Machine Learning Notes

Weslley Matheus

Contents

Hands on Machine Learning

Machine Learning project with Python

Full Code: Machine Learning project with Python

Listing 1: Housing Project: Linear Regression of the median house value prices. Computes prediction values for the median house value prices based on the label column.

```
# Python STL
import os
import tarfile
import urllib.request
import shutil
import math

# Packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Test Datasets
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit

# Tranformation Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelBinarizer
from sklearn.preprocessing import StandardScaler
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.pipeline import Pipeline
from sklearn.pipeline import FeatureUnion

# ML Algorithms
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestRegressor

# ML Models
import joblib
from sklearn.model_selection import GridSearchCV

# Paths
FILE_DIR = os.path.dirname(os.path.abspath(__file__))
PARENT_FILE_DIR = os.path.dirname(FILE_DIR)
PARENT_DIR = os.path.dirname(PARENT_FILE_DIR)
# Datasets
DATA_PATH = os.path.join(PARENT_DIR, "datasets")
HOUSING_DATA_PATH = os.path.join(DATA_PATH, "housing_data")
```

```
# Models
MODEL_DIR = os.path.join(PARENT_DIR, "models")
HOUSING_MODEL_DIR = os.path.join(MODEL_DIR, "housing_models")
## Images
IMAGES_DIR = os.path.join(PARENT_DIR, "img")
HOUSING_IMAGES_DIR = os.path.join(IMAGES_DIR, "housing_img")
# Directory Creation
directories = [DATA_PATH, HOUSING_DATA_PATH, MODEL_DIR, HOUSING_MODEL_DIR, IMAGES_DIR,
HOUSING_IMAGES_DIR]
for dir in directories:
    os.makedirs(dir, exist_ok=True)

# Display options
pd.set_option("display.max_columns", None)
pd.set_option("display.width", shutil.get_terminal_size().columns)

# Download or Load the Dataset
def get_data(data_download: bool, data_load: bool) -> pd.read_csv:
    """
␣␣␣␣Downloads␣or␣loads␣the␣housing␣dataset.

␣␣␣␣Parameters:
␣␣␣␣----------
␣␣␣␣data_download␣:␣bool
␣␣␣␣␣␣␣␣If␣True,␣downloads␣the␣dataset␣(housing.tgz)␣and␣extracts␣its␣contents
␣␣␣␣␣␣␣␣to␣the␣'datasets/housing'␣directory.
␣␣␣␣data_load␣:␣bool
␣␣␣␣␣␣␣␣If␣True,␣loads␣the␣housing␣dataset␣(housing.csv)␣from␣the␣local
␣␣␣␣␣␣␣␣directory␣into␣a␣pandas␣DataFrame.

␣␣␣␣Returns:
␣␣␣␣-------
␣␣␣␣pd.DataFrame
␣␣␣␣␣␣␣␣The␣loaded␣housing␣dataset␣if␣`data_load`␣is␣True.␣Otherwise,
␣␣␣␣␣␣␣␣no␣return␣value␣is␣provided.
␣␣␣␣"""

    DATA_URL = "https://raw.githubusercontent.com/ageron/handson-ml/master/datasets/
    housing/housing.tgz"
    ZIP_FILE_PATH = os.path.join(HOUSING_DATA_PATH, "housing.tgz")
    EXTRACTED_PATH = os.path.join(HOUSING_DATA_PATH, "housing")

    if (data_download):
        urllib.request.urlretrieve(DATA_URL, ZIP_FILE_PATH)
        shutil.unpack_archive(ZIP_FILE_PATH, EXTRACTED_PATH)
        # Traverse and move files to the extracted_path directory
        for root, dirs, files in os.walk(EXTRACTED_PATH, topdown=False):

            # Move files to extracted_path
            for file in files:
                src_file_path = os.path.join(root, file)
                dest_file_path = os.path.join(EXTRACTED_PATH, file)
                shutil.move(src_file_path, dest_file_path)

            # Remove empty directories
            if root != EXTRACTED_PATH:
                if not os.listdir(root):
                    os.rmdir(root)

        print(f"\nDataset␣downloaded␣and␣extracted␣to:␣{EXTRACTED_PATH}.\n")

    if (data_load):
        csv_path = os.path.join(EXTRACTED_PATH, "housing.csv")
        print(f"\nDataset␣loaded␣from␣the␣files␣inside:␣{EXTRACTED_PATH}.\n")
        return pd.read_csv(csv_path)

# Transform the dataset into a dataframe
def transform_dataframe(
    dataset_transformed: np.ndarray,
    dataset_numerical: pd.DataFrame = None,
    dataset_text: pd.DataFrame = None,
    new_numerical_columns: list = None,
    new_text_columns: list = None
) -> pd.DataFrame:
    """
␣␣␣␣Transforms␣a␣dataset␣into␣a␣DataFrame␣with␣specified␣numerical␣and␣text␣column␣names.

␣␣␣␣Parameters:
␣␣␣␣␣␣␣␣dataset_transformed:␣A␣numpy␣array␣of␣the␣transformed␣dataset␣to␣be␣converted␣to␣a␣DataFrame.
␣␣␣␣␣␣␣␣dataset_numerical:␣A␣DataFrame␣containing␣numerical␣columns␣from␣the␣dataset.
␣␣␣␣␣␣␣␣dataset_text:␣A␣Dataframe␣containing␣text␣columns␣from␣the␣dataset␣for␣text␣encoding.
␣␣␣␣␣␣␣␣new_numerical_columns:␣A␣list␣of␣additional␣numerical␣column␣names␣created␣during␣the␣transformation␣pipeline.
␣␣␣␣␣␣␣␣new_text_columns:␣A␣list␣of␣additional␣text␣column␣names.

␣␣␣␣Returns:
␣␣␣␣␣␣␣␣A␣DataFrame␣with␣the␣specified␣column␣names.
␣␣␣␣"""

    full_columns = []

    if (dataset_numerical is not None):
        numerical_columns = list(dataset_numerical.columns)
        full_columns += numerical_columns
    if (new_numerical_columns is not None):
        full_columns += new_numerical_columns
    if (dataset_text is not None):
        text_encoder = LabelBinarizer()
        text_encoder.fit_transform(dataset_text)
        text_categories = [str(text_category) for text_category in text_encoder.
        classes_]
        full_columns += text_categories
    if (new_text_columns is not None):
        full_columns += new_text_columns

    dataframe_transformed = pd.DataFrame(dataset_transformed, columns=full_columns)

    return dataframe_transformed

# Stratification
def stratify_dataset(dataset: pd.DataFrame) -> tuple:
    """
␣␣␣␣Stratifies␣the␣dataset␣based␣on␣the␣`median_income`␣column.

␣␣␣␣This␣function␣ensures␣the␣dataset␣is␣split␣into␣stratified␣training
␣␣␣␣and␣test␣sets,␣preserving␣the␣distribution␣of␣the␣`median_income`
␣␣␣␣column␣across␣both␣splits.

␣␣␣␣Parameters:
␣␣␣␣----------
␣␣␣␣dataset␣:␣pd.DataFrame
␣␣␣␣␣␣␣␣The␣input␣dataset␣containing␣a␣`median_income`␣column.

␣␣␣␣Returns:
␣␣␣␣-------
␣␣␣␣tuple
␣␣␣␣␣␣␣␣A␣tuple␣containing␣two␣pandas␣DataFrames:
␣␣␣␣␣␣␣␣-␣The␣stratified␣training␣set.
␣␣␣␣␣␣␣␣-␣The␣stratified␣test␣set.

␣␣␣␣Explanation:
␣␣␣␣----------
␣␣␣␣1.␣Add␣Stratification␣Column:
␣␣␣␣␣␣␣A␣new␣column,␣`income-category`,␣is␣added␣to␣the␣dataset.␣This␣column
␣␣␣␣␣␣␣is␣derived␣by␣scaling␣down,␣dividing␣by␣1.5,␣`median_income`␣values␣(which␣
represent␣decimal␣values␣in␣the␣order␣of␣$10,000)
␣␣␣␣␣␣␣and␣capping␣them␣at␣a␣maximum␣value␣of␣5.␣This␣categorization␣creates␣five␣
distinct␣income␣ranges␣for␣stratification.

␣␣␣␣2.␣Stratification␣Algorithm:
␣␣␣␣␣␣␣The␣`StratifiedShuffleSplit`␣algorithm␣is␣used␣to␣split␣the␣dataset
␣␣␣␣␣␣␣into␣training␣and␣test␣sets,␣ensuring␣the␣`income-category`␣distribution
␣␣␣␣␣␣␣is␣preserved␣in␣both␣splits.␣The␣test␣set␣is␣set␣to␣20%␣of␣the␣data.
```

```
␣␣␣␣3.␣Clean␣Up:
␣␣␣␣␣␣␣The␣temporary␣`income-category`␣column,␣added␣for␣stratification,␣is
␣␣␣␣␣␣␣removed␣from␣the␣original␣dataset␣as␣well␣as␣the␣resulting␣training
␣␣␣␣␣␣␣and␣test␣sets.
␣␣␣␣"""
    # 1: Add Stratification Column
    dataset["income-category"] = np.ceil(dataset["median_income"] / 1.5)
    dataset["income-category"].where(dataset["income-category"] < 5, 5.0)

    # 2: Stratification Algorithm
    split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
    for train_index, test_index in split.split(dataset, dataset["income-category"]):
        dataset_stratified_train = dataset.loc[train_index]
        dataset_stratified_test = dataset.loc[test_index]

    # 3: Clean Up
    for set_ in (dataset, dataset_stratified_train, dataset_stratified_test):
        set_.drop("income-category", axis=1, inplace=True)

    return dataset_stratified_train, dataset_stratified_test

# Transformation Pipeline
def transformation_pipeline(
    numerical_columns: list,
    text_columns: list,
    text_categories: list
) -> Pipeline:
    """
␣␣␣␣Creates␣a␣transformation␣pipeline␣for␣preprocessing␣numerical␣and␣text␣data.

␣␣␣␣This␣pipeline␣includes␣the␣following␣steps:
␣␣␣␣1.␣Selection␣of␣specified␣numerical␣and␣text␣columns.
␣␣␣␣2.␣Imputation␣of␣missing␣numerical␣values␣with␣the␣median.
␣␣␣␣3.␣Creation␣of␣new␣numerical␣columns:␣`rooms␣per␣household`␣and␣`
population␣per␣household`.
␣␣␣␣4.␣Standardization␣of␣numerical␣columns.
␣␣␣␣5.␣Binarization␣of␣text␣columns␣into␣binary␣vectors␣based␣on␣specified␣categories.

␣␣␣␣Parameters:
␣␣␣␣----------
␣␣␣␣numerical_columns␣:␣list
␣␣␣␣␣␣␣␣A␣list␣of␣column␣names␣corresponding␣to␣numerical␣features␣in␣the␣dataset.
␣␣␣␣text_columns␣:␣list
␣␣␣␣␣␣␣␣A␣list␣of␣column␣names␣corresponding␣to␣text␣features␣in␣the␣dataset.
␣␣␣␣text_categories␣:␣list
␣␣␣␣␣␣␣␣A␣list␣of␣categories␣used␣for␣encoding␣the␣text␣columns.␣If␣`None`,␣categories␣
are␣inferred␣from␣the␣dataset.

␣␣␣␣Returns:
```

```
␣␣␣␣-------
␣␣␣␣Pipeline
␣␣␣␣␣␣␣␣A␣complete␣data␣transformation␣pipeline␣that␣processes␣both␣numerical␣and␣text␣
data,␣combining
␣␣␣␣␣␣␣␣them␣into␣a␣single␣unified␣transformation␣using␣`FeatureUnion`.
␣␣␣␣"""

    # Selector: select the numerical and text columns from a dataset
    class CustomDataFrameSelector(BaseEstimator, TransformerMixin):
        def __init__(self, columns):
            self.columns = columns
        def fit(self, dataset, dataset_label = None):
            return self
        def transform(self, dataset, dataset_label = None):
            return dataset[self.columns].values

    # Text Encoder: convert text columns into binary vectors
    class CustomLabelBinarizer(BaseEstimator, TransformerMixin):
        def __init__(self, categories = None):
            self.categories = categories
            self.label_binarizer = None

        def fit(self, dataset, dataset_label = None):
            if self.categories is not None:
                self.label_binarizer = LabelBinarizer()
                self.label_binarizer.fit(self.categories)
            else:
                self.label_binarizer = LabelBinarizer()
                self.label_binarizer.fit(dataset)
            return self

        def transform(self, dataset, dataset_label = None):
            return self.label_binarizer.transform(dataset)

    # Custom Transformer Class: group of functions to modify the dataset
    class CustomTransformer(BaseEstimator, TransformerMixin):

        def __init__(self):
            pass

        def fit(self, dataset, dataset_label = None):
            return self

        # Creation of new columns
        def transform(self, dataset, dataset_label = None):

            rooms_index, population_index, households_index = 3, 5, 6
            rooms_per_household = dataset[:, rooms_index] / dataset[:, households_index
            ]
            population_per_household = dataset[:, population_index] / dataset[:,
            households_index]

            return np.c_[dataset, rooms_per_household, population_per_household]

    # Tranformation Pipeline
    numerical_pipeline = Pipeline([
        ("selector", CustomDataFrameSelector(numerical_columns)),
        ("imputer", SimpleImputer(strategy="median")),
        ("transformer", CustomTransformer()),
        ("std_scaler", StandardScaler())
    ])

    text_pipeline = Pipeline([
        ("selector", CustomDataFrameSelector(text_columns)),
        ("label_binarizer", CustomLabelBinarizer(text_categories))
    ])

    full_pipeline = FeatureUnion(transformer_list=[
        ("numerical_pipeline", numerical_pipeline),
        ("text_pipeline", text_pipeline)
    ])

    return full_pipeline

def display_scores(scores: np.ndarray):
    """
␣␣␣␣Display␣the␣evaluation␣metrics␣for␣a␣model’s␣predictions.

␣␣␣␣Prints␣the␣following␣statistics:
␣␣␣␣1.␣Root␣Mean␣Squared␣Error␣(RMSE)␣for␣each␣subset␣of␣cross-validation.
␣␣␣␣2.␣Mean␣of␣the␣RMSEs␣across␣all␣subsets.
␣␣␣␣3.␣Standard␣deviation␣of␣the␣RMSEs␣across␣all␣subsets.

␣␣␣␣Parameters:
␣␣␣␣----------
␣␣␣␣scores␣:␣array-like
␣␣␣␣␣␣␣␣An␣array␣of␣RMSE␣values␣obtained␣from␣cross-validation.
␣␣␣␣"""
    print("RMSE␣for␣each␣subset:", scores)
    print("Mean␣of␣the␣RMSEs:", scores.mean())
    print("Standard␣deviation␣of␣the␣RMSEs:", scores.std())

def train_models(
    dataset_transformed: np.ndarray,
    dataset_labels: pd.DataFrame,
    save_models: bool
):
    """
    Train, evaluate, and optionally save or load regression models for the housing 
dataset.

    Trains the following models:
    1. Linear Regression
    2. Decision Tree
    3. Random Forest

    For each model:
    - Trains on the transformed dataset.
    - Computes RMSE on the training set.
    - Evaluates performance using 10-fold cross-validation and computes RMSE for each 
subset.
    - Saves the trained models to disk if `save_models` is True, or loads pre-trained 
models if False.

    Parameters:
    ----------
    dataset_transformed : numpy.ndarray
        The preprocessed and transformed feature dataset.
    dataset_labels : numpy.ndarray
        The target labels corresponding to the dataset.
    save_models : bool
        A flag indicating whether to save the trained models to disk (`True`) or load 
pre-trained models from disk (`False`).
