LAB MANUAL

SUBJECT: MACHINE LEARNING

SUBJECT CODE: AI-414

SUBMITTED BY:

NAME: AYESHA IMRAN

REG#: 2023-BS-AI-061

DEGREE: BSAI-4A

SUBMITTED TO:

NAME: MR.SAEED

TABLE OF CONTENTS

PROJECT 1

House Price Regression

PROJECT 2

Extrovert vs. Introvert Behavior Data

PROJECT 1

House Price Regression

Project Summary

This project focuses on preprocessing a house pricing dataset to prepare it for further analysis and machine learning tasks. The main emphasis is on cleaning, transforming, and engineering features from the raw dataset to ensure the data quality is suitable for building predictive models. The notebook centers around preparing a **housing dataset** for predictive modeling—particularly for predicting **house sale prices**. The preprocessing steps are crucial in ensuring that the data is clean, consistent, and formatted in a way that maximizes the performance of machine learning models. The process includes data cleaning, transformation, feature engineering, and encoding—transforming messy, real-world data into a high-quality dataset ready for modeling.

Objectives

- Understand and clean the raw dataset related to house prices.
- Handle missing values, outliers, and irrelevant features.
- Convert categorical variables into numerical formats suitable for modeling.
- Normalize or scale features if necessary.
- Prepare the dataset for use in machine learning algorithms.

Abstract

The dataset contains information about various houses, including features like area, number of rooms, year built, and more. Preprocessing is essential because real-world data is often incomplete, inconsistent, or improperly formatted. This notebook performs the essential preprocessing steps: identifying and handling null values, encoding categorical features, removing outliers, and ensuring the data is in a clean format. These transformations help improve the performance and reliability of machine learning models trained on this dataset.

Explanation of Steps

1. Importing Libraries

Key Python libraries such as pandas, numpy, seaborn, and matplotlib are imported for data manipulation and visualization.

2. Loading the Dataset

- The dataset is read using pd.read_csv() from a CSV file.
- Basic dataset inspection is performed using head(), info(), and describe().

3. Missing Value Treatment

- isnull().sum() is used to identify columns with missing data.
- Columns with high percentages of missing data are dropped.
- For other columns:
  - Numerical missing values are filled using mean/median.
  - Categorical values are filled with the mode or a placeholder like 'None'.

4. Exploratory Data Analysis (EDA)

- Visualizations such as histograms, boxplots, and heatmaps are used to understand data distributions, correlations, and potential outliers.

5. Outlier Detection and Removal

- Outliers are detected using z-score or IQR methods.
- Some features are clipped or filtered to remove extreme values.

6. Encoding Categorical Features

- Label Encoding and One-Hot Encoding are applied depending on the nature of the categorical data.

7. Feature Engineering

- New features are created based on existing ones, such as combining YearBuilt and YearRemodAdd into a new feature for age.
- Features are selected based on correlation with the target variable (SalePrice).

8. Normalization / Scaling

- StandardScaler or MinMaxScaler is applied to normalize numerical features for better model performance.

9. Saving the Cleaned Dataset

- The final cleaned and processed dataset is saved as a CSV file for future use in model building.

NOTEBOOK SCREENSHOTS:

Data Preprocessing Steps

1. Reading Data
2. Exploring Data / Data Insight
3. Cleansing Data
4. Outlier Detection and Removing
5. Data Transformation (Normalize Data / Rescale Data)
6. Categorical into Numerical
7. Dimensionality Reduction(PCA)
8. Handling Imbalanced Data
9. Feature Selection
10. Data Splitting

[130]:
```
import matplotlib.pyplot as plt
import seaborn as sns
color = sns.color_palette()

import numpy as np
import pandas as pd )
```

1: Reading Data ¶

[131]:
```
data = pd.read_csv('house_price_regression_dataset.csv')

data.head()
```

[131]:

	Square_Footage	Num_Bedrooms	Num_Bathrooms	Year_Built	Lot_Size	Garage_Size	Neighborhood_Quality	House_Price
0	1360	2	1	1981	0.599637	0	5	2.623829e+05
1	4272	3	3	2016	4.753014	1	6	9.852609e+05
2	3592	1	2	2016	3.634823	0	9	7.779774e+05
3	966	1	2	1977	2.730667	1	8	2.296989e+05
4	4926	2	1	1993	4.699073	0	8	1.041741e+06

[132]:
```
data.head(2)
```

[132]:

	Square_Footage	Num_Bedrooms	Num_Bathrooms	Year_Built	Lot_Size	Garage_Size	Neighborhood_Quality	House_Price
0	1360	2	1	1981	0.599637	0	5	262382.852274
1	4272	3	3	2016	4.753014	1	6	985260.854490

[133]:
```
data.head(30)
```

[133]:

	Square_Footage	Num_Bedrooms	Num_Bathrooms	Year_Built	Lot_Size	Garage_Size	Neighborhood_Quality	House_Price
0	1360	2	1	1981	0.599637	0	5	2.623829e+05
1	4272	3	3	2016	4.753014	1	6	9.852609e+05
2	3592	1	2	2016	3.634823	0	9	7.779774e+05
3	966	1	2	1977	2.730667	1	8	2.296989e+05

[134]:
```
data.tail()
```

[134]:

	Square_Footage	Num_Bedrooms	Num_Bathrooms	Year_Built	Lot_Size	Garage_Size	Neighborhood_Quality	House_Price
995	3261	4	1	1978	2.165110	2	10	701493.997069
996	3179	1	2	1999	2.977123	1	10	683723.160704
997	2606	4	2	1962	4.055067	0	2	572024.023634
998	4723	5	2	1950	1.930921	0	7	964865.298639
999	3268	4	2	1983	3.108790	2	2	742599.253332

[135]:
```
data.shape
```

[135]: (1000, 8)

[136]:
```
data.tail(20)
```

[136]:

	Square_Footage	Num_Bedrooms	Num_Bathrooms	Year_Built	Lot_Size	Garage_Size	Neighborhood_Quality	House_Price
980	1414	5	1	1996	3.651943	2	9	3.722926e+05
981	1808	1	3	1968	3.967012	1	5	4.023301e+05
982	4677	4	3	1964	2.017863	1	1	9.653048e+05
983	1013	1	1	1977	4.737096	1	7	2.486218e+05
984	2420	5	3	1973	2.883713	2	10	5.608898e+05
985	4094	4	1	2021	3.795748	1	10	9.333327e+05
986	1909	3	2	1958	4.658485	2	4	4.535791e+05
987	3690	3	1	1996	4.888310	0	9	8.148437e+05
988	4991	1	2	1982	4.249420	2	6	1.037268e+06
989	2055	1	3	2011	1.863579	0	3	4.607412e+05

137]:
```
data.sample()
```

137]:

	Square_Footage	Num_Bedrooms	Num_Bathrooms	Year_Built	Lot_Size	Garage_Size	Neighborhood_Quality	House_Price
716	4235	3	3	2000	1.911679	1	8	917235.410532

