Exploratory Data Analysis

Sagi and Dean

2023-08-15

Intro

Before we start building our model, let's take a look at the data we have. We'll start by loading the data into a pandas data frame. After that, we'll take a look at the data, and prepare it for the feature engineering / feature extraction step. Eventually, we'll split the data for train, validation and test sets, and look for interesting patterns in the train data that our model can learn from.

```
import pandas as pd

nutrients = pd.read_csv("data/nutrients.csv")
food_nutrients = pd.read_csv("data/food_nutrients.csv")
food_train = pd.read_csv("data/food_train.csv")
```

Food Nutrients Dataset

```
food_nutrients
```

```
##           idx  nutrient_id  amount
## 0           1         1087  143.00
## 1           1         1089    5.14
## 2           1         1104    0.00
## ...       ...          ...     ...
## 493051  35276         1253    0.00
## 493052  35276         1257    0.00
## 493053  35276         1258    3.57
##
## [493054 rows x 3 columns]
```

The amount column isn't meaningful without the serving units. This gives us a hint that we need to merge information from the two datasets.

There're no missing values within the nutrients dataset:

```
food_nutrients.isna().any().any()
```

```
## False
```

There are only 48 unique nutrients in the snacks dataset:

```
food_nutrients["nutrient_id"].nunique()
```

```
## 48
```

It would be convenient to add a column for each nutrient and fill it with the amount of the nutrient in the food. Default value will be 0, indicating the food doesn't contain the nutrient. This way, we can easily join the food_nutrients with the snacks datasets.

```
food_nutrients_wide = food_nutrients.pivot_table(
    index=["idx"], columns=["nutrient_id"], values=["amount"], fill_value=0
).droplevel(0, axis=1)
food_nutrients_wide  # single row for each snack
```

```
## nutrient_id  1003   1004   1005  1008  1009  ...  1257   1258  1292  1293   2000
## idx                                          ...                                
## 1            7.14  35.71  53.57   536   0.0  ...   0.0  25.00   0.0   0.0  42.86
## 2            2.63  15.79  68.42   421   0.0  ...   0.0   6.58   0.0   0.0  42.11
## 3            3.33  15.00  70.00   433   0.0  ...   0.0   6.67   0.0   0.0  43.33
## ...           ...    ...    ...   ...   ...  ...   ...    ...   ...   ...    ...
## 35274        7.14  32.14  53.57   536   0.0  ...   0.0   3.57   0.0   0.0   7.14
## 35275        7.14  32.14  57.14   536   0.0  ...   0.0   3.57   0.0   0.0   7.14
## 35276        7.14  35.71  53.57   536   0.0  ...   0.0   3.57   0.0   0.0   3.57
## 
## [35276 rows x 48 columns]
```

Some nutrients are very rare:

```
(food_nutrients_wide > 0).mean().sort_values().head(10)
```

```
## nutrient_id
## 1018    0.000000
## 1009    0.000028
## 1062    0.000028
##           ...   
## 1056    0.000397
## 1186    0.000595
## 1072    0.000652
## Length: 10, dtype: float64
```

Still, we'll keep them in the dataset for now, and see if they're useful for our model. It might be due to the fact that some nutrients are only found in a specific food category.

We aim to use the nutrients dataset to predict the food category, as there aren't too many nutrient variables. Therefore, we need to merge the nutrients dataset with the snacks dataset. We'll use the food id to merge the two datasets.

After taking a glance in the nutrients distributions with each other among snacks, we saw that some nutrients are frequently positive, while others are not. In addition **some** of them are correlated. We'll find out later how useful they are, and from what threshold should we eliminate nutrients.

Nutrients Dataset

```
nutrients
```

```
##      nutrient_id                               name unit_name
## 0           1002                           Nitrogen         G
## 1           1003                            Protein         G
## 2           1004                  Total lipid (fat)         G
## ..           ...                                ...       ...
## 232         2029                     trans-Lycopene        UG
## 233         2032                Cryptoxanthin, alpha        UG
## 234         2033  Total dietary fiber (AOAC 2011.25)         G
##
## [235 rows x 3 columns]
```

No missing values in nutrients:

```
nutrients.isna().any().any()
```

```
## False
```

A single duplicated name:

```
nutrients[nutrients["name"].duplicated(keep=False)].sort_values(by="name")
```

```
##     nutrient_id    name unit_name
## 5          1008  Energy      KCAL
## 23         1062  Energy        kJ
```

It might be useful to later unify there 2 nutrients into 1, as they are the same. We'll do that by scaling by the appropriate factor (KCAL -> KJ).