    """

    if (save_models == True):

        # Linear Regression
        lin_reg.fit(dataset_transformed, dataset_labels)
        lin_reg_path = os.path.join(HOUSING_MODEL_DIR, "linear_regression_model.pkl")
        joblib.dump(lin_reg, lin_reg_path)
        housing_predictions_lin = lin_reg.predict(dataset_transformed)
        lin_rmse = np.sqrt(mean_squared_error(dataset_labels, housing_predictions_lin))
        lin_scores = cross_val_score(lin_reg, dataset_transformed, dataset_labels,
        scoring="neg_mean_squared_error", cv=10)
        lin_scores_rmse = np.sqrt(-lin_scores)

        # Decision Tree
        tree_reg.fit(dataset_transformed, dataset_labels)
        tree_reg_path = os.path.join(HOUSING_MODEL_DIR, "decision_tree_model.pkl")
        joblib.dump(tree_reg, tree_reg_path)
        housing_predictions_tree = tree_reg.predict(dataset_transformed)
        tree_rmse = np.sqrt(mean_squared_error(dataset_labels, housing_predictions_tree
        ))
        tree_scores = cross_val_score(tree_reg, dataset_transformed, dataset_labels,
        scoring="neg_mean_squared_error", cv=10)
        tree_scores_rmse = np.sqrt(-tree_scores)
        # Random Forest
        forest_reg.fit(dataset_transformed, dataset_labels)
        forest_reg_path = os.path.join(HOUSING_MODEL_DIR, "random_forest_model.pkl")
        joblib.dump(forest_reg, forest_reg_path)
        housing_predictions_forest = forest_reg.predict(dataset_transformed)
        forest_rmse = np.sqrt(mean_squared_error(dataset_labels,
        housing_predictions_forest))
        forest_scores = cross_val_score(forest_reg, dataset_transformed, dataset_labels
        , scoring="neg_mean_squared_error", cv=10)
        forest_scores_rmse = np.sqrt(-forest_scores)
    else:
        lin_reg_path = os.path.join(HOUSING_MODEL_DIR, "linear_regression_model.pkl")
        tree_reg_path = os.path.join(HOUSING_MODEL_DIR, "decision_tree_model.pkl")
        forest_reg_path = os.path.join(HOUSING_MODEL_DIR, "random_forest_model.pkl")

        # Load pre-trained models
        lin_reg_loaded = joblib.load(lin_reg_path)
        tree_reg_loaded = joblib.load(tree_reg_path)
        forest_reg_loaded = joblib.load(forest_reg_path)

        housing_predictions_lin = lin_reg_loaded.predict(dataset_transformed)
        housing_predictions_tree = tree_reg_loaded.predict(dataset_transformed)
        housing_predictions_forest = forest_reg_loaded.predict(dataset_transformed)

        lin_rmse = np.sqrt(mean_squared_error(dataset_labels, housing_predictions_lin))
        lin_scores = cross_val_score(lin_reg_loaded, dataset_transformed,
        dataset_labels, scoring="neg_mean_squared_error", cv=10)
        lin_scores_rmse = np.sqrt(-lin_scores)

        tree_rmse = np.sqrt(mean_squared_error(dataset_labels, housing_predictions_tree
        ))
        tree_scores = cross_val_score(tree_reg_loaded, dataset_transformed,
        dataset_labels, scoring="neg_mean_squared_error", cv=10)
        tree_scores_rmse = np.sqrt(-tree_scores)

        forest_rmse = np.sqrt(mean_squared_error(dataset_labels,
        housing_predictions_forest))
        forest_scores = cross_val_score(forest_reg_loaded, dataset_transformed,
        dataset_labels, scoring="neg_mean_squared_error", cv=10)
        forest_scores_rmse = np.sqrt(-forest_scores)

    print("Linear␣Regression␣(RMSE␣of␣the␣transformed␣stratified␣dataset):␣", lin_rmse)
    print("Linear␣Regression␣(RMSEs␣for␣10␣subsets):")
    display_scores(lin_scores_rmse)
    print("\n")

    print("Decision␣Tree␣(RMSE␣of␣the␣transformed␣stratified␣dataset):␣", tree_rmse)
    print("Decision␣Tree␣(RMSEs␣for␣10␣subsets):")
```

```
    display_scores(tree_scores_rmse)
    print("\n")

    print("Random␣Forest␣(RMSE␣of␣the␣transformed␣stratified␣dataset):␣", forest_rmse)
    print("Random␣Forest␣(RMSEs␣for␣10␣subsets):")
    display_scores(forest_scores_rmse)
    print("\n")

def fine_tune_model(
    dataset_transformed: np.ndarray,
    dataset_labels: pd.DataFrame,
    dataset_numerical: pd.DataFrame,
    dataset_text: pd.DataFrame,
    model: BaseEstimator,
    save_model: bool,
    model_name="model"
) -> BaseEstimator:
    """
␣␣␣␣Fine-tune␣a␣given␣machine␣learning␣model␣by␣performing␣hyperparameter␣tuning␣with␣
GridSearchCV.

␣␣␣␣This␣function␣searches␣for␣the␣best␣parameters␣for␣the␣model␣using␣cross-validation
␣and␣grid␣search,
␣␣␣␣evaluates␣the␣model␣performance␣on␣the␣training␣data,␣and␣calculates␣the␣feature␣
importance.␣It␣also␣tests
␣␣␣␣the␣model␣on␣a␣separate␣test␣dataset␣and␣saves␣the␣fine-tuned␣model␣if␣requested.

␣␣␣␣Parameters:
␣␣␣␣----------
␣␣␣␣dataset_transformed␣:␣np.ndarray
␣␣␣␣␣␣␣␣The␣transformed␣training␣dataset,␣with␣features␣ready␣for␣model␣training.

␣␣␣␣dataset_labels␣:␣pd.DataFrame
␣␣␣␣␣␣␣␣The␣target␣labels␣corresponding␣to␣the␣dataset,␣containing␣the␣true␣values␣to␣
predict.

␣␣␣␣dataset_numerical␣:␣pd.DataFrame
␣␣␣␣␣␣␣␣A␣DataFrame␣containing␣the␣numerical␣columns␣used␣to␣calculate␣feature␣
importances.

␣␣␣␣dataset_text␣:␣pd.DataFrame
␣␣␣␣␣␣␣␣A␣DataFrame␣containing␣the␣text␣columns␣used␣for␣binary␣encoding␣in␣the␣model.

␣␣␣␣model␣:␣BaseEstimator
␣␣␣␣␣␣␣␣The␣machine␣learning␣model␣to␣be␣fine-tuned.␣This␣model␣must␣implement␣the␣`fit
()`␣and␣`predict()`␣methods
␣␣␣␣␣␣␣␣from␣scikit-learn's␣`BaseEstimator`.

␣␣␣␣save_model␣:␣bool
```

```
␣␣␣␣␣␣␣␣If␣True,␣the␣fine-tuned␣model␣will␣be␣saved␣to␣a␣file.

␣␣␣␣model_name␣:␣str,␣optional␣(default="model")
␣␣␣␣␣␣␣␣The␣base␣name␣to␣be␣used␣for␣saving␣the␣fine-tuned␣model.

␣␣␣␣Returns:
␣␣␣␣-------
␣␣␣␣best_model␣:␣BaseEstimator
␣␣␣␣␣␣␣␣The␣fine-tuned␣model␣after␣grid␣search,␣ready␣for␣predictions.

␣␣␣␣Notes:
␣␣␣␣-----
␣␣␣␣-␣This␣function␣uses␣GridSearchCV␣to␣find␣the␣best␣combination␣of␣hyperparameters␣
for␣the␣given␣model.
␣␣␣␣-␣The␣model's␣feature␣importances␣are␣printed,␣showing␣the␣contribution␣of␣each␣
column␣to␣the␣column␣of␣predictions.
␣␣␣␣-␣If␣`save_model`␣is␣set␣to␣True,␣the␣fine-tuned␣model␣is␣saved␣as␣a␣`.pkl`␣file␣in
␣the␣`models/housing`␣directory.
␣␣␣␣"""

    # Search for the best parameters for the model: minimum RMSE and best performance
    across the subsets
    param_grid = [
        {"n_estimators":[3,10,30], "max_features":[2,4,6,8]},
        {"bootstrap":[False], "n_estimators":[3,10], "max_features":[2,3,4]}
    ]
    grid_search = GridSearchCV(model, param_grid, cv=5, scoring="neg_mean_squared_error
    ")
    grid_search.fit(dataset_transformed, dataset_labels)

    best_parameters = grid_search.best_params_
    best_model = grid_search.best_estimator_
    best_model_predictions = best_model.predict(dataset_transformed)
    best_model_rmse = np.sqrt(mean_squared_error(dataset_labels, best_model_predictions
    ))

    cv_results = grid_search.cv_results_
    print("Mean␣of␣the␣RMSEs␣of␣the␣subsets␣for␣each␣parameter␣combination:")
    for mean_score, params in zip(cv_results["mean_test_score"], cv_results["params"]):
        print("Mean␣of␣the␣RMSEs:␣", np.sqrt(-mean_score), "for␣parameters:␣", params)

    print("Best␣parameters␣for␣the␣model:␣", best_parameters)
    print("Best␣model␣RMSE:␣", best_model_rmse)

    # Calculates the importance of each column and text category in the formation of
    the prediction column
    # Retrieve the feature importances (column weights) and combine with column names
    column_weights = [float(weight) for weight in best_model.feature_importances_]
    numerical_columns = list(dataset_numerical)
    new_columns = ["rooms_per_household", "population_per_household"]
    text_encoder = LabelBinarizer()
    text_encoder.fit_transform(dataset_text)
    text_categories = [str(text_category) for text_category in text_encoder.classes_]
    full_columns = numerical_columns + new_columns + text_categories
    sorted_column_weights = sorted(zip(column_weights, full_columns), reverse=True)
    print("Column␣Weights␣(Sorted␣by␣Importance):")
    print(sorted_column_weights)

    if (save_model):
        model_full_name = model_name + "_fine_tuned_model" + ".pkl"
        fine_tuned_model_path = os.path.join(HOUSING_MODEL_DIR, model_full_name)
        joblib.dump(best_model, fine_tuned_model_path)
        print("Fine-tuned␣model␣saved␣at:", fine_tuned_model_path)

    return best_model

def prediction_columns(dataset_transformed: np.ndarray, dataset_labels: pd.DataFrame):
    """
␣␣␣␣Generate␣and␣display␣predictions␣from␣multiple␣regression␣models␣(Linear␣Regression
,␣Decision␣Tree,
␣␣␣␣and␣Random␣Forest)␣on␣a␣given␣dataset,␣and␣print␣the␣predictions␣alongside␣the␣true
␣labels.

␣␣␣␣This␣function␣trains␣three␣models␣(Linear␣Regression,␣Decision␣Tree,␣and␣Random␣
Forest)␣on␣the␣provided
␣␣␣␣dataset,␣makes␣predictions␣on␣the␣same␣dataset,␣and␣prints␣the␣first␣five␣predicted
␣values␣for␣each␣model
␣␣␣␣and␣the␣true␣labels␣for␣comparison.

␣␣␣␣Parameters:
␣␣␣␣----------
␣␣␣␣dataset_transformed␣:␣np.ndarray
␣␣␣␣␣␣␣␣The␣transformed␣dataset,␣where␣the␣features␣are␣ready␣for␣prediction.

␣␣␣␣dataset_labels␣:␣pd.DataFrame
␣␣␣␣␣␣␣␣The␣target␣labels␣corresponding␣to␣the␣dataset,␣containing␣the␣true␣values␣to␣
predict.

␣␣␣␣Returns:
␣␣␣␣---------
␣␣␣␣␣␣␣␣This␣function␣prints␣the␣first␣five␣predictions␣from␣each␣model␣and␣the␣true␣
labels␣to␣the␣console.
␣␣␣␣"""

    lin_reg.fit(dataset_transformed, dataset_labels)
    tree_reg.fit(dataset_transformed, dataset_labels)
    forest_reg.fit(dataset_transformed, dataset_labels)
    dataset_predictions_lin = lin_reg.predict(dataset_transformed)
    dataset_predictions_tree = tree_reg.predict(dataset_transformed)
    dataset_predictions_forest = forest_reg.predict(dataset_transformed)

    print("Columns␣of␣Predictions␣(Linear␣Regression␣x␣Decision␣Tree␣x␣Random␣Forest):"
    )
    print(dataset_predictions_lin[0:5], "\n")
    print(dataset_predictions_tree[0:5], "\n")
    print(dataset_predictions_forest[0:5], "\n")
    print("Column␣of␣Labels:")
    print(dataset_labels.iloc[0:5].values, "\n")

if __name__ == "__main__":

    # Download and Load Data
    data = get_data(data_download = True, data_load = True)

    if data is not None and not data.empty:
        # Housing Dataset
        housing = data.copy()
        housing_stratified_train, housing_stratified_test = stratify_dataset(housing)
        housing_train = housing_stratified_train.drop(columns=["median_house_value"])
        housing_train_labels = housing_stratified_train["median_house_value"].copy()
        housing_test = housing_stratified_test.drop(columns=["median_house_value"])
        housing_test_labels = housing_stratified_test["median_house_value"].copy()
        # Housing Columns
        housing_train_numerical = housing_train.drop(columns=["ocean_proximity"])
        housing_numerical_columns = list(housing_train.drop(columns=["ocean_proximity"
        ]))
        housing_train_text = housing_train["ocean_proximity"]
        housing_text_columns = ["ocean_proximity"]
        housing_text_categories = ['<1H␣OCEAN', 'INLAND', 'ISLAND', 'NEAR␣BAY', 'NEAR␣
        OCEAN']

        # ML Algorithms
        lin_reg = LinearRegression()
        tree_reg = DecisionTreeRegressor()
        forest_reg = RandomForestRegressor()

        full_pipeline = transformation_pipeline(housing_numerical_columns,
        housing_text_columns, housing_text_categories)
        full_pipeline.fit(housing_train)
        housing_train_transformed = full_pipeline.transform(housing_train)

        print("The␣dataframe␣of␣the␣housing␣dataset:")
        print(housing_train.iloc[0:1])

        print("\n")
        print("The␣dataframe␣of␣the␣housing␣dataset␣after␣the␣transformation␣pipeline:"
        )
        housing_train_transformed_df = transform_dataframe(
            dataset_transformed = housing_train_transformed,
            dataset_numerical = housing_train_numerical,
            dataset_text = housing_train_text,
            new_numerical_columns = ["rooms_per_household", "population_per_household"]
        )
        print(housing_train_transformed_df.iloc[0:1])

        print("\n")

        prediction_columns(housing_train_transformed, housing_train_labels)

        print("\n")

        train_models(housing_train_transformed, housing_train_labels, True)

        print("\n")

        best_model_forest_reg = fine_tune_model(housing_train_transformed,
        housing_train_labels, housing_train_numerical, housing_train_text, forest_reg,
        True, "forest_reg")

        print("\n")

        # Test the best model on the test dataset
        housing_test_transformed = full_pipeline.transform(housing_test)
        best_model_predictions = best_model_forest_reg.predict(housing_test_transformed
        )
        best_model_rmse_test = np.sqrt(mean_squared_error(housing_test_labels,
        best_model_predictions))
        print("RMSE␣of␣the␣best␣model␣on␣the␣test␣dataset:␣", best_model_rmse_test)
```

