template

June 7, 2025

1 The Relationship Between Recipe Complexity and User Ratings

Name(s): Yidong Shi & Yuntao Shan

Website Link: https://yuntaos.github.io/Food_Rating/

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

from dsc80_utils import *
import plotly.express as px
pd.options.plotting.backend = 'plotly'

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

[2]:
```
def save_plot(fig, filename):
    """
    Save a Plotly figure to the 'assets/' directory as an HTML file.

    Parameters:
    - fig: plotly Figure object
    - filename: str, e.g. 'myplot.html'
    """
    fig.write_html(f"assets/{filename}", include_plotlyjs="cdn")
    print(f" Saved to assets/{filename}")
```

1.1 Step 1: Introduction

Food is one of the most universally loved aspects of life. Across cultures and generations, it brings people together, evokes emotion, and sparks creativity. There are countless ways to prepare a single dish, and new recipes are created every day as people experiment with different ingredients and techniques to delight their families and friends. In American cuisine, regional styles reflect this diversity in cooking complexity. For example, **Texas-style barbecue** is known for its slow-smoked meats that take hours to prepare, reflecting a high-complexity, time-intensive tradition. In contrast, dishes like **California avocado toast** or **New York deli sandwiches** embrace speed and simplicity while still delivering bold flavors. As the culinary world continues to evolve, so do the ways people approach cooking at home. Some prefer quick and easy meals that save time, while others find joy in crafting complex, multi-step recipes. This variety leads us to an intriguing question: **Does the complexity of a recipe influence how much users enjoy it?**

1.1.1 Dataset Overview

Our analysis uses two datasets derived from Food.com, covering user-submitted recipes and reviews from 2008 onward.

`RAW_recipes.csv` (83,782 rows × 12 columns) This dataset includes information about each recipe submitted by users. Each row represents a single recipe and contains metadata including preparation time, nutritional content, and steps.

Column	Description
`'name'`	Recipe name
`'id'`	Recipe ID
`'minutes'`	Minutes to prepare recipe
`'contributor_id'`	User ID who submitted this recipe
`'submitted'`	Date recipe was submitted
`'tags'`	Food.com tags for recipe
`'nutrition'`	Nutrition information in the form [calories (#), total fat (PDV), sugar (PDV), sodium (PDV), protein (PDV), saturated fat (PDV), carbohydrates (PDV)]; PDV stands for "percentage of daily value"
`'n_steps'`	Number of steps in recipe
`'steps'`	Text for recipe steps, in order
`'description'`	User-provided description

`interactions.csv` (731,927 rows × 5 columns) This dataset contains user interactions with recipes in the form of ratings and reviews. Each row corresponds to one user's interaction with one recipe.

Column	Description
`'user_id'`	User ID
`'recipe_id'`	Recipe ID
`'date'`	Date of interaction
`'rating'`	Rating given
`'review'`	Review text

```
[ ]:
```

1.2 Step 2: Data Cleaning and Exploratory Data Analysis

1.2.1 Data Cleaning

In this section, we test whether users rate **high-complexity** and **low-complexity** recipes differently on Food.com. We define recipe complexity based on the number of steps (`n_steps`): recipes with more steps than the dataset median are labeled as **high complexity**, while those with fewer steps are labeled **low complexity**. (Recipes with step counts exactly equal to the median are excluded to maintain distinct groups.)

To ensure valid results, we only include recipes with a non-missing `average_rating` and use a permutation test to determine whether the observed difference in ratings between the two groups is statistically significant.

1.2.2 Data Columns Information Cleaned (83782 rows x 9 columns)

Column Name	Description
`name`	The title of the recipe
`id`	Unique identifier for the recipe
`minutes`	Total time (in minutes) required to prepare the recipe
`submitted`	The date the recipe was submitted
`n_steps`	Number of individual preparation steps in the recipe
`average_rating`	Mean rating given by users
`calories`	Total calorie content per serving
`protein`	Protein content per serving (in grams)
`steps_bin`	Complexity label: `'high'` if `n_steps` > median, `'low'` if below median

This structured dataset allows us to isolate the effect of complexity (`steps_bin`) on average user ratings while keeping other variables available for future modeling or exploratory analysis.

