Recommendation System for Yelp Users

Siyuan Wang SW884@CORNELL.EDU
Department of Computer Science
Cornell University

Mingchen Zhang MZ466@CORNELL.EDU
School of Operations Research and Information Engineering
Cornell University

Xuan Zhao XZ544@CORNELL.EDU
Department of Systems Engineering
Cornell University

Abstract

This report describes a recommendation system for Yelp users based on various machining learning algorithms, which could predict a review score (ranging from 1 to 5) from a customer score to a new restaurant based on his or her historical visits and comments on Yelp. This model could be utilized to generate customized recommendations varying from person to person, which could improve the user experience and help business target certain customers at the same time.

We present the four methods adopted to learn customers preferences and compared their effectiveness in terms of prediction accuracy. We also discuss the cons and pros of these algorithms in depth. (need solid results listed here).

1. Introduction

Yelp, as an essential tool in our life, provides users with crowd-sourced and comprehensive reviews about local businesses. The platform brings mutual benefits to both customers and business owners by bringing customers to their desired places. However, the current issue with Yelp is that it gives the same recommendation to everyone, no matter it depends on distance or overall ratings. In other words, they are not able to provide customized recommendations based on users' specific tastes for food and entertainment. In our project, with machine learning algorithms we wish to develop a recommendation engine that could push up businesses to the user's preferences and tastes so that the customers could locate their preferred restaurants more efficiently, and the business owners could also improve their strategies by learning more about the target customers.

2. Data Sets Description

The data sets are available on Yelp Data Challenge website. The data set **User** contain information about the users (User ID, Name, Count of reviews, historical compliments, etc.). Another data set **Business** contains information about the business (Business ID, ambiance, parking options, open hours, food categories, etc.) The last set **Reviews** links the customers and business, which include the rating and comments customers left. A detailed data structure of each data set is listed in Appendix A.

3. Data Cleaning and Constructing Feature Matrix

The output matrix **Y** is straightforward in our case, which is the review score (ranging from 1 to 5) a customer will give to a restaurant he visited. To generate the feature matrix **X** from the data sets, we tried a series of approach:

3.1 Joining User data entry and Business data entry

We first selected business and user attributes that we believe would affect customers' ratings. The first matrix **X** has 45 features. Nine of them comes from data set **User**: Average ratings, number of "cool"/"hot"/"more"/"writer" compliments, user's fan/review/votes counts. The rest of the features comes from the **business**: categories they belong to (e.g., American, Chinese, Italian; or Breakfast, Brunch, Dinner, etc., which accounts for most features), types of alcohol served, the ambiance of the restaurant and attire required.

Since those features have various types of values (boolean, numerical or string), we tried different approaches to transform them into numerical values:

- For nominal values that have relatively fewer possible values(less than 10), we applied one-hot-encoding to convert them into numerical arrays. For example, for attire attribute that have 3 possible values "formal", "dressy", "casual", we encode attire = "formal" into [1,0,0], attire = "dressy" into [0,1,0], attire = "casual" into [0,0,1]. The corresponding w matrix will have 3 rows for this features.
- For ordinal values, we convert them into numerical values directly. For examples, the 4 "noise level" values(very loud, loud, average, quiet) were encoded into 1,2,3,4. Note that these values were later standardized via subtracting each cell by column mean and divide by column standard deviation.
- For Boolean values, we convert the truth value into -1(negative value) and 1(positive value).
- For nominal data with unknown values, such as business "categories", we could no longer use one-hot encoding since we will simply explode the size of feature matrix. The solution we came up with is to calculate the **correlation** metric between the categories values and the frequencies of categories the user visits. When the input data entry contains user X and business Y, we follow the following steps to estimate the correlation between them:
    1. Go through the **Review** data set and find all businesses that user X has written reviews for;
    2. Go through all the businesses we found in the previous step and count the categories each of the businesses is in.
    3. Generate the distribution of each category based on the count from the previous step. e.g., for a customer, he might have Japanese Food - 30%; Italian Food - 40%; American Food - 30%. We consider this distribution as the user's preference setting.
    4. Then add the frequency percentage of each of the categories that business Y have. For example, if business Y has category "American Food" and the user's preference distribution is [Japanese Food = 30%; Italian Food = 40%; American Food = 30%.], we will assign 0.3 as the correlation value.
    5. This approach also works in the situation where one category is a subset of another. Even if we are counting the frequency percentage of both categories, it is fine because matching both category and subcategory intuitively means it correlates with the user's preference more.

