How Long Does it Take to Make?

An quantitative analysis using Food.com's recipe data to create a predictive model to determine the amount of time a given recipe would take to complete.

By: Susana Haing, Rocio Saez, Minnie Zhang

Today's agenda

- Baseline Model
- Exploratory Data Analysis & Data Cleaning
- Modeling
- Final Model
- Evaluation
- Discussion

Research Question

What factors play a role in determining how long it takes to finish making a recipe?

Baseline Model

```
df_filtered = df[df["minutes"] < 240].copy()

X_numeric = df_filtered[["n_steps", "n_ingredients"]].values

y = df_filtered["minutes"].values

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

baseline_model = Ridge(alpha=1.0)
baseline_model.fit(X_train, y_train)

y_pred = baseline_model.predict(X_test)

rmse = np.sqrt(mean_squared_error(y_test, y_pred))
r2 = r2_score(y_test, y_pred)

print("\n===== Baseline Linear Model =====")
print(f"Baseline R²:   {r2:.4f}")
print(f"Baseline RMSE: {rmse:.4f}")
```

Baseline Model Results

RMSE: 34.8861

R^2: 0.0886

Model: Simple Linear Regression

Features: number of steps and number of ingredients

The most intuitive thing when it comes to approximating how long a recipe could take it based off the number of steps and ingredients that are needed to prepare the dish. Therefore, our main baseline model uses the number of steps and number of ingredients as the only features. As the basal assumption is that the number of steps and ingredients has a positive linear relationship with the recipe time, we started with a simple linear regression model.

Exploratory Data Analysis & Data Cleaning

Section II

Recipe Time Prediction

Data - metadata csv

Raw Recipes

- ID: A recipe's unique ID
- Name: The recipe's name
- N_steps: Number of steps listed in a recipe
- Minutes: How long the recipe takes to complete in minutes
- Tags: Keywords and phrases used to describe the recipe
- Steps: Description of the recipe's step by step instructions
- Ingredients: A list of ingredients used in the recipe

PP Recipes

- ID: A recipe's unique ID
- Technique: An array of binary values, tracking whether a technique was used or not for the given recipe

Context

The data is pulled from Food.com's online recipe aggregator
Containing 180K+ recipes and 700k+ reviews collected over 18 years

Used in "Generating Personalized Recipes from Historical User Preferences"

Data Cleaning and Processing

Merged raw_recipes.csv and pp_recipes.csv to create 1 dataframe that contained only variables that we were interested in

```
rr_final = raw_recipes[['id', 'n_steps', 'n_ingredients', 'minutes', 'name', 'tags', 'steps', 'ingredients']].copy()
pp_final = recipes[['id', 'techniques']].copy()

# Clean 'id' column: remove anything that's NOT a digit
pp_final['id'] = pp_final['id'].astype(str).apply(lambda x: re.sub(r'\D', '', x))

# Drop rows where id became empty after cleaning (they were invalid)
pp_final = pp_final[pp_final['id'].apply(len) > 0]

# Convert both to int
rr_final['id'] = rr_final['id'].astype(int)
pp_final['id'] = pp_final['id'].astype(int)

final_recipes = rr_final.merge(pp_final, on='id')
final_recipes
```

Data Cleaning

- Filtered out extreme cooking times (<240 minutes per IQR)
- Dropped NAN values
- Converted tags variable into string literals from array of strings
- Filtered out any tags that were not indicative of the recipe's time (i.e holidays, miscellaneous, regional cuisine names)

Data Cleaning and Processing

```
df = final_recipes.copy()

region_tags = {
    'american', 'asian', 'australian','amish-mennonite', 'angolan', 'austrian', 'belgian', 'brazilian',
    'british-columbian', 'canadian', 'cantonese', 'caribbean', 'chinese',
    'chilean', 'cuban', 'danish', 'dutch', 'english', 'ethiopian', 'finnish',
    'french', 'german', 'greek', 'hungarian', 'icelandic', 'indian',
    'indonesian', 'iranian-persian', 'irish', 'italian', 'japanese',
    'jewish-ashkenazi', 'jewish-sephardi', 'korean', 'laotian', 'lebanese',
    'malaysian', 'mexican', 'moroccan', 'norwegian', 'polish', 'portuguese',
    'russian', 'scandinavian', 'scottish', 'somalian', 'spanish', 'swedish',
    'thai', 'turkish', 'vietnamese', 'african', 'argentine', 'cajun',
    'central-american', 'south-american', 'midwestern', 'middle-eastern',
    'pakistani', 'peruvian', 'philippine', 'north-american', 'beijing', 'californian',
    'cambodian', 'colombian', 'creole', 'czech', 'costa-rican', 'ecuadorean', 'egyptian',
    'filipino', 'georgian', 'guatemalan', 'hawaiian', 'hunan', 'honduran', 'iraqi',
    'micro-melanesia', 'mongolian', 'namibian', 'native-american', 'nepalese', 'new-zealand', 'nigerian',
    'north-eastern-united-states', 'oaxacan', 'palestinian', 'ontario', 'polynesian',
    'puerto-rican', 'quebec', 'saudi-arabian', 'south-african', 'south-west-pacific', 'southern-united-states',
    'southwestern-united-states', 'st-patricks-day', 'sudanese', '  szechuan', 'tex-mex', 'venezuelan', 'welsh'

}

holiday_tags = {
    'christmas', 'thanksgiving', 'halloween', 'easter', 'hanukkah',
    'valentines-day', 'mothers-day', 'fathers-day', 'kwanzaa',
    'new-years', 'ramadan', 'rosh-hashana', 'rosh-hashanah',
    'super-bowl', 'superbowl', 'memorial-day', 'labor-day',
    'independence-day', 'cinco-de-mayo', 'april-fools-day',
    'birthday', 'wedding', 'chinese-new-year', 'holiday-event', 'irish-st-patricks-day',
    'mardi-gras-carnival'
}
```

