Machine Learning Project: AutoTrader Car Price Prediction

6G7V0015 Machine Learning Concepts

Karlo Nahro — 19003070

1. Data/Domain Understanding and Exploration

1.1 Meaning and Type of Features; Analysis of Distributions

The dataset has 402,005 car adverts with 12 columns. I spent a while going through each one to understand what we're actually working with here. The features break down into two main categories:

Numerical features (3): mileage, year_of_registration, and price (which is our target variable that we're trying to predict).

Categorical features (8): reg_code, standard_make, standard_model, standard_colour, vehicle_condition, body_type, fuel_type, and crossover_car_and_van (which is a boolean but I treated it as categorical for encoding purposes).

I dropped public_reference early on since it's just a unique identifier for each advert - there's no predictive value in a random ID number. That left me with 11 actual features to work with.

Missingness check:

```
missing = (df.isna().mean().sort_values(ascending=False) * 100)
display(missing.head(6).to_frame("missing_%"))
```

Feature	Missing (%)
year_of_registration	8.29
reg_code	7.92
standard_colour	1.34
body_type	0.21
fuel_type	0.15
mileage	0.03

So year_of_registration and reg_code have the highest missingness at around 8% each. I noticed something interesting though - these two features are actually related. UK registration codes (like "70", "21", "69" etc.) encode when the car was first registered. This turned out to be really useful later for imputation because if I know the reg_code, I can make a pretty good guess at the registration year.

The other features have fairly minimal missingness (under 1.5%) so they're less of a concern. The pipeline's SimpleImputer can handle those easily with median/mode imputation.

Target distribution (price):

Price is the variable I'm trying to predict, so I spent extra time understanding its distribution. It's heavily right-skewed - the median is around £12,600 but there are outliers going all the way up to £9,999,999 (which is almost certainly either a listing error or some kind of hypercar). The 95th percentile is £43,000 and the 99th percentile is £88,900.

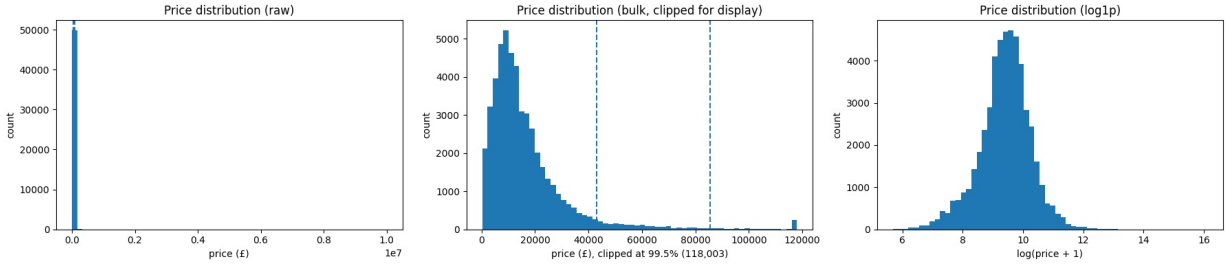

Figure 1: Price distribution shown three ways - raw (left), clipped at 99.5% for visualisation (centre), and log-transformed (right). The log transform produces something much closer to a normal distribution.

The left plot is basically useless - you can't see anything because a few extreme values compress everything else. The middle plot clips at the 99.5th percentile just for visualisation purposes, and you can see it's still right-skewed. But the right plot with the log transform (using log1p to handle any zeros) looks much more normal. This is really important because linear regression assumes normally distributed errors, and having a normal-ish target helps with that.

I also looked at the distributions of the numerical features:

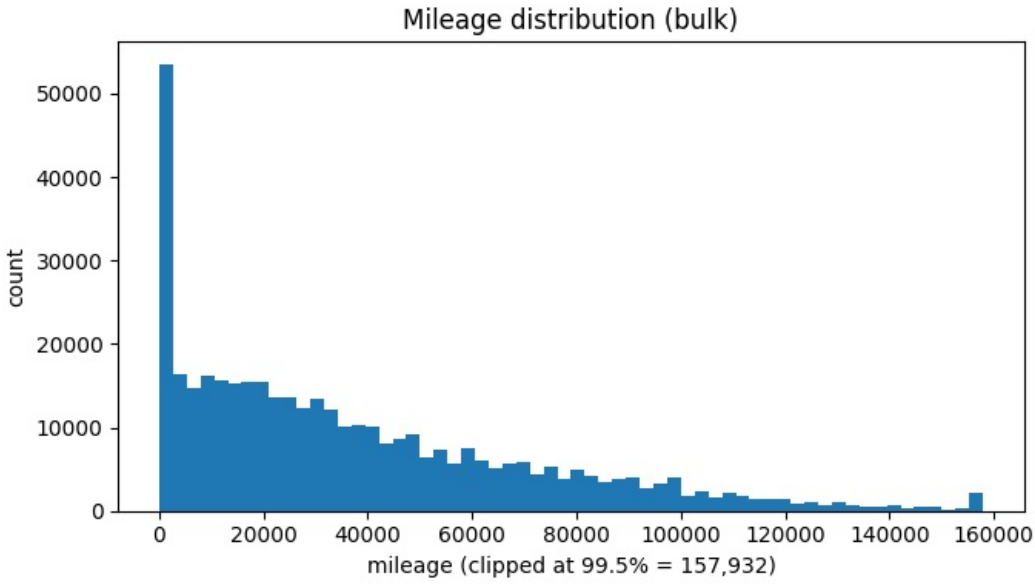

Figure 2: Mileage distribution (clipped at 99.5% = 157,932 miles). Also right-skewed with most cars having relatively low mileage.

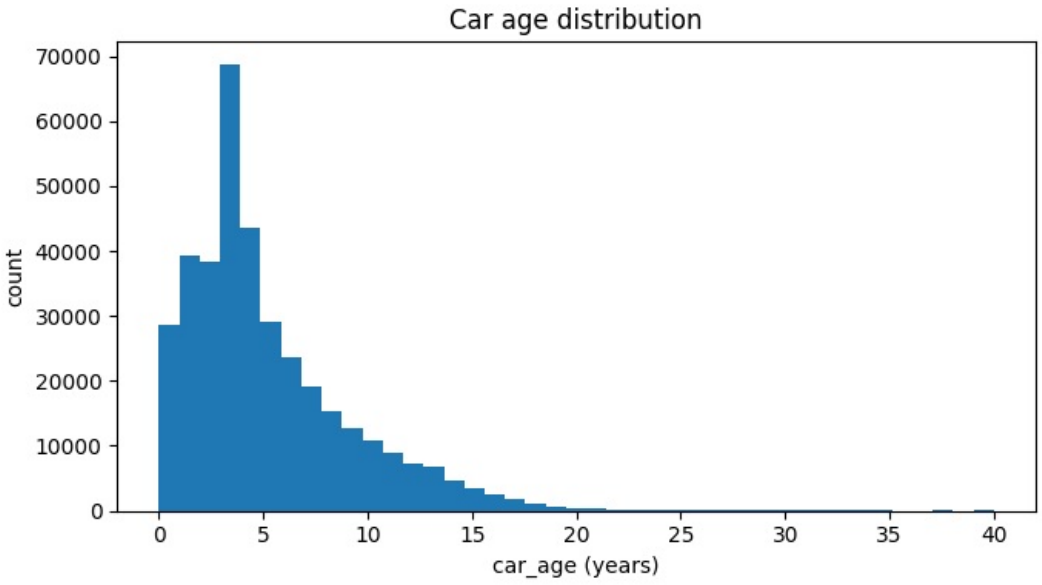

Figure 3: Car age distribution. Most cars are 1-5 years old, with a long tail of older vehicles. The spike at around 3-4 years probably reflects common lease/PCP deal lengths.

1.2 Analysis of Predictive Power of Features

Before building any models, I wanted to get a sense of which features actually correlate with price. This helps with feature selection and also gives intuition about what's driving car prices.

I used Spearman correlation rather than Pearson because the relationships might not be perfectly linear (depreciation curves aren't straight lines, for example):

```
from scipy.stats import spearmanr
for col in num_features:
    rho, pval = spearmanr(X_train[col], np.log1p(y_train))
```

Feature	Spearman ρ	p-value
car_age	-0.681	< 0.001
year_of_registration	+0.681	< 0.001
mileage	-0.646	< 0.001
mileage_per_year	+0.036	< 0.001

A few observations here. First, car_age and year_of_registration have the same magnitude (0.681) but opposite signs - which makes sense because one is literally derived from the other (car_age = 2020 - year_of_registration). A correlation of |0.68| is pretty strong, which confirms that depreciation is a major driver of car prices. Newer cars cost more, shocker.