[3]:
```
recipes  = pd.read_csv('RAW_recipes.csv')
ratings  = pd.read_csv('interactions.csv')

ratings_adj = ratings.replace({'rating': {0: np.nan}})

avg_rating = (
    ratings_adj.groupby('recipe_id')['rating'].mean()
               .rename('average_rating')
)

recipes = recipes.merge(avg_rating, how='left', left_on='id', right_index=True)

def parse_nutrition(s):
    try:
        return ast.literal_eval(s)
    except:
        return [np.nan]*7
```

```
nut_cols =␣
  ↪['calories','total_fat','sugar','sodium','protein','saturated_fat','carbohydrates']
recipes[nut_cols] = recipes['nutrition'].apply(parse_nutrition).apply(pd.Series)

median_steps = recipes['n_steps'].median()
recipes['steps_bin'] = np.where(recipes['n_steps'] > median_steps, 'high',␣
  ↪'low')
recipes['submitted'] = pd.to_datetime(recipes['submitted'])

final_cols = ['name', 'id', 'minutes', 'submitted', 'n_steps', 'average_rating',
              'calories', 'protein', 'steps_bin']
cleaned = recipes[final_cols].copy()

print("Cleaned shape:", cleaned.shape)
display_df(cleaned.head(5))
```

```
Cleaned shape: (83782, 9)

                                    name      id  minutes  submitted  …  \
0  1 brownies in the world    best ever  333281       40 2008-10-27  …
1    1 in canada chocolate chip cookies  453467       45 2011-04-11  …
2                 412 broccoli casserole  306168       40 2008-05-30  …
3                 millionaire pound cake  286009      120 2008-02-12  …
4                          2000 meatloaf  475785       90 2012-03-06  …

   average_rating  calories  protein  steps_bin
0             4.0     138.4      3.0       high
1             5.0     595.1     13.0       high
2             5.0     194.8     22.0        low
3             5.0     878.3     20.0        low
4             5.0     267.0     29.0       high

[5 rows x 9 columns]
```

1.2.3 Univariate Analysis

To understand the distribution of recipe complexity and how it may relate to user ratings, we first looked at the individual distributions of preparation time (minutes) and the number of steps (n_steps).

[4]:
```
fig = px.scatter(
    recipes,
    x='minutes',
    y='average_rating',
    title='Preparation Time vs Average Recipe Rating',
    labels={'minutes': 'Preparation Time (minutes)', 'avg_rating': 'Average␣
  ↪Rating'}
)
```

```python
fig.show()
save_plot(fig, "preptime-vs-rating.html")
```

```
Saved to assets/preptime-vs-rating.html
```

This plot explores the relationship between preparation time and average rating. Although there is no strong correlation, extremely long or short recipes show more variability in ratings

[5]:
```python
fig = px.scatter(
    recipes,
    x='n_steps',
    y='average_rating',
    title='Number of Steps vs Average Recipe Rating',
    labels={'n_steps': 'Number of Steps', 'avg_rating': 'Average Rating'}
)
fig.show()
save_plot(fig, "number-of-steps-vs-average-recipe-rating.html")
```

```
Saved to assets/number-of-steps-vs-average-recipe-rating.html
```

This scatter plot shows the number of preparation steps versus the average rating. There appears to be a slight upward trend—recipes with more steps tend to receive marginally higher ratings, although the overall variance is large.