138]:
```
data.sample(30)
```

138]:

	Square_Footage	Num_Bedrooms	Num_Bathrooms	Year_Built	Lot_Size	Garage_Size	Neighborhood_Quality	House_Price
963	3554	2	2	1994	0.800122	1	9	7.342015e+05
290	1037	2	2	1964	4.889180	2	10	2.559071e+05
191	3869	4	1	2009	3.283372	1	1	8.787041e+05
717	3370	5	3	2021	1.615430	1	2	7.693186e+05
413	1228	2	1	1953	3.318451	2	2	2.697364e+05
758	820	4	1	2018	1.065045	2	2	2.360540e+05

```
[139]: data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 8 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Square_Footage        1000 non-null   int64
 1   Num_Bedrooms          1000 non-null   int64
 2   Num_Bathrooms         1000 non-null   int64
 3   Year_Built            1000 non-null   int64
 4   Lot_Size              1000 non-null   float64
 5   Garage_Size           1000 non-null   int64
 6   Neighborhood_Quality  1000 non-null   int64
 7   House_Price           1000 non-null   float64
dtypes: float64(2), int64(6)
memory usage: 62.6 KB
```

```
[140]: data.describe()
```

[140]:

	Square_Footage	Num_Bedrooms	Num_Bathrooms	Year_Built	Lot_Size	Garage_Size	Neighborhood_Quality	House_Price
count	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1.000000e+03
mean	2815.422000	2.990000	1.973000	1986.550000	2.778087	1.022000	5.615000	6.188610e+05
std	1255.514921	1.427564	0.820332	20.632916	1.297903	0.814973	2.887059	2.535681e+05
min	503.000000	1.000000	1.000000	1950.000000	0.506058	0.000000	1.000000	1.116269e+05
25%	1749.500000	2.000000	1.000000	1969.000000	1.665946	0.000000	3.000000	4.016482e+05
50%	2862.500000	3.000000	2.000000	1986.000000	2.809740	1.000000	6.000000	6.282673e+05
75%	3849.500000	4.000000	3.000000	2004.250000	3.923317	2.000000	8.000000	8.271413e+05
max	4999.000000	5.000000	3.000000	2022.000000	4.989303	2.000000	10.000000	1.108237e+06

2: Data Cleaning

Handling Missing Values

- Imputation: Filling missing values with mean.

142]:
```
data.isnull().sum()
```

142]:
```
Square_Footage          0
Num_Bedrooms            0
Num_Bathrooms           0
Year_Built              0
Lot_Size                0
Garage_Size             0
Neighborhood_Quality    0
House_Price             0
dtype: int64
```

143]:
```
import pandas as pd
import numpy as np

numeric_cols = data.select_dtypes(include=[np.number])
non_numeric_cols = data.select_dtypes(exclude=[np.number])

numeric_cols.fillna(numeric_cols.mean(), inplace=True)


data = pd.concat([numeric_cols, non_numeric_cols], axis=1)

missing_values = data.isnull().sum()
print(missing_values)
```

```
Square_Footage          0
Num_Bedrooms            0
```

```
dtype: int64
```

```
[145]: data.shape
```

```
[145]: (1000, 8)
```

Removal: Deleting rows with missing values.

```
[146]: data.isnull().sum()
```

```
[146]: Square_Footage          0
       Num_Bedrooms            0
       Num_Bathrooms           0
       Year_Built              0
       Lot_Size                0
       Garage_Size             0
       Neighborhood_Quality    0
       House_Price             0
       dtype: int64
```

```
[147]: data.shape
```

```
[147]: (1000, 8)
```

```
[148]: data.dropna(inplace=True)

       missing_values = data.isnull().sum()
       print(missing_values)
```

```
Square_Footage          0
Num_Bedrooms            0
Num_Bathrooms           0
Year_Built              0
Lot_Size                0
Garage_Size             0
Neighborhood_Quality    0
House_Price             0
dtype: int64
```

Removing Duplicates ¶

[150]:
```
data.shape
```

[150]:
```
(1000, 8)
```

[151]:
```
data.drop_duplicates(inplace=True)
data.shape
```

[151]:
```
(1000, 8)
```

3: Outlier Detection and Removal

[152]:
```
data.describe()
```

[152]:

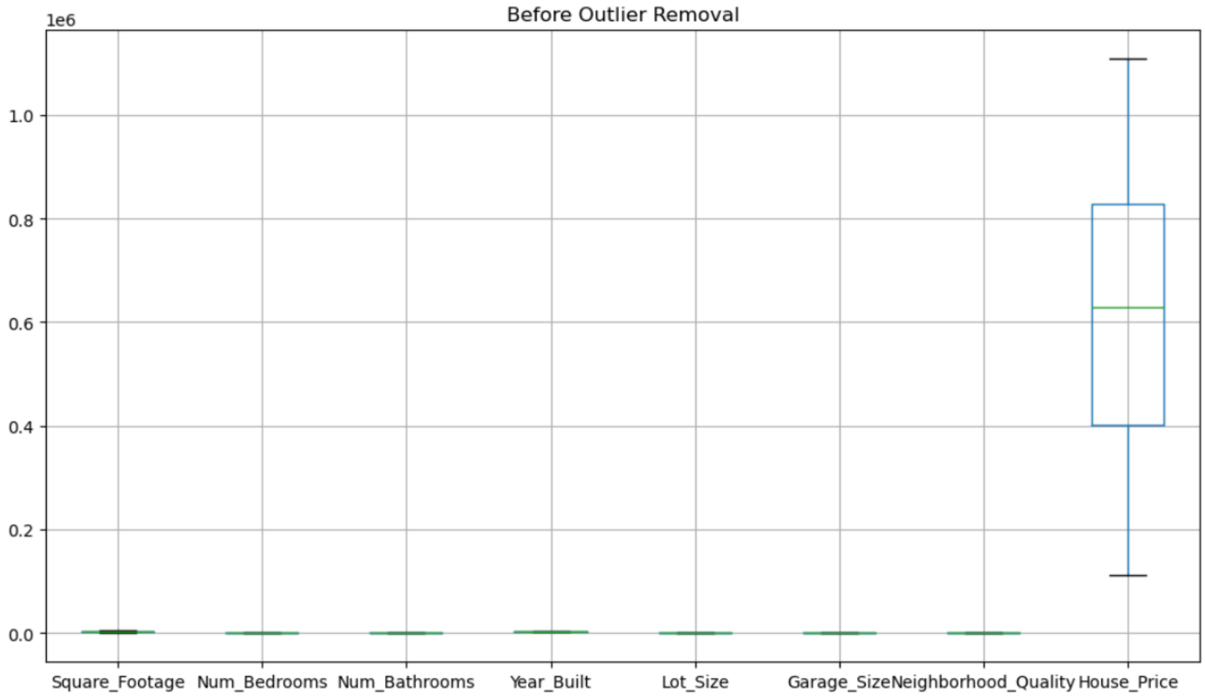
	Square_Footage	Num_Bedrooms	Num_Bathrooms	Year_Built	Lot_Size	Garage_Size	Neighborhood_Quality	House_Price
count	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1.000000e+03
mean	2815.422000	2.990000	1.973000	1986.550000	2.778087	1.022000	5.615000	6.188610e+05
std	1255.514921	1.427564	0.820332	20.632916	1.297903	0.814973	2.887059	2.535681e+05
min	503.000000	1.000000	1.000000	1950.000000	0.506058	0.000000	1.000000	1.116269e+05
25%	1749.500000	2.000000	1.000000	1969.000000	1.665946	0.000000	3.000000	4.016482e+05
50%	2862.500000	3.000000	2.000000	1986.000000	2.809740	1.000000	6.000000	6.282673e+05
75%	3849.500000	4.000000	3.000000	2004.250000	3.923317	2.000000	8.000000	8.271413e+05
max	4999.000000	5.000000	3.000000	2022.000000	4.989303	2.000000	10.000000	1.108237e+06

