides a familiar API for parallel and distributed computing. It is a project of Continuum Analytics[2], and it scales up computation to a cluster or to a single machine using multiple cores. By utilizing blocked algorithms and applying task scheduling, it provides a robust Python framework for distributed and parallel computing, using precompiled algorithms for general purpose data analysis. Furthermore, `Dask` has an admirable trait; it handles data not fitted in RAM, which converts it to an essential tool for data analysts.

`Dask` includes already known Python libraries such as `numpy` and `pandas`, which are running in parallel through this framework. It has many modules for specific functions regarding machine learning and big data analytics. The core idea behind `Dask` is the task graph creation and execution, which relies on the general idea of Map-Reduce. Users, using familiar API of the aforementioned scientific libraries, create task graphs which are later executed. `Dask` includes its own scheduler responsible for task graph execution either on many CPU cores or in many cluster nodes (different machines). The scheduler is also capable of handling the communication between the nodes by sending and receiving requests for data-variable availability, and sending different tasks to the nodes to be executed. Thus, the client (user) creates task graphs using API similar to those of well known scientific libraries, and these graphs are submitted to the scheduler who distributes different jobs to different workers (CPU cores or cluster-nodes). Later, the workers communicate peer-to-peer by sending requests to the scheduler until the task graph execution is completed.

`Dask` was used to perform parallel computation of data preprocessing on 20 cores of the `Mithril` super-computer, which is part of the Data Science Lab of Leiden University. It consists of 64 Intel Xeon E5-4667v3 CPUs 2.00GHz (128 threads), 1TB RAM, and 9TB SSD. Dask helped us to run expensive computations in limited time. Figure 4 visualizes a task graph example of the preprocessing steps described in subsection 3.1, where a Python function implementing steps 1 to 5 were mapped to the review texts. It is an example of a task graph, where the data set was split into 5 chunks and the preprocessing function was mapped onto each one of them. In Figure 4, it can be observed that all threads are exactly the same and they are not connected

[1]`https://dask.pydata.org/en/latest/`

[2]`https://www.anaconda.com/`

since there is no reduce part in this function. So each thread represents a chunk's route from loading the data throughout mapping the function onto each chunk. In our case, data set was split into 320 chunks of 100MB each. The `Dask` scheduler was responsible for distributing different threads to the 20 CPU-cores (workers). The whole process lasted around 3 hours and 10 minutes.

4 Feature Extraction

4.1 Terminology

As features for the machine learning models, we use the TF-IDF matrix of most frequent words, derived from the preprocessed reviews of section 3. TF-IDF stands for Term Frequency-Inverse Document Frequency, and it is calculated for each term that occurs in a document, in this case, a review text. It is a form of term weighting where important words, that is, words characterizing the document obtain high values of TF-IDF, and very common words obtain low values. In this paper, we use the log normalized Term Frequency given by

$$TF(t,d) = \begin{cases} 1 + \log tf_{t,d} & \text{if } tf_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

where $tf_{t,d}$ is the term frequency (count) of term t in document d. We add one so as $\log(1)$ does not produce $TF(t,d)$ of 0. We opt for log normalized TF instead of raw TF, since the relevance or importance of a document does not increase proportionally with term frequency. In other words, document d_1 with x occurrences of term t is not necessarily x times more relevant or important than document d_2 with just one occurrence of term t.

The normalized IDF is given by

$$IDF(t) = \log(\frac{N}{df_t}) \tag{2}$$

where N is the total number of documents, and df_t is the document frequency of term t, that is, the number of documents in which t occurs. IDF assigns high values to terms that occur in less documents and low values to those that occur in many documents. For example, for a term t that occurs in all

