Are Complicated Recipes Unhealthy and Boring?

Name(s): Brian Liu

Website Link: https://brianzliu.github.io/recipes-nutritional-analysis/

In [1]:
```
# Define, tune, and train the Final Random Forest model with engineered f
import pandas as pd
import numpy as np
from pathlib import Path
import ast
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import QuantileTransformer
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.metrics import mean_squared_error

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

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

Step 1: Introduction

Question to Explore: How does the complexity of the recipe (i.e. # of steps) affect the nutritional value and recipe rating?

Step 2: Data Cleaning and Exploratory Data Analysis

Data Cleaning

In [2]:
```
# Load the raw recipe and interaction datasets
recipes = pd.read_csv("RAW_recipes.csv")
interactions = pd.read_csv("RAW_interactions.csv")
```

In [3]:
```
# Merge recipes and interactions to combine metadata with user ratings
recipes_interactions = recipes.merge(interactions, left_on='id', right_on
recipes_interactions['rating'] = recipes_interactions['rating'].apply(lam
recipes_interactions_recipe_rating_avg = recipes_interactions.groupby('na
recipes_interactions_recipe_rating_avg = recipes_interactions_recipe_rati
recipes_interactions = recipes_interactions.merge(recipes_interactions_re
recipes_interactions
```

Out[3]:

	name	id	minutes	contributor_id	submitted	tags	nutritic
0	1 brownies in the world best ever	333281	40	985201	2008-10-27	['60-minutes-or-less', 'time-to-make', 'course...	[138. 10. 50. 3.0, 3. 19. 6.(
1	1 in canada chocolate chip cookies	453467	45	1848091	2011-04-11	['60-minutes-or-less', 'time-to-make', 'cuisin...	[595. 46. 211. 22. 13. 51. 26.(
2	412 broccoli casserole	306168	40	50969	2008-05-30	['60-minutes-or-less', 'time-to-make', 'course...	[194. 20. 6. 32. 22. 36. 3.(
3	412 broccoli casserole	306168	40	50969	2008-05-30	['60-minutes-or-less', 'time-to-make', 'course...	[194. 20. 6. 32. 22. 36. 3.(
4	412 broccoli casserole	306168	40	50969	2008-05-30	['60-minutes-or-less', 'time-to-make', 'course...	[194. 20. 6. 32. 22. 36. 3.(
...	...	...	...	...	...	...	
234424	zydeco ya ya deviled eggs	308080	40	37779	2008-06-07	['60-minutes-or-less', 'time-to-make', 'course...	[59. 6.0, 2. 3.0, 6. 5.0, 0.(
234425	cookies by design cookies on a stick	298512	29	506822	2008-04-15	['30-minutes-or-less', 'time-to-make', 'course...	[188. 11. 57. 11.0, 7. 21.0, 9.(
234426	cookies by design sugar shortbread cookies	298509	20	506822	2008-04-15	['30-minutes-or-less', 'time-to-make', 'course...	[174. 14. 33. 4.0, 4. 11.0, 6.(

	name	id	minutes	contributor_id	submitted	tags	nutritic
234427	cookies by design sugar shortbread cookies	298509	20	506822	2008-04-15	['30-minutes-or-less', 'time-to-make', 'course...	[174. 14. 33. 4.0, 4. 11.0, 6.
234428	cookies by design sugar shortbread cookies	298509	20	506822	2008-04-15	['30-minutes-or-less', 'time-to-make', 'course...	[174. 14. 33. 4.0, 4. 11.0, 6.

234429 rows × 18 columns

Nutritional Feature Extractions

In [4]:
```
# Making separate columns for each nutritional value
ri_custom = recipes_interactions.copy()
ri_custom['nutrition'] = ri_custom['nutrition'].apply(ast.literal_eval) #

# Define the column names in order
nutrition_columns = [
    'calories (#)',
    'total fat (PDV)',
    'sugar (PDV)',
    'sodium (PDV)',
    'protein (PDV)',
    'saturated fat (PDV)',
    'carbohydrates (PDV)'
]

# create a dataframe with the separate nutrition columns
nutrition_df = pd.DataFrame(ri_custom['nutrition'].tolist(), columns=nutr

