final_project_knitted

August 6, 2022

[ ]:
```
%reset -f
```

[ ]:
```
# Import stuff
# !pip install pandas numpy seaborn sklearn nltk missingno tensorflow dask␣
 ↪hvplot
# !pip install xgboost --upgrade
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
from plotly.subplots import make_subplots
import plotly.graph_objects as go
import missingno as msno
from sklearn import model_selection, preprocessing, ensemble, neighbors,␣
 ↪linear_model, svm, metrics
from nltk.util import ngrams
from imblearn import over_sampling, under_sampling, pipeline
from PIL import Image
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer, SimpleImputer
from sklearn import metrics
from xgboost import XGBClassifier, XGBRegressor # 1.6.1
from sklearn.pipeline import Pipeline
import itertools
from sklearn.utils import class_weight
import plotly.io as pio
pio.renderers.default = "notebook+pdf"
```

First we'll define some useful functions for later:

[ ]:
```
# Define useful functions

"""
Returns dictionary of textual column values and their frequency in dataframe
"""
def get_column_values(df, exceptions=[]):
```

```python
    n = len(df)
    text_columns = df.select_dtypes(exclude=[np.number])
    column_values = {}
    for column in text_columns:
        # Make frequencies dictionary and calculated percent of missing values
        if column not in exceptions:
            column_values[column] = df[column].value_counts().to_frame().
 ↪reset_index()
            column_values[column].columns = ['value', 'count']
            column_values[column]['pc'] = column_values[column]['count'] / n *␣
 ↪100
    return column_values


"""
Cleans text values in dataframe for EDA purposes:
0. Replaces spaces with underscore
1. Removes non-ASCII characters
2. Transforms to lowercase
3. Fills null values with "NA"
4. Removes punctuation characters (except underscore and .)
"""
from string import punctuation
def clean_text_values(df):
    # punc = [p for p in punctuation if p not in ['_']]
    column_values = get_column_values(df)
    for column in column_values:
        df[column].str.encode('ascii', 'ignore').str.decode('ascii')
        df[column] = df[column].str.lower()
        df[column].fillna('NA', inplace=True)
        if column != 'category':
            for p in punctuation:
                df[column] = df[column].str.replace(p, '', regex=False)
        df[column] = df[column].str.replace(r'\s+', ' ', regex=True)
    return df


"""
Recieves df, categories and column name as arguments and returns a Histogram␣
 ↪object of that column divided by the different categories, without
zeros and values above the 99th percentile to clear extreme values which mess␣
 ↪up the histogram (calculated after removing zeros).
"""
def create_column_histogram_by_category(df, categories, column):
    all_data = []
    num_99_percentile = 0
    for category in categories:
```

```python
        data_no_zeros = df[(df['category'] == category) & (df[column] > 0)]
        data = data_no_zeros[(data_no_zeros[column] <= data_no_zeros[column].quantile(0.99))]
        num_99_percentile = len(data_no_zeros) - len(data)
        all_data.append(go.Histogram(
            x=data[column],
            name=category,
            opacity=0.7
            # ,xbins={
            #     'start': 0,
            #     'end': data.max(),
            #     'size': data.max() / 500
            # }
            )
        )
    num_zeros = len(df[df[column] == 0])
    layout = go.Layout(
        barmode='overlay',
        title='{} histograms divided by class  |  No. of zeros: {}  |  No. above 99th percentile: {}\n| Percent "bad" rows: {}%'.format(
            column.capitalize().replace('_', ' '), num_zeros, num_99_percentile, round((num_zeros + num_99_percentile) / len(df) * 100, 3))
    )
    fig = go.Figure(data=all_data, layout=layout)
    return fig


def add_ngrams_columns(df, column, n, return_column=False):
    new_ngrams_column = '{}_{}gram'.format(column, n)
    df[new_ngrams_column] = df[df[column].notna()][column].apply(lambda row: list(ngrams(row.split(' '), n)))
    if return_column:
        return df, new_ngrams_column
    return df


def get_ngrams(df, column, n, remove_column=False):
    df, new_ngrams_column = add_ngrams_columns(df, column, n, return_column=True)
    categories = list(df['category'].unique())
    n_grams = {cat: {} for cat in categories}
    for category in categories:
        cat_df = df[df['category'] == category]
        for ngram_list in cat_df[cat_df[new_ngrams_column].notna()][new_ngrams_column]:
            if type(ngram_list) == float:
```

```
                print(ngram_list)
            for n_gram in ngram_list:
                if n_gram in n_grams[category].keys():
                    n_grams[category][n_gram] += 1
                else:
                    n_grams[category][n_gram] = 1
    if remove_column:
        df = df.drop([new_ngrams_column], axis=1)
    return df, n_grams


def get_ngrams_top_k(df, column, n, k=1, remove_column=False):
    df, column_ngrams = get_ngrams(df, column, n, remove_column)
    categories = list(df['category'].unique())
    top_20_column_ngrams_by_category = {}
    for category in categories:
        top_column_ngrams = sorted(column_ngrams[category],␣
 ↪key=column_ngrams[category].get, reverse=True)[:k]
        top_20_column_ngrams = [(" ".join(key), column_ngrams[category][key])␣
 ↪for key in column_ngrams[category].keys() if key in top_column_ngrams]
        top_20_column_ngrams = pd.DataFrame(top_20_column_ngrams,␣
 ↪columns=['ngram', 'count']).sort_values('count')
        top_20_column_ngrams_by_category[category] = top_20_column_ngrams
    return df, top_20_column_ngrams_by_category


def get_dims(file):
    img = Image.open(file)
    try:
        h, w, d = np.array(img).shape
    except:
        h, w = np.array(img).shape
    img.close()
    return h, w
```

Let us load the data:

[ ]:
```
dataset = pd.read_csv('data/food_train.csv')
fact_nutrients = pd.read_csv('data/food_nutrients.csv')
dim_nutrients = pd.read_csv('data/nutrients.csv')
```

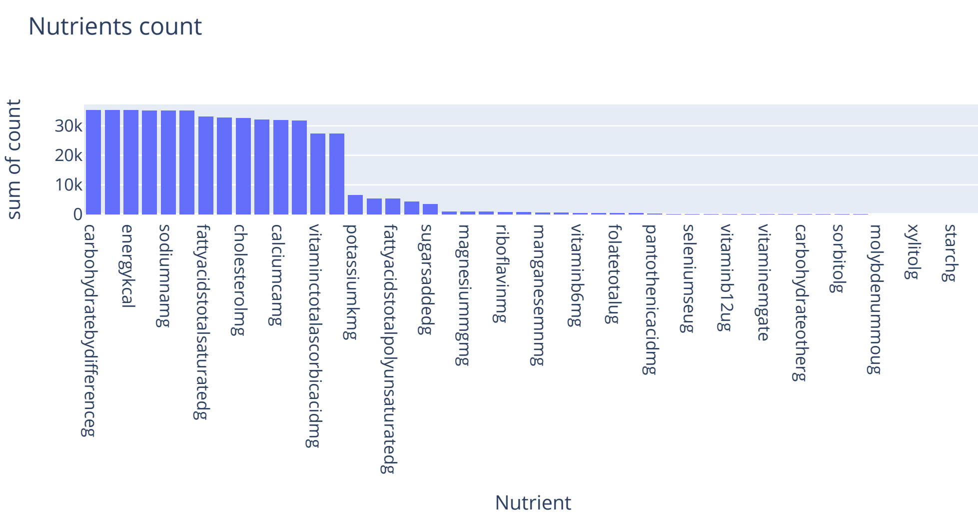
Then merge and clean it a bit:

[ ]:
```
# Join nutrients tables and clean it
nutrients = pd.merge(fact_nutrients, dim_nutrients, how='left',␣
 ↪on='nutrient_id')
nutrients = clean_text_values(nutrients)
```

```
nutrients['name'] = nutrients['name'].str.replace(' ', '_')
nutrients['name'] = nutrients['name'] + '_' + nutrients['unit_name']
nutrients_column_values = get_column_values(nutrients)
```

See how many records we have of each nutrient:

```
[ ]: # Draw histogram of nutrients using plotly
num_nutrients = nutrients.groupby('name')['idx'].count().reset_index()
num_nutrients.columns = ['nutrient', 'count']
num_nutrients['pc'] = num_nutrients['count'] / len(dataset) * 100
num_nutrients = num_nutrients[num_nutrients['count'] > 0]
num_nutrients = clean_text_values(num_nutrients)
px.histogram(
    num_nutrients.sort_values('count', ascending=False), x='nutrient',␣
 ↪y='count', title='Nutrients count', width=800, height=400,
    labels={'nutrient':'Nutrient'}
)
```



Now let's left join the nutrients to our snacks dataset:

```
[ ]: # Left join nutrients to dataset
joined_dataset = pd.merge(dataset, nutrients, how='left', on='idx').
 ↪drop(['nutrient_id', 'unit_name'], axis=1)
```

Good ol' pivot table and join:

```
[ ]: # Pivot table - started by resetting index and rearranging columns but decided␣
 ↪to join again to avoid pandas index trouble
pivoted_dataset = pd.pivot_table(
```

```
    joined_dataset,
    values='amount',
    index='idx',
    columns='name')

dataset_w_nutrients = pd.merge(dataset, pivoted_dataset, how='left', on='idx')
```

Change categories names and encode them for convenience:

```
[ ]: dataset_w_nutrients.loc[dataset_w_nutrients['category'] ==␣
      ↪'cakes_cupcakes_snack_cakes', 'category'] = 'cakes'
     dataset_w_nutrients.loc[dataset_w_nutrients['category'] ==␣
      ↪'chips_pretzels_snacks', 'category'] = 'snacks'
     dataset_w_nutrients.loc[dataset_w_nutrients['category'] == 'cookies_biscuits',␣
      ↪'category'] = 'cookies'
     dataset_w_nutrients.loc[dataset_w_nutrients['category'] ==␣
      ↪'popcorn_peanuts_seeds_related_snacks', 'category'] = 'seeds'
     le = preprocessing.LabelEncoder()
     le.fit(dataset_w_nutrients['category'])
     dataset_w_nutrients['category_enc'] = le.
      ↪transform(dataset_w_nutrients['category'])
     le.classes_
```

```
[ ]: array(['cakes', 'candy', 'chocolate', 'cookies', 'seeds', 'snacks'],
           dtype=object)
```

Now we can split our data for initial analysis!

```
[ ]: # Split to perform EDA only on train data so conclusions won't later leak to␣
      ↪test set
     X = dataset_w_nutrients.drop(['category'], axis=1)
     y = dataset_w_nutrients['category']
     X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,␣
      ↪test_size=0.2, random_state=1337, stratify=y)
     eda_df = pd.concat([X_train, y_train], axis=1)
```

Clean text values:

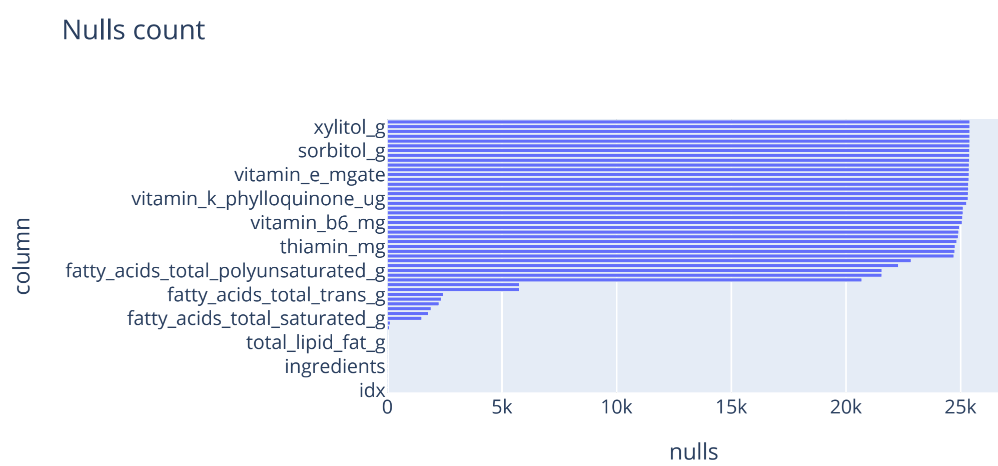
```
[ ]: eda_df = clean_text_values(eda_df)
```

Hajime!

First of all, let's check for nulls:

```
[ ]: nulls = eda_df.isnull().sum().to_frame().reset_index()
     nulls.columns = ['column', 'nulls']
     nulls = nulls.sort_values('nulls')
```

```
px.bar(nulls, x='nulls', y='column', orientation='h', title='Nulls count',␣
↪width=700, height=350)
#nulls.head(2)
```



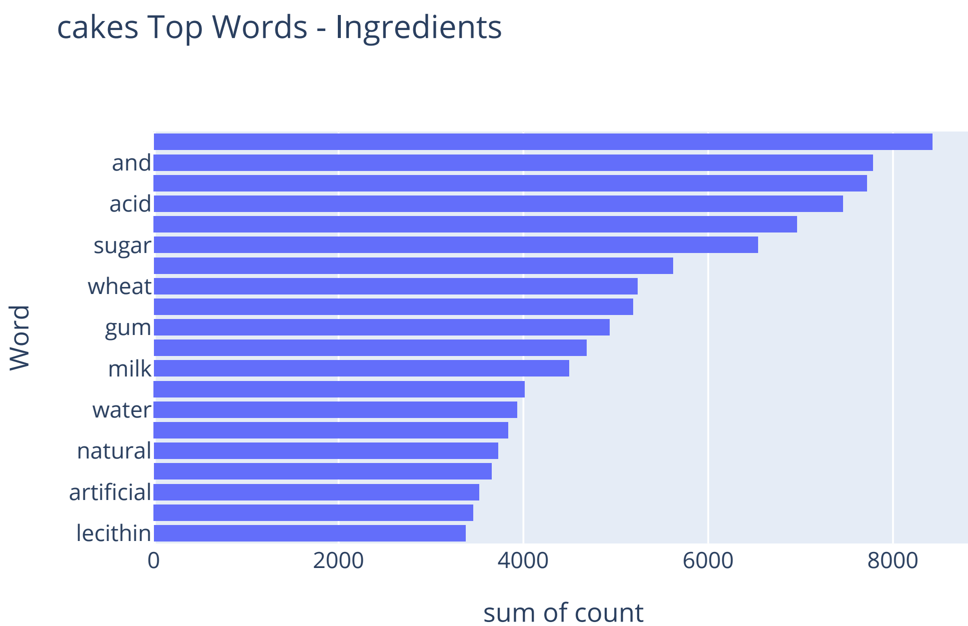
[ ]:
```
# Code to remove high nulls nutrients - would maybe use later

# eda_n = len(eda_df)
# nutrients_to_drop = []
# for nutrient in nutrients_column_values['name']['value']:
#     num_zeros = (eda_df[nutrient].isna()).astype(int).sum()
#     pc_zeros = num_zeros / eda_n * 100
#     if pc_zeros >= 80:
#         nutrients_to_drop.append(nutrient)
# eda_df = eda_df.drop(nutrients_to_drop, axis=1)
```

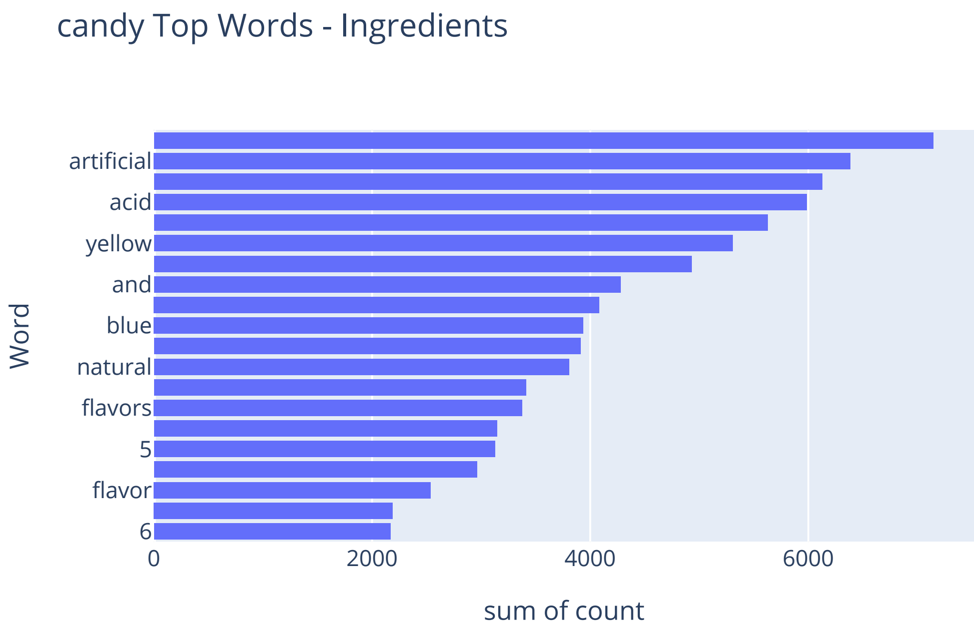
[ ]:
```
eda_df, top20_ingredients_words_by_category = get_ngrams_top_k(eda_df,␣
↪'ingredients', 1, 20)
```

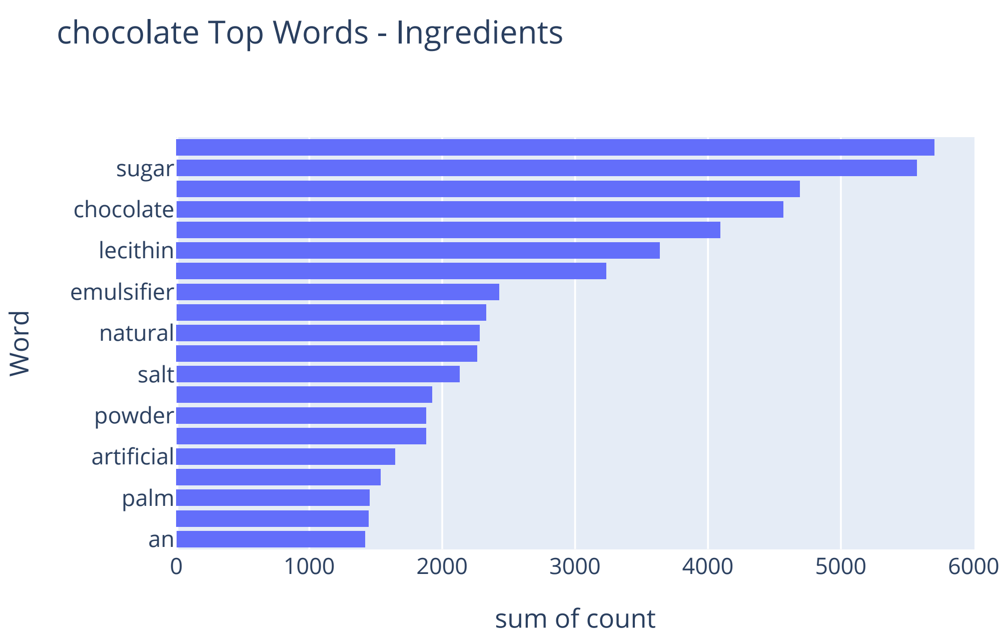
Let's have a look at some of the common words in the ingredients column:

[ ]:
```
# Get an idea of distinct words in 'ingredients' column
eda_df, top20_ingredients_words_by_category = get_ngrams_top_k(eda_df,␣
↪'ingredients', 1, 20)
for category in le.classes_:
    px.histogram(
        top20_ingredients_words_by_category[category], y='ngram', x='count',␣
↪orientation='h', width=600, height=400,
```

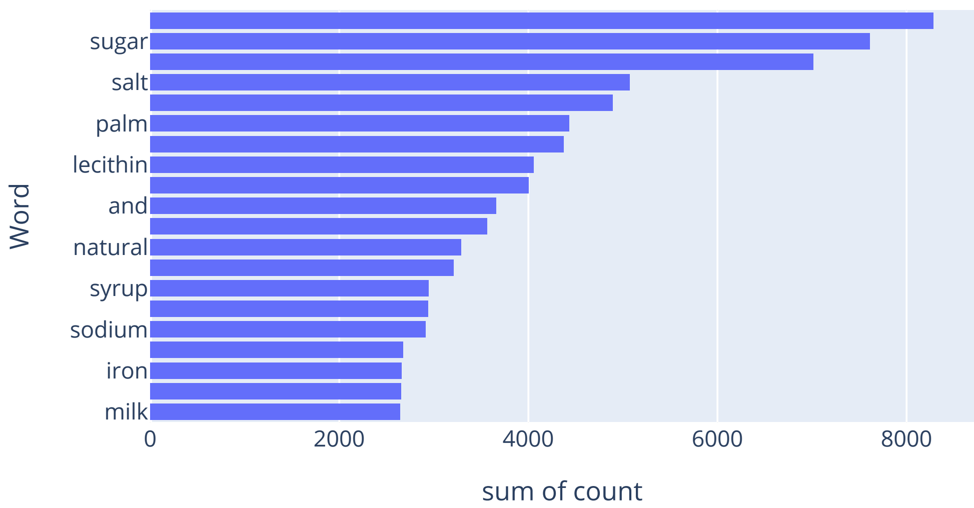
```
        labels={'ngram': 'Word'}, title='{} Top Words - Ingredients'.
↪format(category)
    ).show()
```



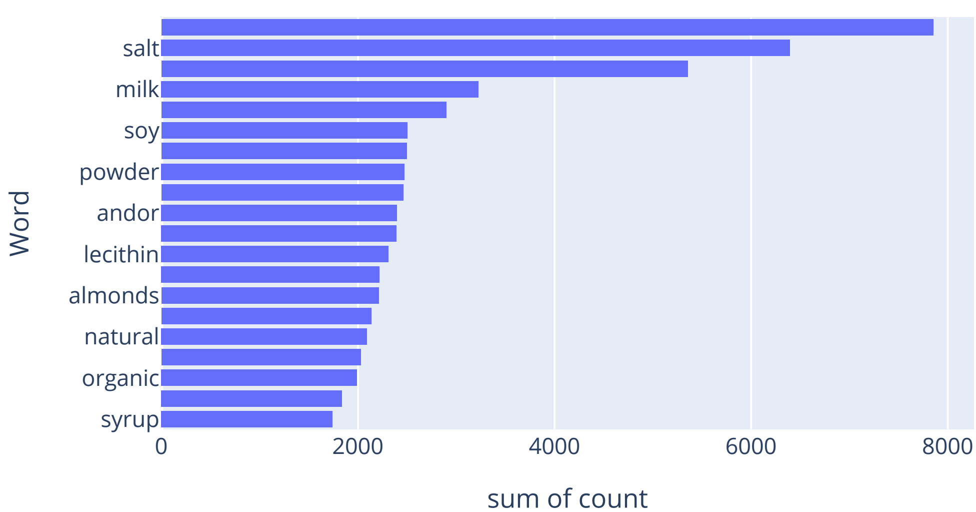

candy Top Words - Ingredients
artificial
acid
yellow
and
blue
natural
flavors
5
flavor
6
Word
0
2000
4000
6000
sum of count

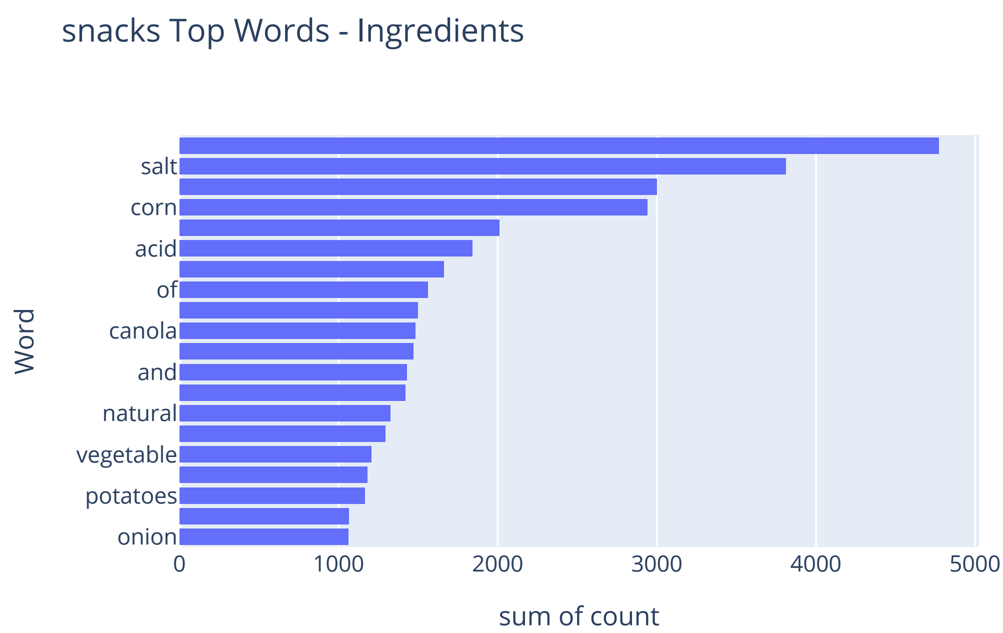

chocolate Top Words - Ingredients
sugar
chocolate
lecithin
emulsifier
natural
salt
powder
artificial
palm
an
Word
0
1000
2000
3000
4000
5000
6000
sum of count

cookies Top Words - Ingredients



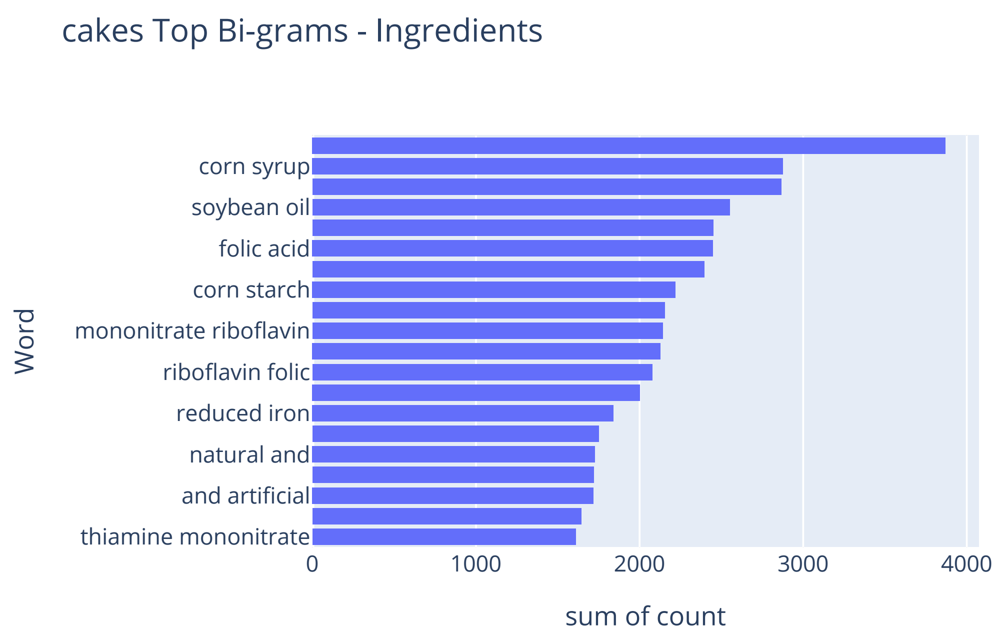
seeds Top Words - Ingredients

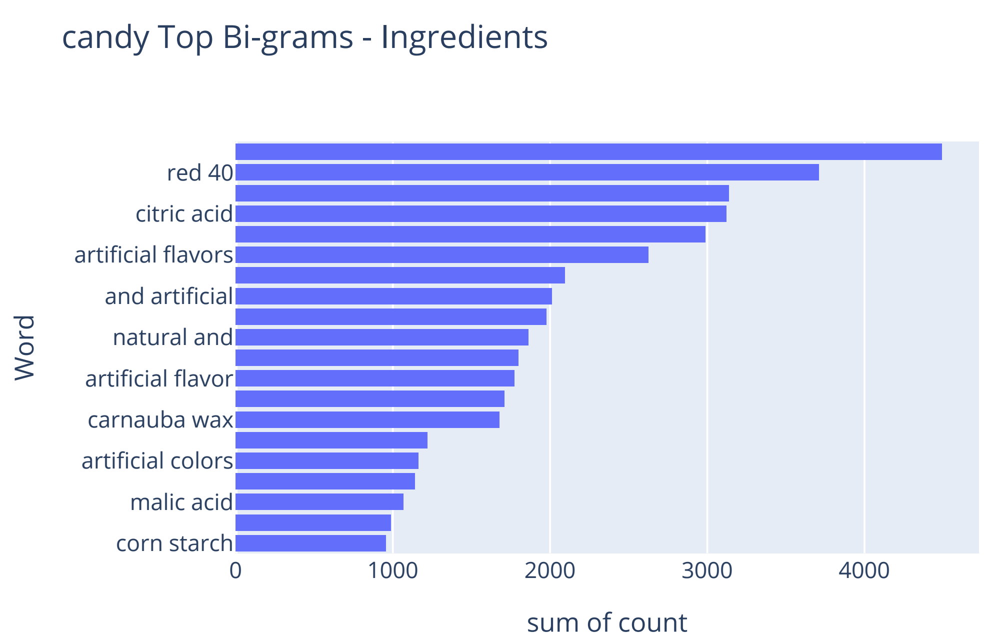




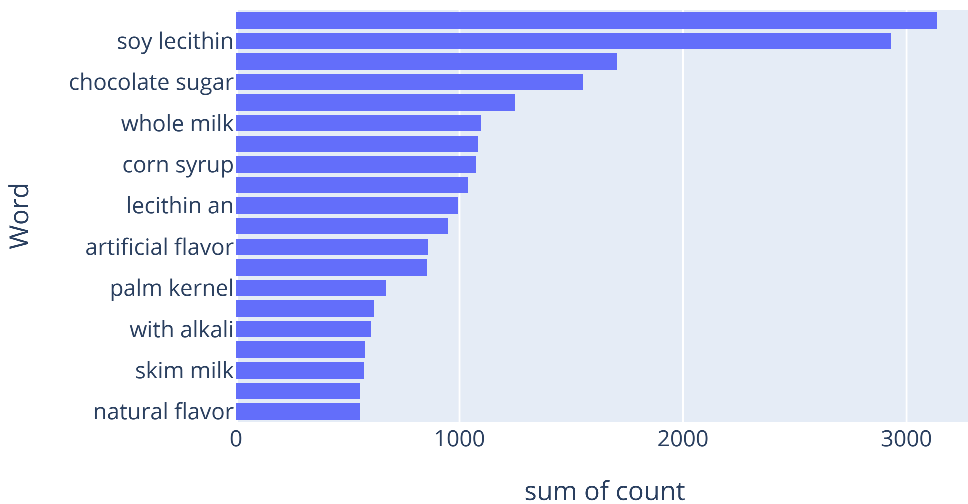
We can see we have a few words like 'and', 'or' in the top. Let us look at bigrams:

```
[ ]: # Do same for bigrams
eda_df, top20_ingredients_bigrams_by_category = get_ngrams_top_k(eda_df,␣
 ↪'ingredients', 2, 20)
for category in le.classes_:
    px.histogram(
        top20_ingredients_bigrams_by_category[category], y='ngram', x='count',␣
 ↪orientation='h', width=600, height=400,
        labels={'ngram': 'Word'}, title='{} Top Bi-grams - Ingredients'.
 ↪format(category)
    ).show()
```