Before applying any machine learning algorithms, we usually need to preprocess the data. This includes feature scaling as well. Thus, `unit_name` variable is redundant, as it's just a string representation for a scaling factor. We'll add the unit name as a suffix to the column name, and then drop the `unit_name` column.

```
nutrients_v2
```

```
##                                                         name
## nutrient_id
## 1002                                           nitrogen__(g)
## 1003                                            protein__(g)
## 1004                                  total_lipid_(fat)__(g)
## ...                                                      ...
## 2029                                    trans-lycopene__(ug)
## 2032                               cryptoxanthin_alpha__(ug)
## 2033         total_dietary_fiber_(aoac_2011.25)__(g)
##
## [235 rows x 1 columns]
```

Now let's merge the information from both two datasets, keeping in mind we need to scale the `KCAL` nutrient by 4.184 to get `KJ`.

```
food_nutrients_merged = food_nutrients_wide.rename(mapper=nutrients_v2["name"], axis=1)

food_nutrients_merged.loc[:, "energy__(kj)"] += (
    4.184 * food_nutrients_merged.loc[:, "energy__(kcal)"]
)  # convert kcal to kJ
food_nutrients_merged.drop(
    columns=["energy__(kcal)"], inplace=True
)  # duplicated nutrient

food_nutrients_merged.head()
```

```
## nutrient_id  protein__(g)  ...  sugars_total_including_nlea__(g)
## idx                        ...
## 1                    7.14  ...                             42.86
## 2                    2.63  ...                             42.11
## 3                    3.33  ...                             43.33
## 4                    5.00  ...                             47.50
## 5                    7.50  ...                             40.00
##
## [5 rows x 47 columns]
```

It might be plausible to later drop infrequent nutrients, as they might not be useful for the classification task. Let's check the most infrequent nutrients:

```
(food_nutrients_merged > 0).sum().sort_values().to_frame().rename(
    columns={0: "frequency"}
).head()
```

```
##                       frequency
## nutrient_id
## alcohol_ethyl__(g)            0
## starch__(g)                   1
## molybdenum_mo__(ug)           2
## biotin__(ug)                  2
## xylitol__(g)                  2
```

All the information we need is now in a single dataframe. Recall that each nutrient column associates with a single unit name. We tried to find a correlation between the category and the sum of nutrients values, grouped by the unit name, but decided to drop it. We can now start exploring the training data, and see if we can find any interesting patterns.

```
food_nutrients_merged.to_csv("data/food_nutrients_merged.csv")
```

Food Training Dataset

Hold-Out

We'll split to train, validation and test sets before we start to explore the data, so we won't have to worry about data leakage. We'll use 15% of the data for validation and 5% for test.

```
from sklearn.model_selection import train_test_split


features_df = food_train.drop("category", axis=1)
labels_df = food_train["category"]

X_train, X_val_test, y_train, y_val_test = train_test_split(
    features_df, labels_df, test_size=0.2, random_state=42
)

X_val, X_test, y_val, y_test = train_test_split(
    X_val_test, y_val_test, test_size=0.25, random_state=42
)

X_train["y"] = y_train
```

Missing Values

```
X_train.isna().sum().sort_values(ascending=False)
```

```
## ingredients                    30
## household_serving_fulltext     10
## idx                             0
##                                ..
## serving_size                    0
## serving_size_unit               0
## y                               0
## Length: 8, dtype: int64
```

Let's check the columns with missing values. The household_serving_fulltext doesn't have many missing values, and I couldn't find anything interesting about them, so I decided to omit them from this section.

```
X_train[X_train["ingredients"].isna()]["y"].value_counts(
    normalize=True
).to_frame().rename(columns={"y": "rate"})
```

```
##             proportion
## y
## popcorn       0.600000
## cakes         0.166667
## candy         0.100000
## cookies       0.066667
## chips         0.033333
## chocolate     0.033333
```

Seems like there's a majority of `ingredients` missing values for the `popcorn_peanuts_seeds_related_snacks` category. That being said, there's less than 1% of the data missing, and I couldn't find strong enough evidence for interesting patterns about the snacks with missing ingredients. Thus, we'll replace the missing values with the string "na", and leave it as is.

```
from helpers.preprocess import FillNA

FillNA().fit_transform(X=X_train)
```

Now, we'll join the snacks dataset with the `food_nutrients_merged` dataset. We'll use the `food_id` column to join the two datasets.

```
from helpers.preprocess import MergeWithFoodNutrients

X_train = MergeWithFoodNutrients().fit_transform(X=X_train)
X_train
```

```
##          idx  ... sugars_total_including_nlea__(g)
## 23212  25784  ...                            39.29
## 22158  24607  ...                            36.67
## 1703    1898  ...                            14.81
## ...      ...  ...                              ...
## 860      960  ...                             0.00
## 15795  17524  ...                            10.71
## 23654  26279  ...                            30.77
##
## [25400 rows x 55 columns]
```

We have a dataset of 55 features, combining information from all 3 tabular datasets.