Data Cleaning and Processing

```
misc_tags = {
    'bear'
}

time_tags = {
    '1-day-or-more', '15-minutes-or-less', '30-minutes-or-less',
'4-hours-or-less', '60-minutes-or-less'
}

drop_tags = time_tags


def parse_tags(x):
    if isinstance(x, list):
        return x
    elif isinstance(x, str):
        try:
            parsed = ast.literal_eval(x)
            if isinstance(parsed, list):
                return parsed
            # fallback if literal_eval returns a string
            return [parsed]
        except:
            # fallback: split on commas (best-effort)
            return [s.strip().strip("'\"") for s in x.split(',') if s.strip()]
    return []

df["tags"] = df["tags"].apply(parse_tags)

df["tags"] = df["tags"].apply(lambda tag_list: [t for t in tag_list if t not in drop_tags])
df = df[df["tags"].apply(len) > 0].reset_index(drop=True)
```

Data Cleaning and Processing

```
df = df.reset_index(drop=True)

mlb = MultiLabelBinarizer()
tag_matrix = mlb.fit_transform(df["tags"])

tag_features = pd.DataFrame(tag_matrix, columns=mlb.classes_, index=df.index)


final_model_df = pd.concat([df.drop(columns=["tags"]), tag_features], axis=1)

rf_df = final_model_df
final_model_df.head()
```

Data Processing

- Kept numeric features such as n_steps and n_ingredients and engineered additional interaction features (i.e prep_complexity, ingredient_step_ratio)
- Extracted number of techniques
- Converted cleaned tags into a multi-hot encoded matrix using MultiLabelBinarizer
- Created text based features from recipe names (i.e has_slow_word as in terms with a long cooking time such as braising, marinating, fermenting)
- Converted ingredients lists into string literals for pattern matching and binarizing of ingredient categories

Data Processing

This is not representative of all preprocessing in our project.

Pulled from an improved multiple linear regression model that implemented the text from steps and ingredients as features

```
df_filtered["steps_text"] = df_filtered["steps"].apply(lambda lst: " ".join(lst).lower())
df_filtered["steps_word_count"] = df_filtered["steps_text"].apply(lambda x: len(x.split()))
df_filtered["steps_char_count"] = df_filtered["steps_text"].apply(lambda x: len(x))
df_filtered["steps_unique_words"] = df_filtered["steps_text"].apply(lambda x: len(set(x.split())))

df_filtered["ingredients"] = df_filtered["ingredients"].apply(
    lambda x: ast.literal_eval(x) if isinstance(x, str) else x
)
df_filtered["ingredients_text"] = df_filtered["ingredients"].apply(lambda lst: " ".join(lst).lower())
df_filtered["ingredients_word_count"] = df_filtered["ingredients_text"].apply(lambda x: len(x.split()))
df_filtered["ingredients_char_count"] = df_filtered["ingredients_text"].apply(lambda x: len(x))
df_filtered["ingredients_unique_words"] = df_filtered["ingredients_text"].apply(lambda x: len(set(x.split())))

df_filtered["ingredient_count"] = df_filtered["ingredients"].apply(len)
```

Exploratory Data Analysis

- Summary statistics
- Observed recipe time distribution
- Correlation matrices
- Regression scatter plots

EDA – Data Summary Statistics

```
df_before_filter = final_model_df
print("Dataset Shape:", df_before_filter .shape)
print("\nColumns:\n", df_before_filter .columns)
print("\nData Types:\n", df_before_filter .dtypes)
print("\nSummary statistics:\n", df_before_filter.describe())
```

```
df = df_before_filter [df_before_filter ['minutes'] < 240]
print("Dataset Shape:", df.shape)
print("\nColumns:\n", df.columns)
print("\nData Types:\n", df.dtypes)
print("\nSummary statistics:\n", df['minutes'].describe())
```

```
Summary statistics:
                  id        n_steps  n_ingredients       minutes  \
count  178265.000000  178265.000000  178265.000000  1.782650e+05
mean   213461.803007       8.994329       9.017984  1.214357e+04
std    138266.712754       4.065679       3.224206  5.086240e+06
min        38.000000       3.000000       4.000000  0.000000e+00
25%     94576.000000       6.000000       7.000000  2.000000e+01
50%    196312.000000       8.000000       9.000000  3.700000e+01
75%    320562.000000      11.000000      11.000000  6.500000e+01
max    537716.000000      37.000000      20.000000  2.147484e+09

                  3-steps-or-less  5-ingredients-or-less  \
count  178265.000000    178265.000000          178265.000000
mean        0.000449         0.167722               0.120422
std         0.021179         0.373620               0.325455
min         0.000000         0.000000               0.000000
25%         0.000000         0.000000               0.000000
50%         0.000000         0.000000               0.000000
75%         0.000000         0.000000               0.000000
max         1.000000         1.000000               1.000000
```

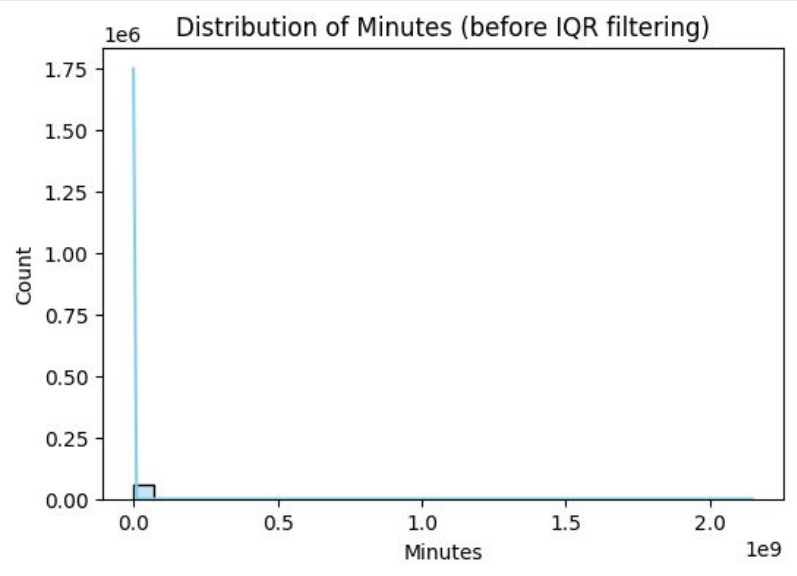
```
Summary statistics:
                  id        n_steps  n_ingredients       minutes  \
count  168604.000000  168604.000000  168604.000000  168604.000000
mean   213993.978079       9.010350       8.998244      44.836736
std    138396.462288       4.060681       3.211623      36.406773
min        40.000000       3.000000       4.000000       0.000000
25%     94944.750000       6.000000       7.000000      20.000000
50%    197121.500000       8.000000       9.000000      35.000000
75%    321379.500000      11.000000      11.000000      60.000000
max    537716.000000      37.000000      20.000000     235.000000

                  3-steps-or-less  5-ingredients-or-less  \
count  168604.000000    168604.000000          168604.000000
mean        0.000421         0.167327               0.120656
std         0.020517         0.373269               0.325727
min         0.000000         0.000000               0.000000
25%         0.000000         0.000000               0.000000
50%         0.000000         0.000000               0.000000
75%         0.000000         0.000000               0.000000
max         1.000000         1.000000               1.000000
```