Machine Learning Project: Food Dataset from the U.S Department of Agriculture (USDA)

Listing 2: Food Dataset Project from the USDA: Linear Regression on the Protein Column. The code predicts protein values for each food item based on the label column.

```
# Python STL
import os
import sys
import tarfile
import urllib.request
import shutil
import typing
import math

# Packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Test Datasets
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit

# Tranformation Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelBinarizer
from sklearn.preprocessing import StandardScaler
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.pipeline import Pipeline
from sklearn.pipeline import FeatureUnion

# ML Algorithms
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestRegressor

# ML Models
import joblib
from sklearn.model_selection import GridSearchCV

# Modules
# Dataframe PD Display options
pd.set_option("display.max_columns", None)
pd.set_option("display.max_rows", None)
pd.set_option("display.max_colwidth", None)
pd.set_option("display.width", shutil.get_terminal_size().columns)
# Array NP Display Options
np.set_printoptions(threshold=np.inf)



# Paths
FILE_DIR = os.path.dirname(os.path.abspath(__file__))
PARENT_FILE_DIR = os.path.dirname(FILE_DIR)
PARENT_DIR = os.path.dirname(PARENT_FILE_DIR)
## Datasets
DATA_PATH = os.path.join(PARENT_DIR, "datasets")
FOOD_DATA_PATH = os.path.join(DATA_PATH, "food_data")
## Models
MODEL_DIR = os.path.join(PARENT_DIR, "models")
FOOD_MODEL_DIR = os.path.join(MODEL_DIR, "food_models")
## Images
IMAGES_DIR = os.path.join(PARENT_DIR, "img")
FOOD_IMAGES_DIR = os.path.join(IMAGES_DIR, "food_img")
GRAPHS_IMAGES_DIR = os.path.join(FOOD_IMAGES_DIR, "graphs")

directories = [DATA_PATH, FOOD_DATA_PATH, MODEL_DIR, FOOD_MODEL_DIR, IMAGES_DIR,
FOOD_IMAGES_DIR, GRAPHS_IMAGES_DIR]
for dir in directories:
    os.makedirs(dir, exist_ok=True)



# Download and Load the Dataset
def get_data(data_download: bool = False, data_load: bool = False) -> pd.read_csv:
    """
␣␣␣␣Downloads␣and␣loads␣the␣USDA␣Food␣Data␣Central␣dataset.

␣␣␣␣This␣function␣provides␣two␣functionalities:
␣␣␣␣1.␣Downloads␣and␣extracts␣the␣USDA␣Food␣Data␣Central␣dataset␣from␣the␣official␣URL.
␣␣␣␣2.␣Loads␣the␣dataset␣into␣pandas␣DataFrames␣if␣the␣files␣have␣already␣been␣
downloaded.

␣␣␣␣Parameters:
␣␣␣␣----------
␣␣␣␣data_download␣:␣bool,␣optional
```

```
␣␣␣␣␣␣␣␣If␣set␣to␣True,␣downloads␣the␣dataset␣from␣the␣USDA␣Food␣Data␣Central␣website␣
and␣extracts
␣␣␣␣␣␣␣␣the␣files␣to␣the␣specified␣directory.␣Defaults␣to␣False.

␣␣␣␣data_load␣:␣bool,␣optional
␣␣␣␣␣␣␣␣If␣set␣to␣True,␣loads␣the␣extracted␣dataset␣files␣(e.g.,␣`food.csv`,␣`
food_category.csv`,
␣␣␣␣␣␣␣␣`food_nutrient.csv`,␣and␣`nutrient.csv`)␣into␣pandas␣DataFrames.␣Defaults␣to␣
False.

␣␣␣␣Returns:
␣␣␣␣-------
␣␣␣␣A␣tuple␣of␣DataFrames␣containing␣the␣following␣data:
␣␣␣␣␣␣␣␣-␣`food_csv`:␣Information␣about␣food␣items.
␣␣␣␣␣␣␣␣-␣`food_category_csv`:␣Information␣about␣food␣categories.
␣␣␣␣␣␣␣␣-␣`food_nutrient_csv`:␣Nutritional␣details␣associated␣with␣food␣items.
␣␣␣␣␣␣␣␣-␣`nutrient_csv`:␣Information␣about␣specific␣nutrients.
␣␣␣␣"""

    DATA_URL = "https://fdc.nal.usda.gov/fdc-datasets/
    FoodData_Central_foundation_food_csv_2024-10-31.zip"
    ZIP_FILE_PATH = os.path.join(FOOD_DATA_PATH, "food_data_files.zip")
    EXTRACTED_PATH = os.path.join(FOOD_DATA_PATH, "food_data_files")

    if (data_download):

        urllib.request.urlretrieve(DATA_URL, ZIP_FILE_PATH)
        shutil.unpack_archive(ZIP_FILE_PATH, EXTRACTED_PATH)

        for root, dirs, files in os.walk(EXTRACTED_PATH, topdown=False):

            # Move files to extracted_path
            for file in files:
                src_file_path = os.path.join(root, file)
                dest_file_path = os.path.join(EXTRACTED_PATH, file)
                shutil.move(src_file_path, dest_file_path)

            # Remove empty directories
            if root != EXTRACTED_PATH:
                if not os.listdir(root):
                    os.rmdir(root)

        print(f"\nDataset␣downloaded␣and␣extracted␣to:␣{EXTRACTED_PATH}.\n")

    if (data_load):

        FOOD_PATH = os.path.join(EXTRACTED_PATH, "food.csv")
        FOOD_CATEGORY_PATH = os.path.join(EXTRACTED_PATH, "food_category.csv")
        FOOD_NUTRIENT_PATH = os.path.join(EXTRACTED_PATH, "food_nutrient.csv")
        NUTRIENT_PATH = os.path.join(EXTRACTED_PATH, "nutrient.csv")

        food_csv = pd.read_csv(FOOD_PATH)
        food_category_csv = pd.read_csv(FOOD_CATEGORY_PATH)
        food_nutrient_csv = pd.read_csv(FOOD_NUTRIENT_PATH, dtype={"footnote": "str"})
        nutrient_csv = pd.read_csv(NUTRIENT_PATH)

        print(f"Dataset␣loaded␣from␣the␣files␣inside:␣{EXTRACTED_PATH}.\n")

        return food_csv, food_category_csv, food_nutrient_csv, nutrient_csv




# Rename Columns
def rename_columns(df: pd.DataFrame, column_suffix_name: str) -> pd.DataFrame:
    """
␣␣␣␣Renames␣the␣columns␣of␣a␣DataFrame␣by␣appending␣a␣given␣suffix␣to␣each␣column␣name.

␣␣␣␣Parameters:
␣␣␣␣␣␣␣␣dataframe␣(pd.DataFrame):␣Input␣DataFrame␣whose␣columns␣need␣renaming.
␣␣␣␣␣␣␣␣column_suffix_name␣(str):␣Suffix␣to␣append␣to␣each␣column␣name.

␣␣␣␣Returns:
␣␣␣␣␣␣␣␣pd.DataFrame:␣DataFrame␣with␣renamed␣columns.

␣␣␣␣Raises:
␣␣␣␣␣␣␣␣TypeError:␣If␣input␣is␣not␣a␣pandas␣DataFrame.
␣␣␣␣"""
    if not isinstance(df, pd.DataFrame):
        raise TypeError("The␣input␣must␣be␣a␣pandas␣DataFrame.")

    return df.rename(columns={col: col + column_suffix_name for col in df.columns})




# Merge DataFrames
def merge_dataframes(df_left: pd.DataFrame, df_right: pd.DataFrame, key_left: str,
key_right: str, how: str = "left") -> pd.DataFrame:
    """
␣␣␣␣Merges␣two␣DataFrames␣on␣specified␣keys␣with␣a␣given␣merge␣method.

␣␣␣␣Parameters:
␣␣␣␣␣␣␣␣df_left␣(pd.DataFrame):␣The␣left␣DataFrame␣to␣be␣merged.
␣␣␣␣␣␣␣␣df_right␣(pd.DataFrame):␣The␣right␣DataFrame␣to␣be␣merged.
␣␣␣␣␣␣␣␣key_left␣(str):␣The␣column␣name␣in␣the␣left␣DataFrame␣to␣use␣as␣the␣merge␣key.
␣␣␣␣␣␣␣␣key_right␣(str):␣The␣column␣name␣in␣the␣right␣DataFrame␣to␣use␣as␣the␣merge␣key
.
```

```
␣␣␣␣␣␣␣␣how␣(str,␣optional):␣The␣type␣of␣merge␣to␣perform.␣Default␣is␣"left".␣Other␣
options␣are␣"inner",␣"outer",␣and␣"right".

␣␣␣␣Returns:
␣␣␣␣␣␣␣␣pd.DataFrame:␣The␣merged␣DataFrame.

␣␣␣␣Raises:
␣␣␣␣␣␣␣␣TypeError:␣If␣either␣input␣is␣not␣a␣pandas␣DataFrame.
␣␣␣␣␣␣␣␣ValueError:␣If␣the␣specified␣keys␣do␣not␣exist␣in␣the␣respective␣DataFrames.
␣␣␣␣"""
    if not isinstance(df_left, pd.DataFrame) or not isinstance(df_right, pd.DataFrame):
        raise TypeError("Both␣inputs␣must␣be␣pandas␣DataFrames.")

    if key_left not in df_left.columns or key_right not in df_right.columns:
        raise ValueError(f"Specified␣keys␣'{key_left}'␣or␣'{key_right}'␣not␣found␣in␣
        the␣respective␣DataFrames.")

    return df_left.merge(df_right, left_on=key_left, right_on=key_right, how=how)




# Data Cleaning: Reshape Dataset
def reshape_dataset(food_dataset: pd.DataFrame, save_dataset: bool = False) -> pd.
DataFrame:

    """
␣␣␣␣Processes␣and␣reshapes␣a␣DataFrame␣containing␣food␣nutrient␣information.

␣␣␣␣This␣function␣filters␣the␣dataset␣to␣retain␣only␣specific␣nutrients␣of␣interest␣and
␣reshapes
␣␣␣␣it␣into␣a␣table␣where␣rows␣correspond␣to␣food␣categories␣and␣items,␣and␣columns␣
represent␣the
␣␣␣␣selected␣nutrient␣amounts.␣The␣resulting␣dataset␣is␣cleaned,␣formatted,␣and␣saved␣
as␣a␣CSV␣file
␣␣␣␣for␣further␣analysis.␣The␣function␣returns␣the␣reshaped␣DataFrame.

␣␣␣␣Parameters:
␣␣␣␣-----------
␣␣␣␣food_dataset␣:␣pd.DataFrame
␣␣␣␣␣␣␣␣A␣DataFrame␣containing␣detailed␣information␣about␣food␣items,␣including␣their␣
categories,
␣␣␣␣␣␣␣␣descriptions,␣nutrient␣names,␣and␣nutrient␣amounts.

␣␣␣␣Returns:
␣␣␣␣--------
␣␣␣␣pd.DataFrame
␣␣␣␣␣␣␣␣A␣reshaped␣and␣cleaned␣DataFrame␣where:
␣␣␣␣␣␣␣␣-␣Rows␣represent␣food␣categories␣and␣specific␣food␣items.
```

```
␣␣␣␣␣␣␣␣-␣Columns␣represent␣selected␣nutrient␣amounts␣(e.g.,␣proteins,␣carbohydrates,␣
fats,␣energy).
␣␣␣␣␣␣␣␣-␣Rows␣with␣missing␣values␣for␣key␣nutrients␣are␣removed.
␣␣␣␣␣␣␣␣-␣The␣index␣is␣reset␣to␣sequential␣integers␣starting␣from␣0.
␣␣␣␣"""

    # Create the Nurtients Column
    relevant_nutrients = [
        "Protein", "Carbohydrate,␣by␣difference", "Total␣lipid␣(fat)", "Energy␣(Atwater
        ␣General␣Factors)",
        "Water", "Nitrogen"
    ]

    # Get all unique nutrient names dynamically
    all_nutrients = food_dataset["name_nutrient"].unique()

    # Filter the dataset based on all nutrients (noisy) or only on the relevant
    nutrients
    nutrients = food_dataset[food_dataset["name_nutrient"].isin(relevant_nutrients)]

    # Drop columns where all values are missing (Nan)
    nutrients = nutrients.dropna(axis=1, how="all")

    food_dataset_reshaped = nutrients.pivot_table(
        index=["description_food_category", "description_food"],
        columns="name_nutrient",
        values="amount_food_nutrient",
        aggfunc="mean"
    ).reset_index()

    # Rename column names
    food_dataset_reshaped.columns.name = None
    food_dataset_reshaped.rename(
        columns={
            "description_food_category": "Category",
            "description_food": "Food",
            "Protein": "Proteins␣(g)",
            "Carbohydrate,␣by␣difference": "Carbohydrates␣(g)",
            "Total␣lipid␣(fat)": "Fats␣(g)",
            "Energy␣(Atwater␣General␣Factors)": "Energy␣(kcal)",
            "Water": "Water␣(g)",
            "Nitrogen": "Nitrogen␣(g)"
        },
        inplace=True
    )

    # Remove rows where any relevant nutrient value is Nan
    columns_to_check = list(food_dataset_reshaped.drop(columns={"Category", "Food"}))
    food_dataset_reshaped = food_dataset_reshaped.dropna(subset=columns_to_check)
    # Resets the index of rows to 0, 1, 2, 3, ...
    food_dataset_reshaped.reset_index(drop=True, inplace=True)

    # Convert the dataset to a CSV file
    if (save_dataset):
        food_dataset_reshaped_path = os.path.join(FOOD_DATA_PATH, "
        food_dataset_reshaped.csv")
        food_dataset_reshaped.to_csv(food_dataset_reshaped_path, index=False)
        print(f"\nFile␣converted␣to␣food_dataset_reshaped.csv␣and␣saved␣to␣{
        food_dataset_reshaped_path}.\n")

    return food_dataset_reshaped




# Transformation Pipeline
def transformation_pipeline(columns: list) -> Pipeline:
    """
␣␣␣␣The␣output␣or␣return␣value␣of␣a␣transformer␣becomes␣the␣input␣for␣the␣next␣
transformer.
␣␣␣␣"""

    # Selector: select the nutrients columns
    class CustomDataFrameSelector(BaseEstimator, TransformerMixin):

        def __init__(self, columns):
            self.columns = columns

        def fit(self, dataset, dataset_label = None):
            # self.columns = list(dataset.drop(columns=["Category", "Food", "Proteins (
            g)"]).columns)
            return self

        def transform(self, dataset, dataset_label = None):
            return dataset[self.columns]

    # Tranformation Pipeline
    full_pipeline = Pipeline([
        ("selector", CustomDataFrameSelector(columns)),  # Selects nutrient columns
        ("imputer", SimpleImputer(strategy="mean")),  # Handles missing values
        ("std_scaler", StandardScaler())  # Standardizes the columns
    ])

    return full_pipeline
```

```
# Standardized Columns
def standardization_column(dataset: pd.DataFrame, dataset_numerical_columns: list):
    """
␣␣␣␣Check␣the␣Standardization␣of␣the␣Columns␣transformed␣in␣the␣Transformation␣Pipeline
.
␣␣␣␣"""
    print("\nStandardization␣of␣Columns␣(Mean␣=␣0,␣Standard␣Deviation␣=␣1):")
    for col in list(dataset_numerical_columns):
        print(f"Mean␣of␣{col}:␣", dataset[col].mean())
        print(f"Standard␣Deviation␣of␣{col}:␣", dataset[col].std(), "\n")




# Train Model
def train_models(
    dataset_transformed: np.ndarray,
    dataset_labels: pd.DataFrame,
    save_models: bool
):
    """
␣␣␣␣Train,␣evaluate,␣and␣optionally␣save␣or␣load␣regression␣models␣for␣the␣dataset.

␣␣␣␣Trains␣the␣following␣models:
␣␣␣␣1.␣Linear␣Regression
␣␣␣␣2.␣Decision␣Tree
␣␣␣␣3.␣Random␣Forest

␣␣␣␣For␣each␣model:
␣␣␣␣-␣Trains␣on␣the␣transformed␣dataset.
␣␣␣␣-␣Computes␣RMSE␣on␣the␣training␣set.
␣␣␣␣-␣Evaluates␣performance␣using␣10-fold␣cross-validation␣and␣computes␣RMSE␣for␣each␣
subset.
␣␣␣␣-␣Saves␣the␣trained␣models␣to␣disk␣if␣`save_models`␣is␣True,␣or␣loads␣pre-trained␣
models␣if␣False.

␣␣␣␣Parameters:
␣␣␣␣----------
␣␣␣␣dataset_transformed␣:␣numpy.ndarray
␣␣␣␣␣␣␣␣The␣preprocessed␣and␣transformed␣feature␣dataset.
␣␣␣␣dataset_labels␣:␣numpy.ndarray
␣␣␣␣␣␣␣␣The␣target␣labels␣corresponding␣to␣the␣dataset.
␣␣␣␣save_models␣:␣bool
␣␣␣␣␣␣␣␣A␣flag␣indicating␣whether␣to␣save␣the␣trained␣models␣to␣disk␣(`True`)␣or␣load␣
pre-trained␣models␣from␣disk␣(`False`).
␣␣␣␣"""

    if (save_models == True):
        # Linear Regression
        lin_reg.fit(dataset_transformed, dataset_labels)
        lin_reg_path = os.path.join(FOOD_MODEL_DIR, "linear_regression_model.pkl")
        joblib.dump(lin_reg, lin_reg_path)
        dataset_predictions_lin = lin_reg.predict(dataset_transformed)
        lin_rmse = np.sqrt(mean_squared_error(dataset_labels, dataset_predictions_lin))
        lin_scores = cross_val_score(lin_reg, dataset_transformed, dataset_labels,
        scoring="neg_mean_squared_error", cv=10)
        lin_scores_rmse = np.sqrt(-lin_scores)

        # Decision Tree
        tree_reg.fit(dataset_transformed, dataset_labels)
        tree_reg_path = os.path.join(FOOD_MODEL_DIR, "decision_tree_model.pkl")
        joblib.dump(tree_reg, tree_reg_path)
        dataset_predictions_tree = tree_reg.predict(dataset_transformed)
        tree_rmse = np.sqrt(mean_squared_error(dataset_labels, dataset_predictions_tree
        ))
        tree_scores = cross_val_score(tree_reg, dataset_transformed, dataset_labels,
        scoring="neg_mean_squared_error", cv=10)
        tree_scores_rmse = np.sqrt(-tree_scores)

        # Random Forest
        forest_reg.fit(dataset_transformed, dataset_labels)
        forest_reg_path = os.path.join(FOOD_MODEL_DIR, "random_forest_model.pkl")
        joblib.dump(forest_reg, forest_reg_path)
        dataset_predictions_forest = forest_reg.predict(dataset_transformed)
        forest_rmse = np.sqrt(mean_squared_error(dataset_labels,
        dataset_predictions_forest))
        forest_scores = cross_val_score(forest_reg, dataset_transformed, dataset_labels
        , scoring="neg_mean_squared_error", cv=10)
        forest_scores_rmse = np.sqrt(-forest_scores)
    else:
        lin_reg_path = os.path.join(FOOD_MODEL_DIR, "linear_regression_model.pkl")
        tree_reg_path = os.path.join(FOOD_MODEL_DIR, "decision_tree_model.pkl")
        forest_reg_path = os.path.join(FOOD_MODEL_DIR, "random_forest_model.pkl")

        # Load pre-trained models
        lin_reg_loaded = joblib.load(lin_reg_path)
        tree_reg_loaded = joblib.load(tree_reg_path)
        forest_reg_loaded = joblib.load(forest_reg_path)

        dataset_predictions_lin = lin_reg_loaded.predict(dataset_transformed)
        dataset_predictions_tree = tree_reg_loaded.predict(dataset_transformed)
        dataset_predictions_forest = forest_reg_loaded.predict(dataset_transformed)

        lin_rmse = np.sqrt(mean_squared_error(dataset_labels, dataset_predictions_lin))
        lin_scores = cross_val_score(lin_reg_loaded, dataset_transformed,
        dataset_labels, scoring="neg_mean_squared_error", cv=10)
        lin_scores_rmse = np.sqrt(-lin_scores)
        tree_rmse = np.sqrt(mean_squared_error(dataset_labels, dataset_predictions_tree
        ))
        tree_scores = cross_val_score(tree_reg_loaded, dataset_transformed,