Let's analyze the data a bit more.

Ingredients

The `ingredients` is an interesting column, as it may be considered as a nested column. Some of the ingredients contains list of ingredients, and some contains a single ingredient. We'll need to preprocess this column before we can use it for training.

```
X_train["ingredients"].head()
```

```
## 23212    sugar, bleached wheat flour, soybean oil, wate...
## 22158    filling (sugar, vegetable shortening (may cont...
## 1703     enriched wheat flour (flour, niacin, reduced i...
## 20886    sugar; wheat flour; nonfat milk; cocoa butter;...
## 18703    sugar, enriched bleached flour (wheat flour, n...
## Name: ingredients, dtype: object
```

Note that the data is noisy and contains typos in a small percentage of the data:

```
from collections import Counter

ingredients_example = X_train.loc[18201, "ingredients"]
chars_counter = Counter(ingredients_example)

chars_counter["("] == chars_counter[")"]
```

```
## False
```

First, we'll omit text between () and []:

```
from helpers.preprocess import CleanAndListifyIngredients

CleanAndListifyIngredients(keep_top_n=3).fit_transform(X_train)["ingredients"].head()
```

```
## 23212               sugar bleached_wheat_flour soybean_oil
## 22158                    filling wheat_flour baking_powder
## 1703                      enriched_wheat_flour water sugar
## 20886    sugar_wheat_flour_nonfat_milk_cocoa_butter_cho...
## 18703    sugar enriched_bleached_flour vegitable_shorening
## Name: ingredients, dtype: object
```

20 most correlated ingredients with one of the categories (target variable), sorted by their frequency in the dataset:

```
from helpers.utils import highest_accuracy_category


ingredients = X_train["ingredients"].str.split(" ").explode().str.strip()
ingredients_frequencies = ingredients.value_counts()


important_ingredients = highest_accuracy_category(
    df=X_train,
    frequent_tokens=ingredients_frequencies,
    colname="ingredients",
    min_token_frequeny=100,
).head(20)

important_ingredients
```

```
##           ingredients  count      rate   category
## 10           potatoes    960  0.994832      chips
## 17            gelatin    602  0.992722      candy
## 77  sunflower_kernels    105  0.992593    popcorn
## ..                ...    ...       ...        ...
## 28            raisins    348  0.868914    popcorn
## 14       cocoa_butter    821  0.841991  chocolate
## 29               eggs    310  0.839744      cakes
##
## [20 rows x 4 columns]
```

Some ingredients are very correlated with one of the categories. For example, the `ingredients` column contains `potatoes` in 99.4% of the `chips_pretzels_snacks` category. Recall there are many ingredients, and we tested each one of them with each category. Thus, we'll need to be careful not to overfit the model to the ingredients column.

household_serving_fulltext

The numeric value of the serving size is usually the first word in the `household_serving_fulltext` column. As we can see, there are many different ways to write the serving size, and the correlation for each one of them with the category is not significant, considering the number of different serving sizes we have.

```
serving_frequencies = (
    X_train["household_serving_fulltext"]
    .str.split(" ")
    .apply(lambda x: x[0])
    .value_counts()
)

highest_accuracy_category(
    df=X_train,
    frequent_tokens=serving_frequencies,
    colname="household_serving_fulltext",
    min_token_frequeny=30,
    verbose=True,
).head(10)
```

```
##    household_serving_fulltext  count      rate category
## 9                         1/4    335  0.941860  popcorn
## 3                        0.25   1804  0.892086  popcorn
## 34                        0.2     78  0.854796  popcorn
## ..                        ...    ...       ...      ...
## 19                      0.125    176  0.772727    cakes
## 37                         21     56  0.746032    candy
## 43                         23     49  0.689655    candy
##
## [10 rows x 4 columns]
```