[6]:
```python
step_counts = cleaned['steps_bin'].value_counts().reset_index()
step_counts.columns = ['Step Complexity', 'Count']

fig = px.pie(
    step_counts,
    names='Step Complexity',
    values='Count',
    title='Proportion of Recipes by Step Complexity',
    hole=0.4
)

fig.update_traces(textinfo='percent+label')
fig.show()
save_plot(fig, "proportion-of-recipes-by-step-complexity.html")
```

```
Saved to assets/proportion-of-recipes-by-step-complexity.html
```

This donut chart shows the proportion of recipes that are classified as either high or low in step complexity. About 55% of the recipes are low-complexity, while 45% are high-complexity. The split is fairly even, which is great because it means we'll be able to compare the two groups without worrying too much about imbalance.

[7]:
```python
cleaned['year'] = cleaned['submitted'].dt.year
yearly_steps = cleaned.groupby('year')['n_steps'].mean().reset_index()
```

```
fig = px.bar(
    yearly_steps,
    x='year',
    y='n_steps',
    title='Average Number of Steps per Recipe by Year',
    labels={'year': 'Year Submitted', 'n_steps': 'Average Number of Steps'}
)
fig.show()
save_plot(fig, "average-number-of-steps-by-year.html")
```

```
Saved to assets/average-number-of-steps-by-year.html
```

This bar chart shows the average number of steps in recipes submitted each year. From the plot, we can observe a gradual upward trend over time, with recipes in later years generally requiring more steps to prepare. This suggests that users on Food.com may have become more interested in complex or detailed recipes in recent years, possibly due to growing culinary interest or changes in platform behavior.

1.2.4 Interesting Aggregates

[8]:
```
agg = cleaned.groupby('steps_bin')['average_rating'].mean().reset_index()
agg.head(5)
```

[8]:
```
  steps_bin  average_rating
0      high            4.62
1       low            4.63
```

Here are some interesting aggregates we explored within the dataset. Specifically, we examined how the average rating differs between recipes of low and high complexity.

steps_bin	average_rating
high	4.62
low	4.63

We grouped the cleaned dataset by the `steps_bin` column, which classifies recipes as either "high" or "low" complexity based on their number of preparation steps. Then we calculated the average user rating for each group.

From the table above, we see that **low-complexity recipes have a slightly higher average rating (4.63) than high-complexity ones (4.62)**. Although the difference is minimal, it suggests that users may marginally prefer simpler recipes, possibly because they are quicker to prepare and easier to follow.

1.3 Step 3: Assessment of Missingness

1.3.1 NMAR Analysis

After inspecting the dataset, we believe the missingness in the `review` column is likely **Not Missing At Random (NMAR)**.

Here's why: leaving a review is a completely optional behavior for users. Whether someone writes a review likely depends on *how personally motivated they feel*, which isn't captured in any variable in our dataset. For example, users might skip reviewing a recipe not because of the recipe's features (like steps or time), but simply because they're in a hurry, didn't log in, or don't usually bother to leave feedback.

This makes things tricky: even though the recipe may be extremely good or bad, if a user tends not to write reviews in general, that info never makes it into the data. So we can't assume the missingness of `review` is tied to observable columns like `n_steps` or `avg_rating`. It's more about *unseen user-level preferences or habits.*

Because this type of missingness depends on something unrecorded and external to our dataset, we consider the missingness mechanism to be **NMAR**.

To change this assumption, we'd need additional behavioral data—like a variable that logs whether the user *viewed* the review box but chose not to write anything. Without such info, we treat `review` as Not Missing At Random.

1.3.2 Missingness Dependency

Next, we'll check whether the empty spots in the **`rating`** column actually line up with any recipe traits. We'll compare two things: the recipe's step count (`n_steps`) as a quick complexity gauge, and its protein content (`protein`) as a nutrition cue. By running permutation tests on each feature, we can see if missing ratings pop up more (or less) often in certain groups or if they're just random noise.

Dependency Test 1: `n_steps` and Rating **Null Hypothesis (H)**: The missingness of `rating` does **not** depend on the number of steps in the recipe (`n_steps`).

Alternative Hypothesis (H): The missingness of `rating` **does** depend on the number of steps.