        dataset_labels, scoring="neg_mean_squared_error", cv=10)
        tree_scores_rmse = np.sqrt(-tree_scores)

        forest_rmse = np.sqrt(mean_squared_error(dataset_labels,
        dataset_predictions_forest))
        forest_scores = cross_val_score(forest_reg_loaded, dataset_transformed,
        dataset_labels, scoring="neg_mean_squared_error", cv=10)
        forest_scores_rmse = np.sqrt(-forest_scores)

    print("Linear␣Regression␣(RMSE␣of␣the␣transformed␣dataset):␣", lin_rmse)
    print("Linear␣Regression␣(RMSEs␣for␣10␣subsets):")
    print("RMSE␣for␣each␣subset:", lin_scores_rmse)
    print("Mean␣of␣the␣RMSEs:", lin_scores_rmse.mean())
    print("Standard␣deviation␣of␣the␣RMSEs:", lin_scores_rmse.std())
    print("\n")

    print("Decision␣Tree␣(RMSE␣of␣the␣transformed␣dataset):␣", tree_rmse)
    print("Decision␣Tree␣(RMSEs␣for␣10␣subsets):")
    print("RMSE␣for␣each␣subset:", tree_scores_rmse)
    print("Mean␣of␣the␣RMSEs:", tree_scores_rmse.mean())
    print("Standard␣deviation␣of␣the␣RMSEs:", tree_scores_rmse.std())
    print("\n")

    print("Random␣Forest␣(RMSE␣of␣the␣transformed␣dataset):␣", forest_rmse)
    print("Random␣Forest␣(RMSEs␣for␣10␣subsets):")
    print("RMSE␣for␣each␣subset:", forest_scores_rmse)
    print("Mean␣of␣the␣RMSEs:", forest_scores_rmse.mean())
    print("Standard␣deviation␣of␣the␣RMSEs:", forest_scores_rmse.std())
    print("\n")

    column_names = ["Linear␣Regression", "Decision␣Tree", "Random␣Forest"]
    RMSE_columns_arrays = np.column_stack([lin_scores_rmse, tree_scores_rmse,
    forest_scores_rmse])
    RMSE_columns = pd.DataFrame(RMSE_columns_arrays, columns=column_names)

    # Graph of RMSE Scores across subsets
    fig, ax = plt.subplots(figsize=(10, 6))
    RMSE_columns.plot(kind='bar', ax=ax)
    ax.set_title("Comparison␣of␣RMSE␣Scores␣Across␣Models", fontsize=14)
    ax.set_xlabel("Subsets␣(Cross-Validation)", fontsize=12)
    ax.set_ylabel("RMSE␣Scores", fontsize=12)
    ax.legend(title="Models", fontsize=12)
    plt.xticks(ticks=range(len(RMSE_columns)), labels=[f"Subset␣{i+1}" for i in range(
    len(RMSE_columns))], rotation=0)
    plt.savefig(os.path.join(GRAPHS_IMAGES_DIR, "RMSE_scores.png"))
```

```
# Prediction Columns
def prediction_columns(dataset_transformed: np.ndarray, dataset_labels: pd.DataFrame):
    """
␣␣␣␣Generate␣and␣display␣predictions␣from␣multiple␣regression␣models␣(Linear␣Regression
,␣Decision␣Tree,
␣␣␣␣and␣Random␣Forest)␣on␣a␣given␣dataset,␣and␣print␣the␣predictions␣alongside␣the␣true
␣labels.

␣␣␣␣This␣function␣trains␣three␣models␣(Linear␣Regression,␣Decision␣Tree,␣and␣Random␣
Forest)␣on␣the␣provided
␣␣␣␣dataset,␣makes␣predictions␣on␣the␣same␣dataset,␣and␣prints␣the␣first␣five␣predicted
␣values␣for␣each␣model
␣␣␣␣and␣the␣true␣labels␣for␣comparison.

␣␣␣␣Parameters:
␣␣␣␣----------
␣␣␣␣dataset_transformed␣:␣np.ndarray
␣␣␣␣␣␣␣␣The␣transformed␣dataset,␣where␣the␣features␣are␣ready␣for␣prediction.

␣␣␣␣dataset_labels␣:␣pd.DataFrame
␣␣␣␣␣␣␣␣The␣target␣labels␣corresponding␣to␣the␣dataset,␣containing␣the␣true␣values␣to␣
predict.

␣␣␣␣Returns:
␣␣␣␣---------
␣␣␣␣␣␣␣␣This␣function␣prints␣the␣first␣five␣predictions␣from␣each␣model␣and␣the␣true␣
labels␣to␣the␣console.
␣␣␣␣"""

    lin_reg.fit(dataset_transformed, dataset_labels)
    tree_reg.fit(dataset_transformed, dataset_labels)
    forest_reg.fit(dataset_transformed, dataset_labels)

    dataset_predictions_lin = lin_reg.predict(dataset_transformed)
    dataset_predictions_tree = tree_reg.predict(dataset_transformed)
    dataset_predictions_forest = forest_reg.predict(dataset_transformed)

    column_names = ["Linear␣Regression", "Decision␣Tree", "Random␣Forest", "Label␣
    Column"]
    prediction_columns_arrays = np.column_stack([dataset_predictions_lin,
    dataset_predictions_tree, dataset_predictions_forest, dataset_labels])
    prediction_columns = pd.DataFrame(prediction_columns_arrays, columns=column_names)

    print("Prediction␣Columns:")
    print(prediction_columns.iloc[0:5])
    # Save the prediction columns dataframe as a csv file
    prediction_columns_path = os.path.join(FOOD_DATA_PATH, "prediction_columns.csv")
    prediction_columns.to_csv(prediction_columns_path, index=False)
    print(f"\nPrediction␣columns␣converted␣to␣prediction_columns.csv␣and␣saved␣to␣{
    prediction_columns_path}.\n")

    # Scatter plot: prediction columns vs label column
    plt.figure(figsize=(12,8))
    plt.scatter(prediction_columns["Label␣Column"], prediction_columns["Linear␣
    Regression"],color="blue",alpha=0.6,label="Linear␣Regression")
    plt.scatter(prediction_columns["Label␣Column"], prediction_columns["Decision␣Tree"
    ],color="green",alpha=0.6,label="Decision␣Tree")
    plt.scatter(prediction_columns["Label␣Column"], prediction_columns["Random␣Forest"
    ],color="orange",alpha=0.6,label="Random␣Forest")
    plt.plot(prediction_columns["Label␣Column"],prediction_columns["Label␣Column"],
    color="red",alpha=0.2,linestyle="--",label="Label␣Column")
    plt.title("Scatter␣Plot␣of␣Model␣Predictions␣vs␣Label␣Column", fontsize=14)
    plt.xlabel("Label␣Column␣Values", fontsize=12)
    plt.ylabel("Predicted␣Values", fontsize=12)
    plt.legend(fontsize=12)
    plt.grid(True, linestyle="--", alpha=0.5)
    plt.savefig(os.path.join(GRAPHS_IMAGES_DIR, "scatter_plot_prediction_columns.png"))




# Fine Tune Model
def fine_tune_model(
    dataset_transformed: np.ndarray,
    dataset_labels: pd.DataFrame,
    model: BaseEstimator,
    save_model: bool,
    model_name = "model",
) -> BaseEstimator:
    """
␣␣␣␣Fine-tune␣a␣given␣machine␣learning␣model␣by␣performing␣hyperparameter␣tuning␣with␣
GridSearchCV.

␣␣␣␣This␣function␣searches␣for␣the␣best␣parameters␣for␣the␣model␣using␣cross-validation
␣and␣grid␣search,
␣␣␣␣evaluates␣the␣model␣performance␣on␣the␣training␣data,␣and␣calculates␣the␣feature␣
importance.␣It␣also␣tests
␣␣␣␣the␣model␣on␣a␣separate␣test␣dataset␣and␣saves␣the␣fine-tuned␣model␣if␣requested.

␣␣␣␣Parameters:
␣␣␣␣----------
␣␣␣␣dataset_transformed␣:␣np.ndarray
␣␣␣␣␣␣␣␣The␣transformed␣training␣dataset,␣with␣features␣ready␣for␣model␣training.
```

```
␣␣␣␣dataset_labels␣:␣pd.DataFrame
␣␣␣␣␣␣␣␣The␣target␣labels␣corresponding␣to␣the␣dataset,␣containing␣the␣true␣values␣to␣
predict.

␣␣␣␣model␣:␣BaseEstimator
␣␣␣␣␣␣␣␣The␣machine␣learning␣model␣to␣be␣fine-tuned.␣This␣model␣must␣implement␣the␣`fit
()`␣and␣`predict()`␣methods
␣␣␣␣␣␣␣␣from␣scikit-learn's␣`BaseEstimator`.

␣␣␣␣save_model␣:␣bool
␣␣␣␣␣␣␣␣If␣True,␣the␣fine-tuned␣model␣will␣be␣saved␣to␣a␣file.

␣␣␣␣model_name␣:␣str,␣optional␣(default="model")
␣␣␣␣␣␣␣␣The␣base␣name␣to␣be␣used␣for␣saving␣the␣fine-tuned␣model.

␣␣␣␣Returns:
␣␣␣␣-------
␣␣␣␣best_model␣:␣BaseEstimator
␣␣␣␣␣␣␣␣The␣fine-tuned␣model␣after␣grid␣search,␣ready␣for␣predictions.

␣␣␣␣Notes:
␣␣␣␣-----
␣␣␣␣-␣This␣function␣uses␣GridSearchCV␣to␣find␣the␣best␣combination␣of␣hyperparameters␣
for␣the␣given␣model.
␣␣␣␣-␣The␣model's␣feature␣importances␣are␣printed,␣showing␣the␣contribution␣of␣each␣
column␣to␣the␣column␣of␣predictions.
␣␣␣␣-␣If␣`save_model`␣is␣set␣to␣True,␣the␣fine-tuned␣model␣is␣saved␣as␣a␣`.pkl`␣file␣in
␣the␣`models/housing`␣directory.
␣␣␣␣"""

    # Search for the best parameters for the model: minimum RMSE and best performance
    across the subsets
    param_grid = [
        {"n_estimators":[3,10,30], "max_features":[2,4,6,8]},
        {"bootstrap":[False], "n_estimators":[3,10], "max_features":[2,3,4]}
    ]
    grid_search = GridSearchCV(model, param_grid, cv=5, scoring="neg_mean_squared_error
    ")
    grid_search.fit(dataset_transformed, dataset_labels)

    best_parameters = grid_search.best_params_
    best_model = grid_search.best_estimator_
    best_model_predictions = best_model.predict(dataset_transformed)
    best_model_rmse = np.sqrt(mean_squared_error(dataset_labels, best_model_predictions
    ))

    cv_results = grid_search.cv_results_
    print("Mean␣of␣the␣RMSEs␣of␣the␣subsets␣for␣each␣parameter␣combination:")
    for mean_score, params in zip(cv_results["mean_test_score"], cv_results["params"]):
        print("Mean␣of␣the␣RMSEs:␣", np.sqrt(-mean_score), "for␣parameters:␣", params)

    print("Best␣parameters␣for␣the␣model:␣", best_parameters)
    print("Best␣model␣RMSE:␣", best_model_rmse)

    if (save_model):
        model_full_name = model_name + "_fine_tuned_model" + ".pkl"
        fine_tuned_model_path = os.path.join(FOOD_MODEL_DIR, model_full_name)
        joblib.dump(best_model, fine_tuned_model_path)
        print("\nFine-tuned␣model␣saved␣at:", fine_tuned_model_path, "\n")

    return best_model




# Column Weights
def column_weights(
    model: BaseEstimator,
    dataset_transformed: pd.DataFrame = None
):
    """
␣␣␣␣Calculates␣the␣importance␣(weights)␣of␣each␣column␣based␣on␣the␣trained␣model’s␣
feature␣importances.

␣␣␣␣Parameters
␣␣␣␣----------
␣␣␣␣model␣:␣BaseEstimator
␣␣␣␣␣␣␣␣A␣trained␣machine␣learning␣model␣with␣a␣`feature_importances_`␣attribute.
␣␣␣␣dataset_transformed␣:␣pd.DataFrame,␣optional
␣␣␣␣␣␣␣␣The␣processed␣dataset␣(numerical␣and␣textual␣features)␣transformed␣in␣the␣
transformation␣pipeline␣and␣used␣by␣the␣ML␣algorithms.

␣␣␣␣Returns
␣␣␣␣-------
␣␣␣␣None
␣␣␣␣␣␣␣␣Prints␣the␣feature␣importance␣weights␣and␣their␣corresponding␣columns,␣sorted␣
in␣descending␣order␣of␣importance.

␣␣␣␣Notes
␣␣␣␣-----
␣␣␣␣-␣Ensure␣the␣model␣is␣already␣trained␣and␣includes␣a␣`feature_importances_`␣
attribute.
␣␣␣␣-␣`dataset_processed`␣should␣match␣the␣features␣used␣to␣train␣the␣model.
␣␣␣␣"""
    # Extract feature importance weights from the model
    column_weights = [float(weight) for weight in model.feature_importances_]
    # Get the column names of the transformed dataset
    columns = list(dataset_transformed.columns)

    # Pair and sort columns with their corresponding weights in descending order
    sorted_column_weights = sorted(zip(column_weights, columns), reverse=True)

    print("Feature␣Importances␣(Column␣Weights):")
    print(sorted_column_weights, "\n")

    # Feature Importance Graph
    # Extract feature names and weights
    weights, features = zip(*sorted_column_weights)
    plt.figure(figsize=(10, 6))
    plt.bar(features, weights, color='skyblue', alpha=0.7)
    plt.xticks(rotation=90)  # Rotate feature names for better readability
    plt.xlabel('Features␣(Column␣Names)')
    plt.ylabel('Feature␣Importance␣(Column␣Weights)')
    plt.title('Feature␣Importance␣Distribution')
    plt.savefig(os.path.join(GRAPHS_IMAGES_DIR, "column_weights.png"))




# Linear Correlation
def linear_correlation(dataset_numerical: pd.DataFrame, column_name: str, save_graphs:
bool):

    # Compute the correlation matrix
    corr_matrix = dataset_numerical.corr()
    linear_corr = corr_matrix[column_name].sort_values(ascending=False)

    # Plot scatter plots for each feature
    for col in dataset_numerical:
        if col == column_name:
            continue
        plt.figure(figsize=(8, 6))
        plt.scatter(dataset_numerical[col], dataset_numerical[column_name], alpha=0.5,
        c="gray")
        plt.title(f"{column_name}␣vs.␣{col}")
        plt.xlabel(col)
        plt.ylabel(column_name)
        plt.savefig(os.path.join(GRAPHS_IMAGES_DIR, f"linear_correlation_{column_name}_
        {col}.png"))

    return linear_corr
```

```
if __name__ == "__main__":

    # sys.exit("Sys.exit(): stops the code right here.")

    # Download and load the dataset files as DataFrames
    food_csv, food_category_csv, food_nutrient_csv, nutrient_csv = get_data(
    data_download=True, data_load=True)

    # Add suffixes to column names to indicate their source
    food_csv = rename_columns(food_csv, "_food")
    food_category_csv = rename_columns(food_category_csv, "_food_category")
    food_nutrient_csv = rename_columns(food_nutrient_csv, "_food_nutrient")
    nutrient_csv = rename_columns(nutrient_csv, "_nutrient")

    # Merge DataFrames to combine food, categories, and nutrient information
    food_category_df = merge_dataframes(food_category_csv, food_csv, "id_food_category"
    , "food_category_id_food")
    food_nutrient_df = merge_dataframes(food_nutrient_csv, nutrient_csv, "
    nutrient_id_food_nutrient", "id_nutrient")
    food_merged_df = merge_dataframes(food_category_df, food_nutrient_df, "fdc_id_food"
    , "fdc_id_food_nutrient")

    print("Food␣Dataset␣merged␣dataframes:")
    print(food_merged_df.iloc[0:1], "\n")

    # Reshape the merged dataset: create 'Category', 'Food', and 'Nutrients' columns,
    and remove unnecessary columns
    food_dataset_reshaped = reshape_dataset(food_dataset=food_merged_df, save_dataset=
    True)
    print(f"Food␣Dataset␣reshaped␣{food_dataset_reshaped.shape}:")
    print(food_dataset_reshaped.iloc[0:1], "\n")

    # Split the dataset into label, nutrient, and textual columns
    food_dataset = food_dataset_reshaped.drop(columns=["Proteins␣(g)"])
    food_dataset_labels = food_dataset_reshaped["Proteins␣(g)"].copy()
    food_dataset_nutrients = food_dataset.drop(columns=["Category", "Food"])
    food_dataset_text = food_dataset.drop(columns=food_dataset_nutrients.columns)

    print("Nutrient␣columns␣to␣be␣included␣in␣the␣transformation␣pipeline:")
    print(list(food_dataset_nutrients.columns), "\n")

    # Transformation Pipeline
    full_pipeline = transformation_pipeline(list(food_dataset_nutrients.columns))
    full_pipeline.fit(food_dataset_reshaped)
    food_dataset_nutrients_transformed = full_pipeline.transform(food_dataset_reshaped)

    # Merge the text, transformed nutrient columns, and labels into a single dataFrame
    food_dataset_nutrients_transformed_df = pd.DataFrame(
    food_dataset_nutrients_transformed, columns=food_dataset_nutrients.columns)
```

```
food_dataset_transformed = pd.concat([food_dataset_text,
food_dataset_nutrients_transformed_df, food_dataset_labels], axis=1)

print("Food␣Dataset␣after␣the␣transformation␣pipeline:")
print(food_dataset_transformed.iloc[0:1], "\n")

# Check the standardization of the columns
print("Standardization␣of␣the␣nutrient␣columns␣made␣by␣the␣transformation␣pipeline
:\n")
standardization_column(food_dataset_transformed, list(food_dataset_nutrients.
columns))

# Train the models: use the ML algorithms to create the prediction column

# ML Algorithms
lin_reg = LinearRegression()
tree_reg = DecisionTreeRegressor()
forest_reg = RandomForestRegressor()

print("Root␣Mean␣Square␣Errors␣of␣the␣ML␣Algorithms:\n")
train_models(
    dataset_transformed = food_dataset_nutrients_transformed,
    dataset_labels = food_dataset_labels,
    save_models = True
)

print("Prediction␣Columns␣created␣by␣the␣ML␣Algorithms:\n")
prediction_columns(
        dataset_transformed = food_dataset_nutrients_transformed,
        dataset_labels = food_dataset_labels
    )

print("Results␣of␣the␣best␣model␣(Random␣Forest␣Algorithm):\n")
best_model = fine_tune_model(
    dataset_transformed = food_dataset_nutrients_transformed,
    dataset_labels = food_dataset_labels,
    model = forest_reg,
    save_model = True,
    model_name = "random_forest"
)

print("Column␣Weights␣(Feature␣Importances)␣created␣by␣the␣best␣model␣(Random␣
Forest␣Algorithm):")
column_weights(
    model = best_model,
    dataset_transformed = food_dataset_transformed.drop(columns=["Category", "Food"
    , "Proteins␣(g)"])
)
```

```
    print("Linear␣Correlation␣between␣the␣processed␣nutrient␣columns␣and␣the␣label␣
    column␣(Proteins):")
    linear_corr_protein = linear_correlation(
        dataset_numerical=food_dataset_transformed.drop(columns=["Category", "Food"]),
        column_name="Proteins␣(g)",
        save_graphs=True
    )
    print(linear_corr_protein)
```