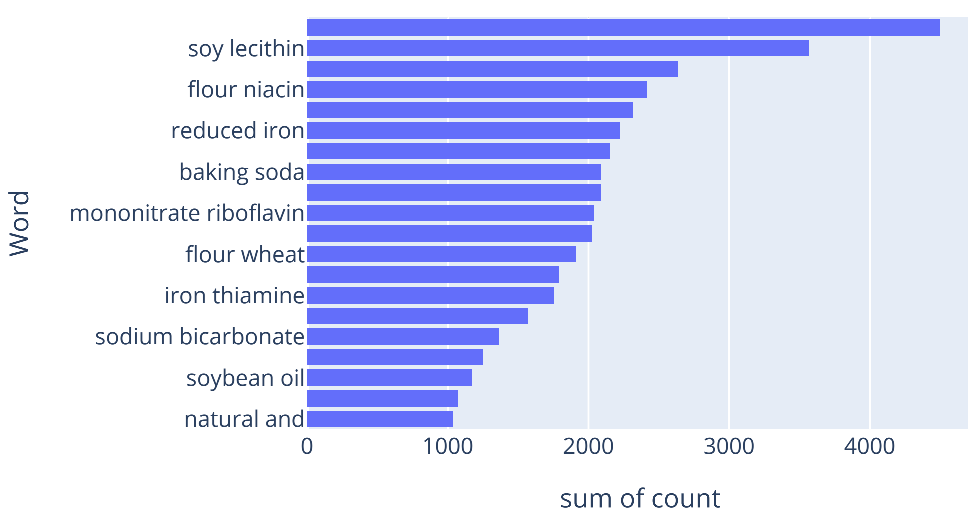

cakes Top Bi-grams - Ingredients
corn syrup
soybean oil
folic acid
corn starch
mononitrate riboflavin
riboflavin folic
reduced iron
natural and
and artificial
thiamine mononitrate
Word
0
1000
2000
3000
4000
sum of count

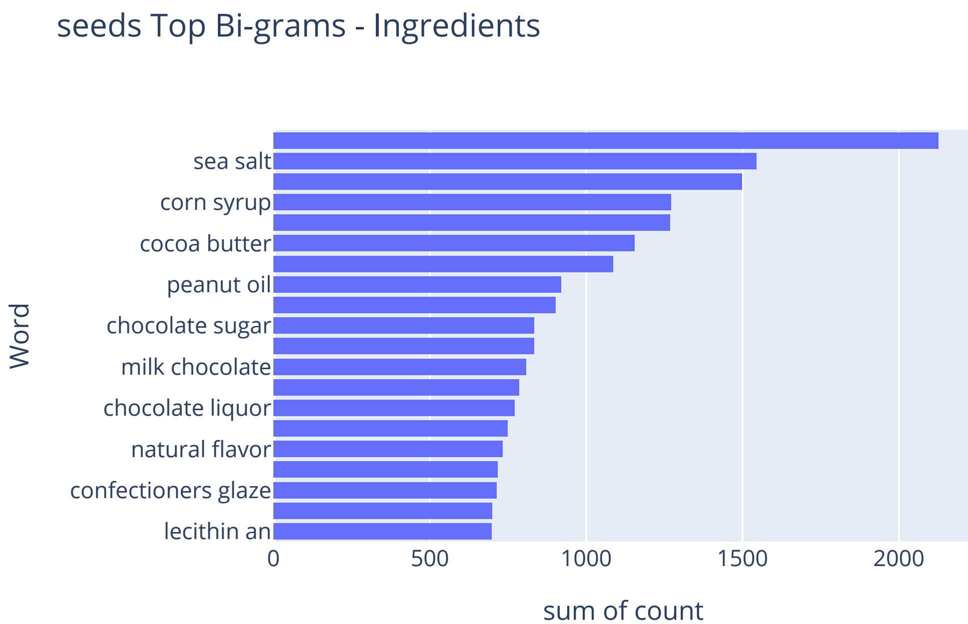

candy Top Bi-grams - Ingredients
red 40
citric acid
artificial flavors
and artificial
natural and
artificial flavor
carnauba wax
artificial colors
malic acid
corn starch
Word
0
1000
2000
3000
4000
sum of count

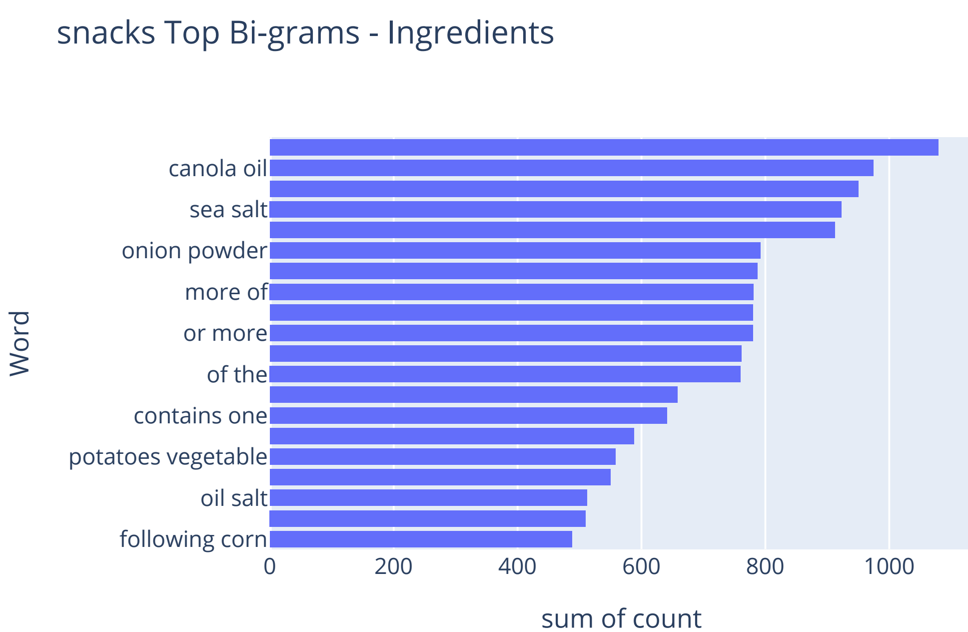
chocolate Top Bi-grams - Ingredients



cookies Top Bi-grams - Ingredients

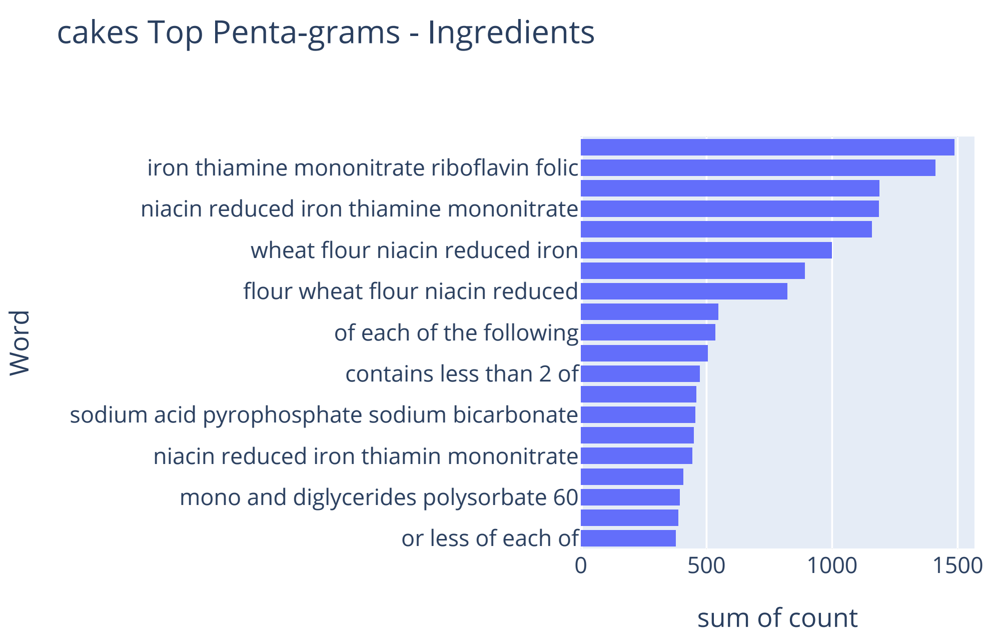


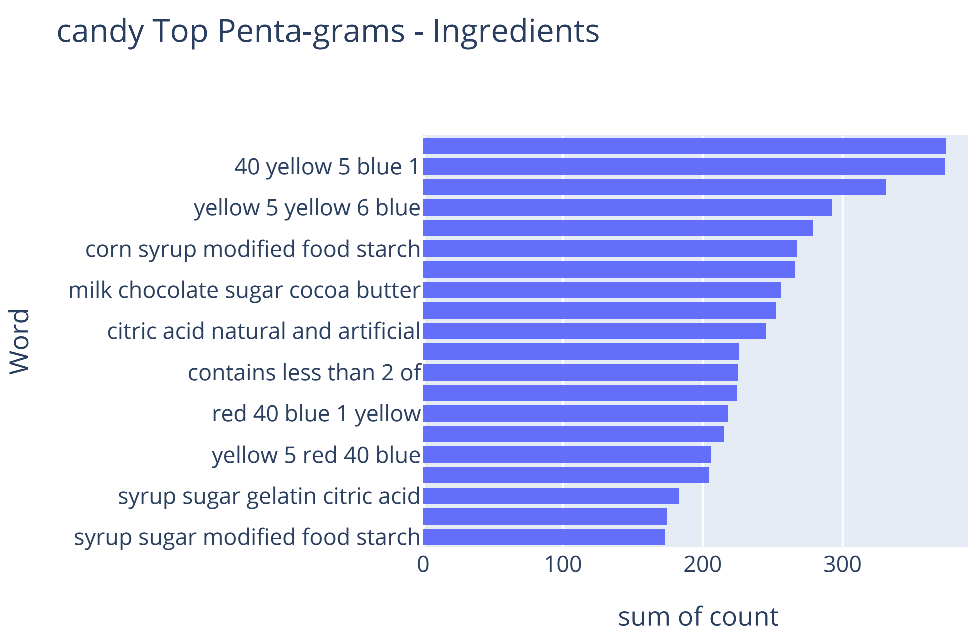

seeds Top Bi-grams - Ingredients
Word
sea salt
corn syrup
cocoa butter
peanut oil
chocolate sugar
milk chocolate
chocolate liquor
natural flavor
confectioners glaze
lecithin an
0
500
1000
1500
2000
sum of count

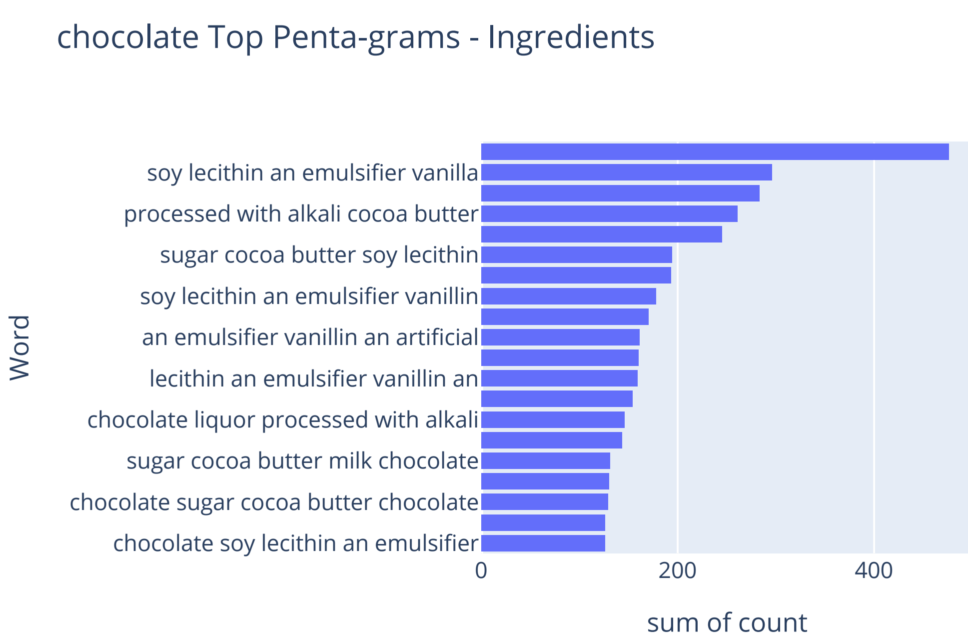

snacks Top Bi-grams - Ingredients
Word
canola oil
sea salt
onion powder
more of
or more
of the
contains one
potatoes vegetable
oil salt
following corn
0
200
400
600
800
1000
sum of count

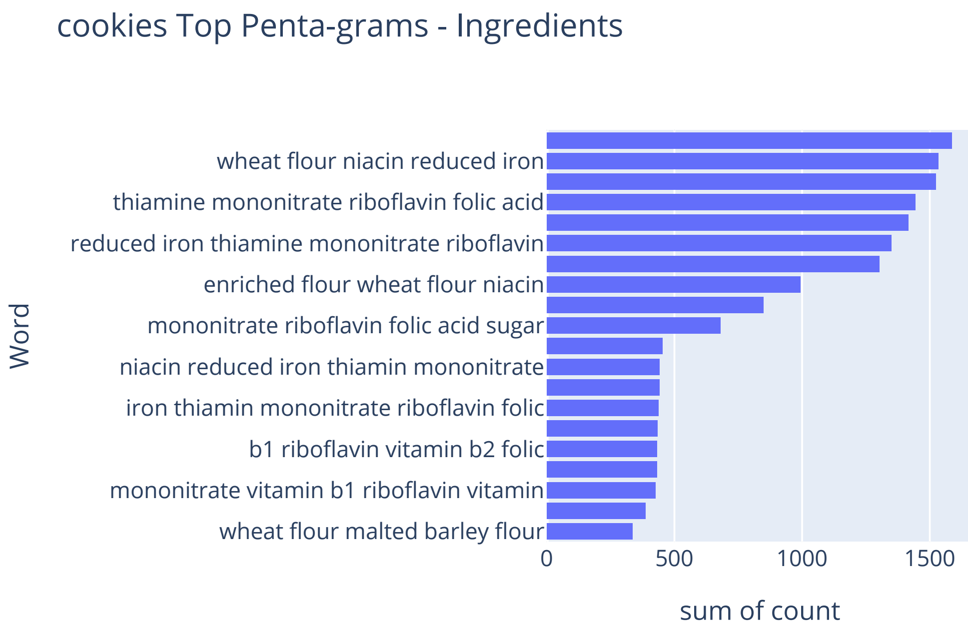
Seems we have some noise from the artificial colors and sentences. We'll keep those for now. Let us look for sentences by looking for common 5-grams:

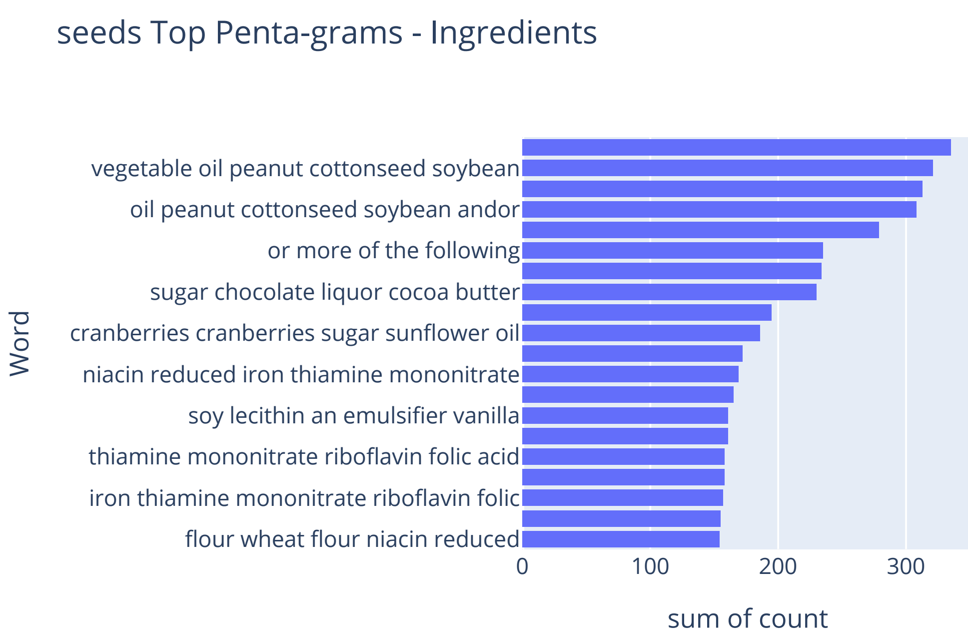
```
[ ]: eda_df, top20_ingredients_bigrams_by_category = get_ngrams_top_k(eda_df,␣
      ↪'ingredients', 5, 20)
     for category in le.classes_:
         px.histogram(
             top20_ingredients_bigrams_by_category[category], y='ngram', x='count',␣
      ↪orientation='h', width=600, height=400,
             labels={'ngram': 'Word'}, title='{} Top Penta-grams - Ingredients'.
      ↪format(category)
         ).show()
```

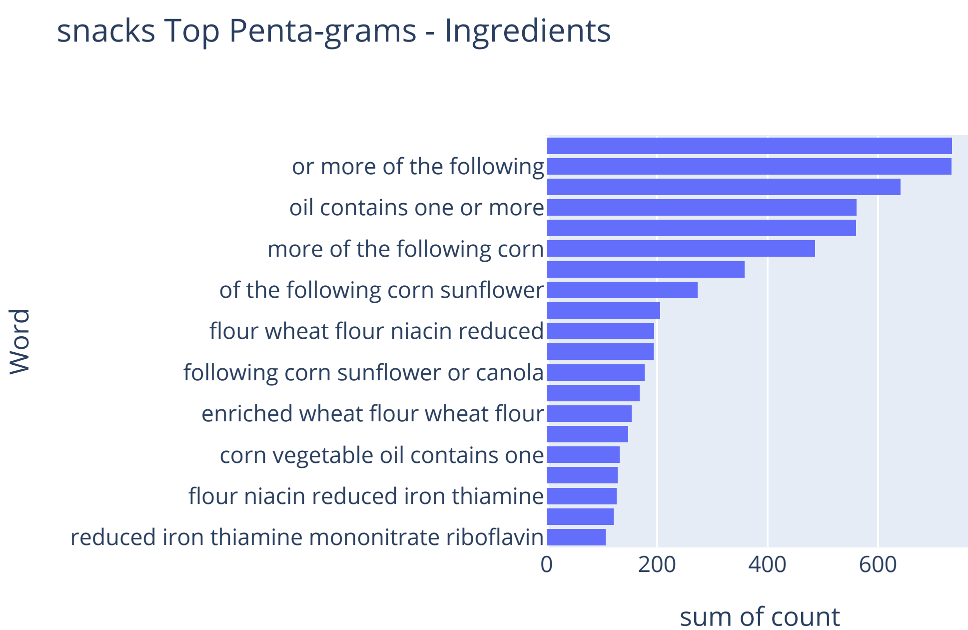



candy Top Penta-grams - Ingredients
40 yellow 5 blue 1
yellow 5 yellow 6 blue
corn syrup modified food starch
milk chocolate sugar cocoa butter
citric acid natural and artificial
contains less than 2 of
red 40 blue 1 yellow
yellow 5 red 40 blue
syrup sugar gelatin citric acid
syrup sugar modified food starch
Word
0
100
200
300
sum of count


chocolate Top Penta-grams - Ingredients
soy lecithin an emulsifier vanilla
processed with alkali cocoa butter
sugar cocoa butter soy lecithin
soy lecithin an emulsifier vanillin
an emulsifier vanillin an artificial
lecithin an emulsifier vanillin an
chocolate liquor processed with alkali
sugar cocoa butter milk chocolate
chocolate sugar cocoa butter chocolate
chocolate soy lecithin an emulsifier
Word
0
200
400
sum of count


cookies Top Penta-grams - Ingredients
wheat flour niacin reduced iron
thiamine mononitrate riboflavin folic acid
reduced iron thiamine mononitrate riboflavin
enriched flour wheat flour niacin
mononitrate riboflavin folic acid sugar
niacin reduced iron thiamin mononitrate
iron thiamin mononitrate riboflavin folic
b1 riboflavin vitamin b2 folic
mononitrate vitamin b1 riboflavin vitamin
wheat flour malted barley flour
Word
0
500
1000
1500
sum of count


seeds Top Penta-grams - Ingredients
vegetable oil peanut cottonseed soybean
oil peanut cottonseed soybean andor
or more of the following
sugar chocolate liquor cocoa butter
cranberries cranberries sugar sunflower oil
niacin reduced iron thiamine mononitrate
soy lecithin an emulsifier vanilla
thiamine mononitrate riboflavin folic acid
iron thiamine mononitrate riboflavin folic
flour wheat flour niacin reduced
Word
0
100
200
300
sum of count



We see that we have some sentences and numbers in the ingredients column.

```
[ ]: # eda_df, pentagrams = get_ngrams(eda_df, 'ingredients', 5)
# top_ingredient_pentagrams = sorted(pentagrams, key=pentagrams.get,␣
↪reverse=True)[:20]
# top_20_ingredient_pentagrams = [(key, pentagrams[key]) for key in pentagrams.
↪keys() if key in top_ingredient_pentagrams]
# top_20_ingredient_pentagrams
```

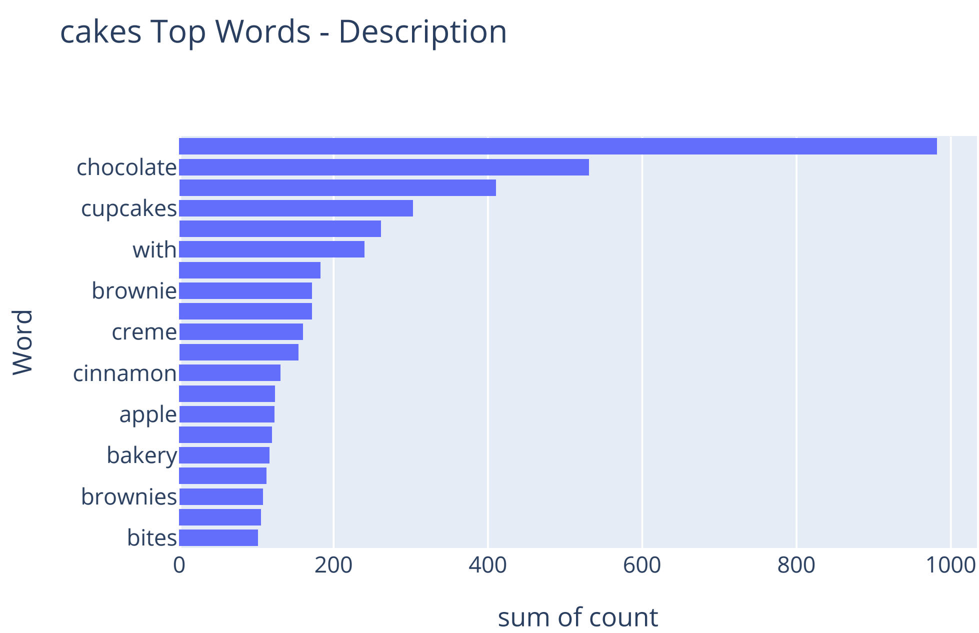
Let us do the same for the description column:

```
[ ]: eda_df, top20_description_words_by_category = get_ngrams_top_k(eda_df,␣
↪'description', 1, 20)
eda_df, top20_description_bigrams_by_category = get_ngrams_top_k(eda_df,␣
↪'description', 2, 20)
eda_df, top20_description_trigrams_by_category = get_ngrams_top_k(eda_df,␣
↪'description', 3, 20)
```

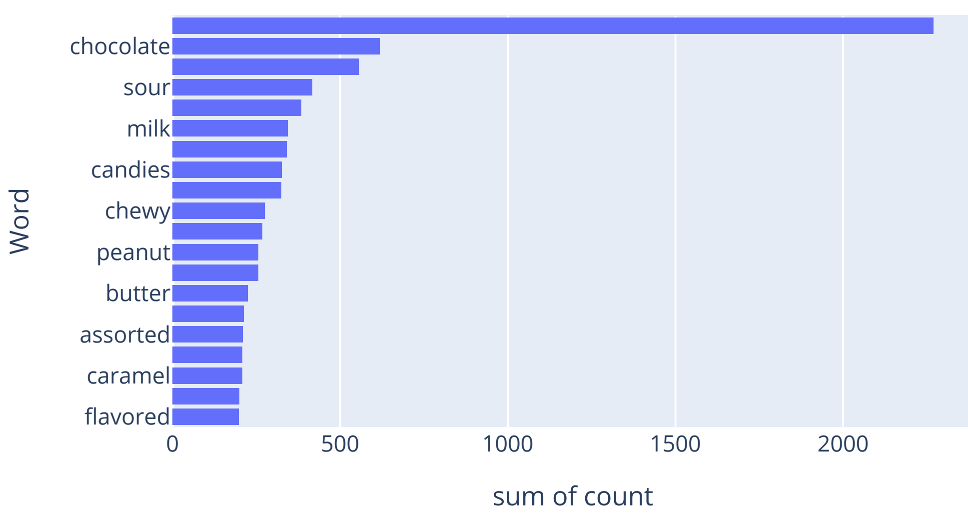
Top words:

```
[ ]: for category in le.classes_:
    px.histogram(
        top20_description_words_by_category[category], y='ngram', x='count',␣
↪orientation='h', width=600, height=400,
```

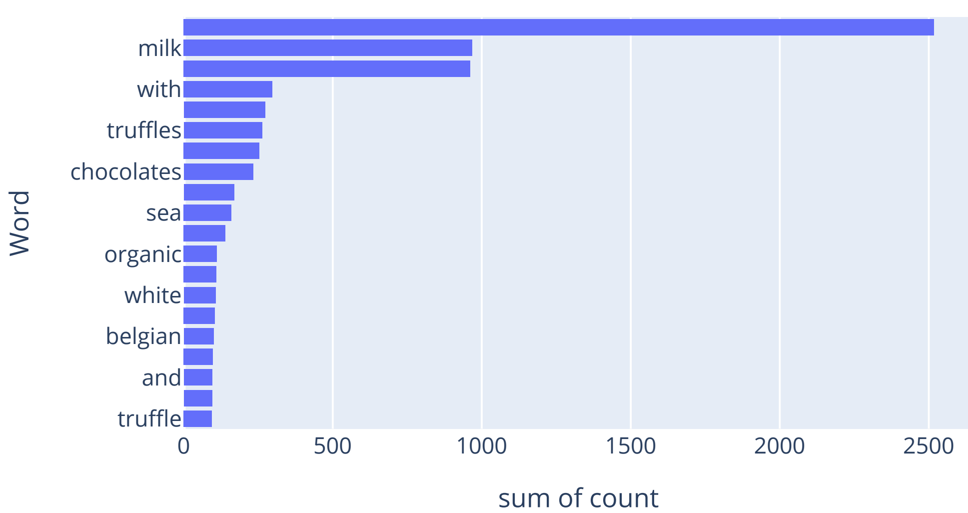
```
        labels={'ngram': 'Word'}, title='{} Top Words - Description'.
↪format(category)
    ).show()
```