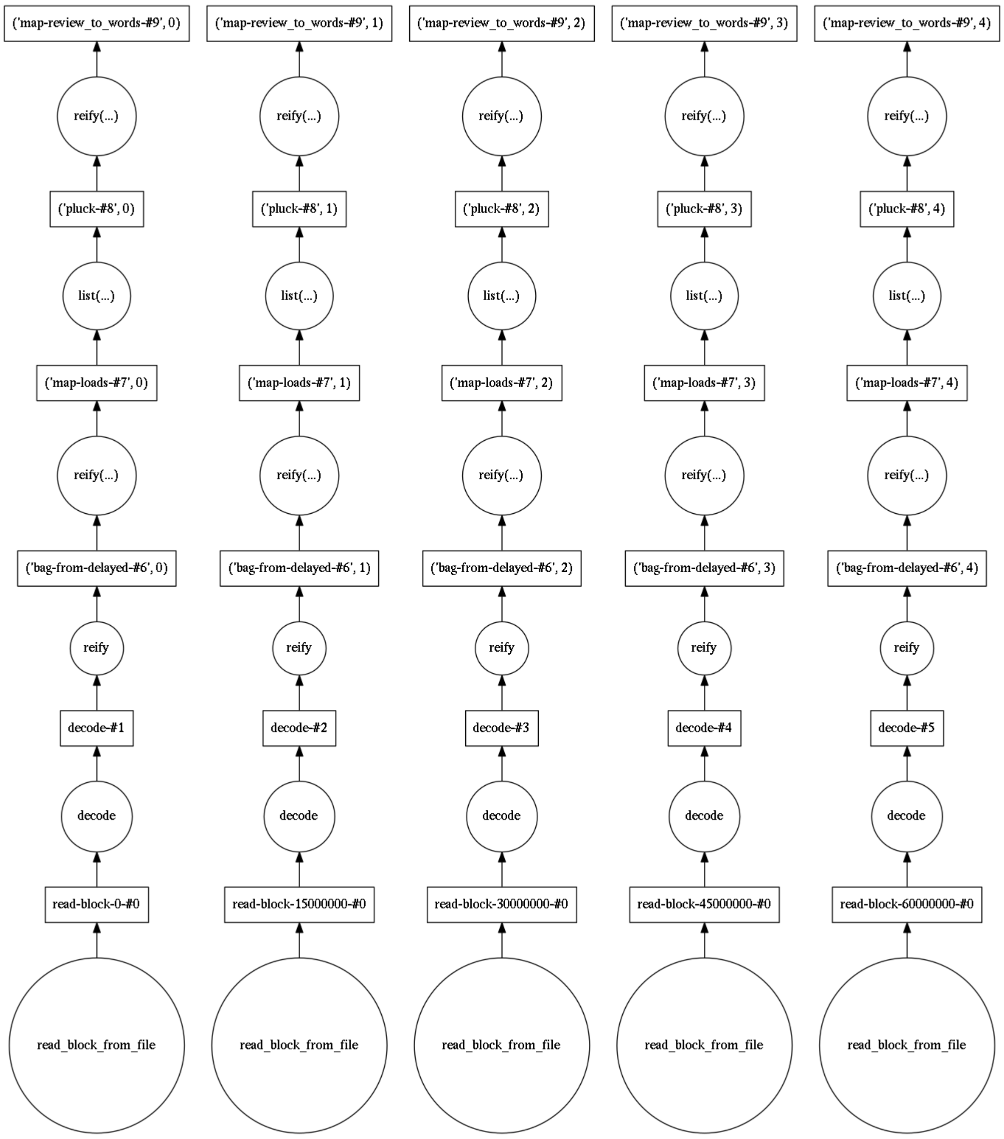

Figure 4: A task graph example, implementing a function that performs the preprocessing steps of subsection 3.1. The whole computation process, which is split in 5 chunks runs either distributed (on a cluster), or in parallel (in many cores).

N documents, $IDF(t) = \log(1) = 0$.

TF-IDF is the product of $TF(t,d)$ and $IDF(t)$, that is

$$TF\text{-}IDF(t,d) = TF(t,d) \cdot IDF(t) = (1 + \log tf_{t,d}) \cdot \log(\frac{N}{df_t}) \qquad (3)$$

TF-IDF combines $TF(t,d)$, which weights frequent terms in a document more heavily, and $IDF(t)$, which assigns more weight to rare terms with respect to document frequency. For example, we expect the term t_1="the" to have low TF-IDF, since t_1 is expected to have IDF close to 0. On the other hand, the term t_2="sports" would have high TF-IDF for a document related to sports, since TF would be high for this specific document, and additionally its IDF would also be high, because its document frequency is expected to be low in a document collection with a wide variety of topics (including sports).

The feature matrix of TF-IDF values is of size $N \times M$, where N is the total number of reviews (82,677,140) and $M = 5,559,501$ is the number of unique terms of the preprocessed reviews. Such a matrix would be of very large size, and would require high computational complexity to calculate all $N \cdot M$ TF-IDF values. Last but not least, a model with M predictor variables would certainly result in overfitting. To avoid these problems, we calculate a smaller TF-IDF matrix of size $N \times M'$, where M' indicates the M' most frequent terms across the whole data set, with $M' << M$. We developed several Linear Regression models in order to predict the rating of the review text on a small random sample of the data set with varying values of M'. We found that $M' = 5000$ produced the best results in terms of Root Mean Squared Error (RMSE). Thus, the TF-IDF matrix will be of size $N \times M'$ where $M' = 5000$, corresponding to the 5000 most frequent terms across the whole data set.