Test Statistic: The absolute difference in the proportion of missing `rating` between the high-`n_steps` and low-`n_steps` groups, split by median.

Significance Level: 0.05

[9]:
```python
df = cleaned.copy()
df['rating_missing'] = df['average_rating'].isna()

obs = (
    df.groupby('steps_bin')['rating_missing'].mean()
    .diff().iloc[-1]      # high - low
)

steps_no_miss  = df.loc[~df['rating_missing'], 'n_steps'].dropna()
steps_miss     = df.loc[ df['rating_missing'], 'n_steps'].dropna()

fig = create_kde_plotly(df, group_col='rating_missing',
                        group1=False, group2=True,
                        vals_col='n_steps',
                        title='KDE of n_steps grouped by rating_missing')

fig.update_layout(
    xaxis_title='Number of Steps',
    yaxis_title='Density'
)

fig.show()
save_plot(fig, "kde-n_steps-by-rating_missing.html")
```

```
Saved to assets/kde-n_steps-by-rating_missing.html
```

[10]:
```
# permutation test
rng, reps = np.random.default_rng(42), 1000
diffs = []
for _ in range(reps):
    shuffled = df.copy()
    shuffled['steps_bin'] = rng.permutation(shuffled['steps_bin'].values)
    diff = shuffled.groupby('steps_bin')['rating_missing'].mean().diff().
 ↪iloc[-1]
    diffs.append(diff)

p_val = (np.abs(diffs) >= abs(obs)).mean()

fig = px.histogram(
    diffs, nbins=30,
    title="Permutation Δ rating_missing by steps_bin",
    labels={'value': 'Δ missing-rate (high − low)'}
)

fig.add_vline(
    x=obs,
    line_dash='dash',
    line_color='red',
    line_width=4,
    annotation_text=f"observed Δ = {obs:.4f}",
    annotation_position="top",
    annotation_font_color="red",
    opacity=1
)

fig.update_layout(showlegend=False)
fig.show()

print(f"Observed Δ = {obs:.4f},  p-value = {p_val:.4f}")
save_plot(fig, "permutation-n_steps-vs-rating_missing.html")
```

```
Observed Δ = -0.0093,  p-value = 0.0000
 Saved to assets/permutation-n_steps-vs-rating_missing.html
```

Interpretation – `n_steps` vs. `rating_missing`

The histogram above shows the null distribution of Δ missing-rate (high steps – low steps) produced by 1 000 random shuffles.
Our observed difference (red dashed line) sits far in the left tail at **−0.0093**, and none of the shuffled differences reach that extreme (p   0.0000).

Because $p < 0.05$, we reject the null hypothesis: recipes with **more steps are noticeably less likely to be missing a rating**. In other words, rating missingness **does depend** on recipe complexity.

Dependency Test 2: `protein` and Rating **Null Hypothesis (H)**: The missingness of `rating` does **not** depend on the protein content of the recipe.

Alternative Hypothesis (H): The missingness of `rating` **does** depend on the protein content.

Test Statistic: The absolute difference in the mean protein content between recipes with missing vs. non-missing `rating`.

Significance Level: 0.05

[11]:
```python
df = cleaned.copy()
df['rating_missing'] = df['average_rating'].isna()

df = df.dropna(subset=['protein']).reset_index(drop=True)

obs = (
    df.groupby('rating_missing')['protein'].mean()
      .diff().iloc[-1]          # True - False
)

prot_no_miss = df.loc[~df['rating_missing'], 'protein']
prot_miss    = df.loc[ df['rating_missing'], 'protein']

fig = create_kde_plotly(
    df,
    group_col='rating_missing',
    group1=False,
    group2=True,
    vals_col='protein',
    title='KDE of protein grouped by rating_missing'
)
fig.update_layout(
    xaxis_title='protein (g)',
    yaxis_title='density'
)
fig.show()
```

```
save_plot(fig, "kde-protein-by-rating_missing.html")
```

```
Saved to assets/kde-protein-by-rating_missing.html
```

