Housing Dataset from California, 1990

Weslley Matheus

Contents

Linear Regression

Dataset Information

- Source: raw.githubusercontent.com/ageron/handson-ml/master/datasets/housing/housing.tgz
- 20,640 rows (house districts) with 10 columns: longitude, latitude, housing_median_age,total_rooms, total_bedrooms, population, households, median_income, median_house_value, ocean_proximity.
- The transformation pipeline created 2 numerical columns –"rooms_per_household" and "population_per_household"–, transformed the numerical columns by doing the imputation of missing values with the column mean and standardization of the columns and encoded the row values of the text column "ocean_proximity" into binary vectors.
- The machine learning algorithms created prediction columns based on the label column "median_house_value" using a stratified version of the dataset.

Full Code

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
```

```python
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"
```

```python
    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.
```

```python
    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
```

```python
        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, combing
        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)
```

Appendix

Formulas

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 $d = \mid \Delta x \mid + \mid \Delta y \mid$
- 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.

Standardization of a Column

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

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

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

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

Bibliography

[1] Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems

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

[3] Mathematics for Machine Learning