4.2 Distributed Computation of TF-IDF

4.2.1 Introduction and Description of Computing Machines

We describe the distributed computation of TF-IDF matrix in three parts. In the first part, hashing is used in order to encode the terms and reviews, in other words, a unique integer is assigned to each review and each unique term. Then, the resulting data set is split into chunks. In the second part, the 5000 most frequent terms (maximum features) are selected. In the third part, we describe the computation of $TF(t,d)$ and $IDF(t)$.

The `DAS3` distributed system of Leiden University is used for the computations. It consists of 32 dual-CPU 2.6 GHz AMD Opteron DP 252, each of them having 4GB memory and 400GB HD space. The master node has a dual-CPU/dual-core 2.4 GHz AMD Opteron DP 280, 16GB of memory, and an additional RAID6 10TB storage system[3]. In the whole process of TF-IDF computation, `Mithril` is used for tasks that require large memory and are in nature difficult to perform in distributed fashion, such as splitting the original data set into chunks.

4.2.2 Encoding Reviews and Terms with Hashing

Hashing is performed in the preprocessed reviews of section 3, that is, each review and each unique term is assigned to an integer. To achieve this, we loop through the review texts and we maintain two counters, the first one for the reviews, and the second one for the terms. When a new review or a new term is encountered, the respective counter is incremented, and the Review ID or Term ID is assigned to the value of the counter. At the same time, regardless of whether the review or term has been encountered in a previous iteration, the combination (Review ID, Term ID) is stored in an array of two columns. The first column corresponds to the codes or IDs of the reviews, while the second column corresponds to the codes or IDs of the terms. Thus, the resulting array is in "long" format, as illustrated in Table 1. For the rest of this paper, we will refer to the data set depicted in Table 1 as `codes`. In total, 5,559,501 unique terms were encoded.

The `codes` data set is around 28GB. We decided to split it into 10 chunks of roughly 2.8GB each, with no overlapping of the same review across chunks. The chunks were then transferred to 10 nodes, specifically, nodes 03-09, 12-14. The aforementioned nodes were used for all computations. Both the encoding and splitting tasks were performed on `Mithril`.

4.2.3 Computation of Maximum Features

To find the 5000 most frequent terms, the frequency of each term across the whole `codes` data set has to be computed. Thus, we calculate the frequencies of the terms for each chunk in distributed fashion (map operation), using the `bincount()` function of `numpy`, and we store the results (frequency arrays) in each node. In the reduce operation, we sum the frequency arrays in the master node, obtaining a single array of frequencies. Then, the terms are

[3]`https://www.cs.vu.nl/das3/sites-lu.shtml`

Review ID	Term ID
0	0
0	1
0	2
⋮	⋮
50	375
50	27
50	375
50	568
⋮	⋮

Table 1: 2-dimensional array of (Review ID, Term ID) containing the codes/IDs of each review and term in "long" format.

sorted with respect to frequency and the 5000 most frequent terms are obtained. Next, these terms are looked up in the chunks of `codes` data set in each node, preserving only the rows where such a term occurs. We will refer to the resulting data set containing the combination of (Review ID, Term ID) of maximum features (most frequent terms) as `codesmf`. Each node contains a chunk of `codesmf` of roughly 2.5GB each.

4.2.4 Computation of TF and IDF

Before proceeding to TF computation, we need to describe the data structure in which, the latter will be represented. Representing the TF-IDF matrix of size $N \times M'$ as a regular matrix would require an enormous size of disk space. Even if we used only integers of 2 bytes (-32768 to 32767), we would require 82,000,000·5,000·2 $\approx$ 820GB. Due to the fact that most elements of the matrix are 0, that is, most of the words are not contained in each review, the matrix can be represented in a sparse format. Specifically, we can store only the indexes and values of non-zero elements, so as the required space is greatly reduced.

The indexes of the sparse matrix are already defined; they are the unique review and term IDs of the `codesmf` data set. Therefore, only the (normalized) frequencies $TF(t,d)$ and (normalized) inverse document frequencies $IDF(t)$ have to be computed. Multiplying the values of $TF(t,d)$ and $IDF(t)$ yields the $TF\text{-}IDF(t,d)$ values, meaning that we have all the necessary information to construct the TF-IDF sparse matrix. In the following paragraphs, the computation of $TF(t,d)$ and $IDF(t)$ is described in detail.