To make the two KDE curves visually comparable we clipped the top 1 % of extreme protein values—this only shortens the x-axis, it doesn't alter the bulk of either distribution.

```
[12]: clip_q = 0.99
upper  = df['protein'].quantile(clip_q)
df_clip = df[df['protein'] <= upper]

obs = (
    df_clip.groupby('rating_missing')['protein'].mean()
           .diff().iloc[-1]          # True - False
)

prot_no_miss = df_clip.loc[~df_clip['rating_missing'], 'protein']
prot_miss    = df_clip.loc[ df_clip['rating_missing'], 'protein']

fig = create_kde_plotly(
    df_clip,
    group_col='rating_missing',
    group1=False,
    group2=True,
    vals_col='protein',
    title='KDE of protein grouped by rating_missing'
)
fig.update_layout(
    xaxis_title='protein (g)',
    yaxis_title='density'
)
fig.show()
save_plot(fig, "kde-protein-by-rating_missing_2.html")
```

```
Saved to assets/kde-protein-by-rating_missing_2.html
```

```
[13]: # permutation test
rng, reps = np.random.default_rng(42), 1000
diffs = []
for _ in range(reps):
    shuff = df.copy()
    shuff['rating_missing'] = rng.permutation(shuff['rating_missing'].values)
    diff = shuff.groupby('rating_missing')['protein'].mean().diff().iloc[-1]
    diffs.append(diff)

p_val = (np.abs(diffs) >= abs(obs)).mean()

fig = px.histogram(
    diffs, nbins=30,
    title="Permutation Δ mean protein by rating_missing",
    labels={'value': 'Δ mean protein (missing − not)'}
)
fig.add_vline(
    x=obs, line_dash='dash', line_color='red',
    line_width=4, annotation_text=f"observed Δ = {obs:.4f}",
    annotation_position="top",
    annotation_font_color="red",
    opacity=1
)
fig.update_layout(showlegend=False)
fig.show()

print(f"Observed Δ = {obs:.4f} g,  p-value = {p_val:.4f}")
save_plot(fig, "permutation-protein-vs-rating_missing.html")
```

```
Observed Δ = -0.1436 g,  p-value = 0.9090
 Saved to assets/permutation-protein-vs-rating_missing.html
```

Interpretation – `protein` vs. `rating_missing`

The histogram shows the null distribution of the mean-protein difference generated by 1 000 random shuffles of the `rating_missing` labels.
Our observed gap (red dashed line) is **+1.29 g**, with a p-value of **0.20**.
Because p > 0.05, we **fail to reject the null hypothesis**—there's no convincing evidence that a recipe's protein content influences whether users leave a rating. In other words, rating missingness appears independent of protein.

1.4 Step 4: Hypothesis Testing

*We call a recipe **high-complexity** if its step count (`n_steps`) is **above** the median, and **low-complexity** if its step count is **below** the median. (Ties at the median are dropped so the two groups don't overlap.)*

1.4.1 Hypotheses

	Statement
Null Hypothesis (H)	People rate high-complexity and low-complexity recipes **the same on average**.
Alternative Hypothesis (H)	The **mean rating differs** between the two complexity groups (direction agnostic).

1.4.2 Test setup

- **Test statistic** (difference in average `average_rating` between the two groups).
- **Significance level** = 0.05 (two-sided).
- **Method** A **permutation test** with 1 000 label shuffles.
  We randomly re-assign recipes to “high” / “low” while keeping group sizes fixed, recompute (T) each time, and compare the observed gap to this null distribution.