- We did have missing data when building feature matrix in this approach. However, all the nominal values don't have missing data and only Boolean values could be missing. When Boolean values are missing, we filled in the value 0. Since True corresponds to 1 and False corresponds to -1, the value 0 will have minimum negative effect when training our model. (Note that this happened before we learned about Low Rank Model and the techniques to fill missing data in lectures).

However, this feature matrix **X** turned out to be very sparse since there were too many large one-hot encoding vectors. Please refer to the figure in Appendix E for the distribution of feature types. When fitting our model(Models are discussed in further details in Section 5) using this feature matrix, the best accuracy we were getting were around 40% when trying with different loss functions and regularizers.

3.2 Further selection on features

After consulting with our TA, we realized that perhaps we included too many features from the raw dataset. The overwhelming amount of features could hurt our model if some of them are linearly dependent on one another, or some of them are simply generating noise (doesn't correlate with output). As a result, we decided to try modeling our problem with a small subset of the previous feature matrix that only includes features that are absolutely essential.

After discussing the relevance of each of the previous 45 features, we condensed our feature matrix to have only 7 features: attire(3 nominal values), noise level(4 ordinal values - 1 column), price range(4 ordinal values - 1 column), waiter services(Boolean), categories(weighted correlation value as explained in the previous approach), average ratings of the restaurant(numerical) and the average score from this customer(numerical). After one hot encoding, there feature matrix has a column size of 10, which is much less than the size of the feature matrix in the previous approach.

When fitting models using ordinal/logistic/quadratic losses (Models are discussed in further details in Section 5), we got better accuracy results than the previous approach. The best accuracy we were getting was 48% accuracy.

3.3 Building User Preference Correlation features

After inspecting our feature matrix again, we found that the model we fit doesn't generalize very well. The way each of the business features affects the user's rating on the business varies from user to user. Intuitively, some user might consider certain attributes of the business(waiter service, ambiance, etc.) more than other users. So our model might work well if we train on a single customer, but it doesn't scale to a large number of customers since our W matrix will constantly adjust to the preferences of single customers and won't eventually converge. We took the following steps to address this issue:

- Similar to what we have done for the "category" feature in our initial approach, we need to evaluate a preference correlation value for each of the nominal features that seem depend on personal preferences.
  - Example for calculating preference correlation: if a user X previously visited businesses with 'attire' distribution of (dressy = 10%, casual = 60%, and formal = 30%), and the business in the input data has "formal" as its "attire" attribute, then we will assign the preference correlation value as 0.3.
  - Example for not calculating preference correlation: the noise level has a universal negative effect on the user rating. Therefore we don't have to calculate the user preference correlation here.

After computing the user preference correlation for attire, price range, waiter service, categories, we reduced the column size of our feature matrix to just 7.

4. Dimensionality Reduction and Clustering

Our goal for this project is to provide a customized rating system to a given user. As what we have mentioned before, the attributes from a given business data entry may have different impact on users with different preferences. That is why we converted our raw input matrix X into a user preference correlation matrix. Another issue that needs to be solved is that different user may also value the same feature preference differently. For example, some customer may treat preference correlation for 'attire' as a more important factor than other people when rating a given business. To tackle with this, we decided to use unsupervised learning to cluster users with similar weights on their preference correlations. When we finished building the input matrix, we applied the K-means algorithm on it to find sub groups of users where each group shares similar preferences. In this case, we randomly picked up 2000 data points from input matrix X for several times and tried to visualize each 2000 data points respectively. To achieve this, firstly, we applied GLRM via SVD approach to reduce the noise in our input matrix X. Then, we used dimensionality reduction method via PCA to project each data into three dimensions and visualized each 2000 data points group. Since we choose our data randomly, data points in each plot basically shares the same distribution. Hence, we could get the intuition about how our overall input data sets distribute as well as the cluster size we can put in when running the k-means algorithm. One 3-D visualization of 2000 sample data sets in X is shown as following:

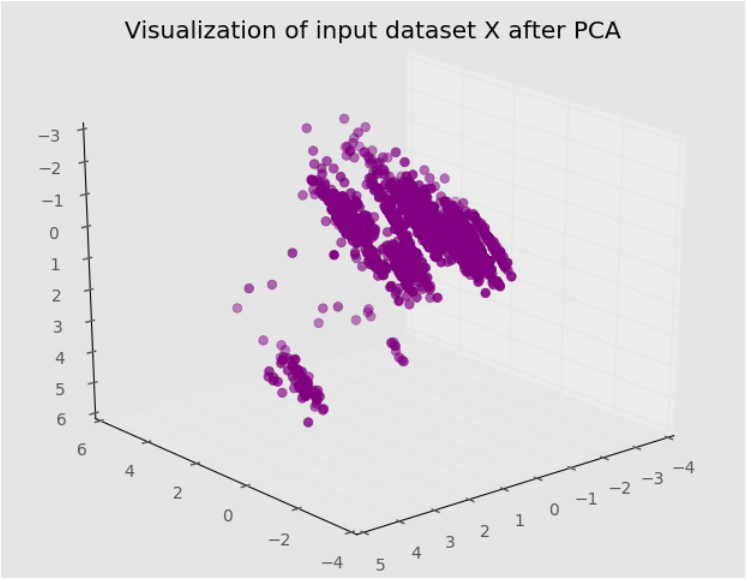

Figure 1: Visualization of Feature Matrix after PCA

From the visualization plot, we could determine that the input data points in the input data can be aggregated into three groups. Hence, we applied K-means algorithm on our input data set with 3 as our cluster size parameter. The visualization of the input data set after clustering is shown as below:

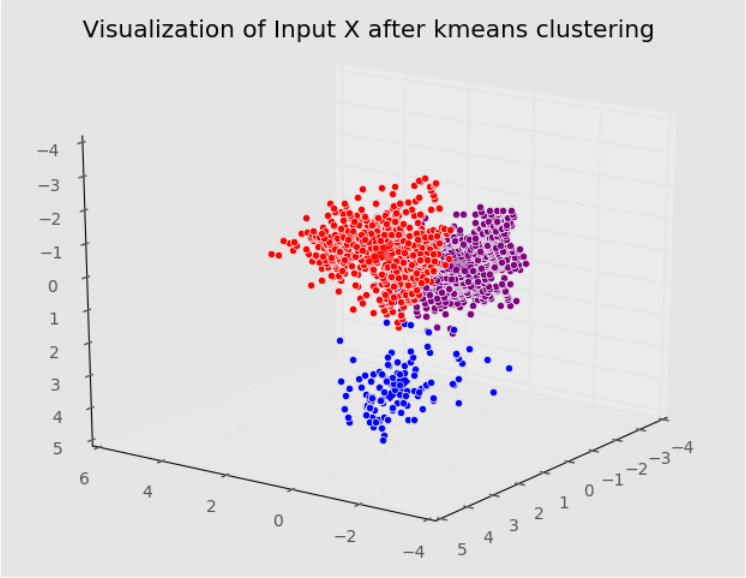

Figure 2: Visualization of Feature Matrix after k-means Clustering

Next, we implemented the K-means algorithm to cluster the initial training data into three groups. We can assume that users in the same group basically share similar weights on their preferences(how each preference affects the final rating). When plotting all the data points in a 3-dimensional space, we were able to get 3 centroid coordinates for each cluster(average of all the coordinates for all data points in each cluster). When we use the existing models to predict the output for test data, we follow the steps below:

- Apply PCA to reduce to the same dimension as the training data.
- Find out which of the centroid from the 3 clusters is closest to this data point using the data point's coordinates in reduced dimensions.
- After identifying the cluster to use, we use the model we trained for that cluster to predict output value.