# assign each column to ri_custom
for col in nutrition_columns:
    ri_custom[col] = nutrition_df[col]
```

Univariate Analysis

In [5]:
```
# Perform data manipulation/calculation
fig = px.histogram(ri_custom, 'rating_avg', nbins=20, title="Distribution
fig.update_layout(bargap=0.1)
fig.show()
```

In [6]:
```
# Visualize the distribution of step counts to assess recipe complexity
fig = px.histogram(ri_custom, 'n_steps', title="Distribution of Recipes'
fig.show()
```

Bivariate Analysis

In [7]:
```
# Bivariate Analysis: Explore relationship between steps and calories (us
fig = px.scatter(ri_custom, x='n_steps', y='calories (#)', title='Calorie
fig.update_layout(yaxis_type="log")
fig.add_vline(ri_custom['n_steps'].median(), annotation_text='median numb
fig.show()
```

In [8]:
```
# Perform data manipulation/calculation
cap_val = ri_custom['saturated fat (PDV)'].quantile(0.99)
ri_plot_subset = ri_custom[ri_custom['saturated fat (PDV)'] <= cap_val]

fig = px.scatter(ri_plot_subset, x='saturated fat (PDV)', y='rating_avg',
fig.update_traces(marker=dict(opacity=0.3))
fig.add_vline(ri_custom['saturated fat (PDV)'].median(), annotation_text=
fig.add_hline(ri_custom['rating_avg'].median(), annotation_text='median a
fig.show()
```

Interesting Aggregates

In [9]:
```
# group by number of steps and calculate average rating
agg_rating_by_steps = ri_custom.groupby('n_steps').agg({
    'rating_avg': 'mean',
    'id': 'count'
}).rename(columns={'id': 'count'})

agg_rating_by_steps = agg_rating_by_steps[agg_rating_by_steps['count'] >

agg_rating_by_steps = agg_rating_by_steps.sort_values('n_steps')

agg_rating_by_steps.sort_values('rating_avg', ascending=False).head()

print(agg_rating_by_steps.sort_values('rating_avg', ascending=False).head
```

n_steps	rating_avg	count
54	5	22
43	5	24
80	4.875	18
58	4.86555	17
48	4.84615	40

In [10]:
```
# Visualize the aggregated trend of ratings vs steps with an OLS regressi
fig = px.scatter(agg_rating_by_steps,
                 x=agg_rating_by_steps.index,
                 y='rating_avg',
                 trendline='ols',
                 title='Average Rating vs. Number of Steps',
                 labels={'n_steps': 'Number of Steps', 'rating_avg': 'Ave
fig.update_layout(width=800, height=480) # Adjust figure size for better
fig.show()
```

In [11]:
```
# Create a pivot table to compare nutrition metrics across different comp
ri_custom['complexity_level'] = pd.cut(ri_custom['n_steps'],
                                       bins=[0, 5, 10, 20, 100],
                                       labels=['Low (1-5)', 'Medium (6-10)'

agg_nutrition_by_complexity = ri_custom.pivot_table(
    index='complexity_level',
```

```
    values=['calories (#)', 'sugar (PDV)', 'total fat (PDV)'],
    aggfunc='mean'
)

print(agg_nutrition_by_complexity.to_markdown())
```

complexity_level	calories (#)	sugar (PDV)	total fat (PDV)
Low (1-5)	315.745	59.7003	23.3981
Medium (6-10)	397.962	59.3847	30.0082
High (11-20)	480.02	66.5914	37.0473
Very High (20+)	654.781	95.4448	51.343

```
/var/folders/ng/bts1b1qn5_gfyd59gql9sf2m0000gn/T/ipykernel_44923/130936419.py:5: FutureWarning:

The default value of observed=False is deprecated and will change to observed=True in a future version of pandas. Specify observed=False to silence this warning and retain the current behavior
```

Step 3: Assessment of Missingness

Goal: Examine if the missingness of the 'rating' column depends on 'n_steps' and 'minutes'

Missingness Dependency ('n_steps')

In [12]:
```
# Calculate observed statistics for Missingness Dependency test
ri_custom['rating_missing'] = ri_custom['rating'].isna()

mean_missing = ri_custom[ri_custom['rating_missing']]['n_steps'].mean()
mean_present = ri_custom[~ri_custom['rating_missing']]['n_steps'].mean()
observed_diff = abs(mean_missing - mean_present)

