Introduction to Machine Learning

Fall 2022

University of Science and Technology of China

Project Report | PB20061254, Yifei Zuo, 33%

Lecturer: Jie Wang | PB20061210, Yongfan Lai, 33%

Due: Jan. 11, 2023 | PB20061372, Yunqin Zhu, 33%

1 INTRODUCTION

Recommending clothes of suitable sizes to customers based on the information of clothes and users are very important for E-commerce platforms. In this project, we implement several classifiers to predict customers' fit feedback based on a dataset collected from RentTheRunWay.

2 PREPARATION

The training dataset contains 87766 samples, each of which has 14 features and 1 label. We summarize these records as follows:

(1) Item attributes, i.e. `item_name`, `size` and `price`; (2) User attributes. The first one is `user_name`, which is excluded in the test set. The others describe the body characteristics of each user: `age`, `height`, `weight`, `body_type` and `bust_size`; (3) Transaction attributes, i.e. `rented_for`, `usually_wear`; (4) Feedback, i.e. `fit`, `review_summary review` and `rating`, among which `fit` is the target variable we want to predict, and the other three are supposed to be inaccessible on the test set.

By observing that most of the inputs has missing values and inconsistent formats, we need to design a thorough data cleansing (and transforming) pipeline that converts the raw data into either categorical or numerical variables we can leverage for training. We also need to deal with the data imbalance issue, since the number of `True to Size` samples, consisting of 70% of the whole dataset, is much larger than the other two classes. These two challenges will be discussed in detail in the following sections.

2.1 Data Cleansing

We load the provided json file into memory as tabular data, with each rows being a training sample and each column representing features or labels. Before diving into each separate input column, we first drop the 190 samples corrupted by the byte order mark `\udeff`. Moreover, as the labels are encoded differently in the training and test sets, we unify them to integers 1, 2, 3 for `Small`, `True to Size` and `Large`, respectively. This encoding approach captures the ordinal nature of the labels, which is important for the design of our model.

2.1.1 Numerical Features

There are 5 input columns that can be considered purely numerical: `price`, `age`, `height`, `weight`, `usually_wear` and `rating`. We remove the prepended dollar sign from `price`, convert the unit of `height` from feet and inches to centimeters, convert the unit of `weight` from pounds to kilograms, and treat them as float point numbers along with the other three columns.

We also notice that there are some anomalous values, i.e. `age` ≤ 5 or ≥ 100, `height` > 200 cm and `weight` < 30 kg or > 150 kg, which are more likely to be caused by typos or data corruption than real values. To avoid the harmful effect of these outliers, we set them as `NaN` and replace them with the corresponding median values in later steps.

The challenge is to handle the `bust_size`. We observe that a valid `bust_size` alway contains two part: (1) number part in inches; (2) letter part, implying the `cup_size`. Since these two parts are different measurements of women busts, it is necessary to split them into 2 features. The `cup_size`s, ordered as `AA` < `A` < `B` < `C` < `D` < `D+` < `DD` < `DDD/E` < `F` < `G` < `H` < `I` < `J`, are ordinally encoded as $0 \sim 12$. We now have `bust_size` in 2 numerical values.

After cleansing, the 4 numerical features that measures user's body characteristics approximately follow a normal distribution, as is shown in the figure below. We will normalize them to have zero mean and unit variance in the later steps.