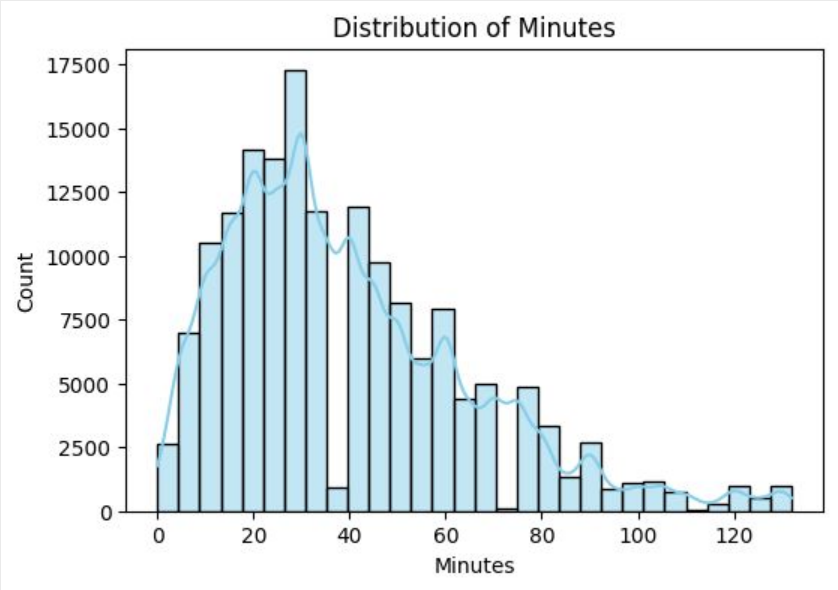
EDA – Observed Recipe Time Distribution

Histograms of Predicted Variable Minutes to determine skewness

Before IQR Filtering



After IQR Filtering



EDA – Correlation Matrix (Top 100)

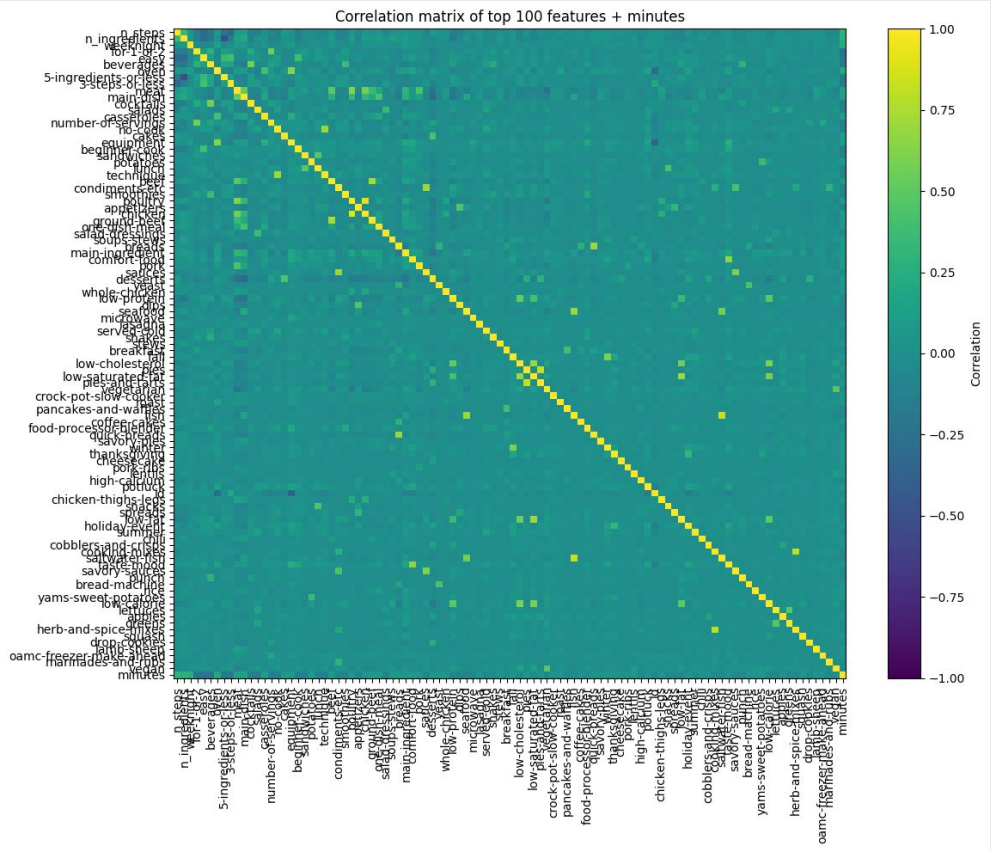
```
Regular Correlation

minutes          1.000000
n_steps          0.244341
weeknight        0.241205
n_ingredients    0.226491
yeast            0.187379
                    ...
irish            0.016447
soul             0.016369
manicotti        0.016227
pork-chops       0.015819
creole           0.015531
Name: minutes, Length: 100, dtype: float64
```

