In [ ]:
```
# A2 - Predicting Car Prices
# Name: Alston Alvares     Student ID: st126488

import os
import json
import joblib
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import mlflow
import mlflow.sklearn

from sklearn.model_selection import train_test_split, KFold
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder, PolynomialFeatures
from sklearn.metrics import mean_squared_error
```

In [8]:
```
# Utility functions

def r2(y_true, y_pred):
    y_true = np.asarray(y_true)
    y_pred = np.asarray(y_pred)
    ss_res = np.sum((y_true - y_pred) ** 2)
    ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
    return 1 - ss_res / ss_tot


# Custom LinearRegression

from custom_linear import LinearRegression, r2, plot_feature_importance


# Data loading & preprocessing

DATA_PATH = "Cars.csv"
df = pd.read_csv(DATA_PATH)
df.columns = df.columns.str.lower()

# Owner mapping
owner_map = {
    'First Owner': 1, 'Second Owner': 2, 'Third Owner': 3,
    'Fourth & Above Owner': 4, 'Test Drive Car': 5
}
df['owner'] = df['owner'].astype(str).str.strip().replace(owner_map)

# Remove CNG/LPG entries
df = df[~df['fuel'].isin(['CNG', 'LPG'])]
df = df[df['owner'] != 5]  # remove test drive cars

# Convert numeric fields
df['mileage'] = pd.to_numeric(df['mileage'].astype(str).str.extract(r'([\d\.]+)')[0
```

```
# Extract brand from name
df['brand'] = df['name'].astype(str).str.split().str[0]

# Target variable
y = np.log(df['selling_price'])
X = df[['year', 'km_driven', 'mileage', 'owner', 'brand']]

# Drop rows with NA
df_model = pd.concat([X, y], axis=1).dropna()
X = df_model.drop(columns=['selling_price']).reset_index(drop=True)
y = df_model['selling_price'].reset_index(drop=True)

print("Dataset prepared. Shape:", X.shape)
```

```
Dataset prepared. Shape: (7814, 5)
```

```
C:\Users\alsto\AppData\Local\Temp\ipykernel_23512\2044250968.py:27: FutureWarning: D
owncasting behavior in `replace` is deprecated and will be removed in a future versi
on. To retain the old behavior, explicitly call `result.infer_objects(copy=False)`.
To opt-in to the future behavior, set `pd.set_option('future.no_silent_downcasting',
True)`
  df['owner'] = df['owner'].astype(str).str.strip().replace(owner_map)
```

In [9]:
```
# Train/test split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

num_cols = ['year', 'km_driven', 'mileage', 'owner']
cat_cols = ['brand']

# Base preprocessor
base_preprocessor = ColumnTransformer([
    ('num', Pipeline([
        ('imputer', SimpleImputer(strategy='median')),
        ('scaler', StandardScaler())
    ]), num_cols),
    ('cat', OneHotEncoder(handle_unknown='ignore', sparse_output=False), cat_cols)
], remainder='drop')

# Polynomial preprocessor
poly_preprocessor = ColumnTransformer([
    ('num', Pipeline([
        ('imputer', SimpleImputer(strategy='median')),
        ('poly', PolynomialFeatures(degree=2, include_bias=False)),
        ('scaler', StandardScaler())
    ]), num_cols),
    ('cat', OneHotEncoder(handle_unknown='ignore', sparse_output=False), cat_cols)
], remainder='drop')
```

In [ ]:
```
# Model selection & CV

if mlflow.active_run() is not None:
    mlflow.end_run()

# Initialize variables to keep track of the best performing model and its metrics.
best_mse = float('inf')
```

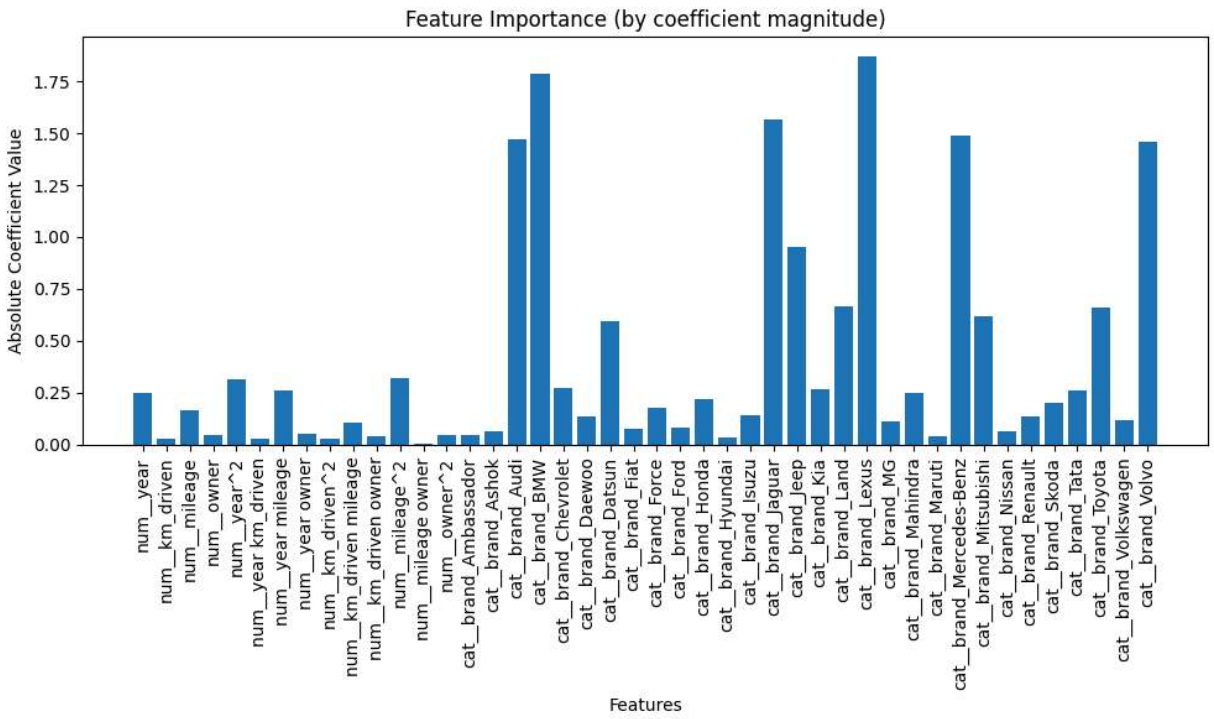
```
best_model = None
best_preprocessor = None
best_feature_names = None
best_params = None

# Define the hyperparameter search space for our linear regression model.
# These lists define the different model configurations to be tested.
reg_types = ['normal', 'lasso', 'ridge', 'polynomial']
momentums = [False, True]
gd_types = ['batch', 'mini', 'stochastic']
init_types = ['zeros', 'xavier']
lrs = [0.01, 0.001, 0.0001]

# Create a list to store the results of each cross-validation run for later analysi
all_cv_results = []

# Configure K-Fold cross-validation.
# We use a fixed random state to ensure the splits are reproducible across runs.
kf = KFold(n_splits=5, shuffle=True, random_state=42)


# Hyperparameter Tuning Loop ---
# This section iterates through all possible combinations of hyperparameters.
for reg in reg_types:
    for use_mom in momentums:
        for gd in gd_types:
            for init in init_types:
                for lr in lrs:
                    with mlflow.start_run():
                        # Map reg_type
                        if reg == 'normal':
                            reg_param = 'none'
                            preprocessor = base_preprocessor
                        elif reg == 'lasso':
                            reg_param = 'l1'
                            preprocessor = base_preprocessor
                        elif reg == 'ridge':
                            reg_param = 'l2'
                            preprocessor = base_preprocessor
                        elif reg == 'polynomial':
                            reg_param = 'none'
                            preprocessor = poly_preprocessor

                        mlflow.log_params({
                            'reg_type': reg, 'mapped_reg': reg_param,
                            'use_momentum': use_mom, 'gd_type': gd,
                            'init_type': init, 'lr': lr
                        })

                        # preprocess
                        X_train_preproc = preprocessor.fit_transform(X_train)
                        y_train_np = np.asarray(y_train)

                         # Lists to store performance metrics for each fold.
                        mse_scores, r2_scores = [], []
```

```
                    # Perform K-fold cross-validation.
                    for train_idx, val_idx in kf.split(X_train_preproc):
                        X_tr, X_val = X_train_preproc[train_idx], X_train_prepr
                        y_tr, y_val = y_train_np[train_idx], y_train_np[val_idx

                        # Initialize and train the linear regression model wit
                        model = LinearRegression(
                            lr=lr, num_iter=6000,
                            reg_type=reg_param, init_type=init,
                            use_momentum=use_mom, momentum=0.9,
                            gd_type=gd, batch_size=32
                        )
                        model.fit(X_tr, y_tr)

                        # Make predictions on the validation set.
                        y_val_pred = model.predict(X_val)

                        # Calculate and store the performance metrics for this
                        mse_cv = mean_squared_error(y_val, y_val_pred)
                        r2_cv = r2(y_val, y_val_pred)
                        mse_scores.append(mse_cv)
                        r2_scores.append(r2_cv)

                    # Calculate the average metrics across all folds.
                    avg_mse = float(np.mean(mse_scores))
                    avg_r2 = float(np.mean(r2_scores))

                    # Log the average metrics to MLflow for the current run.
                    mlflow.log_metric('cv_mse', avg_mse)
                    mlflow.log_metric('cv_r2', avg_r2)

                    # Store all results for comprehensive analysis outside the
                    all_cv_results.append({
                        'reg_type': reg,
                        'gd_type': gd,
                        'init_type': init,
                        'momentum': use_mom,
                        'lr': lr,
                        'cv_mse': avg_mse,
                        'cv_r2': avg_r2
                    })

                    # Check if the current model is the best performing one so
                    if avg_mse < best_mse:
                        best_mse = avg_mse
                        best_model = LinearRegression(
                            lr=lr, num_iter=6000,
                            reg_type=reg_param, init_type=init,
                            use_momentum=use_mom, momentum=0.9,
                            gd_type=gd, batch_size=32
                        )
                        best_model.fit(X_train_preproc, y_train_np)
                        best_preprocessor = preprocessor
                        # Safely get the feature names from the preprocessor.
                        try:
                            best_feature_names = preprocessor.get_feature_names
```

```
                except Exception:
                    best_feature_names = num_cols + ['OH_'+c for c in c
                # Save the parameters of the best model.
                best_params = {
                    'reg': reg, 'reg_param': reg_param,
                    'use_momentum': use_mom, 'gd_type': gd,
                    'init_type': init, 'lr': lr,
                    'cv_mse': avg_mse, 'cv_r2': avg_r2
                }
```

In [37]:
```
#plot feature importance
def plot_feature_importance(model, feature_names):
    coeffs = model.weights[1:]
    feature_names = list(feature_names)
    if len(coeffs) != len(feature_names):
        min_len = min(len(coeffs), len(feature_names))
        coeffs = coeffs[:min_len]
        feature_names = feature_names[:min_len]

    plt.figure(figsize=(10, 6))
    plt.bar(feature_names, np.abs(coeffs))
    plt.xlabel('Features')
    plt.ylabel('Absolute Coefficient Value')
    plt.title('Feature Importance (by coefficient magnitude)')
    plt.xticks(rotation=90)
    plt.tight_layout()
    plt.show()

print("\nPlotting feature importance for the best model...")
plot_feature_importance(best_model, best_feature_names)
```

```
Plotting feature importance for the best model...
```



In [11]:
```
# Final evaluation on test set using best model

X_train_best = best_preprocessor.fit_transform(X_train)
X_test_best = best_preprocessor.transform(X_test)
best_model.fit(X_train_best, np.asarray(y_train))

y_test_pred = best_model.predict(X_test_best)
test_mse = mean_squared_error(y_test, y_test_pred)
test_r2 = r2(y_test, y_test_pred)

print("\n=== Best Model Evaluation on Test Set ===")
print("Best params:", best_params)
print(f"Test MSE: {test_mse:.4f}")
print(f"Test R2: {test_r2:.4f}")

# log table artifacts
import os
os.makedirs("mlflow_artifacts", exist_ok=True)

# 1. CV results table
cv_results_df = pd.DataFrame(all_cv_results)
cv_csv_path = "mlflow_artifacts/cv_results.csv"
cv_results_df.to_csv(cv_csv_path, index=False)
mlflow.log_artifact(cv_csv_path)

# 2. Test evaluation table
test_eval_df = pd.DataFrame([{
    "test_mse": test_mse,
    "test_r2": test_r2,
    "reg_type": best_params['reg'],
    "reg_param": best_params['reg_param'],
    "gd_type": best_params['gd_type'],
    "init_type": best_params['init_type'],
    "momentum": best_params['use_momentum'],
    "lr": best_params['lr']
}])
test_eval_csv_path = "mlflow_artifacts/test_evaluation.csv"
test_eval_df.to_csv(test_eval_csv_path, index=False)
mlflow.log_artifact(test_eval_csv_path)
```

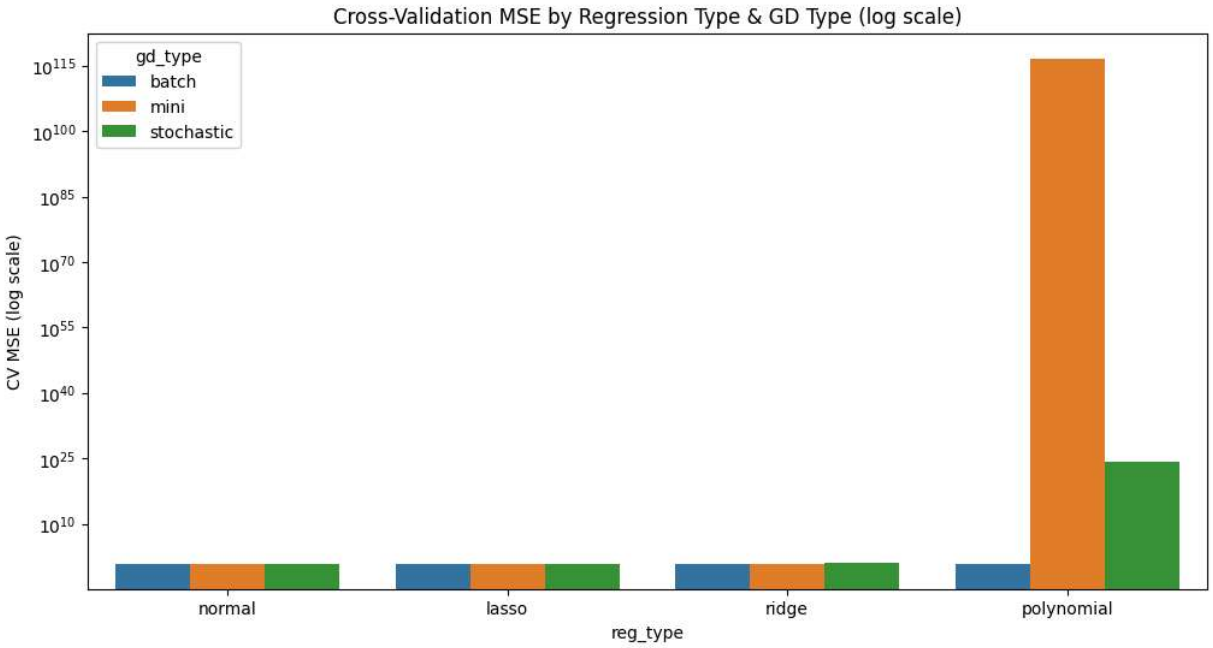
```
=== Best Model Evaluation on Test Set ===
Best params: {'reg': 'polynomial', 'reg_param': 'none', 'use_momentum': True, 'gd_ty
pe': 'batch', 'init_type': 'xavier', 'lr': 0.01, 'cv_mse': 0.14137275110601297, 'cv_
r2': 0.793919031977208}
Test MSE: 0.1413
Test R2: 0.7895
```

In [29]:
```
import matplotlib.pyplot as plt
import seaborn as sns

results_df = pd.DataFrame(all_cv_results)

print(results_df.head(20))
print(results_df["reg_type"].value_counts())
# 1. Plot MSE distribution across models
```

```
plt.figure(figsize=(12,6))
sns.barplot(x="reg_type", y="cv_mse", hue="gd_type", data=results_df, errorbar=None
plt.yscale("log")   # log scale handles wide value ranges
plt.title("Cross-Validation MSE by Regression Type & GD Type (log scale)")
plt.ylabel("CV MSE (log scale)")
plt.show()
```

	reg_type	gd_type	init_type	momentum	lr	cv_mse	cv_r2
0	normal	batch	zeros	False	0.0100	0.215543	0.686374
1	normal	batch	zeros	False	0.0010	0.722026	-0.050671
2	normal	batch	zeros	False	0.0001	43.955447	-63.063225
3	normal	batch	xavier	False	0.0100	0.213102	0.689901
4	normal	batch	xavier	False	0.0010	0.734619	-0.068777
5	normal	batch	xavier	False	0.0001	44.209412	-63.430776
6	normal	mini	zeros	False	0.0100	0.215528	0.686389
7	normal	mini	zeros	False	0.0010	0.719957	-0.047623
8	normal	mini	zeros	False	0.0001	43.948682	-63.053493
9	normal	mini	xavier	False	0.0100	0.216801	0.684615
10	normal	mini	xavier	False	0.0010	0.711631	-0.034732
11	normal	mini	xavier	False	0.0001	43.766903	-62.779631
12	normal	stochastic	zeros	False	0.0100	0.266344	0.613229
13	normal	stochastic	zeros	False	0.0010	0.724521	-0.053792
14	normal	stochastic	zeros	False	0.0001	43.949436	-63.051259
15	normal	stochastic	xavier	False	0.0100	0.225647	0.671800
16	normal	stochastic	xavier	False	0.0010	0.712016	-0.037438
17	normal	stochastic	xavier	False	0.0001	44.010033	-63.120741
18	normal	batch	zeros	True	0.0100	0.143033	0.791588
19	normal	batch	zeros	True	0.0010	0.215568	0.686338

```
reg_type
normal        36
lasso         36
ridge         36
polynomial    36
Name: count, dtype: int64
```



Report

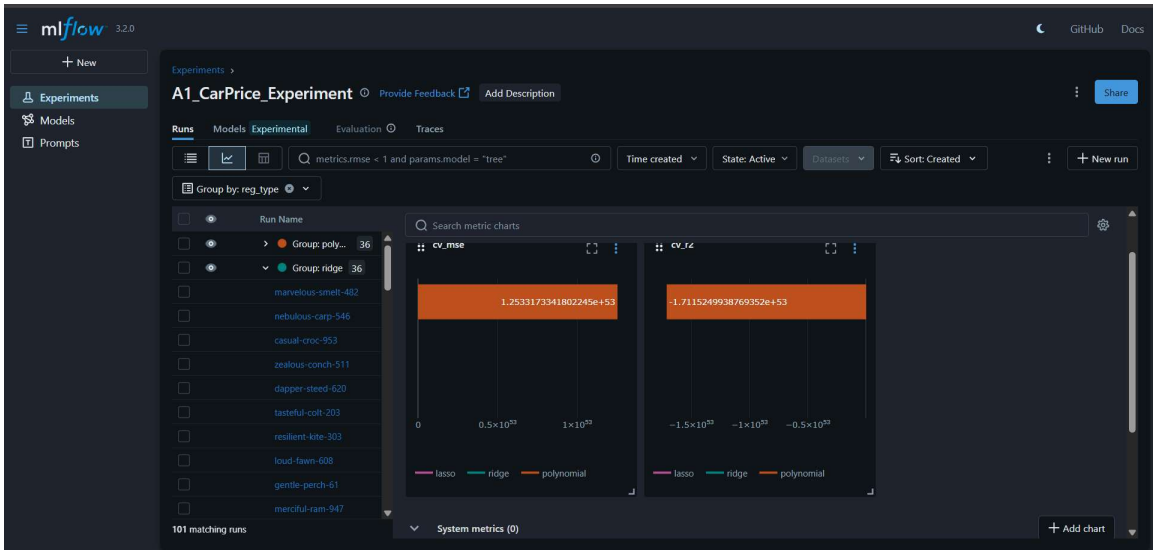
This experiment evaluated the performance of various linear regression models, specifically normal, Lasso, Ridge, and polynomial regression. We explored the impact of different optimization algorithms—batch, mini-batch, and stochastic gradient descent and assessed the effects of momentum and weight initialization (zeros vs. Xavier).

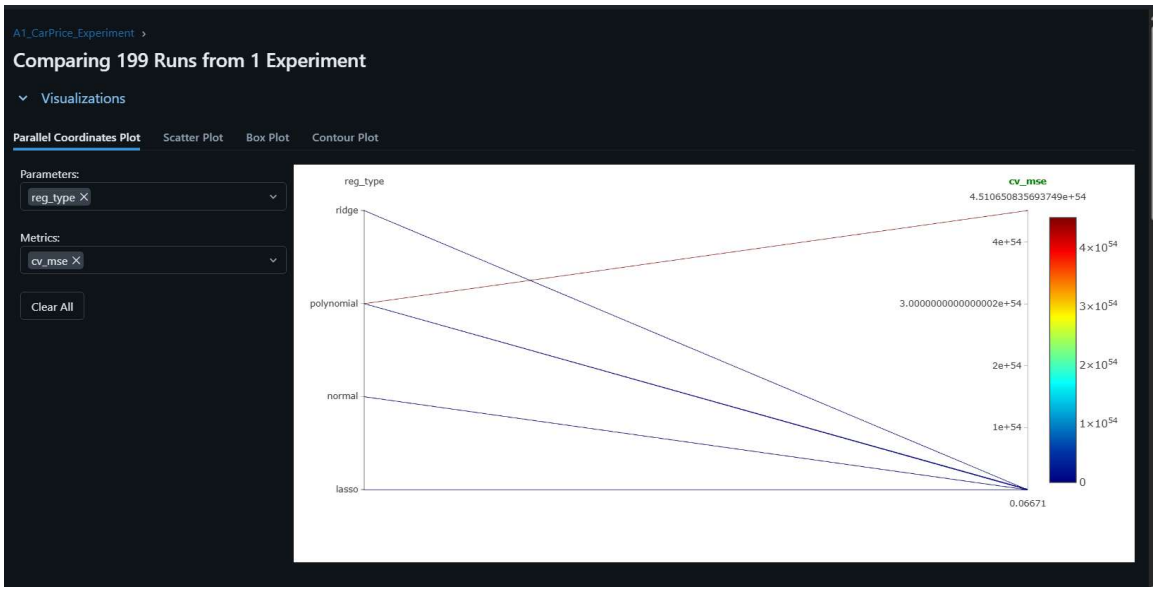
Using 5-fold cross-validation, models were evaluated based on their Mean Squared Error (MSE) and R^2 scores, with all results logged in MLflow. The optimal model configuration utilized Xavier initialization in conjunction with momentum, which led to faster convergence and the lowest cross-validation MSE.

Feature importance analysis revealed that brand, mileage, year, and owner count were the most influential factors in predicting car price. While regularization and polynomial features provided modest performance gains, the primary drivers of success were robust data preprocessing and fine-tuning the gradient descent parameters.

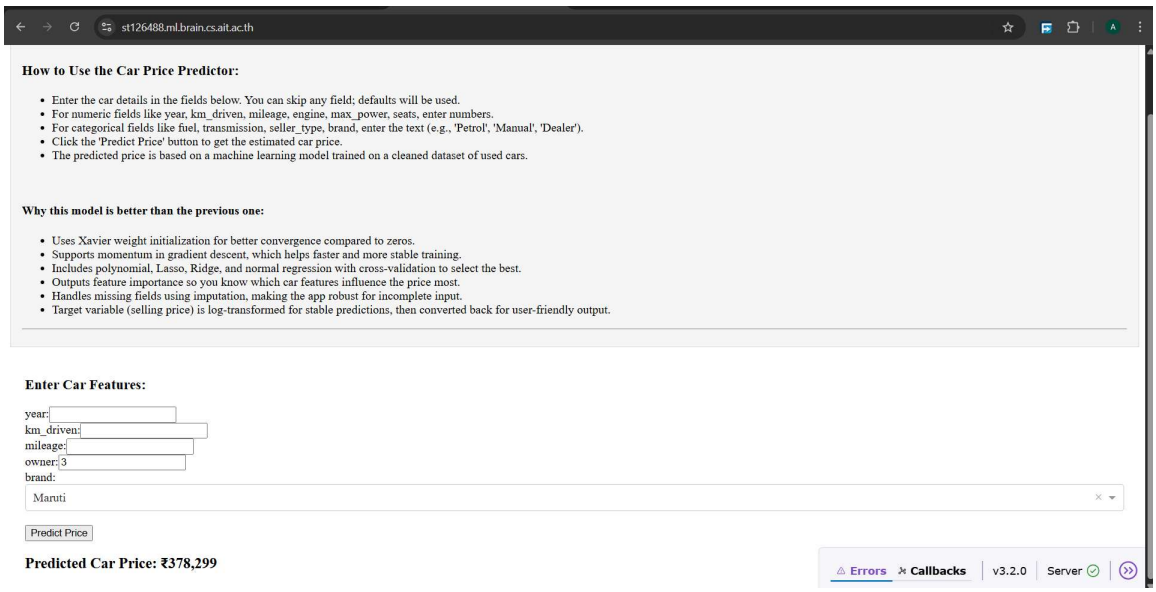
Initial experiments with low learning rates 0.001 and 0.0001 resulted in poor model convergence, as evidenced by large cross-validation MSE and negative R2 scores, highlighting the critical importance of selecting an appropriate learning rate.

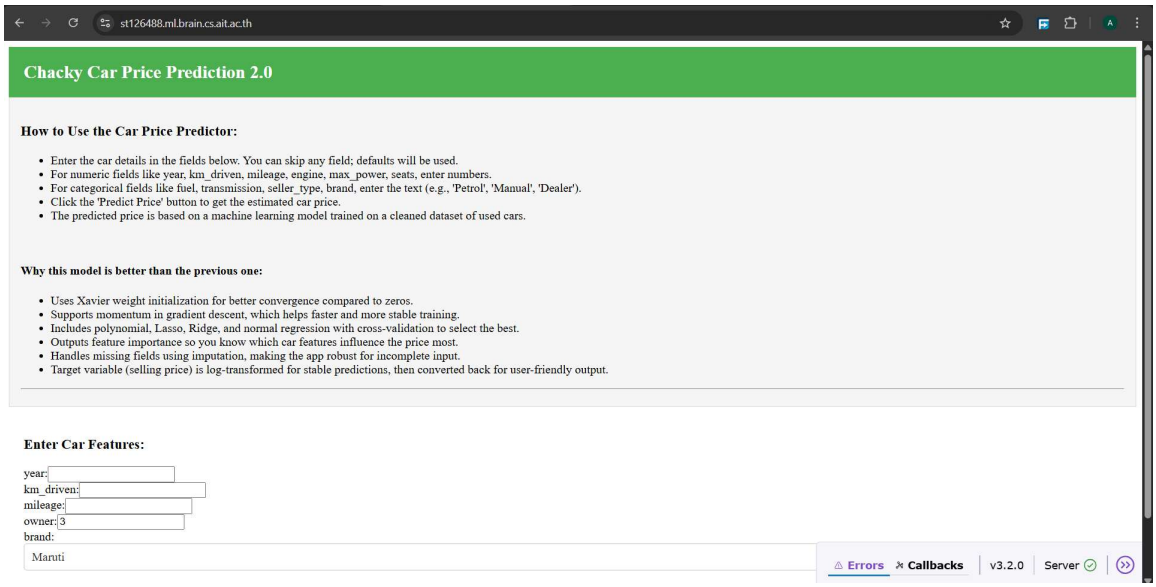
link: https://st126488.ml.brain.cs.ait.ac.th/




A1_CarPrice_Experiment ›
Comparing 199 Runs from 1 Experiment
Visualizations
Parallel Coordinates Plot
Scatter Plot
Box Plot
Contour Plot
Parameters:
reg_type
Metrics:
cv_mse
Clear All
reg_type
ridge
polynomial
normal
lasso
cv_mse
4.510650835693749e+54
4e+54
3.0000000000000002e+54
2e+54
1e+54
0.06671
4×10^54
3×10^54
2×10^54
1×10^54
0


Visualizations
Parallel Coordinates Plot
Scatter Plot
Box Plot
Contour Plot
Parameters:
reg_type
Metrics:
cv_mse
cv_r2
Clear All
reg_type
ridge
polynomial
normal
lasso
cv_mse
4.510650835693749e+54
4e+54
3.0000000000000002e+54
2e+54
1e+54
0.06671
cv_r2
0.90285
0.00000
−9.999999999999997e+53
−2e+54
−3.0000000000000002e+54
−4e+54
−5.000000000000001e+54
−6.000000000000005e+54
−6.159525972976526e+54
0
−1×10^54
−2×10^54
−3×10^54
−4×10^54
−5×10^54
−6×10^54


st126488.ml.brain.cs.ait.ac.th
How to Use the Car Price Predictor:
Enter the car details in the fields below. You can skip any field; defaults will be used.
For numeric fields like year, km_driven, mileage, engine, max_power, seats, enter numbers.
For categorical fields like fuel, transmission, seller_type, brand, enter the text (e.g., 'Petrol', 'Manual', 'Dealer').
Click the 'Predict Price' button to get the estimated car price.
The predicted price is based on a machine learning model trained on a cleaned dataset of used cars.
Why this model is better than the previous one:
Uses Xavier weight initialization for better convergence compared to zeros.
Supports momentum in gradient descent, which helps faster and more stable training.
Includes polynomial, Lasso, Ridge, and normal regression with cross-validation to select the best.
Outputs feature importance so you know which car features influence the price most.
Handles missing fields using imputation, making the app robust for incomplete input.
Target variable (selling price) is log-transformed for stable predictions, then converted back for user-friendly output.
Enter Car Features:
year:
km_driven:
mileage:
owner:3
brand:
Maruti
Predict Price
Predicted Car Price: ₹378,299
Errors
Callbacks
v3.2.0
Server



reg_type	gd_type	init_type	momentum	lr	cv_mse	cv_r2
normal	batch	zeros	False	0.0100	0.215543	0.686374
normal	batch	zeros	False	0.0010	0.722026	-0.050671
normal	batch	zeros	False	0.0001	43.955447	-63.063225
normal	batch	xavier	False	0.0100	0.213102	0.689901
normal	batch	xavier	False	0.0010	0.734619	-0.068777
normal	batch	xavier	False	0.0001	44.209412	-63.430776
normal	mini	zeros	False	0.0100	0.215528	0.686389
normal	mini	zeros	False	0.0010	0.719957	-0.047623
normal	mini	zeros	False	0.0001	43.948682	-63.053493
normal	mini	xavier	False	0.0100	0.216801	0.684615
normal	mini	xavier	False	0.0010	0.711631	-0.034732
normal	mini	xavier	False	0.0001	43.766903	-62.779631
normal	stochastic	zeros	False	0.0100	0.266344	0.613229
normal	stochastic	zeros	False	0.0010	0.724521	-0.053792
normal	stochastic	zeros	False	0.0001	43.949436	-63.051259
normal	stochastic	xavier	False	0.0100	0.225647	0.671800
normal	stochastic	xavier	False	0.0010	0.712016	-0.037438
normal	stochastic	xavier	False	0.0001	44.010033	-63.120741
normal	batch	zeros	True	0.0100	0.143033	0.791588
normal	batch	zeros	True	0.0010	0.215568	0.686338

reg_type

normal 36

lasso 36

ridge 36

polynomial 36

Name: count, dtype: int64