print(f'Mean n_steps (Rating Missing): {mean_missing}')
print(f'Mean n_steps (Rating Present): {mean_present}')
print(f'Observed Statistic: {observed_diff}')

n_permutations = 1000
simulated_diffs = []

shuffled_missing_pooled = ri_custom['rating_missing'].values
n_steps_values = ri_custom['n_steps'].values

for _ in range(n_permutations):
    shuffled_missing = np.random.permutation(shuffled_missing_pooled)

    mean_m = n_steps_values[shuffled_missing].mean()
    mean_p = n_steps_values[~shuffled_missing].mean()

    simulated_diffs.append(abs(mean_m - mean_p))
```

```
p_value = (np.array(simulated_diffs) >= observed_diff).mean()
print(f'p-value: {p_value}')

# plotting
fig = px.histogram(pd.DataFrame({'simulated_diffs': simulated_diffs}), x=
fig.add_vline(x=observed_diff, line_color='red', annotation_text='Observe
fig.show()
```

```
Mean n_steps (Rating Missing): 11.270617185421655
Mean n_steps (Rating Present): 9.931975951830733
Observed Statistic: 1.3386412335909217
p-value: 0.0
```

Missingness Non-dependency ('minutes')

In [13]:
```
# Calculate observed statistics for Missingness Dependency test
ri_custom['rating_missing'] = ri_custom['rating'].isna()

mean_missing = ri_custom[ri_custom['rating_missing']]['minutes'].mean()
mean_present = ri_custom[~ri_custom['rating_missing']]['minutes'].mean()
observed_diff = abs(mean_missing - mean_present)

print(f'Mean minutes (Rating Missing): {mean_missing}')
print(f'Mean minutes (Rating Present): {mean_present}')
print(f'Observed Statistic: {observed_diff}')

n_permutations = 1000
simulated_diffs = []

shuffled_missing_pooled = ri_custom['rating_missing'].values
minutes_values = ri_custom['minutes'].values

for _ in range(n_permutations):
    shuffled_missing = np.random.permutation(shuffled_missing_pooled)

    mean_m = minutes_values[shuffled_missing].mean()
    mean_p = minutes_values[~shuffled_missing].mean()

    simulated_diffs.append(abs(mean_m - mean_p))

p_value = (np.array(simulated_diffs) >= observed_diff).mean()
print(f'p-value: {p_value}')

# plotting
fig = px.histogram(pd.DataFrame({'simulated_diffs': simulated_diffs}), x=
fig.add_vline(x=observed_diff, line_color='red', annotation_text='Observe
fig.show()
```

```
Mean minutes (Rating Missing): 154.94193934557063
Mean minutes (Rating Present): 103.48956894704936
Observed Statistic: 51.45237039852127
p-value: 0.127
```

Step 4: Hypothesis Testing

Pair of Hypotheses

Null Hypothesis: Recipes with 10+ steps and recipes with <10 steps are equal in average calories.

Alternative Hypothesis: Recipes with 10+ steps have higher average calories than recipes with <10 steps.

Test Statistic: Difference between mean calories of recipes with 10+ steps and mean calories with <10 steps.

Ad Hoc Data Preprocessing

In [14]:
```
ri_custom_calories_nsteps = ri_custom.loc[:, ['n_steps', 'calories (#)']]
ri_custom_calories_nsteps['>=10 steps'] = ri_custom_calories_nsteps['n_st
```

Permutation Testing

In [15]:
```
observed_stat_df = ri_custom_calories_nsteps.groupby('>=10 steps').mean()

observed_diff = observed_stat_df.loc[True, 'calories (#)'] - observed_sta
observed_diff
```

Out[15]:
```
np.float64(128.10419925093294)
```

In [16]:
```
n_permutations = 1000
permuted_diffs = []

shuffled_df = ri_custom_calories_nsteps.copy()