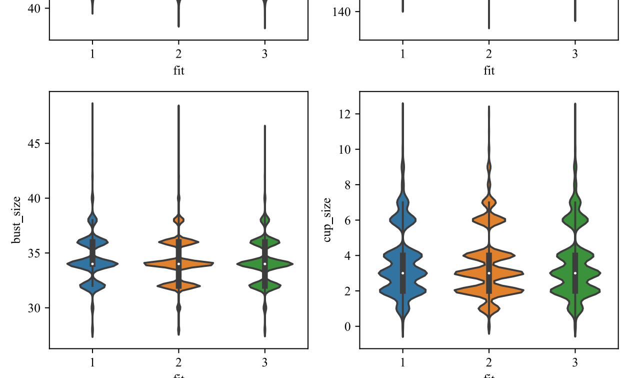

Figure 1: Violin plots of `weight`, `height`, `bust_size` and `cup_size`

2.1.2 Item Names

Next, we take a look at the categorical features. For `item_name`, we notice that it demonstrates some common patterns and can be utilized to extract extra infomation; that is, the first line of `item_name` typically contains the brand name, while the second line contains the product name with the last word indicating its category, e.g. `Jumpsuit`, `Romper`, `Skirt`, `Top`, `Blouse`. Therefore, we parse each `item_name` string and yield two new features: `brand` and `category`. We also take into account and handle the samples that have no `brand` information.

2.1.3 Item Sizes

The `size` column is the most dirty and challenging one, since there are diversified real-world specifications for clothing sizes. It seems infeasible to manually map all the possible values to a unified format without any misinterpretation. However, it is possible to determine the relative magnitude of different sizes for a certain item. This idea is based on the observation that samples having the same `item_name` conforms to the same size standard.

To be specific, we first roughly group all size values into 4 `size_scheme`s: `letter`, `number`, `mixed` and `onesize`. For `letter` scheme, size values are expressed in capitalized letters. Using our prior knowledge, we can map them into `4XS` $\sim$ `XS`, `XS-S`, `S`, `S-M`, `M`, `M-L`, `L`, `L-XL` and `XL` $\sim$ `4XS` and assign an order within each "parent item" (i.e. the set of items that share the same `item_name`). For `number` scheme, size values are expressed as a number or as between two numbers, and thus can be easily sorted within each parent item.

After obtaining the sorted indices of size values (e.g. 0 for `S`, 1 for `M`, 2 for `L` and 3 for `XL`), we subtract the mean of the indices from each index and yield a new feature `size_bias` (e.g. -1.5 for `S`, -0.5 for `M`, 0.5 for `L` and 1.5 for `XL`). This feature is supposed to capture the difference between the size of a sample and the average size of its parent item. We will discuss how to leverage this feature in Section 3.2.1.

There are also some special cases where the size values within a parent item are expressed in a `mixed` format or only `onesize` is available. In the former case, as the samples are relatively rare, we simply treat number sizes as `NaN` and sort them as the `letter` scheme. In the latter case, we set `size_bias` as zero for all items.

Last but not least, we extract some meaningful `size_suffix`s from the raw size strings, e.g. `P` for petite, `R` for regular and `L` for long, which will be one-hot encoded in later steps.

2.1.4 Other Features

Among the other features, we note that `body_type` and `rented_for` are already clean categories and can be directly one-hot encoded. The 2 textual feedback features, i.e. `review` and `review_summary` have close relationship with `fit` and can only be used to predict missing labels in the training set (See Section 3.4 for details). We omitted their cleansing process here.

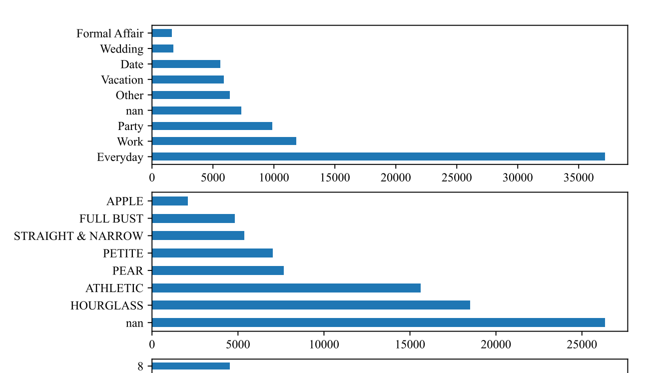

Figure 2: Bar plots of `rented_for`, `body_type` and `size_main` (`size` without suffix)

2.2 Exploratory Data Analysis

2.2.1 Numerical Features

Based on the current transformation on data format, we measure the relationship among all features to further filter the dataset. First of all we look at numerical features. According to the literal meaning, features like `price`, `age` are less likely to impact the `fit` feature, and by calculating their correlation index, we found that actually none of these numerical features are highly related to `fit`. We also analyze the possible combination of features, e.g `bmi` in the first figure below.