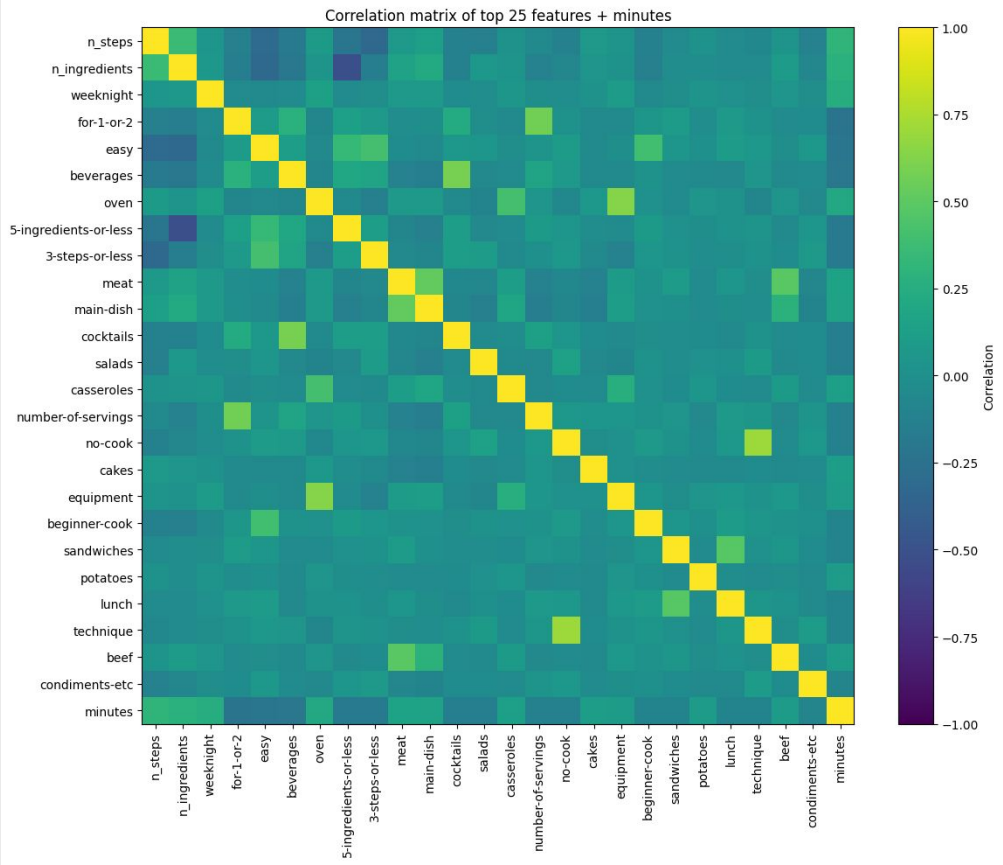
```
Absolute Correlation

minutes                  1.000000
n_steps                  0.244341
weeknight                0.241205
n_ingredients            0.226491
for-1-or-2               0.191533
                            ...
pork-loins               0.032506
served-cold              0.031559
tuna                     0.031486
herb-and-spice-mixes     0.031467
whole-turkey             0.031409
Name: minutes, Length: 100, dtype: float64
```

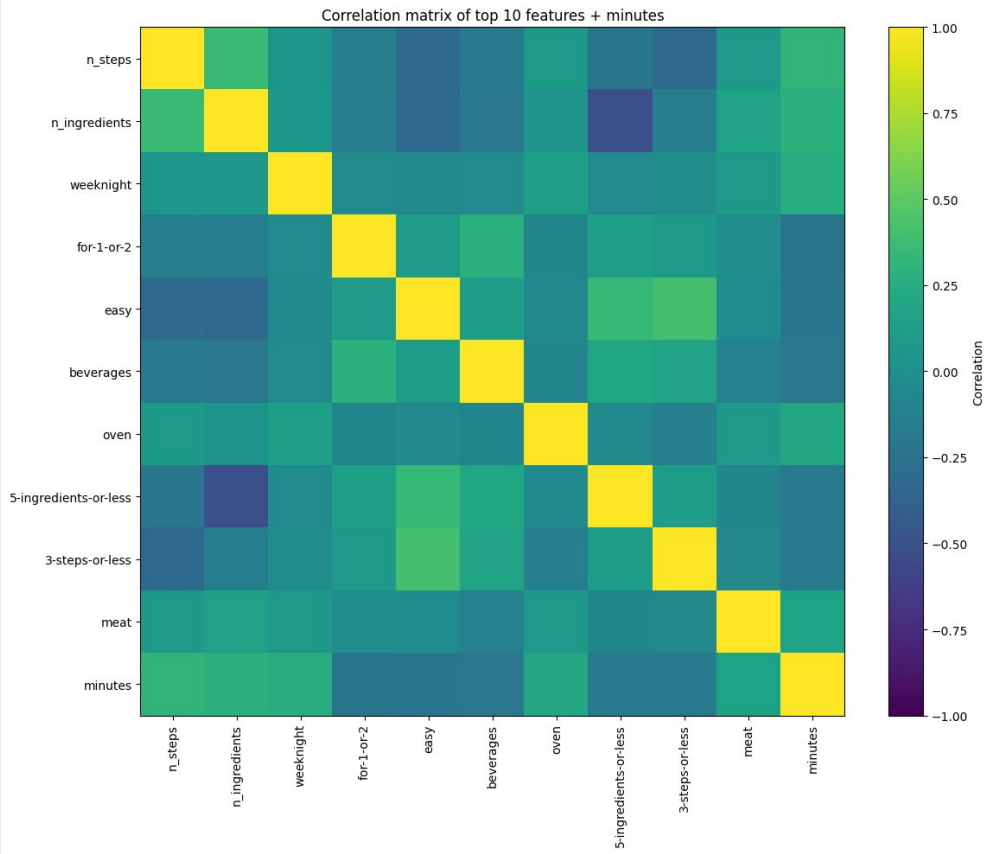
EDA – Correlation Matrix (Top 100)