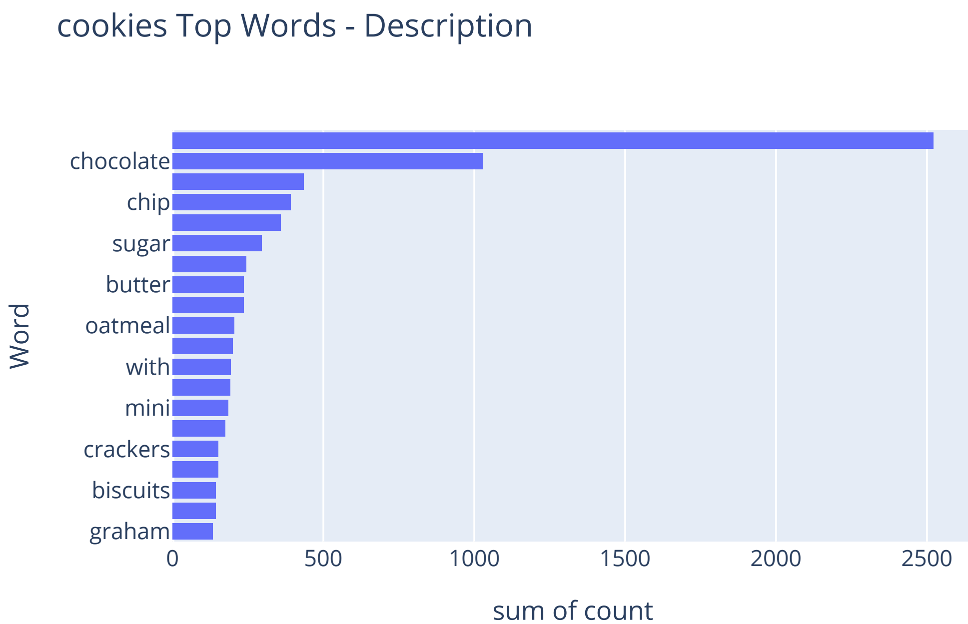


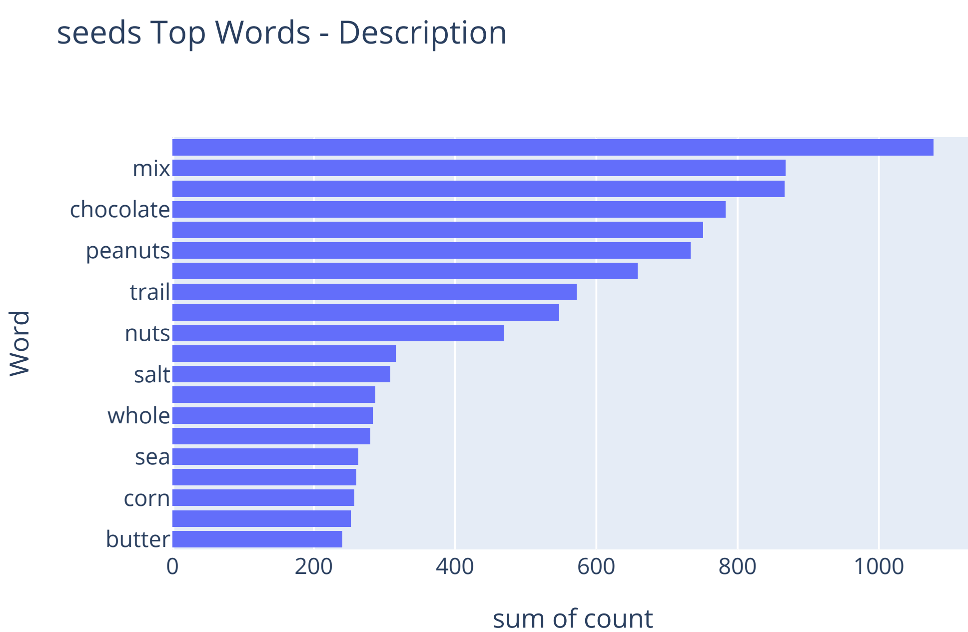
candy Top Words - Description

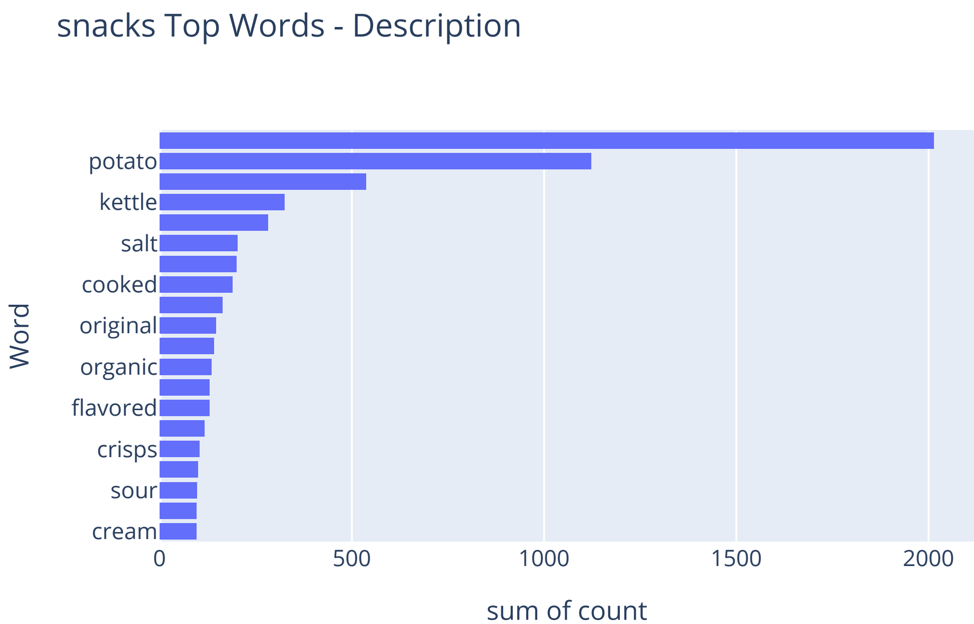


chocolate Top Words - Description



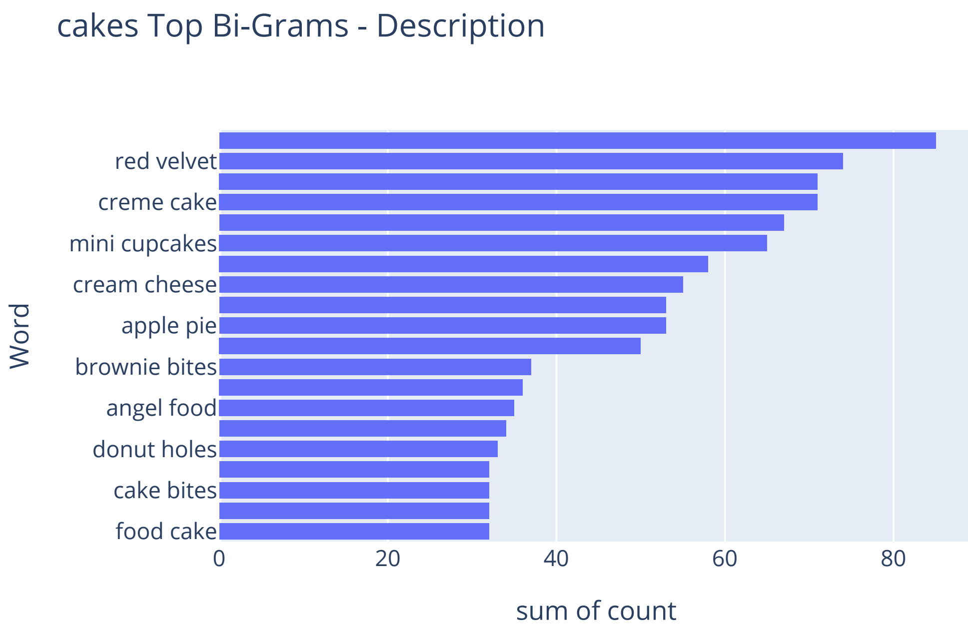

cookies Top Words - Description
Word
chocolate
chip
sugar
butter
oatmeal
with
mini
crackers
biscuits
graham
0
500
1000
1500
2000
2500
sum of count


seeds Top Words - Description
Word
mix
chocolate
peanuts
trail
nuts
salt
whole
sea
corn
butter
0
200
400
600
800
1000
sum of count

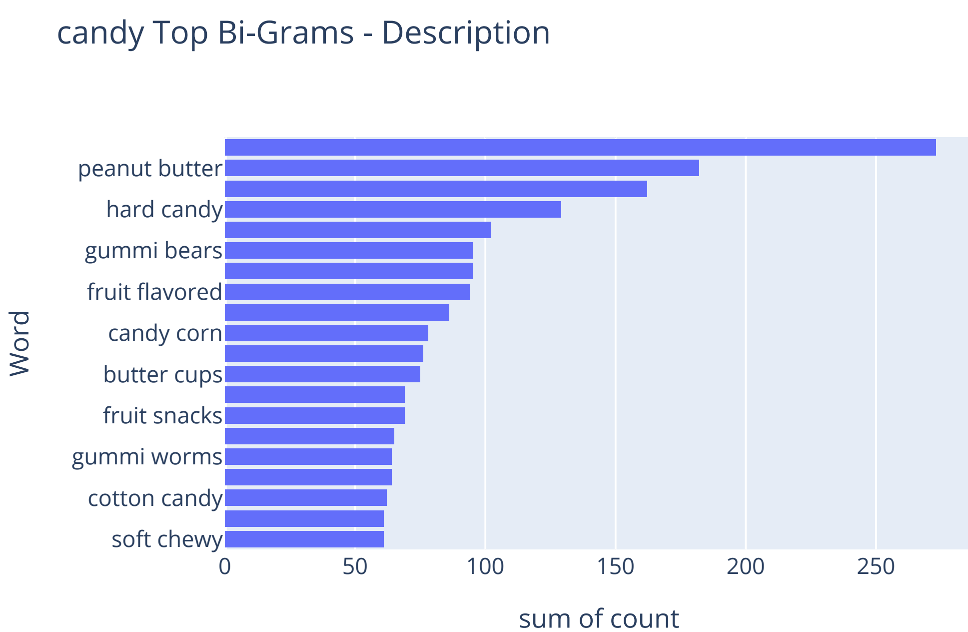


Top Bi-grams:

```
[ ]: for category in le.classes_:
         px.histogram(
             top20_description_bigrams_by_category[category], y='ngram', x='count',␣
     ↪orientation='h', width=600, height=400,
             labels={'ngram': 'Word'}, title='{} Top Bi-Grams - Description'.
     ↪format(category)
         ).show()
```

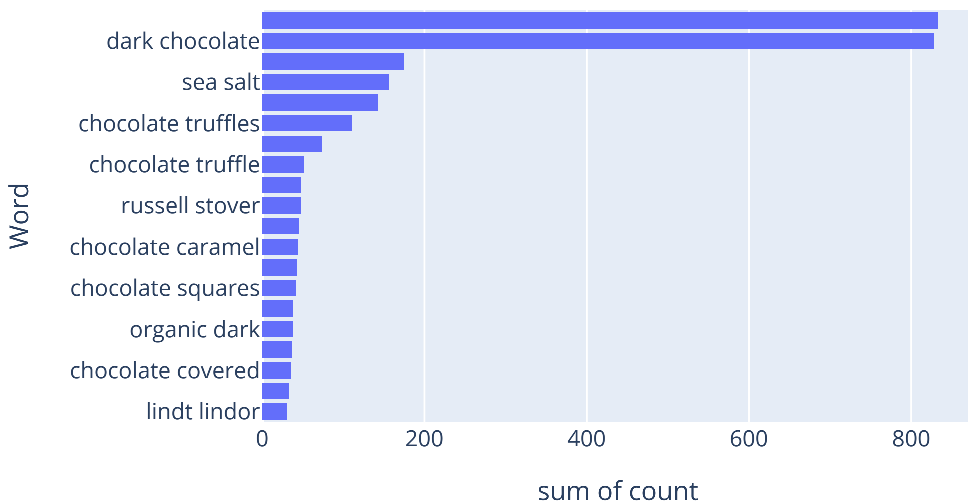


cakes Top Bi-Grams - Description
red velvet
creme cake
mini cupcakes
cream cheese
apple pie
brownie bites
angel food
donut holes
cake bites
food cake
Word
0
20
40
60
80
sum of count

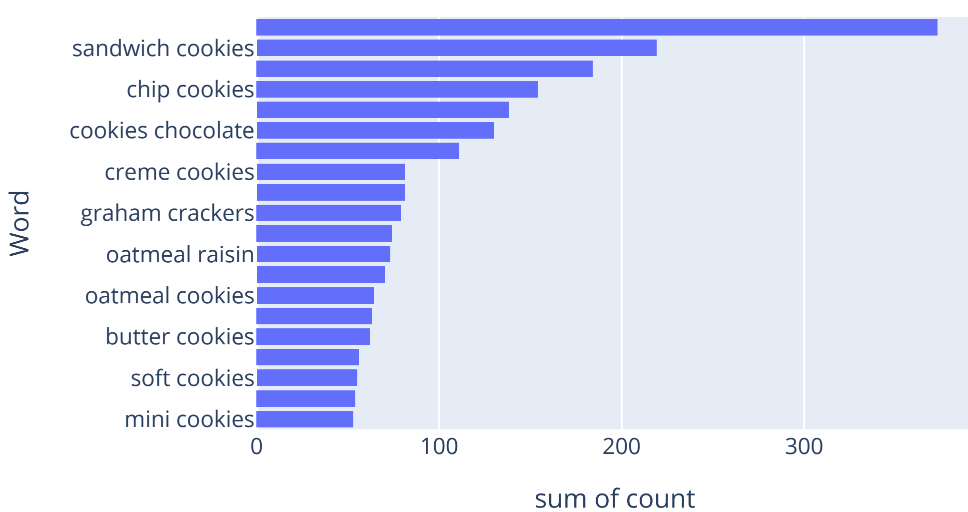


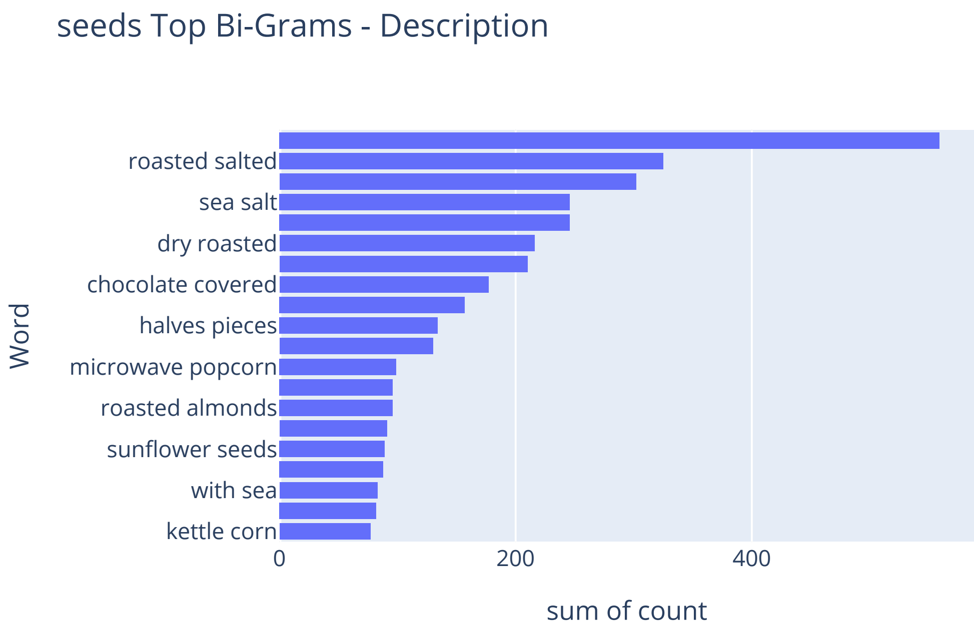
candy Top Bi-Grams - Description
peanut butter
hard candy
gummi bears
fruit flavored
candy corn
butter cups
fruit snacks
gummi worms
cotton candy
soft chewy
Word
0
50
100
150
200
250
sum of count

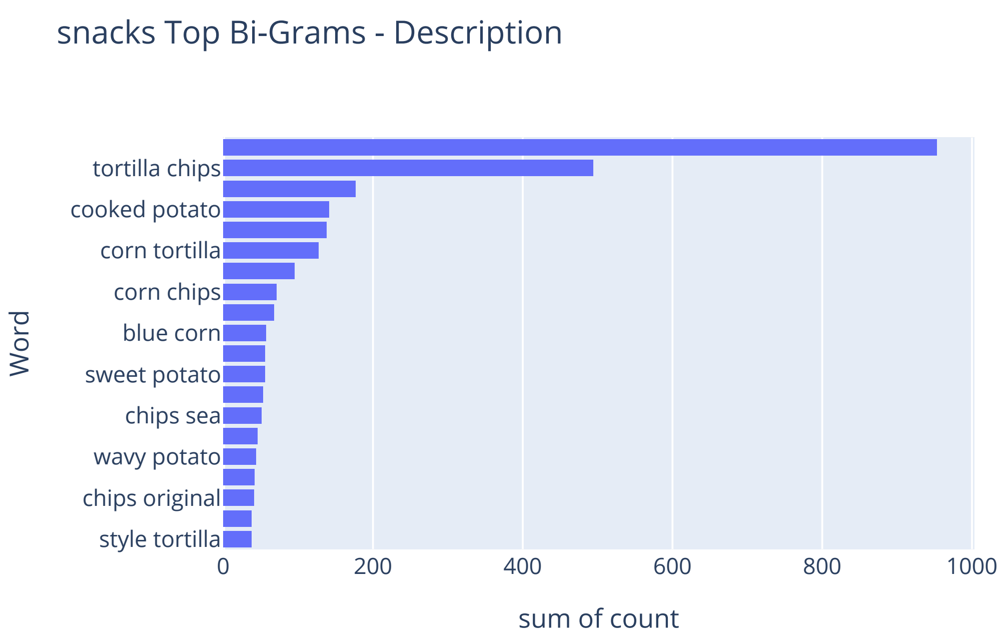
chocolate Top Bi-Grams - Description



cookies Top Bi-Grams - Description




seeds Top Bi-Grams - Description
roasted salted
sea salt
dry roasted
chocolate covered
halves pieces
microwave popcorn
roasted almonds
sunflower seeds
with sea
kettle corn
Word
0
200
400
sum of count


snacks Top Bi-Grams - Description
tortilla chips
cooked potato
corn tortilla
corn chips
blue corn
sweet potato
chips sea
wavy potato
chips original
style tortilla
Word
0
200
400
600
800
1000
sum of count

Now, let us move on to some analyses for the nutrients.

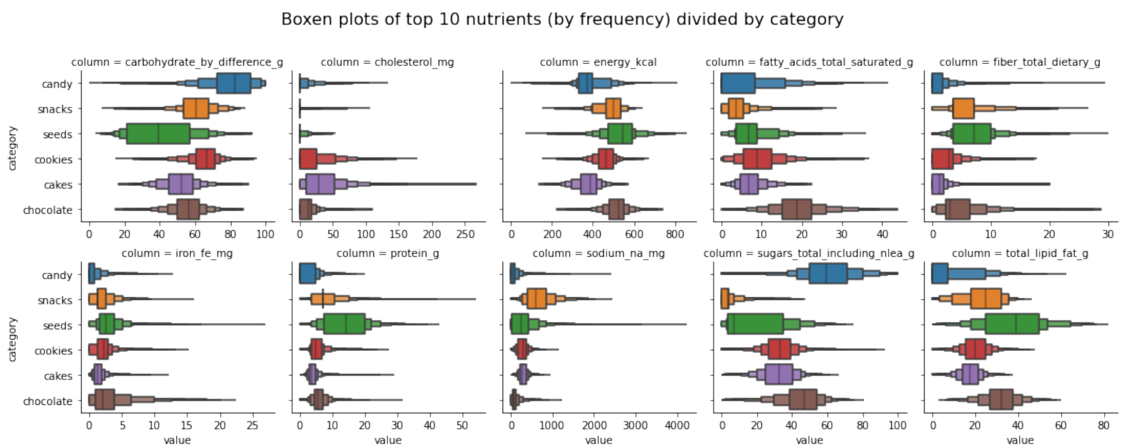
Let us first take the top 10 most common nutrients:

[ ]:
```python
top_10_nutrients = nutrients_column_values['name'].head(10)['value']
```

Some boxen plots by category:

[ ]:
```python
long = pd.melt(eda_df, id_vars=['idx', 'category', 'category_enc'], var_name='column')
long = long[long['column'].isin(top_10_nutrients)].reset_index()
boxen_plots = sns.catplot(data=long, kind='boxen', x='value', y='category', showfliers = False, col='column', col_wrap=5, height=3, sharex=False, aspect=1,
orient='h', margin_titles=True)
boxen_plots.fig.subplots_adjust(top=0.85)
boxen_plots.fig.suptitle('Boxen plots of top 10 nutrients (by frequency) divided by category', fontsize=16)
```

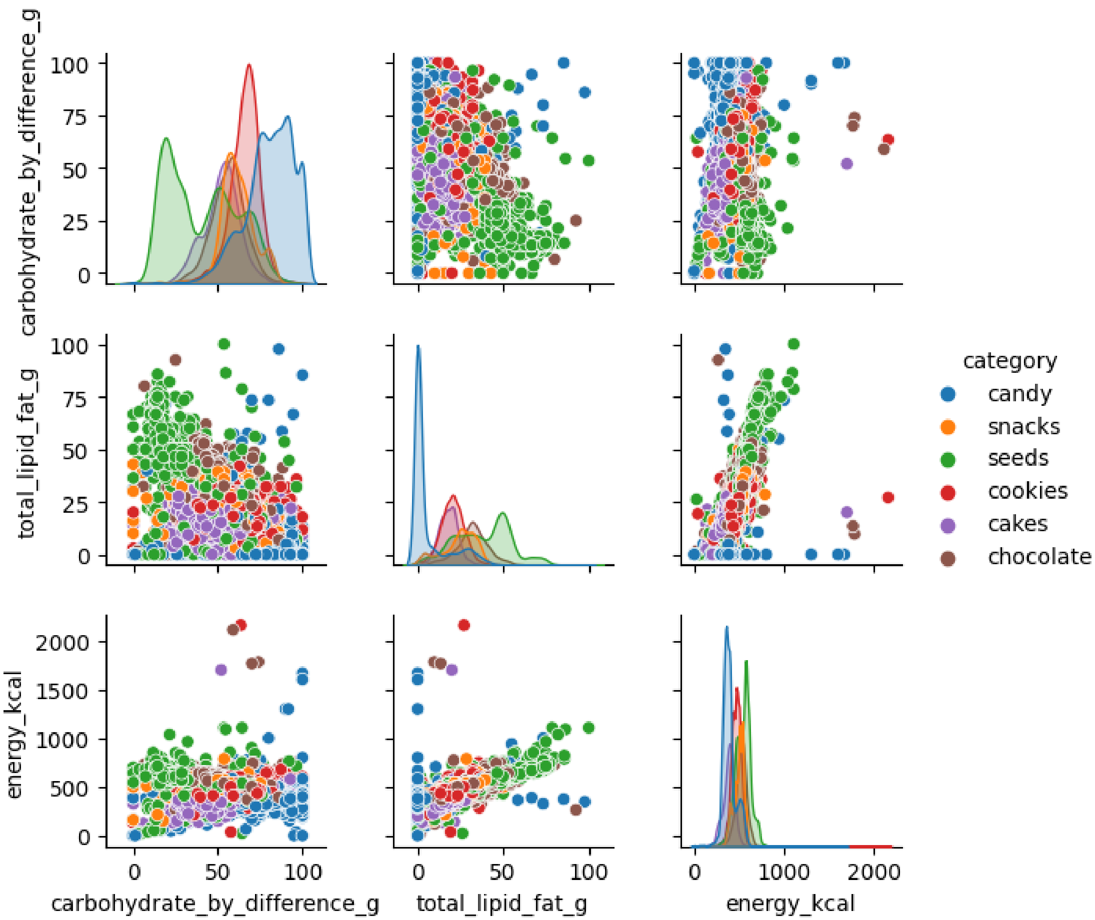
[ ]:
```
Text(0.5, 0.98, 'Boxen plots of top 10 nutrients (by frequency) divided by
category')
```



Pairwise plots by category for top 3 nutrients (to avoid a larger than necessary plot):

[ ]:
```python
eda_df_nutrients = eda_df[list(top_10_nutrients.head(3)) + ['category']]
sns.pairplot(eda_df_nutrients, dropna=True, height=2, aspect=1, hue='category')
```

[ ]:
```
<seaborn.axisgrid.PairGrid at 0x164ad56e940>
```



We will opt to use a nueral network for the tabular data, and then use its last layer to be added to the CNN which will classify the images to get a classifier which uses both types of data.

Let us add functions which would assist us in inputing the data to a model

[ ]:
```
def get_oh_categories(df, columns, sort=True):
    def sort_or_not(X):
        if not sort:
            return X
        else:
            return sorted(X)

    full_list=[]
    for column in columns:
        full_list.append(sort_or_not(df[column].unique()))
    return full_list


def get_oh_encoder(df):
```

```
    cat_columns = list(get_column_values(df).keys())
    categories = get_oh_categories(df, cat_columns)
    ohe = preprocessing.OneHotEncoder(categories=categories, sparse=False,␣
↪handle_unknown='ignore')
    return ohe, cat_columns


def get_oh_matrix(df):
    ohe, cat_columns = get_oh_encoder(df)
    if cat_columns:
        enc = ohe.fit_transform(df[cat_columns])
        numerical = df.select_dtypes(include=[np.number])
        result = np.concatenate([enc, numerical], axis=1)
        return result
    return df


def make_param_grids(steps, param_grids):
    final_params=[]
    # Itertools.product will do a permutation such that
    # (pca OR svd) AND (svm OR rf) will become ->
    # (pca, svm) , (pca, rf) , (svd, svm) , (svd, rf)
    for estimator_names in itertools.product(*steps.values()):
        current_grid = {}
        # Step_name and estimator_name should correspond
        # i.e preprocessor must be from pca and select.
        for step_name, estimator_name in zip(steps.keys(), estimator_names):
            for param, value in param_grids.get(estimator_name).items():
                if param == 'object':
                    # Set actual estimator in pipeline
                    current_grid[step_name]=[value]
                else:
                    # Set parameters corresponding to above estimator
                    current_grid[step_name+'__'+param]=value
        #Append this dictionary to final params
        final_params.append(current_grid)
    return final_params


def create_features_from_strings(df, column, strings):
    if type(strings) == str:
        strings = [strings]
    for string in strings:
        mask = df[column].str.contains(string)
        new_column = (mask * 1).to_frame().rename(columns={column: '{}_{}'.
↪format(column, string.replace(' ', '_'))}) # Doing it like this because␣
↪apperantly
```

```
        df = pd.concat([df, new_column], axis=1)                              ␣
↪                                    # otherwise performance is hurt
    return df


def fix_brands(df):
    brands_regex_dict = {
    'walmart': ['walmart'],
    'target': ['target'],
    'ferrara': ['ferrara'],
    'whole foods': ['whole foods'],
    'safeway': ['safeway'],
    'walgreens': ['walgreens'],
    'mars chocolate': ['mars chocolate'],
    'lindt': ['lindt'],
    'star snacks': ['star snacks'],
    'north star': ['north star'],
    'weis': ['weis'],
    'harristeeter': ['harristeeter', 'harris teeter'],
    'other': ['not a branded item']
    }
    for string in brands_regex_dict:
        mask = df['brand'].str.contains('|'.join(brands_regex_dict[string]))
        df.loc[mask, 'brand'] = string
    return df


def fe(df, feature_strings_dict, drop_brand=True):
    for column, dict in feature_strings_dict.items():
        df = create_features_from_strings(df, column, dict)
    df = df.drop(['description', 'ingredients', 'serving_size_unit',␣
↪'household_serving_fulltext'], axis=1)
    if drop_brand:
      df = df.drop(['brand'], axis=1)
    return df
```

First of all, let us remove all columns with more than 80% nulls

[ ]:
```
nulls = X.isnull().sum().to_frame().reset_index().rename(columns={'index':␣
↪'column', 0:'nulls'})
nulls['pc'] = nulls['nulls'] / len(X) * 100
columns_to_drop = nulls[nulls['pc'] >= 80]['column'].to_list()
```

So we have 25.4K records in our training set;

0.05 of them to find the best way to fill missing data

0.3 for feature engineering, type of model, and 0.35 model tuning

```
[ ]: columns_to_drop = columns_to_drop + ['category', 'category_enc']
X = clean_text_values(dataset_w_nutrients.drop(columns_to_drop, axis=1)).
    ↪set_index('idx')
y = dataset_w_nutrients[['idx', 'category_enc']].
    ↪set_index('idx')['category_enc']
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,␣
    ↪test_size=0.2, random_state=1337, stratify=y)

# 0.05 for missing data imputation
X_rest, X_rest2, y_rest, y_rest2 = model_selection.train_test_split(X_train,␣
    ↪y_train, test_size=0.1, random_state=1337, stratify=y_train)
X_mdi, X_coc, y_mdi, y_coc = model_selection.train_test_split(X_rest2, y_rest2,␣
    ↪test_size=0.5, random_state=1337, stratify=y_rest2)

# 0.3 for feature engineering, resampling, and 0.35 model tuning
X_rest, X_fe, y_rest, y_fe = model_selection.train_test_split(X_rest, y_rest,␣
    ↪test_size=1/3, random_state=1337, stratify=y_rest)
X_rs, X_mt, y_rs, y_mt = model_selection.train_test_split(X_rest, y_rest,␣
    ↪test_size=0.5, random_state=1337, stratify=y_rest)
X_mt, y_mt = pd.concat([X_mt, X_coc], axis=0), pd.concat([y_mt, y_coc], axis=0)
```

Initial feature engineering

First, let's clean the text columns.