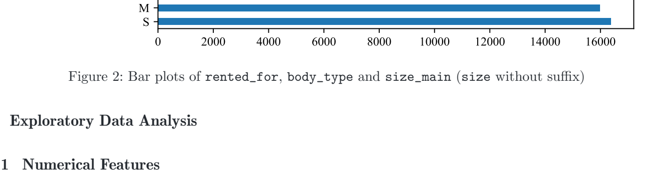

Figure 3: Heatmap of correlation between numerical features

Trivial these numerical columns are, we still try to exploit latent relationship from it, because just discarding them seems brute. The desire to gain futher insight of the data results in the Detecting User Prototypes phase in our methodology, which will be explained in section 3.2.2.

2.2.2 Categorical Features

Then, we measure the categorical features. Note that we still keep the original `item_name` despite we have split it into `brand` and `category`. This is because we are not sure yet the performance of splitting. We first analysis the distribution of `fit` value inside each of these categories, e.g in `rented_for` and `body_type`. Meanwhile, we here keeps the empty values because sometimes empty values could be seen as a category and resulting more robust performance than imputing them.

	Work	Vacation	Date	Everyday	Wedding	Party	Other		Formal Affair
Small	0.141108	0.123253	0.169825	0.110612	0.124058	0.152537	0.158621	0.180494	0.153374
True to Size	0.710100	0.719400	0.719878	0.698057	0.765217	0.728147	0.677273	0.637841	0.723313
Large	0.148792	0.157347	0.110297	0.191331	0.110725	0.119315	0.164107	0.181664	0.123313

Table 1: Proportion of `fit` in each `rented_for` category

		HOURGLASS	STRAIGHT & NARROW	ATHLETIC	PETITE	FULL BUST	PEAR	APPLE
Small	0.140840	0.135484	0.135347	0.125928	0.114990	0.150924	0.130304	0.159615
True to Size	0.696263	0.695525	0.717002	0.717414	0.704759	0.683621	0.702833	0.662019
Large	0.162897	0.168990	0.147651	0.156658	0.180251	0.165456	0.166863	0.178365

Table 2: Proportion of `fit` in each `body_type` category

We notice that except for `item_name` and its derivative column, the distribution inside each one do not have manifest difference, which implies that these features are still not the dominant one in terms of predicting `fit`. While `item_name` is too diverse and dirty for us to analyze in this way. So we measure the direct relationship of `fit` and all of these features by Chi-square test. The result matches our expectation from previous observation, `item_name` is the key point and it is even more important than the features we extracted from it.

col2	body_type	brand	category	fit	item_name	rented_for	size
col1							
body_type	614362.000000	5.198896e+03	7.800368e+02	104.291123	3.221431e+04	957.639037	1.567955e+04
brand	5198.895594	5.439463e+07	1.130522e+06	9645.844958	4.247737e+07	42839.056924	7.310006e+05
category	780.036832	1.130522e+06	6.480624e+06	4370.985614	6.480624e+06	35610.855228	1.794788e+05
fit	104.291123	9.645845e+03	4.370986e+03	175532.000000	3.776797e+04	940.406270	6.598512e+03
item_name	32214.314240	4.247737e+07	6.480624e+06	37767.970915	3.786786e+08	121225.494414	2.151424e+06
rented_for	957.639037	4.283906e+04	3.561086e+04	940.406270	1.212255e+05	702128.000000	1.941636e+04
size	15679.551631	7.310006e+05	1.794788e+05	6598.511928	2.151424e+06	19416.362596	1.421809e+07

Table 3: Chi-square test result of `fit` and all categorical features

In this perspective, we keep `item_name` as the original strings and remove `brand`, `category` in our final preprocessing pipeline.