for _ in range(n_permutations):
    shuffled_df['shuffled_calories (#)'] = np.random.permutation(shuffled
    shuffled_calorie_means = shuffled_df.groupby('>=10 steps')['shuffled_
    permuted_diffs.append(shuffled_calorie_means.loc[True] - shuffled_cal
```

In [17]:
```
# Perform data manipulation/calculation
p_value = (np.array(permuted_diffs) >= observed_diff).mean()
p_value
```

Out[17]:
```
np.float64(0.0)
```

Step 5: Framing a Prediction Problem

Problem Identificaton

Problem Type: Regression

Response Variable: rating_avg. The reason I chose average rating is because I want to understand the relationship between a recipe's complexity and nutritional value with its perceived quality

Evaluation Metric: RMSE; I chose RMSE over other metrics like R^2 as RMSE is more interpretable in the same units as the average ratings, and RMSE penalizes large errors more heavily, which is important to consider for user satisfaction.

Predictor Variables:

- Baseline Model: calories and n_steps. Since these variables are known before any user rates the recipe, they are valid variables to predict average rating.
- Final Model: minutes, n_ingredients, total fat (PDV), sugar (PDV), protein (PDV), calories_per_minute, complexity_density, in addition to n_steps and calories (#).

Step 6: Baseline Model

In [ ]:
```
ri_custom_valid_rating = ri_custom.dropna(subset=['rating_avg'])
```

In [19]:
```
# Train and evaluate the Baseline Linear Regression model using simple fe
X = ri_custom_valid_rating.loc[:, ['n_steps', 'calories (#)']]
y = ri_custom_valid_rating.loc[:, 'rating_avg']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,

preprocessing = ColumnTransformer(
    transformers=[
        ('scaling', StandardScaler(), ['n_steps', 'calories (#)'])
    ],
    remainder='drop'
)

baseline_model = Pipeline([
    ('preprocessing', preprocessing),
    ('regression', LinearRegression())
])

baseline_model.fit(X_train, y_train)

y_pred_train = baseline_model.predict(X_train)
y_pred_test = baseline_model.predict(X_test)

rmse_train = np.sqrt(mean_squared_error(y_train, y_pred_train))
rmse_test = np.sqrt(mean_squared_error(y_test, y_pred_test))

print(f'Train RMSE: {rmse_train}')
print(f'Test RMSE: {rmse_test}')
```

```
Train RMSE: 0.4975345315818557
Test RMSE: 0.49726919021617294
```

Step 7: Final Model

Hyperparameter Tuning and Final Model

We will use a Random Forest Regressor as our final model. We chose this model because it can capture non-linear relationships and interactions between features better than a simple linear model.

Preprocessing:

- **QuantileTransformer**: Applied to `minutes` and `calories (#)` to handle skewness and outliers.
- **StandardScaler**: Applied to other numerical features to normalize their range.

Hyperparameters to Tune:

- `n_estimators`: Number of trees in the forest. We will try [50, 100]. **Why**: More trees usually give better performance but with diminishing returns and higher cost.
- `max_depth`: Maximum depth of the tree. We will try [5, 10, 15] to control overfitting. **Why**: Deeper trees capture more complex patterns but can memorize noise (overfit). Shallower trees generalize better but might underfit.
- `min_samples_split`: Minimum number of samples required to split an internal node. We will try [2, 5]. **Why**: Higher values prevent the model from learning distinct rules for very small groups of samples, further reducing overfitting.

In [20]:
```
# Feature Engineering
ri_custom_valid_rating['calories_per_minute'] = ri_custom_valid_rating['c
ri_custom_valid_rating['complexity_density'] = ri_custom_valid_rating['n_

from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import QuantileTransformer

features = ['n_steps', 'calories (#)', 'minutes', 'n_ingredients', 'total
X_final = ri_custom_valid_rating[features]
y_final = ri_custom_valid_rating['rating_avg']

X_train_final, X_test_final, y_train_final, y_test_final = train_test_spl

preprocessing_final = ColumnTransformer(
    transformers=[
        ('quantile', QuantileTransformer(output_distribution='normal'), [
        ('scaling', StandardScaler(), ['n_steps', 'n_ingredients', 'total
    ],
    remainder='drop'
)

final_pipeline = Pipeline([
    ('preprocessing', preprocessing_final),
    ('regression', RandomForestRegressor(random_state=42))
])

param_grid = {
    'regression__n_estimators': [50, 100],
    'regression__max_depth': [5, 10, 15],
    'regression__min_samples_split': [2, 5]
}

grid_search = GridSearchCV(final_pipeline, param_grid, cv=3, scoring='neg
grid_search.fit(X_train_final, y_train_final)