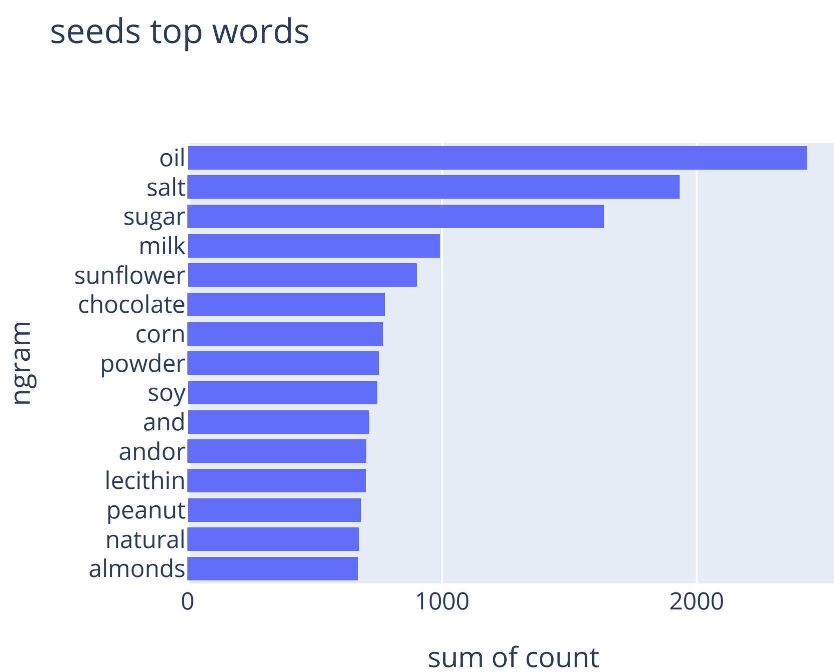
We saw in the EDA that there are some sentences in the ingredients column, as well as numbers.

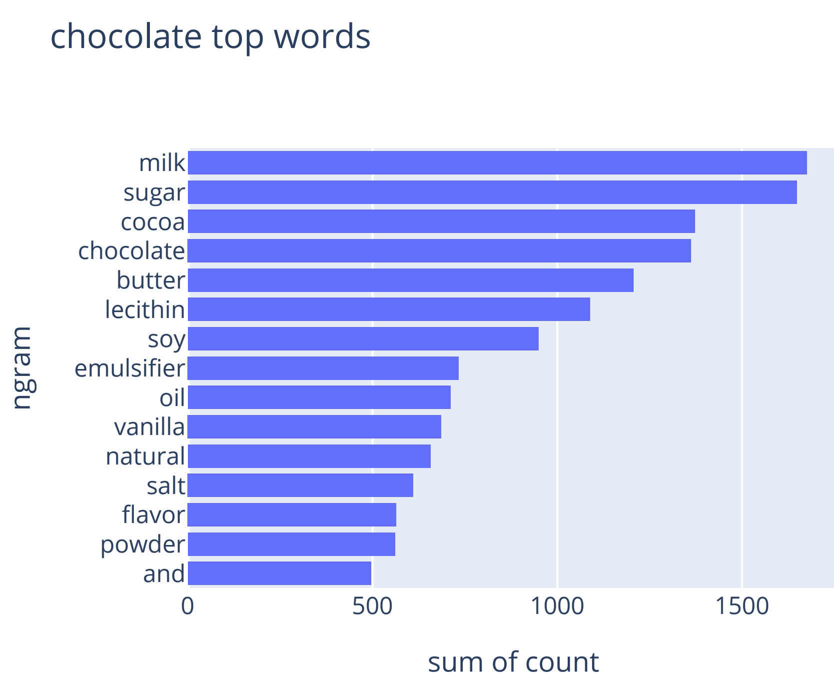
Let us look for those sentences again:

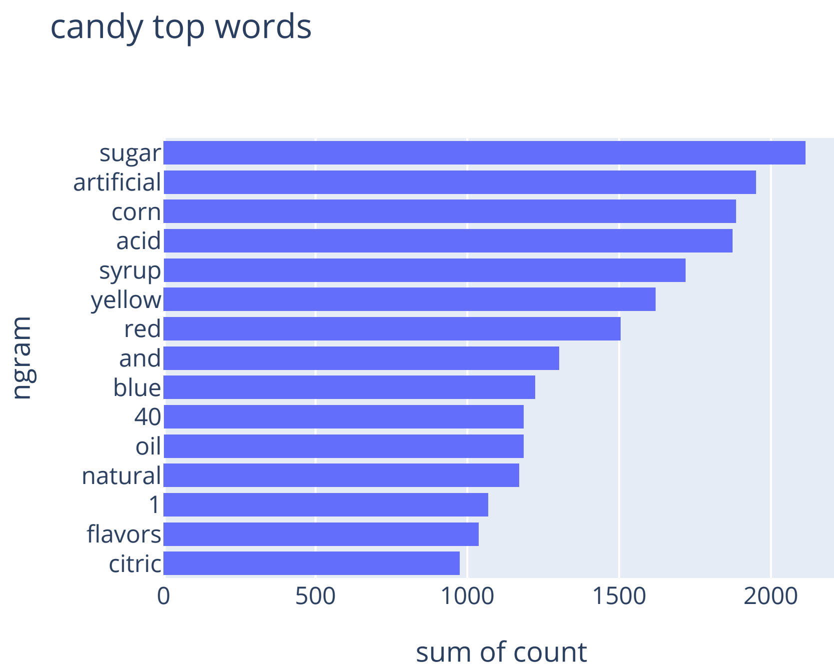
Let us extract a list of most common ingredients from each category to look for fishy strings and get some ideas for features:

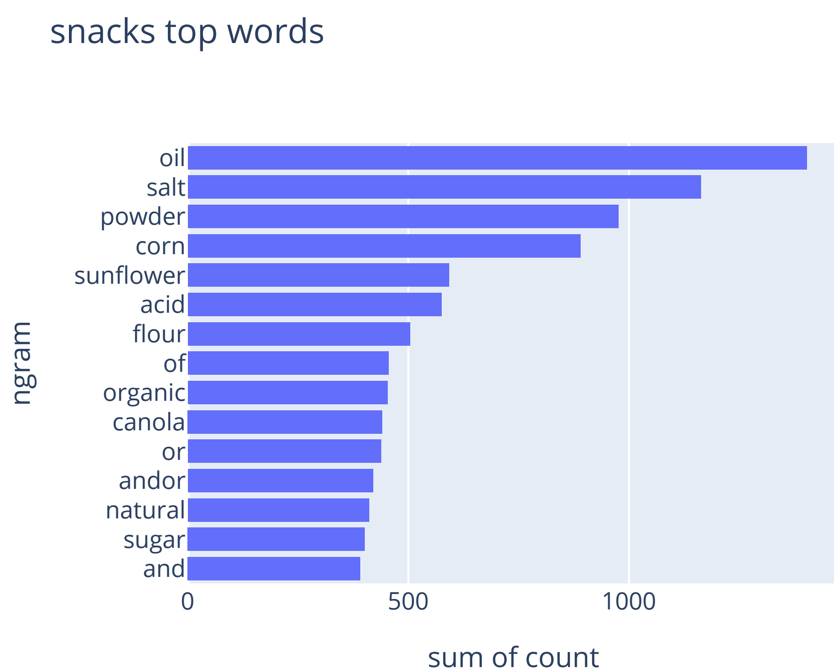
```
[ ]: X_fe_with_cat = pd.concat([X_fe.reset_index(), pd.Series(le.
    ↪inverse_transform(y_fe))], axis=1).rename(columns={0: 'category'})
```

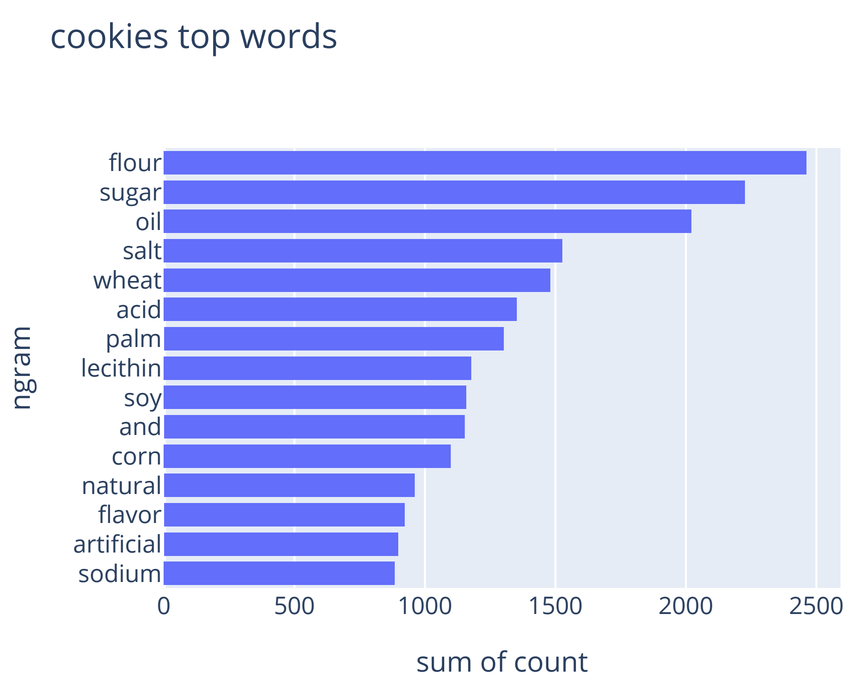
```
[ ]: X_fe_with_cat, top_15_words = get_ngrams_top_k(X_fe_with_cat, 'ingredients', 1,␣
    ↪15, True)
for cat, top in top_15_words.items():
    px.histogram(top, orientation='h', y='ngram', x='count', title='{} top␣
    ↪words'.format(cat), height=400, width=500).show()
```

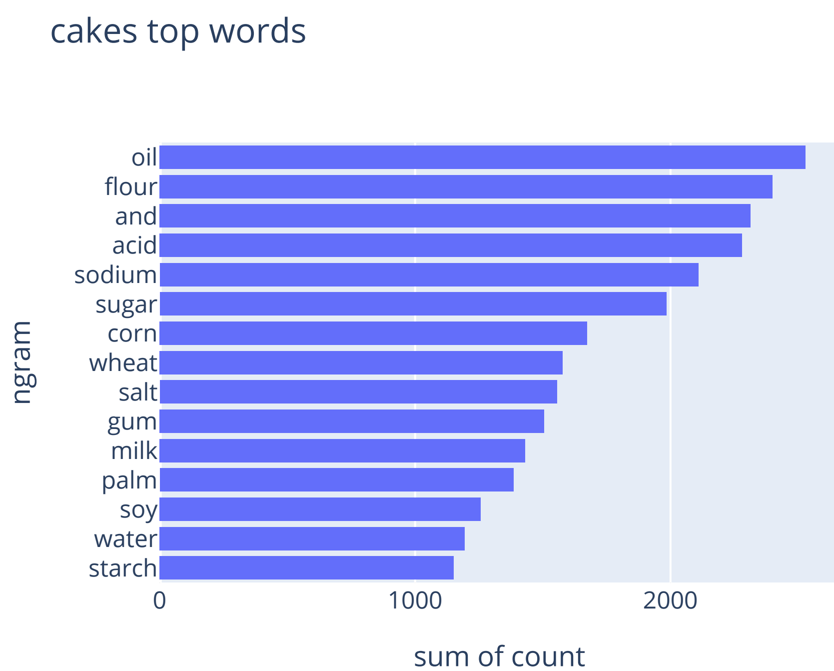

seeds top words
oil
salt
sugar
milk
sunflower
chocolate
corn
powder
soy
and
andor
lecithin
peanut
natural
almonds
ngram
0
1000
2000
sum of count


chocolate top words
milk
sugar
cocoa
chocolate
butter
lecithin
soy
emulsifier
oil
vanilla
natural
salt
flavor
powder
and
ngram
0
500
1000
1500
sum of count


candy top words
sugar
artificial
corn
acid
syrup
yellow
red
and
blue
40
oil
natural
1
flavors
citric
ngram
0
500
1000
1500
2000
sum of count

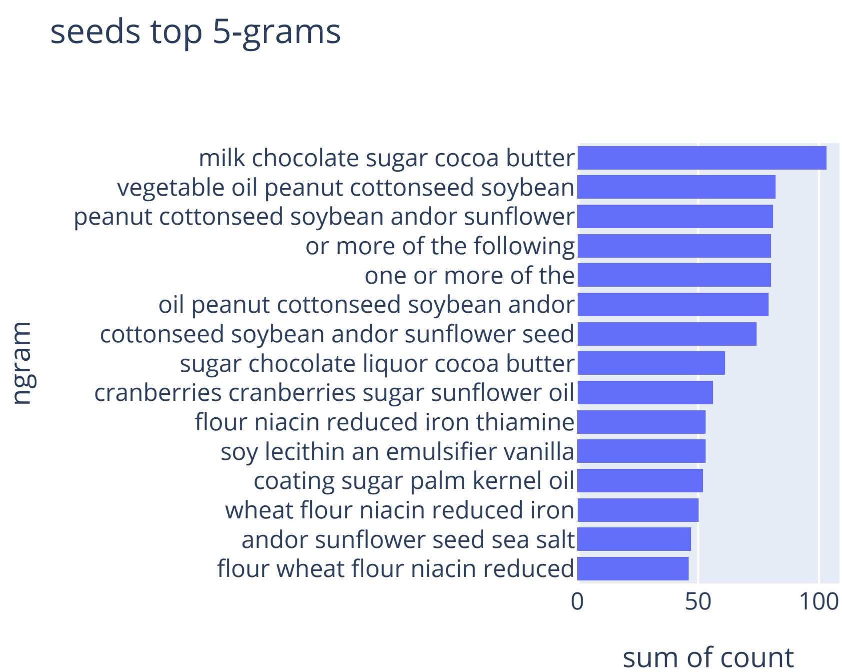

snacks top words
oil
salt
powder
corn
sunflower
acid
flour
of
organic
canola
or
andor
natural
sugar
and
ngram
0
500
1000
sum of count

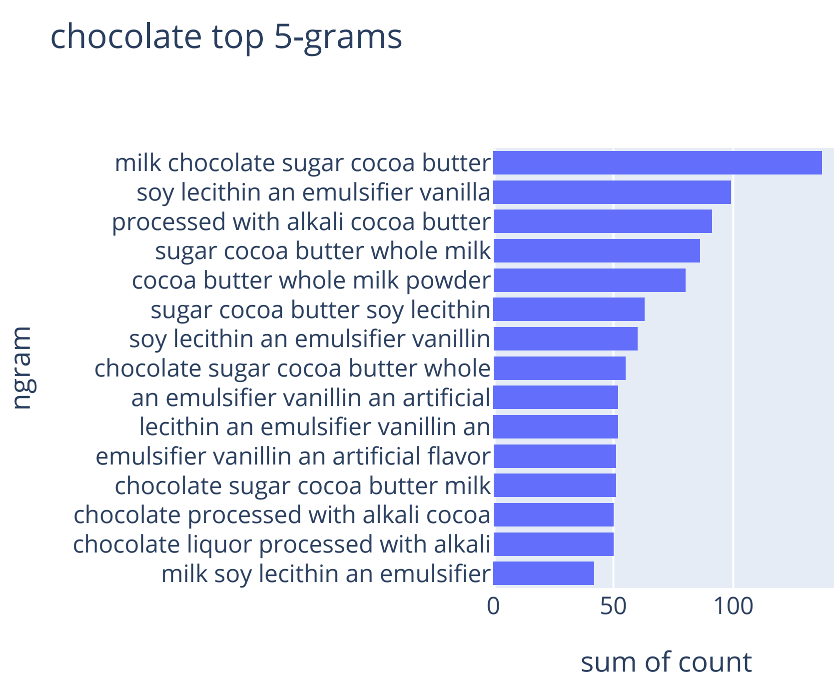

cookies top words
flour
sugar
oil
salt
wheat
acid
palm
lecithin
soy
and
corn
natural
flavor
artificial
sodium
ngram
0
500
1000
1500
2000
2500
sum of count

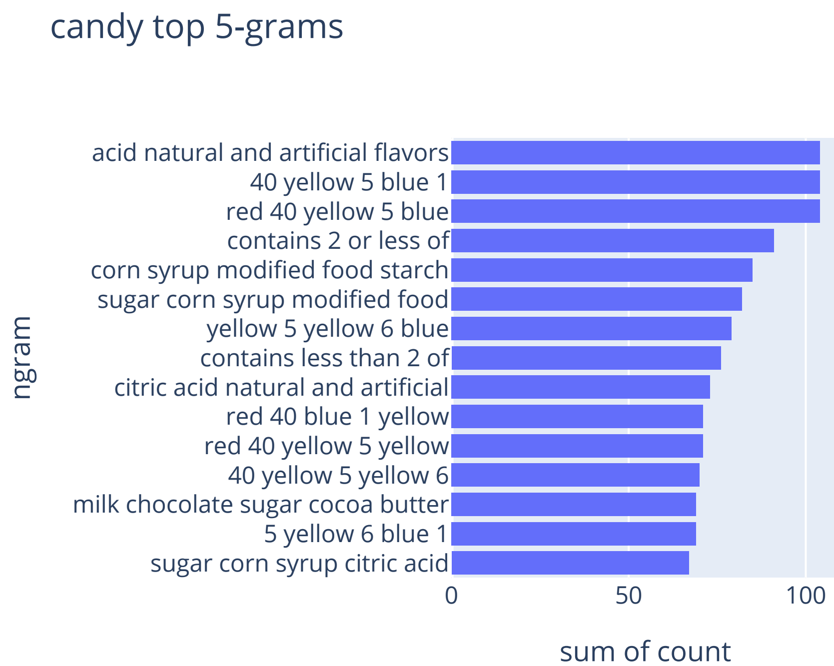


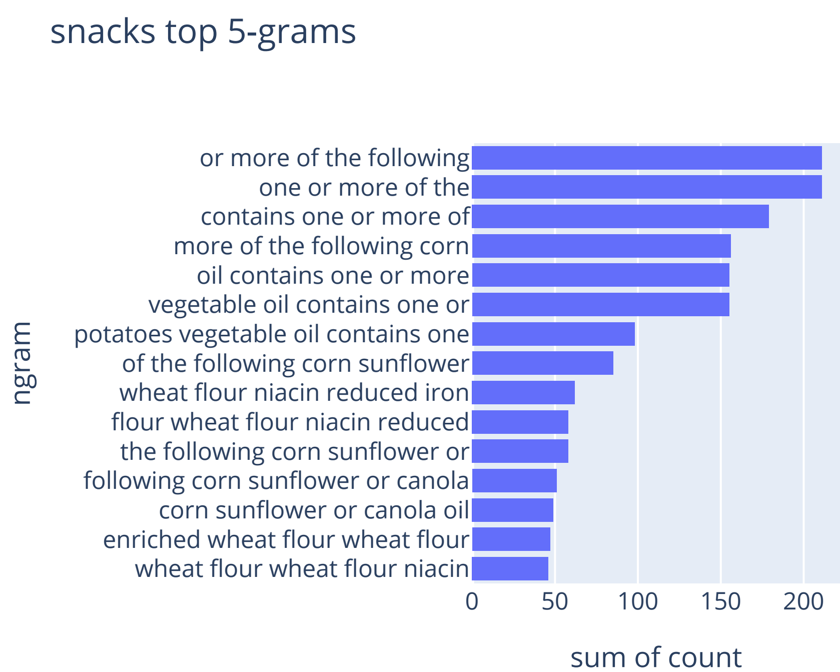
Let's do the same for 5-grams to identify common sentences:

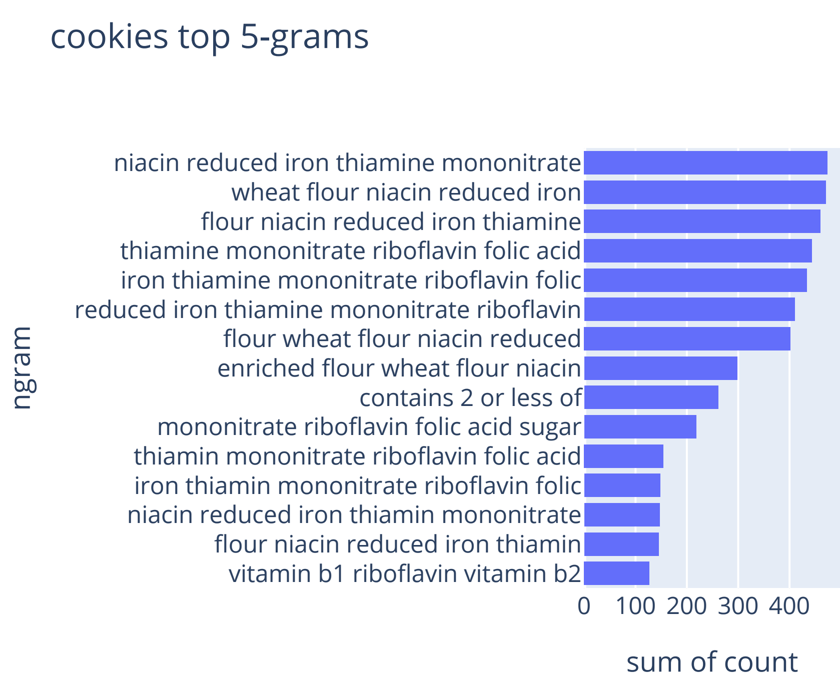
```
[ ]: X_fe_with_cat, top_15_5grams = get_ngrams_top_k(X_fe_with_cat, 'ingredients',␣
      ↪5, 15, True)
     for cat, top in top_15_5grams.items():
         px.histogram(top, orientation='h', y='ngram', x='count', title='{} top␣
      ↪5-grams'.format(cat), height=400, width=500).show()
```

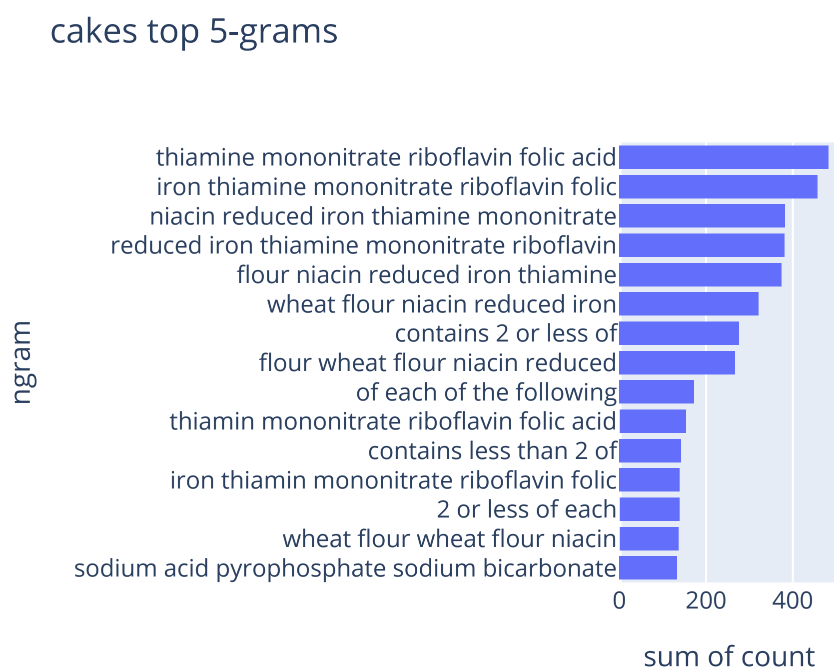

seeds top 5-grams
milk chocolate sugar cocoa butter
vegetable oil peanut cottonseed soybean
peanut cottonseed soybean andor sunflower
or more of the following
one or more of the
oil peanut cottonseed soybean andor
cottonseed soybean andor sunflower seed
sugar chocolate liquor cocoa butter
cranberries cranberries sugar sunflower oil
flour niacin reduced iron thiamine
soy lecithin an emulsifier vanilla
coating sugar palm kernel oil
wheat flour niacin reduced iron
andor sunflower seed sea salt
flour wheat flour niacin reduced
ngram
0
50
100
sum of count


chocolate top 5-grams
milk chocolate sugar cocoa butter
soy lecithin an emulsifier vanilla
processed with alkali cocoa butter
sugar cocoa butter whole milk
cocoa butter whole milk powder
sugar cocoa butter soy lecithin
soy lecithin an emulsifier vanillin
chocolate sugar cocoa butter whole
an emulsifier vanillin an artificial
lecithin an emulsifier vanillin an
emulsifier vanillin an artificial flavor
chocolate sugar cocoa butter milk
chocolate processed with alkali cocoa
chocolate liquor processed with alkali
milk soy lecithin an emulsifier
ngram
0
50
100
sum of count


candy top 5-grams
acid natural and artificial flavors
40 yellow 5 blue 1
red 40 yellow 5 blue
contains 2 or less of
corn syrup modified food starch
sugar corn syrup modified food
yellow 5 yellow 6 blue
contains less than 2 of
citric acid natural and artificial
red 40 blue 1 yellow
red 40 yellow 5 yellow
40 yellow 5 yellow 6
milk chocolate sugar cocoa butter
5 yellow 6 blue 1
sugar corn syrup citric acid
ngram
0
50
100
sum of count


snacks top 5-grams
ngram
or more of the following
one or more of the
contains one or more of
more of the following corn
oil contains one or more
vegetable oil contains one or
potatoes vegetable oil contains one
of the following corn sunflower
wheat flour niacin reduced iron
flour wheat flour niacin reduced
the following corn sunflower or
following corn sunflower or canola
corn sunflower or canola oil
enriched wheat flour wheat flour
wheat flour wheat flour niacin
0
50
100
150
200
sum of count


cookies top 5-grams
ngram
niacin reduced iron thiamine mononitrate
wheat flour niacin reduced iron
flour niacin reduced iron thiamine
thiamine mononitrate riboflavin folic acid
iron thiamine mononitrate riboflavin folic
reduced iron thiamine mononitrate riboflavin
flour wheat flour niacin reduced
enriched flour wheat flour niacin
contains 2 or less of
mononitrate riboflavin folic acid sugar
thiamin mononitrate riboflavin folic acid
iron thiamin mononitrate riboflavin folic
niacin reduced iron thiamin mononitrate
flour niacin reduced iron thiamin
vitamin b1 riboflavin vitamin b2
0
100
200
300
400
sum of count

