Homogenous Ensemble Learning in Highly Unbalanced Data

Name(s): Kaiwen Bian & Bella Wang

Website Link: https://kevinbian107.github.io/ensemble-unbalanced-data/

```
In [ ]:  # for eda and modeling
         import pandas as pd
         import numpy as np
         pd.options.plotting.backend = 'plotly'
         from utils.dsc80_utils import *
         from itertools import chain
```

Step 1: Introduction

```
In [ ]:  interactions = pd.read_csv('food_data/RAW_interactions.csv')
         recipes = pd.read_csv('food_data/RAW_recipes.csv')
```

Step 2: Data Cleaning and Exploratory Data Analysis

Merging

Initial merging is needed for the two dataset to form 1 big data set

1. Left merge the recipes and interactions datasets together.
2. In the merged dataset, fill all ratings of 0 with np.nan. (Think about why this is a reasonable step, and include your justification in your website.)
3. Find the average rating per recipe, as a Series.
4. Add this Series containing the average rating per recipe back to the recipes dataset however you'd like (e.g., by merging). Use the resulting dataset for all of your analysis. (For the purposes of Project 4, the 'review' column in the interactions dataset doesn't have much use.)

Transformation

1. Some columns, like `nutrition`, contain values that look like lists, but are actually strings that look like lists. We turned the strings into actual columns for every unique value in those lists
2. Convert to list for `steps`, `ingredients`, and `tags`
3. Convert `date` and `submitted` to Timestamp object and rename as `review_date` and `recipe_date`
4. Convert Types
5. Drop same `id` (same with `recipe_id`)
6. Replace 'nan' with np.NaN

Type Logic

1. `String` : [name, contributor_id, user_id, recipe_id,]
   - quantitative or qualitative, but cannot perform mathamatical operations (**quntitative discrete**)
   - `name` is the name of recipe
   - `contributor_id` is the author id of the recipe *(shape=7157)*
   - `recipe_id` is the id of teh recipe *(shape=25287)*
     - `id` from the original dataframe also is the id of the recipe, dropped after merging

- `user_id` is the id of the reviewer *(shape=8402)*
2. `List` : [tags, steps, description, ingredients, review]
   - qualitative, no mathamatical operation (**qualitative discrete**)
3. `int` : [n_steps, minutes, n_ingredients, rating]
   - quantitative mathamatical operations allowed (**quantitative continuous**)
4. `float` : [avg_rating, calories, total_fat sugar, sodium, protein, sat_fat, carbs]
   - quantitative mathamatical operations allowed (**quantitative continuous**)
5. `Timestamp` : [recipe_date, review_date]
   - quantitative mathamatical operations allowed (**quantitative continuous**)

Below are the full implementation of `initial` , which does the merge conversion, then `transform` , whcih carries out the neccessary transformation described above

```
In [ ]:  def initial(df):
             '''Initial claeaning and megrging of two df, add average ratings'''
             # fill 0 with np.NaN
             df['rating'] = df['rating'].apply(lambda x: np.NaN if x==0 else x)

             # not unique recipe_id
             avg = df.groupby('recipe_id')[['rating']].mean().rename(columns={'rating':'avg_rating'})
             df = df.merge(avg, how='left', left_on='recipe_id',right_index=True)
             return df

         def transform_df(df):
             '''Transforming nutrition to each of its own catagory,
             tags, steps, ingredients to list,
             submission date to timestamp object,
             convert types,
             and remove 'nan' to np.NaN'''

             # Convert nutrition to its own caatgory
             data = df['nutrition'].str.strip('[]').str.split(',').to_list()
             name = {0:'calories',1:'total_fat',2:'sugar',3:'sodium',4:'protein',5:'sat_fat',6:'carbs'}
             #zipped = data.apply(lambda x: list(zip(name, x)))
             new = pd.DataFrame(data).rename(columns=name)

             df = df.merge(new,how='inner',right_index=True, left_index=True)
             df = df.drop(columns=['nutrition'])

             # Convert to list
             def convert_to_list(text):
                 return text.strip('[]').replace("'",'').split(', ')

             df['tags'] = df['tags'].apply(lambda x: convert_to_list(x))
             df['ingredients'] = df['ingredients'].apply(lambda x: convert_to_list(x))

             # it's correct, just some are long sentences, doesn't see "'", notice spelling
             df['steps'] = df['steps'].apply(lambda x: convert_to_list(x)) #some white space need to be h

             # submission date to time stamp object
             format ='%Y-%m-%d'
             df['submitted'] = pd.to_datetime(df['submitted'],format=format)
             df['date'] = pd.to_datetime(df['date'],format=format)

             # drop not needed & rename
             df = df.drop(columns=['id']).rename(columns={'submitted':'recipe_date','date':'review_date'}

             # Convert data type
             df[['calories','total_fat','sugar',
                 'sodium','protein','sat_fat','carbs']] = df[['calories','total_fat','sugar',
                                                              'sodium','protein','sat_fat','carbs']].asty

             df[['user_id','recipe_id','contributor_id']] = df[['user_id','recipe_id','contributor_id']].

             # there are 'nan' values, remove that
```

```python
    for col in df.select_dtypes(include='object'):
        df[col] = df[col].apply(lambda x: np.NaN if x=='nan' else x)

    return df
```

In [ ]:
```python
merged = recipes.merge(interactions, how='left', left_on='id', right_on='recipe_id')
cleaned = (merged
           .pipe(initial)
           .pipe(transform_df))
```

In [ ]:
```python
display_df(cleaned)
```

	name	minutes	contributor_id	recipe_date	...	sodium	protein	sat_fat	carbs
0	1 brownies in the world best ever	40	985201	2008-10-27	...	3.0	3.0	19.0	6.0
1	1 in canada chocolate chip cookies	45	1848091	2011-04-11	...	22.0	13.0	51.0	26.0
2	412 broccoli casserole	40	50969	2008-05-30	...	32.0	22.0	36.0	3.0
...	...	...	...	...	...	...	...	...	...
234426	cookies by design sugar shortbread cookies	20	506822	2008-04-15	...	4.0	4.0	11.0	6.0
234427	cookies by design sugar shortbread cookies	20	506822	2008-04-15	...	4.0	4.0	11.0	6.0
234428	cookies by design sugar shortbread cookies	20	506822	2008-04-15	...	4.0	4.0	11.0	6.0

234429 rows × 23 columns

Now this code would be used later on when we need to groupby using the `recipe_id` column or the `user_id` column for different purposes. The handling for different columns are also defined as below, which is diffeernt according to waht we need the columns are for later on in the modeling process.

In [ ]:
```python
def group_recipe(df):
    func = lambda x: list(x)
    check_dict = {'minutes':'mean', 'n_steps':'mean', 'n_ingredients':'mean',
                  'avg_rating':'mean', 'rating':'mean', 'calories':'mean',
                  'total_fat':'mean', 'sugar':'mean', 'sodium':'mean',
                  'protein':'mean', 'sat_fat':'mean', 'carbs':'mean',
                  'steps':'first', 'name':'first', 'description':'first',
                  'ingredients':func, 'user_id':func, 'contributor_id':func,
                  'review_date':func, 'review':func,  'recipe_date':func,
                  'tags':lambda x: list(chain.from_iterable(x))}

    grouped = df.groupby('recipe_id').agg(check_dict)
    #grouped['rating'] = grouped['rating'].astype(int)

    return grouped

def group_user(df):
    '''function for grouping by unique user_id and concating all steps/names/tags of recipe and

    return (df #[df['rating']==5]
            .groupby('user_id')['steps','rating','name','tags','minutes','calories','description
            .agg({'steps':lambda x: list(chain.from_iterable(x)),
                  'name':lambda x: list(x),
                  'tags':lambda x: list(chain.from_iterable(x)),
                  'rating':'mean',
                  'minutes':'mean',
                  'calories':'mean',
                  'description':lambda x: list(x),
                  'n_ingredients':'mean',
```

Processing math: 100%

```
                        'ingredients':lambda x: list(chain.from_iterable(x)),
                        'contributor_id':lambda x: list(x),
                        'review':lambda x: list(x),
                        })
        )
```
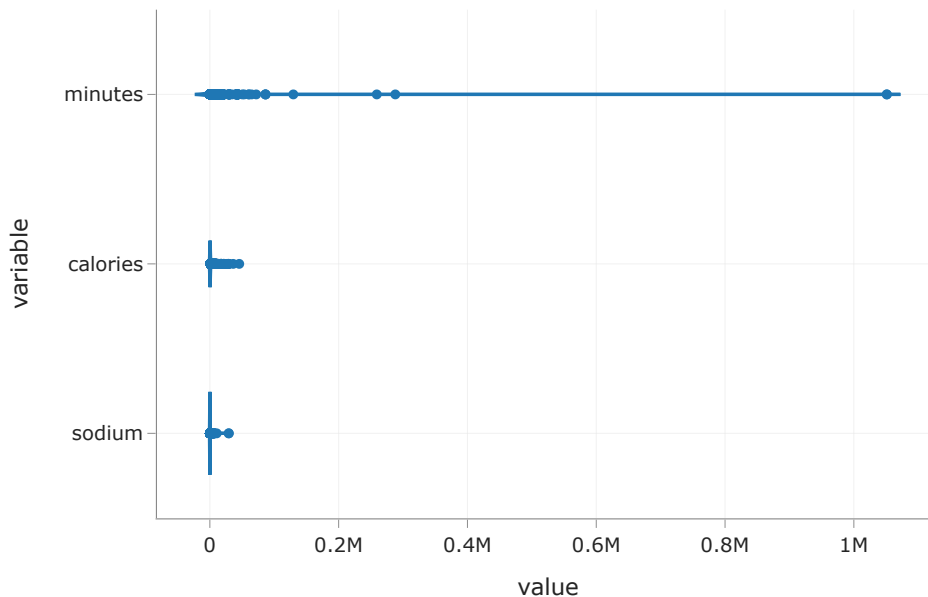
Univariate & Bivariate Analysis

Okay, after data cleaning, let's draw some graph to see what kind of data we are dealing with

```
In [ ]: px.violin(cleaned, x=['sodium','calories','minutes'])
```


Looks like that our data have a lot of outliers! we might want to write a function to deal with that. Here we are writing the function `outlier` , which will be used quite often later on.

```
In [ ]: def outlier(df):
            '''take care of outliers in the data frame'''
            # Remove outlier in graph dierctly

            check = ['minutes', 'n_steps', 'n_ingredients', 'calories', 'total_fat', 'sugar', 'sodium',
            for col in check:#df.select_dtypes(include='number'):
                q_low = df[col].quantile(0.01)
                #print(q_low)
                q_hi  = df[col].quantile(0.99)
                #print(q_hi)
                df = df[(df[col]<q_hi) & (df[col]>q_low)]

            return df #same name so update df
```
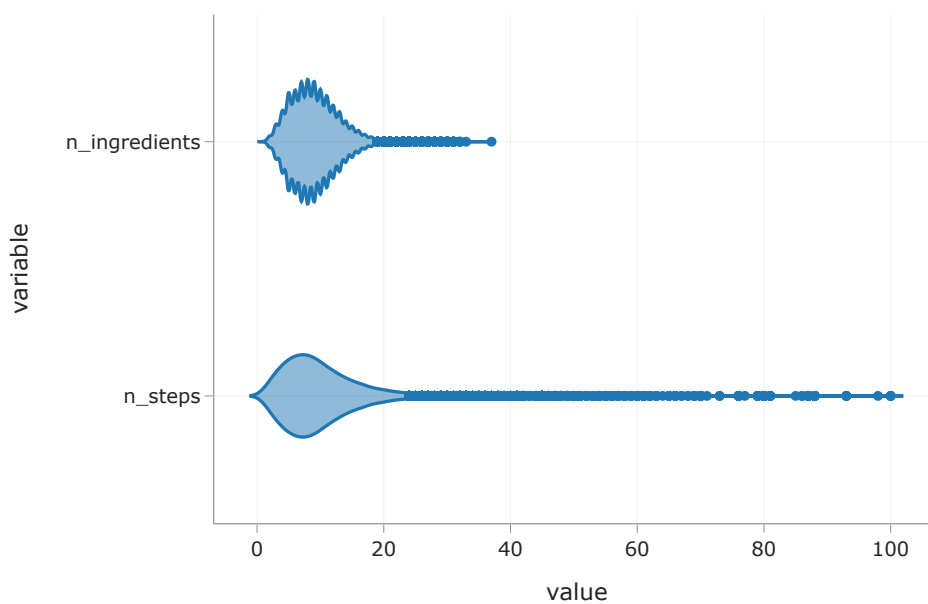
```
In [ ]: px.violin(cleaned, x='rating')
```