Formulas for performance measure

Mean

$$m = \frac{1}{n}\sum_{i=1}^{n} v_i$$

Variance

$$V = \frac{1}{n}\sum_{i=1}^{n}(v_i - m)^2$$

- Deviation from the mean: $v_i - mean$

Standard Deviation

$$SD = \sqrt{V}$$

Root Mean Square Error

$$RMSE(\boldsymbol{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h(x^{(i)}) - y^{(i)}\right)^2}$$

$$RMSE(\boldsymbol{Dataset}, MLAlgorithm) = \sqrt{\frac{1}{rows}\sum_{i=1}^{rows}\left(MLAlgorithm(predicted\,value^{(i)}) - label\,value^{(i)}\right)^2}$$

- Euclidean distance: straight line $d = \sqrt{\Delta x^2 + \Delta y^2}$
- The ML Algorithm takes into consideration all the column values of the dataset to form a column of predicted values.
- The RMSE measures the standard deviation of the predicted values from the label values.

Mean Absolute Error

$$MAE(\boldsymbol{X}, h) = \frac{1}{m} \sum_{i=1}^{m} \mid h(x^{(i)}) - y^{(i)} \mid$$

$$MAE(Dataset, MLAlgorithm) = \frac{1}{rows} \sum_{i=1}^{rows} \mid MLAlgorithm(predictedvalue^{(i)}) - labelvalue^{(i)} \mid$$

- Manhattan distance: grid $d = \mid \mathit{\Delta} x \mid + \mid \mathit{\Delta} y \mid$
- Both the RMSE and the MAE are ways to measure the distance between two vectors: the column of predicted values from the column of label values.
- The mean absolute error is preferred when the data has many outliers.

Difference between RMSE and Standard Deviation:

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^{n} \left(y_{predicted}^{(i)} - y_{label}^{(i)} \right)^2}$$

$$STD = \sqrt{\frac{1}{n} \sum_{i=1}^{n} \left(y_{predicted}^{(i)} - mean \right)^2}$$

- RMSE (Root Mean Squared Error) measures the average magnitude (value) of the differences (errors) between the predicted values and the true values (labels). In other words, it's the average "distance" between the predicted values and the label values. It is the deviation from the label.
- Standard Deviation measures the average distance of the differences between the predicted values from their own mean. It measures how spread out the values (in a dataset) are from the mean value. When applied to predictions, it measures how spread out the predicted values are from their own mean. It is the deviation from the mean.

- i: Instance (Row or Data-points)
- m: Number of instances (Number of rows)
- h: Function that predicts the desired output value (Machine Learning Algorithms: Linear Regression, Decision Tree, Random Forest, ...)
- $h(x^{(i)}) - y^{(i)}$: prediction error of the i^{th}instance. (row i: ML_Algorithm(Value from the Prediction Column) - Value from the Label Column)
- y: Value of the Label Column from the row i (Column Label with the Label Values)

$$y^{(i)} = (median\ house\ value)$$

- **x**: Vector with the column values of the row i (Columns of the Dataset with the Column Values)

$$x^{(i)} = \begin{pmatrix} longitude \\ latitude \\ population \\ median\ income \end{pmatrix}$$

$$(x^{(i)})^T = \begin{pmatrix} longitude & latitude & population & median\ income \end{pmatrix}$$

- **X**: Matrix of the transposed column vectors (Dataset: Columns + Column Label)

$$X = \begin{pmatrix} (x^{(1)})^T & (y^{(1)})^T \\ (x^{(2)})^T & (y^{(2)})^T \\ (x^{(3)})^T & (y^{(3)})^T \\ \vdots & \vdots \\ (x^{(n)})^T & (y^{(n)})^T \end{pmatrix} = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} & y^{(1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

- $x_1^{(1)}$ = value of the column 1 from the row 1
- $y^{(1)}$ = value of the column label from the row 1

Workspace

- Python
- Python modules: NumPy, Pandas, Matplotlib, and Scikit-Learn.

Listing 3: Install python modules and creates the virtual environment

```
:: Install and upgrade pip
pip --version
python -m pip install --upgrade pip

:: Create the virtual environment inside the ML directory
cd %ML%
python -m venv venv

:: Activate the environment
:: Every package will be installed in this environment while it is activated
cd %ML%
.\venv\Scripts\activate

set PATH=%ML%\venv\Scripts;%PATH%
where python
:: -> %ML%\venv\Scripts\python.exe

:: Install packages (env -> lib -> site-packages)
python -m pip install matplotlib numpy pandas scipy scikit-learn joblib
python -c "import matplotlib, numpy, pandas, scipy, sklearn, joblib"

:: Import packages inside a python file
:: On file.py inside VSCode: select the python interpreter from the virtual environment
:: Search >Python: Select Interpreter (C:\...\ML\venv\scripts\python.exe)

:: The virtual environmenet serves to install packages and run python files from there
:: The command "where python" lists the python.exe interpreters, the first one is being
 used

:: Quit the virtual environment
:: The terminal session returns to using the default python.exe from the system-wide
installation
cd %ML%
.\venv\scripts\deactivate
```

Download the Data

Listing 4: Download and load the dataset.