best_model = grid_search.best_estimator_

print("Best hyperparameters:", grid_search.best_params_)
```

```
y_pred_train_final = best_model.predict(X_train_final)
y_pred_test_final = best_model.predict(X_test_final)

rmse_train_final = np.sqrt(mean_squared_error(y_train_final, y_pred_train
rmse_test_final = np.sqrt(mean_squared_error(y_test_final, y_pred_test_fi

print(f'Final Model Train RMSE: {rmse_train_final}')
print(f'Final Model Test RMSE: {rmse_test_final}')

try:
    print(f"Improvement over Baseline (Test RMSE): {rmse_test - rmse_test
except NameError:
    print("Baseline RMSE variable not found for comparison.")
```

```
/var/folders/ng/bts1b1qn5_gfyd59gql9sf2m0000gn/T/ipykernel_44923/10873922
1.py:2: SettingWithCopyWarning:


A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-doc
s/stable/user_guide/indexing.html#returning-a-view-versus-a-copy

/var/folders/ng/bts1b1qn5_gfyd59gql9sf2m0000gn/T/ipykernel_44923/10873922
1.py:3: SettingWithCopyWarning:


A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-doc
s/stable/user_guide/indexing.html#returning-a-view-versus-a-copy

Best hyperparameters: {'regression__max_depth': 15, 'regression__min_sampl
es_split': 2, 'regression__n_estimators': 100}
Final Model Train RMSE: 0.4180491592168629
Final Model Test RMSE: 0.45552845954473187
Improvement over Baseline (Test RMSE): 0.04174073067144107
```

Step 8: Fairness Analysis

Fairness Analysis

Question: Does our model perform differently for "Short" recipes (< 30 minutes) vs. "Long" recipes (>= 30 minutes)?

Groups:

- **Group X:** Short recipes (`minutes` < 30)
- **Group Y:** Long recipes (`minutes` >= 30)

Evaluation Metric: Root Mean Squared Error (RMSE)

Hypotheses:

- **Null Hypothesis:** The model's RMSE for Short and Long recipes are roughly the same, and any differences are due to random chance.
- **Alternative Hypothesis:** The model's RMSE for Short and Long recipes are significantly different.

Test Statistic: Absolute Difference in RMSE.

In [21]:
```
# Calculate RMSE to quantify prediction error
test_results = X_test_final.copy()
test_results['rating_actual'] = y_test_final
test_results['rating_pred'] = y_pred_test_final

test_results['is_short'] = test_results['minutes'] < 30

def calculate_rmse(df):
    return np.sqrt(mean_squared_error(df['rating_actual'], df['rating_pre

rmse_short = calculate_rmse(test_results[test_results['is_short']])
rmse_long = calculate_rmse(test_results[~test_results['is_short']])

observed_diff_rmse = abs(rmse_short - rmse_long)

print(f"RMSE (Short): {rmse_short}")
print(f"RMSE (Long): {rmse_long}")
print(f"Observed Absolute Difference: {observed_diff_rmse}")

n_permutations = 500
simulated_diffs_rmse = []

shuffled_labels_pool = test_results['is_short'].values

for _ in range(n_permutations):
    shuffled_labels = np.random.permutation(shuffled_labels_pool)

    group_a = test_results[shuffled_labels]
    group_b = test_results[~shuffled_labels]

    rmse_a = calculate_rmse(group_a)
    rmse_b = calculate_rmse(group_b)

    simulated_diffs_rmse.append(abs(rmse_a - rmse_b))

p_value_fairness = (np.array(simulated_diffs_rmse) >= observed_diff_rmse)
print(f"P-value: {p_value_fairness}")

if p_value_fairness < 0.05:
    print("Reject the null, the model is unfair with respect to recipe du
else:
    print("Fail to reject the null hypothesis, we do not have evidence th

fig = px.histogram(pd.DataFrame({'diffs': simulated_diffs_rmse}), x='diff
fig.add_vline(x=observed_diff_rmse, line_color='red', annotation_text='Ob
fig.show()
```

```
RMSE (Short): 0.4222119984271988
RMSE (Long): 0.47467116210367977
Observed Absolute Difference: 0.05245916367648096
P-value: 0.0
Reject the null, the model is unfair with respect to recipe duration.
```

In [ ]: