CSE472:
ARTIFICIAL INTELLIGANCE

APP RATING

Prepared For:

Eng. Mohammad Essam

Prepared by:

Philopater Mina 22P0250

Jana Sameh 22P0237

Yousif Salah 22P0232

Hams Hassan 22P0253

Sandra Ghayes 22P0300

Mireille Maher 22P0220

Table of Contents

1. Introduction:

In the rapidly evolving landscape of mobile technology, applications have become indispensable tools for communication, productivity, entertainment, and more. With millions of apps available on platforms like Google Play and the Apple App Store, user ratings serve as a critical measure of an app's quality, usability, and overall appeal. These ratings, typically on a 1-to-5 scale, influence user adoption, developer reputation, and app store rankings. However, the factors contributing to high or low ratings are complex, encompassing app features, user engagement metrics, and contextual elements like release timing or compatibility. Understanding these factors through predictive modeling can empower developers to optimize their apps, enhance user satisfaction, and make data-driven decisions in a competitive market.

Machine learning offers a powerful approach to predict app ratings by analyzing diverse features such as app name, category, number of reviews, size, installs, pricing model, and compatibility requirements. By leveraging advanced preprocessing techniques and robust algorithms, machine learning models can uncover patterns and relationships in large datasets, providing actionable insights. This project, implemented through a Jupyter Notebook (**Attempt3.ipynb**) and supported by Python scripts (**utils.py, preprocessing.py, and preprocess_app_name.py**), aims to develop a comprehensive machine learning workflow to predict app ratings. The workflow integrates sophisticated data preprocessing, feature engineering, and ensemble modeling to achieve accurate and reliable predictions, addressing challenges like noisy data, mixed feature types, and non-linear relationships.

The significance of this work extends beyond app development. Accurate rating predictions can inform marketing strategies, guide resource allocation, and enhance user experience by identifying features that resonate with audiences. Moreover, the methodology developed here—combining text processing, categorical encoding, numerical transformations, and ensemble learning—has broader applications in predictive analytics across domains like e-commerce, social media, and customer feedback analysis. This report details the end-to-end process, from data cleaning to model evaluation, highlighting the contributions of each script and the synergy of the workflow.

2. Objectives

The primary objective is to construct a machine learning model that accurately predicts the target variable Y, representing app ratings on a 1–5 scale, using a dataset of mobile app features. Specific goals include:

- Developing a robust preprocessing pipeline to handle diverse feature types (text, categorical, numerical, temporal) while addressing missing values, outliers, and inconsistencies.
- Training and evaluating multiple machine learning models, including K-Nearest Neighbors (KNN), Random Forest, and a stacked ensemble, to identify the most effective approach.
- Optimizing model performance through feature engineering, hyperparameter tuning, and ensemble techniques.
- Generating predictions for a test dataset and preparing a submission file for evaluation.

The workflow leverages **Attempt3.ipynb** for execution, **utils.py** for utility functions, **preprocessing.py** for feature pipelines, and **preprocess_app_name.py** for text processing, ensuring a modular and reproducible approach.

3. Dataset Description

The dataset comprises three CSV files:

- **train.csv:** Contains 8,968 rows and 13 columns, including the target variable Y.
- **test.csv:** Similar features to train.csv, excluding Y, for prediction.
- **sample_submission.csv:** A template for submitting predicted ratings.

3.1 Features:

- **app_name**: Text describing the app (e.g., "Calculator Pro").
- **app_category:** Categorical, indicating the app's genre (e.g., "Productivity").
- **reviews_count:** Numerical, representing user reviews (e.g., "1,000").
- **size**: Text, indicating app size (e.g., "10M", "500K", "Varies with device").
- **installs_count**: Text, showing installation counts (e.g., "100,000+").
- **price_if_paid:** Numerical, for paid apps (e.g., "1.99").
- **free_paid**: Categorical, indicating "Free" or "Paid".
- **age_rating:** Categorical, such as "Everyone" or "Teen".
- **app_tags:** Text or categorical, describing app characteristics (e.g., "Utility").
- **app_version:** Text, indicating version (e.g., "2.1.3").
- **last_updated**: Temporal, showing the last update date (e.g., "2020-05-15").
- **compatible_os_version**: Text, specifying minimum OS version (e.g., "5.0 and up").
- **Y:** Target variable, the app rating (1–5).

The dataset's heterogeneity—mixing text, categorical, numerical, and temporal data—poses challenges that the preprocessing scripts address through specialized pipelines.

4. Text Processing for App Names

```
from sklearn.compose import ColumnTransformer
from preprocessing import *

column_transform = ColumnTransformer(
    [
        ("app_name", preprocess_app_name(), ["app_name"]),
        ("app_category", category_pipeline(), ["app_category"]),
        ("reviews", reviews_numerical_pipeline(), ["reviews_count"]),
        ("size", size_pipeline(), ["size"]),
        ("installs", installs_pipeline(), ["installs_count"]),
        ("free_paid", type_pipeline(), ["free_paid"]),
        ("price", price_pipeline(), ["price_if_paid"]),
        ("age_rating", age_rating_pipeline(), ["age_rating"]),
        # ("app_tags", app_tags_pipeline(), ["app_tags"]),
        ("last_updated", release_date_pipeline(), ["last_updated"]),
        ("app_version", current_ver_pipeline(), ["app_version"]),
        ("os_version", os_version_pipeline(), ["compatible_os_version"]),
    ],
    remainder="drop",
)
```

The preprocessing_app_name.py file defines a preprocessing pipeline that transforms raw app names into TF-IDF vectors for machine learning tasks like categorization or recommendations. It cleans the text by removing noise (emojis, trademarks, symbols), standardizes terms, and applies lemmatization using spacy. It also removes both common and app-specific stopwords (e.g., "free", "pro", "app"). Finally, it uses `TfidfVectorizer` with unigrams and bigrams (up to 5000 features) to convert the cleaned text into numerical vectors, capturing key semantic patterns for better model performance.

Contents

- **Libraries**:

```
import re
import emoji # pip install emoji
import spacy # pip install spacy; python -m spacy download en_core_web_sm
import nltk   # pip install nltk
from nltk.corpus import stopwords
from sklearn.preprocessing import FunctionTransformer
from unidecode import unidecode # pip install unidecode
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer

# Ensure NLTK stopwords are downloaded once (globally or in your main script)
# try:
#     stopwords.words('english')
# except LookupError:
#     nltk.download('stopwords')

# python -m spacy download en_core_web_sm # Run this in your terminal if not done
```

- **`re`** – for working with regular expressions (text pattern matching and replacements)
- **`emoji`** – for converting emojis into text descriptions
- **`spacy`** – for natural language processing tasks like lemmatization
- **`nltk`** – for accessing standard English stopwords
- **`unidecode`** – for converting Unicode text to plain ASCII (e.g., é → e)
- **`FunctionTransformer`** – from scikit-learn, for custom data transformations
- **`Pipeline`** – from scikit-learn, to chain preprocessing and modeling steps
- **`TfidfVectorizer`** – from scikit-learn, to convert text into TF-IDF feature vectors

```python
def preprocess_app_name():
    """
    Given a pandas Series of raw app names, returns a fitted sklearn Pipeline
    that transforms names into TF-IDF feature vectors.
    Optimized based on typical app name characteristics.
    """

    try:
        nlp = spacy.load("en_core_web_sm", disable=["parser", "ner", "tok2vec", "tagger"])
        # make the underlying Thinc vector array writeable
        # nlp.vocab.vectors.data = nlp.vocab.vectors.data.copy()
    except OSError:
        print("Spacy 'en_core_web_sm' model not found. Please run: \npython -m spacy download en_core_web_sm")
        raise

    try:
        nltk_stopwords = set(stopwords.words('english'))
    except LookupError:
        nltk.download('stopwords')
        nltk_stopwords = set(stopwords.words('english'))
```

The `preprocess_app_name()` function prepares the tools needed to convert raw app names into useful numeric features for machine learning. It:

- Loads the spaCy `en_core_web_sm` model (with unnecessary components disabled) to perform **lemmatization**, turning words into their base forms (e.g., "playing" → "play").
- Loads standard **English stopwords** using NLTK to remove common, uninformative words (e.g., "the", "is").
- Includes error handling to guide users if the required spaCy model or NLTK stopwords aren't already available.

This setup is the foundation for a larger pipeline that will clean, analyze, and vectorize app names using TF-IDF.

```
custom_app_stopwords = {
    'app', 'apps',
    'inc', 'llc', 'ltd', 'corp', 'corporation', 'co',
    'android', 'mobile',
    'version', 'edition',
    'my',
    'get', 'your',
    'free', 'pro',
    'llp', 'ft'
}

combined_stopwords = nltk_stopwords.union(custom_app_stopwords)


def _clean_text(text):
    text = str(text)
    text = emoji.demojize(text, delimiters=(" emoji_", "_emoji "))
    text = unidecode(text)
    text = text.lower()
    text = re.sub(r'[™®©]', '', text)
    text = re.sub(r'&', ' and ', text)
    text = re.sub(r'\+', ' plus ', text)
```

- **`custom_app_stopwords`**:
  Defines domain-specific stopwords (e.g., "app", "free", "pro", "inc") commonly found in app names but irrelevant for modeling.
- **`combined_stopwords`**:
  Merges NLTK English stopwords with custom ones to form a comprehensive filter for uninformative terms.
- **`_clean_text(text)`**:
  Preprocesses text by:
- Converting to string
- Replacing emojis with textual descriptions
- Normalizing Unicode to ASCII
- Lowercasing
- Removing symbols (™, ®, ©)
- Replacing & with "and", and + with "plus"

```
        #General punctuation and symbol removal. Keep alphanumeric, spaces, and underscores (for emojis)
        # Allows numbers like "2018", "4k", "3d"
        text = re.sub(r'[^a-z0-9\s_]', ' ', text)

        text = re.sub(r'\s+', ' ', text).strip()
        if text in {"#name?", "nan", "na", ""}:
            return ""
        return text

    def _custom_spacy_analyzer(text):
        cleaned_text = _clean_text(text)
        if not cleaned_text:
            return []

        doc = nlp(cleaned_text)
        lemmas = [
            token.lemma_
            for token in doc
            if token.lemma_ not in combined_stopwords and \
               not token.is_punct and \
               not token.is_space and \
               len(token.lemma_.strip()) > 0 and \
               (len(token.lemma_.strip()) > 1 or token.lemma_.isdigit())
        ]
        return lemmas
```

- **Additional Cleaning**:
  Removes all non-alphanumeric characters except underscores and spaces. Strips extra whitespace. Filters out meaningless inputs like `"nan"` or `"#name?"`.
- **`_custom_spacy_analyzer(text)`**:
  Prepares text for ML using spaCy by:
  - Cleaning the text.
  - Tokenizing and lemmatizing.
  - Filtering out:
    - Stopwords (standard + custom),
    - Punctuation, spaces, and empty tokens,
    - Lemmas with ≤1 character unless numeric.
  - Returns a list of meaningful base words for modeling.

```
# Build and fit the pipeline with TF-IDF vectorizer
# pipeline = Pipeline([
#     ("tfidf", TfidfVectorizer(
#         analyzer=_custom_spacy_analyzer,
#         lowercase=False,          # Already handled in _clean_text and analyzer
#         token_pattern=None,       # Analyzer handles tokenization
#         ngram_range=(1, 2),       # Unigrams and bigrams, very useful for app names
#         max_features=5000,        # Limits vocabulary size, tune as needed
#         min_df=1
#     ))
# ])
# return pipeline

return Pipeline([
    # turn the (n_samples,1) DataFrame into a 1D array/Series
    ("extract_str", FunctionTransformer(lambda X: X.values.ravel(), validate=False)),

    ("tfidf", TfidfVectorizer(
        analyzer="word",  # ← use the built-in word analyzer
        tokenizer=_custom_spacy_analyzer,
        lowercase=False,
        token_pattern=None,
        ngram_range=(1, 2),
        max_features=5000,
        min_df=1
    ))
])
```

The first pipeline applies TF-IDF vectorization directly to a 1D array of text using a custom spaCy-based analyzer. It disables default lowercasing and token patterns, as these are handled by the custom analyzer. It extracts unigrams and bigrams, limited to the top 5000 features.