The second word of the `household_serving_fulltext` column usually contains the serving unit. Some of them are very correlated with the category, but again, there are many different serving units, and some literally describe the category (e.g. `cake`).

```
X_train[X_train["household_serving_fulltext"].str.contains("wafer", regex=False)][
    "y"
].value_counts(normalize=True)
```

```
## y
## cookies      0.983607
## candy        0.008197
## chocolate    0.008197
## Name: proportion, dtype: float64
```

```
serving_frequencies = (
    X_train["household_serving_fulltext"]
    .str.split(" ")
    .apply(lambda x: x[1] if len(x) > 1 else "na")
    .value_counts()
)

highest_accuracy_category(
    df=X_train,
    frequent_tokens=serving_frequencies,
    colname="household_serving_fulltext",
    min_token_frequeny=100,
    verbose=True,
).head(20)
```

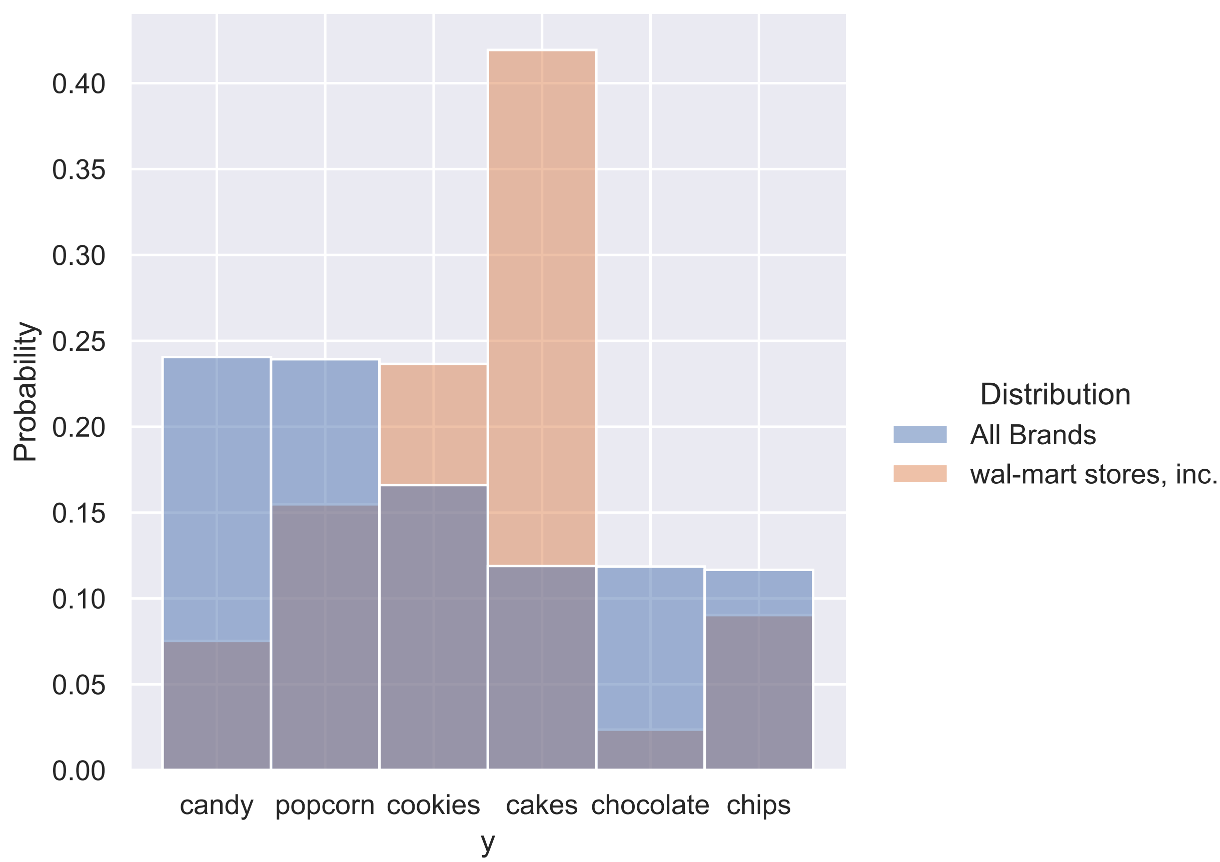
```
##    household_serving_fulltext  count      rate category
## 20                   cupcakes    133  1.000000    cakes
## 23                     wafers    105  0.990909  cookies
## 11                       cake    457  0.988121    cakes
## ..                        ...    ...       ...      ...
## 6                       piece    663  0.606217    candy
## 15                        pie    225  0.577733    candy
## 17                        bag    199  0.411017    chips
##
## [20 rows x 4 columns]
```