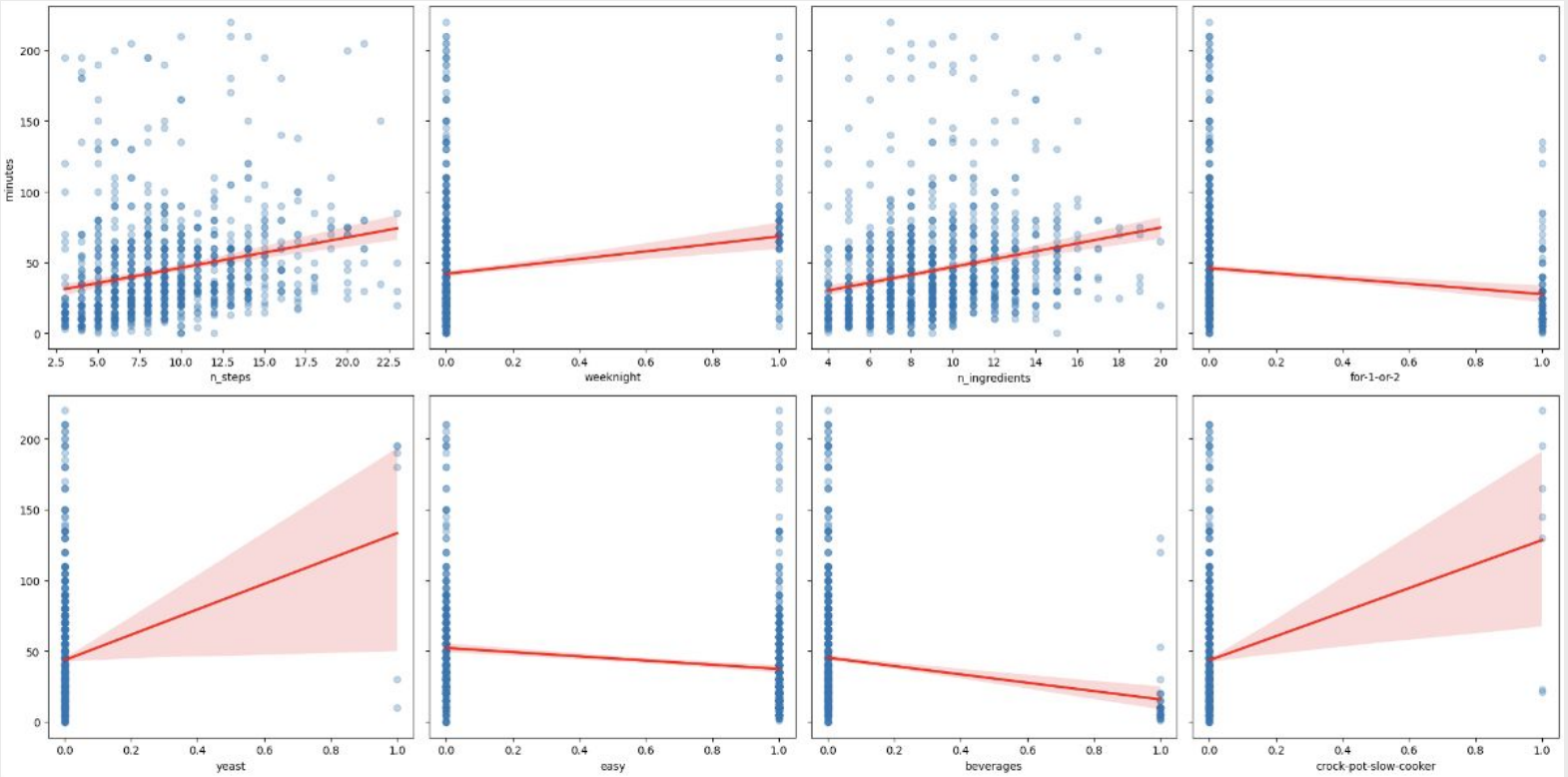
EDA – Correlation Matrix (Top 25)



EDA – Correlation Matrix (Top 10)



EDA – Regression Line Scatter Plots (Top 8)



Modeling

Section III

Recipe Time Prediction

Modeling Context

The predictive task is formulated as a regression problem, where the goal is to predict total recipe cooking time

Inputs:

- Numeric structure features
- Text derived numeric summaries
- Multi-hot encoded tag features
- Cooking techniques
- Binary ingredient category signals

Output: Total cooking time in minutes

Optimization Target: Minimize error between predicted time and true cooking time

Evaluation:

- R^2: explained variance
- RMSE: Root mean squared error

Appropriate Models:

- Baseline: Linear regression models
  - OLS
  - Ridge
- Comparison: Tree-based models

Modeling Discussion

Linear Models

Advantages:

- Extremely efficient to train even with hundreds of features (tags multi-hot encoded added 270 features alone)
- Produces coefficients that allow us to validate logical relationships
- Minimal risk of overfitting

Disadvantages:

- Cannot capture non-linear relationships
- All features are weighted globally, even if only one recipe uses a certain ingredient or technique

Tree Based Models

Advantages:

- Can capture non-linear relationships

Disadvantages:

- Inaccurate results for linear relationships
- High computational cost

Modeling Discussion

Different Filtering Levels

We chose to test out our model on different amounts of filtering:

- < 240 minutes
- < 180 minutes
- < 120 minutes
- < 60 minutes

Although our baseline uses <240 minutes from the IQR, we decided to test out on smaller increments of recipes that were faster to make.

Linear Model Results (<240 minutes)

Baseline

```
===== Baseline Linear Model =====
Baseline R²:   0.1297
Baseline RMSE: 34.4797
```

Linear Regression with Shrunken Text Features

```
===== Linear Regression (with shrinkage) =====
RMSE: 29.0591
R²: 0.3643
```

Linear Regression with Ingredient Features

```
===== Linear Regression + Ingredient Features =====
RMSE: 17.4844
R²: 0.4563
```

Linear Model Results – Best Accuracy

Baseline (< 60 minutes)

```
===== Baseline Linear Model =====
Baseline R²:   0.1657
Baseline RMSE: 12.7869
```

Linear Regression with Shrunken Text Features (< 120 minutes)

```
===== Linear Regression (with shrinkage) =====
RMSE: 18.1458
R²: 0.4165
```

Linear Regression with Ingredient Features (< 60 minutes)

```
===== Linear Regression + Ingredient Features =====
RMSE: 10.2167
R²: 0.4841
```