The computation of $TF(t, d)$ is performed on each chunk of `codesmf` in distributed fashion. Combining the functions `cumsum()` and `bincount()` and applying them on the Review IDs of each chunk of `codesmf`, the starting and ending pointers of each review are obtained. Hence, instead of looping through each element of `codesmf` to calculate $TF(t, d)$, the looping takes place on the aforementioned pointers. This greatly speeds up the looping procedure. Looping across those pointers, we extract each Review ID, along with its corresponding Term IDs. We then initialize a dictionary and perform a second loop through the Term IDs. When a term is encountered for the first time, the dictionary creates a key for the term and assigns the frequency of one to the key. When the term is encountered more than once, the frequency value of they key corresponding to the term is incremented by one. When the loop of Term IDs finishes, the dictionary values are converted to an array. We have calculated the raw frequencies $tf_{t,d}$ for a specific Review ID, but some of the Term IDs are duplicated, since some terms are included more than once in the review text. Thus, for each Review ID, the unique Term IDs which represent the indexes for the terms in the sparse matrix have to be calculated. Then, we overwrite the values of Term IDs in each chunk of `codesmf` with the unique Term IDs. Overwriting the values in the chunks of `codesmf` provides two advantages. Firstly, there is no need to create a new array and dynamically allocate rows to store the indexes. Secondly, it is memory efficient, since storing both the chunk of `codesmf` (around 2.5GB) and the arrays of indexes would overflow memory. Nevertheless, the overwriting procedure requires that the number of unique combinations of (Review ID, Term ID) is known. Thus, prior to calculating $tf_{t,d}$, an additional scan has to be performed on the original chunks of `codesmf` in order to compute the number of unique (Review ID, Term ID) combinations.

The result of the computation task are 10 chunks similar to `codesmf` of 1.9-2GB each, containing only the unique combinations of (Review ID, Term ID) that comprise the indexes of the sparse matrix. Furthermore, 10 arrays of about 500MB each contain the (raw) $tf_{t,d}$ values of each node. Finally, we can apply the normalization of Equation 1 to calculate the (normalized) $TF(t, d)$ values.

Another alternative to looping through the Term IDs for each Review ID would be to use the `bincount()` function to calculate the frequencies for each Term ID. We found that this is actually a very slow process, requiring several hours to calculate the $tf_{t,d}$ values and overwrite the original `codesmf` chunks. We believe that this is due to the fact that the `bincount()` function is called too frequently (around 82 million times, once for each Review ID). Therefore, we believe that the function is considerably faster than a for loop when it comes to calculate the frequencies of one large array, but slower than a for loop when called multiple times for many small arrays.

$IDF(t)$ values are calculated by looping through the pointers of the original `codesmf` chunks, as described in the $TF(t,d)$ calculation. As in the case of $TF(t,d)$, $IDF(t)$ computation is performed in distributed fashion. As a first step, a dictionary is initialized. For each Review ID, we iterate through the unique Term IDs, creating a key when a term is encountered for the first time and setting the frequency to one. When the term is encountered more than once, the frequency value of the key corresponding to the term is incremented by one. The result are 10 dictionaries of the 5000 most frequent terms (maximum features), one for each node, where the dictionaries' values are the document frequencies df_t. In the reduce operation, which is performed on the master node, the values of the 10 dictionaries are summed. Finally, we apply the normalization of Equation 2, and the dictionary is converted to an array.

Having calculated the indexes of the sparse matrix, along with the $TF(t,d)$ and $IDF(t)$ values, we have all the necessary information to create the sparse matrix. First, the 10 chunks of indexes and $TF(t,d)$ values are combined, and a sparse matrix is created. The sparse matrix is multiplied with the array of $IDF(t)$ values, yielding the $TF\text{-}IDF(t,d)$ sparse matrix. The sparse matrix has size 82,644,191$\times$ 5000. The number of rows is less than the original number of reviews (82,677,140). This is due to the fact that firstly, some of the rows were removed in the preprocessing part of section 3 because the reviews contained only stop words, and secondly, there were reviews comprised of terms not included in the 5000 maximum features. The sparse matrix is around 20GB in uncompressed, and 5GB in compressed format, respectively. Table 2 illustrates the times for each task of the distributed TF-IDF computation. The most expensive task was the $TF(t,d)$ calculation that took 22 minutes and 4 seconds, followed by the look-up operation of maximum features in the original `codes` chunks that produced the `codesmf` chunks. This task lasted for 17 minutes and 52 seconds. Finally, the third most expensive task was the $IDF(t)$ calculation, with 15 minutes and 17 seconds. In total, the whole computation process lasted for 2 hours and 40 minutes.