Brand

The brand column contains useful information. The probability for each catgory dramatically changes depending on the brand:

```
import seaborn as sns

most_frequesnt_brand = X_train["brand"].mode()[0]

categories_all_brands = pd.DataFrame(
    {
        "y": X_train["y"],
```

```
        "Distribution": "All Brands",
    }
)

categories_most_freq_brand = pd.DataFrame(
    {
        "y": X_train[X_train["brand"] == most_frequesnt_brand]["y"],
        "Distribution": most_frequesnt_brand,
    }
)

df_combined = pd.concat([categories_all_brands, categories_most_freq_brand])
df_combined["y"] = pd.Categorical(
    df_combined["y"], list(categories_all_brands["y"].value_counts().index)
)

sns.set_theme()
sns.displot(
    df_combined, x="y", hue="Distribution", stat="probability", common_norm=False
)
```



We'll get the posterior distributions for each category using Naive Bayes, while treating brands as tokens.

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
```

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report

sentences = X_train["brand"].str.replace(" ", "")
val_sentences = X_val["brand"].str.replace(" ", "")

# Step 1: Create the bag-of-words representation
vectorizer = CountVectorizer(ngram_range=(1, 1))
train_matrix = vectorizer.fit_transform(sentences)
val_matrix = vectorizer.transform(val_sentences)

# Step 2: Train the Naive Bayes classifier
nb_classifier = MultinomialNB()
nb_classifier.fit(train_matrix, y_train)
```

```
## MultinomialNB()
```

```
# Step 3: Predict using the classifier
y_pred_nb = nb_classifier.predict(val_matrix)

# Step 4: Evaluation
print("Accuracy:", accuracy_score(y_val, y_pred_nb))
```

```
## Accuracy: 0.6233466302750368
```

```
print("Classification Report:")
```

```
## Classification Report:
```

```
print(classification_report(y_val, y_pred_nb))
```

```
##               precision    recall  f1-score   support
##
##        cakes       0.72      0.62      0.66       561
##        candy       0.56      0.74      0.64      1128
##        chips       0.81      0.48      0.60       533
##    chocolate       0.74      0.51      0.61       556
##      cookies       0.80      0.49      0.61       798
##      popcorn       0.54      0.72      0.62      1187
##
##     accuracy                           0.62      4763
##    macro avg       0.69      0.59      0.62      4763
## weighted avg       0.66      0.62      0.62      4763
```

62% Accuracy only by using the `brand`! Such a simple algorithm attained nice accuracy for predicting the category, conditioned only by the snack `brand`.

Note that the maximum snacks per brand for a certain category is not very high, hence we have many unique brands. We'll have to consider than in case we desire to vectorize the `brand` column.

Let's try do the same with `description`.

Description

The `description` is a very interesting column, as it contains a lot of unstructured information about the food. We'll expect to achieve much better accuracy with the Naive Bayes approach. We use CountVectorizer to vectorize the column to the words count, while utilizing some nice features such as eliminating stop-words, considering n-grams as single tokens, strip accents of non alphanumeric characters and more.

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, classification_report

sentences = X_train["description"]
val_sentences = X_val["description"]

# Step 1: Create the bag-of-words representation
vectorizer = CountVectorizer(
    stop_words="english", ngram_range=(1, 6), strip_accents="unicode"
)
train_matrix = vectorizer.fit_transform(sentences)
val_matrix = vectorizer.transform(val_sentences)

# Step 2: Train the Naive Bayes classifier
nb_classifier = MultinomialNB()
nb_classifier.fit(train_matrix, y_train)
```

```
## MultinomialNB()
```

```
# Step 3: Predict using the classifier
y_pred_nb = nb_classifier.predict(val_matrix)

