Analysis of Food.com Recipes and Reviews

Name(s): Rohan Marda, Mianzhi Hu

Click here to view our website!

In [1]:
```
import pandas as pd
import numpy as np
from pathlib import Path

import plotly.express as px
pd.options.plotting.backend = 'plotly'

import plotly.io as pio
pio.renderers.default = "vscode"

from dsc80_utils import * # Feel free to uncomment and use this.
```

Step 1: Introduction

Project Overview We analyze Food.com recipes (2008–2018) and their user reviews to understand what makes recipes popular and how content features relate to preparation time and user satisfaction.

Dataset We use two datasets:

1. `RAW_recipes.csv` : recipes with metadata (name, tags, ingredients, steps, nutrition, minutes, submission date, etc.).
2. `RAW_interactions.csv` : user interactions with ratings and free-text reviews linked to recipes.

Research Questions

1. Do nutritional attributes—specifically protein content—correspond to meaningful differences in calorie distributions among recipes?
2. Which recipe features most strongly determine preparation time, and how accurately can we predict cooking duration using these features?

In [2]:
```
interactions = pd.read_csv('RAW_interactions.csv')
recipes = pd.read_csv('RAW_recipes.csv')
```

In [3]:
```
merged = recipes.merge(interactions, how='left', left_on = 'id', right_on='recipe_id')
merged['rating'] = merged['rating'].replace(0, np.nan)

avg_rating = merged.groupby('id')['rating'].mean()

data = merged.set_index('id')
data['avg_rating'] = avg_rating

# drop duplicate columns
data = data.drop(columns=['recipe_id'])

data
```

Out[3]:

id	name	minutes	contributor_id	submitted	...	date	rating	review	avg_rating
333281	1 brownies in the world best ever	40	985201	2008-10-27	...	2008-11-19	4.0	These were pretty good, but took forever to ba...	4.0
453467	1 in canada chocolate chip cookies	45	1848091	2011-04-11	...	2012-01-26	5.0	Originally I was gonna cut the recipe in half ...	5.0
306168	412 broccoli casserole	40	50969	2008-05-30	...	2008-12-31	5.0	This was one of the best broccoli casseroles t...	5.0
...	...	...	...	...	...	...	...	...	...
298509	cookies by design sugar shortbread cookies	20	506822	2008-04-15	...	2008-06-19	1.0	This recipe tastes nothing like the Cookies by...	3.0
298509	cookies by design sugar shortbread cookies	20	506822	2008-04-15	...	2010-02-08	5.0	yummy cookies, i love this recipe me and my sm...	3.0
298509	cookies by design sugar shortbread cookies	20	506822	2008-04-15	...	2014-11-01	NaN	I work at a Cookies By Design and can say this...	3.0

234429 rows × 16 columns

In [4]:
```
data.shape
```

Out[4]:
```
(234429, 16)
```

In [5]:
```
data.dtypes
```

Out[5]:
```
name              object
minutes            int64
contributor_id     int64
                   ...
rating           float64
review            object
avg_rating       float64
Length: 16, dtype: object
```

The aggregated recipe and interaction dataset has 234,429 entries, each measuring 16 features:

- `name` : Recipe name
- `minutes` : Preparation time in minutes
- `contributor_id` : Unique contributor ID
- `submitted` : Submission date
- `tags` : Recipe tags (e.g., "dessert", "vegetarian")
- `nutrition` : Nutritional information (calories, protein, fat, sodium, etc)
- `n_steps` : Number of preparation steps
- `steps` : List of preparation steps
- `ingredients` : List of ingredients
- `n_ingredients` : Number of ingredients
- `user_id` : Unique user ID
- `date` : Review date
- `review` : Free-text review
- `rating` : User rating (1-5 stars)
- `avg_rating` : Average recipe rating

In [6]:
```
data.describe()
```

Out[6]:

	minutes	contributor_id	n_steps	n_ingredients	user_id	rating	avg_rating
count	2.34e+05	2.34e+05	234429.00	234429.00	2.34e+05	219393.00	231652.00
mean	1.07e+02	1.24e+07	10.02	9.07	2.22e+08	4.68	4.68
std	3.29e+03	1.48e+08	6.44	3.82	6.19e+08	0.71	0.50
...	...	...	...	...	...	...	...
50%	3.50e+01	4.47e+05	9.00	9.00	4.97e+05	5.00	4.86
75%	6.00e+01	7.77e+05	13.00	11.00	1.30e+06	5.00	5.00
max	1.05e+06	2.00e+09	100.00	37.00	2.00e+09	5.00	5.00

8 rows × 7 columns

Step 2: Data Cleaning and Exploratory Data Analysis

Extract Nutrition Values

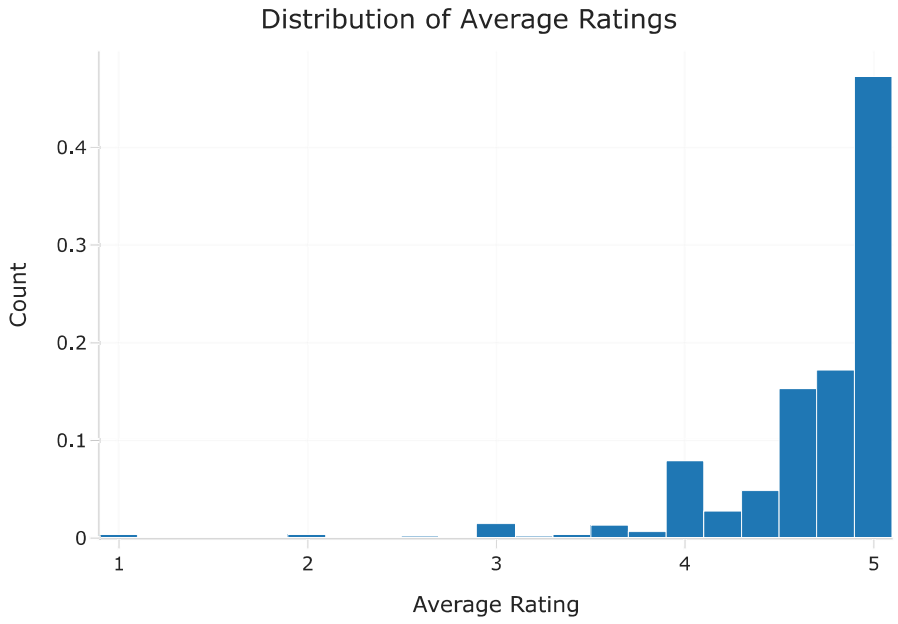
In [7]:
```
nutrition = data["nutrition"].str.strip("[]").str.split(",", expand=True).astype(float)
nutrition.columns = ["calories","total_fat","sugar","sodium","protein","saturated_fat","carbs"]
data = pd.concat([data, nutrition], axis=1)

print(f'Updated dataset shape: {data.shape}')
```

```
Updated dataset shape: (234429, 23)
```

Univariate Analysis

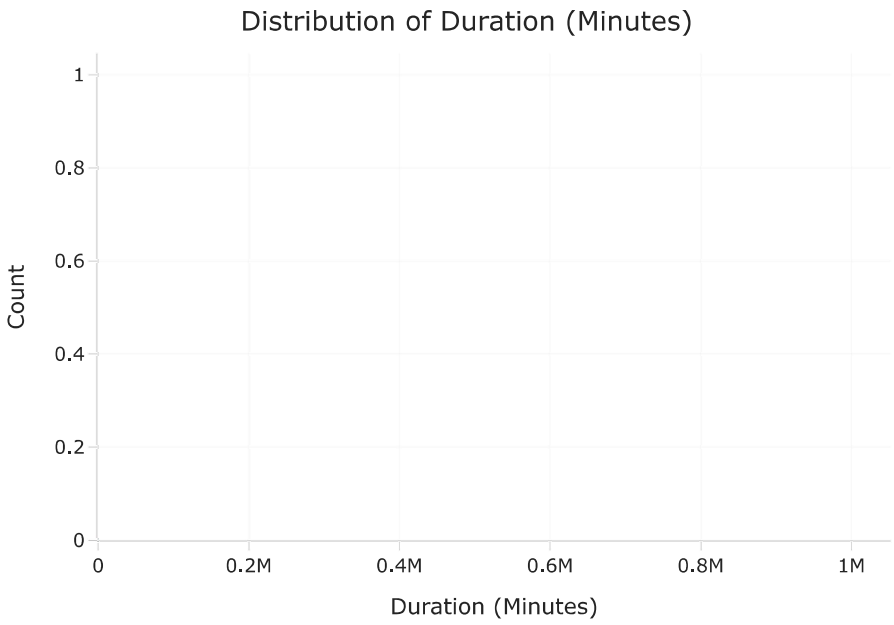
- Histograms

In [8]:
```
fig = px.histogram(
    data_frame=data,
    x='avg_rating',
    nbins=30,
    title='Distribution of Average Ratings',
    histnorm='probability'
)

fig.update_layout(
    xaxis_title="Average Rating",
    yaxis_title="Count",
)
fig.show()
```