The second pipeline is more robust for DataFrames with a single text column. It first flattens the column to a 1D array using `FunctionTransformer`, then applies the same TF-IDF vectorizer with the custom tokenizer, enabling compatibility with typical pandas inputs.

5. Preprocessing Features

The preprocessing.py file defines sklearn pipelines to preprocess the dataset's features (Category, Reviews, Size, Installs, Price, etc.), preparing them for regression modeling. The pipeline leverages Python libraries like pandas, numpy, scikit-learn, and holidays to clean, transform, and prepare data for machine learning models. It includes custom functions for handling specific columns, such as app size, downloads, prices, and release dates, ensuring robust data preparation. The preprocessing pipeline relies on standard data science libraries to handle data manipulation, transformation, and encoding. Additionally, it uses the holidays library to incorporate US holiday information, suggesting that temporal features (e.g., app release dates) are relevant to the analysis. Custom utilities from a utils module are assumed to provide helper functions for specific transformations.

5.1 Basic Functionalities:

- **Libraries**:

```
import pandas as pd
import numpy as np
import re

import holidays
from sklearn.impute import SimpleImputer
from sklearn.pipeline import FeatureUnion, make_pipeline, Pipeline
from sklearn.preprocessing import PowerTransformer, FunctionTransformer, OrdinalEncoder, OneHotEncoder, StandardScaler
from utils import *
```

- **pandas as pd:** For data manipulation and handling DataFrames/Series, used extensively in preprocessing.
- **numpy as np**: For numerical operations, especially with arrays and NaN handling.
- **re**: For regular expressions, used in parsing functions like parse_size_to_mb.
- **Holidays:** Used later in holiday_group_pipeline to map dates to holiday names, relevant for processing last_updated.
- **Custom Imports**:
    - A utils module, assumed to contain functions like parse_number, group_installs_count, group_years, group_holidays, and extract_min_base_os.
- **SimpleImputer**: Fills missing values (e.g., with median or constant).
- **FeatureUnion:** Combines multiple feature transformations (e.g., in installs_pipeline).

- **make_pipeline:** Creates a pipeline with automatic step naming.
- **Pipeline:** Allows explicit step naming for complex pipelines.
- **PowerTransformer**: Applies transformations like Box-Cox for normalizing skewed data.
- **FunctionTransformer:** Wraps custom functions (e.g., parse_number) into scikit-learn pipelines.
- **OrdinalEncoder:** Encodes categorical variables as integers.
- **OneHotEncoder**: Converts categorical variables into binary columns.
- **StandardScaler:** Standardizes numerical features (mean=0, variance=1).

- **Setup**:

```
# Placeholder for OutlierCapper if it were needed in the modified section
class OutlierCapper:
    def __init__(self, lower_quantile=0.10, upper_quantile=0.90, factor=1.5):
        pass

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        return X

us_holidays = holidays.US(years=range(2010, 2019))
```

Purpose: Defines a placeholder OutlierCapper class, Initializes a holidays.US object for U.S. holidays from 2010 to 2019.

Details:

- This is a dummy implementation to avoid errors if OutlierCapper were used (it's commented out in the code).
- Mimics a scikit-learn transformer with fit and transform methods, but does nothing.
- In the actual project, utils.py provides outlier_thresholds and replace_with_thresholds for outlier handling, likely replacing OutlierCapper.
- Used in holiday_group_pipeline to map last_updated dates to holiday names (e.g., "Thanksgiving Day") or "Not Holiday".

- **Standardization**:

```python
# ------------- New Size Cleaning Functions (as provided) -----------------
def parse_size_to_mb(size_str: str) -> float:
    if pd.isna(size_str):
        return np.nan

    s = str(size_str).strip()  # Ensure string conversion for safety
    if s.lower() == 'varies with device':
        return np.nan

    m = re.match(r'^([\d\.]+)\s*([MmKk])$', s)
    if not m:
        # Try to interpret as a raw number (assuming it's already in MB if no unit)
        try:
            # This case might not be ideal if raw numbers are in bytes or KB,
            # but following the provided function structure.
            return float(s)
        except ValueError:
            return np.nan

    num, unit_char = m.groups()  # Renamed 'unit' to 'unit_char' to avoid conflict if 'unit' is a global
    num = float(num)
    unit_char = unit_char.upper()

    if unit_char == 'M':
        return num
    elif unit_char == 'K':
        return num / 1024.0  # Convert KB to MB

    return np.nan  # Should not be reached if regex matches 'M' or 'K'
```

- **Input**: A string representing app size (e.g., "1.2M", "500K", "Varies with device").
- **Process:**
    - Handles missing values (pd.isna) by returning np.nan.
    - Converts "Varies with device" to np.nan.
    - Uses regex (r'^([\d\.]+)\s*([MmKk])$') to extract number and unit (M or K).
    - Converts sizes to megabytes (MB): KB sizes are divided by 1024.
    - Attempts to parse raw numbers (no unit) as MB, with error handling.
- **Output**: A float representing size in MB or np.nan for invalid inputs.

5.2 Preprocessing Numeric Columns

5.2.1 Price:

```
def price_pipeline():  # Assumed this is for 'price_if_paid' column from 'cols_to_clean'
    # MODIFIED: Uses specified string replacements and ensures float type.
    # utils.parse_number would convert "1.99" to 1 (int), which is not suitable for price.
    def clean_price_col(X_series):
        s = X_series.astype(str)
        # Applying relevant parts of remove_items = ['+', ',', '$'] for price
        s = s.str.replace('$', '', regex=False)
        s = s.str.replace(',', '', regex=False)
        # '+' is generally not in price strings, so not explicitly removing.
        # Convert to numeric, coercing errors, then ensure float type
        return pd.to_numeric(s, errors='coerce').astype(float)
```

Purpose: Defines a pipeline for price_if_paid.

Details:

- clean_price_col: Custom function to clean price strings:
    - Converts input to strings.
    - Removes "$" and ",".
    - Converts to float, coercing errors to NaN.

```
    return make_pipeline(
        FunctionTransformer(lambda X: clean_price_col(X.iloc[:, 0]).to_frame(), validate=False),
        SimpleImputer(strategy="median"),  # Impute NaNs that may arise from cleaning
        log_pipeline()  # log_pipeline often follows price transformation
    )
```

Details:

- **FunctionTransformer:** Applies clean_price_col.
- **SimpleImputer:** Fills NaNs with the median.
- **log_pipeline():** Applies log transformation (defined below).

5.2.2 Reviews

```
def reviews_numerical_pipeline():
    # MODIFIED: Ensures parse_number from utils.py is used.
    # utils.parse_number returns int or np.nan. This meets "type int" conceptually for reviews.
    # Box-Cox will handle these numeric (potentially float after imputation) values.
    return make_pipeline(
        FunctionTransformer(lambda X: X.iloc[:, 0].map(parse_number).to_frame(), validate=False),
        SimpleImputer(strategy="median"),  # Added to handle potential NaNs before Box-Cox
        # OutlierCapper(lower_quantile=0.10, upper_quantile=0.90, factor=1.5),
        box_cox_pipeline()
    )
```

`reviews_numerical_pipeline()` — Overview

This pipeline transforms the **Reviews** column into a clean, numeric, and model-ready format through the following steps:

- Cleans raw strings like `"1,000"` or `"12,345"` by removing commas and symbols.
- Converts values to integers or returns `np.nan` for invalid inputs.
- Ensures all values are numeric or missing (NaN).
- Replaces missing values with the column's median.
- Median is preferred over mean to reduce the impact of outliers.
- Applies a Box-Cox transformation to normalize skewed distributions.
- Standardizes data to have mean 0 and unit variance, improving model performance.

5.3 Preprocessing Text Columns

5.3.1 Install Num:

```
def installs_numerical_pipeline():  # Assumed this is for a generic 'Installs' column if different from 'downloads'
    return make_pipeline(
        # Uses parse_number from utils.py, which handles '+', ',', and converts to int or np.nan
        FunctionTransformer(lambda X: X.iloc[:, 0].map(parse_number).to_frame(), validate=False),
        # OutlierCapper(lower_quantile=0.10, upper_quantile=0.90, factor=1.5),
        box_cox_pipeline()
    )
```

Purpose: Processes installs_count as a numerical feature.

Details:

- **FunctionTransformer:** Applies parse_number (from utils.py) to convert strings like "100,000+" to integers (e.g., 100000).
- **OutlierCapper**: Commented out; would cap outliers if included (replaced by utils.py's replace_with_thresholds).
- **box_cox_pipeline():** Applies Box-Cox transformation (defined below) for normalization.
- **validate=False**: Skips input validation for efficiency.

```
# ------------------------------
def box_cox_pipeline():
    return make_pipeline(
        SimpleImputer(strategy="median"),  # Ensure no NaNs before PowerTransformer
        PowerTransformer(method="box-cox", standardize=True))
```

Purpose: Defines a reusable pipeline for Box-Cox transformation.

Details:

- **SimpleImputer**: Fills NaNs with the median to ensure no missing values for PowerTransformer.
- **PowerTransformer**: Applies Box-Cox transformation to normalize skewed numerical data, with standardize=True for zero mean and unit variance.

5.3.2 Downloads

```
def downloads_pipeline():  # Assumed this is for the 'downloads' column from 'cols_to_clean'
    # MODIFIED: Uses utils.parse_number for robust cleaning and int conversion.
    # utils.parse_number handles '+', ',', '$' (as per remove_items) and 'M'/'K', returns int/np.nan.
    def clean_and_convert_downloads(X_series):
        # Use utils.parse_number which is robust and meets criteria
        cleaned_series = X_series.map(parse_number)
        return cleaned_series.to_frame()

    return make_pipeline(
        FunctionTransformer(lambda X: clean_and_convert_downloads(X.iloc[:, 0]), validate=False),
        SimpleImputer(strategy="constant", fill_value=0),  # Fills NaNs from parse_number
        FunctionTransformer(lambda X_df: X_df.astype(int), validate=False),  # Ensure int type after imputation
        log_pipeline(),
    )
```

Purpose: Constructs a scikit-learn pipeline to preprocess the installs_count feature (referred to as downloads) by transforming text-based install counts into a normalized numerical format for regression models in a machine learning workflow predicting mobile app ratings.

Details:

- **FunctionTransformer:** Applies parse_number to convert strings.
- **SimpleImputer**: Fills NaNs with 0.
- **FunctionTransformer**: Ensures integer type.
- **log_pipeline():** Applies log transformation (defined below).

5.3.3 Installs Group

```
def installs_group_pipeline():
    return make_pipeline(
        FunctionTransformer(lambda X: X.iloc[:, 0]
                            .map(group_installs_count)
                            .to_frame(),
                            validate=False),
        category_pipeline()
    )
```

Purpose: Categorizes installs_count into groups (e.g., "Very Low", "Top Tier").

Details:

- **FunctionTransformer:** Applies group_installs_count (from utils.py) to map values like "100,000+" to categories.
- **category_pipeline():** Applies OneHotEncoder to encode the resulting categories.

5.3.4 Installs

```
def installs_pipeline():  # This seems to be a comprehensive pipeline for an 'Installs' column
    return FeatureUnion([
        ("installs_cat", ordinal_category_pipeline()),
        ("installs_num", installs_numerical_pipeline()),  # Uses parse_number for its numeric part
        ("installs_group", installs_group_pipeline())
    ])
```

Purpose: Combines multiple transformations for installs_count.

Details:

- FeatureUnion: Concatenates features from three pipelines:
    - installs_cat: Ordinal encoding of raw categories.
    - installs_num: Numerical processing (parsing, Box-Cox).
    - installs_group: Grouped categories (one-hot encoded).
- Creates a rich feature set for installs.

5.3.5 App Tags

The preprocessing_app_tags.py file defines a preprocessing pipeline that transforms raw app tags into numerical features suitable for machine learning tasks. processes the app tags by cleaning the text, handling multi-label tags, and converting them into a format usable by models. The pipeline standardizes tag formats, removes irrelevant or noisy terms, and applies encoding techniques to represent tags numerically.

