SHM

Data Modelling and Processing

Technical Report

Łukasz Sawala - lukaszsawala2003@gmail.com

October 29, 2025

Preface

I'd like to start this report by thanking you for giving me the opportunity to share something I am deeply excited about (AI) in such a unique way. I've had boring coding interviews, complex and unrealistic problems, and straight up annoying interview questions asking the same thing over and over again. This was a fun challenge for me to take, and even though I had assignments due, I spent way more on this than I should've. Thanks again and hope you like it!

Introduction

The dataset obtained from you turned out to be fairly large, complex, and quite challenging to work with. At the very beginning, the main issue became clear: a huge number of variables, a lot of missing values, and a large portion of categorical columns. In a real-world setup, this kind of dataset would require heavy domain knowledge to clean, interpret and process properly. However, I was sad to learn from Alon that it was not a trick designed to test my thinking, but that there is indeed no knowledge on this dataset.

For this challenge, I focused on following good practices, using my intuition, and relying on data analysis to form my own assumptions. With more time, I would have spent significantly longer understanding what each variable represents.

First Look at the Data

My first impression was not great: every row had at least one missing value, and around 60% of all values were NaN. The dataset was dominated by categorical variables, with almost no continuous ones. That immediately screamed: "bad dataset." With no information on what the features actually meant, I had to go old-school data science style.

The first step was to remove useless identifiers such as `Machine ID` and `Sales ID`, as they carry no predictive information. Then, I noticed that one of the date columns was not parsed properly, so I recast it into the right format. I also found a clear outlier: around 20% of entries had `year = 1000`, which obviously makes no sense. I replaced those values with NaNs to ensure the model does not get confused with valid datapoints.

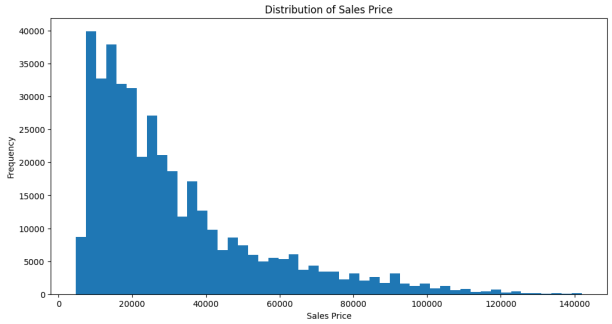
Because so many variables were partially missing, I decided to drop those where more than 90% of entries were NaN. The threshold of 90% was arbitrary, but reasonable under no domain knowledge - trimming only extreme cases that would add more noise than signal.

Finally, I engineered a new variable: **the age of the machine**. There were both production and purchase dates, but no explicit interaction term between them. Creating this feature helps the model learn conditional dependencies (e.g., older machines tend to sell for less), which is especially useful when much of the other information is missing. I had plans to add more features like this, but time didn't allow it.

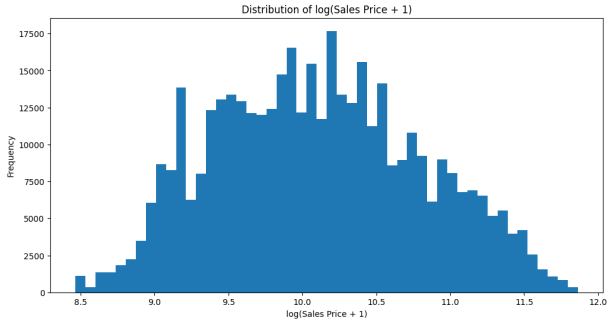
Target Transformation and Data Assumptions

When I looked at the target variable (`Sales Price`), it was heavily right-skewed. To make it easier to predict and stabilize variance, I applied a logarithmic transformation. The goal here was to make the model perform well in log-space and then simply exponentiate the outputs to recover the real prices. This is necessary because non-uniform (skewed) target distributions are much harder to predict than normal-like ones. Normally distributed targets have balanced variance and fewer extreme outliers, which makes it easier for models to learn consistent patterns and minimize error across the full output range. Figure 1 shows the difference between the original and log-normalized target distributions.

Since I had no domain knowledge and so many missing values, it was clear to me that I couldn't rely on any interpolation, injection or filling. Filling missing categorical or numeric values arbitrarily (e.g., setting engine hours to 0 or interpolating customer IDs) would introduce heavy bias with so



(a) Original (skewed) target distribution.



(b) Log-normalized (approximately Gaussian) target distribution.

Figure 1: Comparison of target variable distributions before and after log normalization.

little information given. That ruled out most neural-based models or standard statistical methods like correlation analysis/PCA or t-SNE.

Model Choice and Rationale

Given the structure of the dataset, I chose to use models famous for handling missing values: **boosting trees** (see Figure 2). Boosting methods, such as **XGBoost**, **LightGBM**, and **CatBoost**, build an ensemble of weak learners (typically shallow decision trees) in a sequential manner, where each new tree attempts to correct the errors of the previous ones. This iterative process allows the model to gradually minimize the loss function and capture complex, nonlinear dependencies in the data.