Processing math: 100%
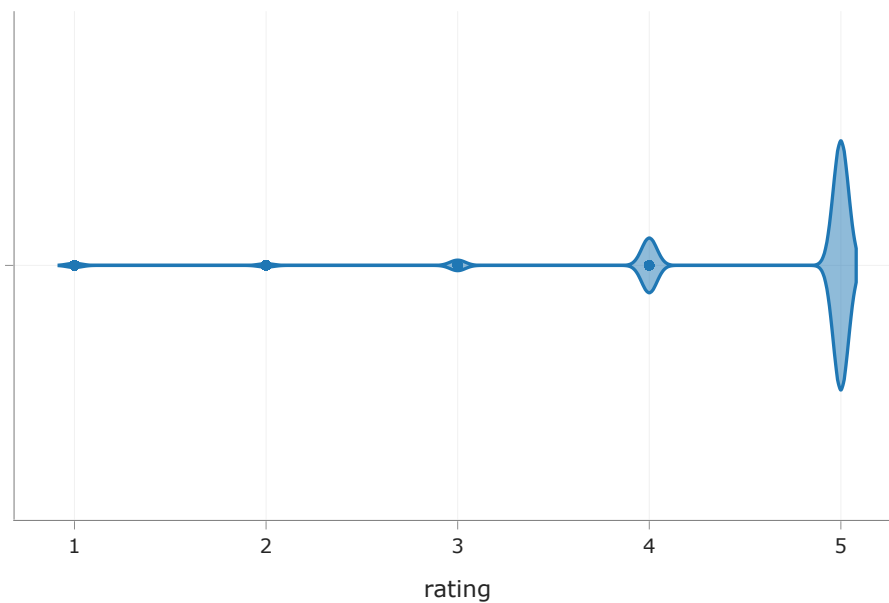
Looks like the data are kind of **unbalanced** in `rating` (at this point, we thought that this wouldn't effect our modle too much, but it turns out later to be one of the main challenge that we need to deal with during the moeling phase)

```
In [ ]:  px.violin(cleaned, x=['n_steps', 'n_ingredients'])
```


Seems like there is a **threshold point** for `n_ingredients` and `n_steps`, this will be utilized later in our **feature engineering** section

```
In [ ]:  px.scatter_3d(cleaned, x='sugar', y='total_fat', z='rating',)
```

Processing math: 100%
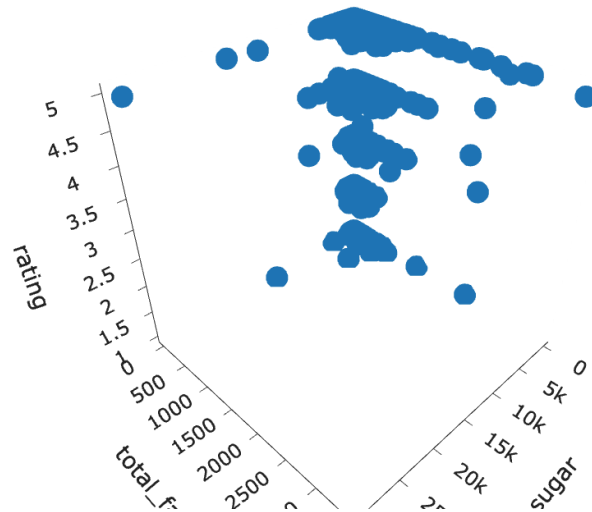
It also seems like more `sugar` and more `total_fat` (transformed from `nutrition`) seems to be related to higher `rating` ! This is quite suprising!

We actually made more edas and feature engineering with **textual features**, but we will introduce those later in the section as it is much more relevant to our modeling process.

Step 3: Assessment of Missingness

There are data missing! Why is that happening?

```
In [ ]:  from scipy.stats import ks_2samp
         df = (cleaned
              .pipe(group_recipe)
              .pipe(outlier))
```

We are specifically working with the version of the data set that have been grouped by with `recipe_id` to check the missingness, each `recipe_id` in this case would be unique. We can start with checking whcih column is missing. For the easiness of graphing, we will first slice out the outliers in each of the numerical columns using `outlier` function, which slices out ouliers that's out of the 99th percentile of the dataset

```
In [ ]:  display_df(pd.DataFrame(df.isna().sum()), 23)
```

	0
minutes	0
n_steps	0
n_ingredients	0
avg_rating	1679
rating	1679
calories	0
total_fat	0
sugar	0
sodium	0
protein	0
sat_fat	0
carbs	0
steps	0
name	0
description	48
ingredients	0
user_id	0
contributor_id	0
review_date	0
review	0
recipe_date	0
tags	0

NMAR Analysis

From the first step analysis by just observing the data and looking at the website, it seems like the column `name` is **Missing Completely At Random (MCAR)** as it doesn't seem to be fitting to any of the all three other catagories.

However, on the other hand, the `rating` column seems to be **Not Missing At Random (NMAR)** becuase from what the website is showing, some people just didn't give rating, so the rating itself doesn't exist during the data collection process, so it makes sense for it to be null. We manually added `np.NaN` into the data set where previously it was filled a zero in the data set. Since `avg_ratng` is calculated from using the `rating` column, `avg_rating` would then be **Missing At Random (MAR)** dependent on `rating`.

One interesting one to analyze is `description`, because it is hard to say directly how it may be correlated to any other columns in this data set, we suspect it to be **MAR**, but we will prove it to be **MAR** in the next section.

MAR Analysis

Decision Rule for `description`

Let's assume that the missingess of `description` column is related to the `col` column for **continuous** columns, wouldn't depend on **discrete** columns.

The below functions are used for conducting graphing for checking potential MAR columns and also for conducting permutation testing

```python
In [ ]: def create_kde_plotly(df, group_col, group1, group2, vals_col, title=''):
            '''Create the kde plot for checking column potential dependencies'''
            fig = ff.create_distplot(
                hist_data=[df.loc[df[group_col] == group1, vals_col], df.loc[df[group_col] == group2, va
                group_labels=[group1, group2],
                show_rug=False, show_hist=False
            )
            return fig.update_layout(title=title)

        def mar_check_continuous(df,miss_col, dep_col):
            '''Full checking mar by simulating mar data then graphing it,
            miss_col must be catagorical and dep_col must be continuous'''

            missing = df[miss_col].isna()
            df_missing = df.assign(mar_missing = missing)[['mar_missing', dep_col]]

            fig = create_kde_plotly(df_missing, 'mar_missing', True, False, dep_col, title=f'MAR Graph o

            return fig.show()
```

```python
In [ ]: def permutation_ks(df, miss_col, dep_col, rep):
            '''conduct permutation testing for testing mar in data frame '''

            def permutation_test(df, rep, dep_col):
                '''test_statistics is the KS statistics'''

                # line of missing of description that may base on dep_col?
                observe = ks_2samp(df_missing.query('mar_missing')[dep_col],
                                   df_missing.query('not mar_missing')[dep_col]).statistic

                # making a distrbution where missing of description does not depend on dep_col
                n_repetitions = rep
                null = []
                for _ in range(n_repetitions):
                    with_shuffled = df.assign(shuffle = np.random.permutation(df['mar_missing']))
                    difference = ks_2samp(with_shuffled.query('shuffle')[dep_col],
                                          with_shuffled.query('not shuffle')[dep_col]).statistic
                    null.append(difference)
                return observe, null

            missing = df[miss_col].isna()
            df_missing = df.assign(mar_missing = missing)[['mar_missing', dep_col]]

            observe, null = permutation_test(df_missing, rep, dep_col)

            fig = px.histogram(pd.DataFrame(null), x=0, histnorm='probability', title=f'KS Distribution
            fig.add_vline(x=observe, line_color='red', line_width=1, opacity=1)

            p = (observe <= null).mean()
            print(f'p_value is {p}')

            return fig.show()
```
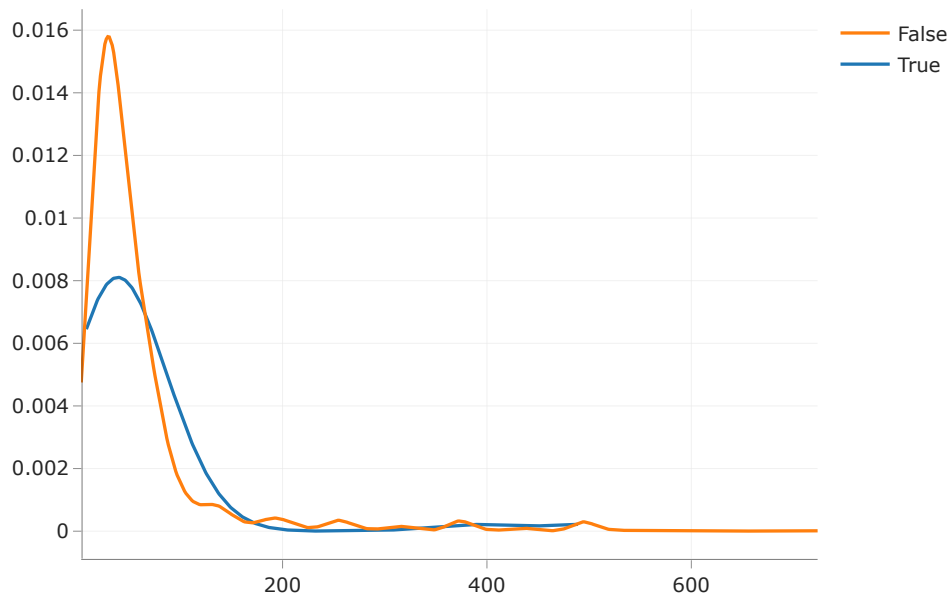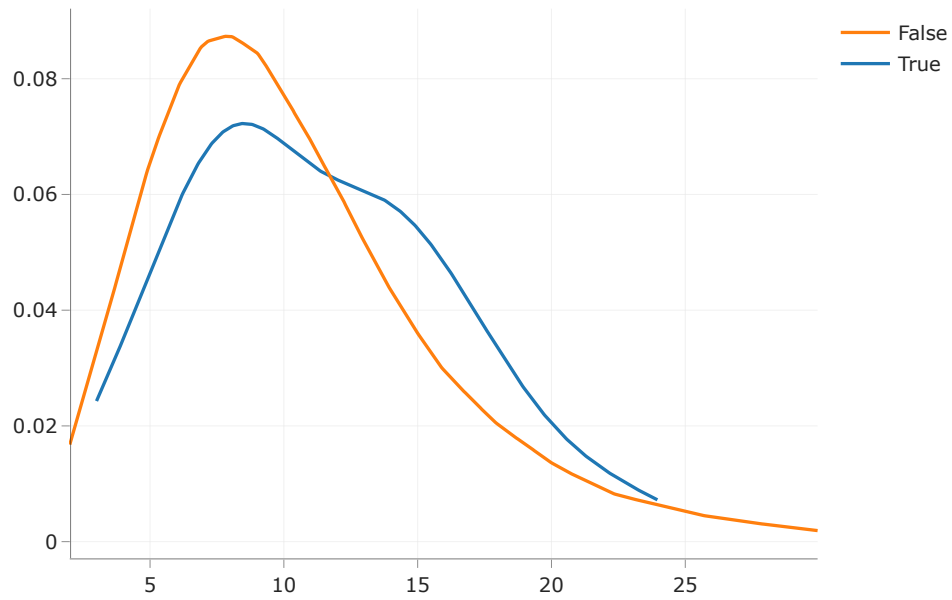
```python
In [ ]: for col in df.drop(columns=['avg_rating','rating']).select_dtypes(include='number'):
            mar_check_continuous(df, 'description', col)
```
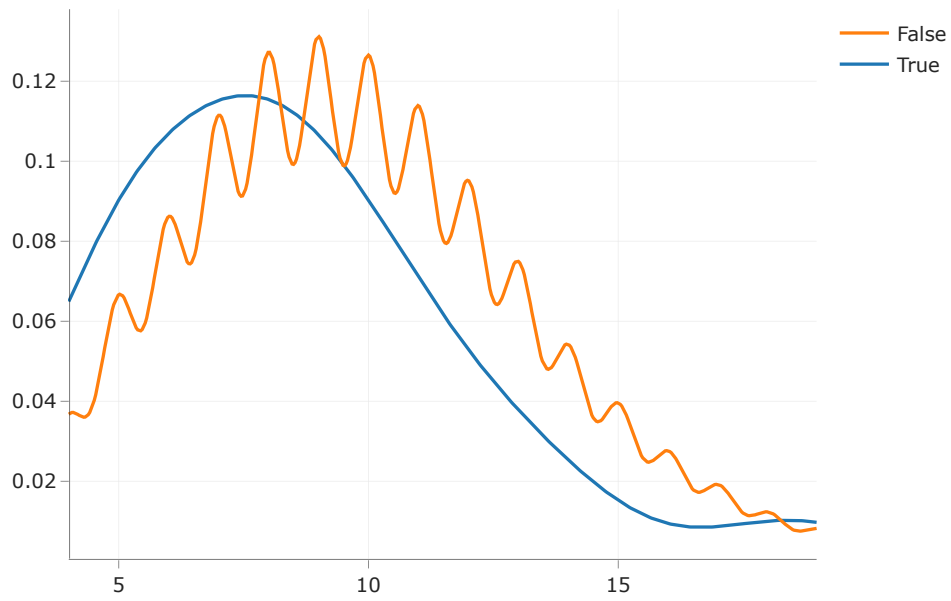
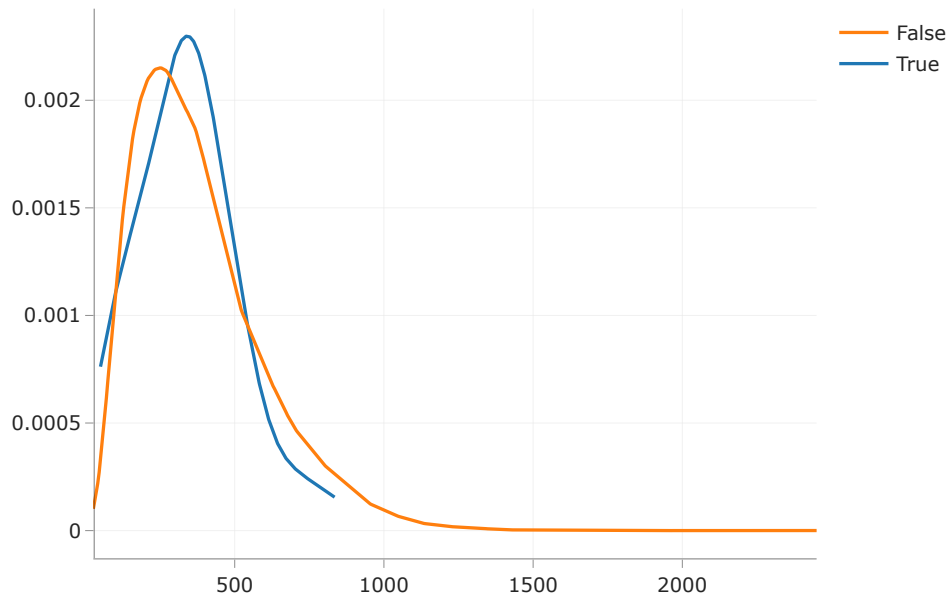## MAR Graph of description Dependent on minutes