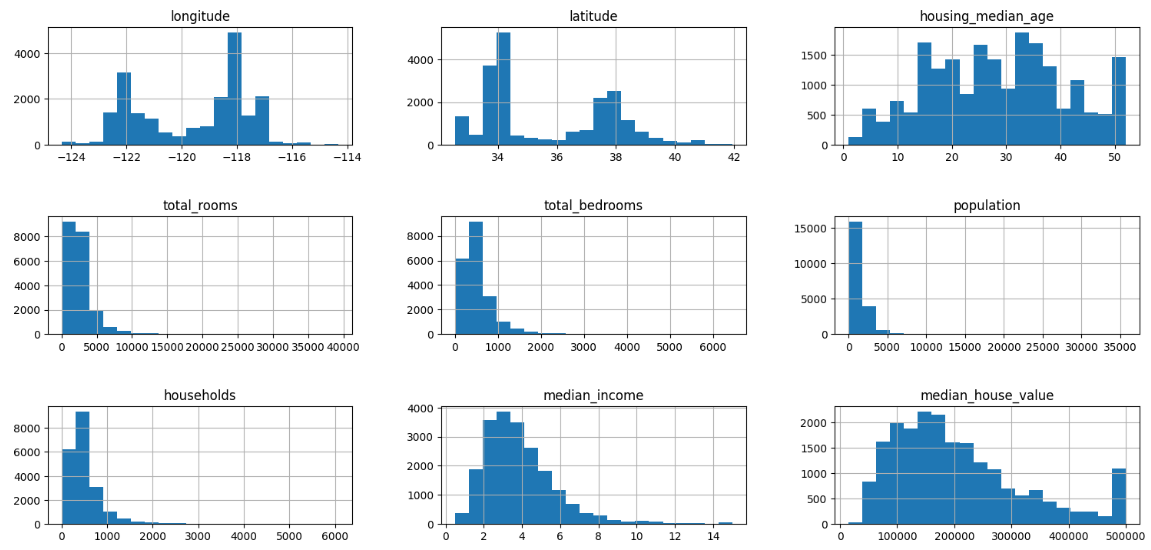
```python
# data_download.py
"""
This␣module␣downloads␣and␣extracts␣the␣dataset.
"""

# Python STL
import os
import tarfile
import urllib.request

# Packages
import pandas as pd

# Data Path
FILE_DIR = os.path.dirname(os.path.abspath(__file__))
PARENT_DIR = os.path.dirname(FILE_DIR)
DATA_PATH = os.path.join(PARENT_DIR, "datasets", "housing")
os.makedirs(DATA_PATH, exist_ok=True)
DATA_URL = "https://raw.githubusercontent.com/ageron/handson-ml/master/datasets/housing
/housing.tgz"

# Download and extract data
def download_data(data_url=DATA_URL, data_path=DATA_PATH):
    if not os.path.isdir(data_path):
        os.makedirs(data_path)
    tgz_path = os.path.join(data_path, "housing.tgz")
    urllib.request.urlretrieve(data_url, tgz_path)
    data_tgz = tarfile.open(tgz_path)
    data_tgz.extractall(path=data_path)
    data_tgz.close()

# Only download data when this file is run directly
if __name__ == "__main__":
    download_data()

# Load the data
def load_data(data_path=DATA_PATH):
    csv_path = os.path.join(data_path, "housing.csv")
    return pd.read_csv(csv_path)

data = load_data()
```

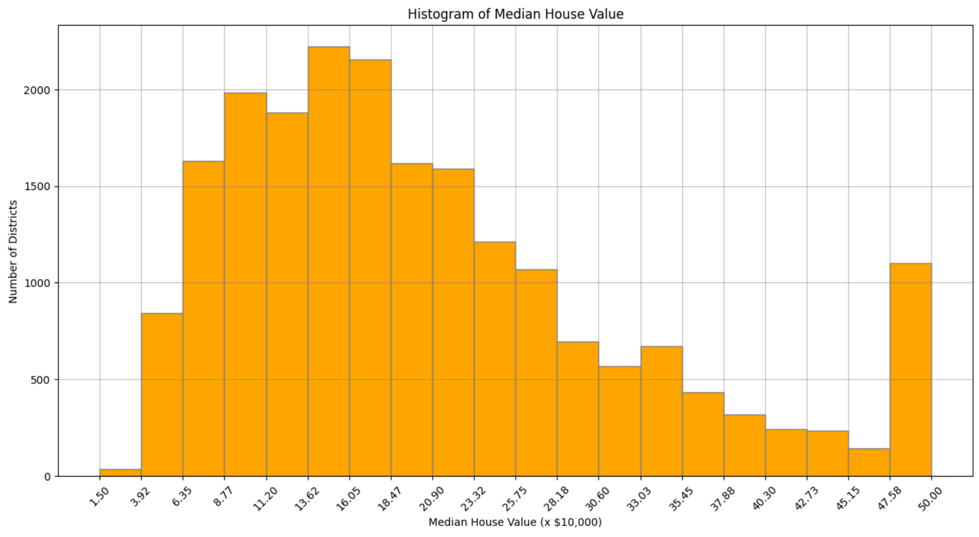
Histogram: Graph of Rows / Data Points vs Columns

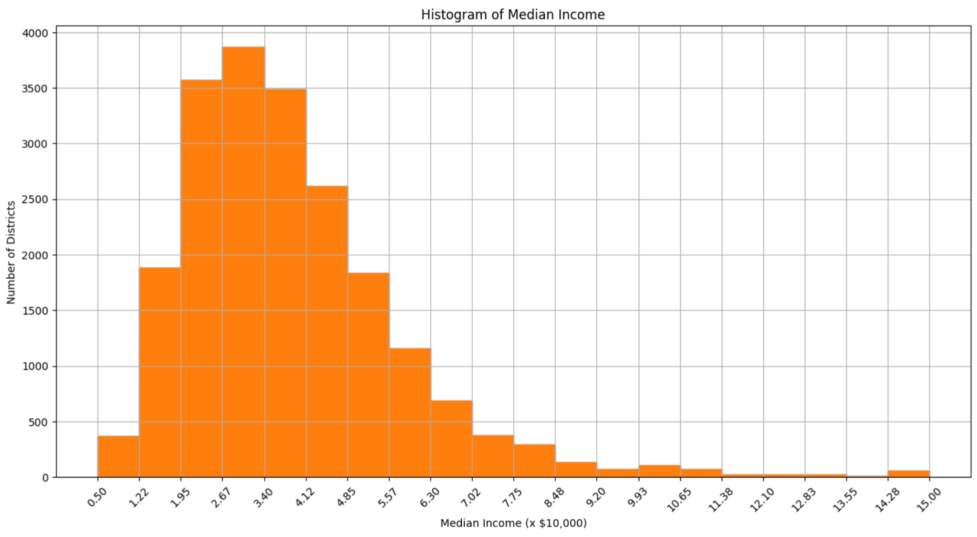
Listing 5: Histogram for the full dataset, random dataset and stratified dataset.

```
# histogram.py
"""
This␣file␣shows␣how␣to␣create␣a␣graph␣of␣the␣data-points␣/␣number␣of␣rows␣(Y)␣and␣the␣
ranges␣of␣values␣from␣a␣column␣(X).
It␣shows␣how␣many␣rows␣have␣values␣from␣a␣specific␣column␣inside␣some␣specific␣range.
"""
import os
import tarfile
import urllib.request
import warnings
warnings.filterwarnings("ignore")
import math
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit

import data_download

data = data_download.data

# Histogram of the Dataset
data.hist(bins=20, figsize=(10, 5))
plt.tight_layout()
plt.show()

# Histogram for the median house value
data_scaled = data["median_house_value"] / 10000
_, bin_edges, _ = plt.hist(data_scaled, bins=20, edgecolor="grey", color="orange")
plt.grid(True, axis='both', linestyle='-', color='grey', alpha=0.5)
plt.xlabel("Median␣House␣Value␣(x␣$10,000)")
plt.ylabel("Number␣of␣Districts")
plt.title("Histogram␣of␣Median␣House␣Value")

# Range labels
plt.xticks(bin_edges, rotation=45)

plt.show()

# Histogram for the median income
data["median_income"].hist(bins=20, figsize=(10, 6), edgecolor="black")
plt.xlabel("Median␣Income␣(x␣$10,000)")
plt.ylabel("Number␣of␣Districts")
plt.title("Histogram␣of␣Median␣Income")
```

```
# Range labels
bin_edges = plt.hist(data["median_income"], bins=20)[1]
plt.xticks(bin_edges, rotation=45)

plt.show()
```







- **Y**: Number of rows or data-points (districts of houses).
- **X**: Columns or categories / attributes (median_income, median_house_value...).
- **Bins**: Number of ranges of column values. x bins = x bars in the graph. 1 bin = 1 range of column values = Δx of 1 bar.
- **Bar**: frequency / number of rows or data-points (Δy) inside 1 (bin) range of column values (Δx).
- **Histogram for Median House Values**: More than 1000 districts (1 district = multiple houses with singular house prices) have median house values between $475,800 and $500,000.
- **Histogram for Median Income**: More than 1000 districts (1 district = multiple houses with singular incomes) have a median income value in the range $55,700 to $63,000.

Analysis of the Median Income vs Districts and Comparison Table: Dataset and Test Datasets (Random vs Stratified)

Listing 6: Comparison Table

```
# comparison_table_analysis.py
"""
This␣file␣shows␣how␣many␣data-points/rows␣(districts)␣pertain␣to␣some␣specific␣range␣of
␣median␣income␣values.
It␣also␣shows␣the␣percentual␣error␣between␣the␣proportions␣of␣data-points/rows␣in␣these
␣specific␣ranges
of␣the␣random␣and␣stratified␣test␣datasets␣in␣relation␣to␣the␣full␣dataset.
"""

import os
import tarfile
import urllib.request
import warnings
warnings.filterwarnings("ignore")
import math
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit

import data_download
import stratification

# References
data = data_download.data
data_stratified_test = stratification.data_stratified_test
data_train, data_random_test = train_test_split(data, test_size=0.2, random_state=42)

# Comparison Table (Dataset vs. Test Datasets)

# Calculate the district percentages for the full dataset, random test dataset, and
stratified test dataset
percentage_data = data["median_income"].value_counts(bins=5, normalize=True).sort_index
()
percentage_random = data_random_test["median_income"].value_counts(bins=5, normalize=
True).sort_index()
percentage_stratified = data_stratified_test["median_income"].value_counts(bins=5,
normalize=True).sort_index()

# Combine the columns into a DataFrame ($ x10,000 vs %)
comparison_table = pd.DataFrame({
    "Dataset␣Proportion": percentage_data,
    "Random␣Proportion": percentage_random,
    "Stratified␣Proportion": percentage_stratified,
})

# Calculate signed percentage errors
comparison_table["Random␣Error␣(%)"] = ((comparison_table["Random␣Proportion"] /
comparison_table["Dataset␣Proportion"]) - 1) * 100
comparison_table["Stratified␣Error␣(%)"] = ((comparison_table["Stratified␣Proportion"]
/ comparison_table["Dataset␣Proportion"]) - 1) * 100

# Round the columns to 2 decimal places
comparison_table = comparison_table.round({"Dataset␣Proportion": 2, "Random␣Proportion"
: 2, "Stratified␣Proportion": 2 })
comparison_table = comparison_table.round({"Random␣Error␣(%)": 2, "Stratified␣Error␣(%)
": 2})

if __name__ == "__main__":

    # Dataset: Median Income x Districts
    print("\nMedian␣Income␣($10,000)␣x␣%␣Districts")
    income_districts = data["median_income"].value_counts(bins=5).sort_index() / len(
    data) * 100
    for income, district in income_districts.items():
        income_range = f"{income.left*10000:.0f}␣-␣{income.right*10000:.0f}"
        print(f"$␣{income_range}:␣{district:.2f}␣%")

    # Dataset x Test Datasets: Median Income

    # Dataset
    print("\nDataset:␣Median␣Income␣(x10,000$)␣x␣Districts")
    print(data["median_income"].value_counts(bins=5)) # bins = 5: divides the data in 5
     income ranges

    # Random Test Dataset
    print("\nRandomized:␣Median␣Income␣(x10,000$)␣x␣Districts")
    print(data_random_test["median_income"].value_counts(bins=5))

    # Stratified Test Dataset
    print("\nStratified:␣Median␣Income␣(x10,000$)␣x␣Districts")
    print(data_stratified_test["median_income"].value_counts(bins=5))

    # Dataset x Random x Stratified: percentual error of data-points in the same median
     income ranges
    print("\nRnadom␣vs␣Stratified␣(error␣%):")
    print(comparison_table)
```

$$Error(\%) = (\frac{Test\ Proportion}{Dataset\ Proportion} - 1) \times 100$$

- Rows = data-points. Columns = attributes or categories.
- Strata: It refers to the ranges of values from the columns of the dataset.
- Dataset-portion or Bar: X % or number of rows are inside the range of values A-B from one column.
- Dataset: The sum of all rows or data points.

Stratification of a Dataset

Listing 7: Data Stratification

```
# stratification.py
"""
This␣module␣constructs␣a␣smaller␣stratified␣test␣dataset␣from␣the␣dataset.
"""

# Python STL
import os
import tarfile
import urllib.request
import math

# Packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Test Datasets
from sklearn.model_selection import StratifiedShuffleSplit

# Modules
import data_download

# References
data = data_download.data

# Stratification

def stratify_dataset(dataset):
    # 1: Add Stratification Column: Stratification column based on a dataset column
    # It divides the median_income column values in 5 ranges according to 5 values on
    the income-category column
    # np.ceil: gives the integer greater than or equal to the result of the division
    # dataset["median_income"] / 1.5: you divide by 1.5 to scale the median_income
    values to a smaller range
    # .where(condition, value_changed): you limit the maximum value of the income-
    category column to 5
    # if the condition is True, the value will remain the same, but if False, the value
     will be changed
    dataset["income-category"] = np.ceil(dataset["median_income"] / 1.5)
    dataset["income-category"].where(dataset["income-category"] < 5, 5.0)

    # 2: Stratification Algorithm
    split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
    for train_index, test_index in split.split(dataset, dataset["income-category"]):
        dataset_stratified_train = dataset.loc[train_index]
        dataset_stratified_test = dataset.loc[test_index]
    # 3: Clean up: Remove stratification column from all the datasets
    for set_ in (dataset, dataset_stratified_train, dataset_stratified_test):
        set_.drop("income-category", axis=1, inplace=True)

    return dataset_stratified_train, dataset_stratified_test

data_stratified_train, data_stratified_test = stratify_dataset(data)

if __name__ == "__main__":
    print("Dataset (rows, columns):", data.shape)
    print("Stratified train dataset:", data_stratified_train.shape)
    print("Stratified test dataset:", data_stratified_test.shape)
```

You create a smaller dataset with the same proportions of the original dataset regarding a category.

- The stratification process starts by creating a column “c-category" based on the values of a column “c” inside the original dataset.
- The column “c-category” holds individual values based on specific ranges (strata) of values of column “c”.
- Finally, based on the frequency / proportion of the values of column "c-category", you add the rows from the Dataset into the Test Dataset.
- If the column “c-category” has a value “x” with frequency 10% corresponding to the range of values (a-b) in column “c”, the algorithm will keep adding rows from the Dataset with values within the range (a-b) of column “c” until they make up for 10% of the size of the Test Dataset.

Note: A stratified test dataset tends to perform better overall because it ensures the test set proportions closely match the full dataset’s proportions. Smaller datasets or smaller strata can lead to high variability in estimates for groups and produce unstable results.

Example:

Let’s say in the original dataset, you have 1,000 rows, and the income-category values, based on the values of the median_income column, distribution is like this:

income-category = 1.0: 100 rows (10% of the dataset) - median_income: 45,000 - 60,000

income-category = 2.0: 300 rows (30% of the dataset) - median_income: 60,000 - 80,000

income-category = 3.0: 400 rows (40% of the dataset) - median_income: 80,000 - 100,000

income-category = 4.0: 100 rows (10% of the dataset) - median_income: 100,000 - 150,000

income-category = 5.0: 100 rows (10% of the dataset) - median_income: 150,000 - 200,000

Now, when you perform the stratified split:

80% of the rows will go into the training set, and 20% will go into the test set (since test size = 0.2).

The algorithm will ensure that 40% of both the training and test sets will come from the income-category = 3.0 category (because 40% of the original dataset is in that category).

So, for the test set:

If 20% of the data goes to the test set, 40% of the test set will be from income-category = 3.0, and the remaining percentages will be distributed based on the original proportions.

Data Visualization: Graph utilizing the values of 4 Columns

Listing 8: Data Visualization

```
# data_visualization.py
"""
This␣file␣shows␣how␣to␣create␣a␣graph␣with␣multiple␣columns␣of␣the␣dataset␣as␣visual␣
elements␣of␣the␣graph.
"""
import os
import tarfile
import urllib.request
import shutil
import warnings
warnings.filterwarnings("ignore")
import math
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit

import data_download
import stratification

data = data_download.data
housing = stratification.data_stratified_train

# Data Visualization
def housing_visualization():
    housing.plot(kind="scatter", x = "longitude", y = "latitude", alpha=0.4,
                 s=housing["population"]/100, label="population",           # s =
                  size of the data-points
                 c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True) # c =
                  color of the data-points
    plt.legend()
    plt.show()

if __name__ == "__main__":
    housing_visualization()
```

- **s:** Controls the size of the data points. `s = housing[“population”]/100` means that the size of each point is proportional to the values of the “population” column divided by 100. Districts (data points) with larger populations will have larger data points.
- **c:** Controls the color of the data points. `c=”median house value”` means that the color of each point will represent the values of the “median_house_value” column, and it’s mapped to a colormap. Districts (data points) with higher median house values will have red colors.

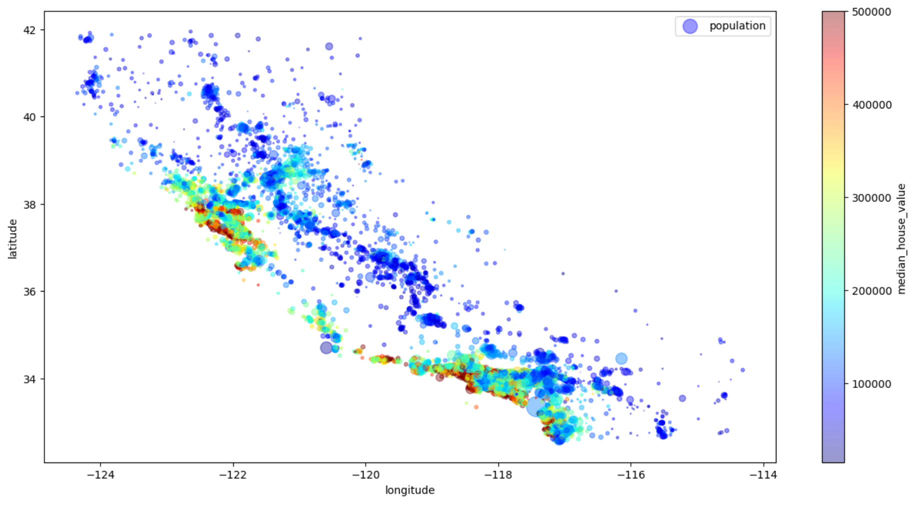


Figure 0.0.1: The housing prices are related to the ocean proximity and population density.

Linear Correlation: Graph x-y of a linear function between 2 Columns

Listing 9: Linear Correlation

```python
# linear_correlation.py
"""
This␣file␣shows␣how␣to␣create␣a␣graph␣with␣the␣linear␣function␣between␣2␣columns.
"""
import os
import tarfile
import urllib.request
import hashlib
import warnings
warnings.filterwarnings("ignore")
import math
import pandas as pd
from pandas.plotting import scatter_matrix
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit

import data_download

# References
data = data_download.data
housing = data.drop(columns=["ocean_proximity"])

# Linear Correlation
# Columns vs Median House Value
corr_matrix = housing.corr()
linear_corr = corr_matrix["median_house_value"].sort_values(ascending=False)
print(linear_corr)

columns = housing.copy()

for col in columns:
    plt.figure(figsize=(8, 6))
    plt.scatter(housing[col], housing["median_house_value"], alpha=0.5, c="gray")
    plt.title(f"{col}␣vs.␣median_house_value")
    plt.xlabel(col)
    plt.ylabel("median_house_value")
    plt.show()
# Median Income vs Median House Value
columns_2 = ["median_house_value", "median_income"]
scatter_matrix(housing[columns_2], figsize=(10,6))
plt.show()
```

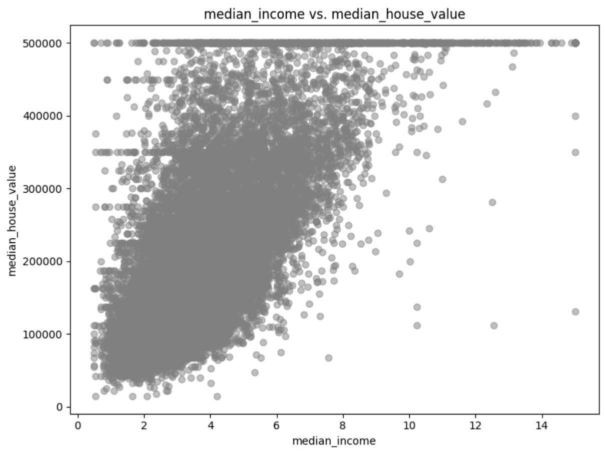


Figure 0.0.2: Median House Value versus Median Income with Linear Correlation = + 0.68

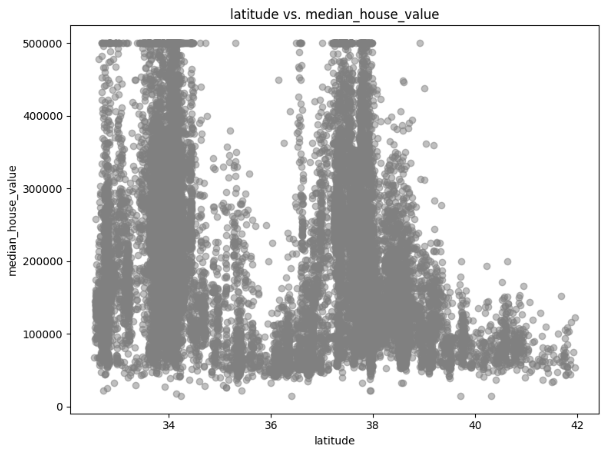


Figure 0.0.3: Median House Value versus Latitude with Linear Correlation = - 0.14

The Linear Correlation coefficient ranges from 1 to -1. A coefficient close to 0 signifies little or no linear correlation, but nonlinear relationships between the attributes might still exist.

- +0.68: the median house value tends to increase when the median income increases.
- -0.14: the median house value has a slight tendency to decrease when the latitude increases.

Column division

Listing 10: Column Division

```python
# column_division.py
"""
This␣file␣shows␣how␣to␣divide␣columns␣to␣create␣new␣columns␣/␣attributes␣for␣the␣
dataset.
"""
import os
import tarfile
import urllib.request
import warnings
warnings.filterwarnings("ignore")
import math
import pandas as pd
from pandas.plotting import scatter_matrix
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit

import data_download
import data_display

# References
data = data_download.data
housing = data.drop(columns=["ocean_proximity"])

# Column Division
housing["rooms_per_household"] = housing["total_rooms"] / housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"] / housing["total_rooms"]
housing["population_per_household"] = housing["population"] / housing["households"]


if __name__ == "__main__":

    # Dataset with new columns
    print(housing.iloc[0:1])

    # Linear Correlation
    corr_matrix = housing.corr()
    linear_corr_median_house_value = corr_matrix["median_house_value"].sort_values(
    ascending=False)
    print(linear_corr_median_house_value)

    # Scatter Plot
    cols = ["rooms_per_household", "bedrooms_per_room", "population_per_household"]
    for col in cols:
        plt.scatter(housing[col], housing["median_house_value"])
        plt.show()
```

Data Cleaning: Column with missing values

Fill the missing values of the columns with the mean or median of each respective column.

Listing 11: Separates the dataset from the column with the labels and replace the missing values of the columns with the median of each respective column.

```python
# data_cleaning.py
"""
This␣file␣shows␣how␣to␣fill␣the␣missing␣column␣values␣of␣the␣dataset␣with␣the␣median␣of
␣each␣respective␣column.
"""
import os
import tarfile
import urllib.request
import warnings
warnings.filterwarnings("ignore")
import math
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.impute import SimpleImputer

import data_download
import stratification

# Dataset
housing = stratification.data_stratified_train.drop(columns=["median_house_value"])
# Column Label
housing_labels = stratification.data_stratified_train["median_house_value"].copy()
# Numerical dataset
housing_numerical = housing.drop(columns=["ocean_proximity"])

# Impute the median of each column into all the the missing column values
def impute_median(dataset_numerical):
    imputer = SimpleImputer(strategy="median")
    imputer.fit(dataset_numerical)
    dataset_transformed = imputer.transform(dataset_numerical)
    dataset_df = pd.DataFrame(dataset_transformed, columns = housing_numerical.columns)
    return dataset_df

if __name__ == "__main__":

    housing_transformed = impute_median(housing_numerical)

    # Missing Values
    print(housing_numerical.isnull().sum())
    print(pd.isnull(housing_transformed).sum())
```

Text Encoding: Column with text values

How to encode the text values of a column into integers or binary vectors.

Listing 12: Text Encoding: Text, Int, and Binary Vector

```
# text_encoding.py
"""
This␣file␣shows␣how␣to␣encode␣a␣text␣column␣into␣integers␣and␣binary␣vectors␣for␣data␣
preprocessing.
"""
import os
import tarfile
import urllib.request
import warnings
warnings.filterwarnings("ignore")
import math
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.cluster import KMeans
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelBinarizer

import data_download
import stratification

# Dataset
housing = stratification.data_stratified_train.drop(columns=["median_house_value"])
housing_text = housing["ocean_proximity"]

print(housing_text.head(3), "\n")

# Encoding the Text Column: Text -> Int
encoder = LabelEncoder()
housing_text_encoded = encoder.fit_transform(housing_text)

print("Text␣Encoding␣(text␣->␣int):\n")
print(housing_text_encoded[0:3])
print(encoder.classes_)
print("'<1H␣OCEAN'␣=␣0,␣'INLAND'␣=␣1,␣'ISLAND'␣=␣2,␣'NEAR␣BAY'␣=␣3,␣'NEAR␣OCEAN'␣=␣4\n"
)

# Binary Encoding: Int -> Binary Vector
encoder = OneHotEncoder()
```

```
housing_text_encoded_binary = encoder.fit_transform(housing_text_encoded.reshape(-1,1))

print("Binary␣Encoding␣(int␣->␣binary␣vector):\n")
print(housing_text_encoded_binary.toarray()[0:3])

# Text -> Int -> Binary Vector
encoder = LabelBinarizer()
housing_text_encoded_int_bin = encoder.fit_transform(housing_text)

print(housing_text_encoded_int_bin[0:3])
print(encoder.classes_)
print("'<1H␣OCEAN'␣=␣[1,0,0,0,0],␣'INLAND'␣=␣[0,1,0,0,0],␣'ISLAND'␣=␣[0,0,1,0,0],␣'NEAR
␣BAY'␣=␣[0,0,0,1,0],␣'NEAR␣OCEAN'␣=␣[0,0,0,0,1]\n")
```

Transformers: Classes with Functions that modify the dataset

They are classes or group of functions created to handle data cleanup operations or to combine columns and create new columns, attributes, or categories to modify the dataset.

Listing 13: Custom Classes to modify the dataset.

```
# transformer.py
"""
This␣module␣contains␣the␣classes/transformers␣holding␣functions␣that␣modify␣the␣dataset
for␣the␣full␣data␣transformation␣pipeline␣to␣preprocess␣numerical␣and␣categorical/text␣
data.
"""

# Python STL
import os
import tarfile
import urllib.request
import math

# Packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Test Datasets
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit

# Tranformation Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelBinarizer
from sklearn.preprocessing import StandardScaler
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.pipeline import Pipeline
from sklearn.pipeline import FeatureUnion

# ML Algorithms
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error

# Modules
import data_download
import data_display
import stratification

# Selector: to select the numerical and text columns from a dataset
```

```
class CustomDataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, columns):
        self.columns = columns
    def fit(self, dataset, dataset_label = None):
        return self
    def transform(self, dataset, dataset_label = None):
        return dataset[self.columns].values

# Text Encoder: to convert text columns into binary vectors
class CustomLabelBinarizer(BaseEstimator, TransformerMixin):
    def __init__(self, categories = None):
        self.categories = categories
        self.label_binarizer = None

    def fit(self, dataset, dataset_label = None):
        if self.categories is not None:
            self.label_binarizer = LabelBinarizer()
            self.label_binarizer.fit(self.categories) # text encode the specific text
            categories
        else:
            self.label_binarizer = LabelBinarizer()
            self.label_binarizer.fit(dataset)        # text encode the text column
            values that appear on the dataset
        return self                                  # warning: smaller subsets may not
         show all the possible text categories from the dataset

    def transform(self, dataset, dataset_label = None):
        return self.label_binarizer.transform(dataset)

# Custom Transformer Class: group of functions to modify the dataset
class CustomTransformer(BaseEstimator, TransformerMixin):

    def __init__(self):
        pass

    def fit(self, dataset, dataset_label = None):
        return self

    # Creation of new columns
    def transform(self, dataset, dataset_label = None):

        rooms_index, population_index, households_index = 3, 5, 6
        rooms_per_household = dataset[:, rooms_index] / dataset[:, households_index]
        population_per_household = dataset[:, population_index] / dataset[:,
        households_index]

        return np.c_[dataset, rooms_per_household, population_per_household]
```

Transformation Pipeline

- The Transformation Pipeline only makes modifications on the selected numerical and text columns.
- Warning: Do not feed the Column Label to the Transformation Pipeline (`dataset = module_file_stratification.data_stratified_train.drop(columns=["column_label"])`), otherwise it will memorize the values for the Column of the Predictions and the Root Mean Square Error (RMSE) will be close to 0.
- Warning: Fit the entire dataset (`full_pipeline.fit(housing)`) so the text encoder will read all the possible categorical values of the text column and translate them into all the unique possible binary vectors to smaller dataset portion or sample. You can also create a list with all the possible categorical text values from the text column and pass them as input through the transformation pipeline, and then modify the custom text encoder, the transformer class, so it will take them as a parameter to encode all the text values to integers or binary vectors.
- Fill missing values with the Median of the Column
- Divide Columns to create new ones
- Standardization of Columns (mean = 0, standard deviation = 1)
- Text Binary Encoding of a Column (Text → Int → Binary Vector)

Listing 14: Transformation Pipeline: Numerical and Text Columns

```
# transformation_pipeline.py
"""
This␣module␣contains␣the␣full␣transformation␣pipeline
for␣preprocessing␣numerical␣and␣text␣columns␣from␣a␣dataset.
"""
# Python STL
import os
import tarfile
import urllib.request
import math

# Packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Test Datasets
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit
```

```python
# Tranformation Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelBinarizer
from sklearn.preprocessing import StandardScaler
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.pipeline import Pipeline
from sklearn.pipeline import FeatureUnion

# ML Algorithms
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error

# Modules
import data_download
import data_display
import stratification
import transformer
import transformation_pipeline

# References
custom_dataframeselector = transformer.CustomDataFrameSelector
custom_labelbinarizer = transformer.CustomLabelBinarizer
custom_transformer = transformer.CustomTransformer

# Deletion of the Label Column and separation of the Text Column
housing = stratification.data_stratified_train.drop(columns=["median_house_value"])
housing_numerical = housing.drop(columns=["ocean_proximity"])
housing_text = housing["ocean_proximity"]

# Dataset Columns to be processed by the Pipeline
# Each text category, encoded into a binary vector, is treated as an independent column
 with some specific weight (importance) for the creation of the prediction column
numerical_columns = list(housing_numerical)
text_columns = ["ocean_proximity"]
text_categories = ['<1H␣OCEAN', 'INLAND', 'ISLAND', 'NEAR␣BAY', 'NEAR␣OCEAN']

# Tranformation Pipeline
numerical_pipeline = Pipeline([
    ("selector", custom_dataframeselector(numerical_columns)),
    ("imputer", SimpleImputer(strategy="median")),
    ("transformer", custom_transformer()),
    ("std_scaler", StandardScaler())
])

text_pipeline = Pipeline([
    ("selector", custom_dataframeselector(text_columns)),
    ("label_binarizer", custom_labelbinarizer(text_categories))
])

full_pipeline = FeatureUnion(transformer_list=[
    ("numerical_pipeline", numerical_pipeline),
    ("text_pipeline", text_pipeline)
])

def transform_dataframe(dataset_transformed, dataset_numerical = None, dataset_text =
None, new_numerical_columns = None, new_text_columns = None):

    full_columns = []

    if (dataset_numerical is not None):
        numerical_columns = list(dataset_numerical.columns)
        full_columns += numerical_columns
    if (new_numerical_columns is not None):
        full_columns += new_numerical_columns
    if (dataset_text is not None):
        text_encoder = LabelBinarizer()
        text_encoder.fit_transform(dataset_text)
        text_categories = [str(text_category) for text_category in text_encoder.
        classes_]
        full_columns += text_categories
    if (new_text_columns is not None):
        full_columns += new_text_columns

    dataframe_transformed = pd.DataFrame(dataset_transformed, columns=full_columns)

    return dataframe_transformed

if __name__ == "__main__":

    # Transform the Dataset
    transformer_function = transformer.CustomTransformer()
    housing_transformed = full_pipeline.fit_transform(housing)
    housing_transformed_df = transform_dataframe(housing_transformed, housing_numerical
    , housing_text, ["rooms_per_household", "population_per_household"])

    # Dataset
    print("Dataset:")
    print(housing.iloc[0:1], "\n")

    # Dataset Transformed
    print("Dataset␣Transformed:")
    print(housing_transformed[0:1], "\n")

    print("Dataframe␣Transformed:")
    print(housing_transformed_df.iloc[0:1], "\n")
```