It is worth mentioning that here we choose three as the number of dimensions when using PCA in order to get a better visualization of the clustering pattern. We experimented with different dimension values for the PCA method and found that reducing to 3 dimensions yielded similar predict accuracy as the training set that were reduced to 5 dimensions, which means that we did not lose too much information when reducing to 3 dimensions.

5. Model Fitting

In this section, we introduced the four methods we used to build the regression model: Quadratic Loss + Ridge Regularizer, Ordinal Loss + Ridge Regularizer, Logistic Loss + Ridge Regularizer and Vectorized matrix Y.
The reason we chose ridge regularizer is to avoid overfitting and always guarantee a unique solution for w matrix. Furthermore, after we reduced the number of features to 7, it seemed inappropriate to use lasso regularizer since it could cause solution w unnecessarily sparse. To find the coefficient of the ridge regularizer λ, we chose to use the K-fold technique to perform cross validation.

5.1 Trail 1 - Quadratic Loss + Ridge Regularizer

We start with the ordinary least square loss function and Ridge regularizer. We used Ridge regression package[1] from scikit-learn.com.

The advantage of this loss function is that it penalizes values that deviate from the target more severely than the other loss function. This helps us keep our prediction to be within the expected range of values [0,5].

The disadvantage of this loss function is that it penalizes value incorrectly beyond the limit of the bound of the output. e.g., We are supposed to penalize the value 6 equally as 5 when the actual value is 4 while quadratic loss function penalizes 6 more severely than 5.

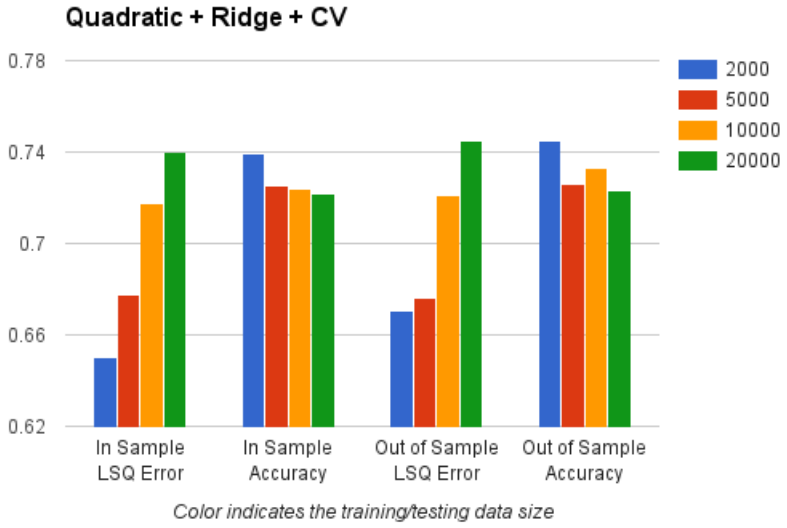

Figure 3: Histogram of Quadratic regression results using different sample sizes

5.2 Trail 2 - Ordinal Loss + Ridge Regularizer

Ordinal Loss is the ideal loss function to fit our model since our output Y is a very typical ordinal value(range from 1 to 5). We used the OrdinalRidge Python API from mord package[2] to perform the regression. Although we don't know the exact underlying loss function the Python library uses, we are certain that its penalty is proportional to prediction's deviation from the actual value. e.g., The prediction value of 5 will receive a higher penalty than prediction value of 3 when the actual output value is 1. One example we learned in lecture is ordinal hinge loss.