Legend: False, True

MAR Graph of description Dependent on n_steps

Legend: False, True

MAR Graph of description Dependent on n_ingredients

MAR Graph of description Dependent on calories

Processing math: 100%

MAR Graph of description Dependent on total_fat
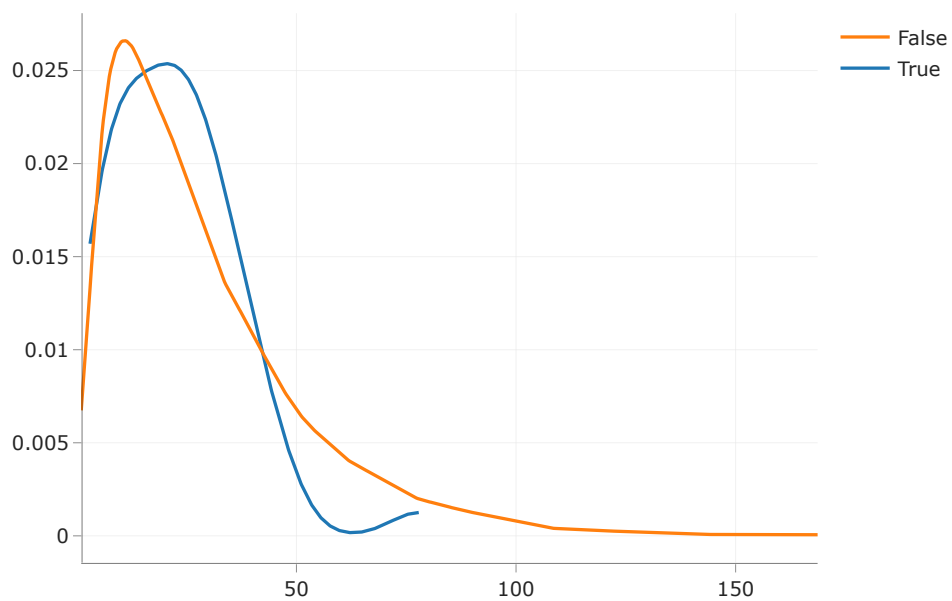
MAR Graph of description Dependent on sugar

Processing math: 100%
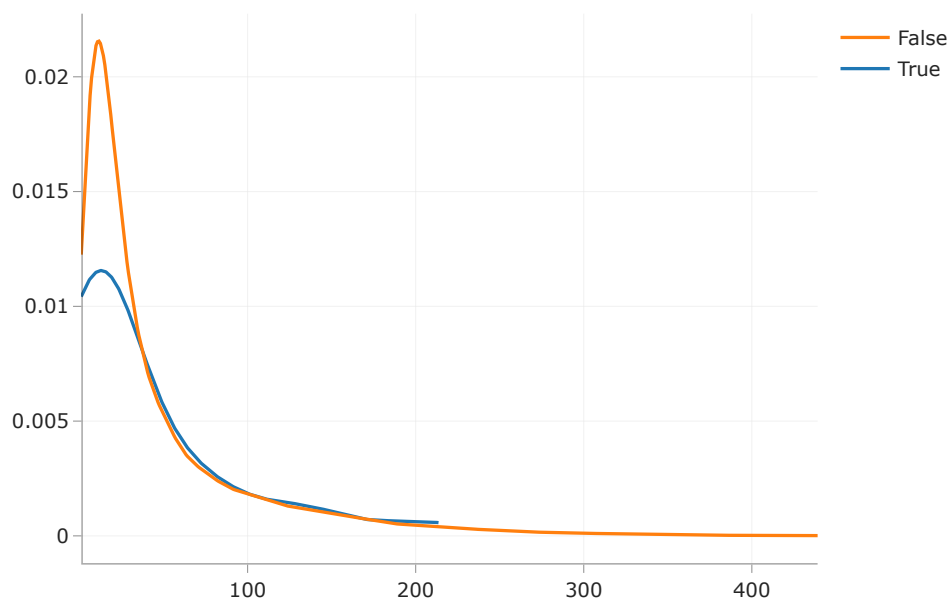
MAR Graph of description Dependent on sodium
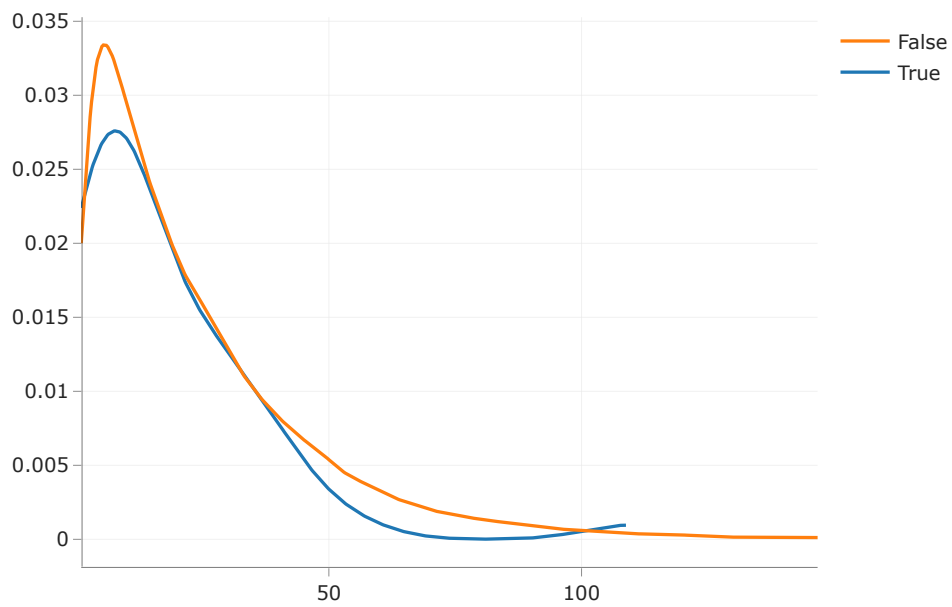
MAR Graph of description Dependent on protein

MAR Graph of description Dependent on sat_fat

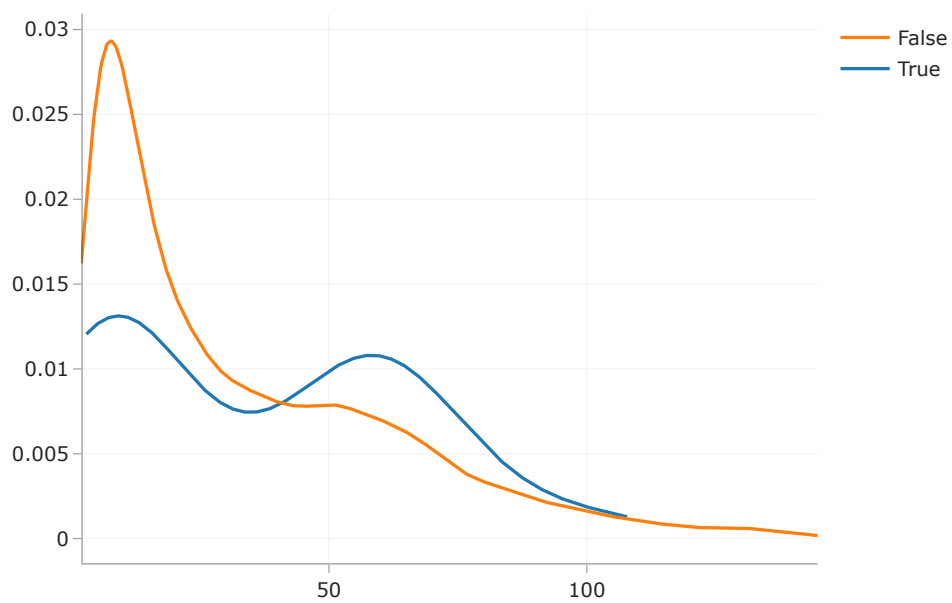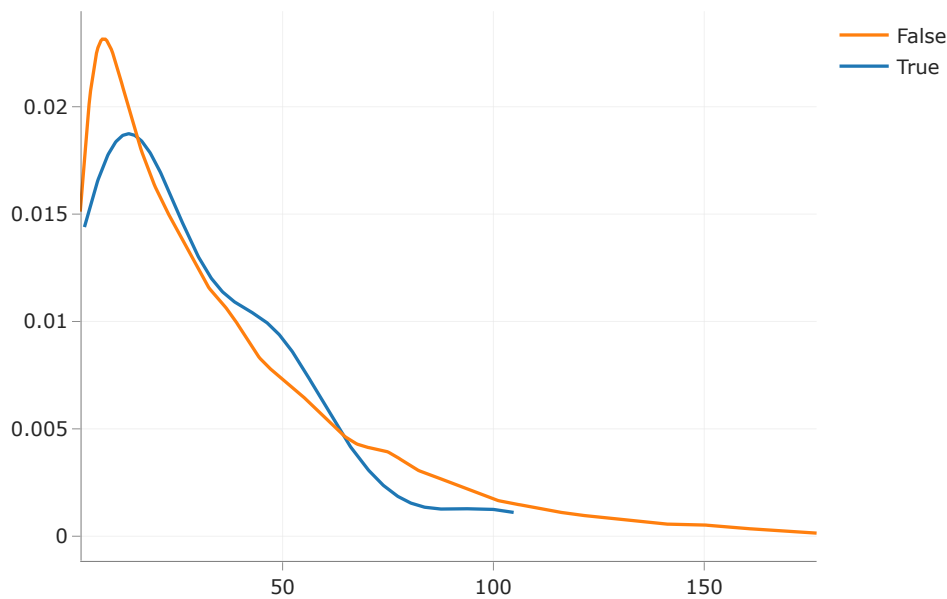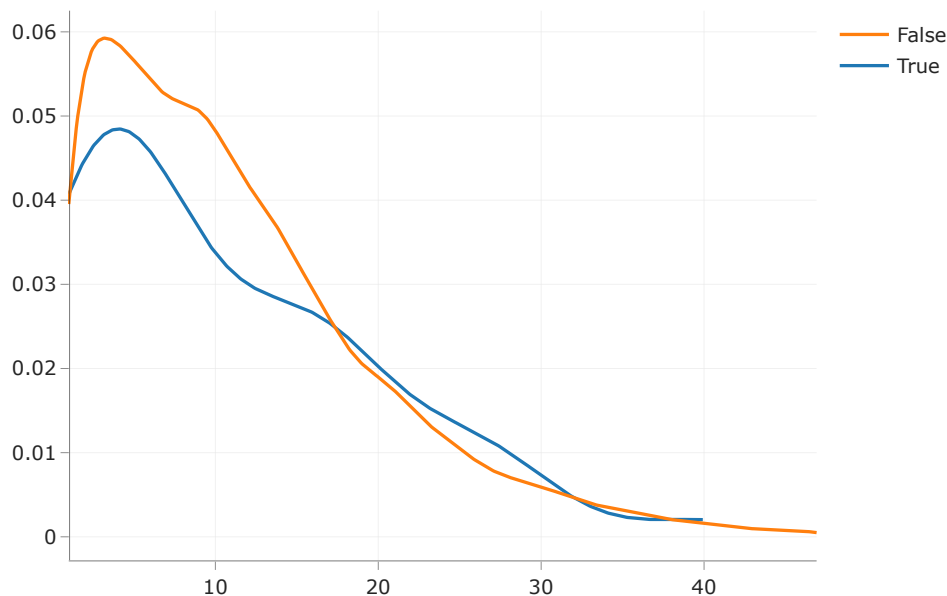## MAR Graph of description Dependent on carbs

`description` seems to also depend on `n_ingredients` . This is a very interesting graph because looks like the graph **shape** is quite different with the **mean** the same, instead of using permutation test statistics that involves **mean** we use **K-S statistics** insteaad (we have also down a test using differences in mean as well, which fail to identify any results).