Standardization of a Column

$$Column = V_0, V_1, V_2, V_3, ...V_n \rightarrow Column' = Z_0, Z_1, Z_2, Z_3, ...Z_n$$

$$Z_i = \frac{V_i - mean(Column)}{Standard\,Deviation(Column)}$$

$$Mean(Column') = \frac{1}{n}\sum_{i=1}^{n}(Z_i) \simeq 0$$

$$SD(Column') = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(Z_i - mean')^2} \simeq 1$$

Train the Machine Learning Model

The LinearRegression(), DecisionTreeRegressor(), and RandomForestRegressor() functions map the transformed columns, the numerical and binary encoded columns output by the transformation pipeline, to the target or label column, the solutions, of the dataset. The regression algorithm calculates a set of weights for each column value, so each transformed column value contributes a certain weight to the value of the column label.

$$L_i = C_1 \cdot W_1 + C_2 \cdot W_2 + C_3 \cdot W_3 ... + C_n \cdot W_n + b$$

- L_i: Value of the Column Label of the row i
- C_x: Value of the Column x of the row i
- W_x: Weight of the Column x
- b: bias

```
# Python STL
import os
import tarfile
import urllib.request
import math

# Packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Test Datasets
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit

# Tranformation Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelBinarizer
from sklearn.preprocessing import StandardScaler
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.pipeline import Pipeline
from sklearn.pipeline import FeatureUnion

# ML Algorithms
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
```

```python
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestRegressor

# Save Model
import joblib

# Fine tune Model
from sklearn.model_selection import GridSearchCV

# Modules
import data_download
import data_display
import stratification
import transformer
import transformation_pipeline

# References
full_pipeline = transformation_pipeline.full_pipeline
data_stratified_train = stratification.data_stratified_train
data_stratified_test = stratification.data_stratified_test

# Load Dataset
housing_train = data_stratified_train.drop(columns=["median_house_value"])
housing_train_labels = data_stratified_train["median_house_value"].copy()
housing_test = data_stratified_test.drop(columns=["median_house_value"])
housing_test_labels = data_stratified_test["median_house_value"].copy()

# Paths
FILE_DIR = os.path.abspath(os.path.dirname(__file__))
PARENT_DIR = os.path.dirname(FILE_DIR)
MODEL_DIR = os.path.join(PARENT_DIR, "models")
HOUSING_MODEL_DIR = os.path.join(MODEL_DIR, "housing")
os.makedirs(HOUSING_MODEL_DIR, exist_ok=True)

# ML Algorithms
lin_reg = LinearRegression()
tree_reg = DecisionTreeRegressor()
forest_reg = RandomForestRegressor()

# Standard Deviation (spread of the predicted values): average distance of the
predicted values from their own mean.
# RMSE (average deviation error of the predicted values): average value of the
differences between the predicted values from their corresponding label values.
# ML Model = ML Algorithm + Dataset
# K-fold cross_val_score: returns the variance (deviation from the label values) of the
 predicted column values from a ML Algorithm applied on K subsets (folds).
# train_models: saves/loads the model + creates the prediction column based on the ML
algorithm + calculates the RMSE
```

```
# of the prediction column values in relation to the label column values for 1
transformed dataset and for 10 subsets of the transformed dataset.
# fine_tune_model: finds the best parameters for a ML Model to have the best
performance and minimum RMSE across the dataset and subsets.
# Random Forest: an ensemble algorithm consisting of multiple decision trees
# Decision Trees: built by recursively splitting the dataset into subsets of columns
and rows at each node
# n_estimators: number of decision trees in the random forest
# max_features: maximum number of columns randomly selected from the dataset’s total
columns to consider for each split (node) in the decision trees
# bootstrap=True: rows for each decision tree are randomly selected **with replacement
** (default behavior)
# bootstrap=False: all rows in the dataset are used to train each decision tree (no
replacement)
# Each text category that is encoded into a binary vector using LabelBinarizer (the
text_encoder)
# becomes its own independent column in the dataset, and the model assigns a separate
weight (importance) to each column.

def display_scores(scores):
    print("RMSE␣for␣each␣subset:", scores)
    print("Mean␣of␣the␣RMSEs:", scores.mean())
    print("Standard␣deviation␣of␣the␣RMSEs:", scores.std())

def train_models(dataset_transformed, dataset_labels, train_models: bool):

    if (train_models == True):

        # Linear Regression
        lin_reg.fit(dataset_transformed, dataset_labels)
        lin_reg_path = os.path.join(HOUSING_MODEL_DIR, "linear_regression_model.pkl")
        joblib.dump(lin_reg, lin_reg_path)
        housing_predictions_lin = lin_reg.predict(dataset_transformed)
        lin_rmse = np.sqrt(mean_squared_error(dataset_labels, housing_predictions_lin))
        lin_scores = cross_val_score(lin_reg, dataset_transformed, dataset_labels,
        scoring="neg_mean_squared_error", cv=10)
        lin_scores_rmse = np.sqrt(-lin_scores)

        # Decision Tree
        tree_reg.fit(dataset_transformed, dataset_labels)
        tree_reg_path = os.path.join(HOUSING_MODEL_DIR, "decision_tree_model.pkl")
        joblib.dump(tree_reg, tree_reg_path)
        housing_predictions_tree = tree_reg.predict(dataset_transformed)
        tree_rmse = np.sqrt(mean_squared_error(dataset_labels, housing_predictions_tree
        ))
        tree_scores = cross_val_score(tree_reg, dataset_transformed, dataset_labels,
        scoring="neg_mean_squared_error", cv=10)
        tree_scores_rmse = np.sqrt(-tree_scores)
        # Random Forest
        forest_reg.fit(dataset_transformed, dataset_labels)
        forest_reg_path = os.path.join(HOUSING_MODEL_DIR, "random_forest_model.pkl")
        joblib.dump(forest_reg, forest_reg_path)
        housing_predictions_forest = forest_reg.predict(dataset_transformed)
        forest_rmse = np.sqrt(mean_squared_error(dataset_labels,
        housing_predictions_forest))
        forest_scores = cross_val_score(forest_reg, dataset_transformed, dataset_labels
        , scoring="neg_mean_squared_error", cv=10)
        forest_scores_rmse = np.sqrt(-forest_scores)
    else:

        lin_reg_path = os.path.join(HOUSING_MODEL_DIR, "linear_regression_model.pkl")
        tree_reg_path = os.path.join(HOUSING_MODEL_DIR, "decision_tree_model.pkl")
        forest_reg_path = os.path.join(HOUSING_MODEL_DIR, "random_forest_model.pkl")

        # Load pre-trained models
        lin_reg_loaded = joblib.load(lin_reg_path)
        tree_reg_loaded = joblib.load(tree_reg_path)
        forest_reg_loaded = joblib.load(forest_reg_path)

        housing_predictions_lin = lin_reg_loaded.predict(dataset_transformed)
        housing_predictions_tree = tree_reg_loaded.predict(dataset_transformed)
        housing_predictions_forest = forest_reg_loaded.predict(dataset_transformed)

        lin_rmse = np.sqrt(mean_squared_error(dataset_labels, housing_predictions_lin))
        lin_scores = cross_val_score(lin_reg_loaded, dataset_transformed,
        dataset_labels, scoring="neg_mean_squared_error", cv=10)
        lin_scores_rmse = np.sqrt(-lin_scores)

        tree_rmse = np.sqrt(mean_squared_error(dataset_labels, housing_predictions_tree
        ))
        tree_scores = cross_val_score(tree_reg_loaded, dataset_transformed,
        dataset_labels, scoring="neg_mean_squared_error", cv=10)
        tree_scores_rmse = np.sqrt(-tree_scores)

        forest_rmse = np.sqrt(mean_squared_error(dataset_labels,
        housing_predictions_forest))
        forest_scores = cross_val_score(forest_reg_loaded, dataset_transformed,
        dataset_labels, scoring="neg_mean_squared_error", cv=10)
        forest_scores_rmse = np.sqrt(-forest_scores)

    print("Linear␣Regression␣(RMSE␣of␣the␣transformed␣stratified␣dataset):␣", lin_rmse)
    print("Linear␣Regression␣(RMSEs␣for␣10␣subsets):")
    display_scores(lin_scores_rmse)
    print("\n")

    print("Decision␣Tree␣(RMSE␣of␣the␣transformed␣stratified␣dataset):␣", tree_rmse)
    print("Decision␣Tree␣(RMSEs␣for␣10␣subsets):")
    display_scores(tree_scores_rmse)
    print("\n")

    print("Random␣Forest␣(RMSE␣of␣the␣transformed␣stratified␣dataset):␣", forest_rmse)
    print("Random␣Forest␣(RMSEs␣for␣10␣subsets):")
    display_scores(forest_scores_rmse)
    print("\n")

def fine_tune_model(dataset_transformed, dataset_labels, dataset_numerical,
dataset_text, model, save_model: bool, model_name="model"):

    # Search for the best parameters for the model: minimum RMSE and best performance
    across the dataset and subsets
    # Grid of parameters
    param_grid = [
        {"n_estimators":[3,10,30], "max_features":[2,4,6,8]},
        {"bootstrap":[False], "n_estimators":[3,10], "max_features":[2,3,4]}
    ]
    grid_search = GridSearchCV(model, param_grid, cv=5, scoring="neg_mean_squared_error
    ")
    grid_search.fit(dataset_transformed, dataset_labels)

    best_parameters = grid_search.best_params_
    best_model = grid_search.best_estimator_
    best_model_predictions = best_model.predict(dataset_transformed)
    best_model_rmse = np.sqrt(mean_squared_error(dataset_labels, best_model_predictions
    ))

    cv_results = grid_search.cv_results_
    print("Mean␣of␣the␣RMSEs␣of␣the␣subsets␣for␣each␣parameter␣combination:")
    for mean_score, params in zip(cv_results["mean_test_score"], cv_results["params"]):
        print("Mean␣of␣the␣RMSEs:␣", np.sqrt(-mean_score), "for␣parameters:␣", params)

    print("Best␣parameters␣for␣the␣model:␣", best_parameters)
    print("Best␣model␣RMSE:␣", best_model_rmse)

    # Calculates the importance of each column and text category in the formation of
    the prediction column
    # Retrieve the feature importances (column weights) and combine with column names
    # The order matches because the columns and computed weights are in the same order
    of the columns fed to the transformation pipeline
    column_weights = [float(weight) for weight in best_model.feature_importances_]
    numerical_columns = list(dataset_numerical)
    new_columns = ["rooms_per_household", "population_per_household"]
    text_encoder = LabelBinarizer()
    text_encoder.fit_transform(dataset_text)
    text_categories = [str(text_category) for text_category in text_encoder.classes_]
    full_columns = numerical_columns + new_columns + text_categories
    sorted_column_weights = sorted(zip(column_weights, full_columns), reverse=True)
    print("Column␣Weights␣(Sorted␣by␣Importance):")
    print(sorted_column_weights)

    # Test the best model on the test dataset
    dataset_test = housing_test
    dataset_test_labels = housing_test_labels
    dataset_test_transformed = full_pipeline.transform(dataset_test)
    best_model_predictions_test = best_model.predict(dataset_test_transformed)
    best_model_rmse_test = np.sqrt(mean_squared_error(dataset_test_labels,
    best_model_predictions_test))
    print("Best␣model␣RMSE␣on␣the␣test␣dataset:␣", best_model_rmse_test)

    if (save_model):
        model_full_name = model_name + "_fine_tuned_model" + ".pkl"
        fine_tuned_model_path = os.path.join(HOUSING_MODEL_DIR, model_full_name)
        joblib.dump(best_model, fine_tuned_model_path)
        print("Fine-tuned␣model␣saved␣at:", fine_tuned_model_path)

    return best_model

def prediction_columns(dataset_transformed, dataset_labels):

    lin_reg.fit(dataset_transformed, dataset_labels)
    tree_reg.fit(dataset_transformed, dataset_labels)
    forest_reg.fit(dataset_transformed, dataset_labels)

    dataset_predictions_lin = lin_reg.predict(dataset_transformed)
    dataset_predictions_tree = tree_reg.predict(dataset_transformed)
    dataset_predictions_forest = forest_reg.predict(dataset_transformed)

    print("Columns␣of␣Predictions␣(Linear␣Regression␣x␣Decision␣Tree␣x␣Random␣Forest):"
    )
    print(dataset_predictions_lin[0:5], "\n")
    print(dataset_predictions_tree[0:5], "\n")
    print(dataset_predictions_forest[0:5], "\n")
    print("Column␣of␣Labels:")
    print(dataset_labels.iloc[0:5].values, "\n")

if __name__ == "__main__":

    dataset_sample = housing_train
    dataset_sample_labels = housing_train_labels
    dataset_sample_numerical = dataset_sample.drop(columns=["ocean_proximity"])
    dataset_sample_text = dataset_sample["ocean_proximity"]
    full_pipeline.fit(dataset_sample)
    dataset_sample_transformed = full_pipeline.transform(dataset_sample)

    prediction_columns(dataset_sample_transformed, dataset_sample_labels)
```

```
train_models(dataset_sample_transformed, dataset_sample_labels, False)

best_model_forest_reg = fine_tune_model(dataset_sample_transformed,
dataset_sample_labels, dataset_sample_numerical, dataset_sample_text, forest_reg,
False, "forest_reg")
```

Machine Learning Model = ML Algorithm + Dataset

The ML model is the trained version of a machine learning algorithm applied to a dataset (pre-processed through a transformation pipeline).

The algorithm is the recipe and the dataset are the ingredients. Once you follow the recipe with the ingredients, the result is the trained model.

- Machine Learning Algorithm: A mathematical framework or method (e.g., Random Forest, Support Vector Machine, etc.) used to learn patterns from data.
- Dataset: The data the algorithm uses to learn patterns, typically transformed into a suitable format (e.g., scaled, encoded, or otherwise prepared).
- Model: When you train the algorithm on the dataset, it becomes a model—a specific instance of the algorithm that has learned from the data.

Fine-tuning the model

Fine-tuning adjusts parameters of the ML Algorithm applied to the Dataset to ensure the Model is both accurate and consistent across different datasets.

The goal of fine-tuning is to minimize the standard deviation and to maximize the model's predictive performance on unseen data by carefully choosing hyperparameters. For example, in Random Forest, hyperparameters include the number of trees, maximum tree depth, etc.

Evaluation: The performance is typically evaluated using metrics (e.g., Mean Absolute Error, Root Mean Square Error (Standard Deviation), etc.) and cross-validation to ensure that the model generalizes well over different subsets.

Bibliography

[1] Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

[2] Learning Deep Learning: Theory and Practice of Neural Networks, Computer Vision, Natural Language Processing, and Transformers Using Tensorflow