1. It can be found at http://scikit-learn.org/stable/modules/generated/sklearn.linear-model.Ridge.html
2. It can be found at http://pythonhosted.org/mord/reference.html

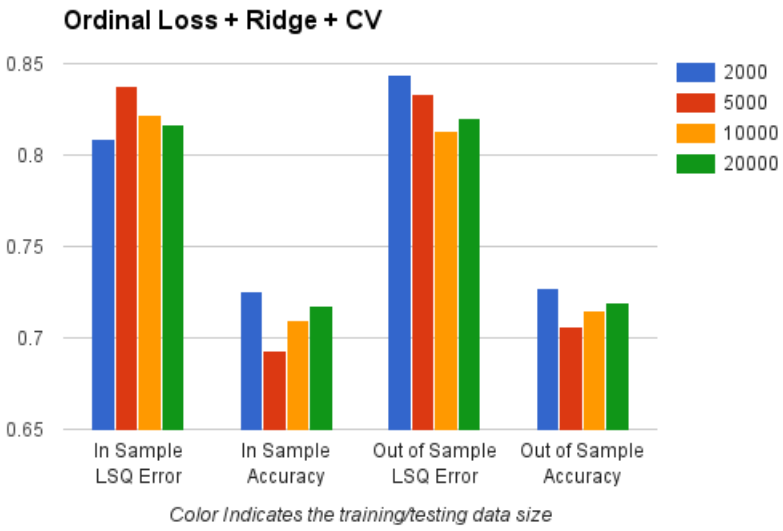

Figure 4: Histogram of Ordinal regression results using different sample sizes

5.3 Trail 3 - Logistic Loss + Ridge Regularizer

To increase our models accuracy, we moved on to tackle it from a multinomial classification perspective and applied logistic regression to our model to specify the rating range a customer should belong to. The disadvantage of logistic loss is that we give all prediction mismtaches the same penalty and it doesn't account for the prediction's deviation from the actual value.

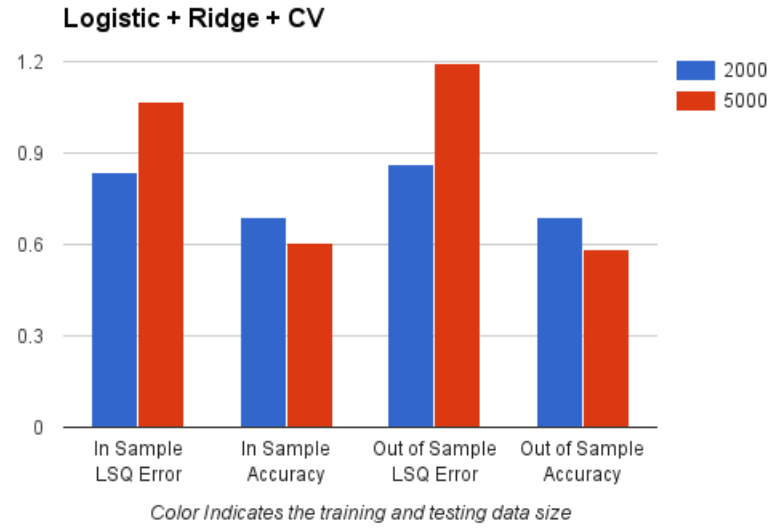

Figure 5: Histogram of Logistic regression results using different sample sizes

5.4 Trail 4 - Ordinal Logistic Regression (Vectorizing Output Matrix Y)

In this approach, we first vectorized our output Y by using the following encoding:

$$\varphi(y) = [[y \geq -1000], [y \geq 1.5], [y \geq 2.5], [y \geq 3.5], [y \geq 4.5]]$$

Then we used logistic loss as our binary loss function:

$$l^{bin}(\varphi_i, z_i) = l_{logistic}(\varphi_i, z_i) = log(1 + exp(-\varphi_i z_i))$$

Then when comparing two output values, we sum up all the binary loss across all five values in the vector:

$$l(y, z) = \sum_{i=1}^{k-1} l^{bin}(\varphi(y)_i, z_i)$$

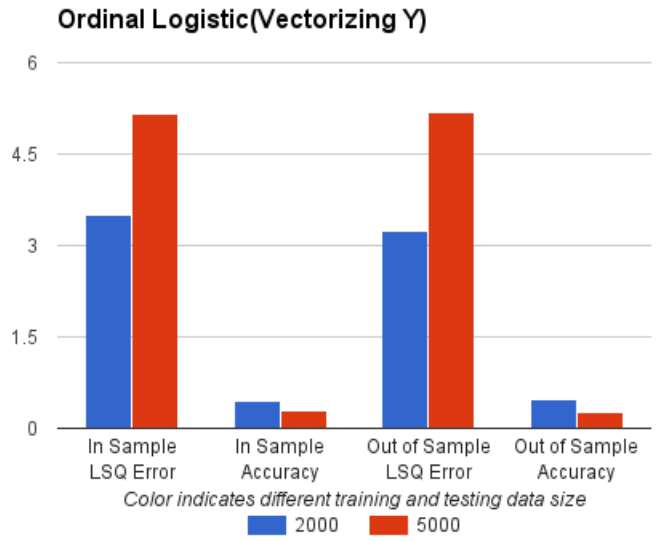

Figure 6: Histogram of Ordinal Logistic regression(vectorizing Y) results using different sample sizes

6. Conclusions

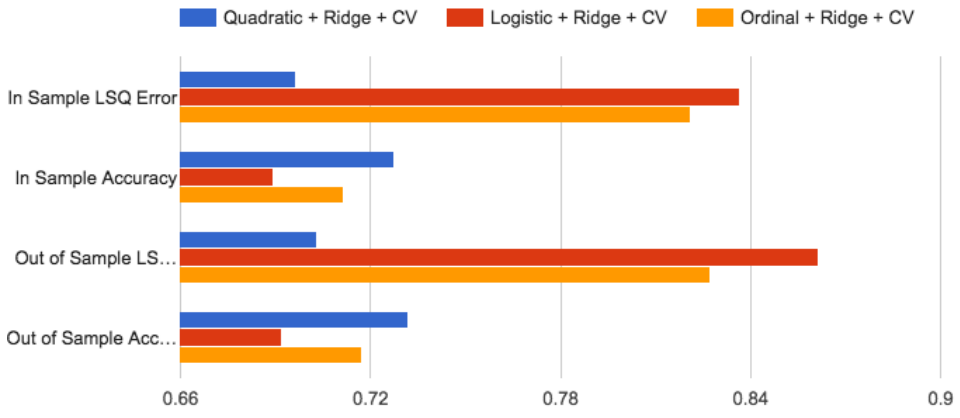

Figure 7: Comparison of Quadratic, Logistic and Ordinal Regression

From the visualization of the final results for each model, we can see that regression using either ordinal loss function or quadratic loss function and ridge regression yields good accuracy(around 70%). This observation matches our expectation, since both quadratic loss and ordinal loss penalizes predictions

that deviate from target value heavily when the deviation is high. On the other hand, the penalty given by logistic regression doesn't account of deviation between the predicted value and the target value.
We can also observe that quadratic loss function performs slightly better than ordinal loss function. This might be caused by the fact that ordinal loss functions, such as ordinal ridge loss function, doesn't penalize values that are out of range enough(for example, prediction 6 and 10 have the same amount of error when the actual value is 5). On the other hand, quadratic loss penalizes more the further you deviate from the target value regardless of the range of ordinal values.
If we were to decide whether or not to use this software in production, I would say yes, since our accuracy doesn't decay as the data size grow. Either ordinal regression or quadratic loss regression yields about 72% accuracy even when both training and test data size is at 20,000.

Acknowledgments

http://scikit-learn.org/

https://pypi.python.org/pypi/mord

Appendix A. Business

```
    {
        'type': 'business',
        'business_id': (encrypted business id),
        'name': (business name),
        'neighborhoods': [(hood names)],
        'full_address': (localized address),
        'city': (city),
        'state': (state),
        'latitude': latitude,
        'longitude': longitude,
        'stars': (star rating, rounded to half-stars),
        'review_count': review count,
        'categories': [(localized category names)]
        'open': True / False (corresponds to closed, not business hours),
        'hours': {
            (day_of_week): {
                'open': (HH:MM),
                'close': (HH:MM)
            },
            ...
        },
        'attributes': {
            (attribute_name): (attribute_value),
            ...
        },
    }
}
```