Mileage also has a fairly strong negative correlation (-0.646). Higher mileage means cheaper cars, which again makes intuitive sense. What's interesting is that mileage_per_year has almost no correlation (0.036). I thought this feature might capture "how hard the car was driven" but apparently the market doesn't really care about that, at least not in a simple linear way.

I also looked at how price varies with categorical features. Here's median price broken down by car age:

Age Bin (years)	Count	Median Price (£)
(0, 1]	27,539	19,990
(1, 2]	26,912	14,999
(3, 4]	30,597	12,295
(5, 7]	29,984	7,995
(7, 10]	27,230	5,350
(10, 15]	21,829	2,995
(15, 20]	4,051	2,200
(20, 40]	1,232	9,000

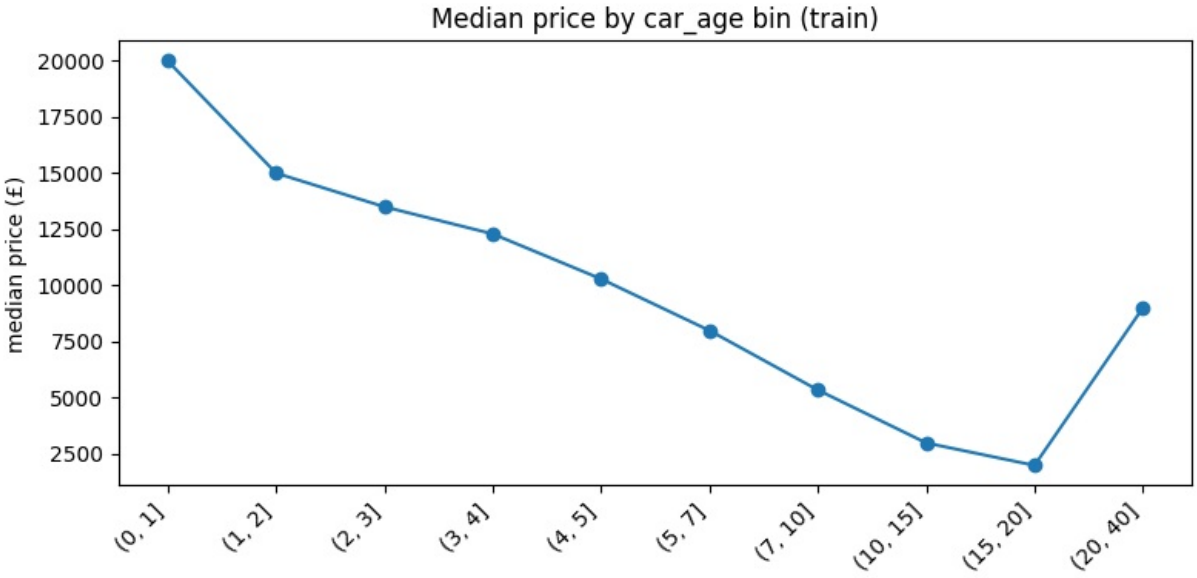

Figure 4: Median price by car age bin. Clear depreciation curve from 0-20 years, then an interesting uptick for very old cars (20-40 years) - probably classics and collectibles.

That last row is really interesting - cars aged 20-40 years actually have a higher median price than 10-20 year old cars! This is probably classic cars and collector vehicles. A 1985 Porsche 911 is worth more than a 2005 Mondeo. The model will probably struggle with this non-monotonic relationship.

I also checked median prices by make to see which brands command premiums:

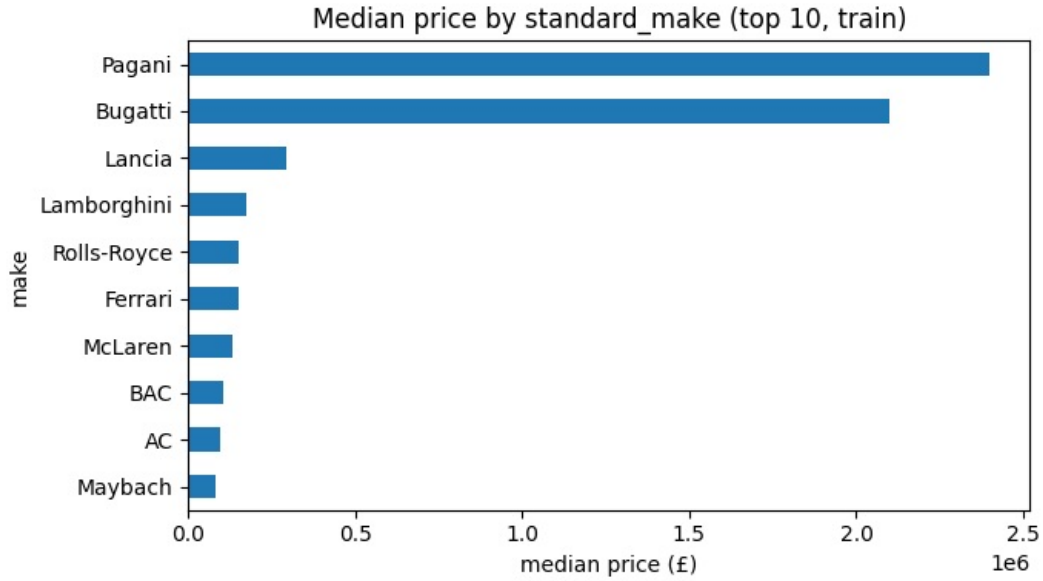

Figure 5: Median price by manufacturer (top 10). Pagani and Bugatti dominate with median prices in the millions, followed by other luxury/supercar brands.

1.3 Data Processing for Exploration and Visualisation

The main challenge for visualisation (and later for modelling) was high-cardinality categorical features. standard_model alone has 1,092 unique values - you can't really create a meaningful bar chart with that many categories, and one-hot encoding it would create over a thousand features.

My solution was a top-K approach: keep the most frequent K categories for each high-cardinality column, and lump everything else into an "Other" category. Crucially, I determined the top-K categories using only the training data to avoid any leakage:

```
TOP_K_BY_COL = {"reg_code": 30, "standard_make": 50, "standard_model": 200}
top_levels = train_df[col].value_counts(dropna=False).head(k).index
# Then map anything not in top_levels to "Other"
```

Column	K	Before	After
standard_model	200	1,092	201
standard_make	50	104	51
reg_code	30	71	31

This made the one-hot encoding way more manageable. After encoding, I ended up with 336 total features instead of potentially thousands. The top 200 models cover about 95% of all observations anyway, so we're not losing much signal by collapsing the rare models into "Other".

2. Data Processing for Machine Learning

2.1 Dealing with Missing Values, Outliers, and Noise

Noise/Invalid values: While exploring the data, I found some records with year_of_registration = 999, which is obviously wrong (unless someone's selling a car from the future or the Middle Ages). I treated anything outside the range 1980-2020 as missing:

```
YEAR_MIN, YEAR_MAX = 1980, 2020
yr_invalid = yr.notna() & ((yr < YEAR_MIN) | (yr > YEAR_MAX))
df.loc[yr_invalid, "year_of_registration"] = np.nan  # 331 records corrected
```

Missing value imputation: This is where the reg_code relationship came in useful. Since UK registration codes encode the registration period, I could use them to impute missing years. I built a lookup table mapping each reg_code to its median year, using only training data:

```
# Learn mapping on TRAIN only to avoid leakage
train_map = X_train.groupby("reg_code_num")["year_of_registration"].median()
X_train.loc[miss, "year_of_registration"] = X_train.loc[miss, "reg_code_num"].map(train_map)
# Apply same mapping to val and test
```

Split	Missing Before	Missing After
Train	23,550	22,355
Validation	5,039	4,773
Test	5,053	4,808

This reduced missingness by about 5% in each split. Not perfect - still around 5-6% missing - but better than before. The remaining missing values get handled by SimpleImputer with median imputation in the preprocessing pipeline.

Outlier handling: Rather than arbitrarily removing extreme prices (where would I draw the line? 99th percentile? 99.9th?), I decided to use a log-transformed target during modelling. The TransformedTargetRegressor applies log1p before fitting and expm1 after predicting, so the model sees a more normally distributed target. This naturally downweights the crazy expensive cars without throwing away any data. It felt cleaner than making subjective decisions about what counts as an "outlier".

