Classification

December 5, 2025

1 Deep Analysis of an Automotive MPG Classification Model

This notebook documents the final classification model used to predict the `mpg_01` (a binary classification derived from continuous MPG) based on various automotive features.

The core strategy involves a **K-Nearest Neighbors ($K = 1$)** classifier combined with a specialized preprocessing pipeline (**V13 Preprocessor**) designed to optimize feature representation.

1.1 1. Setup and Data Loading

The initial setup imports all necessary libraries and loads the training and testing datasets.

[1]:
```
import pandas as pd
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
```

1.1.1 Libraries

- **`pandas` & `numpy`**: Standard libraries for data manipulation and numerical operations.
- **`KNeighborsClassifier`**: The core classification algorithm used. It's an instance-based, non-parametric algorithm that classifies a new point based on the majority class of its K nearest neighbors in the feature space.
- **`Pipeline`**: A sequential estimator that chains multiple steps (like preprocessing and modeling). This is crucial for keeping the data transformations consistent and preventing **data leakage**.
- **`ColumnTransformer`**: Allows different preprocessing steps to be applied to different subsets of columns simultaneously.
- **`StandardScaler`**: Scales features to have a mean of 0 and a standard deviation of 1. This is vital for distance-based algorithms like K-NN, which are sensitive to the scale of features.
- **`OneHotEncoder`**: Converts categorical features (like `origin`) into a numerical format suitable for modeling.
- **`SimpleImputer`**: Handles missing values by replacing them with a specified statistic, like the median.

1.2 2. Data Loading, Cleaning, and Target Preparation

This section loads the data, performs basic cleaning, and prepares the target variable.

[2]:
```
# --- 1. Load Data ---
df_train = pd.read_csv("train.csv")
df_test = pd.read_csv("test.csv")
test_ids = df_test['ID']

# Clean Data
for df in [df_train, df_test]:
    df['horsepower'] = pd.to_numeric(df['horsepower'], errors='coerce')
    df.rename(columns={'year': 'model_year'}, inplace=True)

# Prepare Target
df_train['mpg_01'] = df_train['mpg01']

# Separate X and y
X = df_train.drop(columns=['mpg', 'mpg01', 'mpg_01', 'name', 'ID'],␣
 ↪errors='ignore')
y = df_train['mpg_01']
```

1.2.1 Data Preparation

1. **Data Cleaning:** The `horsepower` column, likely imported as a string due to inconsistent entries, is explicitly converted to a numeric type. The argument `errors='coerce'` handles non-numeric values (like '?' or '–') by setting them to `NaN`, which is then handled by the `SimpleImputer` in the pipeline.
2. **Feature Renaming:** `year` is renamed to `model_year` for clarity.
3. **Target Variable:** The target variable for this binary classification task is `mpg_01` (derived from the original continuous `mpg`). This variable is separated from the features X.
4. **Feature Exclusion:** Columns like `mpg`, `mpg01`, `name`, and `ID` are dropped from the feature matrix X as they are either the original continuous target, redundant binary target, a unique identifier, or a non-predictive text identifier. This is good practice to ensure the model learns from the relevant features.

1.3 3. V13 Preprocessor: Feature Engineering Pipeline

This is the 13th revision of my code. The **V13 Preprocessor** is defined using `ColumnTransformer` to apply specific transformations to different feature groups. This sophisticated preprocessing strategy is key to the model's performance.

[3]:
```
# --- 2. V13 Preprocessor (V7 Hybrid Base + Scaled Model Year) ---
# New list of features to be scaled
```

```
scaled_features = ['displacement', 'horsepower', 'weight', 'acceleration',␣
 ↪'model_year']

# Preprocessor
prep_v13 = ColumnTransformer(
    transformers=[
        # Scaling continuous + model_year
        ('scale', Pipeline([('imp', SimpleImputer(strategy='median')), ('scl',␣
 ↪StandardScaler())]),
         scaled_features),
        # Passing cylinders raw (V7's key to success)
        ('pass', SimpleImputer(strategy='median'), ['cylinders']),
        # One-Hot Encoding only origin
        ('cat', OneHotEncoder(handle_unknown='ignore', sparse_output=False),␣
 ↪['origin'])
    ], remainder='drop'
)
```

1.3.1 Preprocessor Logic

The `ColumnTransformer` handles three distinct sets of features:

1. **`'scale'` (Scaled Features):**
   - **Features:** `displacement`, `horsepower`, `weight`, `acceleration`, and, significantly, `model_year`.
   - **Transformation:** A small `Pipeline` is applied: first, `SimpleImputer(strategy='median')` handles any remaining missing values in these numerical features by replacing them with the column's median. Second, `StandardScaler()` standardizes the features.
   - **Significance:** Scaling is **critical for K-NN**. By including `model_year` in the scaled features, we treat time as a continuous, distance-sensitive variable, assuming that cars from adjacent years are more similar than those separated by decades.
2. **`'pass'` (Cylinders - Passed Raw):**
   - **Features:** `cylinders`.
   - **Transformation:** Only `SimpleImputer(strategy='median')` is applied. Crucially, **it is NOT scaled**.
   - **Significance:** The note "(V7's key to success)" suggests that the number of cylinders has an inherently strong predictive power based on its **raw integer value**. Scaling it might diminish the significance of the categorical/ordinal distance between, say, 4 and 8 cylinders, which is a major engineering difference. Keeping it raw preserves this strong, unscaled relationship.
3. **`'cat'` (One-Hot Encoded):**
   - **Features:** `origin`.
   - **Transformation:** `OneHotEncoder` converts the categories (e.g., USA, Europe, Japan) into binary columns.
   - **Significance:** This is the correct way to handle nominal categorical features, ensuring the model doesn't assume an arbitrary ordinal relationship between countries of origin.

1.4 4. Model Definition and Training

The model is defined as a `Pipeline` combining the robust preprocessor with the chosen classifier.

[4]:
```
# --- 3. Model: K=1 Nearest Neighbors (The Proven Core) ---

final_model = Pipeline([
    ('prep', prep_v13),
    # Back to K=1, as this yielded the highest score
    ('knn', KNeighborsClassifier(n_neighbors=1, p=2))
])

# Fit on the full training data
final_model.fit(X, y)
```

[4]:
```
Pipeline(steps=[('prep',
                 ColumnTransformer(transformers=[('scale',
                                                  Pipeline(steps=[('imp',
SimpleImputer(strategy='median')),
                                                                  ('scl',
StandardScaler())]),
                                                  ['displacement', 'horsepower',
                                                   'weight', 'acceleration',
                                                   'model_year']),
                                                 ('pass',
SimpleImputer(strategy='median'),
                                                  ['cylinders']),
                                                 ('cat',
OneHotEncoder(handle_unknown='ignore',
sparse_output=False),
                                                  ['origin'])])),
                ('knn', KNeighborsClassifier(n_neighbors=1))])
```

1.4.1 Model

1. **Pipeline Integration:** The entire process is encapsulated in a single `Pipeline`. This ensures that when the model is fitted or used for prediction, the test data automatically goes through the exact same `prep_v13` transformations (Imputation, Scaling, One-Hot Encoding) as the training data, preventing transformation errors.
2. **K-Nearest Neighbors ($K = 1$):**
   - **Classifier:** `KNeighborsClassifier` is chosen.
   - **Parameter `n_neighbors=1`:** This is the core strategy, making it a "Nearest Neighbor" classifier. For any new data point, the prediction is simply the class of its **single closest neighbor** in the feature space.
   - **Parameter `p=2`:** This specifies the distance metric as **Euclidean Distance** (Minkowski distance with $p = 2$). This is the standard distance metric for many K-NN applications and aligns with the use of `StandardScaler`.

- **Significance of $K = 1$:** The note indicates this value yielded the **highest score**. In a dataset where the class boundaries are highly non-linear or where the clusters are tightly defined, $K = 1$ can be highly effective, acting almost like an exact lookup table for previous observations. It's the most sensitive to local data structure but can also be susceptible to noise (high variance).

1.5 5. Submission Generation

The final steps involve preparing the test data, generating predictions, and saving the results.

[5]:
```
# --- 4. Generate Submission ---

# Use features from test.csv
X_test = df_test.drop(columns=['ID', 'name'], errors='ignore')
final_predictions = final_model.predict(X_test)

df_submission = pd.DataFrame({
    'ID': test_ids,
    'mpg_01': final_predictions.astype(int)
})

submission_file = "solution.csv"
df_submission.to_csv(submission_file, index=False)

print(f" Final submission file '{submission_file}' created using K=1 with␣
 ↪Scaled Year.")
print(df_submission.head())
```

```
 Final submission file 'solution.csv' created using K=1 with Scaled Year.
                                      ID  mpg_01
0  70_chevrolet chevelle malibu_alpha_3505       0
1          71_buick skylark 320_bravo_3697       0
2       70_plymouth satellite_charlie_3421       0
3              68_amc rebel sst_delta_3418       0
4                 70_ford torino_echo_3444       0
```

1.5.1 Prediction and Output

1. **Test Data Preparation:** The *X_test* feature matrix is created by dropping the non-predictive `ID` and `name` columns, mirroring the cleaning done on the training data.
2. **Prediction:** The `final_model.predict(X_test)` call automatically applies the entire `prep_v13` pipeline to the test data *before* feeding it to the $K = 1$ classifier. This consistent transformation is why using the `Pipeline` object is so important.
3. **Submission Formatting:** The predictions are combined with the original `test_ids` into a DataFrame. The predictions are explicitly cast to `int` to ensure the final output is in the expected integer format (0 or 1).

4. **Output:** The predictions are saved to `solution.csv`, ready for submission. The print statement confirms the successful execution and the key model parameters used.