Tree Based Model Data Prep

```python
dfs = {}
Xs = {}
ys = {}

cutoffs = [60, 120, 180, 240]

for max_minutes in cutoffs:
    df_f = rf_df[rf_df["minutes"] < max_minutes].copy().reset_index(drop=True)

    if isinstance(df_f["techniques"].iloc[0], str):
        df_f["techniques"] = df_f["techniques"].apply(ast.literal_eval)

    tech_arr = np.vstack(df_f["techniques"].to_numpy()).astype(int)
    tech_cols = [f"tech_{i}" for i in range(tech_arr.shape[1])]
    df_tech = pd.DataFrame(tech_arr, columns=tech_cols, index=df_f.index)

    df_f = pd.concat([df_f.drop(columns=["techniques"]), df_tech], axis=1)

    num_df = df_f.select_dtypes(include=[np.number])

    corr_with_minutes = num_df.corrwith(num_df["minutes"])
    abs_corr_sorted = corr_with_minutes.abs().sort_values(ascending=False)

    top_100_features = abs_corr_sorted.drop("minutes").head(100).index.tolist()
    print(f"\n=== cutoff {max_minutes} minutes ===")
    print("Top 5 of top 100 features:", top_100_features[:5])

    X = num_df[top_100_features].to_numpy()
    y = num_df["minutes"].to_numpy(dtype=float)

    print("X shape:", X.shape)
    print("y shape:", y.shape)

    # store them
    dfs[max_minutes] = df_f
    Xs[max_minutes] = X
    ys[max_minutes] = y
```

Why not feature engineering?

- It was tried, but unsuccessful!!

Feature Engineering

```
Under 240 Test R²:   0.3071
Under 240 Test RMSE: 30.1437
```

NO Feature Engineering

```
Under 240 Test R²:   0.3721
Under 240 Test RMSE: 28.6963
```

Choosing Tree Based Models

```python
df_f = rf_df[rf_df["minutes"] < 240].copy().reset_index(drop=True)

if isinstance(df_f["techniques"].iloc[0], str):
    df_f["techniques"] = df_f["techniques"].apply(ast.literal_eval)

tech_arr = np.vstack(df_f["techniques"].to_numpy()).astype(int)
tech_cols = [f"tech_{i}" for i in range(tech_arr.shape[1])]
df_tech = pd.DataFrame(tech_arr, columns=tech_cols, index=df_f.index)

df_f = pd.concat([df_f.drop(columns=["techniques"]), df_tech], axis=1)

num_df = df_f.select_dtypes(include=[np.number])

corr_with_minutes = num_df.corrwith(num_df["minutes"])
abs_corr_sorted = corr_with_minutes.abs().sort_values(ascending=False)

top_100_features = abs_corr_sorted.drop("minutes").head(100).index.tolist()
print(f"\n=== cutoff {240} minutes ===")
print("Top 5 of top 100 features:", top_100_features[:5])

X = num_df[top_100_features].to_numpy()
y = num_df["minutes"].to_numpy(dtype=float)

print("X shape:", X.shape)
print("y shape:", y.shape)
```

known

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

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```python
# try out as bunch of differnet models
models = {
    "LinearRegression": LinearRegression(),
    "Ridge": Ridge(alpha=1.0),
    "Lasso": Lasso(alpha=0.001),
    "RandomForest": RandomForestRegressor(
        n_estimators=300,
        max_depth=None,
        min_samples_leaf=1,
        n_jobs=-1,
        random_state=42,
    ),
    "GradientBoosting": GradientBoostingRegressor(random_state=42),
    "HistGradientBoosting": HistGradientBoostingRegressor(random_state=42),
    "KNN": KNeighborsRegressor(n_neighbors=10),
}

print("\n=== Model comparison on cleaned dataframe ===")
for name, model in models.items():
    if name in ["RandomForest", "GradientBoosting", "HistGradientBoosting"]:
        Xtr, Xte = X_train, X_test
    else:
        Xtr, Xte = X_train_scaled, X_test_scaled

    model.fit(Xtr, y_train)
    preds = model.predict(Xte)

    rmse = np.sqrt(mean_squared_error(y_test, preds))
    r2 = r2_score(y_test, preds)

    print(f"{name:20s}  RMSE={rmse:7.3f}  R²={r2:6.3f}")
```

Choosing Tree Based Models Results

```
=== cutoff 240 minutes ===
Top 5 of top 100 features: ['n_steps', 'weeknight', 'tech_0', 'n_ingredients', 'for-1-or-2']
X shape: (168604, 100)
y shape: (168604,)

=== Model comparison on cleaned dataframe ===
LinearRegression      RMSE= 29.360  R²= 0.333
Ridge                 RMSE= 29.360  R²= 0.333
Lasso                 RMSE= 29.360  R²= 0.333
RandomForest          RMSE= 28.272  R²= 0.381
GradientBoosting      RMSE= 28.389  R²= 0.376
HistGradientBoosting  RMSE= 27.477  R²= 0.415
KNN                   RMSE= 30.097  R²= 0.299
```

Random Forest Model

```
# OG : UNDER 240
X_train, X_test, y_train, y_test = train_test_split(
X_240, y_240  test_size=0.2, random_state=40
)

rf = RandomForestRegressor(
    n_estimators=300,
    max_depth=None,
    min_samples_leaf=1,
    n_jobs=-1,
    random_state=40
)

rf.fit(X_train, y_train)

from sklearn.metrics import r2_score, mean_squared_error
y_pred = rf.predict(X_test)

r2 = r2_score(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)

print(f"Under 240 Test R²:   {r2:.4f}")
print(f"Under 240 Test RMSE: {rmse:.4f}")
```

1m 10.0s

Keep this in mind!

Stated before: decided to test out on smaller filter increments

- Only thing that changes in the code to get the model metrics for the different filters (60, 120, 180, 240 minutes)

Random Forest Model Results