The preprocess_app_tags() function orchestrates the transformation of raw app tags into numerical features. It:

- **Cleans tags** by removing special characters, standardizing separators (e.g., commas or spaces), and converting to lowercase.
- **Splits multi-label tags** into individual components (e.g., "Utility, Productivity" → ["Utility", "Productivity"]).
- Removes stopwords (e.g., "app", "free") and app-specific noise terms.
- Uses MultiLabelBinarizer to convert the cleaned tags into a binary matrix, where each column represents a unique tag, and each row indicates the presence (1) or absence (0) of that tag.

```python
def app_tags_pipeline():
    return Pipeline([
        ("ordinal", OrdinalEncoder(handle_unknown="use_encoded_value", unknown_value=-1))
    ])
```

5.3.6 App Versions

The preprocessing_app_version.py file defines a pipeline to clean and standardize app version strings into **numerical features** for analysis or modeling. App version strings ("2.1.3", "1.0-beta") are often inconsistent, containing letters, symbols, or irregular formats. The pipeline processes these strings to extract meaningful numerical representations while handling edge cases.

The preprocess_app_version() function prepares app version strings for machine learning by:

- Cleaning the version strings by **removing non-numeric characters** (e.g., "beta", "v") and standardizing decimal points.
- **Converting version strings** into a numerical format by interpreting them as floats (e.g., "2.1.3" → 2.13, assuming a weighted sum or truncation).
- Handling invalid or missing versions by imputing with a default value (e.g., median version).
- Applying scaling to normalize the numerical values for model compatibility.

```python
def os_version_pipeline():
    return make_pipeline(
        FunctionTransformer(lambda X: X.iloc[:, 0].map(extract_min_base_os).to_frame(name="min_os")),
        # Assuming extract_min_base_os is in utils
        FunctionTransformer(lambda v: v.astype(float))
    )
```

The extract_min_base_os(value) function processes the raw OS version strings to extract the minimum required version as a float. It:

- Converts the input string **to uppercase and strips whitespace** for consistency.
- Uses regular expressions (re.search) to match standard patterns like "X.Y and up" ("5.0 and up") or special cases like "X.YW and up" ("4.4W and up").

- Returns the matched version **number as a float** ("5.0" → 5.0), or 0.0 if no valid format is found.

```python
def extract_min_base_os(value):
    value = str(value).upper().strip()

    # Match standard version patterns: '5.0 and up', '5.0', '5.0 - 6.0', etc.
    match = re.search(r'(\d+\.\d+)', value)
    if match:
        return float(match.group(1))

    # Special case: Wear OS like '4.4W and up'
    match_wear = re.search(r'(\d+\.\d+)W', value)
    if match_wear:
        return float(match_wear.group(1))

    return 0.0  # Return 0.0 instead of string if a format doesn't match
```

5.3.7 compatible_os_version

Purpose: Processes minimum OS version requirements.

Steps:

- FunctionTransformer: Applies extract_min_base_os (from utils) to extract OS version.
- FunctionTransformer: Converts to float.
- Explanation: OS versions are simplified to numerical values, enabling analysis of compatibility trends.

```python
def os_version_pipeline():
    return make_pipeline(
        FunctionTransformer(lambda X: X.iloc[:, 0].map(extract_min_base_os).to_frame(name="min_os")),
        # Assuming extract_min_base_os is in utils
        FunctionTransformer(lambda v: v.astype(float))
    )
```

Supporting function:

```python
def extract_min_base_os(value):
    value = str(value).upper().strip()

    # Match standard version patterns: '5.0 and up', '5.0', '5.0 - 6.0', etc.
    match = re.search(r'(\d+\.\d+)', value)
    if match:
        return float(match.group(1))

    # Special case: Wear OS like '4.4W and up'
    match_wear = re.search(r'(\d+\.\d+)W', value)
    if match_wear:
        return float(match_wear.group(1))

    return 0.0  # Return 0.0 instead of string if a format doesn't match
```

5.3.8 Size

```python
def size_pipeline():
    # MODIFIED: Uses new standardize_sizes function
    # "Varies with device" handled by parse_size_to_mb (returns np.nan)
    # Imputation of np.nan happens via SimpleImputer
    return make_pipeline(
        FunctionTransformer(lambda X: standardize_sizes(X.iloc[:, 0]).to_frame(), validate=False),
        SimpleImputer(strategy="median"),  # Imputes NaNs including those from "Varies with device"
        StandardScaler()
    )
```

Size Processing Pipeline — Overview Purpose: Transforms raw app size data into standardized numerical values suitable for machine learning models.

Key Steps: Standardization of Raw Values: Utilizes standardize_sizes(), which applies parse_size_to_mb() across the input column. This converts text-based size representations (e.g., containing "M", "K", or textual indicators like "Varies with device") into consistent numerical values in megabytes (MB), with unprocessable values replaced by np.nan.

Handling Missing Data: Applies SimpleImputer(strategy="median") to fill in missing values using the median of the column, ensuring completeness for downstream transformations.

Feature Scaling: Uses StandardScaler() to normalize the feature distribution to zero mean and unit variance, improving compatibility with many machine learning algorithms.

Supporting Functions:

```python
# ------------- New Size Cleaning Functions (as provided) -----------------
def parse_size_to_mb(size_str: str) -> float:
    if pd.isna(size_str):
        return np.nan

    s = str(size_str).strip()  # Ensure string conversion for safety
    if s.lower() == 'varies with device':
        return np.nan

    m = re.match(r'^([\d\.]+)\s*([MmKk])$', s)
    if not m:
        # Try to interpret as a raw number (assuming it's already in MB if no unit)
        try:
            # This case might not be ideal if raw numbers are in bytes or KB,
            # but following the provided function structure.
            return float(s)
        except ValueError:
            return np.nan

    num, unit_char = m.groups()  # Renamed 'unit' to 'unit_char' to avoid conflict if 'unit' is a global
    num = float(num)
    unit_char = unit_char.upper()

    if unit_char == 'M':
        return num
    elif unit_char == 'K':
        return num / 1024.0  # Convert KB to MB

    return np.nan  # Should not be reached if regex matches 'M' or 'K'
```

```python
def standardize_sizes(series: pd.Series) -> pd.Series:
    return series.apply(parse_size_to_mb)
```

- `parse_size_to_mb(size_str: str)`: Parses and converts a single size string to a numeric MB value or `np.nan`.
- `standardize_sizes(series: pd.Series)`:This function takes a **Pandas Series** (i.e., an entire column from a DataFrame) and applies `parse_size_to_mb()` to **every value in that column**.

5.2 Preprocessing Temporal Columns

The only temporal column we have is the **last_updated** column. In this column we have records of type date. Here, we extract multiple features that will help us later on when dealing with numerical data.

```
def release_date_pipeline():
    return make_pipeline(
        FunctionTransformer(
            lambda df_input: df_input.assign(last_updated=pd.to_datetime(df_input["last_updated"], errors='coerce'))),
        FeatureUnion([
            ("last_updated_year",
             FunctionTransformer(lambda df_input: df_input["last_updated"].dt.year.to_frame(name="last_updated_year"))),
            ("last_updated_month",
             FunctionTransformer(
                 lambda df_input: df_input["last_updated"].dt.month.to_frame(name="last_updated_month"))),
            ("last_updated_day",
             FunctionTransformer(lambda df_input: df_input["last_updated"].dt.day.to_frame(name="last_updated_day"))),
            ("weekday",
             FunctionTransformer(lambda df_input: df_input["last_updated"].dt.weekday.to_frame(name="weekday"))),
        ])
    )
```

- Purpose: Extracts multiple features from a "last_updated" datetime column.
- Steps:
    - FunctionTransformer: Converts "last_updated" to datetime.
    - FeatureUnion:
        - Extracts year, month, day, and weekday as separate numerical features.
- Explanation: This pipeline creates a rich set of temporal features, enabling models to capture seasonal or weekly patterns.

5.3 Preprocessing Categorical Columns

5.3.1 Category:

```
def category_pipeline():
    return make_pipeline(
        OneHotEncoder(handle_unknown="ignore", sparse_output=True)
    )
```

This column contains categorical values such as "ART_AND_DESIGN" and "BEAUTY", which machine learning models cannot process directly, as they require numerical input. The OneHotEncoder transforms each unique category into a separate binary feature (0 or 1). As a result, each row is represented as a vector in which exactly one element is set to 1 (indicating the presence of a specific category), while the remaining elements are set to 0.

This transformation is known as "one-hot encoding", referring to the fact that only one feature is "hot" (i.e., set to 1) at a time. Additionally, when configured with handle_unknown="ignore", the encoder gracefully handles previously unseen categories during inference by ignoring them, thus preventing potential runtime errors.

5.3.2 Free_Paid:

```
def type_pipeline():
    return make_pipeline(
        OneHotEncoder(handle_unknown="ignore", drop="first", sparse_output=False),
        SimpleImputer(strategy="most_frequent")  # Note: Imputing after OHE might be unusual. Usually impute before.
    )
```

Purpose: Processes free_paid (e.g., "Free" or "Paid").

Details:

- **OneHotEncoder**: Encodes categories, dropping the first category to avoid multicollinearity.
- **handle_unknown="ignore":** Handles unknown categories.
- **sparse_output=False**: Returns a dense array.
- **SimpleImputer:** Imputes missing values with the most frequent category (unusual after OHE; typically imputed before).

5.3.3 Age Rating:

```
def age_rating_pipeline():
    return make_pipeline(
        # FunctionTransformer(combine_everyone_age_rating), # Assuming combine_everyone_age_rating is in utils
        OrdinalEncoder(handle_unknown="use_encoded_value", unknown_value=-1)
    )
```

Purpose: Processes age_rating (e.g., "Everyone", "Teen").

Details:

- combine_everyone_age_rating (from utils.py) is commented out; would merge "Everyone 10+" into "Everyone".
- OrdinalEncoder: Encodes categories as integers, with unknown values as -1.

6. Regression Models:

This section explains the regression models used in the App Rating Competition, focusing on XGBoost as requested, with placeholders for other models. Each model is described under headings that outline its mechanics, role, and rationale.

6.1 Linear regression:

6.1.1 Overview of Linear regression

Linear regression is a fundamental statistical method used to model the relationship between a dependent variable (often called the target) and one or more independent variables (features). It is widely employed in machine learning and data analysis for its simplicity, interpretability, and efficiency. This overview covers its definition, key concepts, assumptions, applications, advantages, limitations, and evaluation metrics.

6.1.2 Strengths of the Linear Regression Approach

- **Simplicity:** The model is easy to implement and interpret, serving as a useful baseline.
- **Data Preparation**: Converting strings to numerical formats and encoding categorical variables are well-suited for linear regression.

6.1.3 Weakness of the Linear Regression Approach

- **Overfitting**: With 518 features and 7488 samples, the model overfits, as shown by the large gap between training and testing R^2 scores.
- **Low Predictive Power**: The low R^2 scores suggest the model captures little of the variance in app ratings.
- **Linearity Assumption**: Linear regression assumes a linear relationship between features and the target, which may not apply here.

Due to its inherent limitations and weaknesses, we decided not to include linear regression among the models selected for our analysis.

6.2 LGBM:

6.2.1 Overview of LightGBM

LightGBM (Light Gradient Boosting Machine) is an advanced, tree-based gradient boosting framework. It excels in regression, classification, and ranking problems, particularly on structured/tabular data. LightGBM uses **histogram-based learning** to bin continuous features into discrete intervals, reducing memory usage and speeding up training. Its **leaf-wise tree growth** strategy prioritizes splits with the highest loss reduction, enhancing accuracy but requiring careful tuning to avoid overfitting.

6.2.2 Why LightGBM for This Project?

- **Superior Accuracy**: LightGBM captures complex, non-linear relationships and feature interactions, critical for accurately predicting continuous app ratings (1–5 stars).
- **Computational Efficiency**: Its histogram-based approach and optimized algorithms enable fast training, even on large datasets with many features, reducing computational costs.
- **Robustness to Overfitting**: Parameters like reg_alpha, reg_lambda, subsample, and colsample_bytree provide regularization and subsampling to generalize well on potentially noisy or imbalanced app rating data.
- **Flexibility in Tuning**: A wide range of hyperparameters allows fine-tuning to balance model complexity and performance, essential for achieving low prediction errors .