Task	System	Time
Encoding	Mithril	1 hr, 38 min
Splitting into chunks	Mithril	1 min, 15 sec
Bincount (max features)	DAS3 (Map)	5 min, 27 sec
Sum and sort freqs.	DAS3 (Reduce)	2 sec
Filter max features from `codes` data set	DAS3 (Map)	17 min, 52 sec
$TF(t,d)$	DAS3 (Map)	22 min, 4 sec
$IDF(t)$	DAS3 (Map)	15 min, 17 sec
Sum $IDF(t)$	DAS3 (Reduce)	0.11 sec
		2 hr 40 min (Total)

Table 2: Times for each task of the distributed TF-IDF computation.

5 Predictive Modeling

5.1 Introduction

Our main goal is to predict the overall rating that a certain user gave to a product based only on the review text. In order to achieve this, we built and experimented with four prediction models. We decided to build a Linear Regression (LR) model, and three classification models, namely, Multinomial Naive Bayes (MNB), Multinomial Logistic Regression (MLR), and Multilayer Perceptron (MLP). The three classification models convert their (classification) output to a continuous value (regression). The reason behind this approach is that we may have five discrete classes that we want to predict, but they are strongly correlated with each other, there exits a natural ordering between them, and they are not necessarily equidistant from each other. Finally, one consumer's rating value x might not be equivalent to the rating value x of another consumer. For example, the rating of 1 of consumer c_1 with review text r_1, is not necessarily equivalent (in terms of closeness) to the rating of 1 of consumer c_2 with review text r_2.

The models take as input the TF-IDF matrix described in section 4, which is of size 82,644,191$\times$5000. All three classification models output probabilities for each class, and the instance belongs to the class with the highest probability. In order to convert the classification output to a regression one, we calculate the sum of the five ratings (1 to 5), weighted by their respective probabilities. Specifically, the converted regression output is given by

$$\hat{y}_i = \sum_{j=1}^{5} P(j|d_i) \cdot j$$

where $P(j|d_i)$ is the probability that instance i belongs to class j, $j = 1, 2, \ldots, 5$, given the review text d_i. Prediction values are then truncated to 1 or 5, in case they are smaller than 1 or larger than 5, respectively. Prediction values in the range [1, 5] are rounded to the nearest integers. The models are evaluated using the Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE), given by

$$RMSE = \sqrt{\frac{\sum_{i=1}^{m}(y_i - \hat{y}_i)^2}{m}}, \qquad MAE = \frac{\sum_{i=1}^{m}|y_i - \hat{y}_i|}{m}$$

where m indicates the number of instances in the test set. We contrast the developed prediction models with a baseline model, which is the mean rating of the reviews in the training set. The mean value is then used to calculate the RMSE and MAE for the baseline model on the test set.

Python and `scikit-learn` were used to train LR, MNB, and MLR. The `Latinum` super-computer of Leiden University's Data Science Lab was used for the computations. It consists of 16 Intel Xeon E5-2630v3 CPUs 2.4GHz (32 threads), 1.5TB RAM, and 3TB HD space. We used 4 cores (8 threads) for training LR and MLR. A parallel version of MNB is not provided in `scikit-learn`.

5.2 Linear Model

Linear regression is a standard method used for numerical predictions, and describes the relationship between a response variable and a single or many explanatory ones. It is mainly based on a linear approach which tries to calculate an optimal prediction line according to the minimized mean squared error function. Y plays the role of response variable, and in this case, it reflects five values (1 to 5, integers) of Amazon product ratings. Predicted values of linear models represent single points on the regression line described by the following formula:

$$\hat{y} = \alpha + \beta_i x_i, \ i = 1, 2 \ldots, p, \ \hat{y} \in \mathbb{R}$$

where α is the intercept of the regression line, and β_i $i = 1, 2 \ldots, p$ is the regression coefficient of regressor x_i. Intercept α represents the distance of the line from the origin in Cartesian representation. In other words, α is the value of $\hat{y}$ when all $x_i = 0$, and coefficient β_i is the angle between the regression line and the $y = 0$ one. Therefore, linear regression model outcome is the estimated average values of Gaussian distribution of Y given the X_i regressors. The quality of the model is measured with *adjusted* R^2, which represents the percentage of variance in the outcome variable explained by the model (the regressors).

$$R^2 = \frac{Explained\ variance}{Total\ variance} \qquad R^2_{adj} = 1 - \left[\frac{(1-R^2)(n-1)}{n-k-1}\right]$$

where n represents the number of observations with k degrees of freedom.