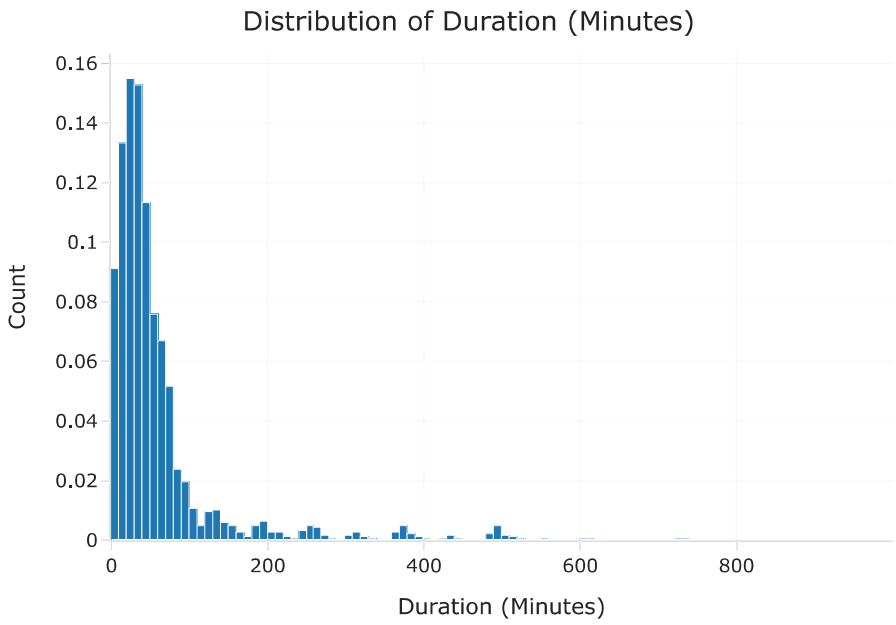
In [9]:
```
fig = px.histogram(
    data_frame=data,
    x='minutes',
    title='Distribution of Duration (Minutes)',
    histnorm='probability',
)
```

```
fig.update_layout(
    xaxis_title="Duration (Minutes)",
    yaxis_title="Count",
)

fig.update_traces(
    xbins=dict(
        size=1000
    )
)

fig.show()
```



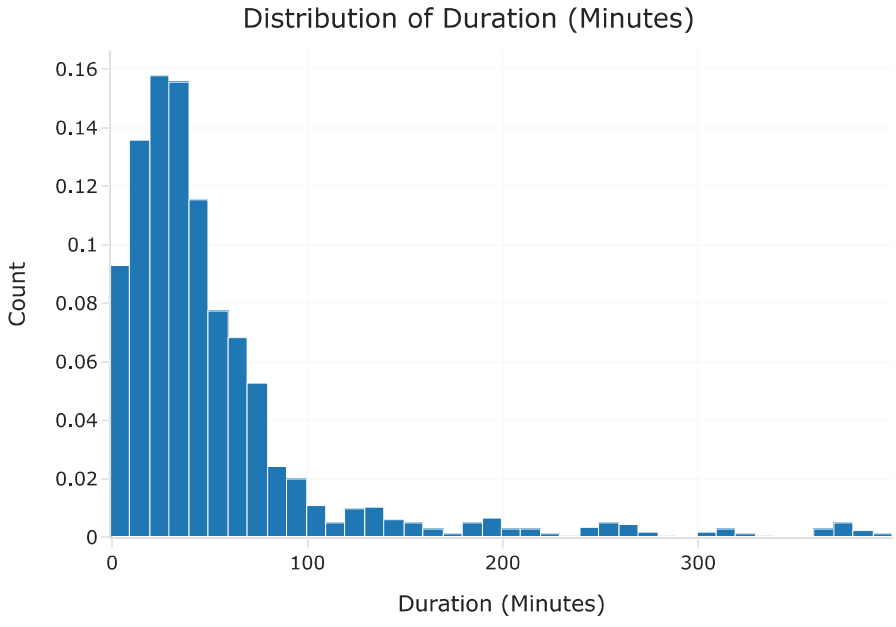
almost 99.9% of recipes take <1000 mins to prepare (almost invisible in plot)

In [10]:
```
fig = px.histogram(
    data_frame=data[data['minutes'] < 1000],
    x='minutes',
    title='Distribution of Duration (Minutes)',
    histnorm='probability'
)

fig.update_layout(
    xaxis_title="Duration (Minutes)",
    yaxis_title="Count",
)

fig.update_traces(
    xbins=dict(
        size=10
    )
)

fig.show()
```



From this plot, we can see that most recipes take less than 400 minutes to prepare. There are a few outliers that take a very long time to prepare, but they are not representative of the majority of recipes, and thus we will be removing them from our dataset.

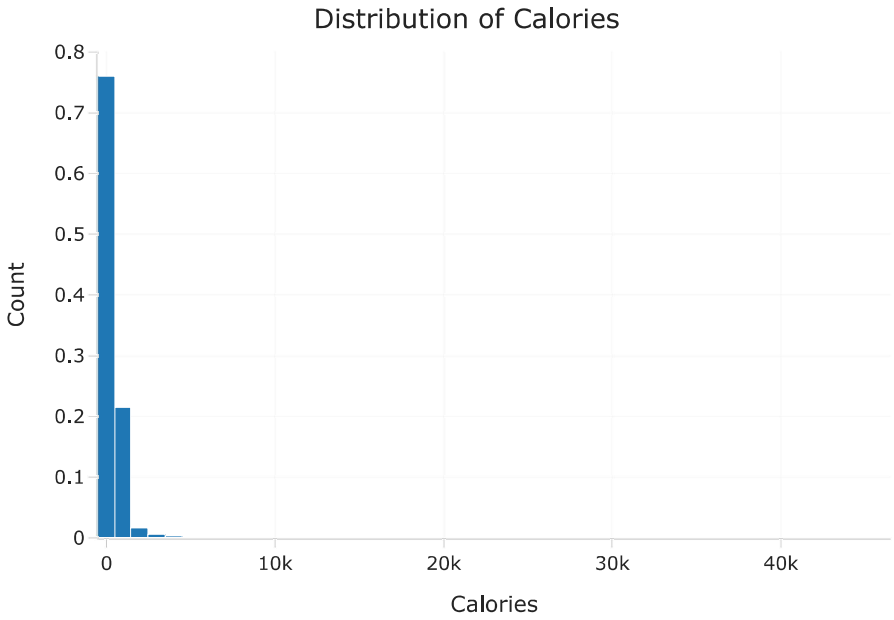
In [11]:
```
data = data[data['minutes'] < 400]
fig = px.histogram(
    data_frame=data,
    x='minutes',
    title='Distribution of Duration (Minutes)',
    histnorm='probability',
)

fig.update_layout(
    xaxis_title="Duration (Minutes)",
    yaxis_title="Count",
)

fig.update_traces(
    xbins=dict(
        size=10
    )
)
fig.show()

print(f'Updated dataset shape: {data.shape}')
```




```
Updated dataset shape: (228374, 23)
```

In [12]:
```
fig = px.histogram(
    data_frame=data,
    x='calories',
    title='Distribution of Calories',
    histnorm='probability',
```

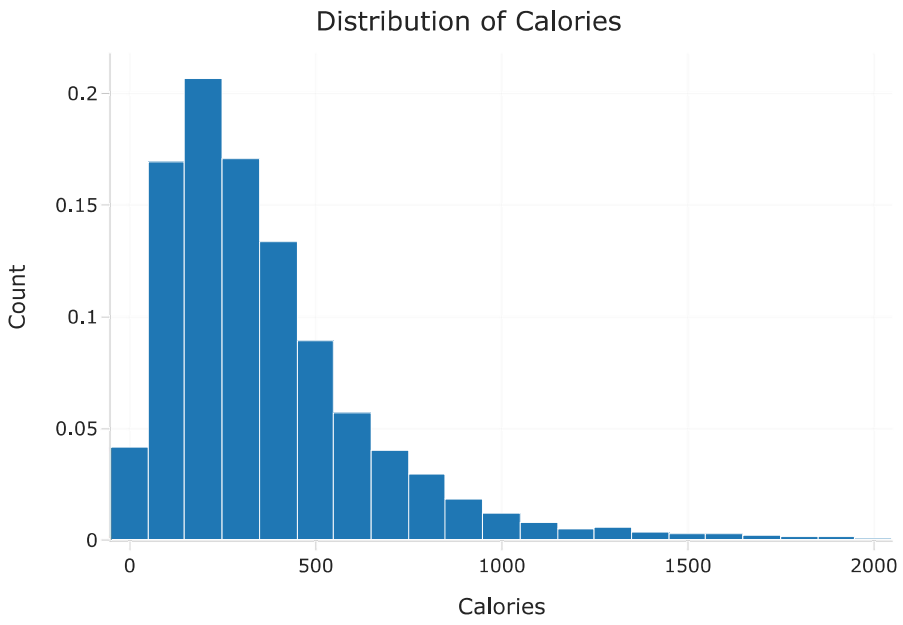
```
)

fig.update_layout(
    xaxis_title="Calories",
    yaxis_title="Count",
)

fig.update_traces(
    xbins=dict(
        size=1000
    )
)

fig.show()
```



The histogram of recipe calories also reveals the presence of outliers. Most recipes have calories ranging from 0 to 2000, with a significant concentration between 0 and 1000 calories. However, there are some recipes with extremely high calorie counts, reaching up to 8000 calories. Also, some recipes report negative calories, which is an error. These outliers can skew the analysis and may not accurately represent the typical recipe in the dataset. Therefore, we will also be removing these outliers* to ensure a more accurate analysis of the data.

***also based on the fact that the reccommended daily calorie intake for a typical adult is around 2000 calories.**

In [13]:
```
data = data[(data['calories'] < 2000) & (data['calories'] >= 0)]

fig = px.histogram(
    data_frame=data,
    x='calories',
    title='Distribution of Calories',
    histnorm='probability',
)

fig.update_layout(
    xaxis_title="Calories",
    yaxis_title="Count",
)

fig.update_traces(
    xbins=dict(
        size=100
    )
)

fig.show()

print(f'Updated dataset shape: {data.shape}')
```




```
Updated dataset shape: (224755, 23)
```

In [14]:
```
data[['total_fat', 'sugar', 'sodium', 'protein', 'saturated_fat', 'carbs']].describe()
```

Out[14]:

	total_fat	sugar	sodium	protein	saturated_fat	carbs
count	224755.00	224755.00	224755.00	224755.00	224755.00	224755.00
mean	27.86	50.56	26.49	30.66	34.60	11.18
std	30.24	90.07	85.23	34.24	42.51	12.13
...	...	...	...	...	...	...
50%	19.00	21.00	14.00	17.00	21.00	8.00
75%	37.00	55.00	32.00	47.00	48.00	15.00
max	340.00	1925.00	4727.00	516.00	648.00	165.00

8 rows × 6 columns

Nutrional values provided are in Percent Daily Values (PDV) based on a 2,000 calorie diet. The table above summarizes the nutritional values for total fat, sugar, sodium, protein, saturated fat, and carbs across all recipes in the dataset - and reveals that some recipes have extreme outliers in nutritional values as well. For example, the maximum sugar content is 2000% of the daily recommended value, which is not realistic for a single recipe. Similarly, the maximum sodium content is nearly 5000% of the daily recommended value. These extreme values can skew the analysis and may not accurately represent the typical recipe in the dataset. Therefore, we will be removing these outliers to ensure a more accurate analysis of the data.