Permutation Testing Using K-S Statistics

Now we want to perform permutation testing with each of the continuous variable within the data set (assuming that the missingness of `description` depends on them) and plot the distribution

We decide to use a testing threshold of $p = 0.05$

```
In [ ]:  for col in df.drop(columns=['avg_rating','rating']).select_dtypes(include='number'):
             permutation_ks(df, 'description', col, 1000)
```

Processing math: 100% lue is 0.936

KS Distribution for Null description_col is dependent on minutes_col
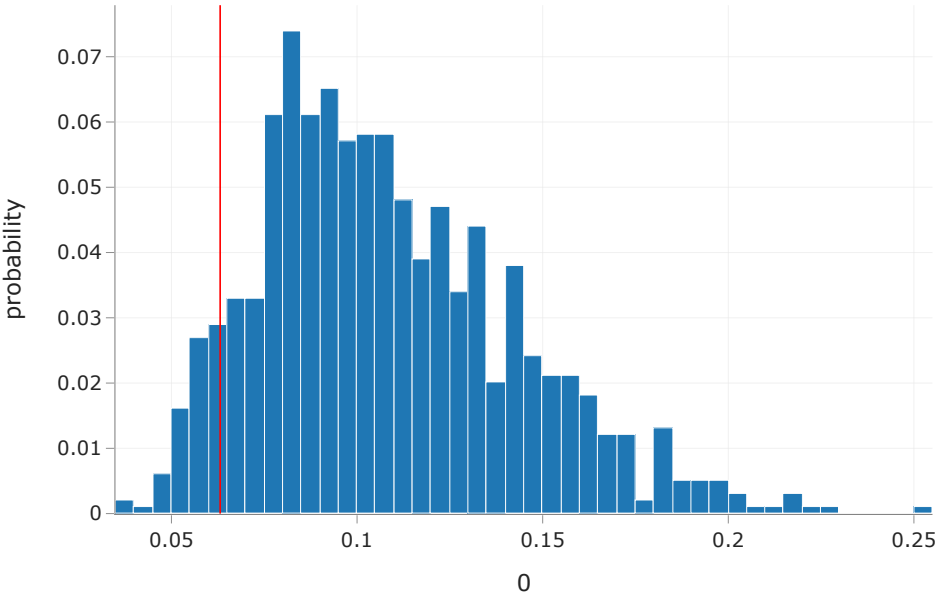
p_value is 0.223

KS Distribution for Null description_col is dependent on n_steps_col

p_value is 0.011

Processing math: 100%
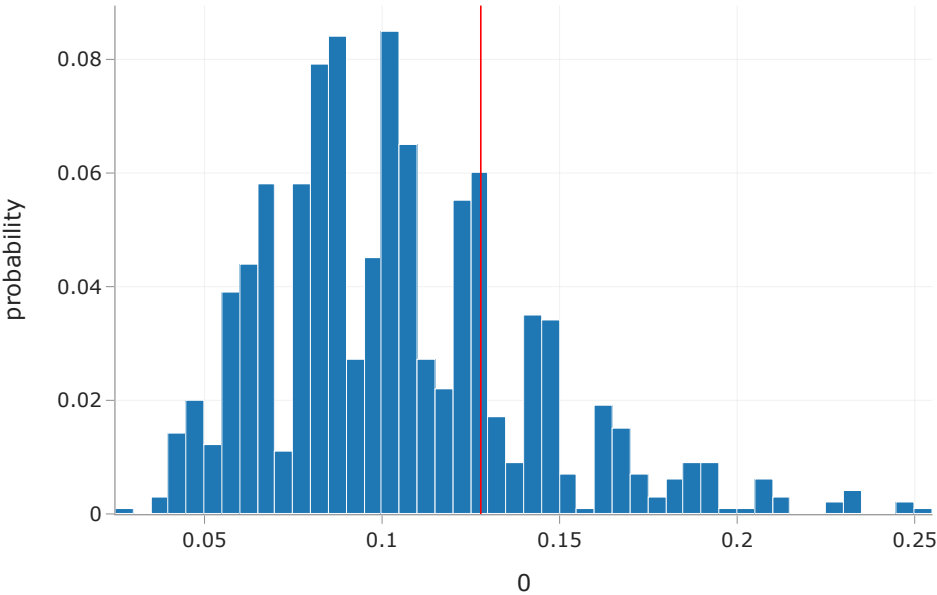
S Distribution for Null description_col is dependent on n_ingredients_c

p_value is 0.417

KS Distribution for Null description_col is dependent on calories_col

p_value is 0.067

Processing math: 100%

KS Distribution for Null description_col is dependent on total_fat_col

p_value is 0.031

KS Distribution for Null description_col is dependent on sugar_col
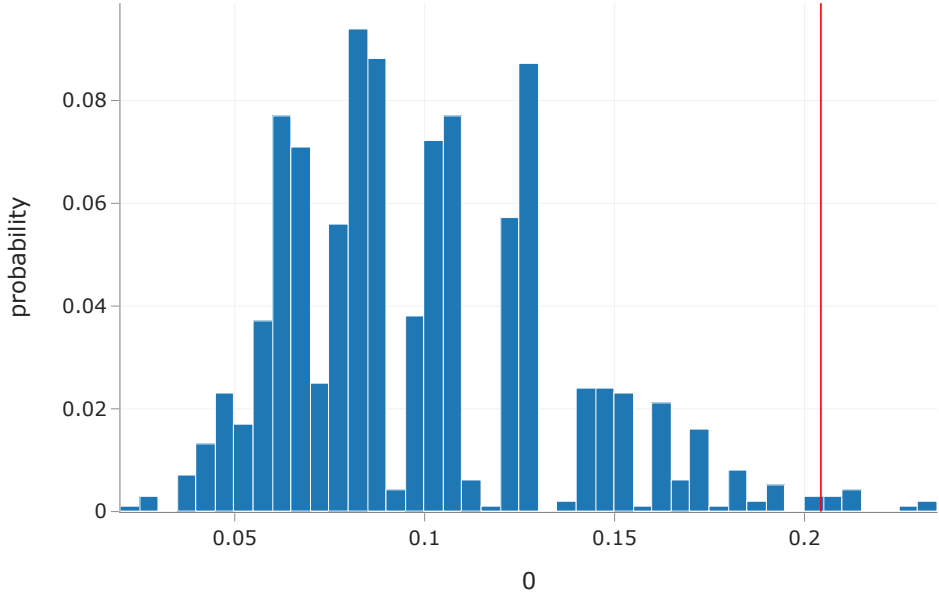

p_value is 0.029

Processing math: 100%

KS Distribution for Null description_col is dependent on sodium_col

p_value is 0.045

KS Distribution for Null description_col is dependent on protein_col

p_value is 0.331

Processing math: 100%
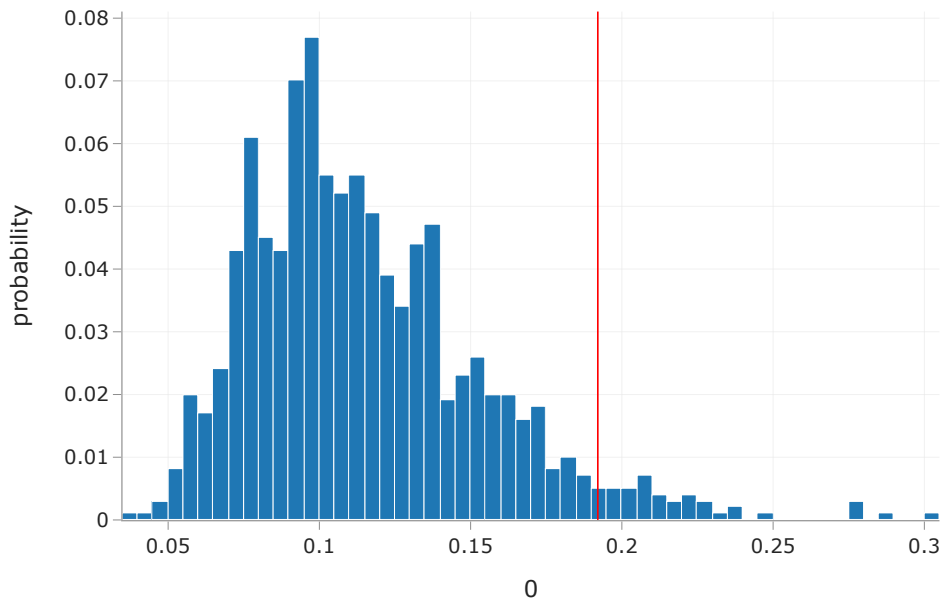
KS Distribution for Null description_col is dependent on sat_fat_col
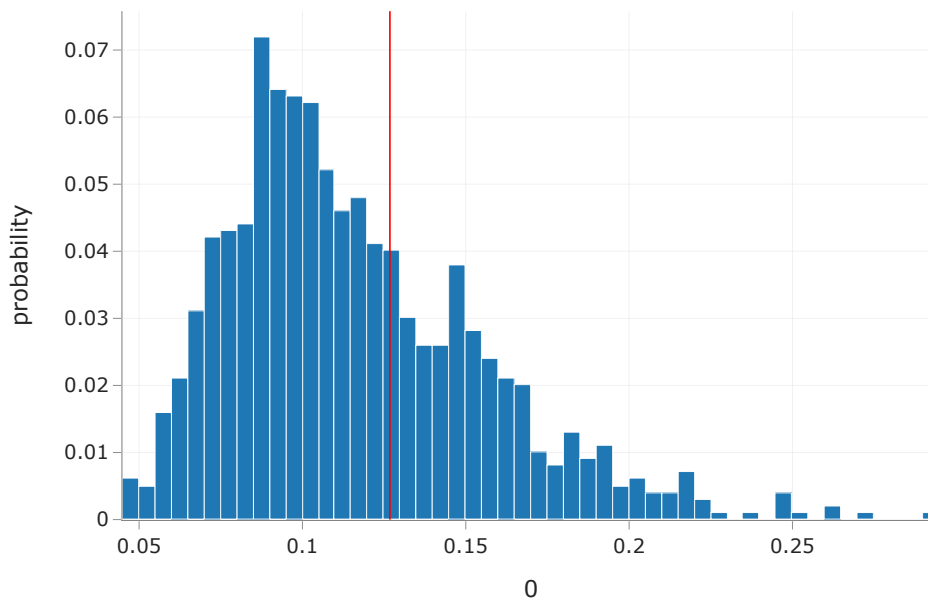
`p_value is 0.099`