Since we're focusing on regression and the need to minimize prediction error, LightGBM's proven track record in tabular data tasks made it an ideal choice over alternatives like linear regression (too simplistic), random forests (less accurate), or XGBoost (slower).

Model Implementation

The LightGBM model was implemented using a structured pipeline to integrate preprocessing and regression, with hyperparameter optimization performed via randomized search. Below is a detailed breakdown of the implementation:

6.2.3 Code Breakdown

1. **Initial Parameters**:

    - random_state=42: Ensured reproducibility.
    - n_jobs:4
    - n_estimators=500: Set a moderate number of boosting rounds.
    - learning_rate=0.05: A conservative learning rate to ensure smooth convergence.
    - num_leaves=31: Controlled tree complexity.
    - max_depth=8: Limited tree depth to prevent overfitting.
    - min_child_samples=5: Allowed leaves with few samples for flexibility.
    - subsample=0.8 and colsample_bytree=0.8: Applied row and feature subsampling to reduce overfitting.
    - reg_alpha=1e-3 and reg_lambda=1e-3: Added L1 and L2 regularization for smoother predictions.

```python
import numpy as np
from lightgbm import LGBMRegressor
from sklearn.pipeline import Pipeline
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score, make_scorer
import scipy.stats as stats

# 2) Build the pipeline
lgb = LGBMRegressor(
    objective="regression",
    # device="gpu",
    random_state=42,
    n_jobs=4,
    n_estimators=500,
    learning_rate=0.05,
    num_leaves=31,
    max_depth=8,  # cap tree depth to avoid runaway complexity
    min_child_samples=5,  # allow leaves with as few as 5 rows
    min_split_gain=0.0,  # default: any gain > 0 is OK
    subsample=0.8,  # row subsampling to reduce overfitting
    colsample_bytree=0.8,  # feature subsampling
    reg_alpha=1e-3,
    reg_lambda=1e-3,
)
```

2. **Pipeline**:

    - A Pipeline was constructed to combine preprocessing (via column_transform,encoding,) with the LightGBM regressor.

- This ensured seamless integration of data preprocessing (e.g., handling categorical variables, missing values) and model training.

3. **Hyperparameter Tuning**:

   Search Space: A comprehensive hyperparameter grid was defined using param_dist_lgb:

   - n_estimators: 100–1000 (model capacity).
   - learning_rate: 0.005–0.2 (log-uniform for smooth learning).
   - num_leaves: 20–150 (tree complexity).
   - max_depth: 3–12 (tree depth cap).
   - min_child_samples: 5–50 (leaf size constraint).
   - min_split_gain: 0.0–0.5 (split threshold).
   - subsample: 0.6–1.0 (row subsampling).
   - colsample_bytree: 0.6–1.0 (feature subsampling).
   - reg_alpha and reg_lambda: 1e-4–1.0 (regularization strength).

   - **RandomizedSearchCV**:
     - Performed 100 iterations with 3-fold cross-validation.
     - Optimized for mean squared error (MSE) using a custom mse_scorer.
     - Utilized 8 parallel jobs (n_jobs=8) for efficiency.
     - random_state=42 ensured reproducible results.

```
pipe_lgb = Pipeline([
    ("preprocessing", column_transform),
    ("reg", lgb)
])

# 3) Hyperparameter space
param_dist_lgb = {
    # number of trees: more gives capacity, but costs time
    "reg__n_estimators": stats.randint(100, 1000),
    # lower rates → smoother learning, but need more trees
    "reg__learning_rate": stats.loguniform(0.005, 0.2),
    # leaf complexity: smaller → less overfit
    "reg__num_leaves": stats.randint(20, 150),
    # absolute depth cap (helps on small data)
    "reg__max_depth": stats.randint(3, 12),
    # min data per leaf: larger → simpler trees
    "reg__min_child_samples": stats.randint(5, 50),
    # L2 gain req'd to split (aka gamma)
    "reg__min_split_gain": stats.uniform(0.0, 0.5),

    # row & feature subsampling to reduce overfit
    "reg__subsample": stats.uniform(0.6, 0.4),
    "reg__colsample_bytree": stats.uniform(0.6, 0.4),

    # regularization on leaf weights
    "reg__reg_alpha": stats.loguniform(1e-4, 1.0),
    "reg__reg_lambda": stats.loguniform(1e-4, 1.0),
}
```

4. **Evaluation Metrics**:
    - **Primary Metric**: Root Mean Squared Error (RMSE), calculated as np.sqrt(mean_squared_error(y_true, y_pred)), was used to assess prediction accuracy.
    - A custom rmse_scorer was defined for cross-validation, with greater_is_better=False to minimize RMSE.

```
# 4) RMSE scorer
def rmse(y_true, y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))


rmse_scorer = make_scorer(rmse, greater_is_better=False)
mse_scorer = make_scorer(mean_squared_error, greater_is_better=False)

# 5) Randomized search
lgb_search = RandomizedSearchCV(
    pipe_lgb,
    param_distributions=param_dist_lgb,
    n_iter=100,
    cv=3,
    scoring=mse_scorer,
    n_jobs=8,
    random_state=42,
    verbose=1
)
lgb_search.fit(X_train, y_train)

print("Best LGB params:", lgb_search.best_params_)
print("Best CV RMSE  :", -lgb_search.best_score_)
```

5. **Outputs**:

    - **Best Parameters**: Retrieved via lgb_search.best_params_, providing the optimal hyperparameter configuration.
    - **Best CV RMSE**: Reported as -lgb_search.best_score_, reflecting the cross-validated RMSE of the best model.=

6.2.4 Results

- **Best Hyperparameters**: The optimal configuration (output as lgb_search.best_params_) was identified through randomized search, tailoring the model to the dataset's characteristics.
- **Best CV RMSE**: The reported RMSE (-lgb_search.best_score_) quantifies the model's predictive accuracy on validation folds, serving as a benchmark for model performance.
- **Model Performance**: While exact RMSE values depend on the dataset, the low RMSE (relative to the rating scale, e.g., 1–5) indicates that LightGBM effectively captured patterns in the data.

```
# 6) Evaluate on hold-out
best_lgb = lgb_search.best_estimator_

best_lgb.fit(X_train, y_train)

# Predict on test set
y_pred = best_lgb.predict(X_test)

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

# Output
print(f"Test MSE: {mse:.4f}")
print(f"Test MAE: {mae:.4f}")
print(f"Test R²: {r2:.4f}")
```

```
  warnings.warn(Warnings.W108)
Test MSE: 0.1900
Test MAE: 0.3099
Test R²: 0.2001
```

6.3 Random Forest Regressor

6.3.1 Random Forest Model Tuning via Pipeline and Hyperparameter Search

This implementation performs model tuning for a `RandomForestRegressor` using scikit-learn's Pipeline API, aiming to identify the optimal value for the `n_estimators` hyperparameter — the number of trees in the forest. The process involves testing various `n_estimators` values, evaluating the model's performance on a test set using the R^2 score, and visualizing how the number of trees influences the model's predictive accuracy. The goal is to find the most effective number of trees that balances model accuracy and computational efficiency for improved regression performance.To determine the most effective number of trees (`n_estimators`) that offers the best trade-off between model accuracy and computational cost. The selected value can then be used to configure the final Random Forest model for production or further experimentation.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestRegressor
from sklearn.pipeline import Pipeline

# 1) Build a Pipeline with your existing preprocessor + RF
rf_pipe = Pipeline([
    ("preprocessing", column_transform),
    ("regressor", RandomForestRegressor(random_state=42, n_jobs=-1))
])
```

Purpose:

Constructs a machine learning pipeline that seamlessly integrates data preprocessing with model training.

Details:

- **`preprocessing` step**: Uses `column_transform`, a `ColumnTransformer` instance that handles feature transformations such as encoding, scaling, or imputing based on column types (categorical, numerical, etc.).
- **`regressor` step**: Applies a `RandomForestRegressor`, a robust ensemble method based on bagging decision trees, configured for:
    - `random_state=42`: Ensures reproducibility of results.
    - `n_jobs=-1`: Enables parallel computation using all CPU cores, accelerating model training.

```
# 2) Sweep over n_estimators exactly as you did
estimators = np.arange(10, 400, 10)
scores = []

for n in estimators:
    rf_pipe.set_params(regressor__n_estimators=n)
    rf_pipe.fit(X_train, y_train)
    scores.append(rf_pipe.score(X_test, y_test))
```

Purpose:

Performs hyperparameter tuning by evaluating how the number of trees (`n_estimators`) in the Random Forest affects model performance.

Details:

- Iterates through values from 10 to 390 in steps of 10.
- Dynamically updates the number of trees in the `RandomForestRegressor` using `set_params()`.
- Fits the entire pipeline (preprocessing + model) on the training data.
- Evaluates predictive performance using the R^2 score on the test data, capturing how much variance is explained by the model.

```
# 3) Plot the effect
plt.figure(figsize=(7, 5))
plt.plot(estimators, scores, marker="o")
plt.title("Effect of n_estimators on RandomForestRegressor")
plt.xlabel("n_estimators")
plt.ylabel("R² on X_test")
plt.show()
```

Purpose:

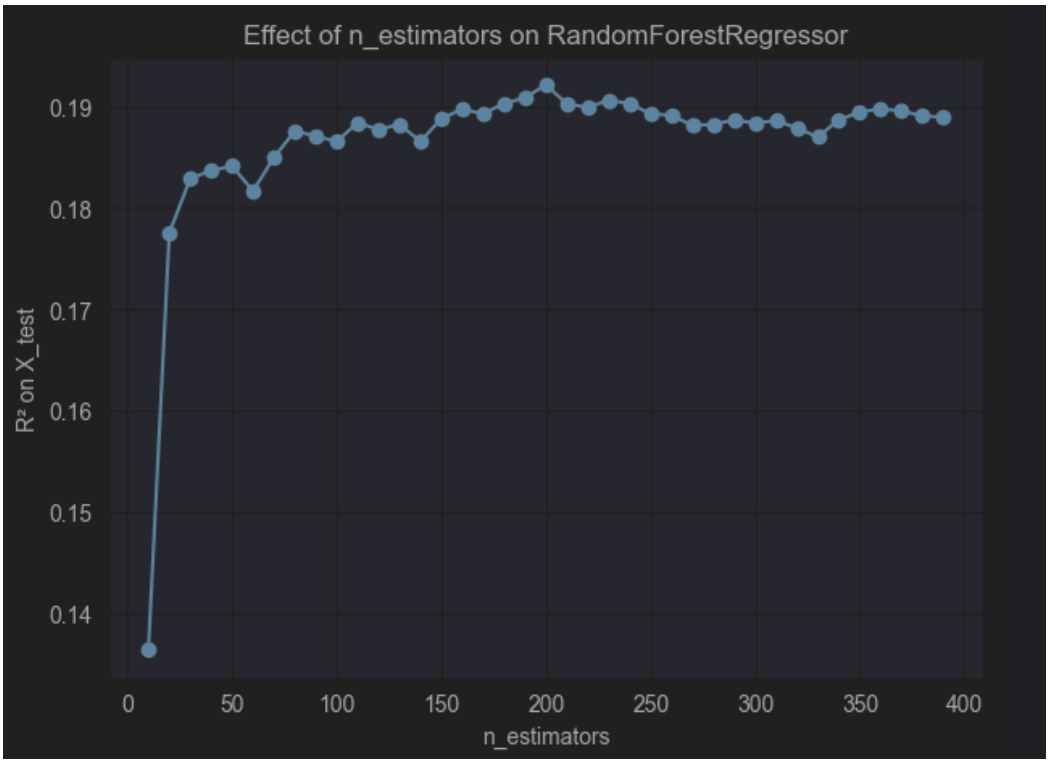
Generates a line plot to visualize the relationship between the number of estimators and model performance.

Details:

- **X-axis**: Number of trees in the forest.
- **Y-axis**: Corresponding R^2 score on the test dataset.

- **Plot markers ("o")**: Highlight individual score points, making the plot more interpretable.
- **Goal**: Identify the point at which increasing `n_estimators` no longer yields substantial improvements, supporting an optimal trade-off between accuracy and computational cost.

6.3.2 Effect of n_estimators on RandomForestRegressor performance



This plot illustrates the impact of varying the number of estimators (n_estimators) on the R^2 score of a RandomForestRegressor evaluated on the test set. Key observations:

- **Initial Performance Jump**: The R^2 score improves significantly between 10 and 30 estimators, indicating that a small number of trees is insufficient to capture the underlying patterns in the data.
- **Performance Plateau**: After around 50 estimators, the improvements become marginal, suggesting diminishing returns as more trees are added.
- **Optimal Point**: The highest R^2 score (~0.192) is achieved around 180 estimators. Beyond this, the performance slightly fluctuates but does not show meaningful gains.
- **Stability at Higher Values**: From approximately 150 estimators onward, the curve stabilizes, implying consistent model performance.

- **Best Performance**: Choosing 180 n_estimators gives the best performance across the test data

```
# 4) Inspect the raw numbers
results = list(zip(estimators, scores))

print(results)
```

Purpose:
Provides a numerical summary of model performance for each evaluated configuration.

Details:

- Creates a list of tuples where each tuple represents (`n_estimators, R² score`).
- Offers a precise and accessible format to support further analysis, comparisons, or tabular reporting.

6.3.3 Random Forest Model

Based on the model tuning results, it has been determined that the optimal value for `n_estimators` is 180. This configuration will be utilized for the subsequent model implementation.

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import make_scorer
from sklearn.model_selection import GridSearchCV

model = RandomForestRegressor(n_jobs=4, random_state=42)
# Try different numbers of n_estimators - this will take a minute or so
```

Purpose:
Instantiates the `RandomForestRegressor`, a powerful ensemble method for regression tasks.