[153]:
```
0.25-1.5*0.5
```

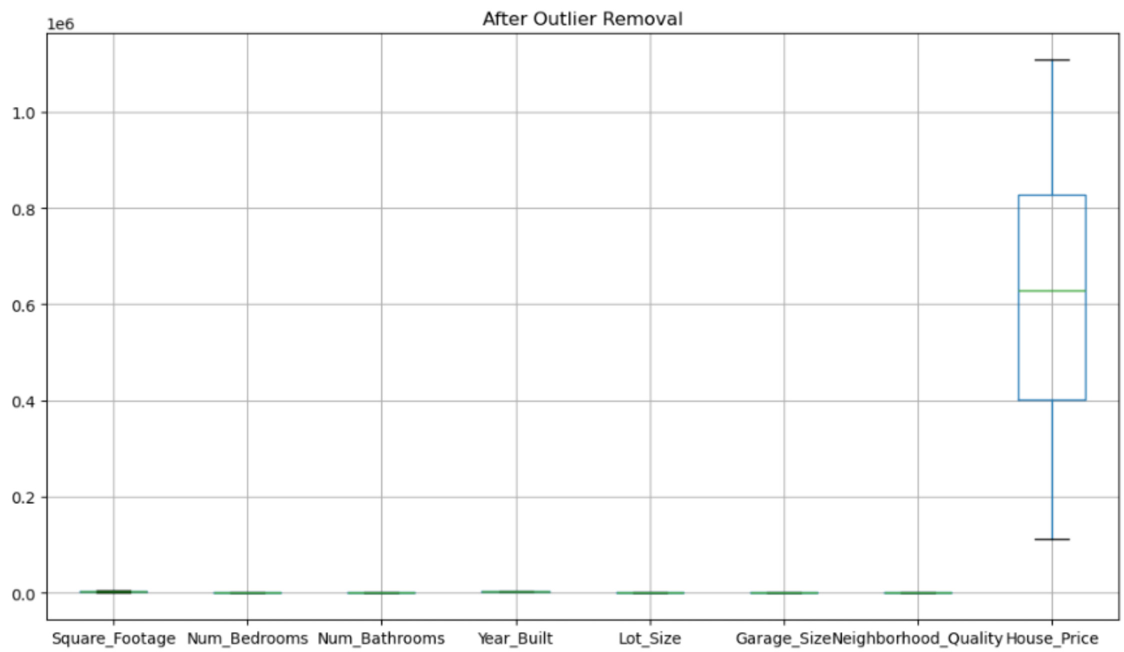
[154]:
```
0.75 + 1.5 * 0.5
```

[154]:
```
1.5
```

[155]:
```
numeric_cols = data.select_dtypes(include=[np.number])

Q1 = numeric_cols.quantile(0.25)
Q3 = numeric_cols.quantile(0.75)
IQR = Q3 - Q1

data_cleaned = data[~((numeric_cols < (Q1 - 1.5 * IQR)) | (numeric_cols > (Q3 + 1.5 * IQR))).any(axis=1)]

plt.figure(figsize=(20, 6))

plt.subplot(1, 2, 1)
numeric_cols.boxplot()
plt.title("Before Outlier Removal")

plt.tight_layout()
plt.show()
```



[156]:
```
plt.figure(figsize=(20, 6))
rs
plt.subplot(1, 2, 2)
data_cleaned.select_dtypes(include=[np.number]).boxplot()
plt.title("After Outlier Removal")

plt.tight_layout()
plt.show()
```



[157]:
```
data_cleaned.head()
```

[157]:

	Square_Footage	Num_Bedrooms	Num_Bathrooms	Year_Built	Lot_Size	Garage_Size	Neighborhood_Quality	House_Price
0	1360	2	1	1981	0.599637	0	5	2.623829e+05
1	4272	3	3	2016	4.753014	1	6	9.852609e+05
2	3592	1	2	2016	3.634823	0	9	7.779774e+05
3	966	1	2	1977	2.730667	1	8	2.296989e+05
4	4926	2	1	1993	4.699073	0	8	1.041741e+06

4. Data Transformation

Key Differences

Range of Values:

Normalization: Values are scaled to a fixed range, typically [0, 1]. Standardization: Values are rescaled to have a mean of 0 and a standard deviation of 1. Effect on Distribution:

Normalization: Compresses or stretches the data to fit within the specified range, potentially altering the original distribution. Standardization: Preserves the shape of the original distribution but changes the scale. Use Cases:

Normalization: Suitable for distance-based algorithms, like k-nearest neighbors and neural networks. Standardization: Suitable for algorithms that assume a normal distribution, like linear regression and logistic regression.

Normalization/Standardization

- Normalization Definition: Normalization rescales the data to a fixed range, typically [0, 1] or [-1, 1].