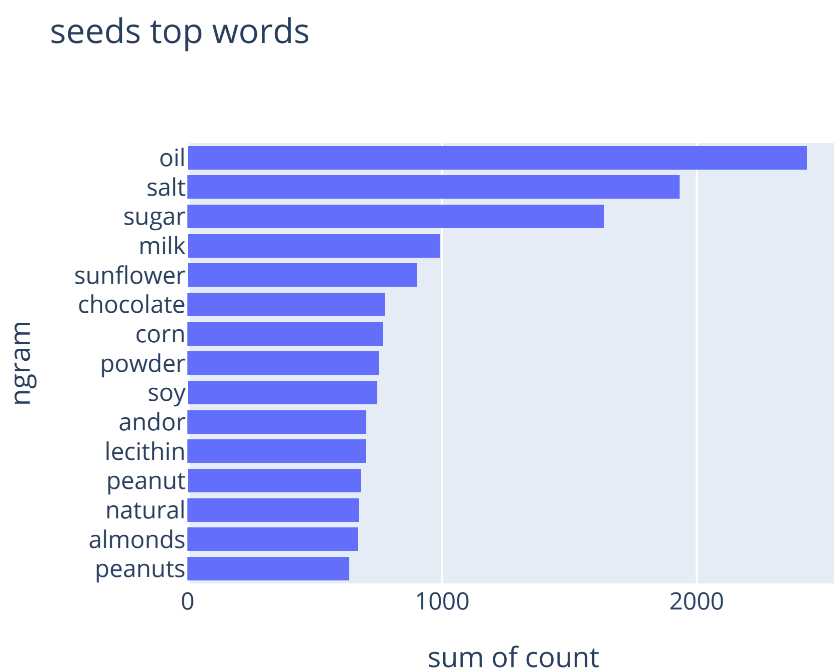


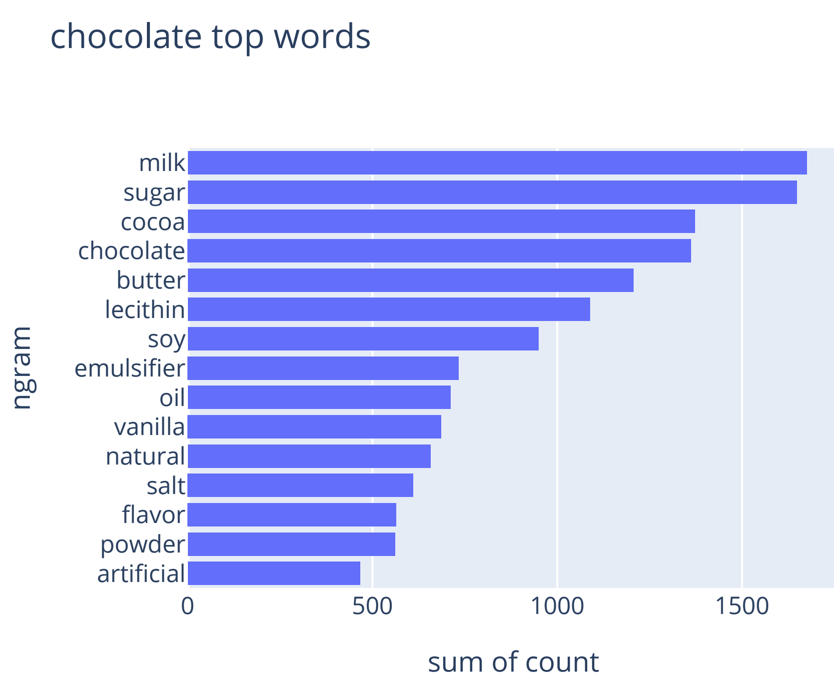
We can see that we have some bad words like ‘or’, ‘and’ etc (also numbers). Let’s fix that:

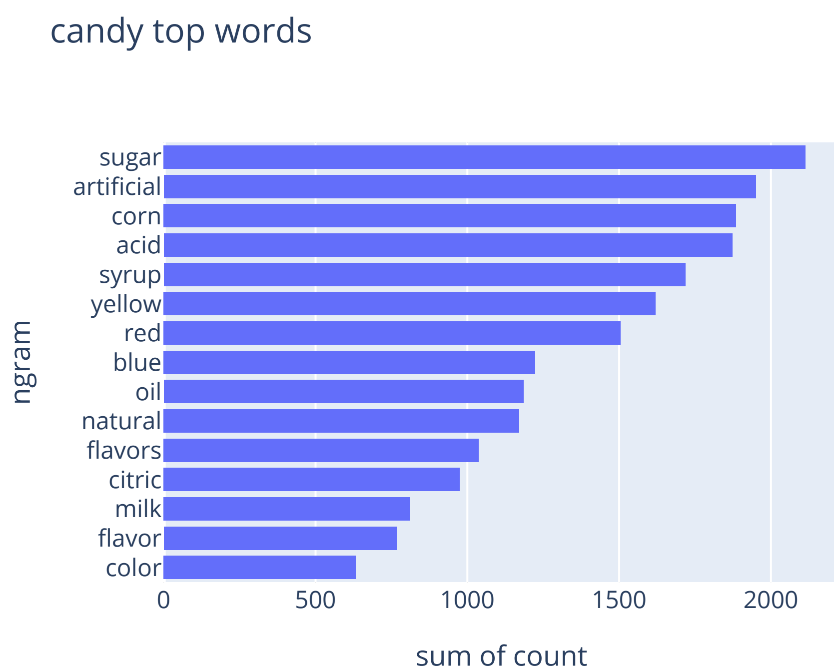
```
[ ]: irrelevant_strings = ['and', 'contains', 'one', 'or', 'more', 'less', 'than',␣
     ↪'of', 'a', 'the', 'following'] + [i for i in range(0, 1000)]
     for word in irrelevant_strings:
         X_fe['ingredients'] = X_fe['ingredients'].str.replace(" {} ".format(word),␣
     ↪" ")
         X_fe_with_cat['ingredients'] = X_fe_with_cat['ingredients'].str.replace("␣
     ↪{} ".format(word), " ")
```

Let’s see the new top words and 5-grams:

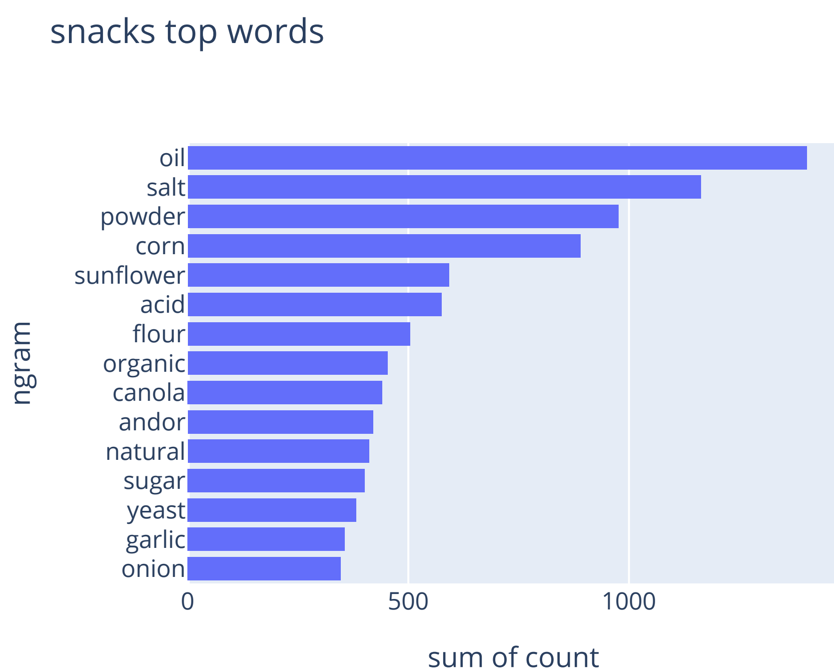
```
[ ]: X_fe_with_cat, top_15_words = get_ngrams_top_k(X_fe_with_cat, 'ingredients', 1,␣
     ↪15, True)
     for cat, top in top_15_words.items():
         px.histogram(top, orientation='h', y='ngram', x='count', title='{} top␣
     ↪words'.format(cat), height=400, width=500).show()
```


seeds top words
oil
salt
sugar
milk
sunflower
chocolate
corn
powder
soy
andor
lecithin
peanut
natural
almonds
peanuts
ngram
0
1000
2000
sum of count

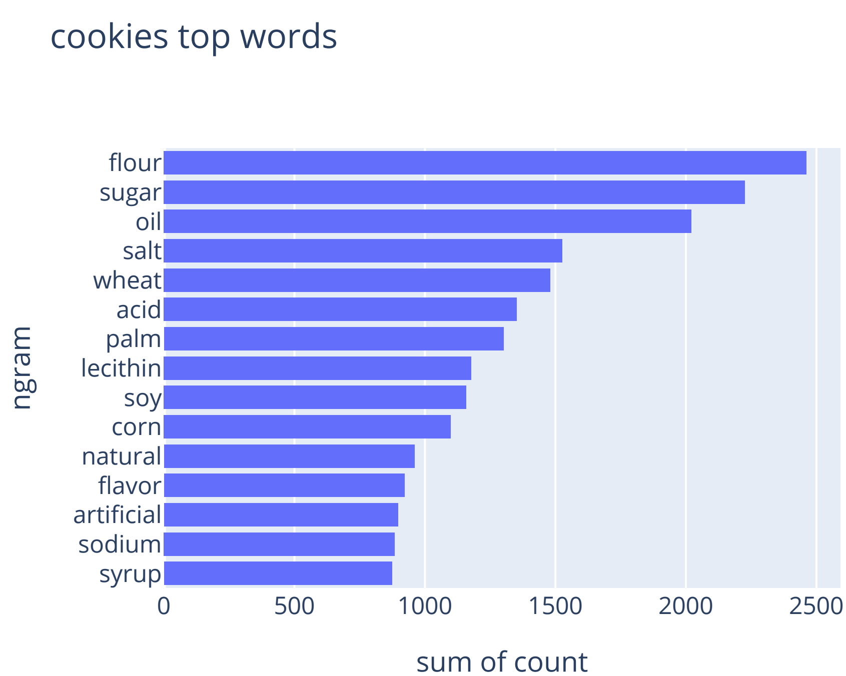

chocolate top words
milk
sugar
cocoa
chocolate
butter
lecithin
soy
emulsifier
oil
vanilla
natural
salt
flavor
powder
artificial
ngram
0
500
1000
1500
sum of count



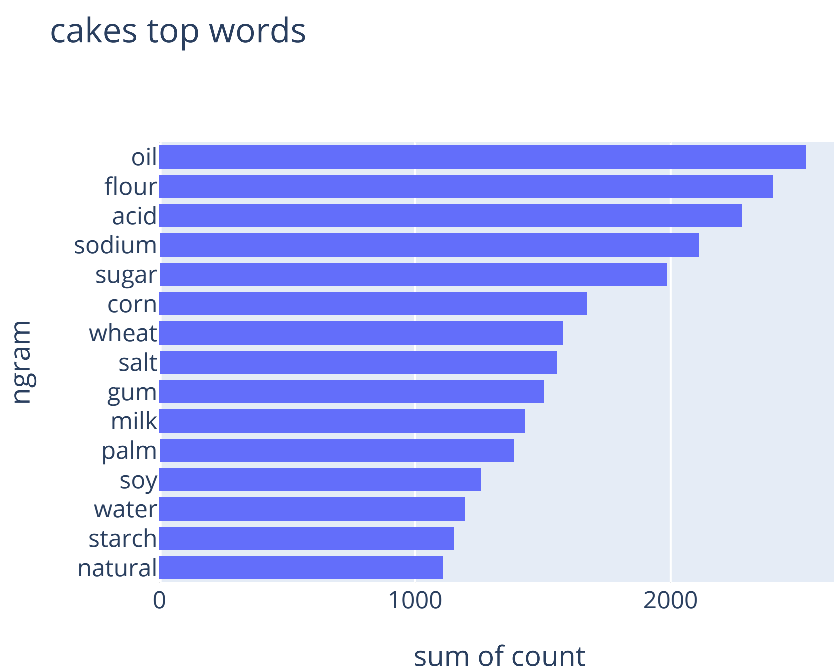
candy top words
sugar
artificial
corn
acid
syrup
yellow
red
blue
oil
natural
flavors
citric
milk
flavor
color
ngram
0
500
1000
1500
2000
sum of count



snacks top words
oil
salt
powder
corn
sunflower
acid
flour
organic
canola
andor
natural
sugar
yeast
garlic
onion
ngram
0
500
1000
sum of count

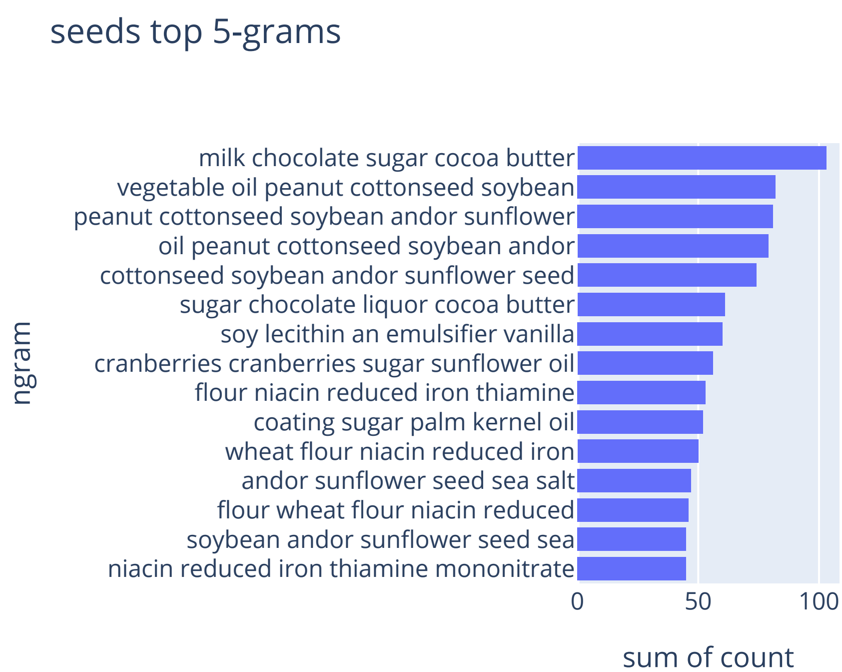


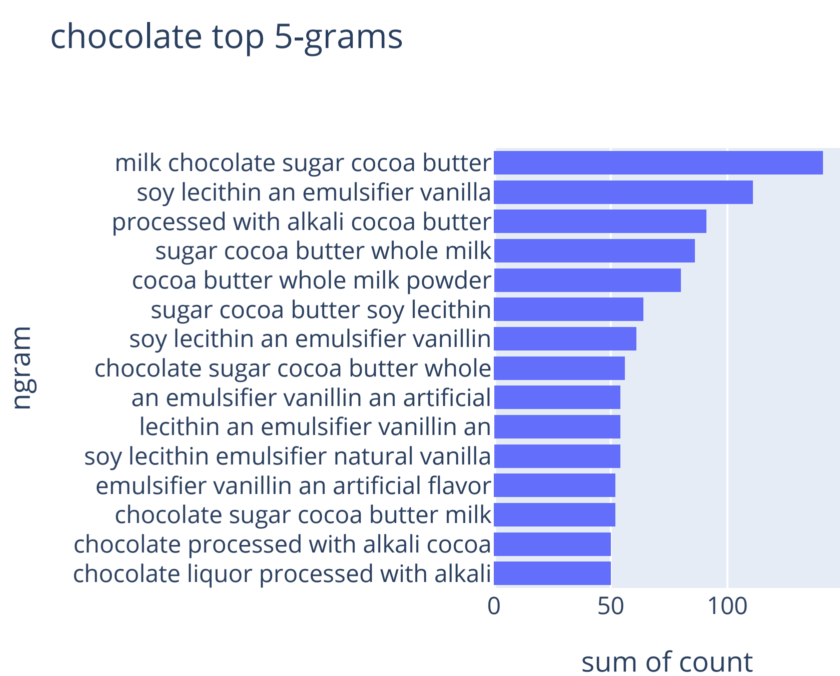
cookies top words
flour
sugar
oil
salt
wheat
acid
palm
lecithin
soy
corn
natural
flavor
artificial
sodium
syrup
ngram
0
500
1000
1500
2000
2500
sum of count

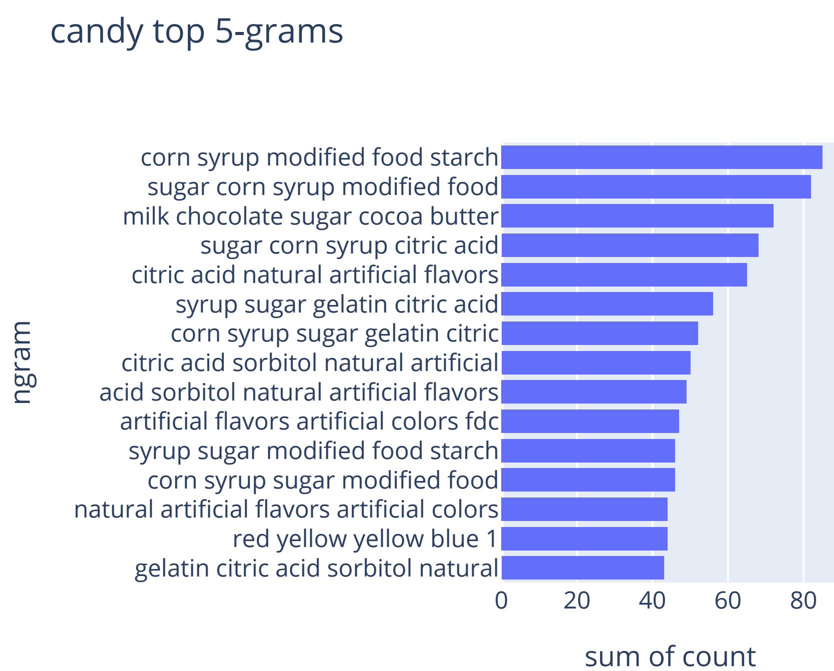


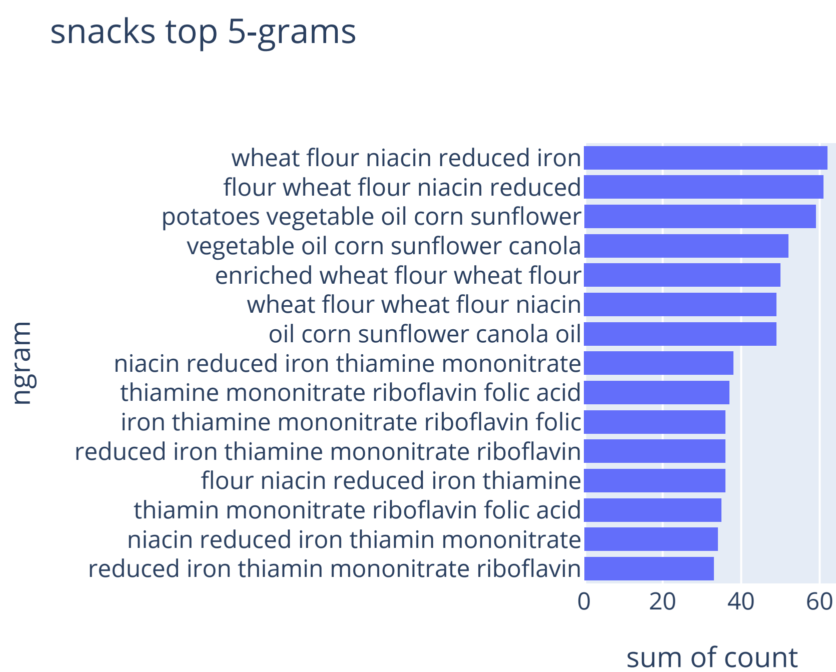
[ ]:

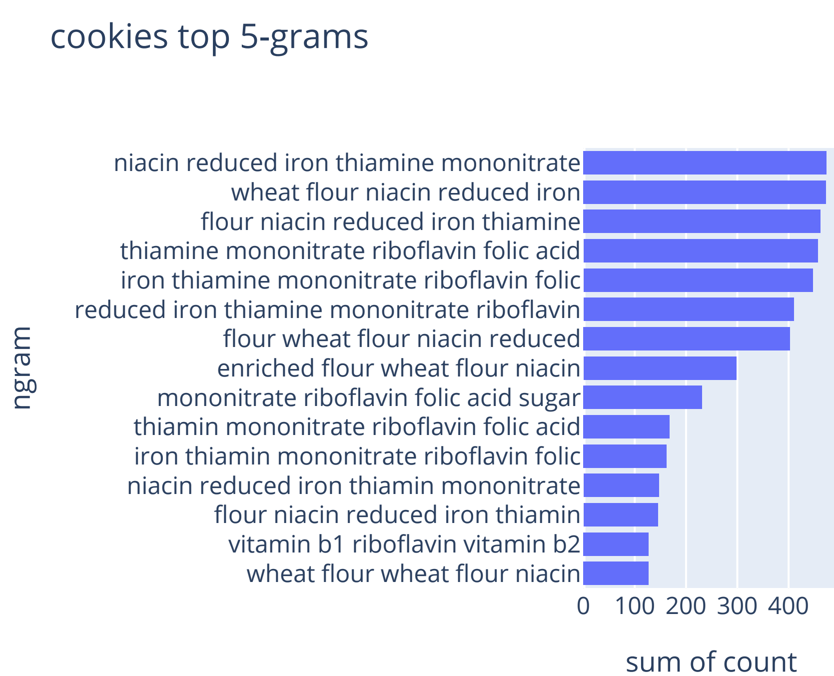
```
X_fe_with_cat, top_15_5grams = get_ngrams_top_k(X_fe_with_cat, 'ingredients',␣
↪5, 15, True)
for cat, top in top_15_5grams.items():
    px.histogram(top, orientation='h', y='ngram', x='count', title='{} top␣
↪5-grams'.format(cat), height=400, width=500).show()
```

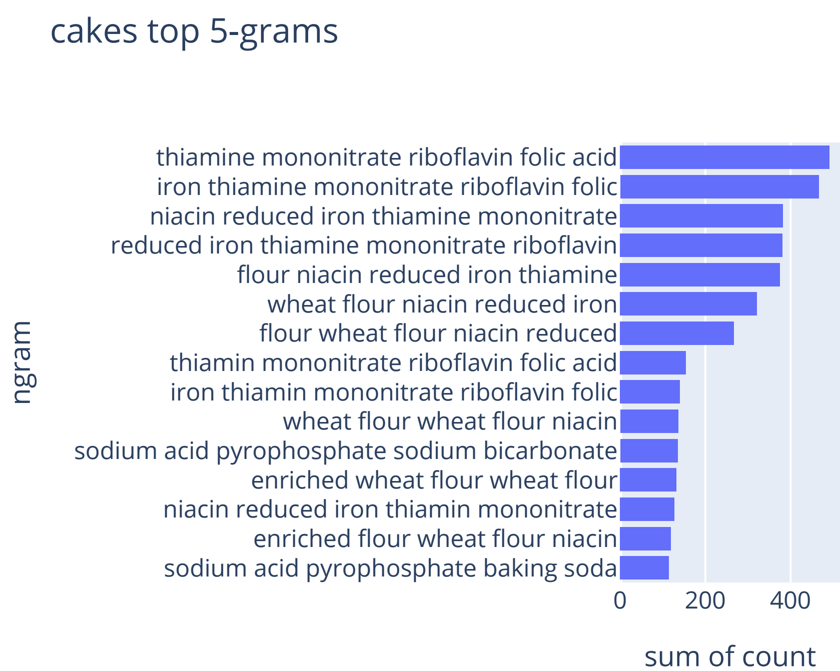

seeds top 5-grams
ngram
milk chocolate sugar cocoa butter
vegetable oil peanut cottonseed soybean
peanut cottonseed soybean andor sunflower
oil peanut cottonseed soybean andor
cottonseed soybean andor sunflower seed
sugar chocolate liquor cocoa butter
soy lecithin an emulsifier vanilla
cranberries cranberries sugar sunflower oil
flour niacin reduced iron thiamine
coating sugar palm kernel oil
wheat flour niacin reduced iron
andor sunflower seed sea salt
flour wheat flour niacin reduced
soybean andor sunflower seed sea
niacin reduced iron thiamine mononitrate
0
50
100
sum of count


chocolate top 5-grams
ngram
milk chocolate sugar cocoa butter
soy lecithin an emulsifier vanilla
processed with alkali cocoa butter
sugar cocoa butter whole milk
cocoa butter whole milk powder
sugar cocoa butter soy lecithin
soy lecithin an emulsifier vanillin
chocolate sugar cocoa butter whole
an emulsifier vanillin an artificial
lecithin an emulsifier vanillin an
soy lecithin emulsifier natural vanilla
emulsifier vanillin an artificial flavor
chocolate sugar cocoa butter milk
chocolate processed with alkali cocoa
chocolate liquor processed with alkali
0
50
100
sum of count


candy top 5-grams
corn syrup modified food starch
sugar corn syrup modified food
milk chocolate sugar cocoa butter
sugar corn syrup citric acid
citric acid natural artificial flavors
syrup sugar gelatin citric acid
corn syrup sugar gelatin citric
citric acid sorbitol natural artificial
acid sorbitol natural artificial flavors
artificial flavors artificial colors fdc
syrup sugar modified food starch
corn syrup sugar modified food
natural artificial flavors artificial colors
red yellow yellow blue 1
gelatin citric acid sorbitol natural
ngram
0
20
40
60
80
sum of count


snacks top 5-grams
wheat flour niacin reduced iron
flour wheat flour niacin reduced
potatoes vegetable oil corn sunflower
vegetable oil corn sunflower canola
enriched wheat flour wheat flour
wheat flour wheat flour niacin
oil corn sunflower canola oil
niacin reduced iron thiamine mononitrate
thiamine mononitrate riboflavin folic acid
iron thiamine mononitrate riboflavin folic
reduced iron thiamine mononitrate riboflavin
flour niacin reduced iron thiamine
thiamin mononitrate riboflavin folic acid
niacin reduced iron thiamin mononitrate
reduced iron thiamin mononitrate riboflavin
ngram
0
20
40
60
sum of count


cookies top 5-grams
niacin reduced iron thiamine mononitrate
wheat flour niacin reduced iron
flour niacin reduced iron thiamine
thiamine mononitrate riboflavin folic acid
iron thiamine mononitrate riboflavin folic
reduced iron thiamine mononitrate riboflavin
flour wheat flour niacin reduced
enriched flour wheat flour niacin
mononitrate riboflavin folic acid sugar
thiamin mononitrate riboflavin folic acid
iron thiamin mononitrate riboflavin folic
niacin reduced iron thiamin mononitrate
flour niacin reduced iron thiamin
vitamin b1 riboflavin vitamin b2
wheat flour wheat flour niacin
ngram
0
100
200
300
400
sum of count



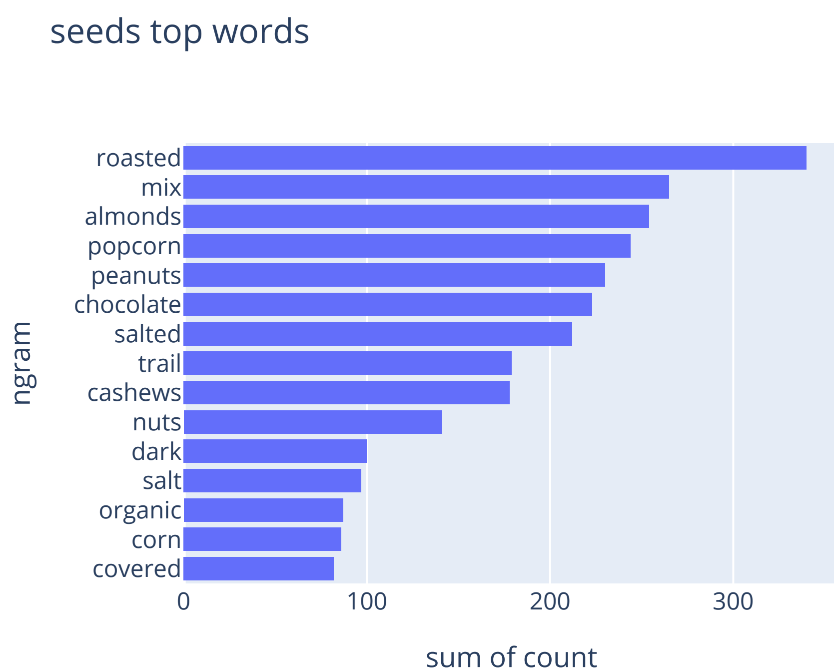
Numbers and sentences eliminated!