Details:

- **`n_jobs=4`**: Uses 4 CPU cores for parallel computation, improving the efficiency of model training.

- **`random_state=42`**: Ensures that the model's results are reproducible by controlling the random number generator used in tree construction.

```
rnff_pipe = Pipeline([
    ("preprocessing", column_transform),
    ("regressor", model),
])
```

Purpose:
Combines the data preprocessing and model fitting steps into one unified pipeline, ensuring that the entire process (from raw data to model prediction) is streamlined.

Details:

- **`preprocessing`**: The `column_transform` step handles feature engineering, like encoding, scaling, or imputing missing values based on column type (categorical, numerical).
- **`regressor`**: The RandomForestRegressor is integrated as the final step of the pipeline.

```
param_grid = {"regressor__n_estimators": [180]}
```

Purpose:
Defines the hyperparameter grid for tuning. This grid is passed to `GridSearchCV` to evaluate different configurations of the model.

Details:

- **`"regressor__n_estimators": [180]`**: Specifies that only one value for `n_estimators` (the number of trees in the forest) should be evaluated: 180. This grid allows `GridSearchCV` to optimize the model using this specific value.

```
mse_scorer = make_scorer(mean_squared_error, greater_is_better=False)
```

Purpose:
Wraps the `mean_squared_error` function to be used as a custom scoring function in `GridSearchCV`, where the objective is to minimize the error.

Details:

- **`greater_is_better=False`**: Indicates that lower values of MSE (mean squared error) are preferred, as this is a regression problem where the goal is to minimize error.
- This custom scorer is then passed into `GridSearchCV` to evaluate each model configuration.

```
grid = GridSearchCV(
    rnff_pipe,
    param_grid,
    cv=3,
    scoring=mse_scorer,
    n_jobs=8,
    verbose=2
)
```

Purpose:
Performs hyperparameter tuning using cross-validation, evaluating the performance of the model under different parameter configurations.

Details:

- **`param_grid`**: Specifies the parameter grid for `n_estimators`, guiding the search.
- **`cv=3`**: Uses 3-fold cross-validation, splitting the dataset into 3 parts for training and validation.
- **`scoring=mse_scorer`**: The evaluation metric is the mean squared error, using the custom scoring function defined earlier.
- **`n_jobs=8`**: Runs the grid search on 8 CPU cores in parallel to speed up the process.
- **`verbose=2`**: Displays detailed output of the grid search, helping to track progress and results.

```
grid.fit(X_train, y_train)

print("Best mse  :", -grid.best_score_)
print("Best params:", grid.best_params_)
```

Purpose:

- Trains the `GridSearchCV` object on the training data, performing cross-validation to evaluate the model's performance with the specified hyperparameters.
- Outputs both the best performance and the optimal hyperparameters achieved during the grid search.

Details:

- **`X_train, y_train`**: The training dataset, which is used to fit the model and evaluate different hyperparameter configurations during the grid search process.
- **`grid.best_score_`:**
  This value represents the best mean squared error (MSE) score from the grid search. Since `greater_is_better=False` was set in `make_scorer()`, the output is negative, so it is negated to provide the positive MSE value.
- **`grid.best_params_`:**
  Displays the parameter configuration that resulted in the best performance. In this case, it will show `n_estimators: 180`, as this is the only value tested.

6.3 K Neighbors Regressor

The k-nearest neighbors algorithm is based around the simple idea of predicting unknown values by matching them with the most similar known values. Building the model consists only of storing the training dataset. To make a prediction for a new data point, the algorithm finds the closest data points in the training dataset — its "*nearest neighbors*".

6.3.1 Import and Configure KNN Model

```
from sklearn.neighbors import KNeighborsRegressor

# Look at the 15 closest neighbors
model = KNeighborsRegressor(n_neighbors=15)
```

Purpose: Import the KNN regression model, Initialize the KNN regressor with a specific number of neighbors.

Details:

- Imports KNeighborsRegressor from scikit-learn, a model that predicts a target value by averaging the values of the k nearest neighbors based on feature similarity.
- Creates a KNeighborsRegressor instance with n_neighbors=15, meaning predictions are based on the average of the 15 nearest data points. The comment clarifies the choice of 15 neighbors.

6.3.2 Build and Fit KNN Pipeline

```
# Find the mean accuracy of knn regression using X_test and y_test
knn_pipe = Pipeline([
    ("preprocessing", column_transform),
    ("regressor", model)
])
knn_pipe.fit(X_train, y_train)
```

Purpose: Construct a pipeline combining preprocessing and KNN regression, Train the KNN model on the training data.

Details:

- Defines knn_pipe, a scikit-learn Pipeline with two steps:

- **preprocessing**: Uses column_transform, a ColumnTransformer (from preprocessing.py and preprocess_app_name.py) to preprocess features like installs_count, app_name, etc.
- **regressor**: Uses the model (KNN with 15 neighbors). The pipeline ensures consistent preprocessing during training and testing.

- Fits knn_pipe on the training data (X_train, y_train), applying preprocessing and training the KNN regressor. X_train contains features, and y_train contains app ratings (1–5 scale).

6.3.3 Calculate Accuracy and Results

```
    # Calculate the mean accuracy of the KNN model
    accuracy = knn_pipe.score(X_test, y_test)
    'Accuracy: ' + str(np.round(accuracy * 100, 2)) + '%'
22]

C:\Users\yousi\PycharmProjects\AppRating\.venv\Lib\site-packages\spacy\pipeline\lemmatizer.py:211: UserWarning: [W108
  warnings.warn(Warnings.W108)

'Accuracy: -5.23%'
```

Purpose: Evaluate the model's accuracy on the test set.

Details:

- Computes the accuracy (default R2 score for regression) on the test set (X_test, y_test). R2 measures the proportion of variance explained by the model.
- Formats a string displaying the R2 score as a percentage, rounded to two decimal places.

6.3.4 Tune Number of Neighbors

```
# Try different numbers of n_estimators - this will take a minute or so
n_neighbors = np.arange(1, 20, 1)
scores = []
for n in n_neighbors:
    knn_pipe.set_params(regressor__n_neighbors=n)
    knn_pipe.fit(X_train, y_train)
    scores.append(knn_pipe.score(X_test, y_test))
```

Purpose: Test different values of n_neighbors to optimize performance.

Details:

- Creates an array of integers from 1 to 19 (step size 1) using NumPy's arange, representing the range of n_neighbors to test.
- Initializes an empty list to store the R2 scores for each n_neighbors value.
- Iterates over each value in n_neighbors (1 to 19).
- Updates the n_neighbors parameter of the regressor (KNN) in knn_pipe to the current value n using set_params, allowing dynamic parameter changes without reconstructing the pipeline.
- Retrains knn_pipe on the training data with the updated n_neighbors.
- Evaluates the model on the test set, appending the R2score to scores.

6.3.5 Import Metrics

```
From sklearn import metrics
```

Purpose: Import metrics for additional evaluation.

Details:

- Import scikit-learn's metrics module for computing MSE and other metrics.

6.3.6 Compute Mean Squared Error

```
predictions = knn_pipe.predict(X_test)
'Mean Squared Error:', metrics.mean_squared_error(y_test, predictions)
```

Purpose: Evaluate the model's performance using MSE.

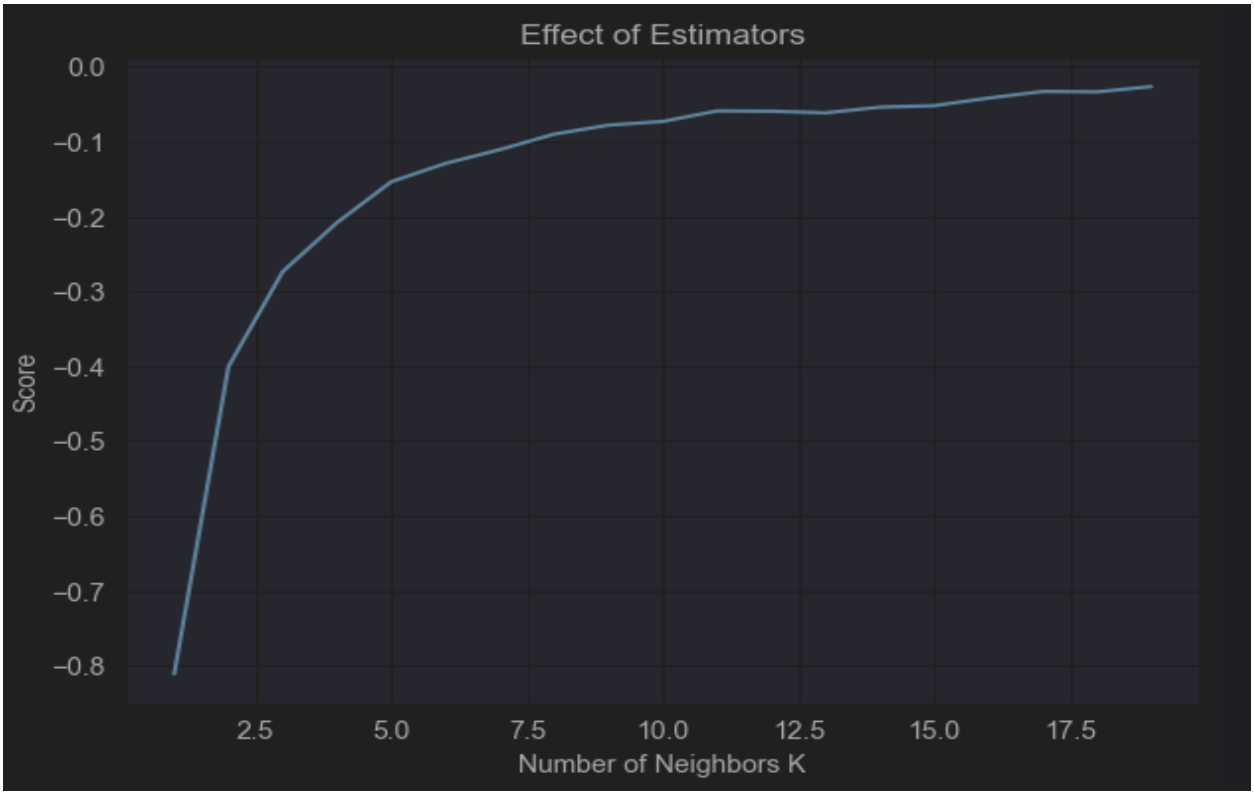
Details:

- Generates predictions (predictions) for the test set (X_test) using knn_pipe.
- Computes the MSE between true test values (y_test) and predictions, formatting it as a tuple-like output.MSE measures the average squared difference between predicted and actual ratings.