2.3 Handling Data Imbalance

In this project, data imbalance has disastrous effect on our model performance; that is, if we simply feed all the training data into our model, it would probably learn to predict all the samples as `True to Size`. Observing the original data, we find that they are unevenly distributed, where the majority class consists of more than 70% of all the samples. To address this, we propose two alternative tactics: (1) Data augmentation; (2) Random split & aggregation.

2.3.1 Data Augmentation

This approach is based on the basis assumption that if an item is small for one user, then it is also small for all users whose `weight`, `height`, `bust_size` and `cup_size` is larger. In this way, we randomly fetch data samples with `Small` label, then duplicate it with bigger values in `weight`, `height`, `cup_size`, `bust_size` columns and vice versa. Moreover, during the experiment phase, we find that letting the number of `Large` data slightly less than other two classes data can have the better performance than averaging three classes. Therefore, setting the upsampling ratio to 2.7 for `Large` data and 3.6 for `Small` data, we can get fairly augmented train data.

2.3.2 Random Split & Aggregation

The second approach is to split the dominant class `True to Size` into n different group randomly, and concatenate them with n identical copy of the others, forming into n generated datasets. Then we feed them into n different workers which train and result in n models. We use each model to predict its own predictions and we sent them into Aggregator, who will give the final predictions by voting from the n workers' predictions.

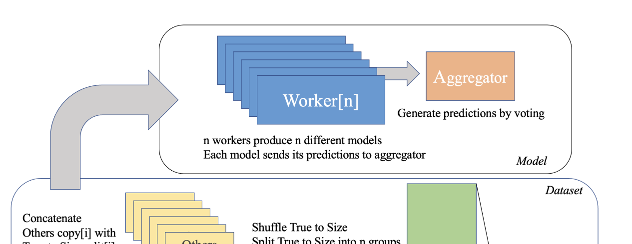

Figure 4: Random split & aggregation method

Noting that we don't necessarily have to split the dominant class into `n` samples so that the number of samples in each group is approximately identical to the others. Due to the fact that this task is relatively hard and our model won't give a tremendously impressing representation, we do need to fine tune the parameter `n`. A reasonable assumption is that by relatively giving more samples in dominant class, we result will be better because there will be more samples fall into this class. Here's some result of parameter search trained on `LogisticRegression` model: (Noting that we could not reach such performance by simply do one hot encoding due to the fact that we may encounter new value of `item_name`, this is just a demostration of how we fine tune this parameter.)

	2 Splits	3 Splits	4 Splits	5 Splits
accuracy	0.642019	0.696195	0.582033	0.519084
precision	0.532105	0.570019	0.507366	0.493443
recall	0.573357	0.539653	0.590706	0.595840
f1	0.547051	0.550630	0.521994	0.488164
f1_weighted	0.654479	0.686797	0.604777	0.542424
#small	2915.000000	2402.000000	4282.000000	4659.000000
#true2size	10803.000000	13127.000000	8750.000000	6780.000000
#large	3836.000000	2025.000000	4522.000000	6115.000000

Table 4: Fine-tuning of random split & aggregation method (one-hot encoded `item_name`)

3 METHODOLOGY

3.1 Overview

The overall architecture of our model is shown in the following figure. We model the fit feedback prediction task in two methods: (1) Multiclass classfication problem; (2) Ordinal regression problem. We use multinomial Logistic Regression and ordinal Logistic Regression to solve the problem respectively and compare the performance.

Feature engineering is also the key part of our model. Based on the analysis that `item_name` is the most important categorical feature, we adopt a latent representaion model that learns the true size of each item, i.e. item vectors in the embedding space. We also attempt to enhance the representation of each user by applying K-Prototype clustering on the user data and use the cluster centroids as additional user vectors.

Moreover, to utilize the 30% training data with missing labels, we experiment on leveraging pre-trained BERT models to classify the textual feedback and fill in the `fit` values.