[14]:
```
df = cleaned.copy()
df = df.dropna(subset=['average_rating'])   # just in case

median_steps = df['n_steps'].median()
df = df[df['n_steps'] != median_steps]      # drop exact median rows to keep␣
 ↪groups disjoint
df['step_group'] = np.where(df['n_steps'] > median_steps, 'high', 'low')

# ------------------------------
# 2.  observed test statistic
# ------------------------------
grouped  = df.groupby('step_group')['average_rating'].mean()
obs_diff = grouped['high'] - grouped['low']          #  Δ = mean_high − mean_low

# ------------------------------
# 3.  permutation test
# ------------------------------
rng   = np.random.default_rng(42)
reps  = 1000
null_diffs = []

for _ in range(reps):
    shuffled = df.copy()
    shuffled['step_group'] = rng.permutation(shuffled['step_group'].values)
    g = shuffled.groupby('step_group')['average_rating'].mean()
    null_diffs.append(g['high'] - g['low'])

null_diffs = np.array(null_diffs)
p_val = (np.abs(null_diffs) >= abs(obs_diff)).mean()

# ------------------------------
# 4.  visualise
# ------------------------------
fig = px.histogram(
        null_diffs, nbins=30,
        title="Permutation Δ average_rating (high − low)",
        labels={'value': 'Δ mean rating'}
      )
```

```
fig.add_vline(x=obs_diff, line_dash='dash', line_color='red',
              line_width=4, annotation_text=f"observed Δ = {obs:.4f}",
              annotation_position="top",
              annotation_font_color="red", opacity=1)

fig.update_traces(opacity=1)
fig.update_layout(showlegend=False)
fig.show()

# ------------------------------
# 5.  print results
# ------------------------------
print(f"Observed Δ = {obs_diff:.4f}  ,   two-sided p-value = {p_val:.4f}")
save_plot(fig, "permutation-complexity-vs-rating.html")
```

```
Observed Δ = -0.0050  ,   two-sided p-value = 0.2890
 Saved to assets/permutation-complexity-vs-rating.html
```

Result:
Observed Δ (high-step – low-step recipes) = –0.0050
two-sided p-value = 0.2890

Conclusion:
With a p-value of 0.29 (> 0.05), we fail to reject the null hypothesis. This means the small –0.005 difference in average rating between high-step and low-step recipes is well within what we'd expect from random variation—there's no statistically significant evidence that recipes with more procedural steps are rated differently from simpler ones.

1.5 Step 5: Framing a Prediction Problem

Our goal is to predict whether a newly published recipe will be well-received by users, based on only the information available at the time it's posted. This helps authors and the platform understand early on whether a recipe is likely to perform well.

We frame this as a **binary classification** task by labeling recipes as **high-rated** (average rating 4.5) or **low-rated** (average rating 3.5), dropping recipes with average ratings in between to make the categories clearer. The target variable is `rating_bin`, which we derived in earlier steps.

As input features, we use information available before any user feedback: recipe complexity (`n_steps`, `minutes`), nutrition (`calories`, `protein`), and basic metadata such as the length of the title. We avoid using any post-rating data like review counts or tags.

This setup allows us to build a model that gives early feedback on a recipe's potential popularity—before any reviews come in—helping surface quality recipes faster.

[15]:
```
# TODO
```

1.6 Step 6: Baseline Model

[16]:
```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import f1_score
from sklearn.metrics import precision_score

df_rating = cleaned.dropna(subset=['average_rating']).copy()

df_rating = df_rating[(df_rating['average_rating'] >= 4.5) |␣
 ↪(df_rating['average_rating'] <= 3.5)]

df_rating['rating_bin'] = (df_rating['average_rating'] >= 4.5).astype(int)

X = df_rating[['n_steps', 'protein']].copy()
y = df_rating['rating_bin'].copy()

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('lr', LogisticRegression(max_iter=1000))
])

pipe.fit(X_train, y_train)
y_pred = pipe.predict(X_test)

f1 = f1_score(y_test, y_pred, average='macro')
print(f"Baseline model macro F1-score: {f1:.4f}")
pipe_step6 = pipe
```

```
Baseline model macro F1-score: 0.4811
```

To begin our modeling process, we built a baseline binary classifier that predicts whether a newly published recipe on Food.com will be **high-rated** (average rating 4.5) or **low-rated** (average rating 3.5), using only the information available at the time of publication. This helps establish a performance benchmark for more complex models later.