In [15]:
```
data = (data[
    (data['total_fat'] <= 100) &
    (data['sugar'] <= 100) &
    (data['sodium'] <= 100) &
    (data['protein'] <= 100) &
    (data['saturated_fat'] <= 100) &
    (data['carbs'] <= 100)
])

data[['total_fat', 'sugar', 'sodium', 'protein', 'saturated_fat', 'carbs']].describe()

print(f'Updated dataset shape: {data.shape}')
```

```
Updated dataset shape: (173222, 23)
```

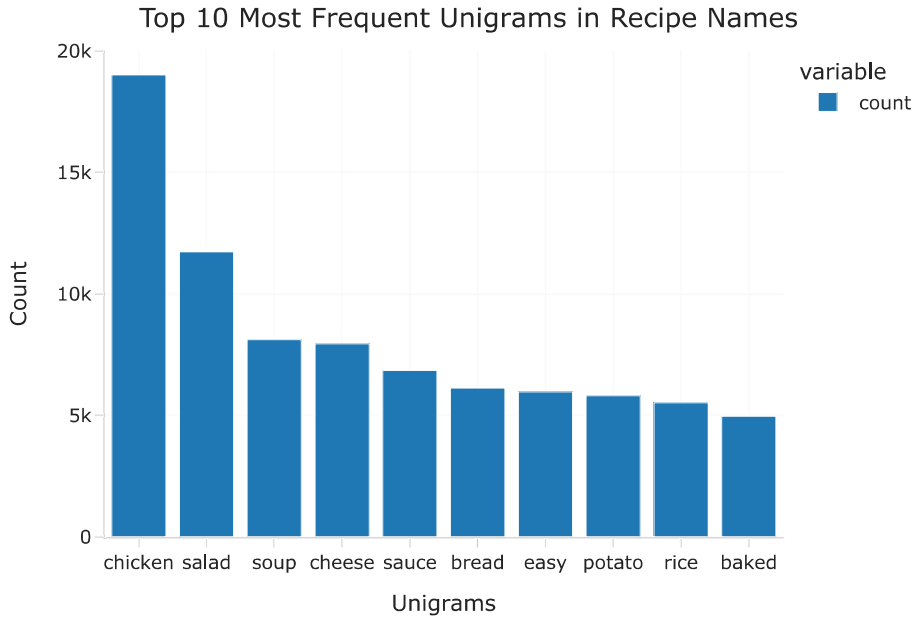
- Analysis of Recipe Names

In [ ]:
```
# define stop words that we want to avoid
stop_words = [
    "the", "a", "an", "and", "or", "with", "without", "from", "of",
    "for", "to", "in", "on", "by", "s",
    "this", "that", "these", "those", "it", "its", "is", "are", "was", "were",
    "be", "been", "being", "at", "as", "but", "if", "then", "so", "too",
    "can", "could", "may", "might", "will", "would", "shall", "should",
    "i", "you", "he", "she", "we", "they", "me", "him", "her", "us", "them",
    "my", "your", "his", "her", "our", "their", "mine", "yours", "hers",
```

```
    "ours", "theirs", "no", "not", "do", "does", "did", "done", "have", "has", "had",
    "up", "down", "over", "under", "again", "once", "here", "there", "when", "where", "why", "how", "like",
    "use", "very", "also", "great", "also", "great"
]

# analyze the most repeated words in name:
word_cleaner = lambda x: x.lower().strip()

# filter out all words present in the recipe names
all_split_names = data['name'].apply(lambda x: str(x).split())
all_unigrams_in_names = (
    pd.Series([word_cleaner(word) for name in all_split_names for word in name if word not in stop_words])
)
all_unigrams_in_names
```

Out[ ]:
```
0                1
1          brownies
2             world
             ...
718872        sugar
718873    shortbread
718874       cookies
Length: 718875, dtype: object
```

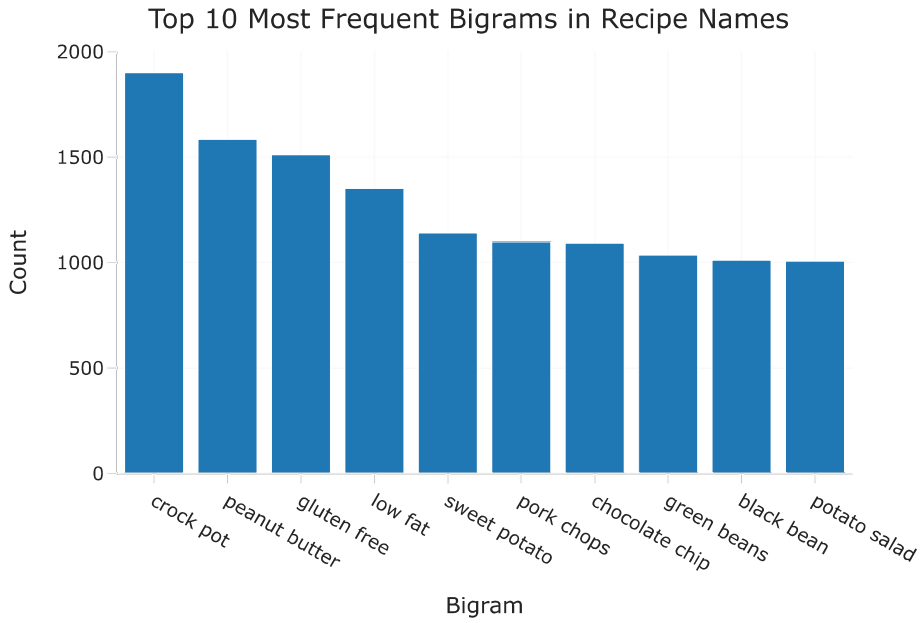
In [17]:
```
# top 10 most common unigrams
unigram_counts = all_unigrams_in_names.value_counts()
top_ten_name_unigrams = unigram_counts.sort_values().nlargest(n = 10)

fig = px.bar(top_ten_name_unigrams, title='Top 10 Most Frequent Unigrams in Recipe Names')

fig.update_layout(
    xaxis_title="Unigrams",
    yaxis_title="Count"
)

fig.show()
```



The barplot of unigram frequencies in recipe names shows that certain words are particularly common. The most frequent unigrams include "chicken," "easy," "best," "sauce," and "quick." This suggests that recipes featuring chicken and those that are easy or quick to prepare are popular among users. Moreover, the prevelance of words like "easy" and "best" could possibly be indicative of recipe duration, which we will explore further in our regression analysis.

In [18]:
```
# analyze the most repeated words in name:
word_cleaner = lambda x: x.lower().strip()

# filter out all words present in the recipe names
all_bigrams = []
for i in range(len(all_unigrams_in_names) - 1):
    all_bigrams.append((all_unigrams_in_names[i], all_unigrams_in_names[i + 1]))

all_bigrams = pd.Series(all_bigrams)

# top 10 most common bigrams
bigram_counts = all_bigrams.value_counts()
top_ten_name_bigrams = bigram_counts.nlargest(n = 10)  # already sorted descending
```

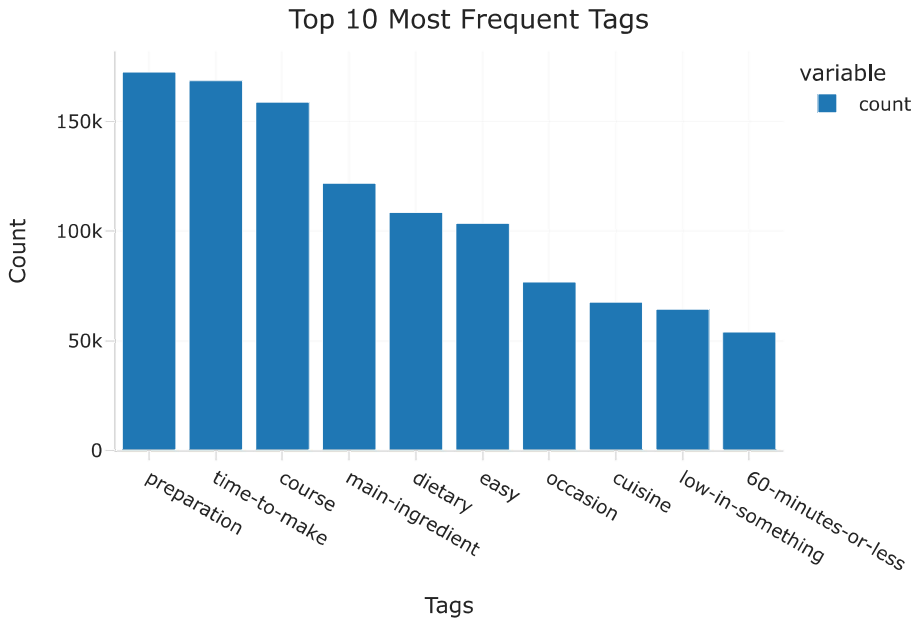
```
bigram_df = top_ten_name_bigrams.reset_index()
bigram_df.columns = ['bigram', 'count']
bigram_df['bigram_str'] = bigram_df['bigram'].apply(lambda x: f"{x[0]} {x[1]}")

fig = px.bar(
    bigram_df,
    x='bigram_str',
    y='count',
    title="Top 10 Most Frequent Bigrams in Recipe Names",
    labels={'bigram_str':'Bigram', 'count':'Count'}
)

fig.show()
```



Since a unigram analysis only captures individual words, we also perform a bigram analysis to identify common ordered two-word phrases in recipe names. The most frequent bigrams include "crock pot," "peanut butter," and "gluten free." These phrases reinforce the findings from the unigram analysis, highlighting that users often search for recipes that are easy to prepare and feature popular ingredients like chicken and chocolate chips.