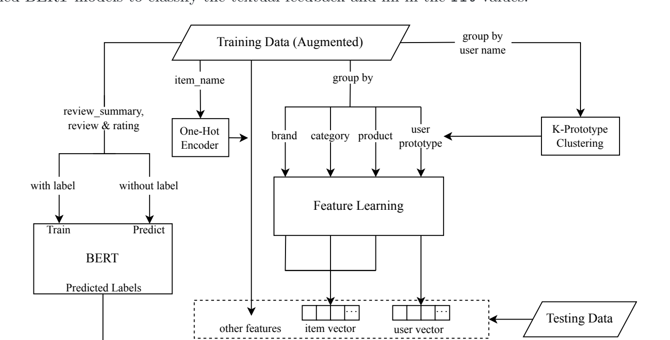

Figure 5: The architecture of our proposed model.

Our model is demonstrated to improve the performance when experimented on each separate component. Unfortunately, due to the limit of time, energy and resource, we are not able to fully implement the algorithms and integrate them all with only whitelist libraries.

3.2 Feature Learning

3.2.1 Leveraging Item Sizes

In the previous sections, we have transformed the item `size`s into `size_bias` feature which captures the difference between the item size and the average size of its parent item. The true size of an item can be viewed as a linear function of its size bias. Denote the item, the user, and the parent item of a transaction t by i, u, and p, respectively. We model the true size of the item i as

$$\mathbf{v}_i = \mathbf{v}_p + \epsilon_i \mathbf{d}_p,$$

where $\mathbf{v}_p$ is the true size of its parent item, ϵ_i is the bias of item size relative to the parent item, and $\mathbf{d}_p$ describes the heterogeneous influence of size bias. We want to learn $\mathbf{v}_p$ and $\mathbf{d}_p$ for each parent item p. A simple approach for estimating $\mathbf{v}_p$ is to use the mean of the true sizes of all users that have rent the parent item with $\epsilon_i = 0$. However, the item can either be too small or too large for some users, leading to noisy results. In order to learn the latent representation of item size more precisely, we drop all irrelevant features and solve the following ordinal regression problem:

$$\begin{aligned} \min_{\mathbf{w}, b_1, b_2, \mathbf{v}_p, \mathbf{d}_p} \quad & \sum_{t \in \mathcal{T}} \ell(y_t, f(\mathbf{w}, \mathbf{v}_i), b_1, b_2), \\ \text{s.t.} \quad & \mathbf{w} \geq \mathbf{0},\ \mathbf{d}_p \geq \mathbf{0}, \end{aligned}$$

where we define $f(\mathbf{w}, \mathbf{v}_i) = \mathbf{w}^\top (\mathbf{v}_i - \mathbf{v}_u)$ as the fitness score between the item and the user, and $\ell(y_t, f(\mathbf{w}, \mathbf{v}_i), b_1, b_2)$ is the total loss of 2 binary logistic classifiers sharing the same weights $\mathbf{w}$:

$$\ell(y_t, f(\mathbf{w}, \mathbf{v}_i), b_1, b_2) = \begin{cases} \log(1 + e^{f(\mathbf{w}, \mathbf{v}_i) - b_1}), & \text{if } y_t = 1; \\ \log(1 + e^{-f(\mathbf{w}, \mathbf{v}_i) + b_1}) + \log(1 + e^{f(\mathbf{w}, \mathbf{v}_i) - b_2}), & \text{if } y_t = 2; \\ \log(1 + e^{-f(\mathbf{w}, \mathbf{v}_i) + b_2}), & \text{if } y_t = 3. \end{cases}$$

Note that, in Prob. (1), we add the constraint $\mathbf{d}_p \geq \mathbf{0}$ to capture the monotonicity of the true size $\mathbf{v}_i$ with respect to the size bias ϵ_i, and also $\mathbf{w} \geq \mathbf{0}$ to ensure the monotonicity of the predicted fitness scores with respect to the item size; that is, if a size is `Large` for some user, then any bigger size would also be `Large`. These two constraints are trivial for the Projected Gradient Descent (PGD) algorithm, since the projection operator onto the non-negative orthant is a simple element-wise $\max\{0, \cdot\}$.