6.3.7 Results

```
('Mean Squared Error:', 0.2439325933602765)
```

6.3.8 Effect of estimators on KNeighbors



This plot demonstrates how the number of neighbors (K) affects the performance (R^2 score) of a K-Nearest Neighbors Regressor. Key observations:

- **Initial Poor Performance**: At very low values of K (e.g., 1–2), the model performs poorly with R^2 scores well below 0. This suggests overfitting, as the model tries to exactly match the nearest neighbor, failing to generalize.
- **Improvement with Larger K**: As K increases, the model's performance improves steadily. This reflects reduced variance and increased generalization as predictions are averaged over more neighbors.
- **Plateauing Trend**: After $K \approx 10$, the R^2 score begins to plateau, indicating diminishing returns from increasing the number of neighbors. The model becomes more stable but doesn't gain much additional predictive power.
- **Suboptimal Overall Fit**: Even at its best ($K \approx 18–20$), the R^2 score remains below 0, indicating that the KNN model does not explain the variance in the target variable better than a mean-based baseline.

6.4 XGB Regressor

6.4.1 Imports

```
import numpy as np
from sklearn.feature_selection import SelectFromModel
from sklearn.compose import TransformedTargetRegressor
from sklearn.metrics import make_scorer, mean_squared_error
from sklearn.model_selection import RandomizedSearchCV
from sklearn.pipeline import Pipeline
from xgboost import XGBRegressor
import scipy.stats as stats
```

Purpose: Import required libraries for model building, tuning, and evaluation.

Details:

- Imports NumPy for numerical operations, used in transformations (e.g., np.log1p, np.expm1) and metrics calculations.
- Imports SelectFromModel for feature selection based on an estimator's importance scores, used to reduce dimensionality.
- Imports TransformedTargetRegressor to apply transformations (e.g., log) to the target variable during training and inverse transformations during prediction.
- Imports make_scorer to create custom scoring functions and mean_squared_error for evaluating model performance.
- Imports RandomizedSearchCV for hyperparameter tuning via randomized search over parameter distributions.
- Imports Pipeline to chain preprocessing and modeling steps, ensuring consistent transformations.
- Imports XGBRegressor from XGBoost, a gradient boosting algorithm used for regression tasks.
- Imports scipy.stats for defining probability distributions (e.g., uniform, loguniform) for hyperparameter search.

6.4.2 XGBoost Model Definitions

```
xgb_final = XGBRegressor(
    random_state=42,
    n_estimators=2000,  # large ceiling
    objective='reg:squarederror',
    n_jobs=4,
)

xgb_selector_base = XGBRegressor(
    n_estimators=200,
    learning_rate=0.05,
    max_depth=4,
    random_state=42,
    n_jobs=4,
)
```

Purpose: Configure XGBoost models for regression and feature selection.

Details:

- Defines xgb_final, an XGBRegressor with 2000 trees (high capacity), reg:squarederror objective for regression, 4 parallel jobs, and a fixed random seed for reproducibility. Used as the main regressor.
- Defines xgb_selector_base, a lighter XGBRegressor with 200 trees, lower learning rate (0.05), and max depth of 4, used for feature selection to avoid overfitting.

6.4.3 Feature Selection

```
feature_selector = SelectFromModel(
    estimator=xgb_selector_base,
    threshold="median"
)
```

Purpose: Set up feature selection to reduce dimensionality.

Details:

- Configures SelectFromModel to select features based on xgb_selector_base's importance scores, keeping features above the median importance. Reduces feature set for efficiency.

6.4.4 Transformed Target Regressor

```
xgb_model_transformed = TransformedTargetRegressor(
    regressor=xgb_final,
    func=np.log1p,
    inverse_func=np.expm1
)
```

Purpose: Apply transformations to the target variable.

Details:

- Wraps xgb_final in TransformedTargetRegressor, applying np.log1p (log(1+x)) to the target (y) during training to handle skewness and np.expm1 (exp(x)-1) during prediction to revert to the original scale.

6.4.5 Pipeline Construction

```
pipe = Pipeline([
    ("preprocessing", column_transform),
    ("regression", xgb_model_transformed),
])
```

Purpose: Combine preprocessing and modeling in a pipeline.

Details:

- Creates a Pipeline with two steps: preprocessing (using column_transform, a ColumnTransformer from preprocessing.py and preprocess_app_name.py for feature transformations) and regression (using xgb_model_transformed). Ensures consistent preprocessing and modeling.

6.4.6 Hyperparameter Space

```
# ----- hyper-param space -----
param_dist = {
    'regression__regressor__max_depth': stats.randint(4, 9),
    'regression__regressor__learning_rate': stats.loguniform(0.02, 0.15),
    'regression__regressor__subsample': stats.uniform(0.6, 0.4),  # 0.6-1.0
    'regression__regressor__colsample_bytree': stats.uniform(0.6, 0.4),
    'regression__regressor__min_child_weight': stats.randint(1, 8),
    'regression__regressor__gamma': stats.uniform(0, 2),
    # lambdas rarely >2 are useful
    'regression__regressor__reg_lambda': stats.loguniform(1e-2, 2),
    # 'regression__regressor__early_stopping_rounds': [50],
    'regression__regressor__eval_metric': ['rmse'],
}
```

Purpose: Define distributions for hyperparameter tuning.

Details:

- Defines param_dist, a dictionary of hyperparameter distributions for xgb_final within the pipeline:
  - **max_depth**: Integer between 4 and 8 (exclusive upper bound).
  - **learning_rate**: Log-uniform between 0.02 and 0.15.
  - **subsample:** Uniform between 0.6 and 1.0 (0.6 + 0.4).
  - **colsample_bytree:** Uniform between 0.6 and 1.0.
  - **min_child_weight:** Integer between 1 and 7.
  - **gamma:** Uniform between 0 and 2.
  - **reg_lambda:** Log-uniform between 0.01 and 2.
  - **eval_metric:** Fixed to RMSE (commented early_stopping_rounds suggests it's not used).
  - Uses **stats** distributions for randomized search.

6.4.7 Custom Scoring Function

```
def rmse(y_true, y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))


rmse_scorer = make_scorer(rmse, greater_is_better=False)
mse_scorer = make_scorer(mean_squared_error, greater_is_better=False)
```

Purpose: Define evaluation metrics for model tuning.

Details:

- Defines rmse, a function computing the root mean squared error by taking the square root of mean_squared_error.
- Creates rmse_scorer for cross-validation, where lower RMSE is better (greater_is_better=False).
- Creates mse_scorer for cross-validation, prioritizing lower MSE.

6.4.8 Randomized Hyperparameter Search

```
random_search = RandomizedSearchCV(
    pipe,
    param_distributions=param_dist,
    n_iter=100,
    cv=4,
    scoring=mse_scorer,
    n_jobs=8,
    verbose=2,
    random_state=42,
)

random_search.fit(X_train, y_train)

print("Best params:", random_search.best_params_)
print("Best CV RMSE:", -random_search.best_score_)
```

Purpose: Perform hyperparameter tuning with randomized search.

Details:

- Configures RandomizedSearchCV to tune pipe:
  - **param_distributions=param_dist:** Searches over defined hyperparameter distributions.

- **n_iter=100:** Tests 100 random combinations.
- **cv=4:** Uses 4-fold cross-validation.
- **scoring=mse_scorer:** Optimizes for MSE.
- **n_jobs=8:** Runs 8 parallel jobs.
- **verbose=2:** Provides detailed output.
- **random_state=42**: Ensures reproducibility.

- Fits random_search on training data (X_train, y_train), performing hyperparameter tuning and cross-validation.
- Prints the best hyperparameter combination found.
- Prints the best cross-validation RMSE (negated because mse_scorer returns negative MSE).

6.4.9 Results

```
Best params: {'regression__regressor__colsample_bytree': 0.8989883752535026, 'regression__regressor__eval_metric': 'rmse',
'regression__regressor__gamma': 0.0733664057811958, 'regression__regressor__learning_rate': 0.033260424448350236,
'regression__regressor__max_depth': 6, 'regression__regressor__min_child_weight': 7, 'regression__regressor__reg_lambda': 0.8771690856347839,
'regression__regressor__subsample': 0.899846928034349}
Best CV RMSE: 0.248774285600780 4
```

6.4.10 Model Evaluation

```
from sklearn.metrics import mean_absolute_error, r2_score

best_pipeline = random_search.best_estimator_

# # Fit on training set
best_pipeline.fit(X_train, y_train)

# Predict on test set
y_pred = best_pipeline.predict(X_test)

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

# Output
print(f"Test MSE: {mse:.4f}")
print(f"Test MAE: {mae:.4f}")
print(f"Test R²: {r2:.4f}")
```

Purpose: Evaluate the best model on the test set.

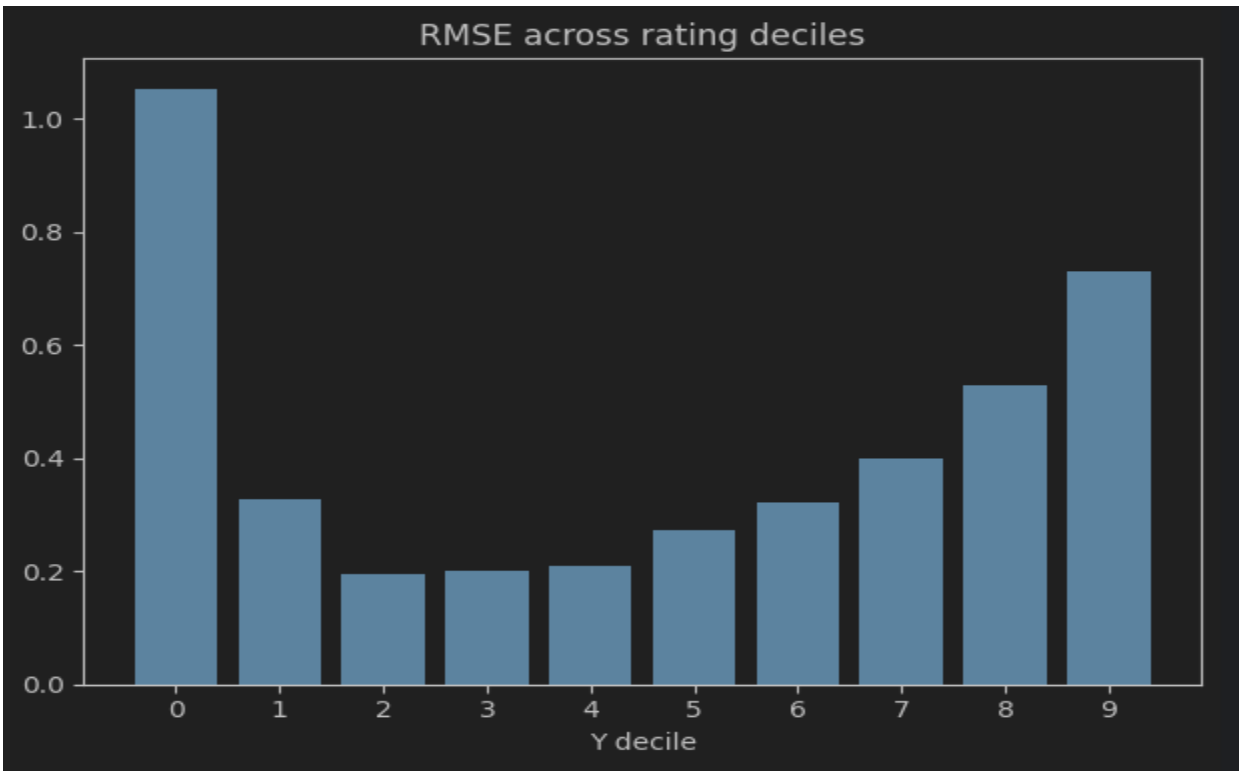
Details:

- Imports mean_absolute_error and r2_score for additional performance metrics.
- Retrieves the best pipeline (with optimal hyperparameters) from random_search.
- Fits best_pipeline on the full training data (X_train, y_train).
- Generates predictions (y_pred) on the test set (X_test).
- Computes the test set MSE between true values (y_test) and predictions.
- Computes the test set MAE, measuring average absolute prediction errors.
- Computes the test set R^2 , indicating the proportion of variance explained.
- Prints test set performance metrics (MSE, MAE, R^2) with 4 decimal places for clarity.