The peculiarity of this approach is that the estimated model has to meet the following assumptions in order to be considered reliable and robust:

- Independence of observations
- Normality of residuals
- No multicollinearity
- No autocorrelation
- Homoscedasticity

In our case, regressors X were represented by vectors of TF-IDF values in $\mathbb{R}^{5000}$ space, that is, X: $x_i = 1, 2, \ldots, 5000$. Due to the large amount of regressors, (some of) the aforementioned assumptions might not hold. Our intuition is to test if a naive model like Linear Regression is able to generate good predictions according to the response variable. The results and an analysis of the model's assumptions are described in subsection 5.6

5.3 Multinomial Naive Bayes

Another model we used to predict the rating though the review text was the Multinomial Naive Bayes classifier, which is one of the most commonly used classifiers for text classification. Multinomial Naive Bayes (C.D.Manning et al., 2009 [4]), is a probabilistic method where the probability of a document d being in class c is computed as

$$P(c|d) = P(c) \prod_{1 \leq k \leq n_d} P(t_k|c)$$

Furthermore, $P(t|c)$ is the conditional probability of a term t occurring in a document of class c, and $P(c)$ is the prior probability of a document occurring in class c. We classify a document to the class that has the maximum posterior probability, given by

$$C_{map} = \arg\max_{c \in C} \hat{P}(c|d)$$

However, as mentioned in subsection 5.1, in our problem we calculated the sum of the five classes/ratings weighted by their respective probabilities in order to produce a regression output.

5.4 Multinomial Logistic Regression

Multinomial Logistic regression was also implemented in order to predict the rating of products through review texts. In contrast to Ordinary Least Squares, Multinomial Logistic regression does not have any particular assumptions that have to be met, and we do not assume any specific distribution for the predictor variables. Response variable Y has categories $j = 1, 2, \ldots, C$, where in this case, $C = 5$. The probability of an instance being in class j is given by

$$\pi_j = P(Y_i = j),\ j = 1, 2, \ldots, C$$

For C classes/categories, choosing one of them as the baseline category results in $C - 1$ regression equations that build together the model. The baseline category is the one to which all other categories are contrasted. Choosing the first category as the baseline, the $C - 1$ regression equations are given by

$$\log \frac{\pi_j(\mathbf{x}_i)}{\pi_1(\mathbf{x}_i)} = \alpha_j + \mathbf{x}_i^T \beta_j = \alpha_j + \sum_p x_{ip}\beta_{pj},\ j = 2, \ldots, C$$

where α_j is the intercept of class j, $\mathbf{x}_i^T$ is the feature vector of the i^{th} instance, and β_j is the vector of regression coefficients for class j. The intercept and vector of regression coefficients for the baseline category are set to $\alpha_1 = \beta_1 = 0$. The probability that instance i belongs to class j is given by

$$\pi_j(\mathbf{x}_i) = \frac{\exp(a_j + \mathbf{x}_i^T \beta_j)}{\sum_h \exp(a_h + \mathbf{x}_i^T \beta_h)}$$

Instance i is assigned to the class with the highest probability. The categorical cross-entropy is minimized, given by

$$L = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j} y_{ij} \log(\pi_j(\mathbf{x}_i))$$

where n is the number of training instances.

5.5 Multilayer Perceptron

A MLP neural network was implemented to predict the ratings of products using the review texts. To build the network we used Python as our software and `Keras`[4] as the frame to build the network, with Tensorflow [1] as backend. The network was trained on `Tritanium`, which is a GPU computing machine of Leiden University's Data Science Lab. `Tritanium` has 1GB RAM, 16 NVIDIA Tesla K80 GPUs each with 12GB memory, and 3TB HD space. The network takes 5,000 inputs, which is the number of extracted features. The input is connected to a 10 nodes hidden layer activated using ReLU and dropout value of 0.2. The hidden layer is then connected to a 5 nodes output layer activated using softmax. The loss function is the categorical cross entropy function, and the Adam optimizer [3] is used, with a batch size of 32. The batch generator generates small batches to be trained in order to avoid any memory errors, given the limited computational power. After determining the batch, it is converted from a sparse format to dense using the function `toarray()`, and fed into the network. The batch generator was implemented with 20 shuffled samples specified at each epoch. The number of epochs was set to 200 with `EarlyStopping` applied to monitor the validation loss with `patience` value of 4. `Patience` value is the number of epochs to wait before early stopping if there is no decrease on the validation set. Furthermore, we applied `ModelCheckpoint` to save the best model found by the network after early stopping.

