template

March 21, 2024

1 Recipe Data Analytics

Name(s): Daniel Warren

Website Link: https://danielpwarren.github.io/Recipe-Data-Analytics

```
[ ]: import pandas as pd
     import numpy as np
     from pathlib import Path
     import ast  # To convert strings that resemble lists into actual lists
     from itertools import chain
     from sklearn.feature_extraction.text import TfidfVectorizer
     import math

     import plotly.express as px

     pd.options.plotting.backend = "plotly"
```

1.1 Step 1: Introduction

1.1.1 Brainstormed Questions

- How do the ingredients and nutrition contribute to the rating of a recipe?
- How do the number of steps and expected time contribute to the rating?
- Are certain food categories (such as desert or dinner) more likely to recieve higher ratings?
- What terms in the description or tags correlate to higher ratings?
- What terms in the reviews correlate to higher ratings?

1.1.2 Selected Question

How do the number of steps and expected time contribute to the rating?

1.1.3 Loading the data:

```
[ ]: recipes_raw = pd.read_csv(Path("data") / Path("RAW_recipes.csv"))
     interactions_raw = pd.read_csv(Path("data") / Path("RAW_interactions.csv"))
```

```
[ ]: interactions_raw.columns
```

```
[ ]: Index(['user_id', 'recipe_id', 'date', 'rating', 'review'], dtype='object')
```

1.2 Step 2: Data Cleaning and Exploratory Data Analysis

1.2.1 Data Cleaning

In this step I will:

- combine the recipies and interactions dataframes into one dataframe.
- seperate `tags`, `steps`, and `ingredients` into arrays
- replace 0 star ratings with NaN and calculate an average rating for each recipe and store it in `avg_rating`
- change the datatypes of rows where applicable

[ ]:
```
interactions_raw["rating"] = interactions_raw["rating"].replace(0, np.nan)

recipes_raw["tags"] = recipes_raw["tags"].apply(ast.literal_eval)
recipes_raw["steps"] = recipes_raw["steps"].apply(ast.literal_eval)
recipes_raw["ingredients"] = recipes_raw["ingredients"].apply(ast.literal_eval)
recipes_raw["nutrition"] = recipes_raw["nutrition"].apply(ast.literal_eval)

merged = recipes_raw.merge(
    interactions_raw, left_on="id", right_on="recipe_id", how="left"
)
merged.head()

avg_rating = merged.groupby("id")["rating"].mean()
avg_rating
merged = merged.set_index("id")
merged["avg_rating"] = avg_rating
recipes = merged.reset_index()

recipes["date"] = pd.to_datetime(recipes["date"])
recipes["submitted"] = pd.to_datetime(recipes["submitted"])
recipes.drop(columns=["recipe_id"], inplace=True)

nutrient_names = [
    "calories",
    "total fat (PDV)",
    "sugar (PDV)",
    "sodium (PDV)",
    "protein (PDV)",
    "saturated fat (PDV)",
    "carbohydrates (PDV)",
]
for index, nutrient in enumerate(nutrient_names):
    recipes[nutrient] = recipes["nutrition"].apply(lambda x: x[index])
    recipes[nutrient] = pd.to_numeric(recipes[nutrient])
```

```python
print(recipes.dtypes)
```

```
id                                int64
name                             object
minutes                           int64
contributor_id                    int64
submitted                datetime64[ns]
tags                             object
nutrition                        object
n_steps                           int64
steps                            object
description                      object
ingredients                      object
n_ingredients                     int64
user_id                         float64
date                     datetime64[ns]
rating                          float64
review                           object
avg_rating                      float64
calories                        float64
total fat (PDV)                 float64
sugar (PDV)                     float64
sodium (PDV)                    float64
protein (PDV)                   float64
saturated fat (PDV)             float64
carbohydrates (PDV)             float64
dtype: object
```