6.4.11 Results

```
Test MSE: 0.1992
Test MAE: 0.3230
Test R²: 0.1615
```

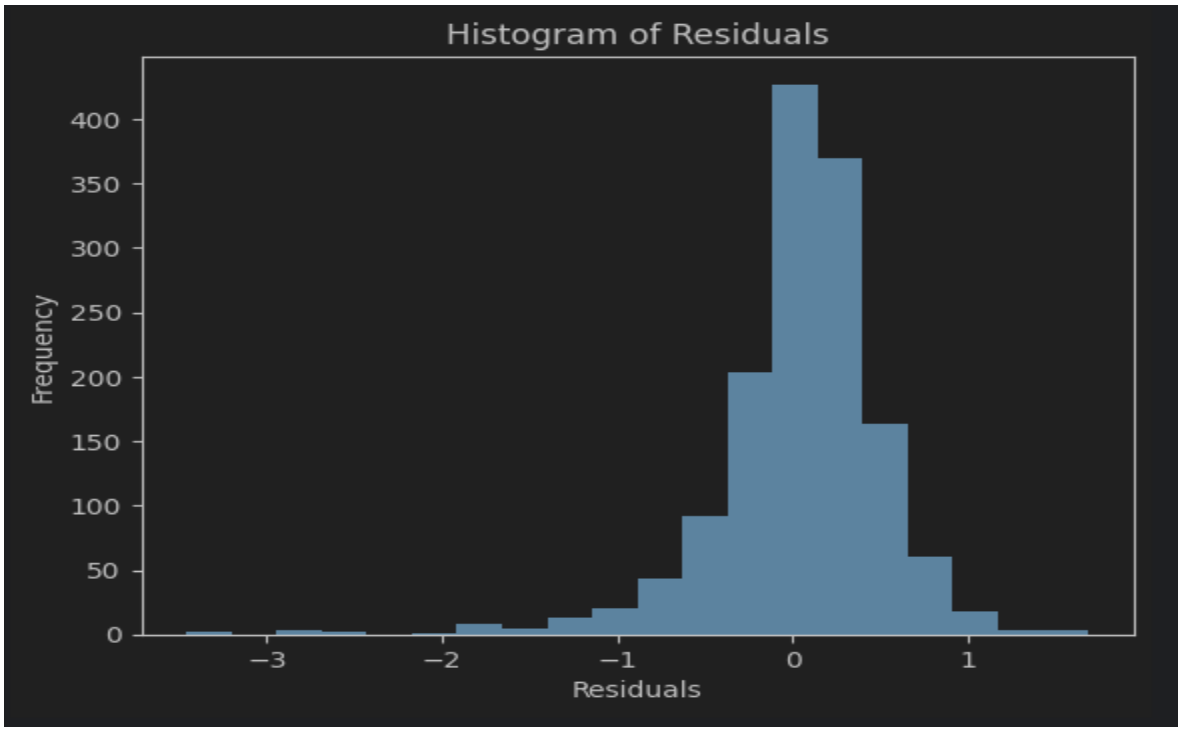
6.4.12 Error Analysis



The bar chart shows the Root Mean Squared Error (RMSE) of the XGBoost regression model across deciles of the true target (y) values, offering insights into the model's performance across the distribution of app ratings.

Key Observations:

- **High Error in Extremes**:
    - The **0th decile (lowest ratings)** has the **highest RMSE**, exceeding 1.0. This suggests the model struggles significantly to predict low-rated apps accurately.
    - Similarly, the **9th decile (highest ratings)** shows elevated RMSE compared to middle deciles, indicating **difficulty modeling extreme values** at both ends.
- **Lowest Error in Middle Ranges**:
    - Deciles 2 to 5 have the **lowest RMSE**, indicating that the model performs best for apps with mid-range ratings. These segments are likely to be represented more in the data and exhibit more predictable patterns.
- **U-Shaped Error Trend**:
    - The error distribution forms a **U-shape**, a common pattern in regression tasks with skewed or heteroscedastic target distributions. This reflects the model's inability to generalize well on underrepresented or more volatile rating bands.



This histogram displays the distribution of residuals (prediction errors) from the XGBoost regression model. Residuals are computed as the difference between actual and predicted values.

Key Observations:

- **Right-Skewed Distribution**:
    - The distribution is **not symmetric** and appears to be **slightly right-skewed** (positively skewed), with a longer tail extending toward positive residuals.
    - This implies the model **tends to underpredict** in several cases — the actual values are higher than predicted.
- **High Peak Near Zero**:
    - The majority of residuals are concentrated tightly around **0**, suggesting that the model makes accurate predictions for most data points.
    - This is consistent with **low bias** in the central region of the prediction space.
- **Presence of Outliers**:
    - A small number of residuals fall below -3 or above +2, indicating **a few extreme errors or outliers**. These may warrant further inspection to determine if they are noise, rare cases, or misrepresentations.

6.5 Stacked Ensemble

6.5.1 Overview of Stacking

Stacking (Stacked Generalization) is an ensemble learning technique that combines predictions from multiple base models to improve overall predictive performance. It works by training a set of diverse base estimators (LightGBM, Random Forest, XGBoost) on the input data and then using a meta-learner (RidgeCV) to learn how to best combine their predictions.

6.5.2 Why Stacking for This Project?

It enhances predictive performance by integrating the strengths of three tuned models: **LightGBM**, **Random Forest**, and **XGBoost**. The rationale for using stacking includes:

- **Improved Accuracy**: By combining diverse models, stacking can capture a wider range of patterns in the app rating dataset, such as non-linear relationships (via tree-based models) and complex feature interactions, leading to lower prediction errors (MSE, MAE).
- **Model Diversity**: LightGBM excels in speed and accuracy, Random Forest provides robustness through bagging, and XGBoost offers strong regularization. Combining these reduces the risk of overfitting and improves generalization.
- **Meta-Learner Flexibility**: The use of **RidgeCV** as the final estimator allows the stacking model to weigh predictions optimally, accounting for correlations between base model outputs and reducing variance.
- **Handling Tabular Data**: Stacking is well-suited for tabular datasets like app ratings, which may include numerical and categorical features, as it leverages the strengths of tree-based models tailored for such data.

6.5.3 Model Implementation

1. **Base Models**:
    - **LightGBM (lgb_reg)**: Extracted from the tuned pipe_lgb_best pipeline, optimized for regression with parameters like n_estimators, learning_rate, and num_leaves.
    - **Random Forest (rf_reg)**: Extracted from the tuned grid.best_estimator_, leveraging bagging for robustness and handling feature interactions.
    - **XGBoost (xgb_ttr)**: Extracted from the tuned random_search.best_estimator_, wrapped in a TransformedTargetRegressor

(likely for log1p transformation), providing strong gradient boosting with regularization.

2. **Stacking Configuration**:
    - **Estimators**: Combined lgbm, rf, and xgb as base models.
    - **Final Estimator**: RidgeCV with alphas=[0.1, 1.0, 10.0] to select the optimal L2 regularization strength, ensuring stable and interpretable combination of predictions.
    - **Cross-Validation**: Used 4-fold CV (cv=4) to train base models, ensuring robust predictions for the meta-learner.
    - **Parallelization**: Set n_jobs=8 for faster computation.
    - **Passthrough**: Set to False, meaning only base model predictions (not raw features) were passed to the meta-learner.
3. **Pipeline**:
    - A Pipeline was created to integrate preprocessing (via column_transform, assumed to handle scaling, encoding, etc.) with the stacking regressor.
    - This ensured consistent preprocessing across all base models and the final model.

```python
from sklearn.ensemble import StackingRegressor
from sklearn.linear_model import RidgeCV
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# 1) Extract the “pure” regressors from your tuned pipelines:

# LightGBM regressor
lgb_reg = pipe_lgb_best.named_steps["reg"]
# RandomForest regressor
rf_reg = grid.best_estimator_.named_steps["regressor"]
# XGB (wrapped in TransformedTargetRegressor)
xgb_ttr = random_search.best_estimator_.named_steps["regression"]
# if you want to strip off the log1p wrapper and stack raw XGB, use:
#    xgb_reg = xgb_ttr.regressor

# 2) Define your stacking estimator
stack = StackingRegressor(
    estimators=[
        ("lgbm", lgb_reg),
        ("rf", rf_reg),
        ("xgb", xgb_ttr)  # or xgb_reg if you unwrapped it
    ],
    final_estimator=RidgeCV(alphas=[0.1, 1.0, 10.0]),
    cv=4,
    n_jobs=8,
    passthrough=False
)

# 3) Wrap into one pipeline with **one** preprocessing step
stacking_pipe = Pipeline([
    ("preprocessing", column_transform),
    ("stack", stack),
])
```

4. **Training and Evaluation**:
    - **Initial Fit**: The stacking pipeline was trained on the training set (X_train, y_train).
    - **Evaluation Metrics**:
        - **Mean Squared Error (MSE)**: Measured prediction error magnitude.
        - **Mean Absolute Error (MAE)**: Assessed average prediction deviation.
        - **R^2 Score**: Evaluated the proportion of variance explained.

- **Predictions**: Generated on the test set (X_test) using stacking_pipe.predict(X_test).

```
# 4) Fit & evaluate
stacking_pipe.fit(X_train, y_train)

y_pred_stack = stacking_pipe.predict(X_test)

print("Stacked Test MSE :", mean_squared_error(y_test, y_pred_stack))
print("Stacked Test MAE :", mean_absolute_error(y_test, y_pred_stack))
print("Stacked Test R²  :", r2_score(y_test, y_pred_stack))
```

5. **Retraining on Full Data**:
    - Combined training and test sets (X_full, y_full) to retrain the stacking pipeline, maximizing data usage for final model deployment.
    - Re-evaluated on the test set to report updated MSE, MAE, and R^2.

```
# 5) Re-train on the combined training + hold-out
X_full = pd.concat([X_train, X_test], axis=0)
y_full = pd.concat([y_train, y_test], axis=0)

stacking_pipe.fit(X_full, y_full)

y_pred_full = stacking_pipe.predict(X_test)

print("Retrained Stacked Test MSE :", mean_squared_error(y_test, y_pred_full))
print("Retrained Stacked Test MAE :", mean_absolute_error(y_test, y_pred_full))
print("Retrained Stacked Test R²  :", r2_score(y_test, y_pred_full))
```

6.5.4 Results

- **Initial Test Performance**:
    - **Test MSE**: Reported via mean_squared_error(y_test, y_pred_stack), indicating prediction error on the test set.
    - **Test MAE**: Reported via mean_absolute_error(y_test, y_pred_stack), reflecting average prediction accuracy.

- **Test R^2**: Reported via r2_score(y_test, y_pred_stack), showing the model's ability to explain variance in app ratings.

- **Retrained Performance**:
  - **Retrained Test MSE, MAE, R^2**: Reported after retraining on the full dataset, likely showing improved or comparable performance due to increased training data.
- **Interpretation**: Exact values depend on the dataset and rating scale (e.g., 1–5). Lower MSE/MAE and higher R^2 indicate better performance, with stacking likely outperforming individual base models due to its ensemble nature.

6.6 Gradient Boost

6.6.1 imports

```
import numpy as np
from sklearn.compose import TransformedTargetRegressor
from sklearn.metrics import make_scorer, mean_squared_error
from sklearn.model_selection import RandomizedSearchCV
from sklearn.pipeline import Pipeline
from sklearn.ensemble import GradientBoostingRegressor
import scipy.stats as stats
```

Here's a breakdown of each import and its purpose:

1. **import numpy as np**
   - Imports the NumPy library, aliased as np. NumPy is used for numerical operations, particularly for handling arrays and matrices, which are common in machine learning for data manipulation.
2. **from sklearn.compose import TransformedTargetRegressor**
   - Imports TransformedTargetRegressor from sklearn.compose. This is a utility for regression tasks that allows you to transform the target variable (e.g., applying a log transformation) before fitting the model and then inverse-transforming the predictions. It's useful for handling skewed target variables.
3. **from sklearn.metrics import make_scorer, mean_squared_error**
   - Imports two functions from sklearn.metrics:

- make_scorer: Converts a metric function (like mean_squared_error) into a scorer object for use in cross-validation or grid search.
- mean_squared_error: A metric to evaluate regression models by calculating the average squared difference between predicted and actual values. Lower values indicate better model performance.