Therefore, we implement PGD algorithm to optimize the objective in rounds: (1) Initialize $\mathbf{w}$, b_1, b_2 randomly, $\mathbf{d}_p = \mathbf{0}$, and $\mathbf{v}_p$ as the mean of the true sizes of all users that have rent the parent item p; (2) In odd round, we fix the parameters $\mathbf{w}$, b_1, b_2 and optimize $\mathbf{v}_p$ and $\mathbf{d}_p$; (3) In even round, we fix $\mathbf{v}_p$ and $\mathbf{d}_p$ and optimize $\mathbf{w}$, b_1, b_2. We repeat the process for 10 rounds and obtain the final parameters $\mathbf{v}_p^*$, $\mathbf{d}_p^*$. See the `ItemVectorOptimizer` class in `preprocess.py` for more details.

3.2.2 Detecting User Prototypes

In this section, we will discuss the technique we use to enhance the representation of each user. Although the training data already contains the true size of each user, i.e. `weight`, `height`, `bust_size` and `cup_size`, these features are unable to express the heterogeneity among different users (e.g. personal preference), let alone considerable amount of missing values.

Following the same logic as we learn the latent variables for item sizes, it is reasonable to group the transactions by each user and learn user-specific features. However, unlike the `item_name`, the column `user_name` cannot uniquely identify a user. Hence, instead of learning the exact latent representation of each user, we attempt to learn several user prototypes by performing clustering algorithms on the user data, and use the cluster centroid as the user vector.

Besides numerical body measurements, our clustering algorithm should be aware of categorical features, i.e. `user_name`, `body_type` and `rented_for`, so as to capture user preferences. We propose using K-Prototype algorithm, which is a hybrid algorithm that combines K-Means for numerical and K-Mode for categorical data. Basically, the algorithm calculates the distance between a data point and a cluster centroid by summing the Euclidean distance between the numerical features and the Hamming distance between the categorical features. By experimenting with different values of cluster number n, we find that: as n increases, the resulting user vectors slightly improve our model performance on both the training and validation set. The classification scores are shown in the following table:

	accuracy	precision	recall	f1	f1_weighted
train (n=500)	60.10%	59.75%	60.10%	59.82%	59.82%
train (n=300)	60.02%	59.66%	60.02%	59.74%	59.74%
train (n=200)	59.99%	59.62%	59.99%	59.70%	59.70%
train (n=400)	59.94%	59.59%	59.94%	59.67%	59.67%
train (n=100)	59.90%	59.55%	59.90%	59.63%	59.63%
train (no user vector)	59.83%	59.47%	59.83%	59.54%	59.54%
test (n=500)	49.16%	46.04%	53.30%	45.24%	51.88%
test (n=200)	48.87%	46.03%	53.34%	45.11%	51.59%
test (n=400)	49.02%	45.86%	53.06%	45.09%	51.73%
test (n=300)	48.84%	45.82%	53.03%	44.98%	51.56%
test (no user vector)	48.61%	45.85%	53.13%	44.89%	51.32%
test (n=100)	48.66%	45.66%	52.73%	44.74%	51.42%

Table 5: Fine-tuning of K-Prototype clustering (one-hot encoded `item_name`)

These results are satisfactory if we ignore the computational cost of the algorithm. Unfortunately, it cost hours to cluster on the entire dataset using the existing `kmodes` library even after enabling parallel computing. Since the improved f1-score on the validation set is not significant, we decide not to implement this technique in our final model.

3.3 Fit Feedback Classification

In preceding part, we already have a penetrating insight of our tasks. As for the final models, considering the model performance and the implementing difficulties, we choose the Logistic Regression model to implement.