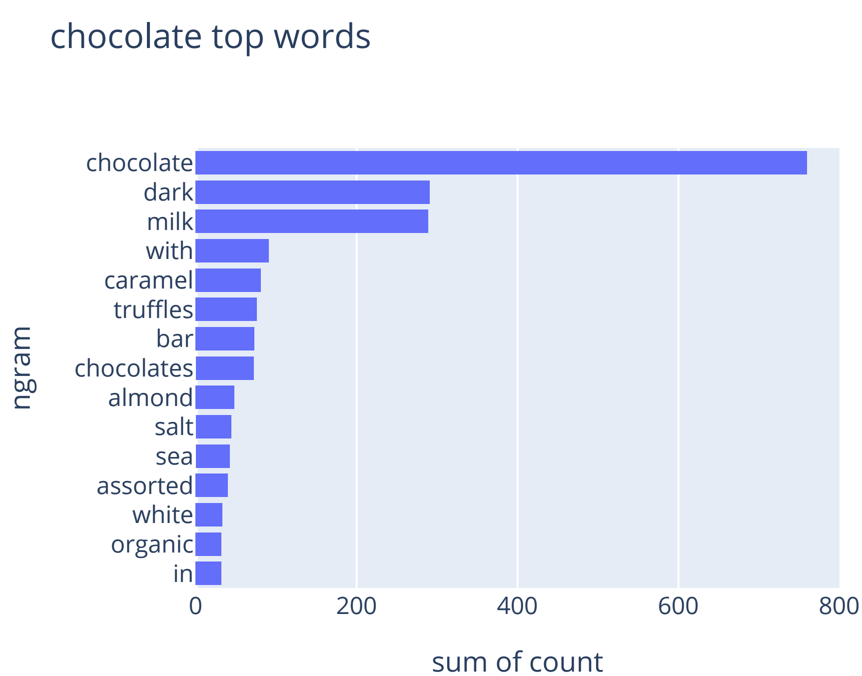
Let's move on to create features from the ingredients:

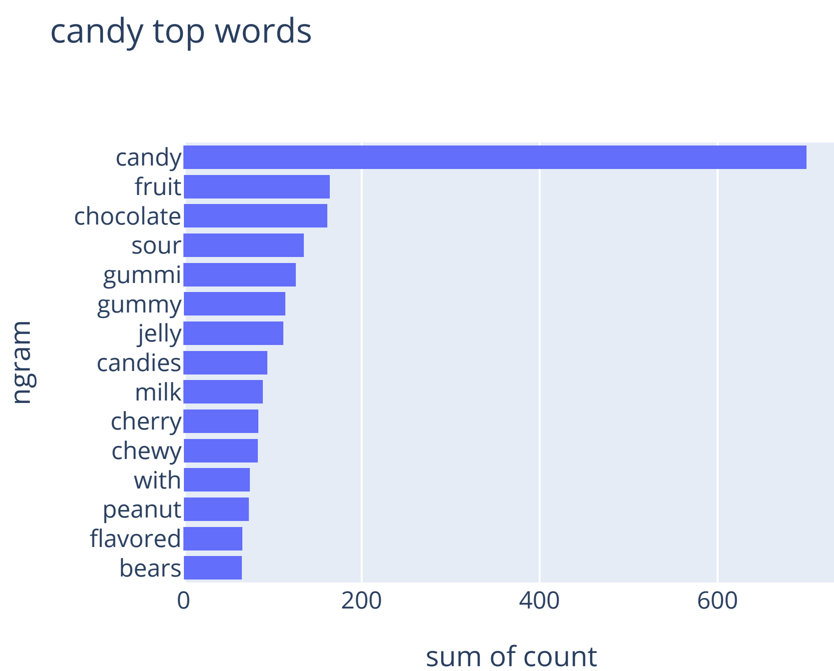
```
[ ]: ingredients_strings = np.unique(np.concatenate([top_15_words[cat]['ngram'].
      ↪values for cat in top_15_words.keys()], axis=0))
```

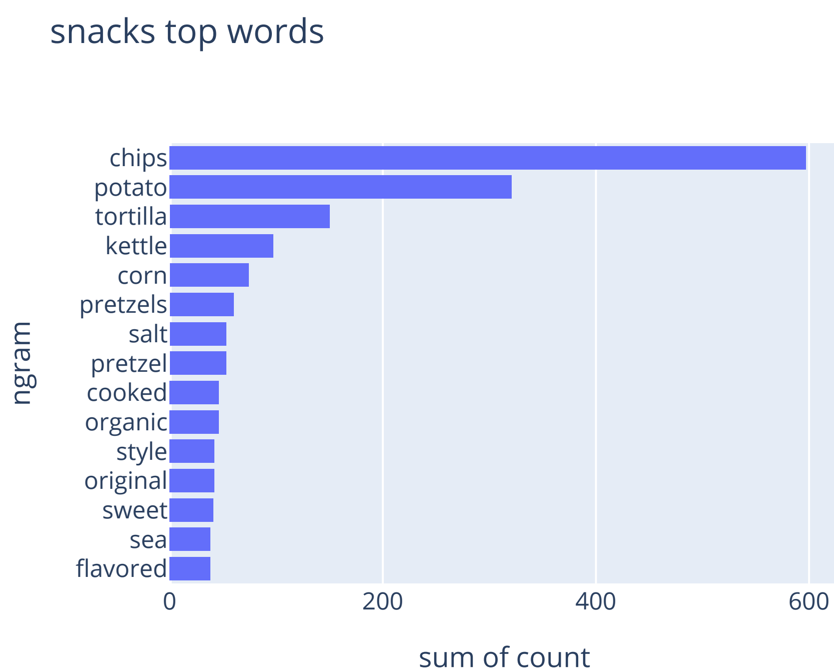
Let's look for common words in the description

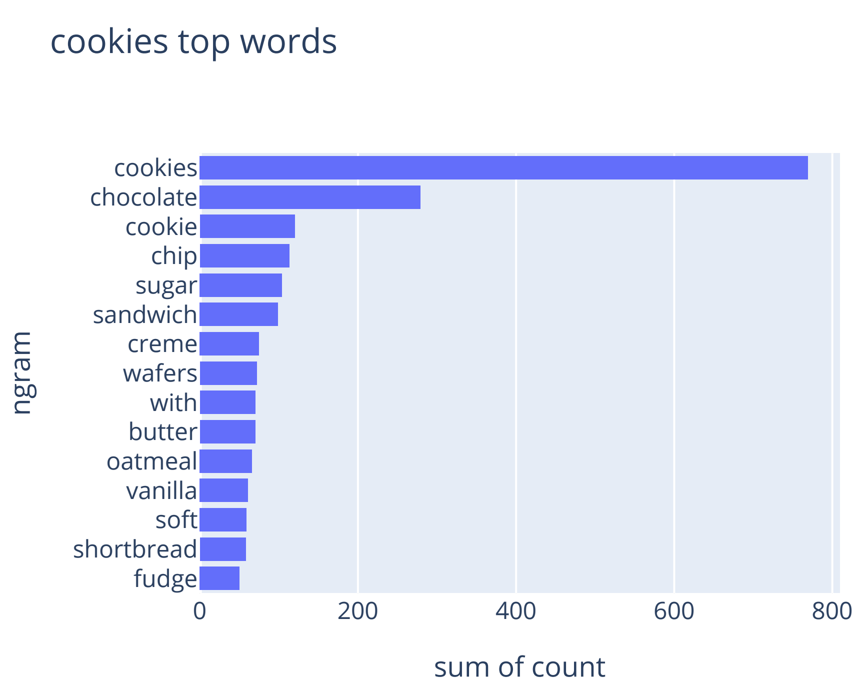
```
[ ]: X_fe_with_cat, top_15_desc_words = get_ngrams_top_k(X_fe_with_cat,␣
      ↪'description', 1, 15)
     for cat, top in top_15_desc_words.items():
         px.histogram(top, orientation='h', y='ngram', x='count', title='{} top␣
      ↪words'.format(cat), height=400, width=500).show()
```

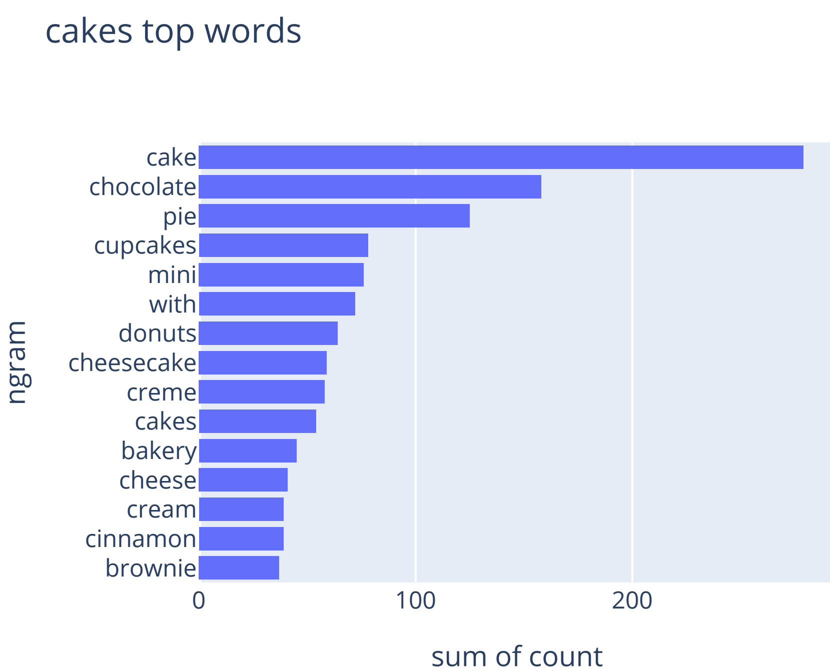

seeds top words
ngram
roasted
mix
almonds
popcorn
peanuts
chocolate
salted
trail
cashews
nuts
dark
salt
organic
corn
covered
0
100
200
300
sum of count


chocolate top words
ngram
chocolate
dark
milk
with
caramel
truffles
bar
chocolates
almond
salt
sea
assorted
white
organic
in
0
200
400
600
800
sum of count


candy top words
candy
fruit
chocolate
sour
gummi
gummy
jelly
candies
milk
cherry
chewy
with
peanut
flavored
bears
ngram
0
200
400
600
sum of count


snacks top words
chips
potato
tortilla
kettle
corn
pretzels
salt
pretzel
cooked
organic
style
original
sweet
sea
flavored
ngram
0
200
400
600
sum of count


cookies top words
cookies
chocolate
cookie
chip
sugar
sandwich
creme
wafers
with
butter
oatmeal
vanilla
soft
shortbread
fudge
ngram
0
200
400
600
800
sum of count

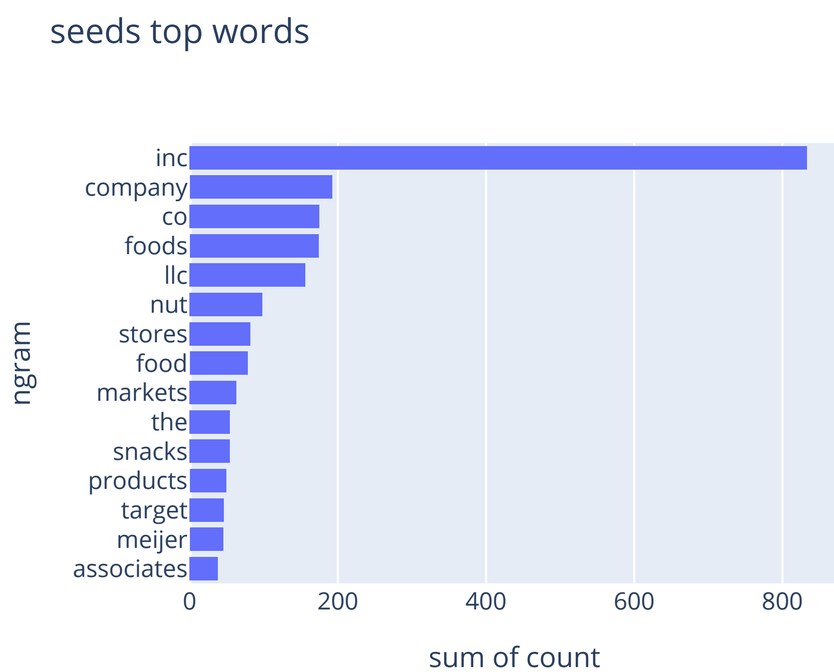


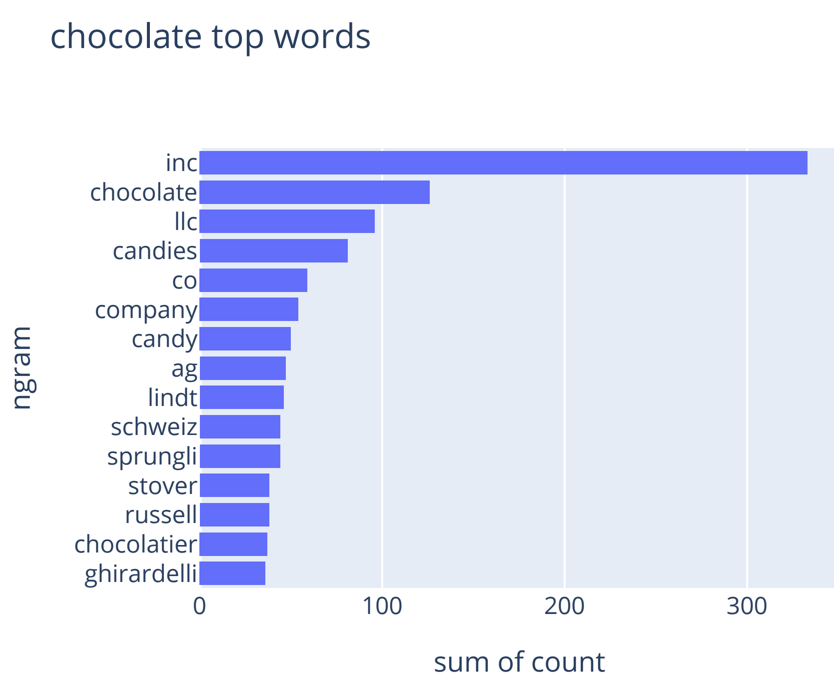
This time we'll compile the strings manually because we don't need all of them:

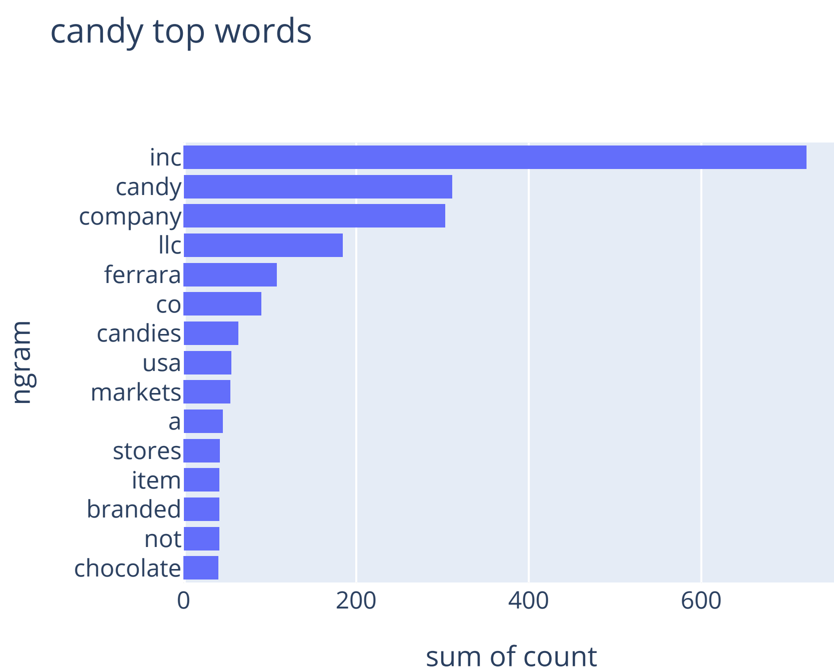
```
[ ]: description_strings = np.array([
         'roasted', 'mix', 'almonds', 'corn', 'nuts', 'chocolate', 'trail',␣
      ↪'cashews', 'salt', 'covered', 'dark', 'milk', 'caramel', 'bar',
         'truffles', 'candy', 'fruit', 'sour', 'gumm', 'jell', 'candies', 'cherry',␣
      ↪'chewy', 'chips', 'potato', 'tortilla', 'kettle', 'pretzel',
         'cookie', 'chip', 'sugar', 'sandwich', 'butter', 'oatmeal', 'vanilla',␣
      ↪'fudge', 'cake', 'pie', 'donut', 'cupcakes', 'cheesecake', 'bakery',
         'brownie'
     ])
```

Let's do the same for the brands:

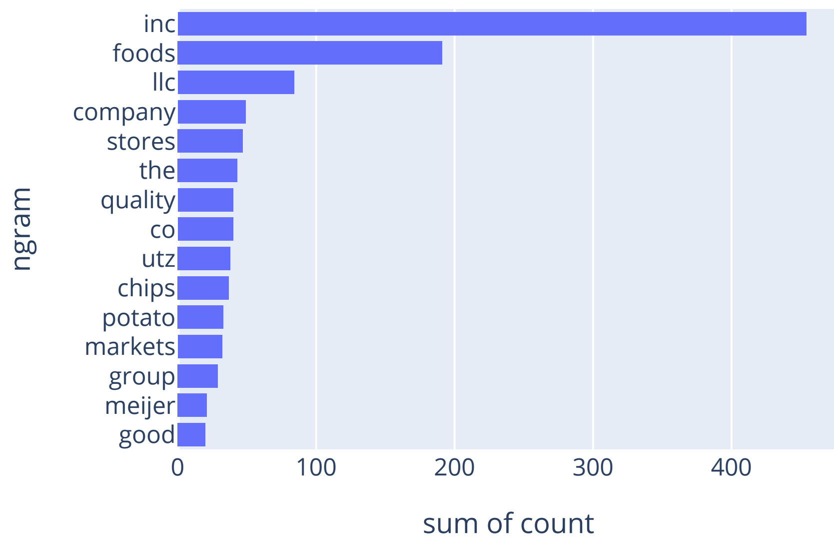
```
[ ]: X_fe_with_cat, top_15_brand_words = get_ngrams_top_k(X_fe_with_cat, 'brand', 1,␣
      ↪15)
     for cat, top in top_15_brand_words.items():
         px.histogram(top, orientation='h', y='ngram', x='count', title='{} top␣
      ↪words'.format(cat), height=400, width=500).show()
```


seeds top words
ngram
inc
company
co
foods
llc
nut
stores
food
markets
the
snacks
products
target
meijer
associates
0
200
400
600
800
sum of count

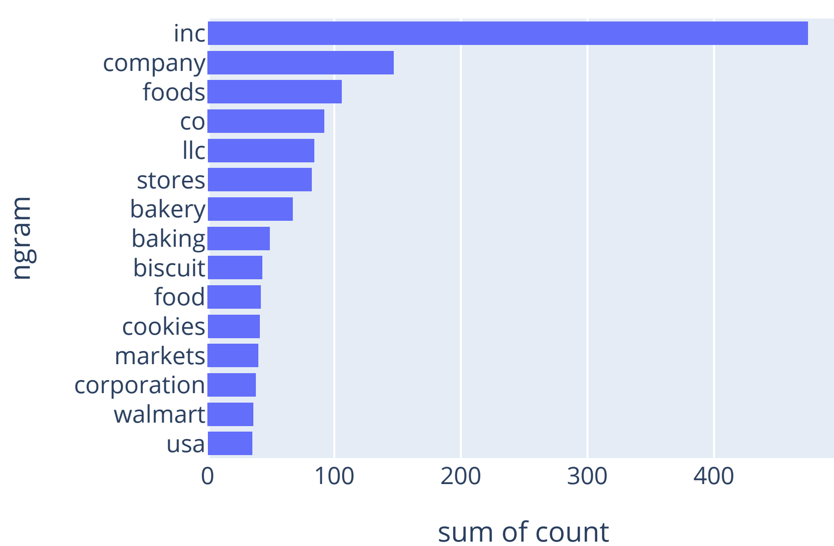

chocolate top words
ngram
inc
chocolate
llc
candies
co
company
candy
ag
lindt
schweiz
sprungli
stover
russell
chocolatier
ghirardelli
0
100
200
300
sum of count

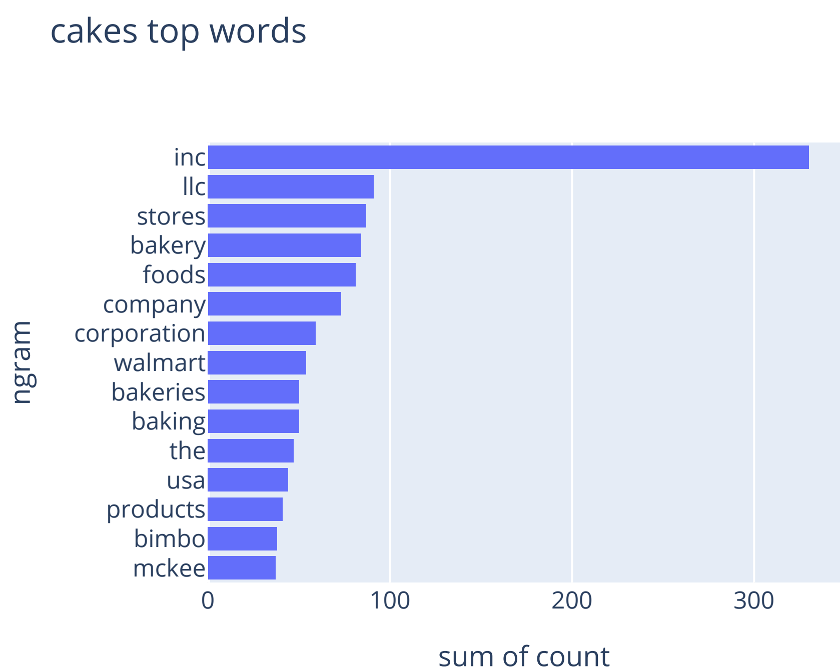

candy top words
inc
candy
company
llc
ferrara
co
candies
usa
markets
a
stores
item
branded
not
chocolate
ngram
0
200
400
600
sum of count

snacks top words



cookies top words



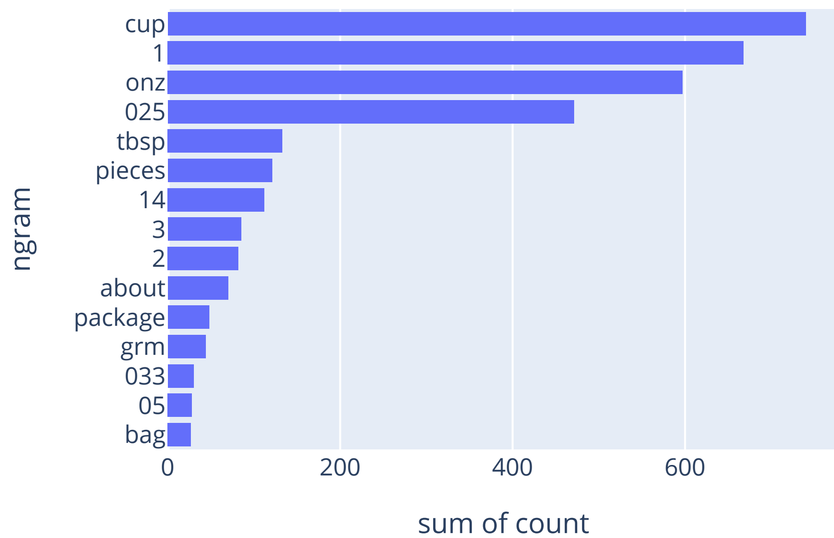



```
[ ]: brand_strings = np.array([
         'snack', 'chocola', 'candy', 'candies', 'chips', 'potato', 'bake',␣
      ↪'baking', 'biscuit', 'cookie', 'jelly', 'nuts'
     ])
```

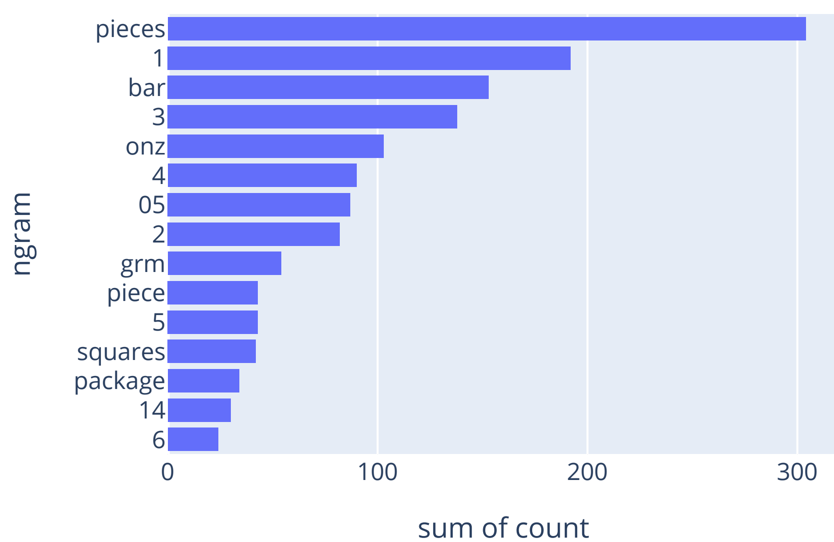
And do the same for the household_serving_fulltext:

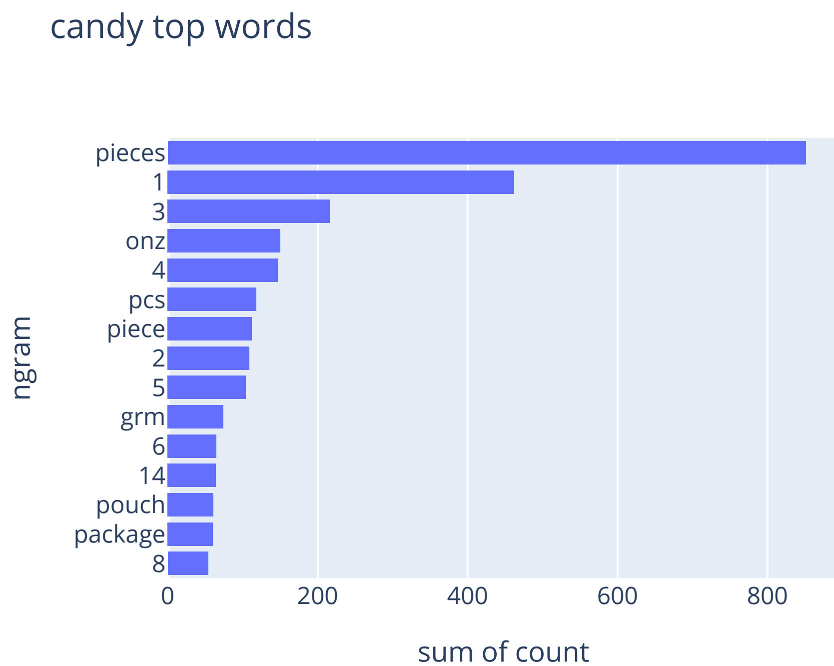
```
[ ]: X_fe_with_cat, top_15_household_serving_fulltext_words =␣
      ↪get_ngrams_top_k(X_fe_with_cat, 'household_serving_fulltext', 1, 15)
     for cat, top in top_15_household_serving_fulltext_words.items():
         px.histogram(top, orientation='h', y='ngram', x='count', title='{} top␣
      ↪words'.format(cat), height=400, width=500).show()
```