Note: `user_id`, `rating`, and `avg_rating` here all need to be floats as the columns contain NaN values that cant be represented as integers.

```python
[ ]: for col in recipes.columns:
         print(f"{col}: {type(recipes[col][0])}")
```

```
id: <class 'numpy.int64'>
name: <class 'str'>
minutes: <class 'numpy.int64'>
contributor_id: <class 'numpy.int64'>
submitted: <class 'pandas._libs.tslibs.timestamps.Timestamp'>
tags: <class 'list'>
nutrition: <class 'list'>
n_steps: <class 'numpy.int64'>
steps: <class 'list'>
description: <class 'str'>
ingredients: <class 'list'>
n_ingredients: <class 'numpy.int64'>
user_id: <class 'numpy.float64'>
date: <class 'pandas._libs.tslibs.timestamps.Timestamp'>
```

```
rating: <class 'numpy.float64'>
review: <class 'str'>
avg_rating: <class 'numpy.float64'>
calories: <class 'numpy.float64'>
total fat (PDV): <class 'numpy.float64'>
sugar (PDV): <class 'numpy.float64'>
sodium (PDV): <class 'numpy.float64'>
protein (PDV): <class 'numpy.float64'>
saturated fat (PDV): <class 'numpy.float64'>
carbohydrates (PDV): <class 'numpy.float64'>
```

1.2.2 Univariate Analysis

In this step I will observe the distributions of relevant columns in the dataframe

```
[ ]: fig1 = px.bar(recipes["rating"].value_counts(), y="rating")
     fig1.update_layout(title="Distribution of ratings")
     fig1.update_xaxes(title="rating")
     fig1.update_yaxes(title="count")
     fig1.show()
```

```
[ ]: fig2 = px.bar(recipes[recipes["n_steps"] <= 40]["n_steps"].value_counts(),␣
      ↪y="n_steps")
     fig2.update_layout(title="Distribution of number of steps")
     fig2.update_xaxes(title="steps")
     fig2.update_yaxes(title="count")
     fig2.show()
```

```
[ ]: bins = np.arange(0, 12 * 60 + 15, 15)
     bin_indices = np.digitize(recipes[recipes["minutes"] <= 12 * 60]["minutes"],␣
      ↪bins) - 1
     result = np.bincount(bin_indices)
     df = pd.DataFrame({"minute_bin": bins, "count": result})
     fig3 = px.bar(
         df, x="minute_bin", y="count", title="Distribution of Minutes in 15-Minute␣
      ↪Bins"
     )
     fig3.update_xaxes(title="minutes")
     fig3.update_yaxes(title="count")
     fig3.show()
```

```
[ ]: all_tags = list(chain.from_iterable(recipes["tags"]))
     tag_counts = pd.Series(all_tags).value_counts()

     percentile_threshold = tag_counts.quantile(0.90)
     truncated_counts = tag_counts[tag_counts >= percentile_threshold]
     truncated_counts
```

```
fig4 = px.bar(truncated_counts)
fig4.update_layout(title="Distribution of most common tags (top 10%)")
fig4.update_xaxes(title="tag")
fig4.update_yaxes(title="count")
fig4.show()
```

1.2.3 Bivariate Analysis

[ ]:
```
fig4 = px.scatter(recipes, x="avg_rating", y="n_steps")
fig4.show()
```

[ ]:
```
recipes["truncated_minutes"] = recipes[recipes["minutes"] <= 12 * 60]["minutes"]
fig5 = px.scatter(recipes, x="avg_rating", y="truncated_minutes")
fig5.show()
```

1.2.4 Interesting Aggregates

[ ]:
```
recipes_df = recipes.copy()

recipes_df = recipes_df[recipes_df["minutes"] <= 200]  # get rid of outliers
pivot_table = recipes_df.pivot_table(
    values="n_steps", index="n_ingredients", aggfunc=["mean", "median", "min",␣
↪"max"]
)

pivot_table
```