- Analysis of Tags

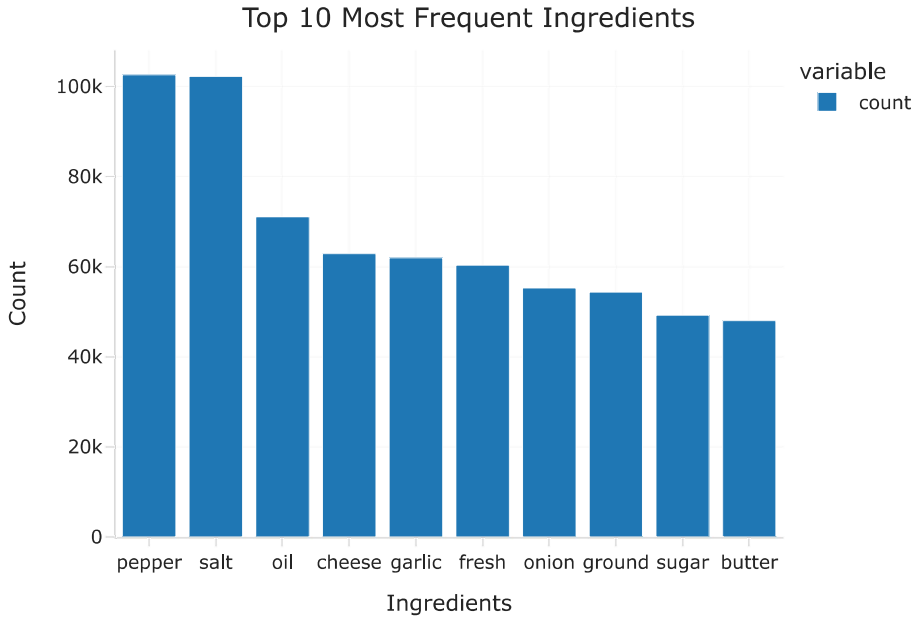
In [19]:
```
# analyze the most repeated words in name:
tag_cleaner = lambda x: x.replace('\'', '').replace('[', '').replace(']', '').replace(',', '')

cleaned_tags = data['tags'].apply(tag_cleaner)
final_tags = pd.Series([word for tag in cleaned_tags for word in tag.split()])

tag_counts = final_tags.value_counts()
top_ten_tags = tag_counts.sort_values().nlargest(n = 10)

fig = px.bar(top_ten_tags, title='Top 10 Most Frequent Tags')

fig.update_layout(
    xaxis_title="Tags",
    yaxis_title="Count"
)

fig.show()
```



Similarly, the barplot of tag frequencies reveals that certain tags are particularly prevalent among recipes. Common tags are descrptive of recipe duration and could serve as a useful feature for predicting preparation time. For example, tags like "easy," "quick," and "30-minutes-or-less" suggest that these recipes are designed to be prepared in a short amount of time. Other frequent tags such as "healthy," "vegetarian," and "desserts" provide additional context about the type of recipe, which may also influence preparation time.

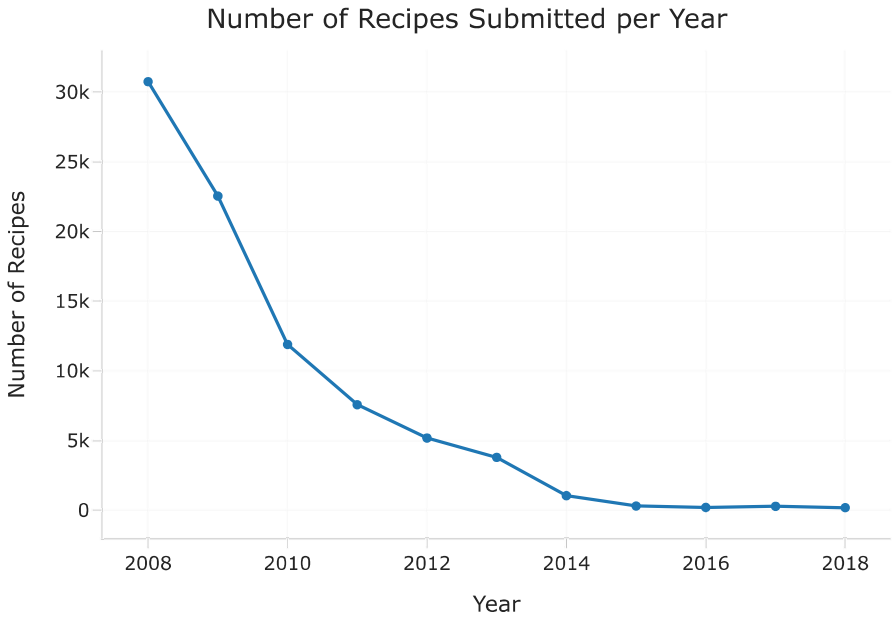
- Analysis of Ingredients

In [ ]:
```
# analyze the most repeated words in name:
steps_cleaner = lambda x: str(x).replace('\'', '').replace('[', '').replace(']', '').replace(',', '')

cleaned_ingredients = data['ingredients'].apply(steps_cleaner)
cleaned_descriptions_ingredients = (
    pd.Series([word for step in cleaned_ingredients for word in step.split() if word not in stop_words])
)

ingredients_counts = cleaned_descriptions_ingredients.value_counts()
top_ten_ingredients = ingredients_counts.sort_values().nlargest(n = 10)

fig = px.bar(top_ten_ingredients, title='Top 10 Most Frequent Ingredients')

fig.update_layout(
    xaxis_title="Ingredients",
    yaxis_title="Count"
)

fig.show()
```



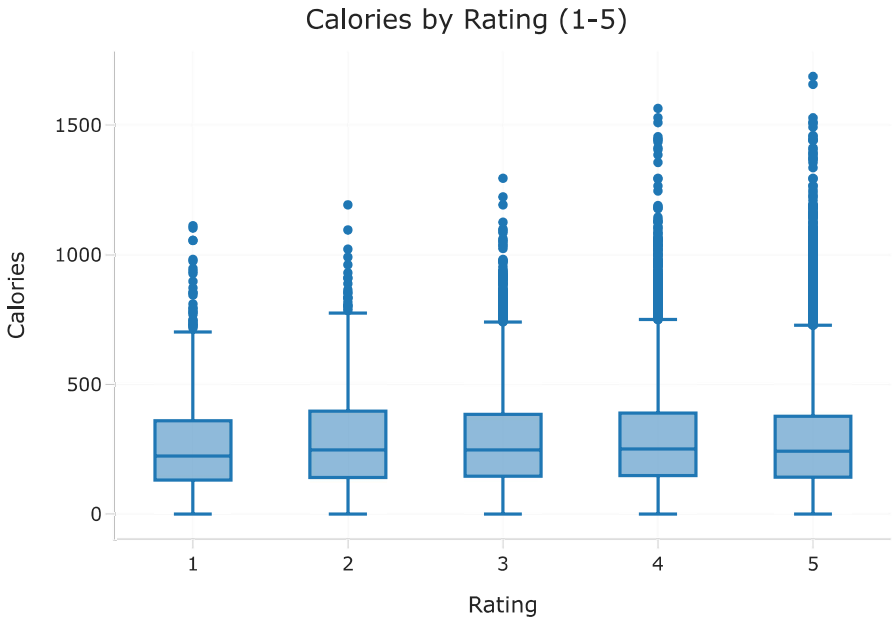
Bivariate Analysis

In [ ]:
```
# count recipes submitted per year and plot a line chart
submitted_dt = pd.to_datetime(recipes['submitted'], errors='coerce')
year_counts = (submitted_dt.dt.year
               .value_counts()
               .sort_index()
               .rename_axis('year')
               .reset_index(name='count'))

fig = px.line(year_counts, x='year', y='count', markers=True,
              title='Number of Recipes Submitted per Year')
fig.update_layout(xaxis_title='Year', yaxis_title='Number of Recipes')
fig.show()
```



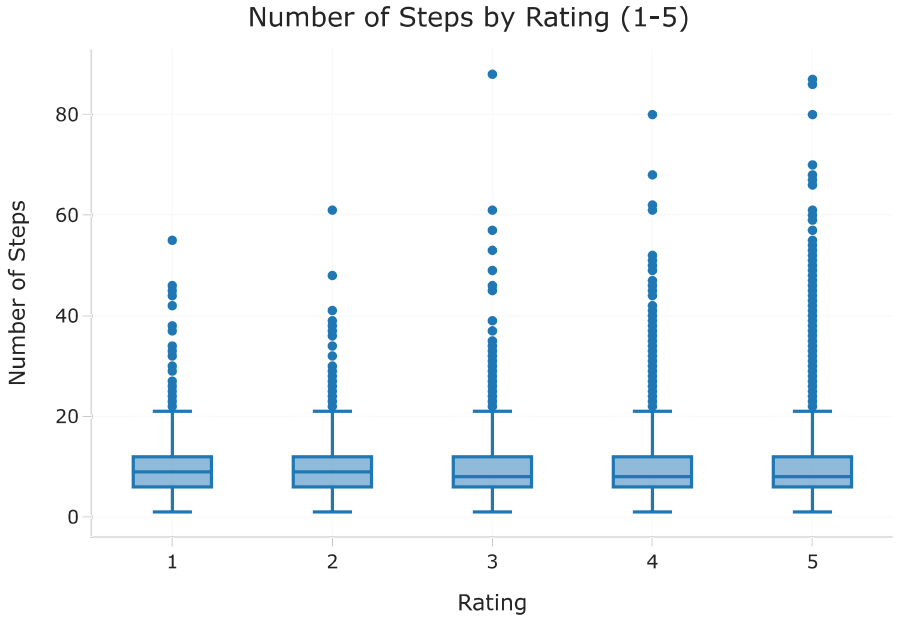
The line chart of recipes submitted per year shows a clear downward trend from 2008 to 2018. The number of recipes submitted decreases exponentially each year, indicating declining user engagement on the Food.com platform. This trend suggests that fewer users were contributing recipes over time, possibly due to decreased popularity of the site or reduced interest in cooking and sharing recipes online.