[158]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler

numeric_cols = data.select_dtypes(include=[np.number])
non_numeric_cols = data.select_dtypes(exclude=[np.number])

scaler = MinMaxScaler()
scaled_numeric_data = scaler.fit_transform(numeric_cols)

scaled_numeric_df = pd.DataFrame(scaled_numeric_data, columns=numeric_cols.columns)

scaled_data = pd.concat([scaled_numeric_df, non_numeric_cols.reset_index(drop=True)], axis=1)

print(scaled_data.shape)
print()
print('*' * 60)
scaled_data.head()
```

```
(1000, 8)

************************************************************
```

[158]:

	Square_Footage	Num_Bedrooms	Num_Bathrooms	Year_Built	Lot_Size	Garage_Size	Neighborhood_Quality	House_Price
0	0.190614	0.25	0.0	0.430556	0.020873	0.0	0.444444	0.151269
1	0.838301	0.50	1.0	0.916667	0.947295	0.5	0.555556	0.876606
2	0.687055	0.00	0.5	0.916667	0.697880	0.0	0.888889	0.668617
3	0.102980	0.00	0.5	0.375000	0.496205	0.5	0.777778	0.118474
4	0.983763	0.25	0.0	0.597222	0.935263	0.0	0.777778	0.933278

Standardization

Definition: Standardization rescales the data so that it has a mean of 0 and a standard deviation of 1.

[159]:
```
from sklearn.preprocessing import StandardScaler

numeric_cols = data.select_dtypes(include=[np.number])
non_numeric_cols = data.select_dtypes(exclude=[np.number])

scaler = StandardScaler()
scaled_numeric_data = scaler.fit_transform(numeric_cols)

scaled_numeric_df = pd.DataFrame(scaled_numeric_data, columns=numeric_cols.columns)

scaled_data = pd.concat([scaled_numeric_df, non_numeric_cols.reset_index(drop=True)], axis=1)

print(scaled_data.shape)
print()
print('*' * 60)
scaled_data.head()
```

```
(1000, 8)

************************************************************
```

[159]:

	Square_Footage	Num_Bedrooms	Num_Bathrooms	Year_Built	Lot_Size	Garage_Size	Neighborhood_Quality	House_Price
0	-1.159803	-0.693836	-1.186699	-0.269122	-1.679278	-1.254658	-0.213126	-1.406552
1	1.160724	0.007008	1.252559	1.428045	1.522390	-0.027008	0.133420	1.445699
2	0.618843	-1.394681	0.032930	1.428045	0.660422	-1.254658	1.173060	0.627824
3	-1.473776	-1.394681	0.032930	-0.463084	-0.036555	-0.027008	0.826514	-1.535512
4	1.681887	-0.693836	-1.186699	0.312764	1.480809	-1.254658	0.826514	1.668552

5: One-Hot Encoding

[160]:
```python
from sklearn.preprocessing import StandardScaler

data.head(2)
```

[160]:

	Square_Footage	Num_Bedrooms	Num_Bathrooms	Year_Built	Lot_Size	Garage_Size	Neighborhood_Quality	House_Price
0	1360	2	1	1981	0.599637	0	5	262382.852274
1	4272	3	3	2016	4.753014	1	6	985260.854490

[161]:
```python
data["Num_Bedrooms"].unique()
```

[161]:
```
array([2, 3, 1, 5, 4], dtype=int64)
```

[162]:
```python
data.Year_Built.unique()
```

[162]:
```
array([1981, 2016, 1977, 1993, 1990, 2012, 1972, 1997, 2006, 1982, 1973,
       1988, 1983, 2005, 1986, 1956, 2017, 2014, 1996, 1969, 1968, 1978,
       2009, 1967, 1984, 1992, 1960, 1998, 1987, 2013, 2018, 1957, 1980,
       1953, 1999, 1979, 2008, 1994, 1975, 1976, 1995, 2000, 1955, 1964,
       1991, 2022, 1966, 1971, 1962, 2002, 1952, 1970, 1950, 1954, 1985,
       2003, 1961, 2019, 2001, 2004, 2011, 2021, 2010, 1959, 2015, 2020,
       1974, 1958, 1963, 1965, 1989, 2007, 1951], dtype=int64)
```

[163]:
```python
from sklearn.preprocessing import StandardScaler

cat_features = [feature for feature in data.columns if data[feature].dtype == 'O']

data1 = pd.get_dummies(cat_features)
data1
```

[163]: —

[164]:
```python
data1.info()
```

[166]:
```python
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler


cat_features = [feature for feature in data.columns if data[feature].dtype == 'O']

data1 = pd.get_dummies(data, columns=cat_features)

scaled_data = pd.concat([data, data1], axis=1)

print(scaled_data.shape)
print()
print('*' * 70)

scaled_data.head()
```

```
(1000, 16)

**********************************************************************
```

[166]:

	Square_Footage	Num_Bedrooms	Num_Bathrooms	Year_Built	Lot_Size	Garage_Size	Neighborhood_Quality	House_Price	Square_Footage	Num_Bedrooms	Num_E
0	1360	2	1	1981	0.599637	0	5	2.623829e+05	1360	2	
1	4272	3	3	2016	4.753014	1	6	9.852609e+05	4272	3	
2	3592	1	2	2016	3.634823	0	9	7.779774e+05	3592	1	
3	966	1	2	1977	2.730667	1	8	2.296989e+05	966	1	
4	4926	2	1	1993	4.699073	0	8	1.041741e+06	4926	2	

[167]:
```python
data.columns
```

[167]:
```
Index(['Square_Footage', 'Num_Bedrooms', 'Num_Bathrooms', 'Year_Built',
       'Lot_Size', 'Garage_Size', 'Neighborhood_Quality', 'House_Price'],
```

6: Data Reduction

Dimensionality Reduction

PCA (Principal Component Analysis)

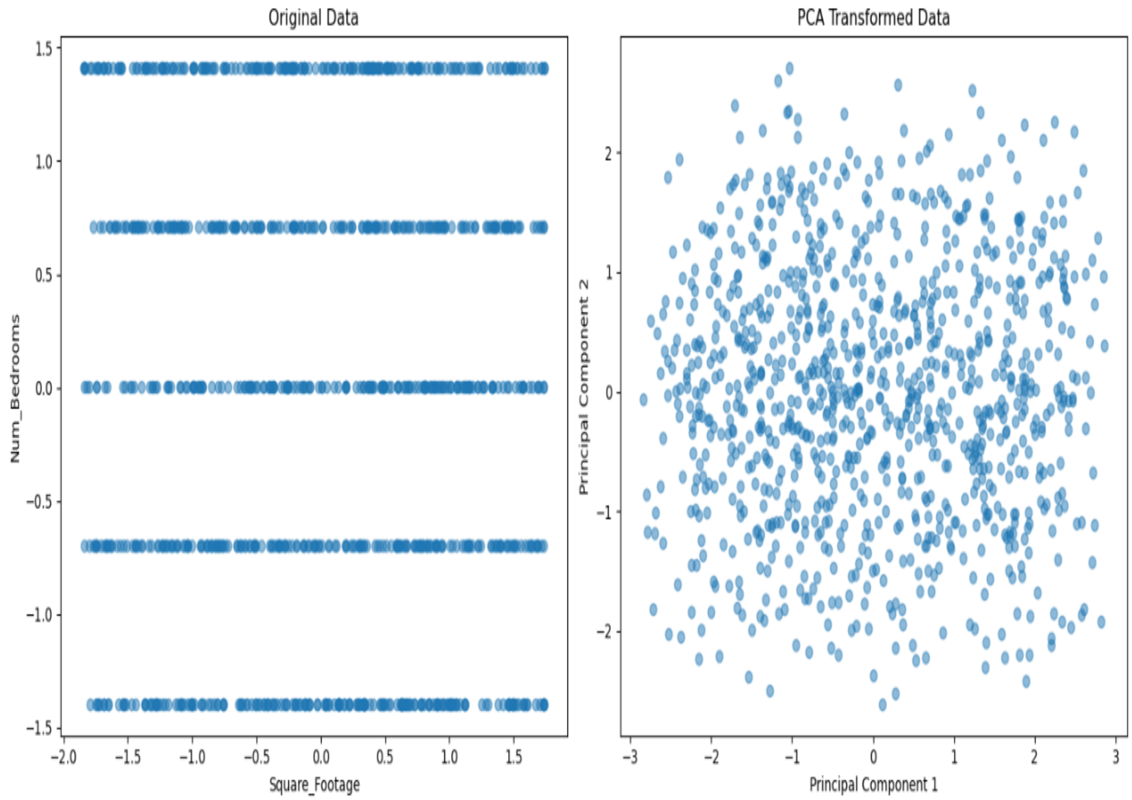
[170]:
```
scaled_data.shape
```

[170]:
```
(1000, 16)
```

•[171]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

data.fillna(data.mean(numeric_only=True), inplace=True)
cat_features = [feature for feature in data.columns if data[feature].dtype == 'O']
numeric_features = [feature for feature in data.columns if data[feature].dtype != 'O']
data = pd.get_dummies(data, columns=cat_features)

scaler = StandardScaler()
data[numeric_features] = scaler.fit_transform(data[numeric_features].values)

pca = PCA(n_components=8)
data_pca = pca.fit_transform(data)

print(data_pca.shape)
print(data_pca[:5])

plt.figure(figsize=(14, 6))
plt.subplot(1, 2, 1)
plt.scatter(data[numeric_features[0]], data[numeric_features[1]], alpha=0.5)
plt.title('Original Data')
plt.xlabel(numeric_features[0])
```

```
plt.figure(figsize=(14, 6))
plt.subplot(1, 2, 1)
plt.scatter(data[numeric_features[0]], data[numeric_features[1]], alpha=0.5)
plt.title('Original Data')
plt.xlabel(numeric_features[0])
plt.ylabel(numeric_features[1])
pca = PCA(n_components=8)
data_pca = pca.fit_transform(data)
plt.subplot(1, 2, 2)
plt.scatter(data_pca[:, 0], data_pca[:, 1], alpha=0.5)
plt.title('PCA Transformed Data')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.tight_layout()
plt.show()
```

```
 [-2.06202693  0.03520243 -0.54716971 -1.55338003 -0.86740338  1.25907066
  -0.27940462 -0.0224252 ]
 [-0.93901192  2.28428342 -0.41162975 -1.17644248  0.08214092  0.69440791
  -0.07494165  0.03658289]
 [ 2.08917476  0.96272009 -1.01831735  0.03808287  0.43449578 -0.45853932
  -0.80363536 -0.02987502]
 [-2.54104601  1.79894453 -0.54939605  0.47763899  0.572957    1.16137808
   0.38636805  0.03046095]]
```




```
[172]: type(data_pca)
```

```
[172]: numpy.ndarray
```