# Step 4: Evaluation
print("Accuracy:", accuracy_score(y_val, y_pred_nb))
```

```
## Accuracy: 0.9030023094688222
```

```
print("Classification Report:")
```

```
## Classification Report:
```

```
print(classification_report(y_val, y_pred_nb))
```

```
##               precision    recall  f1-score   support
##
##        cakes       0.97      0.93      0.95       561
##        candy       0.90      0.89      0.90      1128
##        chips       0.96      0.94      0.95       533
##    chocolate       0.73      0.80      0.77       556
##      cookies       0.92      0.92      0.92       798
##      popcorn       0.93      0.92      0.92      1187
##
##     accuracy                           0.90      4763
##    macro avg       0.90      0.90      0.90      4763
## weighted avg       0.91      0.90      0.90      4763
```

90% accuracy is really good, considering the simplicity of the model. The precision and recall of the `chocolate` category is much lower the the others. In addition, f1 score of `candy` is also disturbing, since most of our snacks in the dataset are candies.

For each category, we'll check the most important words in the description:

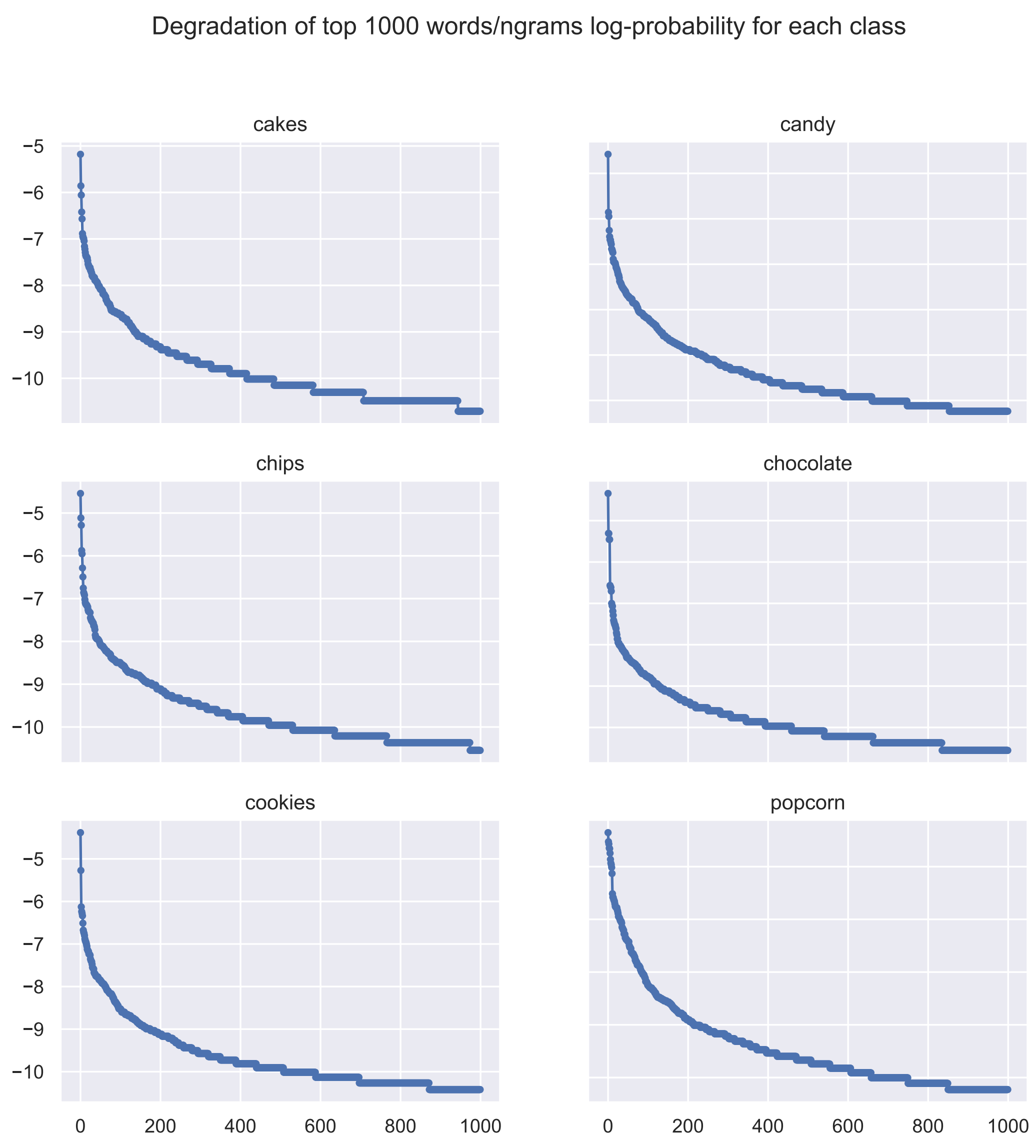
```
feature_names = vectorizer.get_feature_names_out()
class_names = [name.split("_")[0] for name in nb_classifier.classes_]
num_classes = len(class_names)

for i, class_name in enumerate(class_names):
    print(f"Most important words for class '{class_name}':")
    top_features_idx = nb_classifier.feature_log_prob_[i].argsort()[::-1][:10]
    top_features = [feature_names[idx] for idx in top_features_idx]
    print(", ".join(top_features), end="\n\n")
```