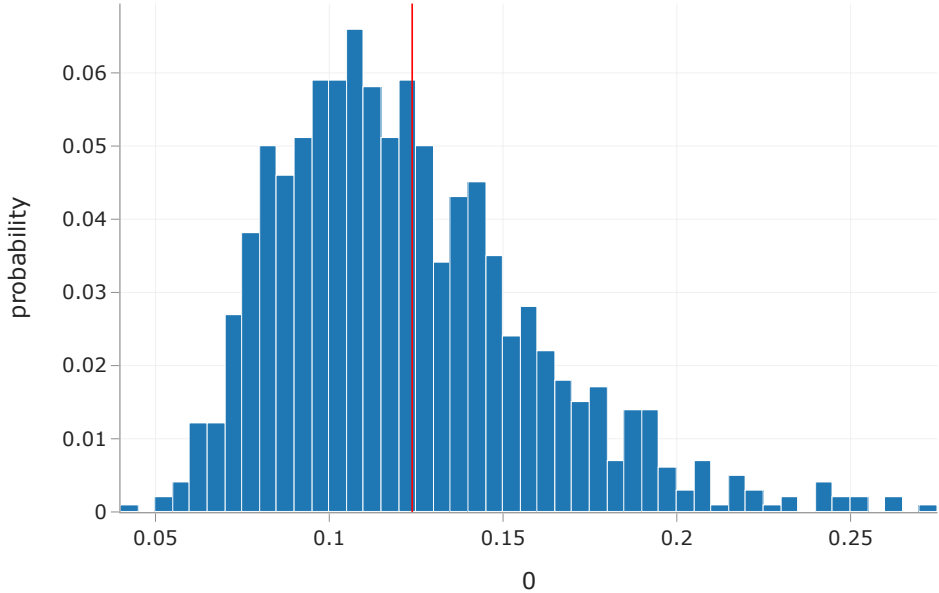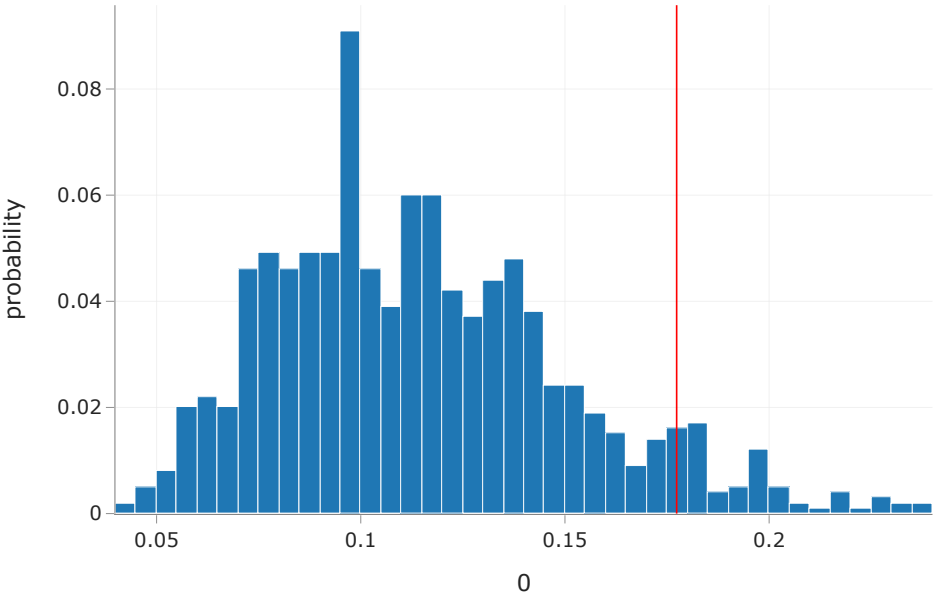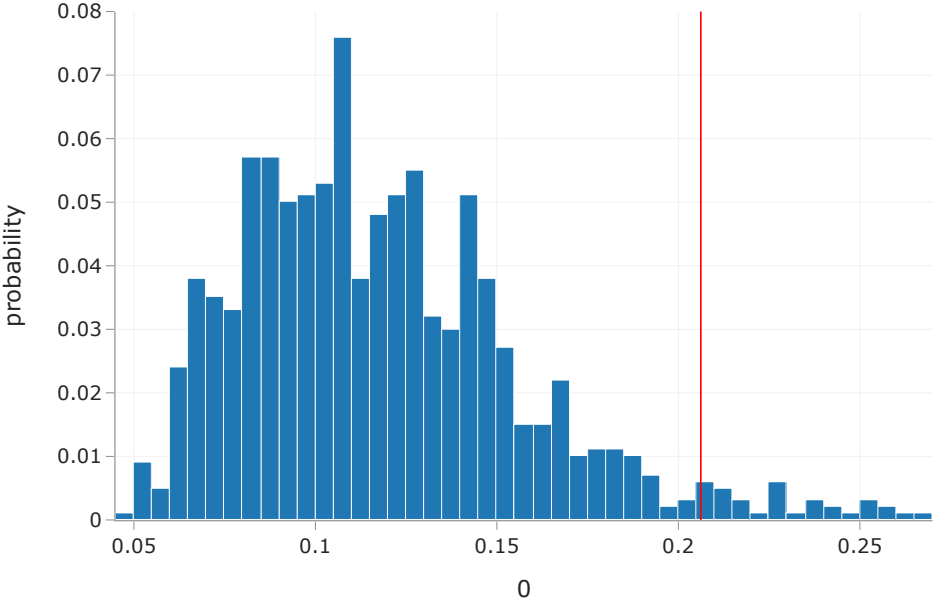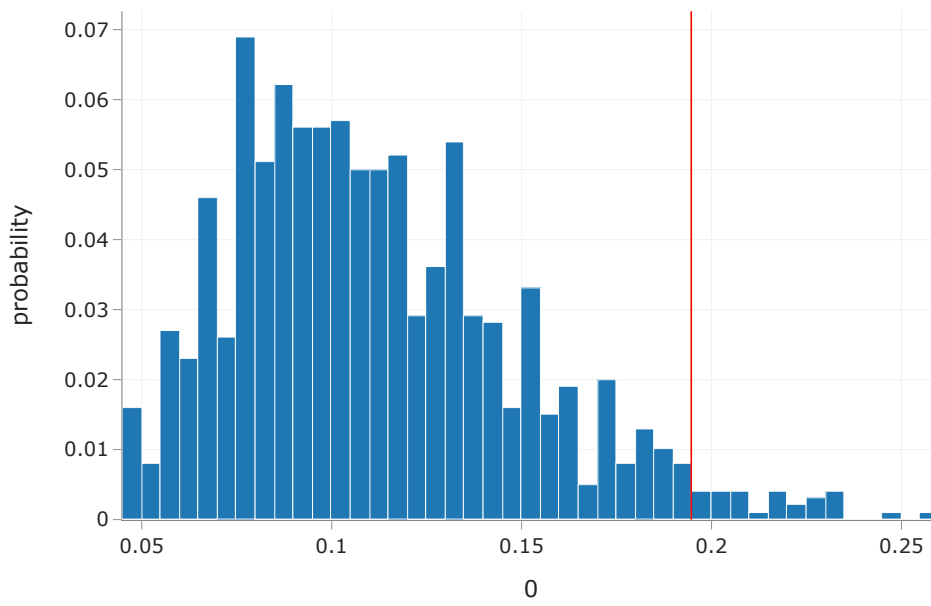KS Distribution for Null description_col is dependent on carbs_col
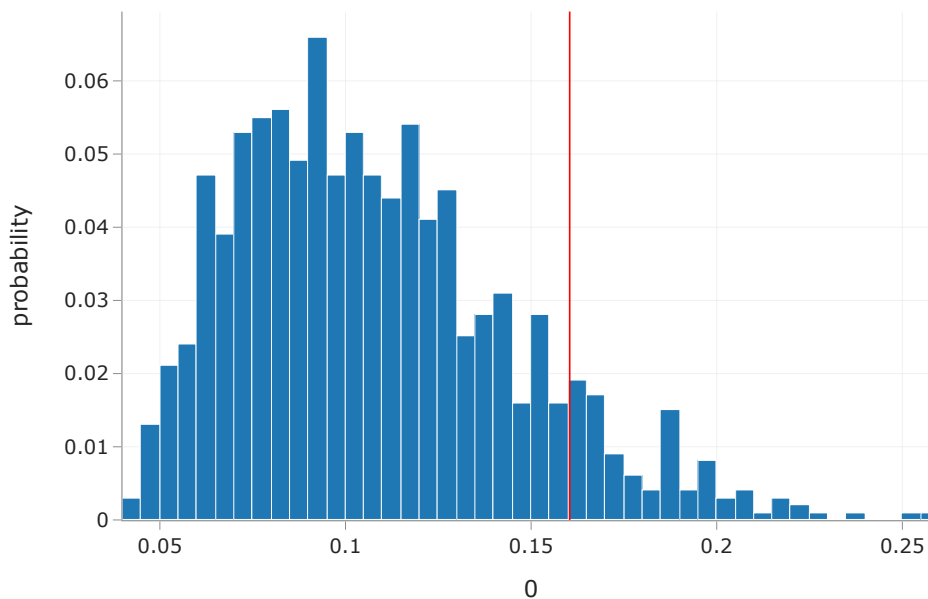
- seems like missingess for `description` is related to `n_ingredients`
- seems like missingness in `description` is not related to `calories` or `n_steps`

Step 4: Hypothesis Testing

For this section, we will be working with the same data frame that was used in the missingness mechanism section, so a data frame that is grouped by `recipe_id`.

Since we want to do certain textual feature analysis for our predictive model, we were wondering whether `TF-IDF` of the `description` columns would actually play a role in deternmining the `rating` of an recipe. This can be deemed as a mini-warmup for our modeling procedure later on.

Processing math: 100%

Term Frequency Inverse Document Frequency

`TF-IDF` is a very naive but common and well performing technique that people use to understand textual features. It essentially meausres the **how important** an word t is for an sentence in comparison with all sentences in the document. The `TF-IDF` Formula is a as follows:

$$\text{tfidf}(t, d) = \text{tf}(t, d) \cdot \text{idf}(t)$$

$$= \frac{\backslash\# \text{ of occurrences of } t \text{ in } d}{\text{total } \backslash\# \text{ of words in } d} \cdot \log\left(\frac{\text{total } \backslash\# \text{ of documents}}{\backslash\# \text{ of documents in which } t \text{ appears}}\right)$$

We will be using `TfidfVectorizer` to help our calculation

```
In [ ]: # import here first as it is usef for computing TF-IDF
        from sklearn.feature_extraction.text import TfidfVectorizer
```

In here we are just splitting the data frame into `high_score` and `low_score`

```
In [ ]: df = df[['name','description','tags','steps','ingredients','contributor_id','rating']] # avg_rat
        df_high = df[df['rating']>=4]
        df_low = df[df['rating']<4]

        lst_high = df_high['description'].explode().astype(str)
        lst_low = df_low['description'].explode().astype(str)
```

In this step we are first using the `TfidfVectorizer` from `sk_learn` to compute the `TF-IDF` table

```
In [ ]: count_high = TfidfVectorizer()
        count_low = TfidfVectorizer()
        count_high.fit(lst_high.values)
        count_low.fit(lst_low.values)

        high_tfidf = pd.DataFrame(count_high.transform(lst_high.values).toarray(),
                            columns=count_high.get_feature_names_out()
                            )

        low_tfidf = pd.DataFrame(count_low.transform(lst_low.values).toarray(),
                            columns=count_low.get_feature_names_out()
                            )
```

Differences in Max for TF-IDF

We want to see whether the distibution of `high_rated` recipes and the distribution of `low_rated` recipes actually come from the same distribution. Thus, we will be performing a **permutation test** here with the following hypothesis:

- **Null hypothesis**: There **are no** differences in the distribution for the `high_rated` recipes and low_rated` recipes.
- **Alternative hypothesis**: There **are** differences in the distribution for the `high_rated` recipes and low_rated` recipes.

We decide to use a testing threshold of $p = 0.05$

As for the **test statistics**, we actually have many options, but they all circles around the **differences** of something:

- Using `sum` -> longer sentences have greater sum
- Using `mean` -> very easy to be influenced by outlier
- Using `partial-mean` -> get the most essence part of the sentence, however, complexity too high because of the sorting
- Using `max` -> most important one word's TF-IDF

With all these considerations, we pick our test statistics to be **differences in max of TF-IDF for each sentence**

This section provide a **solid prove** of why we are using TF-IDF as a feature for our predictive model!

```python
tfidf_max_high = high_tfidf.max(axis=1)
tfidf_max_low = low_tfidf.max(axis=1)

max_high = df_high.reset_index().assign(tfidf = tfidf_max_high, good=True)
max_low = df_low.reset_index().assign(tfidf = tfidf_max_low, good=False)

big_df = pd.concat([max_high, max_low], axis=0)
big_df
```

Out[ ]:

	recipe_id	name	description	tags	...	contributor_id	rating	tfidf	good
0	275030.0	paula deen s caramel apple cheesecake	thank you paula deen! hubby just happened to ...	[60-minutes-or-less, time-to-make, course, pre...	...	[666723, 666723, 666723, 666723, 666723, 66672...	5.0	0.28	True
1	275033.0	penne with bacon spinach mushrooms	from woman's day magazine.	[bacon, 30-minutes-or-less, time-to-make, cour...	...	[166642]	5.0	0.72	True
2	275036.0	easy weeknight corn	i threw some things together in a dutch oven a...	[15-minutes-or-less, time-to-make, course, mai...	...	[590640, 590640]	5.0	0.33	True
...	...	...	...	...	...	...	...	...	...
3642	535783.0	cheesesteak stuffed onion rings	surprise your family and friends with an onion...	[60-minutes-or-less, time-to-make, main-ingred...	...	[33186, 33186]	3.0	0.36	False
3643	536688.0	coco oatmeal honey cookies	the fiber-rich cookies are good for snacking.h...	[30-minutes-or-less, time-to-make, course, pre...	...	[2002170767, 2002170767]	3.0	0.37	False
3644	536843.0	sheet pan turkey caprese meatballs with rosema...	description: try these turkey caprese meatball...	[60-minutes-or-less, time-to-make, course, mai...	...	[2001112113, 2001112113]	3.0	0.36	False

54873 rows × 10 columns

Permutation Testing

```python
observe = big_df.groupby('good')['tfidf'].mean().diff().abs().iloc[-1]

# making a distrbution where missing of description does not depend on dep_col
n_repetitions = 1000
null = []
for _ in range(n_repetitions):
    with_shuffled = big_df.assign(shuffle = np.random.permutation(big_df['good']))
    difference = with_shuffled.groupby('shuffle')['tfidf'].mean().diff().abs().iloc[-1]
    null.append(difference)
```

Processing math: 100%

```
fig = px.histogram(pd.DataFrame(null), x=0, histnorm='probability', title=f'Permutation Testing
fig.add_vline(x=observe, line_color='red', line_width=1, opacity=1)
```

Permutation Testing Using Max TF-IDF
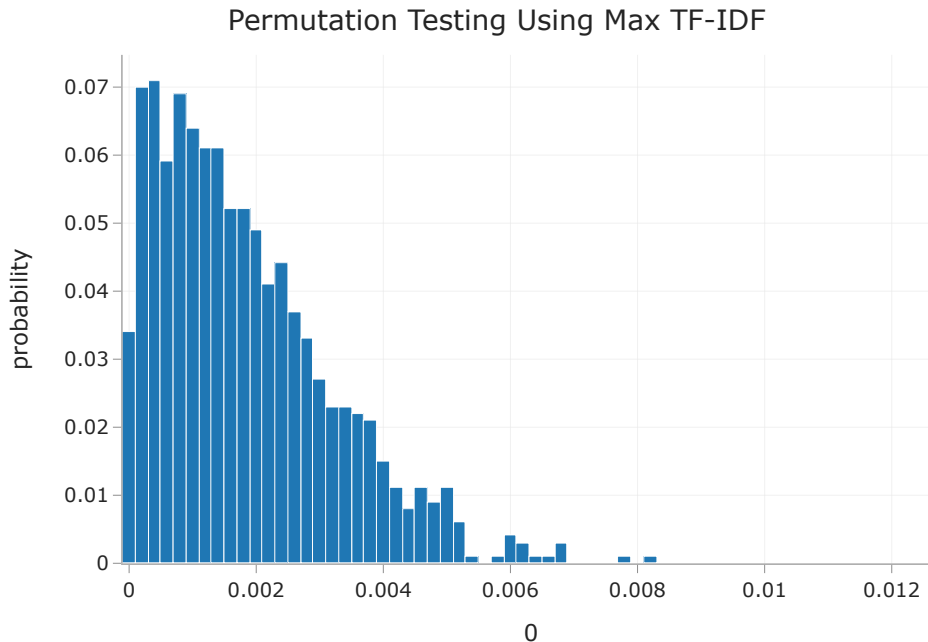
The result is significant! **We reject the null hypothesis! There is a difference in the distribution for `high_rated` recipes and `low_rated` recipes.**

Step 5: Framing a Prediction Problem

From the previous section we have learned that Recipe's `Max TF-IDF` distribution is different for `high_rated` recipe than `low_rated` recipe, so now we want to go a step further: we want to predict `rating` as a classfication problem to demonsrate user preference and as a potential prior to **reconmander system**

Specifically, **we want to predict `rating` (5 catagories) in the original data frame to demonstarte understanding of user preference.** In this section we will be using the original big DataFrame for predicting `rating`.