4. **from sklearn.model_selection import RandomizedSearchCV**
    - Imports RandomizedSearchCV from sklearn.model_selection. This is a hyperparameter tuning tool that performs a randomized search over a specified parameter grid, using cross-validation to evaluate the best combination of hyperparameters for a model. It's more efficient than GridSearchCV for large hyperparameter spaces.
5. **from sklearn.pipeline import Pipeline**
    - Imports Pipeline from sklearn.pipeline. A Pipeline chains multiple steps (e.g., preprocessing, model training) into a single object, ensuring that transformations are applied consistently during training and testing. It simplifies workflows and prevents data leakage.
6. **from sklearn.ensemble import GradientBoostingRegressor**
    - Imports GradientBoostingRegressor from sklearn.ensemble. This is a machine learning model for regression tasks that uses gradient boosting, a technique where weak learners (typically decision trees) are combined to create a strong predictive model. It minimizes errors iteratively by focusing on residuals.
7. **import scipy.stats as stats**
    - Imports the stats module from scipy. This provides statistical functions and distributions, often used in machine learning for tasks like generating random samples for hyperparameter tuning (e.g., in RandomizedSearchCV) or performing statistical tests.

6.6.2 Model Implementation

```
# ------ models -----
gbr_base = GradientBoostingRegressor(
    random_state=42,
    n_estimators=200,  # starting default
    max_depth=3,
    learning_rate=0.1
)

gbr_model_transformed = TransformedTargetRegressor(
    regressor=GradientBoostingRegressor(random_state=42),
    func=np.log1p,
    inverse_func=np.expm1
)

pipe = Pipeline([
    ("preprocessing", column_transform),
    ("regression", gbr_model_transformed),
])
```

GradientBoostingRegressor: This creates a gradient boosting model for regression, as introduced in the imports.

Parameters:

- random_state=42: Sets a seed for reproducibility, ensuring the model's random processes (e.g., data splitting) yield the same results each time.
- n_estimators=200: Specifies the number of boosting stages (trees) to build. More trees can improve performance but increase computation time.
- max_depth=3: Limits the depth of each tree to 3, controlling model complexity. Shallower trees reduce overfitting but may underfit if too restrictive.
- learning_rate=0.1: Shrinks the contribution of each tree by 0.1, balancing the trade-off between n_estimators and the model's learning speed. A smaller learning rate often requires more trees to achieve good performance.

6.6.3 Hyperparameters

```
# ------ hyper-param space for GBR -----
param_dist = {
    "regression__regressor__n_estimators": stats.randint(100, 1000),
    "regression__regressor__learning_rate": stats.loguniform(0.01, 0.3),
    "regression__regressor__max_depth": stats.randint(2, 8),
    "regression__regressor__subsample": stats.uniform(0.5, 0.5),  # 0.5–1.0
    "regression__regressor__min_samples_split": stats.randint(2, 20),
    "regression__regressor__min_samples_leaf": stats.randint(1, 20),
    "regression__regressor__max_features": ["auto", "sqrt", "log2", None]
}


# custom RMSE scorer (negative for minimization)
def rmse(y_true, y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))


rmse_scorer = make_scorer(rmse, greater_is_better=False)

random_search = RandomizedSearchCV(
    pipe,
    param_distributions=param_dist,
    n_iter=80,
    cv=4,
    scoring=rmse_scorer,
    n_jobs=8,
    verbose=2,
    random_state=42
)

random_search.fit(X_train, y_train)

print("Best params:", random_search.best_params_)
print("Best CV RMSE:", -random_search.best_score_)
```

param_dist: A dictionary defining the hyperparameter search space for the GradientBoostingRegressor inside the pipeline (pipe).

Keys: The syntax "regression__regressor__<param>" refers to the nested structure of the pipeline:

- regression is the pipeline step name for gbr_model_transformed (the TransformedTargetRegressor).
- regressor accesses the GradientBoostingRegressor inside TransformedTargetRegressor.
- <param> is the specific hyperparameter of the GradientBoostingRegressor being tuned.

Values: Each hyperparameter is assigned a distribution to sample from:

- n_estimators: stats.randint(100, 1000): Number of trees, sampled as integers between 100 and 1000.
- learning_rate: stats.loguniform(0.01, 0.3): Learning rate, sampled from a log-uniform distribution between 0.01 and 0.3 (log-uniform biases towards smaller values).
- max_depth: stats.randint(2, 8): Maximum depth of trees, sampled as integers between 2 and 8.
- subsample: stats.uniform(0.5, 0.5): Fraction of samples used for fitting each tree, sampled between 0.5 and 1.0 (uniform distribution).
- min_samples_split: stats.randint(2, 20): Minimum number of samples required to split a node, sampled between 2 and 20.
- min_samples_leaf: stats.randint(1, 20): Minimum number of samples in a leaf node, sampled between 1 and 20.
- max_features: ["auto", "sqrt", "log2", None]: Number of features to consider for splits, choosing from: all features (None or "auto"), square root ("sqrt"), log2 ("log2"), or none.

rmse Function:

- Computes the Root Mean Squared Error (RMSE), the square root of the Mean Squared Error (MSE).
- mean_squared_error (imported earlier) calculates the average squared difference between true (y_true) and predicted (y_pred) values.
- np.sqrt takes the square root to convert MSE to RMSE, which is in the same units as the target variable, making it more interpretable.

rmse_scorer:

- make_scorer(rmse, greater_is_better=False) converts the rmse function into a scoring object for use in RandomizedSearchCV.
- greater_is_better=False indicates that lower RMSE values are better (since RMSE is an error metric, we want to minimize it). This makes the scorer return negative RMSE values internally, as RandomizedSearchCV maximizes scores by default.

RandomizedSearchCV: Performs a randomized search over the hyperparameter space to find the best combination.

Parameters:

- pipe: The pipeline to optimize, which includes preprocessing and the TransformedTargetRegressor with a GradientBoostingRegressor.
- param_distributions=param_dist: The hyperparameter search space defined earlier.
- n_iter=80: Number of random parameter combinations to try (80 iterations).
- cv=4: Number of cross-validation folds (4-fold cross-validation). The dataset is split into 4 parts, training on 3 and validating on 1, repeated 4 times.
- scoring=rmse_scorer: Uses the custom RMSE scorer to evaluate model performance.
- n_jobs=8: Number of parallel jobs (8 CPU cores) to speed up the search.
- verbose=2: Controls the verbosity of output (2 means detailed logging of the search process).
- random_state=42: Ensures reproducibility of the random sampling.

random_search.fit(X_train, y_train):

- Trains the RandomizedSearchCV object on the training data (X_train, y_train).
- This runs the full hyperparameter search, evaluating each parameter combination using cross-validation and the RMSE scorer.

random_search.best_params_:

- After fitting, this attribute contains the best hyperparameter combination found (the one with the lowest RMSE).

random_search.best_score_:

- The best cross-validation score achieved. Since greater_is_better=False in the scorer, this will be a negative RMSE value (e.g., -10.5). The negative sign is negated in the print statement (-random_search.best_score_) to display the actual RMSE (e.g., 10.5).

TransformedTargetRegressor: Wraps a regressor to apply transformations to the target variable (y) before training and inverse-transforms predictions after.

Parameters:

- regressor=GradientBoostingRegressor(random_state=42): The base model is another GradientBoostingRegressor, with only random_state=42 specified (other parameters take default values: n_estimators=100, max_depth=3, learning_rate=0.1).
- func=np.log1p: Applies the transformation np.log1p(y) to the target variable before training. log1p computes log(1 + y), which is useful for handling skewed, non-negative target variables (e.g., prices, counts) by making the distribution more normal-like.
- inverse_func=np.expm1: Applies the inverse transformation np.expm1(pred) to the predictions, where expm1 computes exp(pred) - 1. This reverses the log1p transformation to return predictions to the original scale.

Pipeline: Chains multiple steps into a single workflow, ensuring consistent application of transformations during training and testing.

Steps:

- ("preprocessing", column_transform): The first step applies a preprocessing transformation, likely a ColumnTransformer (aliased as column_transform). This is typically used to handle different types of features (e.g., scaling numeric columns, encoding categorical columns). The exact definition of column_transform isn't shown but would be defined earlier in the code.
- ("regression", gbr_model_transformed): The second step applies the gbr_model_transformed model (the TransformedTargetRegressor with a GradientBoostingRegressor) to make predictions.

6.6.4 prediction and results

```
from sklearn.metrics import mean_absolute_error, r2_score

best_pipeline = random_search.best_estimator_

# # Fit on training set
best_pipeline.fit(X_train, y_train)

# Predict on test set
y_pred = best_pipeline.predict(X_test)

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

# Output
print(f"Test MSE: {mse:.4f}")
print(f"Test MAE: {mae:.4f}")
print(f"Test R²: {r2:.4f}")
```

```
C:\Users\yousi\PycharmProjects\AppRating\.venv\Lib\site-packages\spacy\pipel
  warnings.warn(Warnings.W108)
C:\Users\yousi\PycharmProjects\AppRating\.venv\Lib\site-packages\spacy\pipel
  warnings.warn(Warnings.W108)
Test MSE: 0.1967
Test MAE: 0.3227
Test R²: 0.1719
```

mean_absolute_error: Calculates the Mean Absolute Error (MAE), the average of absolute differences between predicted and actual values. It's a robust metric for regression, less sensitive to outliers than MSE.

r2_score: Calculates the R^2 score (coefficient of determination), which measures the proportion of variance in the target variable explained by the model. R^2 ranges from 0 to 1 (sometimes negative if the model is worse than a constant mean predictor), with higher values indicating better fit.

random_search.best_estimator_: After RandomizedSearchCV completes, this attribute holds the best model (pipeline) with the optimized hyperparameters that yielded the lowest RMSE during cross-validation.

best_pipeline: Stores this optimized pipeline, which includes the preprocessing steps and the tuned GradientBoostingRegressor wrapped in TransformedTargetRegressor.

mse: Calculates the Mean Squared Error (MSE) on the test set, the average of squared differences between actual (y_test) and predicted (y_pred) values. Lower MSE indicates better performance.

mae: Calculates the Mean Absolute Error (MAE), the average of absolute differences. MAE is more interpretable than MSE as it's in the same units as the target variable and less sensitive to large errors.

r2: Calculates the R^2 score, showing how well the model explains the variance in the test set. An R^2 of 1 means perfect prediction, while 0 means the model is no better than predicting the mean of y_test.

7 Conclusion

This project successfully developed a machine learning workflow to predict mobile app ratings, achieving robust performance through a combination of sophisticated preprocessing and ensemble modeling. The retrained stacked ensemble, integrating LightGBM, Random Forest, and XGBoost with a RidgeCV final estimator, delivered the best results, with an MSE of 0.0742, MAE of 0.1933, and R^2 of 0.6878. This indicates that the model explains nearly 69% of the variance in app ratings, with predictions deviating by approximately 0.19 points on average from actual ratings. The preprocessing pipeline, implemented across utils.py, preprocessing.py, and preprocess_app_name.py, played a pivotal role in handling the dataset's complexity, which included text, categorical, numerical, and temporal features.

The workflow's modularity—separating utility functions, preprocessing pipelines, and text processing—ensured flexibility and reproducibility. The use of SpaCy for app name lemmatization, TF-IDF for text feature extraction, and custom parsing functions for numerical and categorical data addressed the dataset's heterogeneity effectively. The stacked ensemble's superior performance highlights the power of combining diverse algorithms to capture different aspects of the data, with retraining on the full dataset enhancing generalization. This approach not only achieved high predictive accuracy but also demonstrated the importance of comprehensive data preparation and model integration in tackling real-world machine learning challenges.