Appendix B. Review

```
{
    'type': 'review',
    'business_id': (encrypted business id),
    'user_id': (encrypted user id),
    'stars': (star rating, rounded to half-stars),
    'text': (review text),
    'date': (date, formatted like '2012-03-14'),
    'votes': {(vote type): (count)},
}
```

Appendix C. User

```
{
    'type': 'user',
    'user_id': (encrypted user id),
    'name': (first name),
    'review_count': (review count),
    'average_stars': (floating point average, like 4.31),
    'votes': {(vote type): (count)},
    'friends': [(friend user_ids)],
    'elite': [(years_elite)],
    'yelping_since': (date, formatted like '2012-03'),
    'compliments': {
        (compliment_type): (num_compliments_of_this_type),
        ...
    },
    'fans': (num_fans),
}
```

Appendix D. Check-in

```
{
    'type': 'checkin',
    'business_id': (encrypted business id),
    'checkin_info': {
        '0-0': (number of checkins from 00:00 to 01:00 on all Sundays),
        '1-0': (number of checkins from 01:00 to 02:00 on all Sundays),
        ...
        '14-4': (number of checkins from 14:00 to 15:00 on all Thursdays),
        ...
        '23-6': (number of checkins from 23:00 to 00:00 on all Saturdays)
    }, # if there was no checkin for a hour-day block it will not be in the dict
}
```

Appendix E

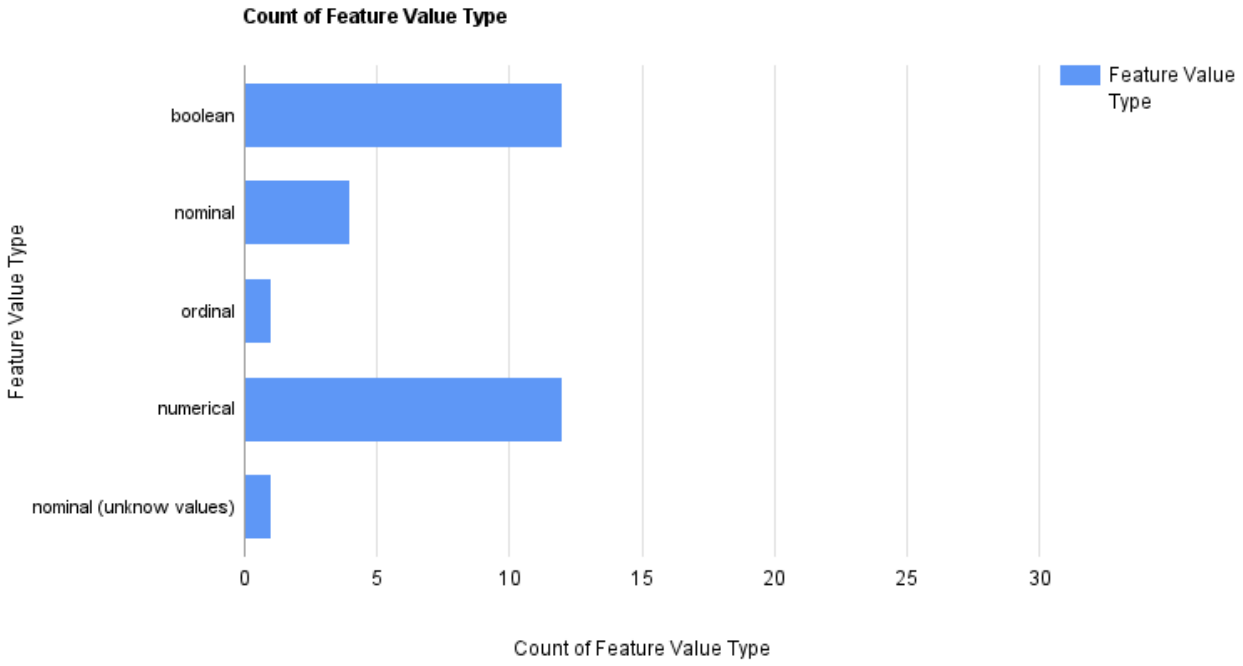

Figure 8: Data types distribution of finalized features