7: Handling Imbalanced Data

- Resampling Techniques
- Oversampling

[175]:
```
data.House_Price.value_counts(True)
```

[175]:
```
House_Price
2.623829e+05    0.001
2.501235e+05    0.001
1.021135e+06    0.001
8.343286e+05    0.001
1.040389e+06    0.001
                ...
3.584584e+05    0.001
2.643951e+05    0.001
7.495713e+05    0.001
2.637609e+05    0.001
7.425993e+05    0.001
Name: proportion, Length: 1000, dtype: float64
```

[176]:
```
data.shape
```

[176]:
```
(1000, 8)
```

8: Splitting Data

[184]:
```
from sklearn.model_selection import train_test_split

X = data.drop('Year_Built', axis=1)
y = data['House_Price']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

[184]:
```
((700, 7), (300, 7), (700,), (300,))
```

Conclusion

The house price data preprocessing project successfully transformed a raw, unstructured dataset into a clean and structured format suitable for machine learning and statistical analysis. Through systematic steps—including missing value imputation, outlier handling, categorical encoding, feature engineering, and scaling—the data was refined to remove noise and enhance the quality of information available to predictive models.

The preprocessing pipeline not only improved data consistency and integrity but also uncovered important insights about the features that influence house prices. By creating new meaningful variables and carefully selecting relevant features, the dataset is now better aligned with the assumptions and requirements of various regression and classification algorithms.

Overall, this preprocessing workflow lays a solid foundation for building accurate and robust

predictive models. With a well-prepared dataset, future steps can confidently focus on training, evaluating, and optimizing machine learning models to forecast house prices with greater precision.

PROJECT 2

Extrovert vs. Introvert Behavior Data

Summary

This machine learning project focuses on classifying individuals based on various attributes related to their personality. Personality prediction has applications in psychology, HR analytics, and recommendation systems. The classification pipeline includes data loading, preprocessing (handling missing values, encoding, scaling), model training (Logistic Regression and Random Forest), and evaluation using classification metrics and visual tools. This project illustrates the complete life cycle of a classification problem, from raw data to insights. It emphasizes the importance of proper preprocessing, model tuning, and rigorous evaluation in achieving reliable predictions.

The models developed here are scalable and adaptable to various real-world problems where personality classification or other label-based prediction tasks are needed.

Objectives

1. **Understand and prepare the dataset** for ML tasks.
2. **Treat missing data** using statistical imputation.
3. **Encode categorical variables** to numeric formats suitable for ML.
4. **Standardize features** to bring all variables to a common scale.
5. **Split the dataset** into train and test sets to prevent overfitting.
6. **Train and evaluate** classification models.
7. **Interpret model results** using reports and confusion matrices.

Abstract

Classification is a fundamental problem in supervised machine learning. In this project, the task is to predict a categorical target variable — *Personality type* — using several input features. The process involves thorough data preprocessing followed by training two commonly used classification models: **Logistic Regression** and **Random Forest**.

These models are selected for their simplicity and effectiveness in handling structured/tabular data. Evaluation is done using statistical performance measures to understand their accuracy and robustness.

The primary objective of this study is to build, train, and evaluate predictive models that can classify individuals into predefined personality categories based on input features. To achieve this, a step-by-step machine learning pipeline was implemented, starting with data ingestion and followed by cleaning, transformation, and modeling.

The dataset used in this project included both categorical and numerical variables, many of which required preprocessing. Missing values were handled using **statistical imputation**, and categorical variables were encoded using **Label Encoding**.

Numerical features were scaled using **StandardScaler** to standardize the input space and enhance model convergence and performance.

Explanation of Each Step

Step 1: Load Libraries and Dataset

- Python libraries for data manipulation (pandas), visualization (matplotlib, seaborn), and ML (sklearn) are imported.
- The dataset is read using pd.read_csv()pandas is essential for tabular data manipulation.

1. Load Libraries and Dataset