2.2 Feature Engineering, Data Transformations, Feature Selection

Engineered features: I created two new features based on domain knowledge about car pricing:

```
CURRENT_YEAR = int(np.nanmax(df["year_of_registration"]))  # 2020
df["car_age"] = CURRENT_YEAR - df["year_of_registration"]
df["mileage_per_year"] = np.where(df["car_age"] >= 1, df["mileage"] / df["car_age"], np.nan)
```

car_age is pretty standard for car pricing models - it directly captures depreciation. mileage_per_year was my attempt to capture "intensity of use" - a 5-year-old car with 100k miles has been driven much harder than one with 25k miles. Unfortunately, as shown in section 1.2, this feature didn't correlate well with price, but I kept it in case it helps in combination with other features.

Preprocessing pipelines: I built two versions using sklearn's ColumnTransformer:

1. **Basic pipeline:** SimpleImputer (most_frequent) + OneHotEncoder for categoricals; SimpleImputer (median) for numerics. Used for Decision Tree and plain Linear Regression.
2. **Scaled pipeline:** Same as basic but adds StandardScaler for numerics. Required for kNN (which uses Euclidean distance) and Ridge (helps with regularisation).

Final feature count after one-hot encoding: 336 features. I kept the output as a sparse matrix to save memory since most of those features are zeros (one-hot encoded categoricals).

3. Model Building

3.1 Algorithm Selection, Model Instantiation and Configuration

I used the three model families covered in the lectures:

1. Ridge Regression: A linear model with L2 regularisation. I chose this over plain linear regression because with 336 features (mostly one-hot encoded), there's risk of overfitting or numerical instability. The regularisation helps prevent coefficient explosion. I wrapped it in TransformedTargetRegressor to handle the log-transformed target:

```
ridge_pipe = Pipeline([
    ("preprocess", preprocess_for_knn),  # scaled version
    ("model", TransformedTargetRegressor(
        regressor=Ridge(alpha=10.0, solver="sag", max_iter=5000),
        func=np.log1p, inverse_func=np.expm1))
])
```

2. k-Nearest Neighbours: A non-parametric method that makes predictions based on similar cars in the training set. Requires scaled features since it uses Euclidean distance. It's computationally expensive on 400k rows, so I used subsamples for hyperparameter tuning.

3. Decision Tree: Can capture non-linear interactions and automatically finds useful feature thresholds. Prone to overfitting though, so I had to constrain it with max_depth and min_samples_leaf parameters.

3.2 Grid Search, Model Ranking and Selection

Ridge alpha sweep:

Alpha	Val MAE (£)	Val RMSE (£)	Val R^2
10.00	2,989	14,774	0.588
1.00	3,006	14,767	0.589
0.10	3,009	14,769	0.589
100.00	3,079	15,364	0.555
1000.00	3,906	17,814	0.401

Alpha=10 gives the best MAE by a small margin. Interestingly, alpha values from 0.1 to 10 give almost identical results, which suggests the features aren't too badly collinear. Only at very high regularisation (1000) does performance really degrade.

kNN k-sweep (on 40k subsample for computational feasibility):

k	Val MAE (£)	Val RMSE (£)	Val R^2
5	3,353	17,214	0.487
7	3,412	17,469	0.472
3	3,425	17,459	0.472
1	4,303	71,932	-7.96

k=1 is a disaster - negative R^2 means it's worse than just predicting the mean! This is classic overfitting: the model memorises the training data instead of learning patterns. k=5 looks like the sweet spot.

Decision Tree sweep (on 80k subsample):

Depth	Min Leaf	Val MAE (£)	Val R^2
20	5	3,354	0.546
20	20	3,702	0.494
15	5	3,853	0.518
12	5	4,324	0.478

Best tree configuration: depth=20, min_samples_leaf=5.

5-Fold Cross-Validation comparison:

Model	CV MAE Mean	CV MAE Std	CV R^2 Mean	CV R^2 Std
Ridge (α=10)	£3,187	£273	0.349	0.227
Decision Tree (d=20)	£3,337	£245	0.338	0.227
kNN (k=5)	£3,785	£483	0.271	0.280

Ridge has the lowest mean MAE and also the smallest standard deviation, indicating stable performance across folds. The R^2 variance is high for all models - I think this is because different folds have different proportions of luxury cars, which are hard to predict.

Model selection: Going with Ridge. It has the best CV performance, the most stable results, and it's interpretable (I can look at coefficients). It's also way faster to train than kNN.