Step 6: Baseline Model

Just to keep everything clear, we recalled all the cleaning function here and some neccessary extraction performed

```python
# for modeling transformation
from sklearn.preprocessing import FunctionTransformer, OneHotEncoder, Binarizer, RobustScaler
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import PCA

# for modeling hyperparameter tuning
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.dummy import DummyClassifier
```

Processing math: 100%

```
# for modeling evaluation
from sklearn.metrics import ConfusionMatrixDisplay, precision_recall_fscore_support
from sklearn import metrics
```

```
interactions = pd.read_csv('food_data/RAW_interactions.csv')
recipes = pd.read_csv('food_data/RAW_recipes.csv')
step0 = recipes.merge(interactions, how='left', left_on='id', right_on='recipe_id', indicator=Tr
base_df = (step0
          .pipe(initial)
          .pipe(transform_df)
          .pipe(outlier)
          )[['n_ingredients','minutes','n_steps','description','sugar','calories','sodium','tot

base_df = base_df.assign(is_good = base_df['rating']==5)
base_df = base_df.assign(is_low = base_df['rating']<=3)
base_df
```

Out [ ]:

	n_ingredients	minutes	n_steps	description	...	recipe_date	review	is_good	is_low
0	9	40	10	these are the most; chocolatey, moist, rich, d...	...	2008-10-27	These were pretty good, but took forever to ba...	False	False
1	11	45	12	this is the recipe that we use at my school ca...	...	2011-04-11	Originally I was gonna cut the recipe in half ...	True	False
2	9	40	6	since there are already 411 recipes for brocco...	...	2008-05-30	This was one of the best broccoli casseroles t...	True	False
...	...	...	...	...	...	...	...	...	...
234426	7	20	5	i've heard of the 'cookies by design' company,...	...	2008-04-15	This recipe tastes nothing like the Cookies by...	False	True
234427	7	20	5	i've heard of the 'cookies by design' company,...	...	2008-04-15	yummy cookies, i love this recipe me and my sm...	True	False
234428	7	20	5	i've heard of the 'cookies by design' company,...	...	2008-04-15	I work at a Cookies By Design and can say this...	False	False

154763 rows × 15 columns

Handling Missing Data

It have been shwon earlier that the missingness of the `rating` columns seems to be **NMAR**, so it is not dependent on the column but rather depending on itself. Thus, we will be imputing the ratings through **random imputation**.

- This may cause more `rating` of 5 to come up

```
def prob_impute(s):
    s = s.copy()
    num_null = s.isna().sum()
    fill_values = np.random.choice(s.dropna(), num_null)
    s[s.isna()] = fill_values
```

Processing math: 100%

```
      return s

base_df['rating'] = prob_impute(base_df['rating'])
base_df = base_df.dropna()
```

Train/Validate/Test Split

We are splitting the main data set into 3 components of `train`, `validate`, and `test`. The main data set is plit to `big_train` and `test` first with big_train being 75% of the data. Then, the big_train data set is split again into the `validate` and the actual `train` data set with 75% in the train data set again. Each set is taking the percentatge as calculated below:

- Test: 25%
- Train_big: 75%
- Validate: 18.75%
- Train: 56.25%

```
In [ ]:  X = base_df.drop('rating', axis=1)
         y = base_df['rating']
         X_big_train, X_test, y_big_train, y_test = train_test_split(X, y, test_size=0.25, random_state=1
         X_train, X_val, y_train, y_val = train_test_split(X_big_train, y_big_train, test_size=0.25, rand
```

Feature Engineering & Transformations

```
In [ ]:  class StdScalerByGroup(BaseEstimator, TransformerMixin):
             '''takes in two separate, fitting data may not be transforming data (training)'''

             def __init__(self):
                 pass

             def fit(self, X, y=None):
                 '''fit using one type of data'''

                 # X might not be a pandas DataFrame (e.g. a np.array)
                 df = pd.DataFrame(X)

                 # Compute and store the means/standard-deviations for each column (e.g. 'c1' and 'c2'),
                 mean_group = df.groupby(df.columns[0]).mean()
                 std_group = df.groupby(df.columns[0]).std()

                 for col in mean_group:
                     mean_group = mean_group.rename(columns={col:f'{col}_mean'})
                     std_group = std_group.rename(columns={col:f'{col}_std'})
                 self.grps_ = pd.concat([mean_group,std_group],axis=1)
                 return self

             def transform(self, X, y=None):
                 '''may be different data'''
                 try:
                     getattr(self, "grps_")
                 except AttributeError:
                     raise RuntimeError("You must fit the transformer before tranforming the data!")

                 def standardize(x, col):
                     group = x.name
                     mean = self.grps_.loc[group, f'{col}_mean']
                     std = self.grps_.loc[group, f'{col}_std']
                     norm = (x - mean) / std
                     return norm

                 df = pd.DataFrame(X)
                 new=pd.DataFrame()
                 for col in df.columns[1:]:
```

Processing math: 100%

```
        out = df.groupby(df.columns[0])[col].transform(lambda x: standardize(x, col)) # thin
        new = pd.concat([new, out], axis=1)
    return new.assign(group=df[df.columns[0]]).set_index('group')
```

In [ ]:
```
def tag_counts(df):
    '''number of tags counted'''
    return pd.DataFrame(df['tags'].apply(lambda x: len(x)).rename('counts'))
```

Baseline Model's Pipeline

In the basic model pipeline we are working with not a great number of features:

1. binarized `n_step` with threshold 25
2. binarized `n_ingredients` with threshold 20
3. normalized `minutes` with respects to binarized `n_steps` using the customized class `StdScalerByGroup`
4. normalized `minutes` with respects to binarized `n_ingredients` using the customized class as above
5. simple counts of `tags` column

The pipeline for the model is constituted with a simple **Random Forest** multi-class classfier with hyperparameter tuned

In [ ]:
```
norm_relative = Pipeline([
    ('bi_nsteps',Binarizer(threshold=25)),
    ('norm_minutes_binary_nsteps', FunctionTransformer(lambda x: StdScalerByGroup().fit(x).trans
])

preproc_rf = ColumnTransformer(
    transformers=[
        ('bi_nsteps', Binarizer(threshold=25),['n_steps']),
        ('bi_ningredients', Binarizer(threshold=20),['n_ingredients']),
        ('norm_minutes_binary_nsteps',norm_relative,['n_steps','minutes']),
        ('norm_minutes_binary_ningredients',norm_relative,['n_ingredients','minutes']),
        ('tag_counts',FunctionTransformer(tag_counts),['tags']),
    ],
    remainder='drop'
)

pl_base = Pipeline([
    ('preprocessor', preproc_rf),
    ('rfc', RandomForestClassifier(max_depth=2,
                                   n_estimators=140,
                                   criterion='entropy',
                                   min_samples_split=2))
])
```

In [ ]:
```
pl_base.fit(X_train, y_train)
pl_base.score(X_val, y_val)
```
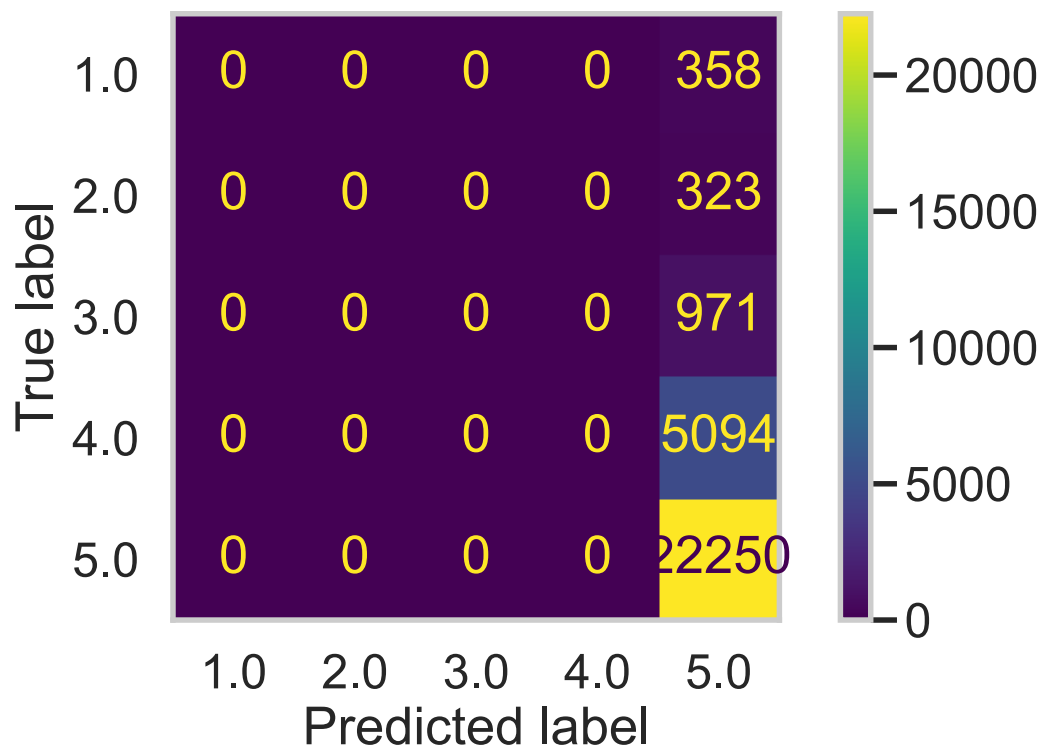
Out[ ]: 0.7673472203062491

This looks pretty good! Roughly 77% on validation set is pretty good! But let's dig deeper first

In [ ]:
```
ConfusionMatrixDisplay.from_estimator(pl_base, X_val, y_val)
plt.grid(False)
```

Processing math: 100%

```
In [ ]:  base_df['rating'].value_counts() / base_df.shape[0]
```

```
Out[ ]:  5.0    0.77
         4.0    0.18
         3.0    0.03
         1.0    0.01
         2.0    0.01
         Name: rating, dtype: float64
```

Turns out the original dataset is highly **unbalanced**, making the model always predicting a `rating` of 5 not missing many of the other details. This also means that as long as the model is always predicting the `rating` of 5, it will get an accuracy of 77% because 77% of the `rating` is 5 -> **accuracy doesn't entell everything!**. Thus, we need a better model than this that can capture some what more feature information, more engineering is needed!

Step 7: Final Model

Now with the previous baseline model's problem in mind, let's make some actual useful feature engineering, mainly we will be utilizing these features:

The previous features are carried over to this model, which includes:

1. binarized `n_step` with threshold 25, this is a result from eda
2. binarized `n_ingredients` with threshold 20, this is a result from eda
3. normalized `minutes` with respects to binarized `n_steps`
4. normalized `minutes` with respects to binarized `n_ingredients`
5. simple counts of `tags` column, showing how many tags are in each `tag` column

In addition, awe also added afew more features to capture the relationship we saw from EDA, whcih includes:

1. Some numerical columns of `sugar`, `sodium`, `calories`, `total_fat` that have being standerlized using `RobustScaler`

Two `TF-IDF` that have been `one hot encoded`:

- In particular, the naive approach is to use the highest TF-IDF for each of the words are extracted for each of the sentence using `argmax` , representing the most important words in a sentence (we are using `argmax` here is for considering the complexity of this model, later implementations can utilzie more words that have high TF-IDF)
- We then construct a pool of highest TF-IDF words in the **low** `rating` dataset, which was originally defined as `rating` lower than or equal to 3 and it is stored as a boolean indicator in the `is_low` column.
- Finally, we want to see whether or not the current sentence's highest TF-IDF word is in such pool of words
- We perform such operations with both the `name` column and also the `description` column
- **Remark**: this feature improved the final model by roughly 10% accuracy, this is the `detect_key_low(df)` function
    - We ahve also tried to trade off some complexity with better accuracy by using the count of the 5 top TF-IDF words in each row (just this function runs for about 3m)
    - However, the performance didn't perform as well as `argmax` , whihch may be due to extra noise added (48% accuracy with 5 words and 50% accuracy with one word)

3. The `recipe_dtae` column have also being taken out with only the year of the recipe and then `one hot encoded` as well.