In [ ]:
```
df_plot = data[['rating', 'calories']]
fig = px.box(
    df_plot,
    x='rating',
    y='calories',
    title='Calories by Rating (1-5)',
    labels={'rating': 'Rating', 'calories': 'Calories'},
    points='outliers'
)
fig.update_layout(xaxis=dict(categoryorder='array',
                             categoryarray=['1','2','3','4','5']))
fig.show()
```

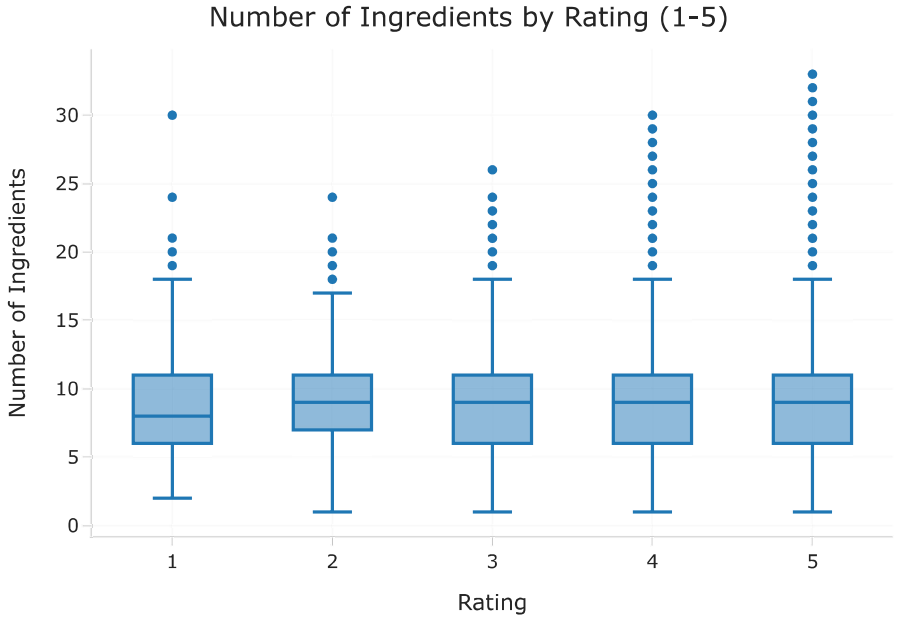


A boxplot of calories stratifed by rating reveals no significant relationship between recipe calories and user ratings. The median calories remain relatively constant across different rating levels, and there is no clear trend indicating that higher-rated recipes have more or fewer calories. This suggests that calorie content does not play a major role in how users rate recipes on Food.com.

In [ ]:
```
df_plot = data[['rating', 'n_steps']]
fig = px.box(
    df_plot,
    x='rating',
    y='n_steps',
    title='Number of Steps by Rating (1-5)',
    labels={'rating': 'Rating', 'n_steps': 'Number of Steps'},
    points='outliers'
)

fig.update_layout(xaxis=dict(categoryorder='array',
                             categoryarray=['1','2','3','4','5']))
fig.show()
```



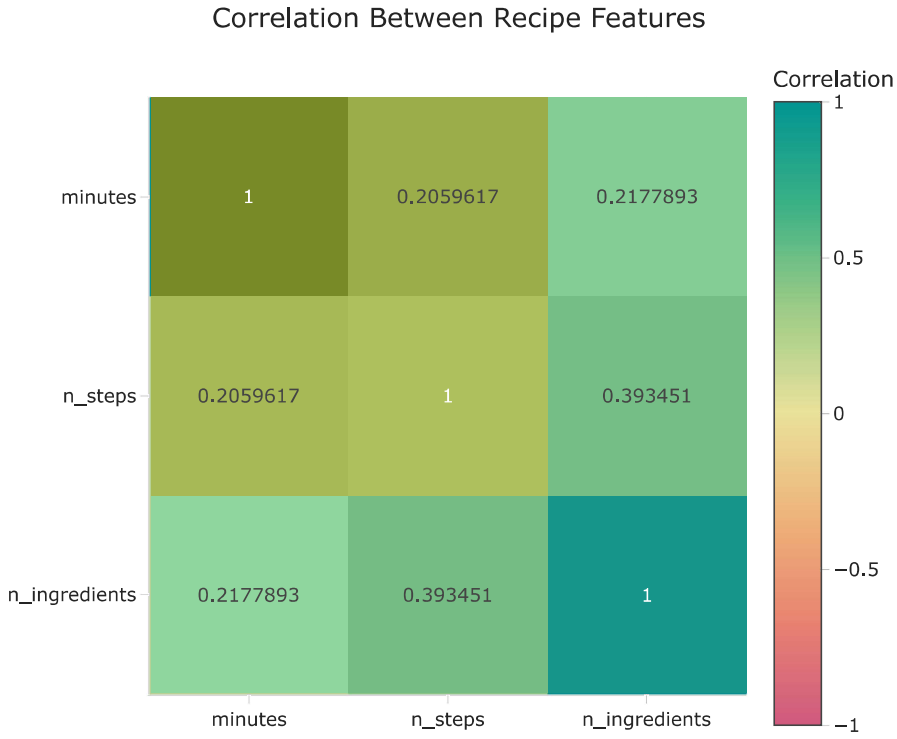
In [ ]:
```
df_plot = data[['rating', 'n_ingredients']]
fig = px.box(
    df_plot,
    x='rating',
    y='n_ingredients',
    title='Number of Ingredients by Rating (1-5)',
    labels={'rating': 'Rating', 'n_ingredients': 'Number of Ingredients'},
    points='outliers'
)
fig.update_layout(xaxis=dict(categoryorder='array',
                             categoryarray=['1','2','3','4','5']))
fig.show()
```



A similar analysis of ratings by number of steps and number of ingredients also shows no significant relationship. The median ratings remain relatively stable across different levels of steps and ingredients, indicating that the complexity of a recipe does not strongly influence user satisfaction.

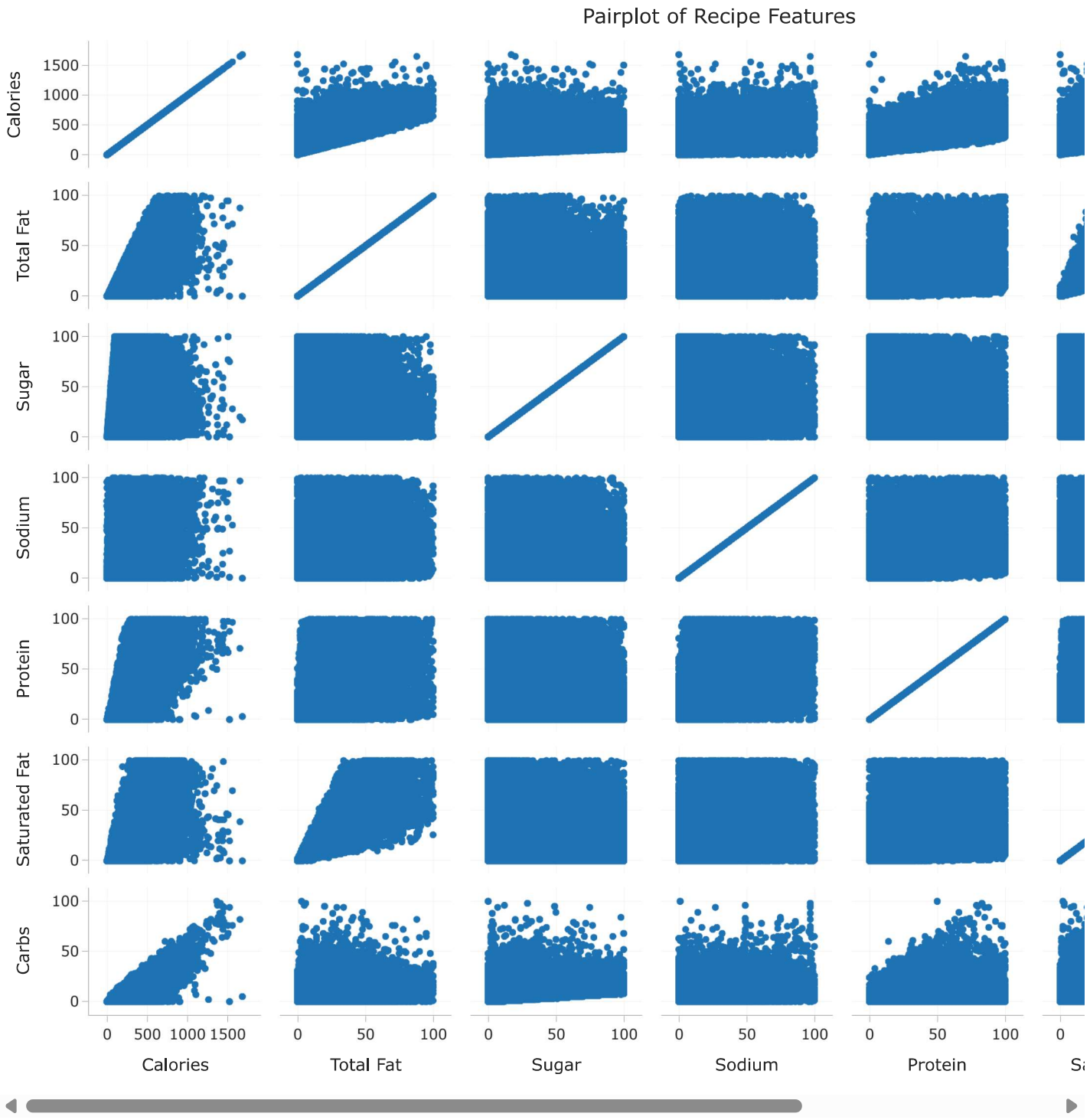
Multivariate Analysis

In [25]:
```
# select numeric columns only
numeric_features = data[['minutes', 'n_steps', 'n_ingredients']]

# create correlation matrix and plot with plotly express
corr = numeric_features.corr()

fig = px.imshow(
    corr,
    text_auto=True,
    color_continuous_scale='temps_r',
    zmin=-1, zmax=1,
    labels=dict(x='', y='', color='Correlation'),
    x=corr.columns,
    y=corr.index,
    title='Correlation Between Recipe Features'
)

fig.update_layout(width=600, height=500)
fig.show()
```