```
Under 240 Test R²:   0.3721
Under 240 Test RMSE: 28.6963
```

```
Under 180 Test R²:   0.3801
Under 180 Test RMSE: 23.6412
```

```
Under 120 Test R²:   0.4606
Under 120 Test RMSE: 17.3393
```

```
Under 60 Test R²:   0.4516
Under 60 Test RMSE: 10.5168
```

Findings?

- With a general Random Forest model, the filter needed to get the best r^2 is recipes under 120 minutes.

Random Forest Optimal Model Parameter

```python
rf = RandomForestRegressor(random_state=40, n_jobs=-1)

param_dist = {
    "n_estimators":    [200, 400, 800],
    "max_depth":       [None, 10, 20, 40],
    "min_samples_split": [2, 5, 10],
    "min_samples_leaf":  [1, 2, 5, 10],
    "max_features":    ["sqrt", 0.3, 0.5, 0.8],
}

kf = KFold(n_splits=3, shuffle=True, random_state=40)

search = RandomizedSearchCV(
    rf,
    param_distributions=param_dist,
    n_iter=20,
    scoring="r2",
    cv=kf,
    verbose=1,
    n_jobs=-1,
    random_state=40
)

search.fit(X_120, y_120)

print("Best Under 120 R² (CV):", search.best_score_)
print("Best Under 120 params:", search.best_params_)

results = pd.DataFrame(search.cv_results_)
cols_to_show = [c for c in results.columns if c.startswith("param_")] + [
    "mean_test_score", "std_test_score", "rank_test_score"
]
print(
    results[cols_to_show]
        .sort_values("rank_test_score")
        .head(10)
)
```

```
Best Under 120 R² (CV): 0.4848436604344011
Best Under 120 params: {'n_estimators': 400, 'min_samples_split': 10, 'min_samples_leaf': 1, 'max_features': 0.3, 'max_depth': 40}
    param_n_estimators  param_min_samples_split  param_min_samples_leaf  \
14                 400                       10                       1
16                 200                       10                       2
12                 400                        5                       2
19                 200                        2                       5
1                  400                       10                       2
2                  400                       10                       2
10                 200                        5                       2
4                  200                       10                       2
18                 400                       10                      10
7                  400                       10                      10

   param_max_features  param_max_depth  mean_test_score  std_test_score  \
14                0.3               40         0.484844        0.001949
16                0.3             None         0.484782        0.002076
12                0.3               20         0.484395        0.001478
19                0.3               20         0.482010        0.001652
1                sqrt             None         0.480565        0.000859
2                sqrt               40         0.480484        0.001282
10               sqrt             None         0.480181        0.000919
4                sqrt             None         0.480098        0.000812
18                0.5               40         0.479950        0.001827
7                 0.5             None         0.479909        0.001826
```

Findings?

- The best model parameters for a **Random Forest Model** that was fed data points/recipes that take less than 120 minutes are:
    - n_estimators: 400
    - min_samples_split: 10
    - min_samples_leaf: 1
    - max_features: 0.3
    - max_depth: 40

Random Forest (CV) Results with Optimal Parameters

CV Results

```
from sklearn.model_selection import KFold, cross_val_score

kf = KFold(n_splits=5, shuffle=True, random_state=40)

rf_cv = RandomForestRegressor(
    n_estimators=400,
    max_depth=40,
    min_samples_leaf=1,
    min_samples_split= 10,
    max_features= 0.3,
    n_jobs=-1,
    random_state=40
)

cv_r2 = cross_val_score(rf_cv, X_120, y_120, cv=kf, scoring="r2", n_jobs=-1)
print("Under 120 CV R² scores:", cv_r2)
print("Under 120 CV R² mean:", cv_r2.mean(), "±", cv_r2.std())
```

✓ 2m 15.3s

```
Under 120 CV R² scores: [0.48472993 0.48717157 0.48741379 0.48422927 0.49183599]
Under 120 CV R² mean: 0.48707611039746707 ± 0.002697726383746162 6
```

Train/Test Results

```
Under 120 Test R²:   0.4849
Under 120 Test RMSE: 16.9447
```

0.4870 ≈ 0.4849 → **NO OVERFITTING!!**

Random Forest Model Results

RMSE: 16.9447

R^2: 0.4849

Features:

- Top 100 features
  - 'tech_0'
    - first index of the features hot-encoded list
  - 'n_steps'
  - 'n_ingredients'
  - 'weeknight'
  - 'for-1-or-2'
  - ...

Random Forest regressor was used due to its ability to handle nonlinear relationships between variables. The top 100 most correlated features were used after exploding the techniques column to let me reduce noise while also focusing on the strongest predictors. I tested multiple hyperparameter to tune the Random Forest (number of trees, depth, leaf size, etc.) on recipes under 120 minutes, and that setting gave me the highest R^2 of all the models I tried.

Final Model

Section III.2

Recipe Time Prediction

Hist Gradient Boosting Model

```
# hist gradient boosting regressor UNDER 240
from sklearn.model_selection import train_test_split
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

X_train, X_test, y_train, y_test = train_test_split(
    X_240, y_240, test_size=0.2, random_state=42
)

hgb = HistGradientBoostingRegressor(random_state=42)
hgb.fit(X_train, y_train)

y_pred = hgb.predict(X_test)

rmse = np.sqrt(mean_squared_error(y_test, y_pred))
r2 = r2_score(y_test, y_pred)

print("===== HistGradientBoostingRegressor =====")
print(f"Under 240 RMSE: {rmse:.4f}")
print(f"Under 240 R²:   {r2:.4f}")

✓ 3.4s
```

way faster than RF!

Same set-up as Random Forest, but we are using **Histogram Gradient Boosting**

Hist Gradient Boosting Model

```
===== HistGradientBoostingRegressor =====
Under 240 RMSE: 27.4773
Under 240 R²:   0.4155
```

```
===== HistGradientBoostingRegressor =====
Under 180 RMSE: 22.9326
Under 180 R²:   0.4214
```

```
===== HistGradientBoostingRegressor =====
Under 120 RMSE: 16.7973
Under 120 R²:   0.4983
```

```
===== HistGradientBoostingRegressor =====
Under 60 RMSE: 10.1789
Under 60 R²:   0.4879
```

Findings?

- With a general Histogram Gradient Boosting model, the filter needed to get the best r^2 is recipes under 120 minutes.

Hist Gradient Boosting Best Model Parameter