4. At last, we also used the `tag` column of each of the sentence to perform `one hot encoding`
    - We first performed `one hot encoding` to transform each tag to a numerical boolean representation. However, this makes the feature space to reahc to about 500 features, which adds too much **sparsity** to the feature space and may introduces **noises**
    - Thus we `filtered` out all the **irrelevant** or **low counted** tags (<1000 counts) and reduces teh feature spac to only adding 80 more features
    - At last, we conducted `pca` to reduce the adding feature space to just abou 10 features and this value seems to work well with the data set experimentally.
    - The `tag_ohe_pca(df)` function takes care of this step

5. Analyzing whether the `review` columns contain certain sentiment words in it, evaluated by the `is_sentiment(df)` function

6. We have taken out irrelevant features such as the `naive_bayes` encoder taht we have implemented

```python
In [ ]:
def tag_counts(df):
    '''number of tags counted'''
    return pd.DataFrame(df['tags'].apply(lambda x: len(x)).rename('counts'))

def detect_key_low(df):
    '''transforming description's tfidf to actual most important word in a description then comp

    def key_largest(row):
        return row.index[row.argmax()] #[row.argsort()][-5:]

    def make_tfidf(series):
        lst = series.explode().astype(str).values # this may be slow
        count = TfidfVectorizer()
        count.fit(lst)
        return pd.DataFrame(count.transform(lst).toarray(), columns=count.get_feature_names_out(

    tfidf_low = make_tfidf(df[df['is_low']==True][df.columns[1]])
    tfidf_base = make_tfidf(df[df.columns[1]])

    keyword_all = tfidf_base.apply(key_largest, axis=1) #argmax a bit faster
    keyword_low = tfidf_low.apply(key_largest, axis=1)
    pool_low = keyword_low.unique() #.explode().unique()

    in_low = keyword_all.apply(lambda x: x in pool_low) #.apply(lambda x: sum([word in pool_low

    return pd.DataFrame(in_low)

def tag_ohe_pca(df):
```

```python
    '''OHE all the tag result after it have being pca dimension reduced to 50'''
    # getting all the unique one quick
    set = [j for i in df['tags'].tolist() for j in i] # explode in a time complexity efficient w
    count = CountVectorizer()
    count.fit(set).transform(set)

    my_dict = np.array(list(count.vocabulary_.keys()))

    def helper_function(list,dict):
        return np.array([i in list for i in dict])

    a = df["tags"].apply(lambda x:helper_function(x, my_dict))

    # change array of array into 2D array
    df_pca = pd.DataFrame(data = np.stack(a.to_numpy()),columns=my_dict)

    flipped = df_pca.T
    filter_df = flipped[flipped.sum(axis=1)>1000].T # keep only useful tags

    # conduct PCA to reduce to just 50 dimensions
    pca = PCA(n_components=10)
    reduced = pca.fit_transform(filter_df)

    return reduced

def is_sentiment(df):
    '''For detecting sentiment words in the review column'''

    word_list = ['awful', 'fav', 'well',
            'yet', 'fantastic',
            'pretty good','dislike','hate', 'bad',
            'delicious', 'wonderful',
            'great', 'but', 'good', 'next',
            'excellent', 'nice', 'bland', 'maybe',
            'loved', 'sorry', 'think', 'however', 'would',
            'perfect', 'very', 'keeper', 'liked', 'made']

    out = df['review'].apply(lambda x: word in x for word in word_list)#.sum(axis=1)

    return pd.DataFrame(out.astype(int))
```

Final Model's Pipeline

Since this is a **multi-class classifictaion** problem and the data is also highly **unbalanced**, we are also adding a **dummy** classifier that classifies uniformly at random to bench mark our modle's performances. Of course, we will also use different evaluation metrics later to demonstarte the model's performances as well, the dummy classfier is just an "easy to view" example.

```
In [ ]:  # dummy classfier with uniformally selections
         dummy_clf = DummyClassifier(strategy="uniform")
         dummy = dummy_clf.fit(X_train, y_train)
```

For the pipeline, we are still doing an **Homogenous Ensemble Learning** with decision tree as it have being shown to perform the best experimentally (we have tried hetergenous ensemble learning using voting/stacking with models such as SVM and logistic regression, but none of the perform as well as the simple random forest).

We balanced the dataset by using automatic balaning argumnet `"balanced"`, we have also tried to use self customized dictionaries for assigning weights, However, this wouldn't be generalizable to unseen data as the distribution of data changes. The `sk_learn` packages does automatic weight assigning by the following formula:

$$n_samples/(n_classes * np.bincount(y))$$

This model pipeline takes about 30 seconds to fit

```python
In [ ]: norm_relative = Pipeline([
            ('bi_nsteps',Binarizer(threshold=25)),
            ('norm_minutes_binary_nsteps', FunctionTransformer(lambda x: StdScalerByGroup().fit(x).trans
        ])

        key_ohe = Pipeline([
            ('tfidf',FunctionTransformer(detect_key_low)),
            ('key_ohe', OneHotEncoder(drop='first'))
        ])

        date_ohe = Pipeline([
            ('date_get', FunctionTransformer(lambda x: pd.DataFrame(x['recipe_date'].dt.year))),
            ('date_ohe', OneHotEncoder())
        ])

        preproc_rf = ColumnTransformer(
            transformers=[
                ('tfidf_key_ohe_description', key_ohe, ['is_low','description']),
                ('tfidf_key_ohe_name', key_ohe, ['is_low','name']),
                ('bi_nsteps', Binarizer(threshold=25),['n_steps']),
                ('bi_ningredients', Binarizer(threshold=20),['n_ingredients']),
                ('norm_minutes_binary_nsteps',norm_relative,['n_steps','minutes']),
                ('norm_minutes_binary_ningredients',norm_relative,['n_ingredients','minutes']),
                ('tag_counts',FunctionTransformer(tag_counts),['tags']), # good for predict 5
                ('robust_nutrition',RobustScaler(),['sugar','sodium','calories','total_fat']),
                ('date_ohe', date_ohe,['recipe_date']), # can't be too sparse, 10 max
                ('tag_pca',FunctionTransformer(tag_ohe_pca),['tags']),
                ('is_sentiment',FunctionTransformer(is_sentiment),['review'])
            ],
            remainder='drop'
        )

        class_weight= 'balanced' # balanced weighting automatically calculates th  best weighting!

        pl_rf = Pipeline([
            ('preprocessor', preproc_rf),
            ('rfc', RandomForestClassifier(max_depth=18,
                                           n_estimators=130,
                                           criterion='entropy',
                                           min_samples_split=2,
                                           class_weight=class_weight
                                           ))
        ])

        pl_rf.fit(X_train, y_train)
```

```
Out[ ]:  Pipeline(steps=[('preprocessor',
                        ColumnTransformer(transformers=[('tfidf_key_ohe_description',
                                                         Pipeline(steps=[('tfidf',
                                                                          FunctionTransformer(func=<fu
nction detect_key_low at 0x2cf1ced30>)),
                                                                         ('key_ohe',
                                                                          OneHotEncoder(drop='firs
t'))])),
                                                        ['is_low', 'description']),
                                                       ('tfidf_key_ohe_name',
                                                        Pipeline(steps=[('tfidf',
                                                                         FunctionTransformer(func=<fu
nction detect_key_low at 0x...
                                                                         FunctionTransformer(func=<fu
nction <lambda> at 0x2cd4d1040>)),
                                                                        ('date_ohe',
                                                                         OneHotEncoder())]),
                                                       ['recipe_date']),
                                                      ('tag_pca',
                                                       FunctionTransformer(func=<function tag_ohe_pc
a at 0x2cf645040>),
                                                       ['tags']),
                                                      ('is_sentiment',
                                                       FunctionTransformer(func=<function is_sentime
nt at 0x2cf645790>),
                                                       ['review'])])),
                       ('rfc',
                        RandomForestClassifier(class_weight='balanced',
                                               criterion='entropy', max_depth=18,
                                               n_estimators=130))])
```

Hyperparameter Tunning

We have performed Grid Search and Random Search for the best parameters for teh Random Forest Classifier

```python
In [ ]:  %time
         hyperparameters = {
         'rfc__max_depth': np.arange(2, 20, 2),
         'rfc__n_estimators': np.arange(100, 150, 10),
         }
         grids = GridSearchCV(pl_rf,
                             n_jobs=-1,
                             param_grid=hyperparameters,
                             return_train_score=False,
                             cv=5
                             )

         grids.fit(X_train, y_train)
         grids.best_params_
```

```
CPU times: user 1e+03 ns, sys: 0 ns, total: 1e+03 ns
Wall time: 3.1 µs
```
```
/Users/kevinb/miniforge3/envs/dsc80/lib/python3.8/site-packages/joblib/externals/loky/process_exe
cutor.py:700: UserWarning:

A worker stopped while some jobs were given to the executor. This can be caused by a too short wo
rker timeout or by a memory leak.
```

```
Out[ ]:  {'rfc__max_depth': 18, 'rfc__n_estimators': 130}
```

Model Evaluation

To really understand what we are evaluating, we need to first understand what metrics matters to us:

Processing math: 100%

Example:

- **Precision for 5**: Out of all 5 we predicted, how many are actually 5
- **Recall for 5**: Out of all actual 5, how many did we get right

We care about getting a correct rating for reconmandation, we care about finding **Recall** but still considering precision, accuracy, F1 scores

Feature Importantness Analysis

```
In [ ]: feature = pd.DataFrame(pl_rf.named_steps['rfc'].feature_importances_)
        fig = px.bar(feature, title='Feature Importantness for Random Forest', labels={'value':'feature
        fig.show()
```
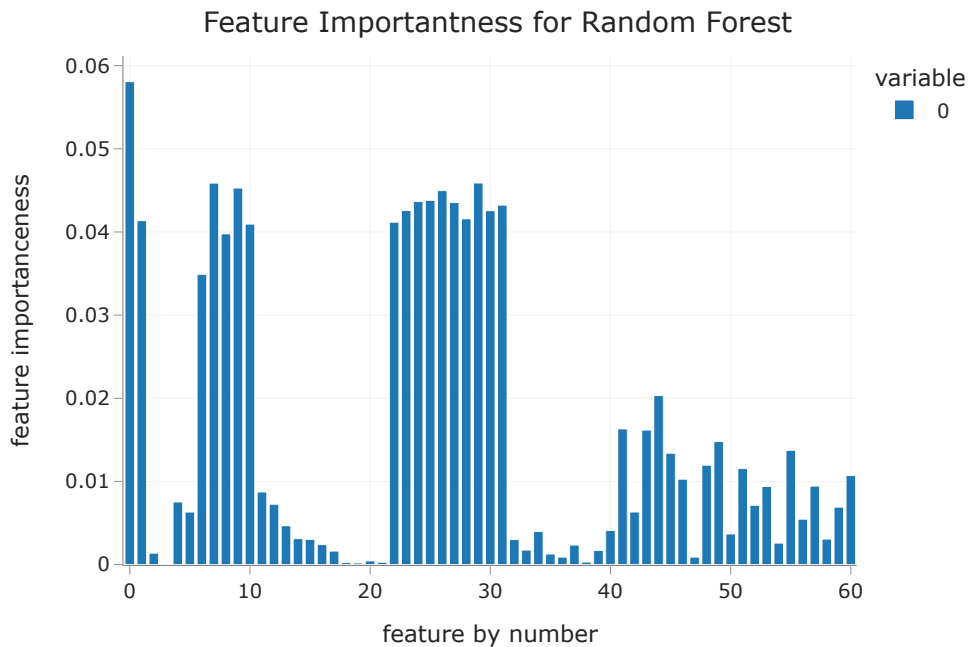