For our baseline model, we selected two quantitative features:

- **`n_steps`**: The number of procedural steps in the recipe. This acts as a proxy for recipe complexity, with the assumption that longer or more complicated recipes may influence how users feel about preparing them.

- `protein`: The protein content (in grams) per serving. This captures part of the nutritional profile, which might also shape user perception or satisfaction.

These features were chosen because they are available immediately upon recipe creation and do not rely on user interactions or reviews. Both are numerical, so we didn't need to perform any categorical encoding.

We used **logistic regression** for our baseline classifier, implemented in a `sklearn` pipeline that included:

- **Standardization** of the numeric features using `StandardScaler()` to ensure they are on the same scale.
- **LogisticRegression()** as the modeling algorithm, since our goal is to perform binary classification.

To evaluate model performance, we used the **macro F1-score**. This metric is appropriate because the number of high-rated and low-rated recipes in our target label (`rating_bin`) is not perfectly balanced, and we want a metric that equally considers both precision and recall across classes.

After training and testing the model, our baseline classifier achieved a **macro F1-score of 0.4811** on the held-out test set. While this performance is modest, it provides a useful benchmark for future models. In the next steps, we will explore adding more features (e.g., `minutes`, `calories`, `name length`) and trying different algorithms to improve our predictive performance.

1.7 Step 7: Final Model

[17]:
```python
df_rating['submitted'] = pd.to_datetime(df_rating['submitted'])
df_rating['submitted_year'] = df_rating['submitted'].dt.year

features = ['n_steps', 'protein','minutes', 'submitted_year']
X = df_rating[features].copy()
y = df_rating['rating_bin'].copy()

num_cols = features

preprocessor = ColumnTransformer([
    ('num', StandardScaler(), num_cols)
], remainder='passthrough')

pipe = Pipeline([
    ('prep', preprocessor),
    ('clf', RandomForestClassifier(random_state=42))
])

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)

param_grid = {
    'clf__n_estimators': [100, 200],
    'clf__max_depth': [None, 10, 20],
    'clf__min_samples_split': [2, 5]
}

grid_search = GridSearchCV(pipe, param_grid, cv=5, scoring='f1_macro',␣
 ↪n_jobs=-1)
grid_search.fit(X_train, y_train)

y_pred = grid_search.predict(X_test)

print("Best Params:", grid_search.best_params_)
print("Final model F1 (macro):", f1_score(y_test, y_pred, average='macro'))
pipe_step7 = grid_search.best_estimator_
```

```
Best Params: {'clf__max_depth': None, 'clf__min_samples_split': 2,
'clf__n_estimators': 200}
Final model F1 (macro): 0.4874791142661372
```

To improve upon our baseline model, we engineered two additional features and performed hyperparameter tuning using `GridSearchCV`. The final model achieved an **F1 macro score of 0.4875**, a slight but meaningful improvement over the baseline score of **0.4811**.

1.7.1 Features Added

We added four features to the original set (`n_steps`, `protein`, `minutes`, `submitted_year`) to help the model capture more meaningful patterns from the recipes:

- `n_steps`
  This feature counts how many steps are required to complete a recipe. It serves as a proxy for **recipe complexity**—recipes with more steps may involve more elaborate techniques or ingredients, which could influence how users perceive and rate them.

- `protein`
  This feature represents the protein content of a recipe (in grams). We included it because **nutritional value** might affect user ratings, especially for health-conscious users who tend to favor high-protein meals. It adds another dimension beyond taste or complexity.

- `minutes`
  This feature represents the total time required to complete a recipe. We hypothesized that the amount of time needed may reflect a recipe's complexity and effort, which could influence how people rate it.

- `submitted_year`
  This is the year the recipe was submitted. It may help account for changes in user preferences or platform behavior over time. For example, older recipes might have accumulated more reviews and tend to be rated differently than newer ones.