5.6 Results

In this subsection, the results of the four models and the baseline are presented. The training and test split comprised 70% and 30% of the data set, respectively, using the `train_test_split()` function of `sklearn`, with random state set to 47. Specifically for MLP, the training set was split again to

[4] `https://keras.io/`

90% training and 10% validation. The models were trained on the training set, validated on the validation set (only for MLP), and tested on the test set. The baseline model, that is, the mean rating of the reviews in the training set produced a MAE of 1.03 and a RMSE of 1.27 on the test set, as illustrated in Table 3.

The Linear Regression model was trained for 7 minutes, and despite the fact that $R^2_{adj} = 0.33$, which means that only 33% of total variance is explained by the model, MAE and RMSE of 0.74 and 1.06, respectively, show that the model generated relatively good predictions, outperforming the Baseline and Multinomial Naive Bayes (MNB) (Table 3).

Regarding the assumptions of Linear Regression, we assume that the observations, and thus the residuals are independent of each other. This means that the rating given to a product by a consumer does not depend on the rating given to the same or different product by another consumer. Additionally, we assume that the same consumer rates two different products independently. Thus, we assume that the assumptions of independence of observations and of no autocorrelation, described in subsection 5.2, are satisfied. To examine the assumption of normality of residuals, the plot on the left-hand side of Figure 5, called QQ-Plot, can be used. The optimal case would be achieved if the points were all fitted on the red line. In this case, it can be observed that there are no large deviations from normality, and hence, the assumption is satisfied. In the right hand side of Figure 5, the plot of residuals against the predicted values $\hat{y}$ is used to examine the assumption of homoscedasticity, that is, the constant variance of residuals. When the residuals have constant variance, we expect them to form a (random) cloud of points, with no specific pattern. The pattern of diagonal lines is formed only because of the constrains in the response variable y that takes integer values between [1, 5], and not necessarily because the variance of residuals is not constant. Nevertheless, given the plot, it is difficult to conclude whether the assumption is satisfied or not. Finally, the last assumption of multicollinearity is almost impossible to be satisfied, given that our model has 5000 predictor variables of TF-IDF values, where many of them are correlated with each other.

The Multinomial Naive Bayes was the fastest model to train, with only half a minute training time. Nevertheless, it produced the worst results out of the four prediction models with respect to MAE (0.85), and the worst result overall with respect to RMSE (1.46). The default value of parameter $\alpha = 1$ was used. On the other hand, Multinomial Logistic Regression had the lowest MAE (0.58) and RMSE (0.95) compared to all models. The training time lasted 3 hours and 25 minutes. For the penalty parameter, the default value of 1 was used.

Regarding the Multilayer Perceptron, it contained a total of 50,065 trainable parameters. The training finished in 106 out of 200 epochs when the validation loss stopped decreasing. Each epoch lasted around 9 minutes and 30 seconds, hence, around 16 hours and 42 minutes in total for 106 epochs. The long training time is due to the slow operation of converting the batches from sparse to dense arrays. Figure 6 shows the loss for both the training and validation sets per epoch. After training the model, we used it to predict the test set, resulting in a MAE of 0.69 and RMSE of 0.96.

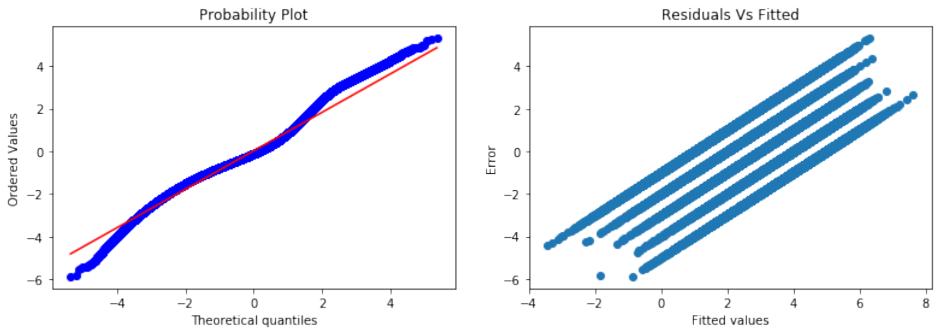

Figure 5: Left: QQ-Plot of residuals. Right: Residuals Vs fitted values.