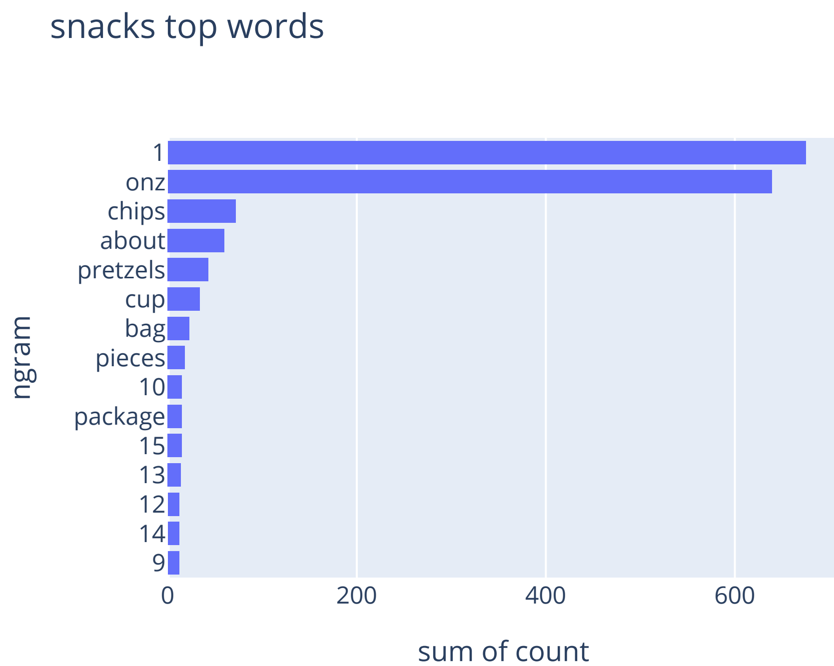
seeds top words

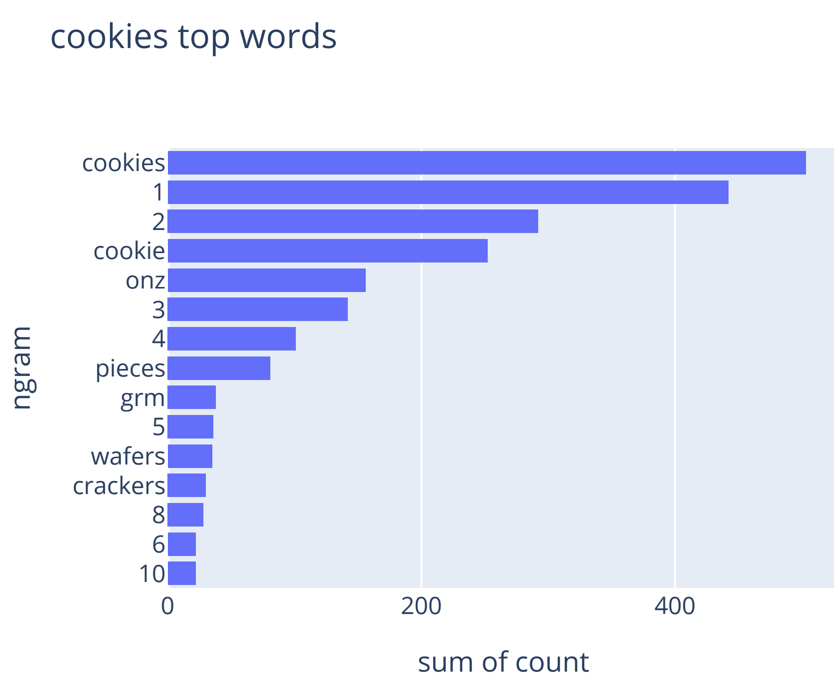


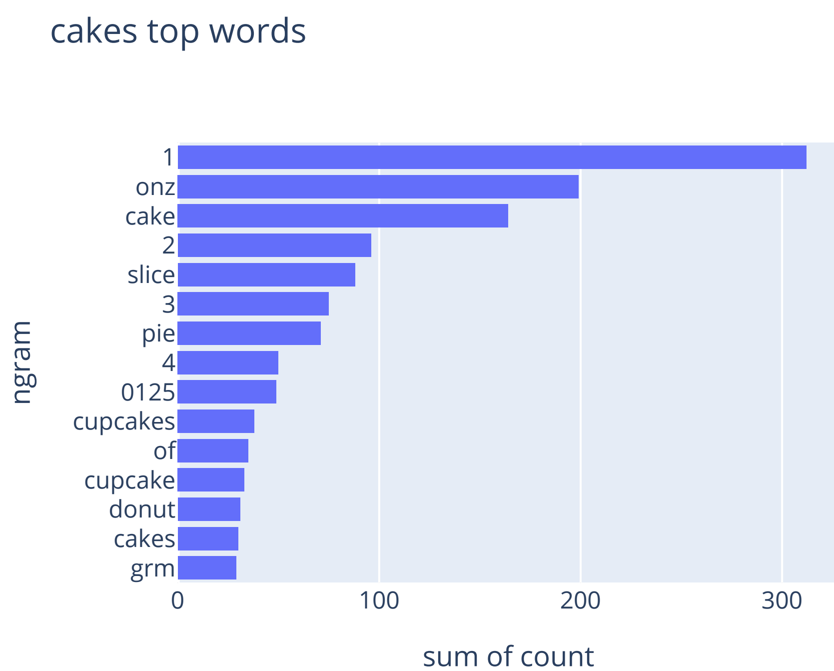
chocolate top words




candy top words
pieces
1
3
onz
4
pcs
piece
2
5
grm
6
14
pouch
package
8
ngram
0
200
400
600
800
sum of count


snacks top words
ngram
1
onz
chips
about
pretzels
cup
bag
pieces
10
package
15
13
12
14
9
0
200
400
600
sum of count

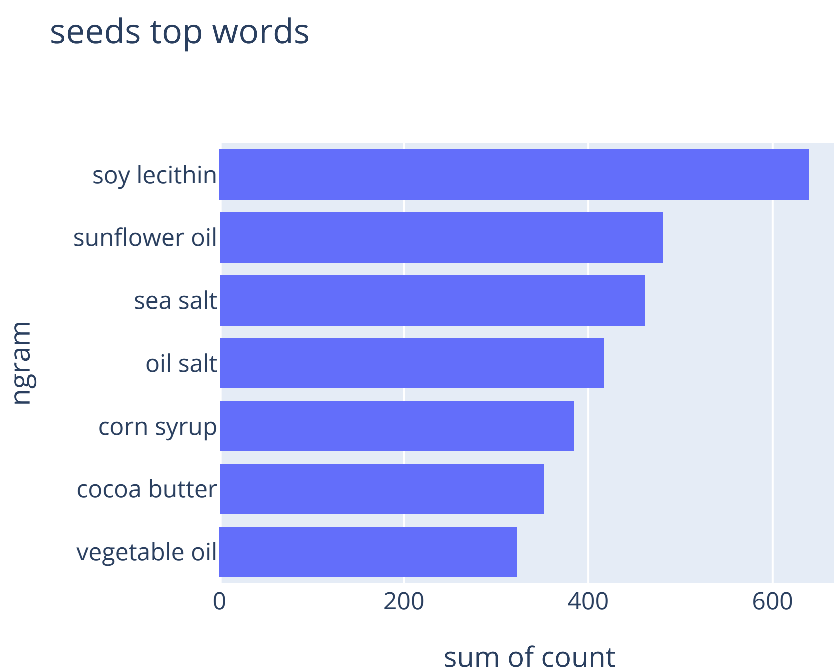

cookies top words
ngram
cookies
1
2
cookie
onz
3
4
pieces
grm
5
wafers
crackers
8
6
10
0
200
400
sum of count

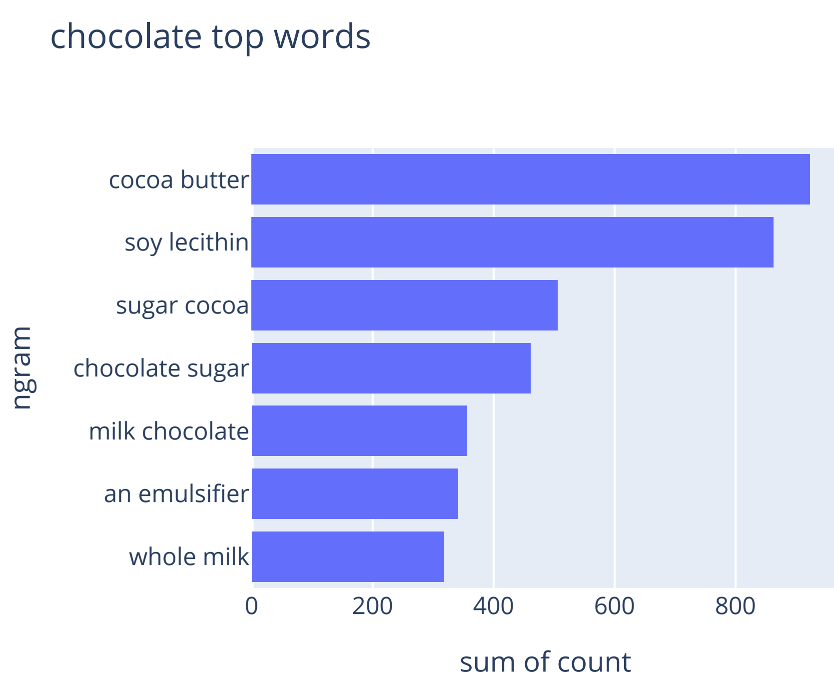


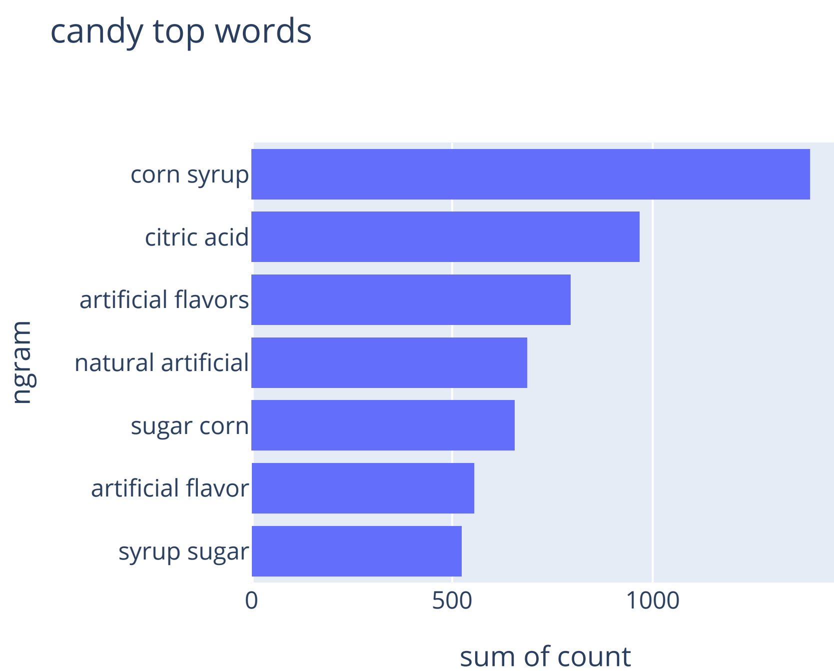

```
[ ]: household_serving_fulltext_strings = np.array([
         'cup', 'onz', 'tbsp', 'pieces', 'about', 'package', 'grm', 'bag', 'bar',␣
      ↪'piece', 'squares', 'pcs', 'pouch', 'chips', 'pretzels', 'cookie',
         'wafers', 'crackers','slice', 'pie', 'cake', 'cupcake', 'donut'
     ])
```

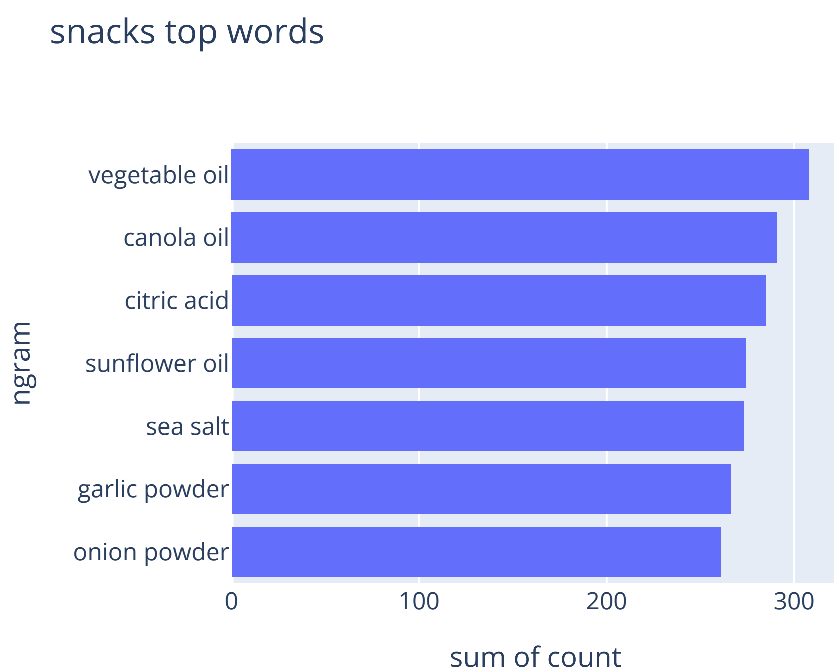
Let us do the same for bi-grams:

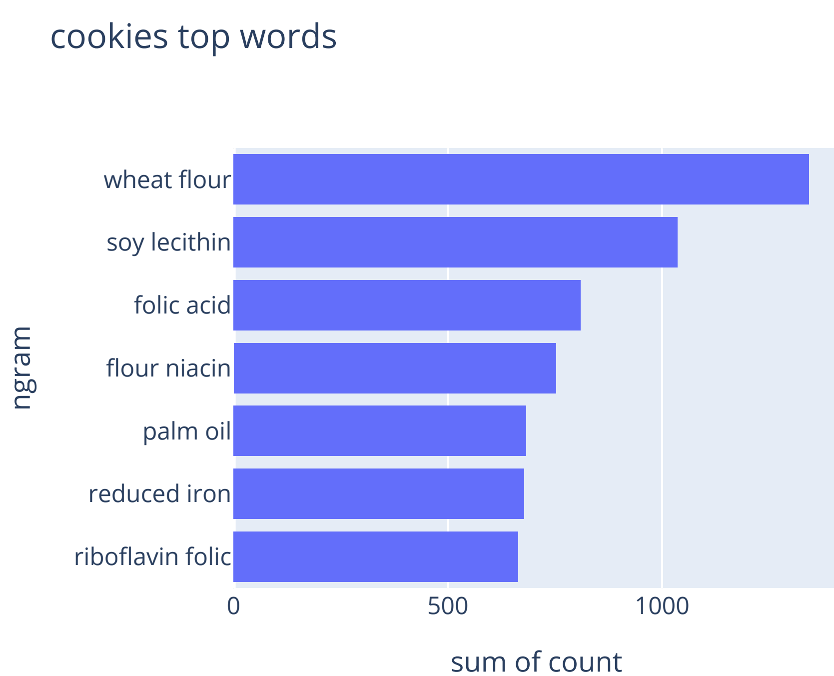
```
[ ]: X_fe_with_cat, top_7_ingredients_bigrams = get_ngrams_top_k(X_fe_with_cat,␣
      ↪'ingredients', 2, 7, True)
     for cat, top in top_7_ingredients_bigrams.items():
         px.histogram(top, orientation='h', y='ngram', x='count', title='{} top␣
      ↪words'.format(cat), height=400, width=500).show()
         ingredients_strings = np.concatenate([ingredients_strings, top['ngram'].
      ↪values], axis=0)
```

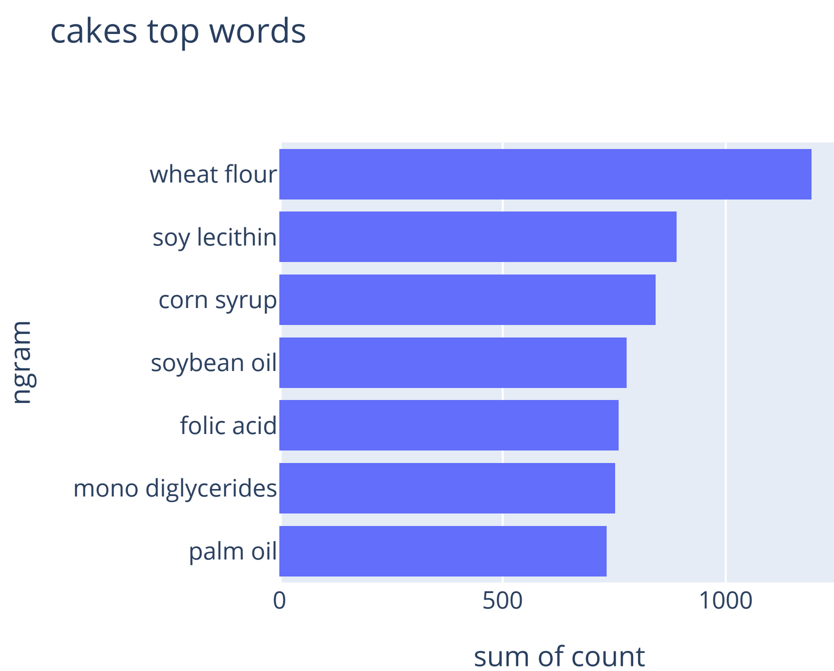

seeds top words
ngram
soy lecithin
sunflower oil
sea salt
oil salt
corn syrup
cocoa butter
vegetable oil
0
200
400
600
sum of count


chocolate top words
ngram
cocoa butter
soy lecithin
sugar cocoa
chocolate sugar
milk chocolate
an emulsifier
whole milk
0
200
400
600
800
sum of count

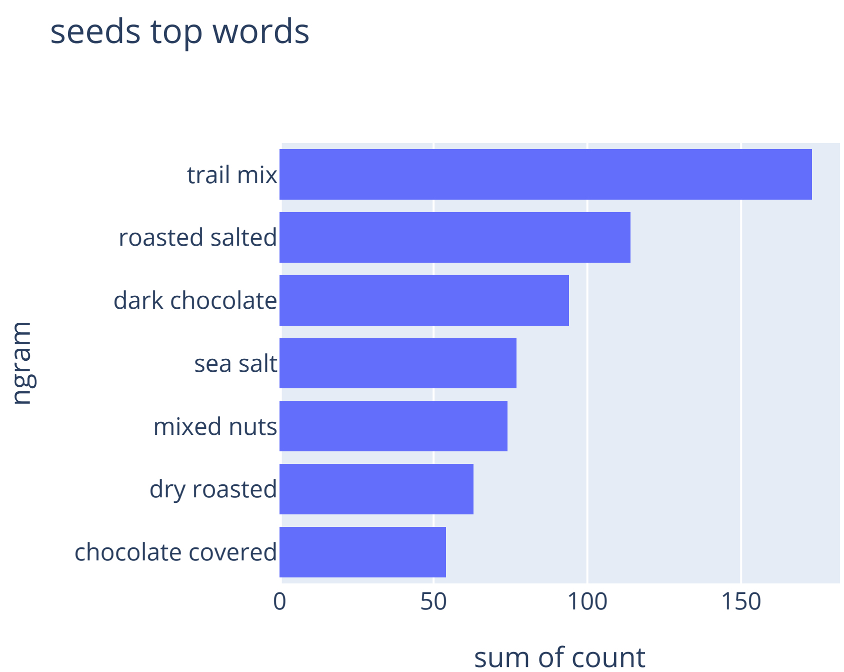

candy top words
corn syrup
citric acid
artificial flavors
natural artificial
sugar corn
artificial flavor
syrup sugar
ngram
0
500
1000
sum of count

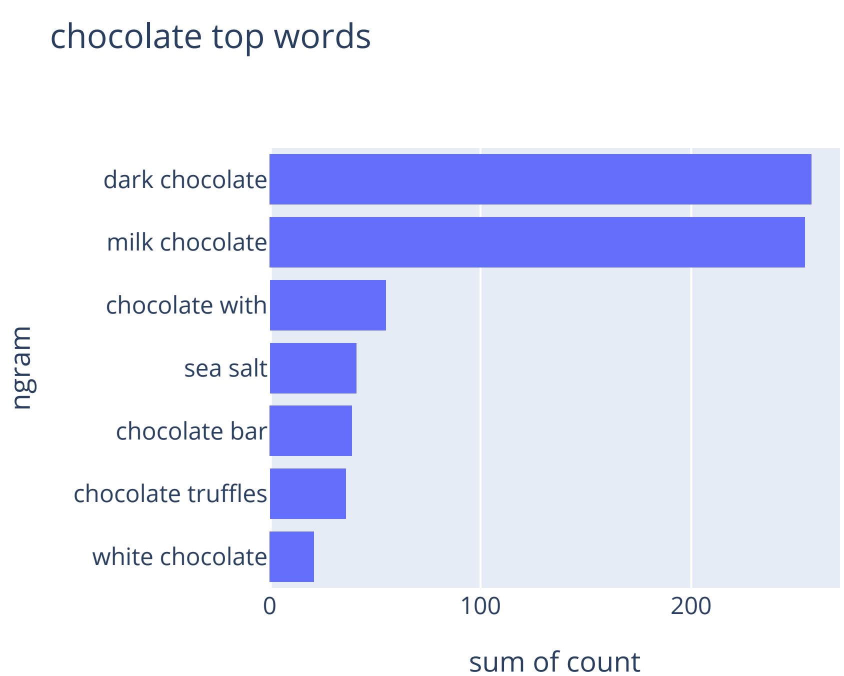

snacks top words
vegetable oil
canola oil
citric acid
sunflower oil
sea salt
garlic powder
onion powder
ngram
0
100
200
300
sum of count

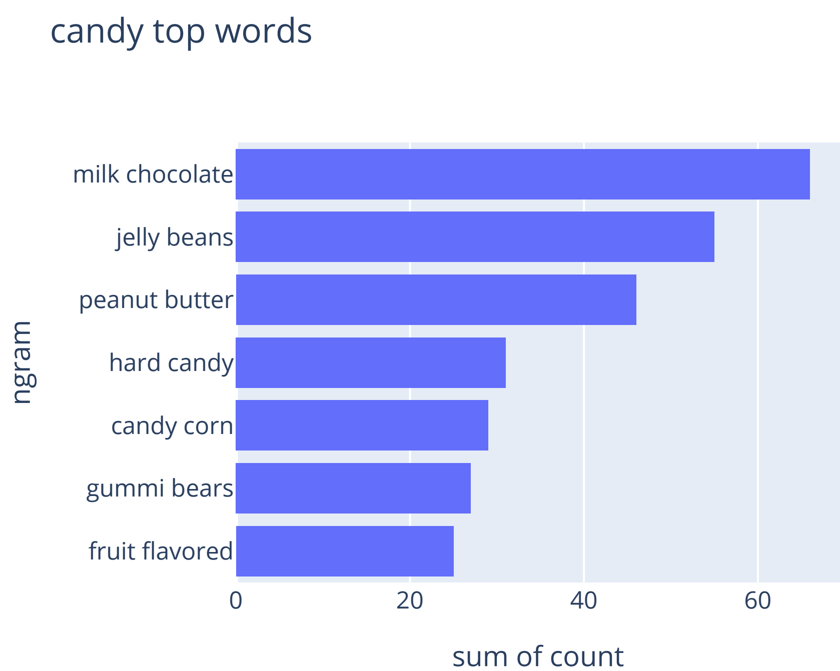

cookies top words
wheat flour
soy lecithin
folic acid
flour niacin
palm oil
reduced iron
riboflavin folic
ngram
0
500
1000
sum of count

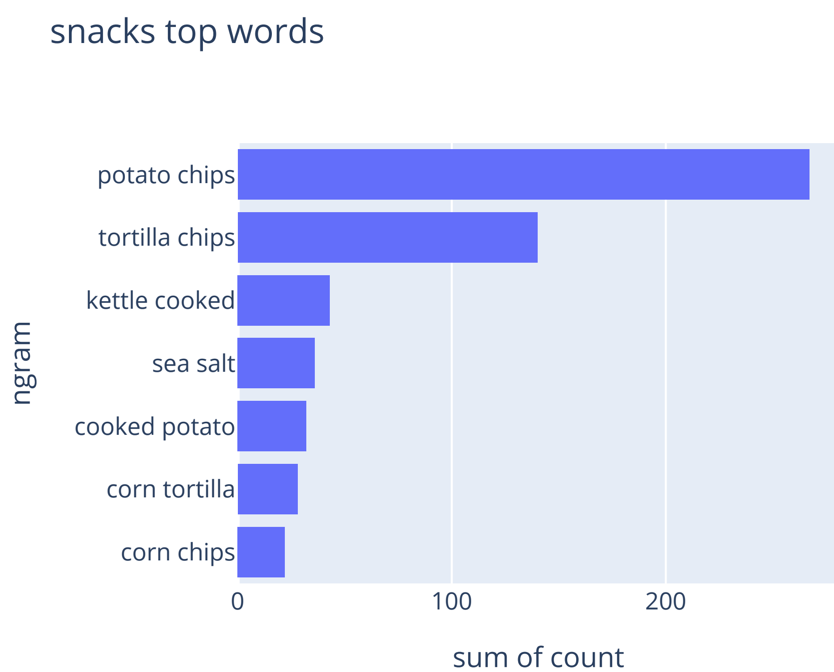


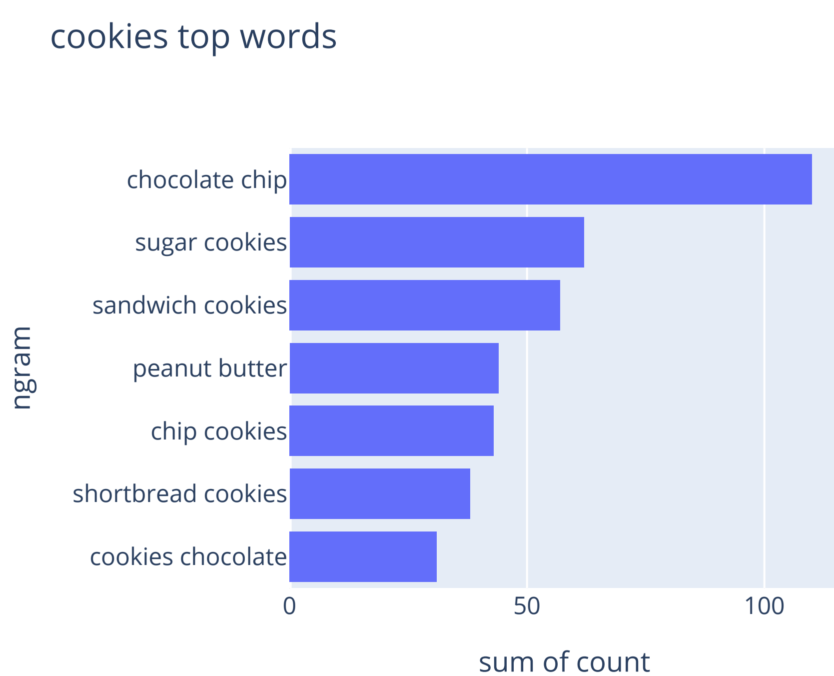
[ ]:
```
X_fe_with_cat, top_7_description_bigrams = get_ngrams_top_k(X_fe_with_cat,␣
↪'description', 2, 7, True)
for cat, top in top_7_description_bigrams.items():
    px.histogram(top, orientation='h', y='ngram', x='count', title='{} top␣
↪words'.format(cat), height=400, width=500).show()
    description_strings = np.concatenate([description_strings, top['ngram'].
↪values], axis=0)
```

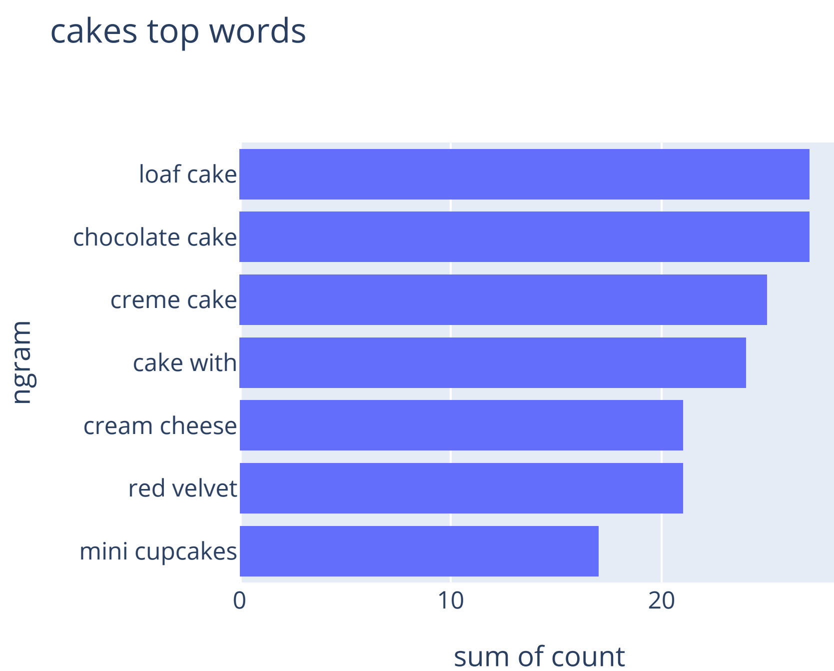

seeds top words
ngram
trail mix
roasted salted
dark chocolate
sea salt
mixed nuts
dry roasted
chocolate covered
0
50
100
150
sum of count


chocolate top words
ngram
dark chocolate
milk chocolate
chocolate with
sea salt
chocolate bar
chocolate truffles
white chocolate
0
100
200
sum of count


candy top words
milk chocolate
jelly beans
peanut butter
hard candy
candy corn
gummi bears
fruit flavored
ngram
0
20
40
60
sum of count


snacks top words
potato chips
tortilla chips
kettle cooked
sea salt
cooked potato
corn tortilla
corn chips
ngram
0
100
200
sum of count


cookies top words
chocolate chip
sugar cookies
sandwich cookies
peanut butter
chip cookies
shortbread cookies
cookies chocolate
ngram
0
50
100
sum of count



Let us create a proper dictionary for convinience later:

[ ]:
```
feature_strings_dict = {
    'ingredients': np.unique(ingredients_strings),
    'description': np.unique(description_strings),
    'brand': np.unique(brand_strings),
    'household_serving_fulltext': np.unique(household_serving_fulltext_strings)
}
```

We also created a function to fix some brand values (which were manually picked from the X_fe dataset).

After the initial feature engineering, let us choose how we will impute our values

We'll create a pipeline which first imputes the missing values with various methods, and then cross validates the results of a vanilla XGB classifier. The methods used are comprised of simple imputations including the mean, median values or filling all missing values with 0, and more complex imputations using linear and trees regressions with various parameters to impute the missing values. Using balanced accuracy score to account for class imbalance, as we have yet to address it.