```
[19]: import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay
import seaborn as sns
import matplotlib.pyplot as plt

df = pd.read_csv("personality_dataset.csv")
```

Step 2: Initial Data Inspection and Class Balance

- df.head(), df.info(), and df.describe() are used to get a sense of the data.
- Class distribution is checked using .value_counts().

- Understanding the dataset is the first step. You look for:
  - **Missing values**
  - **Class imbalance** (which can lead to biased models)
  - **Feature types** (numerical or categorical)

2. Initial Data Inspection and Class Balance

```
[5]: print("Class distribution:")
print(df['Personality'].value_counts(normalize=True) * 100)
```

```
Class distribution:
Personality
Extrovert    51.413793
Introvert    48.586207
Name: proportion, dtype: float64
```

Step 3: Handle Missing Values

- Missing values are replaced with the most frequent value in each column.
- Missing data can skew the model if not handled.
- Common strategies:
- **Mean/median imputation** for numeric features
- **Mode (most frequent)** for categorical features

3. Handle Missing Values

```
[8]: num_cols = df.select_dtypes(include=['float64']).columns
     cat_cols = ['Stage_fear', 'Drained_after_socializing']

     num_imputer = SimpleImputer(strategy='mean')
     df[num_cols] = num_imputer.fit_transform(df[num_cols])

     cat_imputer = SimpleImputer(strategy='most_frequent')
     df[cat_cols] = cat_imputer.fit_transform(df[cat_cols])
```

Step 4: Encode Categorical Features

- LabelEncoder converts string labels into integer values.
- Machine learning models require **numerical input**.
- Encoding transforms:
    - ["Male", "Female"] → [0, 1]
    - ["Introvert", "Extrovert", "Ambivert"] → [0, 1, 2]

4. Encode Categorical Features

```
[11]: label_encoders = {}
      for col in cat_cols + ['Personality']:
          le = LabelEncoder()
          df[col] = le.fit_transform(df[col])
          label_encoders[col] = le
```

Step 5: Feature Scaling and Data Split

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

- Features are standardized to have mean = 0 and standard deviation = 1.
- The dataset is split into 70–80% training and 20–30% testing.
- Many ML models (e.g., Logistic Regression) assume that features are on the **same scale**.
- Standardization helps models converge faster and perform better.
- Data splitting ensures **generalization** and prevents **data leakage**.

5. Feature Scaling and Data Split

```
[14]: X = df.drop('Personality', axis=1)
      y = df['Personality']

      scaler = StandardScaler()
      X_scaled = scaler.fit_transform(X)

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

Step 6: Train ML Models

✅ ***Logistic Regression***
model = LogisticRegression()
model.fit(X_train, y_train)

✅ ***Random Forest***
rf_model = RandomForestClassifier()
rf_model.fit(X_train, y_train)

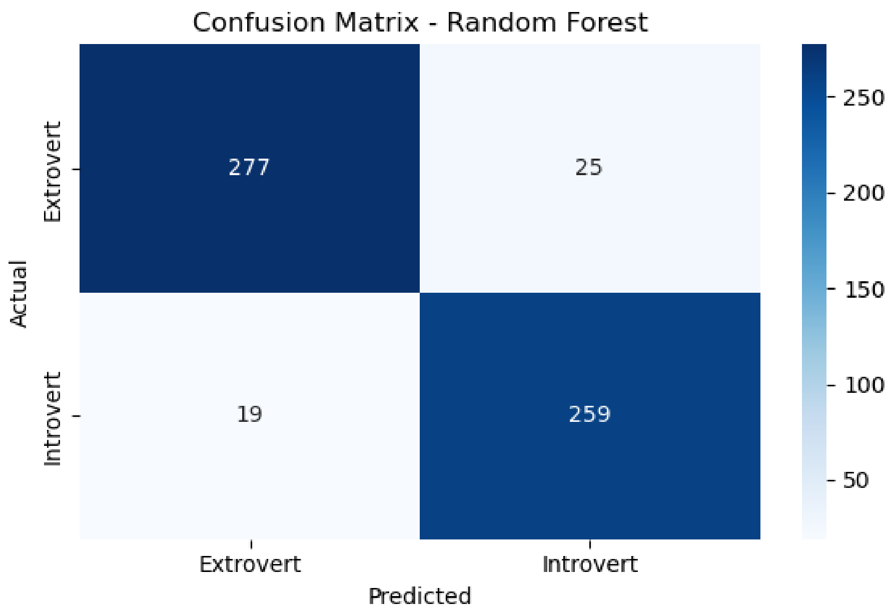
- **Logistic Regression**:
  - A linear model used for binary and multiclass classification.
  - Uses the **sigmoid function** to model probability outputs.
  - Suitable for linearly separable classes.
- **Random Forest**:
  - An **ensemble learning** method using multiple decision trees.
  - Introduces **randomness** during training for better generalizati.

6. Random Forest Classifier

[25]:
```
rf_model = RandomForestClassifier(random_state=42)
rf_model.fit(X_train, y_train)
rf_score = rf_model.score(X_test, y_test)
print(f"Random Forest Accuracy: {rf_score:.4f}")

rf_preds = rf_model.predict(X_test)
cm_rf = confusion_matrix(y_test, rf_preds)

plt.figure(figsize=(6, 4))
sns.heatmap(cm_rf, annot=True, fmt='d', cmap='Blues',
            xticklabels=label_encoders['Personality'].classes_,
            yticklabels=label_encoders['Personality'].classes_)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix - Random Forest')
plt.tight_layout()
plt.show()
```

```
Random Forest Accuracy: 0.9241
```


```
Random Forest Accuracy: 0.9241
```

Step 7: Evaluate Models

print(classification_report(y_test, y_pred))
ConfusionMatrixDisplay(confusion_matrix(y_test, y_pred)).plot()

Key Metrics:

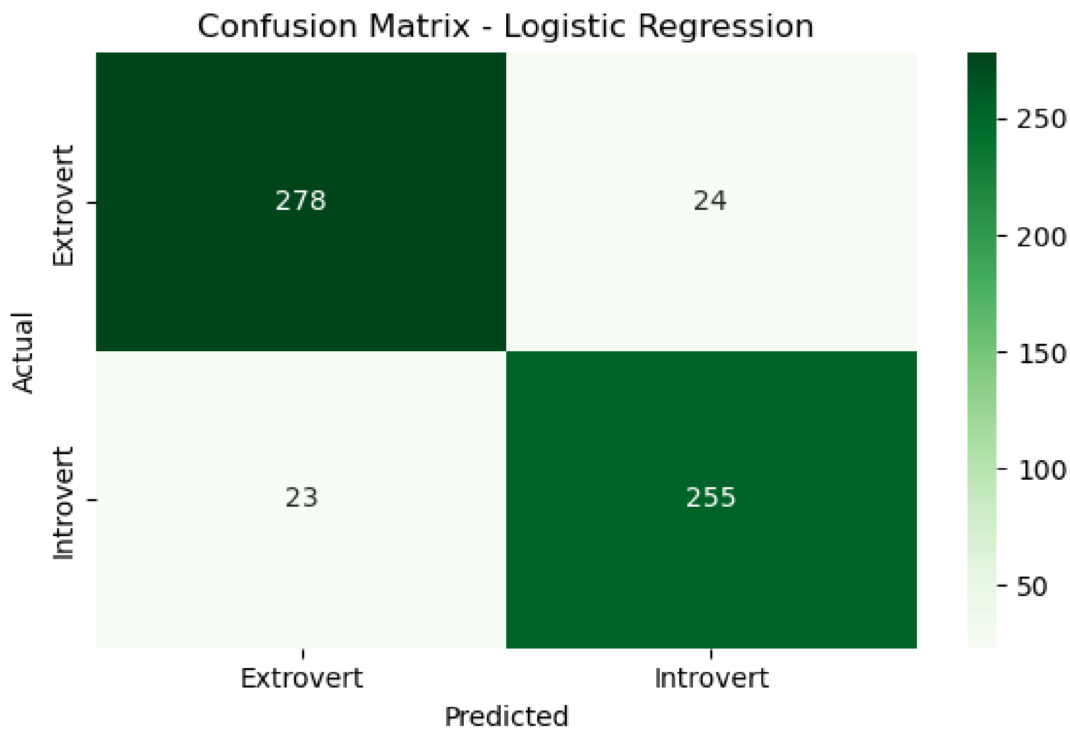
- **Accuracy**: (TP + TN) / Total
- **Precision**: TP / (TP + FP) – correctness of positive predictions.
- **Recall**: TP / (TP + FN) – ability to find all positives.
- **F1 Score**: Harmonic mean of precision and recall.
- **Confusion Matrix**:
    - 2D matrix to visualize model prediction vs. true values.
    - Helps identify types of errors (false positives, false negatives).

7. Logistic Regression Classifier

[28]:
```
lr_model = LogisticRegression(random_state=42)
lr_model.fit(X_train, y_train)
lr_score = lr_model.score(X_test, y_test)
print(f"Logistic Regression Accuracy: {lr_score:.4f}")

lr_preds = lr_model.predict(X_test)

cm_lr = confusion_matrix(y_test, lr_preds)

plt.figure(figsize=(6, 4))
sns.heatmap(cm_lr, annot=True, fmt='d', cmap='Greens',
            xticklabels=label_encoders['Personality'].classes_,
            yticklabels=label_encoders['Personality'].classes_)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix - Logistic Regression')
plt.tight_layout()
plt.show()
```

```
Logistic Regression Accuracy: 0.9190
```

```
Logistic Regression Accuracy: 0.9190
```



8. ANN

[33]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical

y_train_oh = to_categorical(y_train)
y_test_oh = to_categorical(y_test)

model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(X_train.shape[1],)))
model.add(Dense(32, activation='relu'))
model.add(Dense(2, activation='softmax'))

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model.fit(X_train, y_train_oh, epochs=50, batch_size=32, validation_data=(X_test, y_test_oh))

loss, accuracy = model.evaluate(X_test, y_test_oh)
print(f"ANN Accuracy (Keras): {accuracy:.4f}")
```

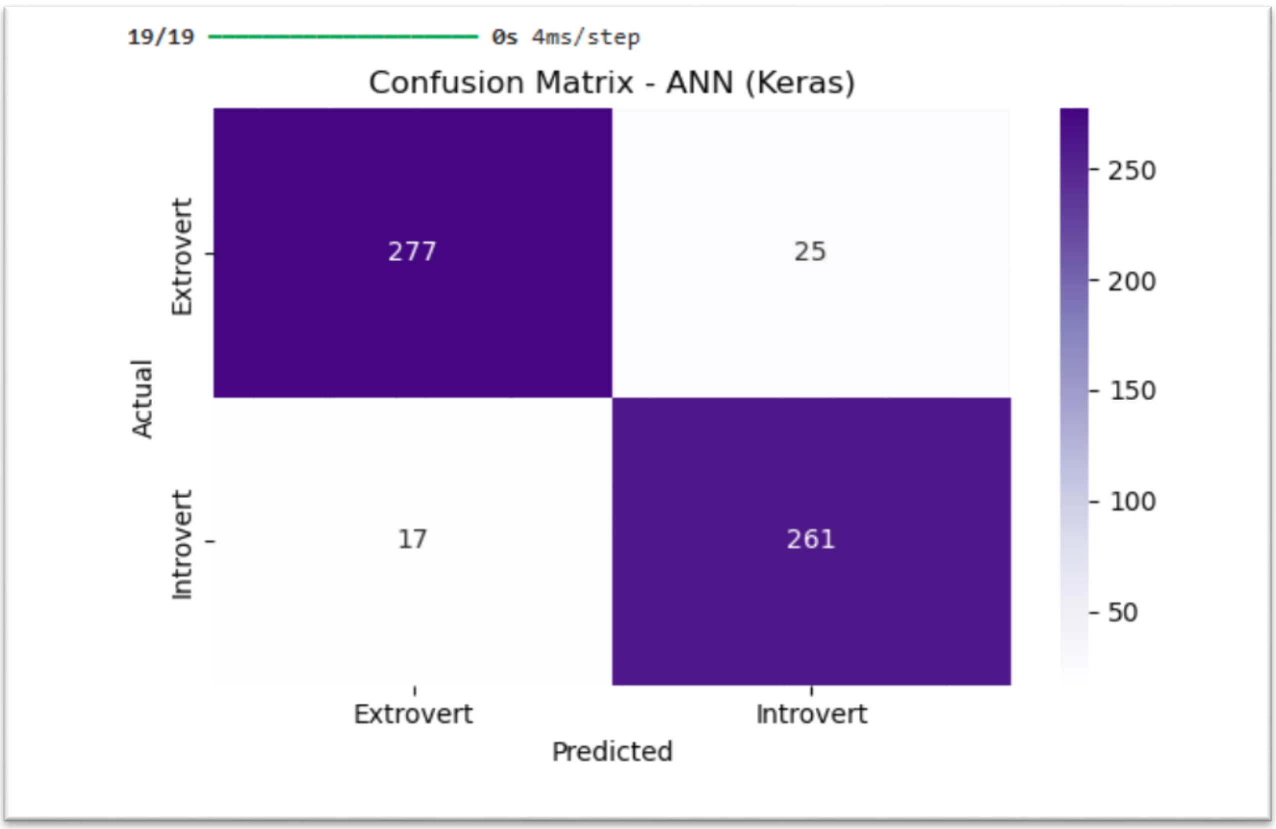
```
Epoch 1/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 5s 12ms/step - accuracy: 0.8407 - loss: 0.4218 - val_accuracy: 0.9293 - val_loss: 0.2645
Epoch 2/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9330 - loss: 0.2567 - val_accuracy: 0.9293 - val_loss: 0.2555
Epoch 3/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9347 - loss: 0.2398 - val_accuracy: 0.9293 - val_loss: 0.2560
Epoch 4/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9356 - loss: 0.2313 - val_accuracy: 0.9293 - val_loss: 0.2525
Epoch 5/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9282 - loss: 0.2455 - val_accuracy: 0.9293 - val_loss: 0.2517
Epoch 6/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9390 - loss: 0.2187 - val_accuracy: 0.9293 - val_loss: 0.2462
Epoch 7/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9417 - loss: 0.2033 - val_accuracy: 0.9293 - val_loss: 0.2455
Epoch 8/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9193 - loss: 0.2596 - val_accuracy: 0.9293 - val_loss: 0.2474
Epoch 9/50
```