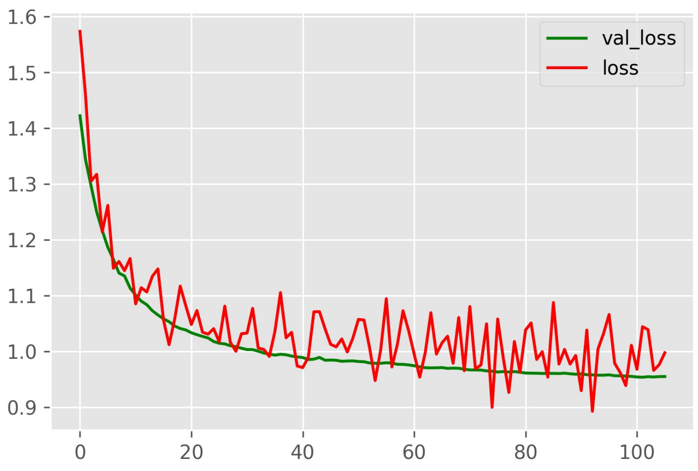

Figure 6: Training (`loss`) and validation (`val_loss`) categorical cross entropy loss.

	Baseline	LR	MNB	MLR	MLP
MAE	1.03	0.74	0.85	0.58	0.69
RMSE	1.27	1.06	1.46	0.95	0.96
Time	-	7 min	30 sec.	3 hr 25 min	16 hr 42 min

Table 3: Performance in the test set, with respect to MAE and RMSE for the Baseline model, Linear Regression (LR), Multinomial Naive Bayes (MNB), Multinomial Logistic Regression (MLR), and Multilayer Perceptron (MLP).

6 Conclusions

The research question we address in this paper is whether we can use machine learning models to accurately predict the rating of a product by a consumer, given the consumer's review text. We used a 58GB data set of 82,677,140 Amazon US product reviews from 1996 to 2014, provided by He and McAuley [2]. In subsection 2.2 descriptive statistics of the data set were provided, where we showed that product ratings are imbalanced towards 4 and 5, with a mean value of 4.16. In section 3 we used `Dask`, a Python library for distributed and parallel computing, in order to perform several text preprocessing tasks such as removal of punctuation marks and numbers, case folding, removal of stop words, and stemming. The computations were carried out on `Mithril` super-computer of Leiden University's Data Science Lab, using 20 CPU cores. The preprocessing task was completed in 3 hours and 10 minutes.

The `DAS3` distributed system of Leiden University was utilized in order to create a sparse matrix of TF-IDF values from the preprocessed reviews, where the features/columns are the 5000 most frequent terms across the whole data set. The whole computation process from encoding the preprocessed reviews to the creation of the sparse matrix lasted for 2 hours and 40 minutes. This matrix comprises the input for our prediction models. In total, four models were developed, namely, Linear Regression, Multinomial Naive Bayes, Multinomial Logistic Regression, and a Multilayer Perceptron with 5000 input nodes, a hidden layer of 10 nodes, and 5 output nodes. Those models were contrasted with a baseline model; the mean value of the ratings in the training set. Multinomial Naive Bayes produced the worst results in the test set with respect to RMSE, improving the predictions compared to the baseline with respect to MAE only. The best results were produced by Multinomial Logistic Regression, with a MAE of 0.58, improving the predictions compared to the baseline by 43.6% (Baseline 1.03). MAE indicates that the predictions deviate, on average, by 0.58 from the true rating values. The same model yielded an RMSE of 0.95, improving the results by 25.2% (Baseline 1.27). The MLP produced the second best results, decreasing the error by 33% and 24.4% compared to the baseline, with respect to MAE and RMSE, respectively, and it was followed by Linear Regression. We conclude that Multinomial Logistic Regression provides sufficiently good results in order to predict the rating of the product given the review text. Further experimentation with models' hyper-parameters, such as the α parameter for MNB, penalty parameter for MLR, different optimizers and number of hidden layers and nodes for the MLP would possibly provide even better rating predictions.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] R. He and J. McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *proceedings of the 25th international conference on world wide web*, pages 507–517. International World Wide Web Conferences Steering Committee, 2016.

[3] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[4] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval.* Cambridge University Press, New York, NY, USA, 2008.