```
## Most important words for class 'cakes':
## cake, chocolate, pie, cupcakes, mini, cakes, creme, donuts, brownie, cheesecake
##
## Most important words for class 'candy':
## candy, chocolate, fruit, sour, gummi, jelly, milk, gummy, candies, chewy
##
## Most important words for class 'chips':
## chips, potato, potato chips, tortilla, tortilla chips, kettle, corn, salt, cooked, pretzels
##
## Most important words for class 'chocolate':
## chocolate, milk, dark, milk chocolate, dark chocolate, truffles, bar, caramel, chocolates, salt
##
## Most important words for class 'cookies':
## cookies, chocolate, cookie, chip, chocolate chip, sandwich, sugar, butter, wafers, creme
##
## Most important words for class 'popcorn':
## roasted, mix, almonds, popcorn, chocolate, peanuts, salted, trail, trail mix, cashews
```

The intersection between the most important words for each category is very small, but not empty. For example, the word `chocolate` is important for all categories but `chips_pretzels_snacks`. The words `creme`, `milk` and `salt` are among the 10 most important words for more than a single category. We'll find out what is the extent of important words/n-grams for each category, until they become uninformative.

```
import matplotlib.pyplot as plt
import numpy as np


fig, axs = plt.subplots(3, 2, figsize=(10, 10))
fig.suptitle("Degradation of top 1000 words/ngrams log-probability for each class")

for c in range(6):
    i, j = c // 2, c % 2
    axs[i, j].scatter(
        range(1000), np.sort(nb_classifier.feature_log_prob_[c])[-1000:][::-1], s=10
    )
    axs[i, j].plot(
        range(1000), np.sort(nb_classifier.feature_log_prob_[c])[-1000:][::-1]
    )
```

```
    axs[i, j].set_title(class_names[c])

for ax in fig.get_axes():
    ax.label_outer()

plt.show()
```



There are many different n-grams, possibly leading to big increase in the number of features. I'd like to try out a simple model first, replacing the `description` and `brand` columns, each with 6 columns - one for each category. The value of each column will be the log-probability of the feature column to associate with the category, according to the Naive Bayes model.

Replacing `brand` and `description` with their Naive Bayes scores:

```python
from helpers.preprocess import NaiveBayesScores

NaiveBayesScores(
    colname="brand", preprocess_func=lambda x: x.replace(" ", "")
).fit_transform(X=X_train, y=y_train)
```

```python
NaiveBayesScores(
    colname="description",
    vectorizer_kwgs=dict(
        stop_words="english", ngram_range=(1, 6), strip_accents="unicode"
    ),
).fit_transform(X=X_train, y=y_train)
```

```python
X_train[[f"{brand}_nb_score_chocolate" for brand in ["brand", "description"]]].head()
```

```
##        brand_nb_score_chocolate  description_nb_score_chocolate
## 23212                 -4.083942                      -11.153186
## 22158                 -2.620499                      -10.779114
## 1703                  -5.340313                      -35.202871
## 20886                 -5.111576                       -1.015086
## 18703                 -4.634516                      -27.093615
```

```python
X_train.iloc[:2, -6:] # description scores
```

```
##        description_nb_score_cakes  ...  description_nb_score_popcorn
## 23212                   -0.000296  ...                    -14.975742
## 22158                   -9.864426  ...                    -15.849332
##
## [2 rows x 6 columns]
```

```python
np.exp(X_train.iloc[:2, -6:]).sum(axis=1) # probabilities sum to 1
```

```
## 23212    1.0
## 22158    1.0
## dtype: float64
```

Eventually, I'd like to try running ensemble based models over the raw vectorization features, to see if we can bypass the accuracy attained by the Naive Bayes score features.

Serving Size Unit

```python
X_train["serving_size_unit"].value_counts()
```

```
## serving_size_unit
## g     25392
## ml        8
## Name: count, dtype: int64
```