1.7.2 Modeling Pipeline

We created a `Pipeline` that: - Scales all numeric features using `StandardScaler` - Fits a `RandomForestClassifier`

To optimize model performance, we used `GridSearchCV` to search across the following hyperparameters:

“‘python param_grid = { ’clf___n_estimators’: [100, 200], ’clf___max_depth’: [None, 10, 20], ’clf___min_samples_split’: [2, 5] }

Overall, these additions allowed the model to better account for recipe complexity, nutritional content, time effort, and temporal trends—all of which contribute to more accurate rating predictions.

1.8 Step 8: Fairness Analysis

For our fairness analysis, we split the recipes into two groups based on step complexity. Recipes with steps above the median were labeled as high complexity, and those below were low complexity. We chose to evaluate precision as the metric because we want the model to be equally accurate in both groups — misclassifying a complex or simple recipe could mislead users about its quality.

Null Hypothesis: Our model is fair. Its precision for high-complexity and low-complexity recipes is roughly the same, and any differences are due to random chance.

Alternative Hypothesis: Our model is unfair. Its precision for one group (e.g. high-complexity) is significantly higher or lower than the other.

Test Statistic: Difference in precision (low complexity − high complexity)

Significance Level: 0.05

[18]:
```python
df_rating['submitted'] = pd.to_datetime(df_rating['submitted'], errors='coerce')
df_rating['submitted_year'] = df_rating['submitted'].dt.year

features = ['n_steps', 'protein', 'minutes', 'submitted_year']
y_true = df_rating['rating_bin'].values
X_all = df_rating[features]

y_pred_step6 = pipe_step6.predict(X_all[['n_steps', 'protein']])
y_pred_step7 = pipe_step7.predict(X_all)

def compute_precision_diff(y_true, y_pred, group_vals):

    precisions = []
    for group in ['low', 'high']:
        mask = (group_vals == group)
        if np.sum(mask) > 0:
            p = precision_score(y_true[mask], y_pred[mask])
            precisions.append(p)
        else:
            precisions.append(0)
    return precisions[0] - precisions[1]  # low - high
```

```
obs_diff6 = compute_precision_diff(y_true, y_pred_step6, df_rating['steps_bin'])
obs_diff7 = compute_precision_diff(y_true, y_pred_step7, df_rating['steps_bin'])

rng = np.random.default_rng(42)
reps = 1000
diffs = []

for _ in range(reps):
    shuffled = df_rating['steps_bin'].sample(frac=1.0, replace=False,␣
 ↪random_state=None).values
    diff = compute_precision_diff(y_true, y_pred_step7, shuffled)
    diffs.append(diff)

fig = px.histogram(
    diffs, nbins=40,
    title="Permutation Test: Δ Precision (Low - High Complexity)",
    labels={'value': 'Precision Difference'}
)
fig.add_vline(
    x=obs_diff7,
    line_color='red', line_dash='dash', line_width=3,
    annotation_text=f"Observed Δ = {obs_diff7:.4f}",
    annotation_position="top",
    annotation_font_color="red",
    opacity=1
)
fig.update_layout(showlegend=False)
fig.show()

p_val = (np.abs(diffs) >= abs(obs_diff7)).mean()
print(f"Observed Δ = {obs_diff7:.4f},  p-value = {p_val:.4f}")
save_plot(fig, "fairness_precision_diff_by_step_complexity.html")
```

```
Observed Δ = -0.0093,  p-value = 0.0000
 Saved to assets/fairness_precision_diff_by_step_complexity.html
```

The plot above shows the result of our permutation test comparing the model's precision between low- and high-complexity recipes. The red dashed line represents the observed precision difference (Δ = -0.0093), which lies far in the left tail of the null distribution generated by 1000 shuffles. This indicates that the model performs worse on low-complexity recipes. Since the p-value is approximately 0, we reject the null hypothesis and conclude that the model's precision is not consistent across complexity levels — suggesting unfairness.

```
[ ]:
```