```python
hgb = HistGradientBoostingRegressor(random_state=40)

param_dist_hgb = {
    "learning_rate":     [0.01, 0.03, 0.1, 0.2],
    "max_depth":         [None, 5, 10, 20],
    "max_leaf_nodes":    [15, 31, 63, 127],
    "min_samples_leaf":  [10, 20, 50, 100],
    "max_bins":          [50, 100, 255],
    "l2_regularization": [0.0, 0.1, 0.5, 1.0],
}

kf = KFold(n_splits=3, shuffle=True, random_state=40)

search_hgb = RandomizedSearchCV(
    hgb,
    param_distributions=param_dist_hgb,
    n_iter=20,
    scoring="r2",
    cv=kf,
    verbose=1,
    n_jobs=-1,
    random_state=40,
)

search_hgb.fit(X_120, y_120)

print("Best R² (CV):", search_hgb.best_score_)
print("Best params:", search_hgb.best_params_)

results_hgb = pd.DataFrame(search_hgb.cv_results_)
cols_to_show = [c for c in results_hgb.columns if c.startswith("param_")] + [
    "mean_test_score", "std_test_score", "rank_test_score"
]
print(
    results_hgb[cols_to_show]
        .sort_values("rank_test_score")
        .head(10)
)
```

```
Fitting 3 folds for each of 20 candidates, totalling 60 fits
Best R² (CV): 0.49440247968440265
Best params: {'min_samples_leaf': 100, 'max_leaf_nodes': 127, 'max_depth': None, 'max_bins': 100, 'learning_rate': 0.1, 'l2_regularization': 0.1}
    param_min_samples_leaf  param_max_leaf_nodes param_max_depth  \
13                     100                   127            None
9                      100                    63              20
7                       50                    63            None
16                      50                    63              20
10                      20                    63            None
3                       10                    63              20
2                       20                    31              10
12                      20                    63            None
1                      100                    31            None
8                      100                    31              20

    param_max_bins  param_learning_rate  param_l2_regularization  \
13             100                  0.1                      0.1
9               50                  0.1                      0.0
7               50                  0.2                      1.0
16             100                  0.2                      0.0
10              50                  0.2                      0.5
3              255                  0.2                      1.0
2              100                  0.2                      0.5
12              50                  0.2                      0.1
1              255                  0.1                      0.5
```

Findings?

- The best model parameters for a **Histogram Gradient Boosting** model that was fed data points/recipes that take less than 120 minutes are:
  - min_samples_leaf: 100
  - max_leaf_nodes: 127
  - max_depth: None
  - max_bias: 100
  - learning _rate: 0.1
  - L2_regularization: 0.1

Hist Gradient Boosting CV Results

CV Results

```
kf = KFold(n_splits=5, shuffle=True, random_state=40)

hgb_cv = HistGradientBoostingRegressor(
    random_state=42,
    learning_rate= 0.1,
    max_depth= None,
    max_leaf_nodes= 127,
    min_samples_leaf= 100,
    max_bins= 100,
    l2_regularization= 0.1
    )

cv_r2 = cross_val_score(hgb_cv, X_120, y_120, cv=kf, scoring="r2", n_jobs=-1)
print("Under 120 CV R² scores:", cv_r2)
print("Under 120 CV R² mean:", cv_r2.mean(), "±", cv_r2.std())
```

✓ 18.8s

```
Under 120 CV R² scores: [0.49383639 0.49743063 0.49765837 0.49378639 0.49848836]
Under 120 CV R² mean: 0.49624002890443686 ± 0.002014054319571189
```

Train/Test Results

```
===== HistGradientBoostingRegressor =====
Under 120 RMSE: 16.7034
Under 120 R²:   0.5039
```

0.4962 ≈ 0.5039 → **NO OVERFITTING!!**

Hist Gradient Boosting Results

RMSE: 16.7034

R^2: 0.5039

Features:

- Top 100 features
  - 'tech_0'
    - first index of the features hot-encoded list
  - 'n_steps'
  - 'n_ingredients'
  - 'weeknight'
  - 'for-1-or-2'
  - ...

Similar to Random Forest regressor, the top 100 most correlated features were used after exploding the techniques column to let me reduce noise while also focusing on the strongest predictors. I tested multiple hyperparameter to tune the Histogram Gradient Boosting (max bins, max depth, number of leaf nodes, etc.) on recipes under 120 minutes, and that setting gave me the highest R^2 of all the models I tried. However, the reasoning of using this one is due to its ability to model nonlinear relationships **effectively** on tabular data.

Final Model Results

RMSE: 16.7034

R^2: 0.5039

Features:

- Top 100 features
  - 'tech_0'
    - first index of the features hot-encoded list
  - 'n_steps'
  - 'n_ingredients'
  - 'weeknight'
  - 'for-1-or-2'
  - ...

The reason Histogram Gradient Boosting is optimal is due to its processing time. Since HGB has the ability to model nonlinear relationships **effectively** on tabular data, it's processing time outperforms both the baseline Linear Regression and Random Forest Model. HGB was trained in 3 seconds while RF was trained in 1 minute. Although RF and HGB have close to similar R^2, HGB's efficiency outperforms both models.

Evaluation

Section IV

Discussion of Related Work

Section V

Recipe Time Prediction

Discussion

Related Work

How has this dataset (or similar) been used before?	How has prior work approached the same (or similar) tasks?	How do your results match or differ from what has been reported in related work?	

1. Generating Personalized Recipes from Historical User Preferences
2. Predicting Cooking Time Based on Different Recipe Features
	Related Work 2: Attempts to create a model that will predict how long a recipe takes given its ingredients, steps, and nutritional content	Related Work 2:
		• Baseline:
		  ○ RMSE: 87.3 minutes
		In comparison to their baseline, we have a better RMSE at 34.8205, which matches with their final model's performance instead.
			• Final Model:
			  ○ RMSE: 34.35 minutes
			In comparison to their final model, we have a RMSE of 16.7034. Which is about half compared to theirs.

Thank you!

Questions? Reach out at shaing@ucsd.edu