3.3.1 Multinomial Logistic Regression

At first, we implemented the classifier with gradient descent algorithm. However, its performance not so good as the model from `scikit-learn` library. So we imitate the sklearn implementation, using the BFGS algorithm in `scipy.optimize` to minimize the multiclass loss function, and get another form of the classifier. The loss function is formulated as:

$$\ell(\mathbf{w}, \alpha) = -\frac{1}{n} \sum_{i=1}^{n} \text{softmax}(x_i \cdot \mathbf{w}^\top)_{y_i} + \frac{1}{2} \alpha ||\mathbf{w}||^2$$

where $\mathbf{w} \in \mathbb{R}^{3 \times d}$ and d is the number of features, y_i at subscript means y_i th component, $||\cdot||$ is 2-norm, and α is the regularization strength. For implementation details, see the `LogisticClassifier` class in `model.py`.

3.3.2 Ordinal Logistic Regression

Besides, notice that the values of fit have ordinal meanings, we also try the ordinal regression. Here we adapt a classic way to implement our ordinal regression classifier, which is constructed by two binary logistic regression model. For binary classifier A, we want it to learn P(`fit` > `Small`) i.e. $\mathrm{P}(y > 1)$. Similarly, let B to learn P(`fit` > `True to Size`), i.e. $\mathrm{P}(y > 2)$. We can acheive this by re-mapping the label into `{Small:0, True to Size:1, Large:1}` and `{Small:0, True to Size:0, Large:1}` while training classifier A and B respectively. Finally, we can get the desired probability by:

$$\begin{aligned} \mathrm{P}(y = 1) &= 1 - \mathrm{P}(y > 1) \\ \mathrm{P}(y = 2) &= \mathrm{P}(y > 2) - \mathrm{P}(y > 1) \\ \mathrm{P}(y = 3) &= \mathrm{P}(y > 2) \end{aligned}$$

3.4 Predicting Missing Labels

This section contains methods we hasn't fully implemented from scratch due to the limit of time, but we think it is valuable to present it here in the report.

Our target it to leverage pre-trained model on Transformer and fill in the sample's empty `fit` value. This approach is promising because `rating review` and `review_summary` are not taken advantage of in our training yet. And the relationship of `fit` and these columns, especially `review` and `review` summary, are great. However these records are highly textual and language models based on Transformer are doing well in these circumstances while pre-trained models could dramatically reduce training expenses.

3.4.1 Implementation Details

You might ask even if we already implement a Transformer from scratch, how can we load a pre-trained language model into our class object? The signature of class and function definition won't match! Well the point is there is bug inside `pickle` function which `torch.load()` make use of. By some hacking techniques like code injection we could eventually load the object into our own designed class object.

We basically convert the format into series of sentences containing both header of column and content in the cell, separated by special token. The we feed them into a language model (we use `roberta-base`) and trained the model. The performance of the model increases from 69% of filling `f1_score` to 84%.

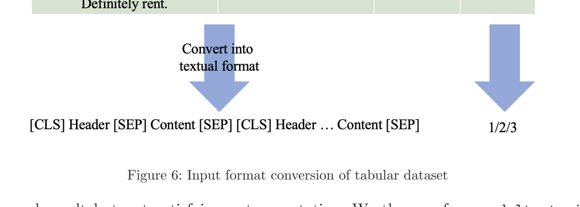

Figure 6: Input format conversion of tabular dataset

It is a good result but not satisfying out expectation. We then perform a `delta-tuning` trick onto the model and improve the result into 94% in filling `f1_score`. The tuning model is basically the trained encoder plus an additional randomly initialized MLP. When training the tuned model we need to freeze the encoder and only update the weight in the MLP layers.

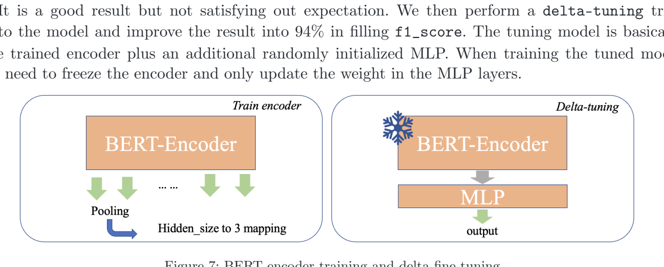

Figure 7: BERT encoder training and delta fine-tuning

By filling the empty `fit` value, we trained our models on filled dataset and each model get a fairly better result. However, as mentioned above, we have not implement it using only whitelist libraries, so we did not use the method in our submission model (though there is a file named `bert.py`, we did not import it in our main file).