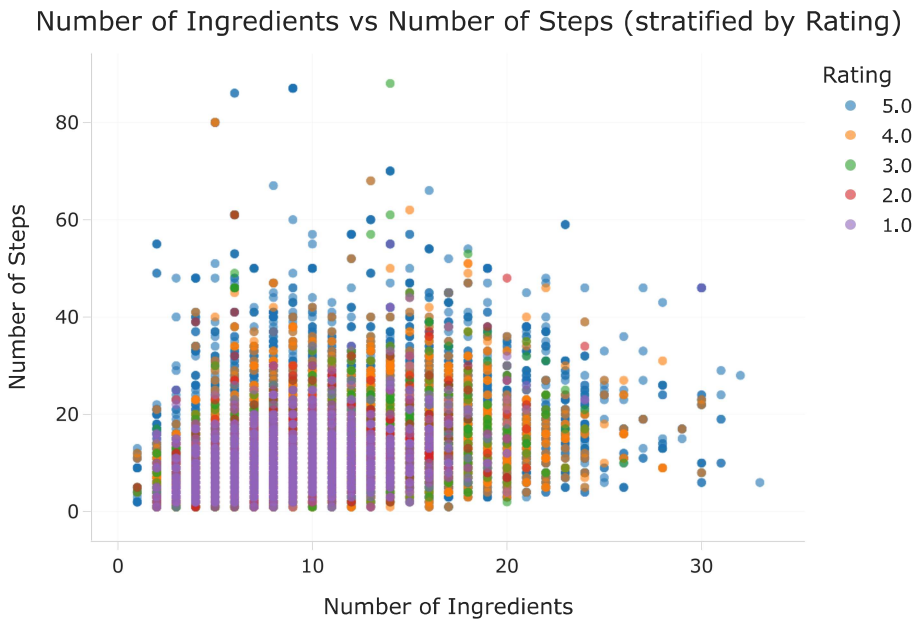
The correlation matrix reveals some notable relationships between features. There is a moderate positive correlation between the number of ingredients and the number of steps (0.39), suggesting that more complex recipes tend to have both more ingredients and more steps. However, there is a weaker correlation between minutes and steps/ingredients, which is likely represntative of the high variance in preparation times for recipes with similar complexity.

In [ ]:
```
fig = px.scatter_matrix(
    data[['calories', 'total_fat', 'sugar', 'sodium', 'protein', 'saturated_fat', 'carbs']],
    dimensions=['calories', 'total_fat', 'sugar', 'sodium', 'protein', 'saturated_fat', 'carbs'],
    title='Pairplot of Recipe Features',
    labels=(
        {col: col.replace('_', ' ').title() for col in ['calories', 'total_fat', 'sugar', 'sodium', 'protein', 'saturated_fa
    )
)

fig.update_layout(width=1200, height=900)
fig.show()
```



Nutritional attributes such as calories, protein, fat, and carbs also show moderate correlations with each other, indicating that recipes higher in one nutritional component tend to be higher in others as well. Overall, while some features are correlated, the relationships are not strong enough to suggest redundancy, indicating that each feature provides unique information about the recipes.

In [27]:
```
# plot ingredients and steps, stratified by rating
df_scatter = data[['n_ingredients', 'n_steps', 'rating']].dropna()
df_scatter['rating_str'] = df_scatter['rating'].astype(str)

fig = px.scatter(
    df_scatter,
    x='n_ingredients',
    y='n_steps',
    color='rating_str',
    title='Number of Ingredients vs Number of Steps (stratified by Rating)',
    labels={'n_ingredients': 'Number of Ingredients', 'n_steps': 'Number of Steps', 'rating_str': 'Rating'},
    opacity=0.6,
    hover_data=['rating'],
    category_orders={
    'rating_str': ['5.0', '4.0', '3.0', '2.0', '1.0']
}
)

fig.update_traces(marker=dict(size=6))
fig.show()
```



Step 3: Assessment of Missingness

In [28]:
```
data.isna().sum()
```

Out[28]:
```
name              0
minutes           0
contributor_id    0
                 ..
protein           0
saturated_fat     0
carbs             0
Length: 23, dtype: int64
```

There are a few missing values in the `description` and `review` columns, but there are not significant enough to impact our analysis as they are relatively few in number, and we will chose to drop them. Therefore, we will be focusing on the missing values in the `avg_rating` column for our assessment of missingness, which has ~2,000 missing values. Note that though there are many missing values in the `rating` column, our analysis uses aggregated average ratings at the recipe level, so we will be focusing on the `avg_rating` column instead.

In [29]:
```
data = data.dropna(subset=['description', 'review'])
```

We conduct a **Missing At Random (MAR)** vs. **Not Missing At Random (NMAR) analysis to reason about whether the missingness of ratings depends on other observable variables in the dataset or on unobserved factors.

We suspect that `avg_rating` may be MAR based on observable recipe features such as preparation time (minutes). Specifically, recipes that are more complex or time-consuming may be less likely to receive ratings, which would make the missingness predictable from these observed variables.

Reasoning and Hypothesis:

If missing ratings are MAR, then the missingness can be explained by other observable features, like minutes or number of ingredients. One possible MAR scenario is that users may be less likely to rate recipes that take a long time to prepare, leading to missing ratings that depend on the preparation time.

If missing ratings are NMAR, then the missingness depends on the value of `avg_rating` itself or other unobserved factors, meaning no observable variable can fully explain why some ratings are missing. One possible NMAR scenario is that users may be less likely to rate recipes they find unsatisfactory, leading to missing ratings that depend on the actual rating value.

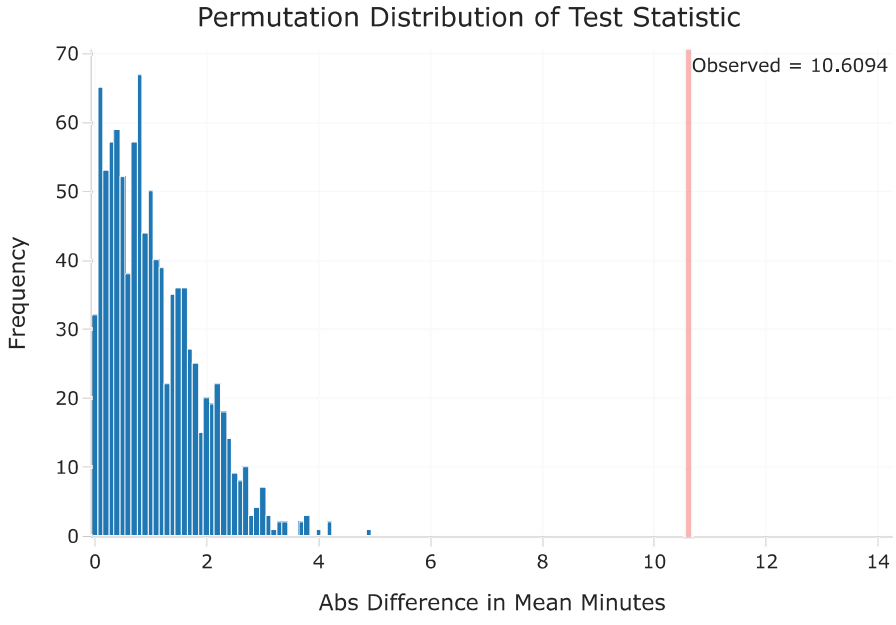
Test Approach (for reasoning about MAR vs NMAR):

1. Create a binary indicator variable for whether avg_rating is missing
2. Compute observed test statistic
3. Permute missingness indicator and compute permuted test statistics
4. Compare observed statistic to permuted distribution to assess significance

In [30]:
```
# 1. Create missing indicator
data['avg_rating_missing'] = data['avg_rating'].isna().astype(int)

df = data[['minutes', 'avg_rating_missing']]
```

```
missing_group = df[df['avg_rating_missing'] == 1]['minutes']
present_group = df[df['avg_rating_missing'] == 0]['minutes']

# 2. Observed statistic
observed_stat = abs(missing_group.mean() - present_group.mean())

# 3. Permutation test
num_permutations = 1000
np.random.seed(42)

feature_values = df['minutes'].values
missingness = df['avg_rating_missing'].values

permuted_stats = []

for _ in range(num_permutations):
    shuffled = np.random.permutation(feature_values)
    perm_missing = shuffled[missingness == 1]
    perm_present = shuffled[missingness == 0]
    permuted_stats.append(abs(perm_missing.mean() - perm_present.mean()))

permuted_stats = np.array(permuted_stats)

# 4. p-value
p_value = (permuted_stats >= observed_stat).mean()

print(f"\nObserved difference in mean minutes: {observed_stat:.4f}")
print(f"P-value: {p_value:.4f}")

# Plot permutation distribution
fig = px.histogram(
    x=permuted_stats,
    nbins=50,
    title="Permutation Distribution of Test Statistic",
    labels={'x': 'Abs Difference in Mean Minutes', 'y': 'Frequency'}
)

fig.add_vline(x=observed_stat, line_width=3, line_color="red", annotation_text=f"Observed = {observed_stat:.4f}")
fig.update_layout(xaxis_title='Abs Difference in Mean Minutes', yaxis_title='Frequency')
fig.show()
```

```
Observed difference in mean minutes: 10.6094
P-value: 0.0000
```



The test yielded a p-value of 0.000, which is less than the significance level of 0.05. Therefore, we reject the null hypothesis and conclude that there is strong evidence that missingness of ratings is related to the duration (`minutes`) of recipes.