[ ]:
```
                    mean  median     min     max
                 n_steps n_steps n_steps n_steps
n_ingredients
1               6.750000       7       2      20
2               5.969932       5       1      55
3               5.539765       5       1      69
4               6.493830       5       1      55
5               7.329812       6       1      80
6               7.740900       7       1      86
7               8.297483       8       1      52
8               9.052886       8       1      67
9               9.759984       9       1      87
10             10.698326      10       1      57
11             11.187419      10       1      65
12             11.782152      11       1      57
13             12.549089      11       1      68
14             13.540655      12       1      88
15             14.356427      13       2      62
16             15.142857      14       2      77
17             15.656934      15       1      55
```

```
18         16.410724   15     3    65
19         16.675875   16     3    59
20         17.226623   17     2    76
21         18.733615   17     3    48
22         21.721402   27     4    59
23         17.753676   17     5    59
24         17.673203   17     4    44
25         19.140000   18     6    45
26         18.948454   21     5    49
27         18.088235   19     7    34
28         23.942857   23     5    76
29         18.551724   17    15    33
30         27.906250   23     6    62
31         18.916667   20    10    29
32         37.000000   40    28    40
33          6.000000    6     6     6
```

1.3 Step 3: Assessment of Missingness

1.3.1 NMAR Analysis

We will consider the `review` column. This column is the most likely to have NMAR missing data, since recipes with medium reviews could be correlated with missing review text.

1.3.2 Missingness Dependency

We will further analyze the `rating` column, against the `n_ingredients` and `minutes` coumns

```
[ ]: recipes["rating"].isna().mean()
```

```
[ ]: 0.06413882241531552
```

```
[ ]: def dependency_test(df, col):
         print(f"Testing dependency of rating on {col}")
         observed_diff = abs(
             df[col][df["rating"].isna()].mean() - df[col][~df["rating"].isna()].
      ↪mean()
         )
         sim_diff = []
         shuffled = df.copy()
         for _ in range(1000):
             shuffled[col] = np.random.permutation(shuffled[col])
             sim_diff.append(
                 abs(
                     shuffled[col][shuffled["rating"].isna()].mean()
                     - shuffled[col][~shuffled["rating"].isna()].mean()
                 )
             )
```

```
    print(observed_diff)
    print(f"{sim_diff[:5]}...{sim_diff[-5:]}")
    print((sim_diff >= observed_diff).mean())


dependency_test(recipes, "n_steps")
dependency_test(recipes, "n_ingredients")
dependency_test(recipes, "minutes")
```

```
Testing dependency of rating on n_steps
1.3386412335909217
[0.003422808128130228, 0.029716887844099205, 0.026874284631562162,
0.03580511620487847, 0.0012197906384159296]…[0.03516553048205928,
0.028414347852283584, 0.0817131580873518, 0.09159120425091771,
0.04407203406741189]
0.0
Testing dependency of rating on n_ingredients
0.1607379066254797
[0.022823195824098974, 0.03227485150578424, 0.00022450028443010694,
0.0763125912131315, 0.03253650174005962]…[0.008610179761413761,
0.059825492580415585, 0.07062738478805564, 0.004963250578450484,
0.05494461212576773]
0.0
Testing dependency of rating on minutes
51.45237039852127
[3.538001943248858, 10.787350786021463, 5.28527907291506, 29.27208882634673,
3.7211366552165543]…[21.879845332454295, 20.743497472852155,
16.779131967567665, 62.03786943660768, 9.079514775929198]
0.114
```

1.4 Step 4: Hypothesis Testing

The hypotheses I plan to test are:

H_0: The number of steps and the expected time to complete a task have no effect on the task's rating. Any observed differences in ratings due to changes in the number of steps and expected time are purely due to chance.