4. Model Evaluation and Analysis

4.1 Coarse-Grained Evaluation

For final evaluation, I refit Ridge on the combined train+validation set, then did a single evaluation on the held-out test set:

```
X_trainval = pd.concat([Xtr, Xva], axis=0)
y_trainval = pd.concat([y_train, y_val], axis=0)
final_model.fit(X_trainval, y_trainval)
test_pred = final_model.predict(Xte)
```

Metric	Test Result
MAE	£3,162.13
RMSE	£23,708.84
R^2	0.3884

The R^2 dropped from 0.59 on validation to 0.39 on test, which worried me at first. But looking at MAE, it only went up by about £170 (from £2,989 to £3,162), which is roughly 6%. The R^2 is being dominated by a few massive errors on super-expensive cars - RMSE is very sensitive to outliers.

Stratified evaluation by price band:

Price Band	n	MAE (£)	R^2
≤ £43,720 (p95)	57,275	1,956	0.858
p95–p99	2,406	12,344	-1.155
> £87,995 (p99)	620	78,938	-0.068

This tells a much clearer story. For the 95% of "normal" cars (under ~£44k), the model does really well: $R^2 = 0.86$ and MAE under £2,000. That's pretty good for a car price predictor! But for expensive cars, performance falls off a cliff. The model is essentially useless for the top 1%.

4.2 Feature Importance

Ridge coefficients (log-scale):

Feature	Coefficient
standard_make_Ferrari	+2.144
standard_make_Lamborghini	+2.011
standard_make_Rolls-Royce	+1.908
standard_make_Porsche	+0.887
standard_make_Chevrolet	-0.951
standard_make_Dacia	-0.928

Since we're predicting log(price), these coefficients have a multiplicative interpretation. A coefficient of +2.14 for Ferrari means exp(2.14) ≈ 8.5, so a Ferrari is predicted to cost about 8.5x more than the baseline (all else equal). Makes sense - the Ferrari badge commands a serious premium.

Permutation importance:

Feature	Importance (MAE increase, £)
standard_make	+8,206
standard_model	+3,274
reg_code	+3,156
mileage	+2,348
body_type	+1,087

Make is by far the most important feature - shuffling it randomly increases MAE by over £8,000! Model and reg_code are next. Interestingly, mileage only ranks fourth. Brand identity matters more than how much the car's been driven.

4.3 Fine-Grained Evaluation

Residual analysis:

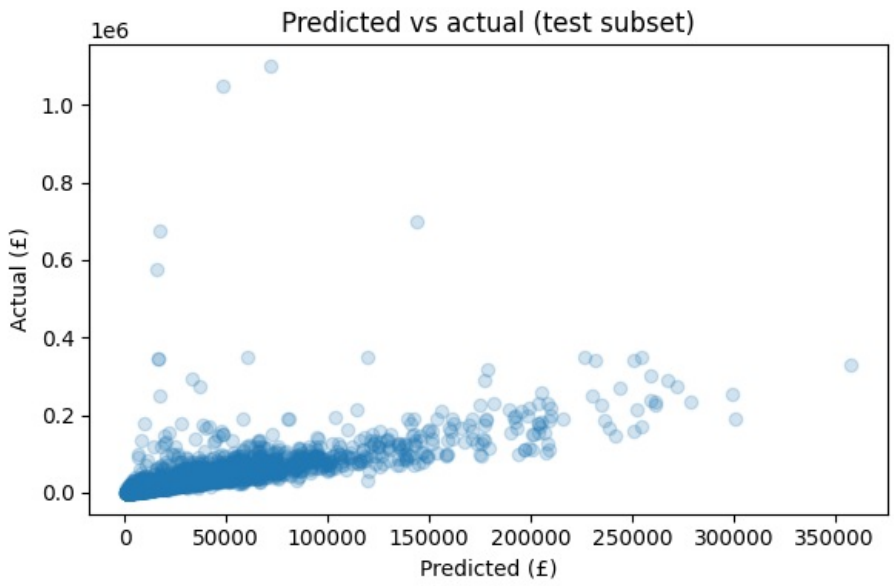

Figure 6: Predicted vs Actual prices on test set. Points should lie along the diagonal. There's a clear pattern of

underprediction for expensive cars (points above the line).