We have 60 features in our model with feature 0 and feature 1 having the most effect! these are the 2 argmax `TF-IDF` encoder that we have implemented, this is consistent with ou previous **permutation testing** that shows the `TF-IDF` distribution for `high_rated` and `low_rated` recipes are different distributions

```
In [ ]: # for evaluating
        def metrics_all(model, X, y):
            return pd.DataFrame(precision_recall_fscore_support(model.predict(X), y),
                        index=['precision','recall','f1_score','count'],
                        columns=[1,2,3,4,5]).T
```
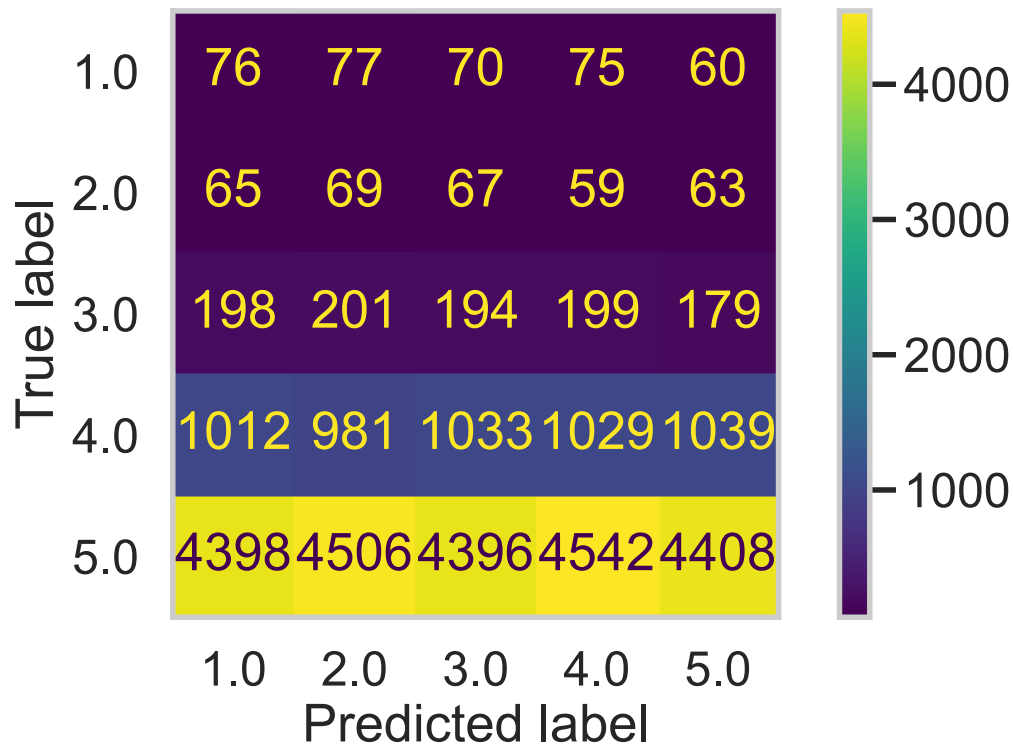
Confusion matrix for random forest classifier

```
In [ ]: ConfusionMatrixDisplay.from_estimator(pl_rf, X_val, y_val)
        plt.grid(False)
```
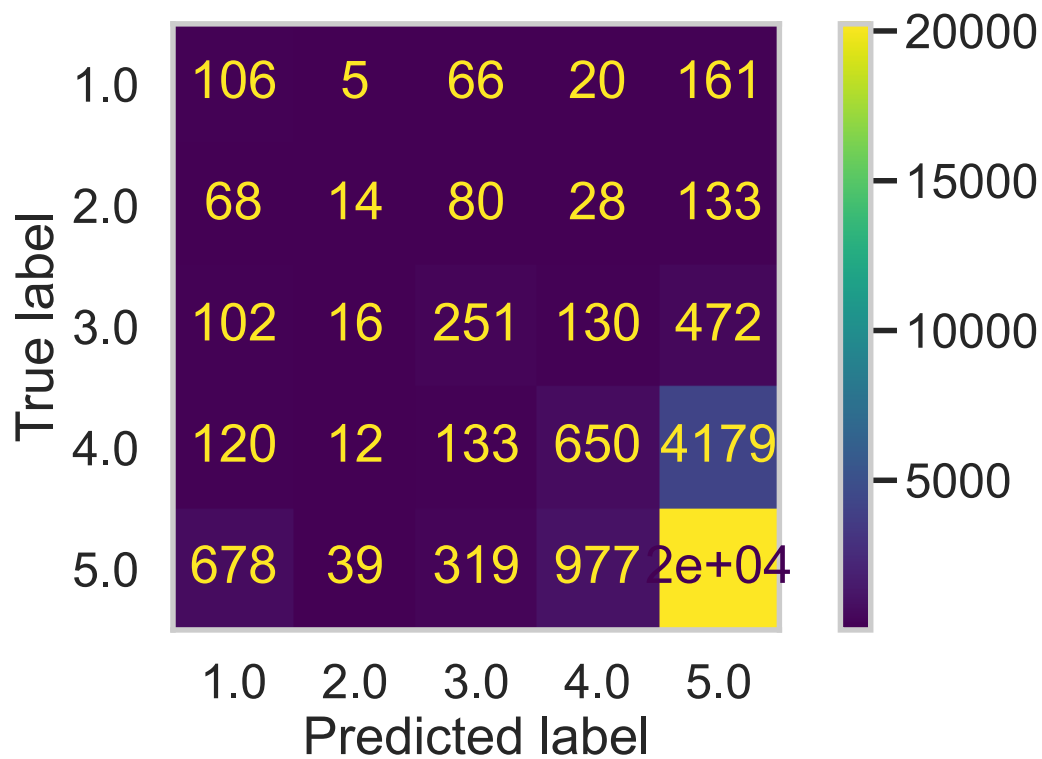
Confusion matrix for dummy classifier

```
In [ ]: ConfusionMatrixDisplay.from_estimator(dummy, X_val, y_val)
        plt.grid(False)
```


Accuracy for random forest classifier

```
In [ ]: pl_rf.score(X_val, y_val)
```

Processing math: 100%

```
Out[ ]:  0.7337908677058904
```

Accuracy for dummy classifier

```
In [ ]:  dummy.score(X_val, y_val)
```

```
Out[ ]:  0.19661332597599668
```

Full metrics for random forest classfier

```
In [ ]:  metrics_all(pl_rf, X_val, y_val)
```

Out[ ]:

	precision	recall	f1_score	count
1	0.30	0.10	0.15	1071.0
2	0.04	0.16	0.07	86.0
3	0.26	0.30	0.28	859.0
4	0.13	0.36	0.19	1800.0
5	0.91	0.80	0.85	25180.0

Full metrics for dummy classifier

```
In [ ]:  metrics_all(dummy, X_val, y_val)
```

Out[ ]:

	precision	recall	f1_score	count
1	0.18	1.11e-02	0.02	5745.0
2	0.17	9.55e-03	0.02	5758.0
3	0.21	3.59e-02	0.06	5733.0
4	0.21	1.77e-01	0.19	5955.0
5	0.20	7.65e-01	0.32	5805.0

Testing on Actual Test Data Set

```
In [ ]:  ConfusionMatrixDisplay.from_estimator(pl_rf, X_test, y_test)
         plt.grid(False)
```

Processing math: 100%
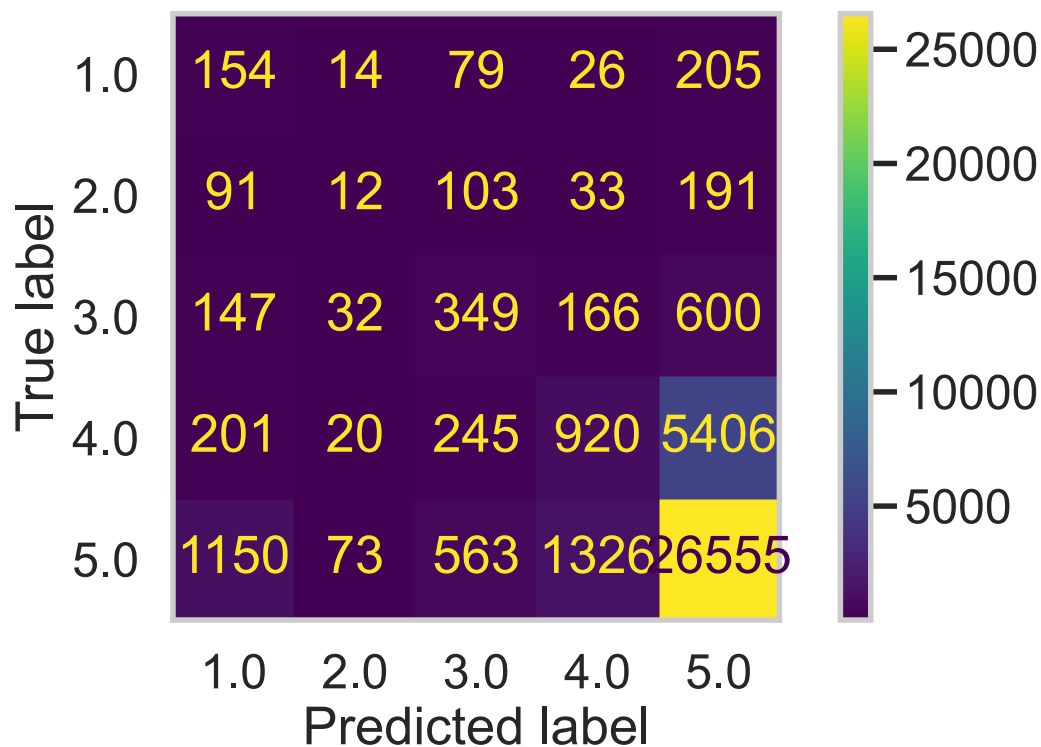
```
In [ ]: pl_rf.score(X_test, y_test)
```

```
Out[ ]: 0.7241923385323711
```

```
In [ ]: metrics_all(pl_rf, X_test, y_test)
```

Out[ ]:

	precision	recall	f1_score	count
1	0.32	0.09	0.14	1742.0
2	0.03	0.08	0.04	154.0
3	0.27	0.27	0.27	1328.0
4	0.13	0.38	0.20	2429.0
5	0.90	0.81	0.85	33008.0

Step 8: Fairness Analysis

We want to evaluate whether the model is fair for treating all populations. In particular, we want to check in the scope of looking at the predictions for the `vegan` group and the `vegetarian` group. Let's first check how many of them are in the data set.

```
In [ ]: X_test['tags'].apply(lambda x: 'vegetarian' in x).sum()
```

```
Out[ ]: 5696
```

```
In [ ]: X_test['tags'].apply(lambda x: 'vegan' in x).sum()
```

```
Out[ ]: 1421
```

```
In [ ]: out = X_test.assign(prediction = pl_rf.predict(X_test))
```

```
in_tag = out['tags'].apply(lambda x: ('vegetarian' in x)|('vegan' in x))
```

Processing math: 100%

```
out = out.assign(is_in = is_in_tag)
```

Let's check the grouped by mean first

In [ ]:
```
out.groupby('is_in')['prediction'].mean()
```

Out[ ]:
```
is_in
False    4.21
True     4.22
Name: prediction, dtype: float64
```

Difference Significant?

We run a **permutation test** to see if the difference in accuracy is significant.

- **Null Hypothesis**: The classifier's accuracy is the same for both `vegan` + `vegetarian` tags and non `vegan` + `vegetarian` tags, and any differences are due to chance.
- **Alternative Hypothesis**: The classifier's accuracy is higher for non `vegan` + `vegetarian` tags.
- Test statistic: Difference in accuracy (is_in minus not_in).
- Significance level: 0.05

In [ ]:
```
compute_accuracy = lambda x: metrics.accuracy_score(x['is_in'], x['prediction'])
obs = out.groupby('is_in').apply(compute_accuracy).diff().iloc[-1]

diff_in_acc = []
for _ in range(1000):
    s = (
        out[['is_in', 'prediction']]
        .assign(shuffle=np.random.permutation(out['is_in']))
        .groupby('shuffle')
        .apply(compute_accuracy)
        .diff()
        .iloc[-1]
    )
    diff_in_acc.append(s)
```

In [ ]:
```
fig = pd.Series(diff_in_acc).plot(kind='hist', histnorm='probability',
                                  title='Difference in Accuracy (Is_in - Not_in)')

fig.add_vline(x=obs, line_color='red')
fig.show()
```

Difference in Accuracy (Is_in - Not_in)

variable
■ 0
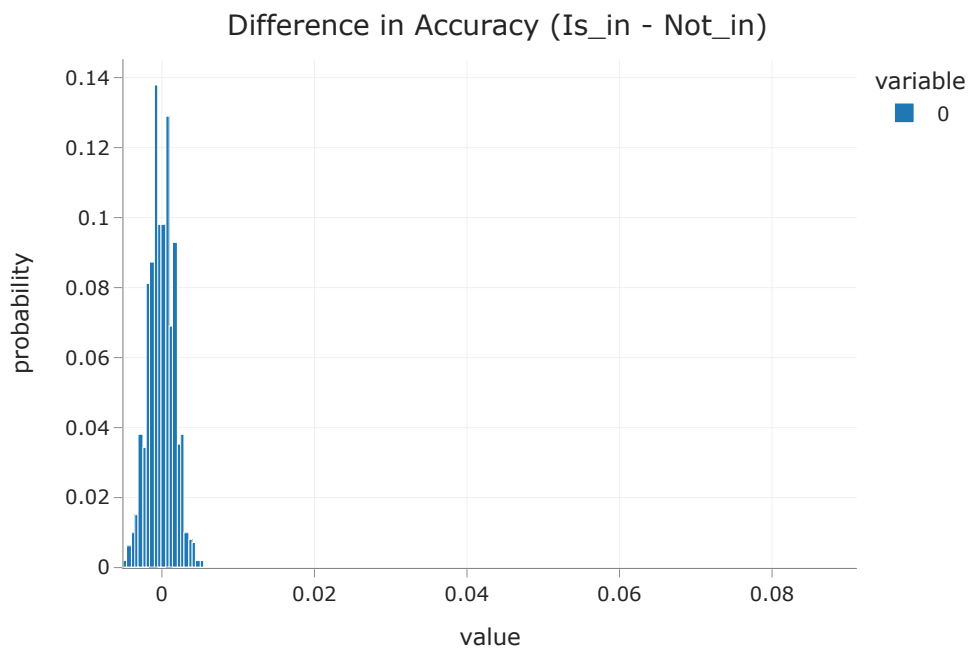
```
In [ ]:  (obs <= diff_in_acc).mean()
```

```
Out[ ]:  0.0
```

The result is **significant**, we reject the null hypothesis!

Processing math: 100%