H_1: The number of steps and the expected time to complete a task both negatively affect the task's rating. Tasks that have a higher number of steps and longer expected times to complete will have lower ratings, not due to chance.

```
[ ]: def permutation_test_correlation(df, col, rating_col="avg_rating"):
         clean_df = df.dropna(subset=[col, rating_col])

         print(f"Testing dependency of {rating_col} on {col}")
         observed_corr = clean_df[[col, rating_col]].corr().iloc[0, 1]

         sim_corr = []
```

```
    for _ in range(1000):

        shuffled_rating = np.random.permutation(clean_df[rating_col].values)

        sim_corr_value = np.corrcoef(clean_df[col].values, shuffled_rating)[0,␣
↪1]
        if not np.isnan(sim_corr_value):
            sim_corr.append(sim_corr_value)

    p_value = np.mean(np.abs(sim_corr) >= np.abs(observed_corr))
    print(f"Observed correlation: {observed_corr}")
    print(f"P-value: {p_value}")


permutation_test_correlation(recipes, "n_steps")
permutation_test_correlation(recipes, "minutes")
```

```
Testing dependency of avg_rating on n_steps
Observed correlation: -0.0018214513463115766
P-value: 0.387
Testing dependency of avg_rating on minutes
Observed correlation: 0.00196393037826293
P-value: 0.259
```

1.5 Step 5: Framing a Prediction Problem

I plan to predict the calories of each recipe.

The features I'll use for this are as follows - `total fat (PDV)` - `protein (PDV)`

1.6 Step 6: Baseline Model

We will begin with a very rudimentary baseline model. We'll use a simple LinearRegression model and train it to predict calories based only on the recipes total fat and sugar.

```
[ ]: from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

data = recipes[["total fat (PDV)", "protein (PDV)", "calories"]].
↪drop_duplicates()

X = data.drop(["calories"], axis=1)
y = data["calories"]

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

```
baseline_pipeline = Pipeline(
    [
        ("regressor", LinearRegression()),
    ]
)

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

mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

(mse, r2)
```

```
[ ]: (156606.05998095873, 0.6582566526599032)
```

1.7 Step 7: Final Model

Here I create new features by dividing calories against the gram measurements of protein, carbs, and fat.

```
[ ]: from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import Lasso
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import Binarizer, PolynomialFeatures

feature_engineering = ColumnTransformer(
    [
        # Fitting a polynomial maps to the total fat should increase the
        # accuracy of the model as it carries more calories
        ("poly_fat", PolynomialFeatures(degree=2), ["total fat (PDV)"]),
        # As well, since the fat seems to be the most important feature, we can
        # binarize it to see if it has a significant impact on the model
        ("high_fat", Binarizer(threshold=21.0), ["total fat (PDV)"]),
    ],
    remainder="passthrough",
)

final_model_pipeline = Pipeline(
    [
        ("feature_engineering", feature_engineering),
        ("regressor", Lasso()),
    ]
)

param_grid = {
    "regressor__alpha": [0.0001, 0.001, 0.01, 0.1],
    "regressor__max_iter": [1000, 10000, 100000],
    "regressor__tol": [0.0001, 0.001, 0.01, 0.1],
    "regressor__fit_intercept": [True, False],
}

grid_search = GridSearchCV(final_model_pipeline, param_grid, cv=5, scoring="r2")
grid_search.fit(X_train, y_train)

best_params = grid_search.best_params_
best_score = grid_search.best_score_

print("Best Parameters:", best_params)
print("Best R-squared:", best_score)

y_pred = grid_search.predict(X_test)
new_mse = mean_squared_error(y_test, y_pred)
new_r2 = r2_score(y_test, y_pred)
print("Test MSE:", mse)
print("Test R-squared:", r2)
```

```
Best Parameters: {'regressor__alpha': 0.1, 'regressor__fit_intercept': False,
'regressor__max_iter': 1000, 'regressor__tol': 0.01}
Best R-squared: 0.7983434554034269
Test MSE: 156606.05998095873
Test R-squared: 0.6582566526599032
```