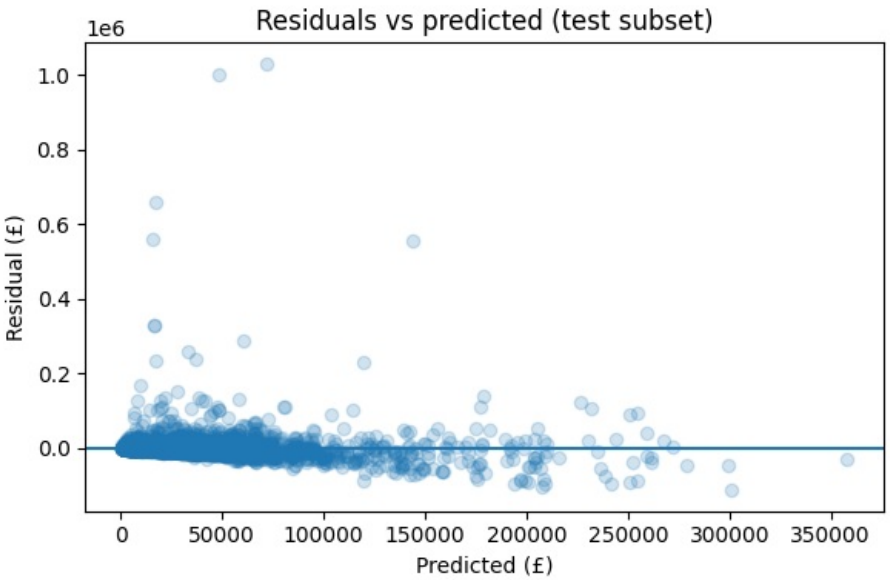

Figure 7: Residuals vs Predicted values. Residuals should be randomly scattered around zero. Instead, they fan out as predicted price increases - heteroscedasticity. High-value cars have larger errors.

The residual plots confirm what the stratified evaluation showed: the model systematically underpredicts expensive cars. The residuals fan out as predicted values increase, which is a classic sign of heteroscedasticity.

Worst predictions:

Make	Model	Age	Actual (£)	Predicted (£)	Error (£)
Ferrari	Other	3	3,799,995	167,952	3,632,043
Ferrari	Other	5	2,150,000	195,251	1,954,749
Other	Other	0	1,400,000	45,428	1,354,572
McLaren	Other	7	1,200,000	124,866	1,075,134

These are collector cars and hypercars. Notice they all have model="Other" - that's because my top-K mapping collapsed rare models. The model can't distinguish a £3.8 million Ferrari 250 GTO from a £150k California T because they're both just "Ferrari, Other". Would probably need a specialist model for the ultra-luxury segment.

Ablation study - removing reg_code:

Setting	Val MAE (£)	Val R^2
With reg_code	2,989	0.588
Without reg_code	3,073	0.599

MAE increases by £84 without reg_code, confirming it provides useful signal. But weirdly, R^2 actually improves slightly without it. Maybe reg_code adds some noise alongside the signal? Or it could be that year_of_registration already captures most of the same information. Either way, the effect is small.

Summary and Reflections

Overall, the Ridge regression model works well for the mainstream car market (under £44k), achieving an R^2 of 0.86 and MAE of under £2,000. This is pretty reasonable for a price prediction task - you could imagine this being useful for a buyer trying to assess whether a listing is fairly priced.

The main limitation is performance on expensive cars. The model systematically underpredicts high-value vehicles, with errors in the hundreds of thousands of pounds for collector cars and hypercars. There are a few reasons for this:

1. **Top-K category mapping:** Rare models get collapsed into "Other", losing the ability to distinguish a £150k Ferrari from a £3.8 million one.
2. **Log-transform limitations:** Even with log-transformed targets, the model struggles with the extreme right tail of the price distribution.
3. **Different market dynamics:** Ultra-luxury cars are priced based on factors like rarity, provenance, and collector demand that aren't captured by standard features like mileage and age.

If I were to improve this further, I might try: (1) building a separate model for the luxury segment, (2) using target encoding instead of one-hot for high-cardinality features, or (3) trying gradient boosting methods like XGBoost which might handle the non-linear relationships better. But for a baseline model on mainstream cars, Ridge with log-transformed target does a solid job.

One thing I learned from this project is how much brand matters in car pricing. I expected mileage and age to dominate, but permutation importance showed that make and model together account for far more predictive power than mileage. The car market is fundamentally about brand perception as much as it is about the physical condition of the vehicle.