```
Epoch 10/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9322 - loss: 0.2205 - val_accuracy: 0.9293 - val_loss: 0.2433
Epoch 11/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9334 - loss: 0.2231 - val_accuracy: 0.9293 - val_loss: 0.2396
Epoch 12/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9385 - loss: 0.2085 - val_accuracy: 0.9293 - val_loss: 0.2401
Epoch 13/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9369 - loss: 0.2110 - val_accuracy: 0.9276 - val_loss: 0.2401
Epoch 14/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9363 - loss: 0.2089 - val_accuracy: 0.9276 - val_loss: 0.2403
Epoch 15/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9354 - loss: 0.2140 - val_accuracy: 0.9276 - val_loss: 0.2401
Epoch 16/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9363 - loss: 0.2241 - val_accuracy: 0.9276 - val_loss: 0.2381
Epoch 17/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9332 - loss: 0.2179 - val_accuracy: 0.9276 - val_loss: 0.2391
Epoch 18/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9378 - loss: 0.2090 - val_accuracy: 0.9293 - val_loss: 0.2436
Epoch 19/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9407 - loss: 0.2026 - val_accuracy: 0.9293 - val_loss: 0.2407
Epoch 20/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9397 - loss: 0.2000 - val_accuracy: 0.9293 - val_loss: 0.2395
Epoch 21/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9423 - loss: 0.1908 - val_accuracy: 0.9293 - val_loss: 0.2407
Epoch 22/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9366 - loss: 0.2074 - val_accuracy: 0.9293 - val_loss: 0.2412
Epoch 23/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9274 - loss: 0.2202 - val_accuracy: 0.9276 - val_loss: 0.2382
Epoch 24/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9320 - loss: 0.2079 - val_accuracy: 0.9276 - val_loss: 0.2363
Epoch 25/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9372 - loss: 0.2072 - val_accuracy: 0.9293 - val_loss: 0.2399
Epoch 26/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - accuracy: 0.9316 - loss: 0.2123 - val_accuracy: 0.9293 - val_loss: 0.2383
Epoch 27/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9432 - loss: 0.1863 - val_accuracy: 0.9293 - val_loss: 0.2379
Epoch 28/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9373 - loss: 0.2038 - val_accuracy: 0.9293 - val_loss: 0.2398
Epoch 29/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9292 - loss: 0.2121 - val_accuracy: 0.9276 - val_loss: 0.2426
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9325 - loss: 0.2040 - val_accuracy: 0.9276 - val_loss: 0.2389
Epoch 34/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9281 - loss: 0.2195 - val_accuracy: 0.9293 - val_loss: 0.2404
Epoch 35/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9311 - loss: 0.2108 - val_accuracy: 0.9293 - val_loss: 0.2376
Epoch 36/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9365 - loss: 0.1941 - val_accuracy: 0.9293 - val_loss: 0.2395
Epoch 37/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.9366 - loss: 0.1961 - val_accuracy: 0.9276 - val_loss: 0.2377
Epoch 38/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9358 - loss: 0.1972 - val_accuracy: 0.9276 - val_loss: 0.2397
Epoch 39/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.9320 - loss: 0.2057 - val_accuracy: 0.9276 - val_loss: 0.2377
Epoch 40/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9328 - loss: 0.1972 - val_accuracy: 0.9293 - val_loss: 0.2359
Epoch 41/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9326 - loss: 0.1990 - val_accuracy: 0.9276 - val_loss: 0.2372
Epoch 42/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9358 - loss: 0.1985 - val_accuracy: 0.9276 - val_loss: 0.2438
Epoch 43/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9323 - loss: 0.1971 - val_accuracy: 0.9276 - val_loss: 0.2367
Epoch 44/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9435 - loss: 0.1851 - val_accuracy: 0.9293 - val_loss: 0.2392
Epoch 45/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9350 - loss: 0.2031 - val_accuracy: 0.9293 - val_loss: 0.2425
Epoch 46/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9323 - loss: 0.2098 - val_accuracy: 0.9276 - val_loss: 0.2480
Epoch 47/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9329 - loss: 0.2072 - val_accuracy: 0.9276 - val_loss: 0.2344
Epoch 48/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9418 - loss: 0.1759 - val_accuracy: 0.9293 - val_loss: 0.2356
Epoch 49/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9382 - loss: 0.1890 - val_accuracy: 0.9276 - val_loss: 0.2368
Epoch 50/50
73/73 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9374 - loss: 0.1987 - val_accuracy: 0.9276 - val_loss: 0.2378
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9296 - loss: 0.2243
ANN Accuracy (Keras): 0.9276
```

ANN Accuracy (Keras): 0.9276

[43]:
```python
import numpy as np
ann_probs = model.predict(X_test)

ann_preds = np.argmax(ann_probs, axis=1)

cm_ann = confusion_matrix(y_test, ann_preds)

plt.figure(figsize=(6, 4))
sns.heatmap(cm_ann, annot=True, fmt='d', cmap='Purples',
            xticklabels=label_encoders['Personality'].classes_,
            yticklabels=label_encoders['Personality'].classes_)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix - ANN (Keras)')
plt.tight_layout()
plt.show()
```

19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step



Conclusion

This ML classification project demonstrates:

- A complete **end-to-end pipeline** for handling real-world data.
- How to **preprocess, model, and evaluate** a dataset.
- The use of **Logistic Regression** for simple, interpretable results.
- The application of **Random Forest** for more robust, nonlinear relationships.

The results, especially from the confusion matrix and classification report, help determine which model generalizes better. This approach is repeatable across various classification problems.

This project aimed to develop a predictive classification model to determine an individual's personality type using structured data. Through a systematic machine learning pipeline, we successfully addressed challenges typically encountered in real-world datasets and leveraged the power of supervised learning algorithms to make informed predictions.

This project successfully demonstrates the end-to-end implementation of a classification problem using machine learning. It highlights the importance of data preprocessing, careful model selection, and performance evaluation. Beyond this project, the same framework can be adapted to a wide range of classification tasks across various domains.

By combining data science best practices with domain understanding, we create models that are not only accurate but also actionable and reliable.