```
[ ]: feature_engineering = ColumnTransformer(
         [
             ("poly_fat", PolynomialFeatures(degree=2), ["total fat (PDV)"]),
             ("high_fat", Binarizer(threshold=33.6), ["total fat (PDV)"]),
         ],
         remainder="passthrough",
     )

     final_model_pipeline = Pipeline(
         [
             ("feature_engineering", feature_engineering),
             ("regressor", Lasso(alpha=0.1, max_iter=1000, tol=0.01,␣
       ↪fit_intercept=False)),
         ]
     )

     final_model_pipeline.fit(X_train, y_train)
```

```
y_pred = final_model_pipeline.predict(X_test)
new_mse = mean_squared_error(y_test, y_pred)
new_r2 = r2_score(y_test, y_pred)

print("Test MSE:", new_mse)
print("Test R-squared:", new_r2)
print("MSE Improvement:", mse - new_mse)
print("R-squared Improvement:", new_r2 - r2)
```

```
Test MSE: 155977.12096335526
Test R-squared: 0.659629113758629
MSE Improvement: 628.9390176034649
R-squared Improvement: 0.0013724610987257968
```

1.8 Step 8: Fairness Analysis

The fairness analysis for this model is an interesting one as the model I ended up with is very simple. Still, we will create a group using the mean `protein (PDV)` as this seems to be a less decisive element in our model so it will be interesting to see if the models RMSE for both sides of the value are the same.

H_0: The model is fair, its mean RMSE for values lower than the mean protein in the dataset is close the the RMSE of values greater than the mean protein

H_1: The model is not fair and there is a bias towards either lower or higher protein values

[ ]:
```
from sklearn.metrics import mean_squared_error
import numpy as np


protein_median = data["protein (PDV)"].mean()

high_protein_test = X_test[
    data.loc[X_test.index, "protein (PDV)"] > protein_median
]
low_protein_test = X_test[
    data.loc[X_test.index, "protein (PDV)"] <= protein_median
]

y_test_high_protein = y_test[high_protein_test.index]
y_test_low_protein = y_test[low_protein_test.index]

y_pred_high_protein = final_model_pipeline.predict(high_protein_test)
y_pred_low_protein = final_model_pipeline.predict(low_protein_test)

rmse_high_calorie = np.sqrt(
    mean_squared_error(y_test_high_protein, y_pred_high_protein)
)
```

```
rmse_low_calorie = np.sqrt(mean_squared_error(y_test_low_protein,␣
 ↪y_pred_low_protein))

observed_diff = rmse_high_calorie - rmse_low_calorie

rmse_high_calorie, rmse_low_calorie, observed_diff
```

```
[ ]: (625.5672862708465, 168.57409034869306, 456.9931959221534)
```

```
[ ]: def permutation_test(
         y_true_high, y_pred_high, y_true_low, y_pred_low, n_permutations=1000
     ):
         y_concat = np.concatenate([y_true_high, y_true_low])

         observed_rmse_diff = np.sqrt(
             mean_squared_error(y_true_high, y_pred_high)
         ) - np.sqrt(mean_squared_error(y_true_low, y_pred_low))

         permuted_diffs = []

         for _ in range(n_permutations):
             np.random.shuffle(y_concat)

             y_perm_high = y_concat[: len(y_true_high)]
             y_perm_low = y_concat[len(y_true_high) :]

             rmse_perm_high = np.sqrt(mean_squared_error(y_perm_high, y_pred_high))
             rmse_perm_low = np.sqrt(mean_squared_error(y_perm_low, y_pred_low))

             permuted_diffs.append(rmse_perm_high - rmse_perm_low)

         p_value = np.mean(np.abs(permuted_diffs) >= np.abs(observed_rmse_diff))

         return p_value


     p_value = permutation_test(
         y_test_high_protein,
         y_pred_high_protein,
         y_test_low_protein,
         y_pred_low_protein,
         n_permutations=10000,
     )
     print(f"P-value from permutation test: {p_value}")
```

```
P-value from permutation test: 0.0933
```