Since the permutation test suggests that the missingness of ratings is MAR, it is statisitcally valid to impute the missing ratings using duration. To account for missingness, we will impute missing values with the mean rating per 'minutes' quantile.

In [31]:
```
# Impute missing avg_rating with mean rating per minutes quantile

# Create quantile bins for minutes
data['minutes_quartile'] = pd.qcut(data['minutes'], q=4, labels=['Q1', 'Q2', 'Q3', 'Q4'], duplicates='drop')
```

```
# Calculate mean avg_rating for each quartile (excluding missing values)
quartile_means = data.groupby('minutes_quartile', observed=True)['avg_rating'].transform('mean')

# Impute missing avg_rating values with quartile means
print("Missing values before imputation:", data['avg_rating'].isna().sum())
data['avg_rating'] = data['avg_rating'].fillna(quartile_means)

# Verify imputation
print("Missing values after imputation:", data['avg_rating'].isna().sum())

# Show mean ratings by quartile
print("\nMean avg_rating by minutes quartile:")
print(data.groupby('minutes_quartile', observed=True)['avg_rating'].mean())
```

```
Missing values before imputation: 1839
Missing values after imputation: 0

Mean avg_rating by minutes quartile:
minutes_quartile
Q1    4.71
Q2    4.68
Q3    4.67
Q4    4.66
Name: avg_rating, dtype: float64
```

In [32]:
```
# drop intermediate columns used for imputation & individual recipe ratings
data = data.drop(columns=['minutes_quartile', 'avg_rating_missing', 'rating'])

print(f'Final dataset shape after cleaning, filtering, and imputation: {data.shape}')
```

```
Final dataset shape after cleaning, filtering, and imputation: (173088, 22)
```

In [33]:
```
# confirm no missing values remain
data.isna().sum().sum()
```

Out[33]: np.int64(0)

Step 4: Hypothesis Testing

We investigate whether recipes with higher protein content have the same calorie distribution as those with lower protein content.

After removing unrealistic outliers (recipes containing more than 100% of the recommended daily protein intake), we find that the median protein level is 16% of daily value. We therefore categorize recipes into:

- **High-protein recipes:** protein ≥ 16%
- **Low-protein recipes:** protein < 16%

Hypothesis:

- **Null Hypothesis (H_0):** High-protein and low-protein recipes have the same calorie distribution.
- **Alternative Hypothesis (H_1):** The calorie distributions differ between high-protein and low-protein recipes.

Test Approach: We use a **two-sample Kolmogorov–Smirnov (K-S) test** to compare the two groups. The K-S test measures the maximum difference between their empirical cumulative distribution functions (CDFs) and is appropriate here because:

1. It is fully non-parametric.
2. It compares entire distributions rather than just averages.
3. It is sensitive to differences in both the shape and location of the distributions.

This approach allows us to assess whether protein content is associated with a meaningful shift in calorie distribution.

In [34]:
```
from scipy.stats import ks_2samp

high = data.loc[data["protein"] > 16, "calories"]#.sort_values(ascending=False)
low = data.loc[data["protein"] <= 16, "calories"]#.sort_values(ascending=False)
stat, p = ks_2samp(high, low)
conclusion = "reject H0" if p < 0.05 else "accept H0"
print(f"p = {p}", conclusion)
```

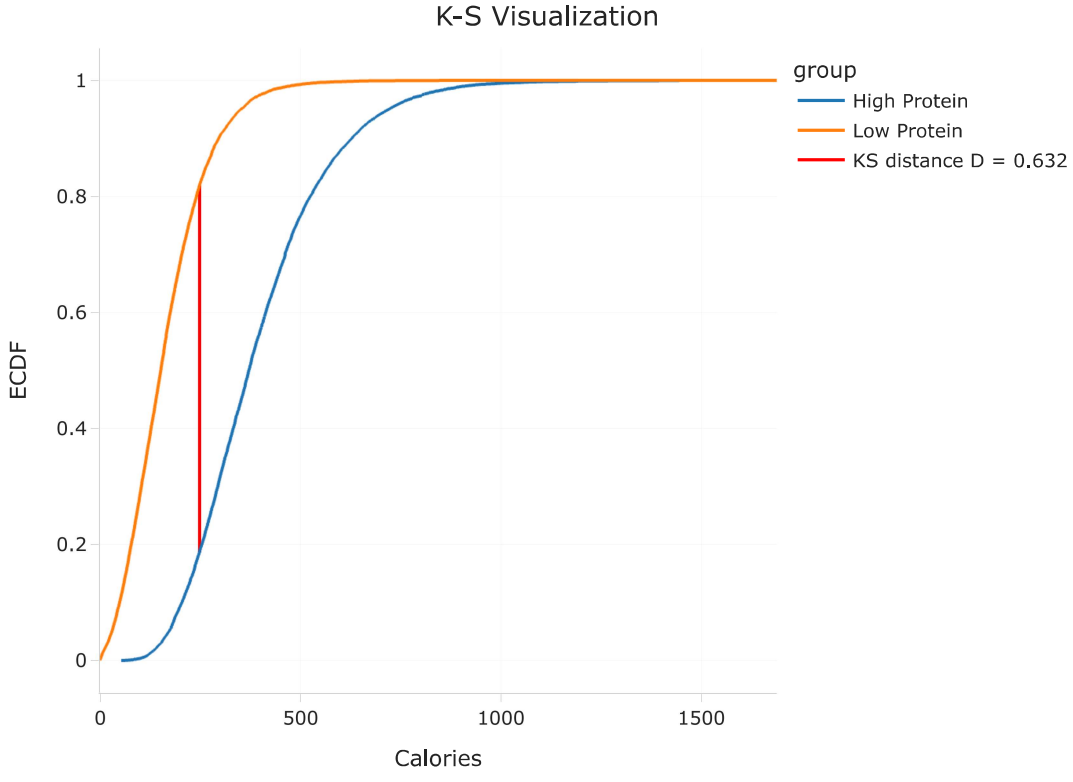
```
p = 0.0 reject H0
```

Based on the results of the K-S test, we reject the null hypothesis, indicating that high-protein and low-protein recipes do **not** share the same calorie distribution.

Interpretation: The difference in distributions suggests that protein content is associated with meaningful variation in calorie levels. High-protein recipes may tend to cluster around different calorie ranges than low-protein ones.

Visualization Approach: To better understand how the two groups differ, we visualize their empirical cumulative distribution functions (CDFs):

In [35]:
```
def ecdf(data):
    data_sorted = np.sort(data)
    yvals = np.arange(1, len(data) + 1) / len(data)
    return data_sorted, yvals

# Calculate ECDFs
high_sort, high_ecdf = ecdf(high)
low_sort, low_ecdf = ecdf(low)

# Create DataFrame for plotting
df_high = pd.DataFrame({'calories': high_sort, 'ecdf': high_ecdf, 'group': 'High Protein'})
df_low = pd.DataFrame({'calories': low_sort, 'ecdf': low_ecdf, 'group': 'Low Protein'})
df_combined = pd.concat([df_high, df_low])

# Plot ECDFs
fig = px.line(
    df_combined,
    x='calories',
    y='ecdf',
    color='group',
    title='K-S Visualization',
    labels={'calories': 'Calories', 'ecdf': 'ECDF'},
    line_shape='hv'
)

# Calculate KS distance line
all_points = np.sort(np.concatenate([high, low]))
high_cdf = np.searchsorted(high_sort, all_points, side='right') / len(high)
low_cdf = np.searchsorted(low_sort, all_points, side='right') / len(low)
idx_max = np.argmax(np.abs(high_cdf - low_cdf))

# Add vertical line for KS distance
fig.add_scatter(
    x=[all_points[idx_max], all_points[idx_max]],
    y=[min(high_cdf[idx_max], low_cdf[idx_max]), max(high_cdf[idx_max], low_cdf[idx_max])],
    mode='lines',
    line=dict(color='red', width=2),
    name=f'KS distance D = {stat:.3f}'
)

fig.update_layout(width=700, height=500)
fig.show()
```



Step 5: Framing a Prediction Problem

We aim to predict the duration of a recipe, using the values in the `minutes` column as our target variable. Because minutes is a continuous numerical variable, this is a regression problem.

Our initial predictors include `n_steps` (the number of steps in the recipe) and ingredient counts, which we expect to correlate with preparation time. However, to better capture meaningful patterns in the dataset—such as meal type, difficulty level, and special requirements—we will perform feature engineering on key categorical columns. This includes transforming `tags` (e.g., "dinner," "beginner," or dietary restrictions) into informative numerical features.

We expect that recipes with more steps, greater ingredient complexity, or tags indicating more involved preparation (e.g., "dinner," easy," "holiday," etc.) will generally require more time to complete.