4 EXPERIMENTS

4.1 Experimental Setup

In experiment phase, we employ the data cleanse and feature engineering method aforementioned. As for `item_name` features, we try two ways to deal with it: just drop it and employ one-hot encoder. For data imbalance problem, we utilize data augmentation, train split aggregation (n set to 3) separately and also use imbalance data to compare the performance. Finally we test on four models: multinomial logistic regression with GD, multinomial logistic regression with BFGS, ordinal logistic regression and random classifier as control group. For all the trainable model, we set the learning rate to 0.01, with max iteration 1000 to assure the gradient is zero in the end. Regularization constant α are all set to 0.1.

4.2 Results and Analysis

All the experiment results are presented at the table below, models performace are evaluated by the macro F1-score.

Table 6: Experimental results

Item Name	Balancing Tactics	LG (GD)	LG (BFGS)	OR	Random
Drop	Data Aug.	0.34	0.35	0.34	0.29
Drop	Split Aggr.	0.32	0.38	0.31	0.29
Drop	None	0.30	0.30	0.30	0.29
One-Hot	Data Aug.	0.51	0.53	0.52	0.29
One-Hot	Split Aggr.	0.54	0.52	0.51	0.29
One-Hot	None	0.30	0.30	0.30	0.29

From the perspective of processing `item_name`, experiment shows that one-hot encoding it can lift the model performance hugely. This result further proved our conclusion drawn in Data Analysis phase that `item_name` is the key point and it is even more important than the features we extracted from it.

From the perspective of balancing tactics, one certain thing is that both data augmentation and train split aggregation can address the imbalance problem to some extend. However, since two methods both can outperform the other while testing on some models, we cannot say which strategy is better. The performance of each definitely related to the hyperparameters in it. Since we have a thorough experiment on train split aggregation, and yet it reaches the best performance, we choose it as the final strategy.

From the perspective of models, those implemented by ourselves can make use of the training data and make predictions rationally compared to random classifier. But because all the models are based on logistic regression so all construct linear boundaries, it is tough for them to find complex relationships beneath the features, so the general performance of the models has its limitation.

Considering the model performance, finally we use the logistic regression model with BFGS optimizer, one-hot encoding the `iten_name` features and using train split aggregation to train our model.

5 CONCLUSION

In summary, we first cleanse the orginal data, fill the nan value and make values of each features in a uniform format, so that it can be input into the machine learning models. Then, after gaining a critical insight of given data, we propose Leveraging Item Sizes and Detecting User Prototypes to exploit latent features. Later, we used two ways to address the data imbalance problem respectively and both are proved to make sense. Finally, we use the Logistic Regression to construct our model and carry out experiment to compare different model performance. In addtion, we use BERT model to predict the missing label for about 30% of the training data. Despite we do not use it in the end because of the library whitelist, we find it practical in the real-world recommending tasks. Our final model reached the macro F1-score of 50%, which meets our expectation basically.

However, the undertaking of the project was beset with challenges and difficulties.

Our team engaged in a comprehensive and collaborative effort to thoroughly explore the dataset, conduct relevant research on prior literature, and implement and analyze experimental methods. Each member of the team contributed to the fullest of their abilities. While the committed code ultimately presented only represents a subset of the full scope of our implemented solutions and the methods employed in the final submission may be more simplistic than our initial exploratory approaches, we felt quite content with our project due to this extend of devotion. Despite instances of disappointment and frustration arising from a lack of performance improvements in some of our more elaborate designs, the team remained resilient and actively encouraged one another to persevere in our efforts. We are deeply greatful to every member of our team, as well as our instructor and TAs, for their unwavering commitment and dedication to this course and project!