[ ]:
```
X_mdi_fe = fe(X_mdi, feature_strings_dict)
```

```
[ ]: mdi_pipeline_steps = {'imputer':['simple_imp', 'iterative_imp'],
                         'classifier':['xgbc']}

     # fill parameters to be searched in this dict
     mdi_all_param_grids = {'xgbc':{'object':XGBClassifier(random_state=1337),
                                    'use_label_encoder': [False],
                                    'objective': ['multi:softmax'],
                                    'eval_metric': ['mlogloss']
                                  , 'tree_method': ['gpu_hist']
                                   },
                          'simple_imp':{
                               'object': SimpleImputer(),
                               'strategy':['mean', 'median', 'constant']
                               },
                          'iterative_imp':{
                               'object':IterativeImputer(random_state=1337),
                               'estimator':[linear_model.
       ↪ElasticNet(random_state=1337)],
                               'max_iter': [200],
                               'skip_complete': [True],
                               'initial_strategy': ['mean', 'median', 'constant'],
                               'imputation_order': ['ascending', 'descending',␣
       ↪'random']
                               }
                         }

     mdi_param_grids_list = make_param_grids(mdi_pipeline_steps, mdi_all_param_grids)
     mdi_pipe = Pipeline(steps=[('imputer', SimpleImputer()), ('classifier',␣
       ↪XGBClassifier())])
     mdi_grid = model_selection.GridSearchCV(mdi_pipe, param_grid =␣
       ↪mdi_param_grids_list, scoring='balanced_accuracy', cv=10, verbose=15)
     mdi_grid.fit(X_mdi_fe, y_mdi)
```

```
[ ]: mdi_grid.best_params_
```

```
[ ]: pd.DataFrame(mdi_grid.cv_results_).sort_values('rank_test_score').
       ↪head(5)[['params', 'mean_test_score']].values
```

Result is:

IterativeImputer(estimator=ElasticNet(random_state=1337), imputation_order='descending', initial_strategy='median', max_iter=200, random_state=1337, skip_complete=True)

On to resampling.

We won't test only downsampling because we want to get a balanced dataset, and the minority classes have too few samples.

We'll test several strategies - Random upsampling, random upsampling with downsampling, SMOTE upsampling, SMOTE upsampling with downsampling, KMeans SMOTE oversampling, KMeans SMOTE oversampling with downsampling.

For downsampling we will test random downsampling, and no downsampling.

Each strategy which involves downsampling will test 10 values - 0.95, 0.9, 0.85, 0.8, 0.75, 0.7, 0.65, 0.6, 0.55, 0.5 of the largest class

The random upsampling strategies will also test shrinkage values - 0, 1, 2, 3

The SMOTE strategies will test different numbers of k_neighbors - 5, 10, 15

In total we get 77 strategies to choose from.

In this case we will also use a vanilla XGBoost to compare out results (this time using accuracy instead of balanced accuracy)

[ ]:
```
chosen_imputer = IterativeImputer(estimator=linear_model.
 ↪ElasticNet(random_state=1337),
                  imputation_order='descending', initial_strategy='mean',
                  max_iter=200, random_state=1337, skip_complete=True)
chosen_imputer.fit(X_mdi_fe)
```

[ ]:
```
IterativeImputer(estimator=ElasticNet(random_state=1337),
                 imputation_order='descending', max_iter=200, random_state=1337,
                 skip_complete=True)
```

[ ]:
```
X_rs_fe = fe(X_rs, feature_strings_dict)
X_rs_imputed = chosen_imputer.transform(X_rs_fe)
```

[ ]:
```
# Had to give up lambdas to recognize functions after CV :(

# # Creates lambas to feed into the strategies argument of the downsamplers␣
 ↪(because of the CV the numbers are not constant)
# def get_strategy_lambda(i):
#     return lambda y: {j: round(max(np.bincount(y)) * i * 0.1) for j in␣
 ↪range(0,6)}
# strategy_lambdas = [get_strategy_lambda(i) for i in range(6, 10)]
strategy_funcs = []
def_strategy_function_exec_code = """def strategy_{}(y): return {{j:␣
 ↪round(max(np.bincount(y)) * {} * 0.05) for j in range(0,6)}}"""
for i in range(10,20):
  exec(def_strategy_function_exec_code.format(round(i * 5), i))
  exec("""strategy_funcs.append(strategy_{})""".format(i * 5))


# Had to use these step names becasue imblearn uses a custom pipe constructor.
# found this out after I finished these dicts and almost cried in fear of a␣
 ↪much more complicated CV
rs_pipeline_steps = {
```

```
     'randomoversampler': ['random_us', 'smote_us'],
     'randomundersampler': ['no_ds', 'random_ds'],
     'xgbclassifier': ['xgbc']
}

rs_all_param_grids = {'xgbc': {'object':XGBClassifier(random_state=1337),
                          'use_label_encoder': [False],
                          'objective': ['multi:softmax'],
                          'eval_metric': ['mlogloss']
                        , 'tree_method': ['gpu_hist']
                         },
                   'random_us': {
                        'object': over_sampling.
 ↪RandomOverSampler(random_state=1337),
                        'shrinkage': [0, 1, 2, 3]
                        },
                   'smote_us': {
                        'object': over_sampling.SMOTE(random_state=1337),
                        'k_neighbors': [5, 10, 15]
                        },
                   'no_ds': {
                        'object': None
                        },
                   'random_ds': {
                        'object': under_sampling.
 ↪RandomUnderSampler(random_state=1337),
                        'sampling_strategy': strategy_funcs
                        }
                  }

rs_param_grids_list = make_param_grids(rs_pipeline_steps, rs_all_param_grids)
# rs_pipe = Pipeline(steps=[('up_sample', None), ('down_sample', None),␣
 ↪('classifier', None)])
rs_pipe = pipeline.make_pipeline(over_sampling.RandomOverSampler(),␣
 ↪under_sampling.RandomUnderSampler(), XGBClassifier()) # imblearn pipe
rs_grid = model_selection.GridSearchCV(rs_pipe, param_grid =␣
 ↪rs_param_grids_list, scoring='accuracy', cv=10, verbose=15)
rs_grid.fit(X_rs_imputed, y_rs)
```

Results are: SMOTE(k_neighbors=10, random_state=1337)

RandomUnderSampler(random_state=1337, sampling_strategy=<function strategy_75 at 0x7fec5bb6def0>)

So, SMOTE with k_neighbors=10 and RandomUnderSampler with a downsampling to 75% of the largest class.

```
[ ]: pd.options.mode.chained_assignment = None

     brands_for_column_values = X_fe[['brand']] # Take brands from FE portion of␣
      ↪dataset
     brands_column_values = get_column_values(brands_for_column_values)
     brands = X_fe[['brand']]
     ok_brands = brands_column_values['brand'][brands_column_values['brand']['pc']␣
      ↪>= 0.1]['value'].values
     brands_mask = ~brands['brand'].isin(ok_brands)
     brands.loc[brands_mask, 'brand'] = 'other'
     brands_ohe, _ = get_oh_encoder(brands)
     brands_ohe.fit(brands)
```

```
[ ]: OneHotEncoder(categories=[['7eleven inc', 'abimar foods inc', 'ahold usa inc',
                                'aldibenner company', 'american halal company inc',
                                'american importing co inc', 'archer farms',
                                'back to nature foods company llc',
                                'bee international inc', 'bergin nut company inc',
                                'best choice', 'better made snack foods inc',
                                'big y foods inc', 'bimbo bakeries usa inc',
                                'bjs wholesale club corporate brands',
                                'blue diamond growers', 'brads raw chips',
                                'brookshire grocery company',
                                'california flavored nuts', 'candyrific llc',
                                'cape cod potato chips inc', 'charms company',
                                'cibo vita inc', 'creative natural products inc',
                                'creative snacks co llc',
                                'csm bakery products na inc', 'cvs pharmacy inc',
                                'dawn food products inc', 'delhaize america inc',
                                'demets candy company', …]],
                  handle_unknown='ignore', sparse=False)
```

Let us remove highly correlated features

Now let us write a function which ties the dataset together, before moving to high correlated columns removal

```
[ ]: def get_numerical_df(df, feature_strings_dict, brands_ohe):
       df = fix_brands(df)
       brands = df[['brand']]
       brands_oh = brands_ohe.transform(brands)
       df_fe = fe(df, feature_strings_dict)
       numerical_df = np.concatenate([brands_oh, df_fe], axis=1)
       return numerical_df
```

```
[ ]: # Refit chosen imputer to the new columns
     X_mdi_numerical = get_numerical_df(X_mdi, feature_strings_dict, brands_ohe)
```

```
chosen_imputer = IterativeImputer(estimator=linear_model.
↪ElasticNet(random_state=1337),
                    imputation_order='descending', initial_strategy='mean',
                    max_iter=200, random_state=1337, skip_complete=True)
chosen_imputer.fit(X_mdi_numerical)
```

[ ]:
```
IterativeImputer(estimator=ElasticNet(random_state=1337),
                 imputation_order='descending', max_iter=200, random_state=1337,
                 skip_complete=True)
```

[ ]:
```
X_fe_numerical = get_numerical_df(X_fe, feature_strings_dict, brands_ohe)
X_fe_numerical_imputed = chosen_imputer.transform(X_fe_numerical)
```

Let's use CV to find best value to consider a high correlation:

[ ]:
```
def strategy_75(y):
  return {j: round(max(np.bincount(y)) * 15 * 0.05) for j in range(0,6)}

chosen_os = over_sampling.SMOTE(k_neighbors=10, random_state=1337)
chosen_us = under_sampling.RandomUnderSampler(random_state=1337,␣
↪sampling_strategy=strategy_75)
vanilla_xgbc = XGBClassifier(eval_metric='mlogloss', objective='multi:softmax',
               random_state=1337, tree_method='gpu_hist',
               use_label_encoder=False)
```

[ ]:
```
corr_pipe = pipeline.make_pipeline(chosen_os, chosen_us, vanilla_xgbc)
cv_scores = {}
for corr in [0.6, 0.7, 0.8, 0.9]:
  X_fe_numerical_df = pd.DataFrame(X_fe_numerical_imputed)
  X_fe_corr = X_fe_numerical_df.corr()
  upper_tri = X_fe_corr.where(np.triu(np.ones(X_fe_corr.shape), k=1).
↪astype(bool))
  corr_columns_to_drop = [column for column in upper_tri.columns if␣
↪any(upper_tri[column] > corr)]
  X_fe_numerical_corr = X_fe_numerical_df.drop(corr_columns_to_drop, axis=1).
↪values
  cv_scores[corr] = model_selection.cross_val_score(corr_pipe,␣
↪X_fe_numerical_corr, y_fe, cv=10,
                                                    verbose=15, n_jobs=-1).
↪mean()
max_score = max(cv_scores, key=cv_scores.get)
max_score
```

So we will remove all columns with correlation above the chosen threshold.

[ ]:
```
X_fe_numerical_df = pd.DataFrame(X_fe_numerical_imputed)
X_fe_corr = X_fe_numerical_df.corr()
```

```
upper_tri = X_fe_corr.where(np.triu(np.ones(X_fe_corr.shape), k=1).astype(bool))
corr_columns_to_drop = [column for column in upper_tri.columns if␣
↪any(upper_tri[column] > max_score)]
```

[ ]:
```
def get_final_matrix(X, feature_strings_dict, brands_ohe, corr_columns_to_drop,␣
↪chosen_imputer):
    numerical_X = get_numerical_df(X, feature_strings_dict, brands_ohe)
    X_imputed_df = pd.DataFrame(chosen_imputer.transform(numerical_X))
    final_matrix = X_imputed_df.drop(corr_columns_to_drop, axis=1).values
    return final_matrix
```

[ ]:
```
X_mt_numerical = get_final_matrix(X_mt, feature_strings_dict, brands_ohe,␣
↪corr_columns_to_drop, chosen_imputer)
```

Now let us tune our XGB Classifier one step at a time, starting with the number of trees for the default learning rate.

[ ]:
```
mt_pipeline_steps_ne = {
     'xgbclassifier': ['xgbc']
}

mt_all_param_grids_ne = {'xgbc': {'object':XGBClassifier(random_state=1337),
                          'n_estimators': [100 + i * 10 for i in range(0, 41)],
                          'use_label_encoder': [False],
                          'objective': ['multi:softmax'],
                          'eval_metric': ['mlogloss']
                        , 'tree_method': ['gpu_hist']
                         }
                  }

mt_param_grids_list_ne = make_param_grids(mt_pipeline_steps_ne,␣
↪mt_all_param_grids_ne)
mt_pipe_ne = pipeline.make_pipeline(chosen_os, chosen_us, vanilla_xgbc)
mt_grid_ne = model_selection.GridSearchCV(mt_pipe_ne, param_grid =␣
↪mt_param_grids_list_ne, scoring='accuracy', cv=10, verbose=15)
mt_grid_ne.fit(X_mt_numerical, y_mt)
```

[ ]:
```
mt_grid_ne.best_params_['xgbclassifier__n_estimators']
```

Now let's tune other parameters:

[ ]:
```
mt_pipeline_steps_mdcw = {
     'xgbclassifier': ['xgbc']
}

mt_all_param_grids_mdcw = {'xgbc': {'object':XGBClassifier(random_state=1337),
```

```
                              'n_estimators': [mt_grid_ne.
 ↪best_params_['xgbclassifier__n_estimators']],
                              'max_depth': [i for i in range(3, 13)],
                              'min_child_weight': [i for i in range(1, 7)],
                              'use_label_encoder': [False],
                              'objective': ['multi:softmax'],
                              'eval_metric': ['mlogloss']
                            , 'tree_method': ['gpu_hist']
                             }
                     }

mt_param_grids_list_mdcw = make_param_grids(mt_pipeline_steps_mdcw,␣
 ↪mt_all_param_grids_mdcw)
mt_pipe_mdcw = pipeline.make_pipeline(chosen_os, chosen_us, vanilla_xgbc)
mt_grid_mdcw = model_selection.GridSearchCV(mt_pipe_mdcw, param_grid =␣
 ↪mt_param_grids_list_mdcw, scoring='accuracy', cv=10, verbose=15)
mt_grid_mdcw.fit(X_mt_numerical, y_mt)
```

[ ]:
```
mt_grid_mdcw.best_params_
```

[ ]:
```
print(mt_grid_mdcw.best_params_['xgbclassifier__max_depth'])
print(mt_grid_mdcw.best_params_['xgbclassifier__min_child_weight'])
```

[ ]:
```
mt_pipeline_steps_gam = {
     'xgbclassifier': ['xgbc']
}

mt_all_param_grids_gam = {'xgbc': {'object':XGBClassifier(random_state=1337),
                              'n_estimators': [mt_grid_ne.
 ↪best_params_['xgbclassifier__n_estimators']],
                              'max_depth': [mt_grid_mdcw.
 ↪best_params_['xgbclassifier__max_depth']],
                              'min_child_weight': [mt_grid_mdcw.
 ↪best_params_['xgbclassifier__min_child_weight']],
                              'gamma': [i / 10.0 for i in range(0, 5)],
                              'use_label_encoder': [False],
                              'objective': ['multi:softmax'],
                              'eval_metric': ['mlogloss']
                            , 'tree_method': ['gpu_hist']
                             }
                     }

mt_param_grids_list_gam = make_param_grids(mt_pipeline_steps_gam,␣
 ↪mt_all_param_grids_gam)
mt_pipe_gam = pipeline.make_pipeline(chosen_os, chosen_us, vanilla_xgbc)
```

```
mt_grid_gam = model_selection.GridSearchCV(mt_pipe_gam, param_grid =␣
 ↪mt_param_grids_list_gam, scoring='accuracy', cv=10, verbose=15)
mt_grid_gam.fit(X_mt_numerical, y_mt)
```

[ ]:
```
mt_grid_gam.best_params_['xgbclassifier__gamma']
```

On to check subsample and colsample_bytree.

We will first check with 0.1 intervals, and then 0.05 intervals around the chosen values.

[ ]:
```
mt_pipeline_steps_sscs = {
     'xgbclassifier': ['xgbc']
}

mt_all_param_grids_sscs = {'xgbc': {'object':XGBClassifier(random_state=1337),
                          'n_estimators': [mt_grid_ne.
 ↪best_params_['xgbclassifier__n_estimators']],
                          'max_depth': [mt_grid_mdcw.
 ↪best_params_['xgbclassifier__max_depth']],
                          'min_child_weight': [mt_grid_mdcw.
 ↪best_params_['xgbclassifier__min_child_weight']],
                          'gamma': [mt_grid_gam.
 ↪best_params_['xgbclassifier__gamma']],
                          'subsample':[i/10.0 for i in range(6,11)],
                          'colsample_bytree':[i/10.0 for i in range(6,11)],
                          'use_label_encoder': [False],
                          'objective': ['multi:softmax'],
                          'eval_metric': ['mlogloss']
                        , 'tree_method': ['gpu_hist']
                         }
                  }

mt_param_grids_list_sscs = make_param_grids(mt_pipeline_steps_sscs,␣
 ↪mt_all_param_grids_sscs)
mt_pipe_sscs = pipeline.make_pipeline(chosen_os, chosen_us, vanilla_xgbc)
mt_grid_sscs = model_selection.GridSearchCV(mt_pipe_sscs, param_grid =␣
 ↪mt_param_grids_list_sscs, scoring='accuracy', cv=10, verbose=15)
mt_grid_sscs.fit(X_mt_numerical, y_mt)
```

[ ]:
```
ss = mt_grid_sscs.best_params_['xgbclassifier__subsample']
csbt = mt_grid_sscs.best_params_['xgbclassifier__colsample_bytree']
print(ss)
print(csbt)
```

[ ]:
```
ss = mt_grid_sscs.best_params_['xgbclassifier__subsample']
csbt = mt_grid_sscs.best_params_['xgbclassifier__colsample_bytree']
print(ss)
```

```
print(csbt)
```

```
[ ]: mt_pipeline_steps_sscs2 = {
         'xgbclassifier': ['xgbc']
     }

     mt_all_param_grids_sscs2 = {'xgbc': {'object':XGBClassifier(random_state=1337),
                                 'n_estimators': [mt_grid_ne.
       ↪best_params_['xgbclassifier__n_estimators']],
                                 'max_depth': [mt_grid_mdcw.
       ↪best_params_['xgbclassifier__max_depth']],
                                 'min_child_weight': [mt_grid_mdcw.
       ↪best_params_['xgbclassifier__min_child_weight']],
                                 'subsample':[ss - 0.05, ss, ss + 0.05],
                                 'colsample_bytree':[csbt - 0.05, csbt, csbt + 0.5],
                                 'use_label_encoder': [False],
                                 'objective': ['multi:softmax'],
                                 'eval_metric': ['mlogloss']
                               , 'tree_method': ['gpu_hist']
                                }
                       }

     mt_param_grids_list_sscs2 = make_param_grids(mt_pipeline_steps_sscs2,␣
       ↪mt_all_param_grids_sscs2)
     mt_pipe_sscs2 = pipeline.make_pipeline(chosen_os, chosen_us, vanilla_xgbc)
     mt_grid_sscs2 = model_selection.GridSearchCV(mt_pipe_sscs2, param_grid =␣
       ↪mt_param_grids_list_sscs2, scoring='accuracy', cv=10, verbose=15)
     mt_grid_sscs2.fit(X_mt_numerical, y_mt)
```

```
[ ]: mt_grid_sscs2.best_params_['xgbclassifier__subsample']
```

Now for regularization parameters:

```
[ ]: mt_pipeline_steps_reg = {
         'xgbclassifier': ['xgbc']
     }

     mt_all_param_grids_reg = {'xgbc': {'object':XGBClassifier(random_state=1337),
                                 'n_estimators': [mt_grid_ne.
       ↪best_params_['xgbclassifier__n_estimators']],
                                 'max_depth': [mt_grid_mdcw.
       ↪best_params_['xgbclassifier__max_depth']],
                                 'min_child_weight': [mt_grid_mdcw.
       ↪best_params_['xgbclassifier__min_child_weight']],
                                 'subsample':[mt_grid_sscs2.
       ↪best_params_['xgbclassifier__subsample']],
```

```
                          'colsample_bytree':[mt_grid_sscs2.
  ↪best_params_['xgbclassifier__colsample_bytree']],
                          'alpha': [0, 1e-5, 1e-2, 0.1, 1],
                          'lambda': [0, 1e-5, 1e-2, 0.1, 1],
                          'use_label_encoder': [False],
                          'objective': ['multi:softmax'],
                          'eval_metric': ['mlogloss']
                        , 'tree_method': ['gpu_hist']
                         }
                  }

mt_param_grids_list_reg = make_param_grids(mt_pipeline_steps_reg,␣
  ↪mt_all_param_grids_reg)
mt_pipe_reg = pipeline.make_pipeline(chosen_os, chosen_us, vanilla_xgbc)
mt_grid_reg = model_selection.GridSearchCV(mt_pipe_reg, param_grid =␣
  ↪mt_param_grids_list_reg, scoring='accuracy', cv=10, verbose=15)
mt_grid_reg.fit(X_mt_numerical, y_mt)
```

[ ]:
```
mt_grid_reg.best_params_
```

For some reason tuning the learning rate parameter didn't work (got exact same results in CV).

So now we have out XGB classifier!

XGBClassifier(alpha=0, colsample_bytree=0.8, eval_metric='mlogloss', lambda=0, max_depth=12, min_child_weight=3, n_estimators=250, objective='multi:softmax', random_state=1337, subsample=1.0, tree_method='gpu_hist', use_label_encoder=False)

Let us move on to choosing a RF classifier using CV:

[ ]:
```
mt_pipeline_steps_rf = {
     'randomforestclassifier': ['rf']
}

mt_all_param_grids_rf = {'rf': {'object':ensemble.
  ↪RandomForestClassifier(random_state=1337),
                          'n_estimators': [100 + i * 50 for i in range(0, 9)],
                          'criterion': ['entropy'],
                          'max_features': ['sqrt'],
                          'min_samples_split': [2, 3, 4, 5]
                         }
                  }

mt_param_grids_list_rf = make_param_grids(mt_pipeline_steps_rf,␣
  ↪mt_all_param_grids_rf)
mt_pipe_rf = pipeline.make_pipeline(chosen_os, chosen_us, vanilla_rf)
mt_grid_rf = model_selection.GridSearchCV(mt_pipe_rf, param_grid =␣
  ↪mt_param_grids_list_rf, scoring='accuracy', cv=10, verbose=15)
```

```
mt_grid_rf.fit(X_mt_numerical, y_mt)
```

So, our chosen RF classifier is is:

RandomForestClassifier(criterion='entropy', n_estimators=200, random_state=1337)

Let su move on to a SVM classifier:

```
[ ]: mt_pipeline_steps_svm = {
         'svc': ['svc']
     }

     mt_all_param_grids_svm = {'svc': {'object': svm.SVC(),
                                 'C': [10000, 100000, 1000000],
                                 'kernel': ['rbf', 'poly']
                                }
                         }

     mt_param_grids_list_svm = make_param_grids(mt_pipeline_steps_svm,␣
      ↪mt_all_param_grids_svm)
     mt_pipe_svm = pipeline.make_pipeline(chosen_os, chosen_us, svm.SVC())
     mt_grid_svm = model_selection.GridSearchCV(mt_pipe_svm, param_grid =␣
      ↪mt_param_grids_list_svm, scoring='accuracy', cv=5, verbose=15)
     mt_grid_svm.fit(X_mt_numerical, y_mt)
```

The result is:

SVC(C=100000)

So now we have our 2 models and we can finally get to trying them on the test set!

```
[ ]: # Prepare estimators:
     xgbc = XGBClassifier(alpha=0, colsample_bytree=0.8, eval_metric='mlogloss',␣
      ↪reg_lambda=0,
                    max_depth=12, min_child_weight=3, n_estimators=250,
                    objective='multi:softmax', random_state=1337, subsample=1.0,
                    tree_method='gpu_hist', use_label_encoder=False)
     rfc = ensemble.RandomForestClassifier(criterion='entropy', n_estimators=200,␣
      ↪random_state=1337)
     svmc = svm.SVC(C=100000)

     estimators_dict = {
         'xgbc': pipeline.make_pipeline(chosen_os, chosen_us, xgbc),
         'rfc': pipeline.make_pipeline(chosen_os, chosen_us, rfc),
         'svmc': pipeline.make_pipeline(chosen_os, chosen_us, svmc)
     }

     # Prepare test_data:
```

```
X_train_numerical = get_final_matrix(X_train, feature_strings_dict, brands_ohe,␣
↪corr_columns_to_drop, chosen_imputer)

# Fit
for name, estimator in estimators_dict.items():
    print("Fitting {}".format(name))
    estimator.fit(X_train_numerical, y_train)
    print("Finished fitting {}".format(name))
```

```
Fitting xgbc
Finished fitting xgbc
Fitting rfc
Finished fitting rfc
Fitting svmc
Finished fitting svmc
```

Let us compute the score on the test set:

```
[ ]: accs = {}
X_test_numerical = get_final_matrix(X_test, feature_strings_dict, brands_ohe,␣
↪corr_columns_to_drop, chosen_imputer)
for name, estimator in estimators_dict.items():
  print("Predicting {}".format(name))
  y_pred = estimator.predict(X_test_numerical)
  accs[name] = metrics.accuracy_score(y_test, y_pred)
  print("Finished predicting {}".format(name))
print(accs)
```

```
Predicting xgbc
Finished predicting xgbc
Predicting rfc
Finished predicting rfc
Predicting svmc
Finished predicting svmc
{'xgbc': 0.9310344827586207, 'rfc': 0.9266257282317745, 'svmc':
0.9044245000787278}
```

Not too bad. Now let us train our classifiers on all the data and make predictions for the test csv.

```
[ ]: X_numerical = get_final_matrix(X, feature_strings_dict, brands_ohe,␣
↪corr_columns_to_drop, chosen_imputer)
final_estimators_dict = {
    'xgbc': pipeline.make_pipeline(chosen_os, chosen_us, xgbc),
    'rfc': pipeline.make_pipeline(chosen_os, chosen_us, rfc),
    'svmc': pipeline.make_pipeline(chosen_os, chosen_us, svmc)
}
for name, estimator in final_estimators_dict.items():
    print("Fitting {}".format(name))
```

```
    estimator.fit(X_numerical, y)
    print("Finished fitting {}".format(name))
```

```
Fitting xgbc
Finished fitting xgbc
Fitting rfc
Finished fitting rfc
Fitting svmc
Finished fitting svmc
```

```
[ ]: test = pd.read_csv('drive/MyDrive/Final_Project-orav94/data/food_test.csv')
joined_test = pd.merge(test, nutrients, how='left', on='idx').
 ↪drop(['nutrient_id', 'unit_name'], axis=1)
pivoted_test = pd.pivot_table(
    joined_test,
    values='amount',
    index='idx',
    columns='name')

test_w_nutrients = pd.merge(test, pivoted_test, how='left', on='idx')
test_ids = test_w_nutrients[['idx']]
excl = ['alcohol_ethyl_g', 'energy_kj', 'folic_acid_ug', 'molybdenum_mo_ug',␣
 ↪'starch_g']
test_col_drop = [e for e in columns_to_drop[:len(columns_to_drop)-2] if e not␣
 ↪in excl]
test_final = clean_text_values(test_w_nutrients.drop(test_col_drop, axis=1)).
 ↪set_index('idx')
test_numerical = get_final_matrix(test_final, feature_strings_dict, brands_ohe,␣
 ↪corr_columns_to_drop, chosen_imputer)

preds = {}
for name, estimator in final_estimators_dict.items():
    print("Predicting {}".format(name))
    preds[name] = estimator.predict(test_numerical)
    print("Finished predicting {}".format(name))
```

```
Predicting xgbc
Finished predicting xgbc
Predicting rfc
Finished predicting rfc
Predicting svmc
Finished predicting svmc
```

```
[ ]: def decode(n):
  decode_dict = {
      0: 'cakes_cupcakes_snack_cakes', 1: 'candy', 2: 'chocolate', 3:
 ↪'cookies_biscuits',
```

```
      4: 'popcorn_peanuts_seeds_related_snacks', 5:'chips_pretzels_snacks'
  }
  return decode_dict[n]

preds_dfs = {name: pd.concat([test_ids, pd.DataFrame(pred)], axis=1) for name,␣
 ↪pred in preds.items()}
for name in preds_dfs.keys():
  preds_dfs[name] = preds_dfs[name].rename(columns={0: 'category_enc'})
  preds_dfs[name]['pred'] = preds_dfs[name]['category_enc'].apply(decode)
  preds_dfs[name] = preds_dfs[name].drop(['category_enc'], axis=1)
  preds_dfs[name].to_csv(path_or_buf='data/pred_{}.csv'.format(name),␣
 ↪columns=['idx', 'pred'], index=False)
```