Step 6: Baseline Model

In [36]:
```
from collections import Counter
import ast

tag_lst_lst = data["tags"].apply(ast.literal_eval).to_list()
tags_cnt = sum([Counter(tag_lst) for tag_lst in tag_lst_lst], Counter())
```

In [37]:
```
import re
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.metrics import make_scorer, mean_squared_error

def find_tag(tag_lst, tag):
    return 1 if len(re.findall(fr"'{tag}'", tag_lst)) != 0 else 0

all_tags = ["preparation", "dietary", "easy", "occasion", "main-dish", "vegetables", "meat"]
numeric_features = ["n_steps","n_ingredients"]
one_hot_features = []
data_pred = data.copy()

for tag in all_tags:
    data_pred[f"with_{tag}"] = data_pred["tags"].apply(find_tag, args = (tag,))
    pred_tag = "with_" + tag
    one_hot_features.append(pred_tag)

model = Pipeline([
    ("regressor", LinearRegression())
])

x_var = data_pred[numeric_features + one_hot_features]
y_var = data_pred["minutes"]
X_train, X_test, y_train, y_test = train_test_split(
    x_var, y_var, test_size=0.2, random_state=42
)

#K-fold validation with evaluation
rmse_scorer = make_scorer(mean_squared_error, squared = False)
cv_scores = cross_val_score(model, X_train, y_train, cv = 5, scoring=rmse_scorer)


model.fit(X_train, y_train)
y_pred = model.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print("RMSE:", rmse)
```

```
RMSE: 53.045515437919754
```

```
c:\.local\share\mamba\envs\dsc80\Lib\site-packages\sklearn\metrics\_regression.py:492: FutureWarning:

'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the functio
n'root_mean_squared_error'.

c:\.local\share\mamba\envs\dsc80\Lib\site-packages\sklearn\metrics\_regression.py:492: FutureWarning:

'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the functio
n'root_mean_squared_error'.

c:\.local\share\mamba\envs\dsc80\Lib\site-packages\sklearn\metrics\_regression.py:492: FutureWarning:

'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the functio
n'root_mean_squared_error'.

c:\.local\share\mamba\envs\dsc80\Lib\site-packages\sklearn\metrics\_regression.py:492: FutureWarning:

'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the functio
n'root_mean_squared_error'.

c:\.local\share\mamba\envs\dsc80\Lib\site-packages\sklearn\metrics\_regression.py:492: FutureWarning:

'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the functio
n'root_mean_squared_error'.
```

Step 7: Final Model

Our baseline linear regression model produced an RMSE of 53, which is not sufficiently accurate for our prediction task. To improve performance, we explored two optimization strategies:

1. Additional feature engineering — including creating interaction and polynomial features, as well as applying logarithmic or power transformations to reduce skewness and stabilize variance.
2. Modifying the model structure — by experimenting with more flexible or nonlinear models.

We first applied strategy (1), then proceeded to strategy (2). Finally, we compared all improved models and selected the best-performing one(s).

In [38]:

```python
import re
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.metrics import make_scorer, mean_squared_error
from sklearn.preprocessing import FunctionTransformer, PolynomialFeatures
from sklearn.compose import ColumnTransformer


def find_tag(tag_lst, tag):
    return 1 if len(re.findall(fr"'{tag}'", tag_lst)) != 0 else 0


all_features = numeric_features + one_hot_features

log_transformer = FunctionTransformer(np.log1p, validate=False)

def manual_features(X):
    steps = X.iloc[:, 0]
    ings = X.iloc[:, 1]
    interaction = steps * ings
    ratio = steps / (1 + ings)
    return np.vstack([interaction, ratio]).T

manual_feature_names = ["steps_x_ings", "steps_div_ings"]
manual_feature_transformer = FunctionTransformer(manual_features, validate=False)
poly_interact = PolynomialFeatures(
    degree=2,
    interaction_only=False,
    include_bias=False
)


preprocessor = ColumnTransformer(
    transformers=[
        ("log_numeric", log_transformer, numeric_features),
        ("manual_feat", manual_feature_transformer, numeric_features),
        ("poly_interaction", poly_interact, all_features)
    ],
```

```
    remainder="passthrough"
)

model = Pipeline([
    ("preprocess", preprocessor),
    ("regressor", LinearRegression())
])


model.fit(X_train, y_train)
y_pred = model.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print("RMSE:", rmse)
```

```
RMSE: 52.23256606605837
```

Using log-transformed numeric features and our engineered ratio features, we improved the RMSE slightly to 52.2. To further enhance model performance, we next explore polynomial regression and random forest models.

Interestingly, the numeric variables do not seem to contribute strongly to prediction accuracy. This is likely because their relationships with `minutes` are complex and highly nonlinear. We consider two approaches to address this:

1. Model these numeric features with polynomial functions, allowing the model to capture more flexible nonlinear patterns. Given the large size of our dataset, the risk of severe overfitting is relatively low.
2. De-emphasize the numeric variables and rely more heavily on the categorical features, which can be more effectively leveraged by tree-based models such as random forests.

In [39]:
```
from sklearn.preprocessing import PolynomialFeatures

poly_preprocessor = ColumnTransformer(
    transformers=[
        ("poly_interaction", poly_interact, all_features)
    ],
    remainder="passthrough"
)

poly_model = Pipeline([
    ("pre_combine", poly_preprocessor),
    ("poly", PolynomialFeatures(degree = 2, include_bias = True)),
    ("linear", LinearRegression())
])


poly_model.fit(X_train, y_train)
y_pred = poly_model.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print("RMSE:", rmse)
```

```
RMSE: 51.23243046325278
```

We use the polynomial regression approach to identify and remove redundant features that do not contribute to improved performance. After refining the feature set, we implement a random forest model.

In [40]:
```
linear_step = poly_model.named_steps['linear']
if hasattr(linear_step, 'coef_'):
    coefs = linear_step.coef_
    poly_features = poly_model.named_steps['poly'].get_feature_names_out(
        poly_model.named_steps['pre_combine'].get_feature_names_out()
    )
    coefs = poly_model.named_steps['linear'].coef_
    feature_importance = pd.Series(coefs, index = poly_features)
    important_features = feature_importance[feature_importance.abs() > 0.01].sort_values(ascending=False)
    print("features that matter:")
    print(important_features.iloc[:10])
```

```
features that matter:
poly_interaction__with_vegetables                                169.73
poly_interaction__with_vegetables^2                              169.73
poly_interaction__with_vegetables^2                              169.73
                                                                  ...
poly_interaction__with_occasion poly_interaction__with_occasion^2  78.30
poly_interaction__with_occasion^2                                 78.30
poly_interaction__with_occasion^2                                 78.30
Length: 10, dtype: float64
```

In [41]:
```
from sklearn.ensemble import RandomForestRegressor

rf_model = Pipeline([
```

```
    ("preprocess", poly_preprocessor),
    ("rf", RandomForestRegressor(n_estimators = 200, max_depth = 10, random_state = 42, n_jobs = -1))
])

rf_model.fit(X_train, y_train)
y_pred_rf = rf_model.predict(X_test)
rmse_rf = np.sqrt(mean_squared_error(y_test, y_pred_rf))
print("\nRandom Forest RMSE:", rmse_rf)
```

```
Random Forest RMSE: 47.98761903038597
```

Across all modeling strategies—including engineered numeric features, polynomial regression, and random forest regression—the Random Forest Regressor delivered the strongest performance, achieving an RMSE of 47, a substantial improvement over both the baseline linear model and the enhanced polynomial variants. This suggests that recipe duration is driven by complex, nonlinear interactions that are better captured by tree-based models than by linear methods, even with extensive feature engineering. Overall, Random Forests provide a more flexible and robust framework for predicting recipe preparation times and represent the most effective approach for this dataset.

Step 8: Fairness Analysis

We conduct a fairness analysis to evaluate whether our model performs equitably across different recipe types. Specifically, we compare prediction errors between meat-based recipes and vegetable-based recipes. By examining metrics such as RMSE, bias, and error distributions for each group, we aim to determine whether the model systematically favors or disadvantages one type of recipe over the other.

In [42]:
```
df_eval = X_test.copy()
df_eval['y_true'] = y_test
df_eval['y_pred'] = y_pred_rf
df_eval['error'] = df_eval['y_true'] - df_eval['y_pred']
df_eval['sq_error'] = df_eval['error'] ** 2

meat = df_eval[df_eval['with_meat'] == 1]
vegetable = df_eval[df_eval['with_vegetables'] == 1]

print(f"Meat-based recipes evaluation: {np.sqrt(meat['sq_error'].mean())}")
print(f"Vegetable-based recipes evaluation: {np.sqrt(vegetable['sq_error'].mean())}")
```

```
Meat-based recipes evaluation: 66.2992687694115
Vegetable-based recipes evaluation: 44.755841002768996
```

There is a significant difference in RMSE between meat-based and vegetable-based recipes. The RMSE for meat-based recipes is 66.3, while the RMSE for vegetable-based recipes is 44.8, resulting in an observed difference of 21.5. To assess the statistical significance of this difference, we perform a permutation test.

In [43]:
```
# Extract squared errors
sq_error_meat = meat['sq_error'].values
sq_error_veg = vegetable['sq_error'].values

# Observed RMSE difference
observed_diff = np.sqrt(sq_error_meat.mean()) - np.sqrt(sq_error_veg.mean())

# Permutation test
combined = np.concatenate([sq_error_meat, sq_error_veg])
n_perm = 5000
count = 0

test_stats = []
for _ in range(n_perm):
    np.random.shuffle(combined)
    perm_meat = combined[:len(sq_error_meat)]
    perm_veg = combined[len(sq_error_meat):]

    perm_diff = np.sqrt(perm_meat.mean()) - np.sqrt(perm_veg.mean())

    test_stats.append(perm_diff)

p_value = sum(abs(stat) >= abs(observed_diff) for stat in test_stats) / n_perm

print("Observed RMSE difference:", observed_diff)
print("Permutation test p-value:", p_value)
```

```
Observed RMSE difference: 21.543427766642502
Permutation test p-value: 0.0
```

The permutation test yields a p-value of 0.000, which is less than the significance level of 0.05. Therefore, we reject the null hypothesis and conclude that there is strong evidence of a significant difference in RMSE between meat-based and vegetable-based recipes. There is a

statistically significant disparity in model performance, indicating potential fairness concerns that warrant further investigation and mitigation strategies in further work.