```
X_train[X_train["serving_size_unit"] == "ml"]["y"]
```

```
## 11626    candy
## 12633    candy
## 19728    cakes
##          ...
## 9445     candy
## 8554     candy
## 23010    candy
## Name: y, Length: 8, dtype: object
```

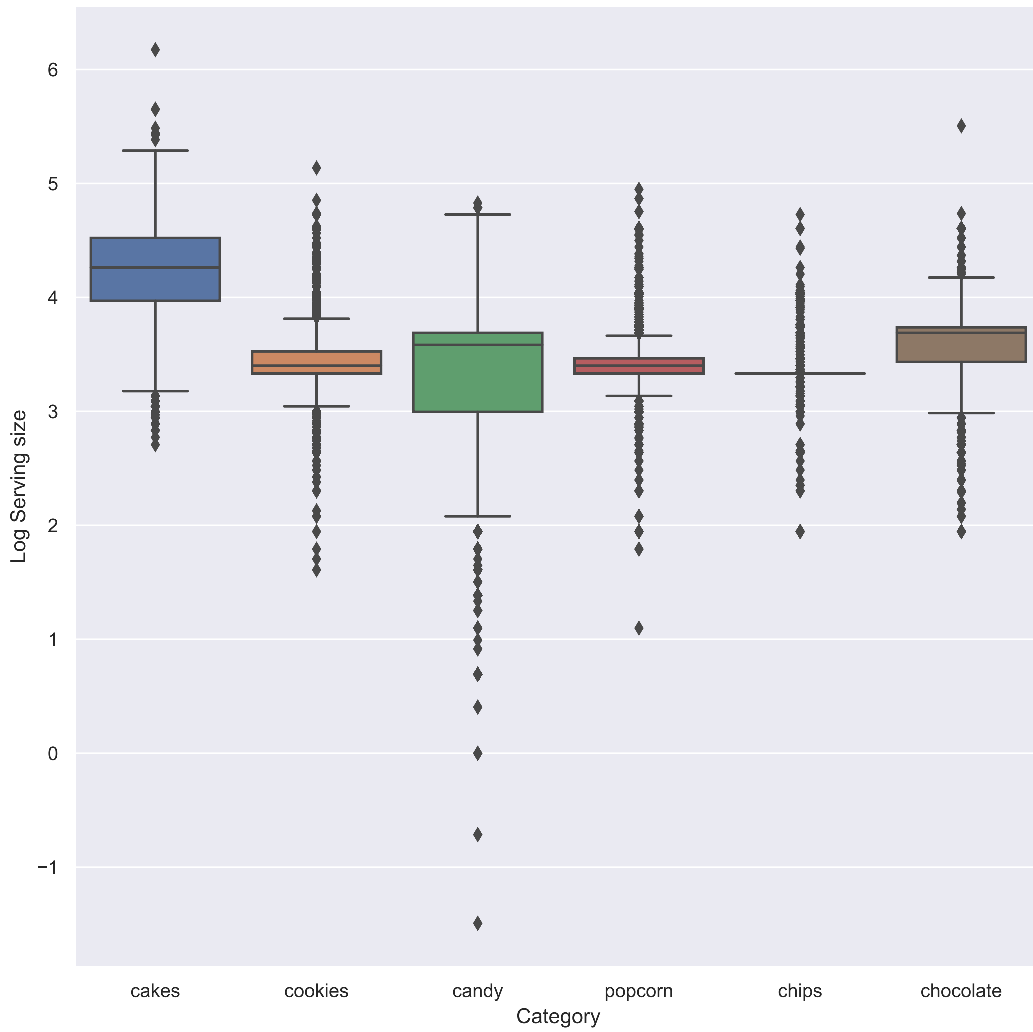
Not a very informative column, as >99.9% of the data is `g`. We'll drop this column.

```
from helpers.preprocess import DropColumns

DropColumns(columns=["serving_size_unit"]).fit_transform(X=X_train)
```

Serving Size

First, we'll take a look at the distribution of the serving size per category:

```
import seaborn as sns

ax = sns.boxplot(
    x=X_train["y"].apply(lambda x: x.split("_")[0]), y=np.log(X_train["serving_size"])
)
ax.set(xlabel="Category", ylabel="Log Serving size")
```



Can't really tell how helpful this column is, as the distribution is somewhat similar for some categories. It may be useful to know that when the serving size is low, the food is probably a candy, and there might be more patterns like that. This column is 'cheap' for our model, as it is a numeric column with no missing values. We'll keep this column for now, after applying log transformation.

```
from helpers.preprocess import LogTransformation

LogTransformation(columns=["serving_size"]).fit_transform(X=X_train)
```

Images Dataset

We tried ResNet18 as **fixed** feature extractor, to see how well can we predict the category by applying a linear classifier on top of the features extracted from the images ($ResNet18 : \mathbb{R}^{224\times224} \rightarrow \mathbb{R}^{1000}$).

Based on https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

The csv's of the extracted features were quite heavy (300mb), so I'll just share the findings (code blocks are still in the rmd file, as we intended to demonstrate the process).

Using ResNet18 as a frozen net, and applying a LogisticRegression over the output layer, results with 52% accuracy, way more than randomness can explain. Recall that we didn't touch any of the original net weights.

Later we'll fine-tune ResNet18 and add the score features as columns.

ResNet code at `resnet.py`.