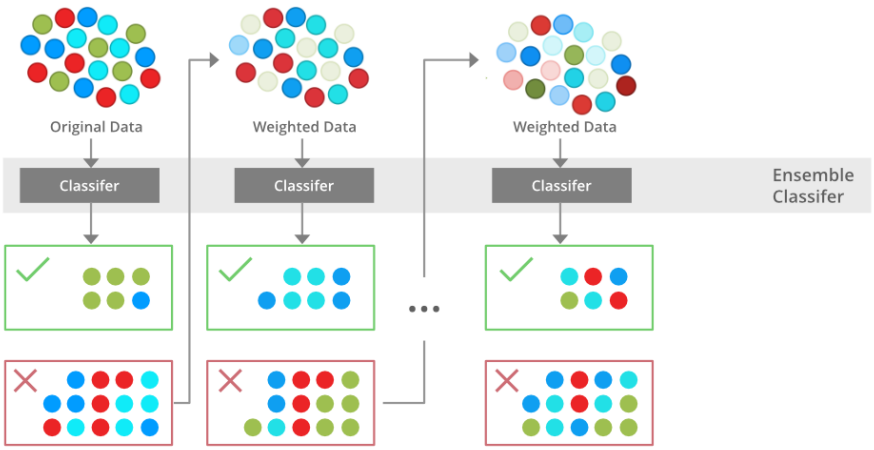


Figure 2: Example architecture of an XGBoost model showing the sequential boosting process.

During training, when a feature value is missing, the algorithm learns an optimal default direction (branch) for missing data at each split, based on which path leads to the lowest loss. This means that the model effectively *learns how to route missing values* in a way that best fits the training data, rather than requiring predefined replacements that would induce bias.

Besides their robustness to missing data, boosting models are known for their strong performance out-of-the-box, limited need for extensive hyperparameter tuning, and high interpretability through feature importance analysis, which made them a perfect choice given the time constraints and explainability desirata of this project.

Validation of ideas

To further validate the ideas introduced during preprocessing and to gain an initial understanding of feature importance and overall performance, I trained a simple **XGBoost** model. The underlying assumption was that if a preprocessing change is beneficial, even a relatively simple model should be able to explain a greater portion of the target variance - an approach similar to selecting the top-M principal components in PCA. The initial model achieved an R^2 score of approximately 0.82 (i.e., 82% of the target variance explained, surprisingly high), which served as the baseline for subsequent experiments.

From there, I started filtering and cleaning step by step:

- Removing only IDs: $R^2 = 0.8325$
- Removing invalid years (`year = 1000`): $R^2 = 0.8328$
- Dropping variables with more than 90% NaNs: $R^2 = 0.8270$ (neglegible decrease)
- Adding the age of machine feature: $R^2 = 0.8950$
- Applying logarithmic target transform: $R^2 = 0.8961$

Adding the new age feature and transforming the target both made the model perform significantly better. I also noticed that **filling NaNs reduced performance**, which confirmed my assumption that I should leave them as-is and let the model handle them internally.

Experiments

Even though XGBoost performed quite well, it doesn't handle categorical variables perfectly. It requires converting them into integer codes (0, 1, 2, ...), which can introduce order bias where none exists. Because of this, I also decided to switch my model to two famous alternatives: *CatBoost* and *LightGBM*.

CatBoost is designed for categorical features and tends to perform very well, but training time was a major bottleneck: around two hours per hyperparameter setting, which gave me time for basically one sweep based on my intuition. That wasn't enough for proper tuning. **LightGBM**, on the other hand, is known to be faster and almost as powerful, so I focused most of my time on it.

After stabilizing the preprocessing and model selection, I wanted to make the code clean, easy to extend, and reproducible. Therefore, I designed a small training and inference pipeline:

- The `main.py` script utilizes a **parser** to be able to handle both training and inference.
- Each model can be initialized in either a lightweight inference mode or a full training mode.
- The training pipeline automatically loads data, runs hyperparameter search, evaluates results, and produces visualizations.
- After training, the model is saved for later use in inference-only mode.

I initially planned to use Weights & Biases for experiment tracking, but it was difficult to integrate since boosting models expose very limited metrics during training. This made the process less transparent, but still manageable with tqdm.

Training and Evaluation

I used **Mean Absolute Error (MAE)** as the loss function. While MSE is standard for regression, the dataset is highly corrupted and contains many potential outliers; MAE ensures the average error remains low without over-penalizing extreme mistakes. All models were trained with standard parameters (not detailed here) with slightly increased capacity (deeper trees, more estimators) and regularization mechanisms (e.g., large leaf pruning) to capture the dataset's intricacies without overfitting to the noise.

The models were trained using a train-validation-test split. The best model was selected based on the lowest validation loss, and then retrained on the combined training and validation sets before evaluation on the test set. This approach ensures a valid performance assessment while leveraging most of the available data. GitHub was used for version control to prevent accidental deletion of trained models.

The performance of the models is summarized in Table 1. Considering the low quality of the data, the baseline XGBoost model, trained on a few hyperparameter settings with unprocessed data, already performs reasonably well, with an average deviation of 18.19% from the expert predictions. Tuned CatBoost significantly improves performance, while fully optimized LightGBM achieves the best results, reducing the error by almost 30%. These results suggest that further data processing and model tuning could potentially yield predictions closely aligned with human experts.

It is important to note that the expert predictions serve only as a *weak ground truth*. The expert can (and hence will) make mistakes, over- or underestimating prices depending on the context. It is therefore possible that the models could actually outperform the human predictions, potentially increasing sales revenue. Consequently, the reported errors should be interpreted with caution; nevertheless, a 13% error represents a very good outcome.

Model	MAE ($)	MAE (% of mean)
XGBoost (U)	5657	18.19
CatBoost (P)	4363	14.03
LGBM (OP)	**4096**	**13.15**

Table 1: Performance of the models on the test set. Currency assumed to be US Dollars ($). U = tuned with a small trial on unprocessed data, P = tuned with a small grid search on preprocessed data, OP = fully tuned with a large Optuna study on preprocessed data.

Feature Importance

One major advantage of tree-based models is their native support for feature importance, which provides direct insight into the patterns learned by the model.

For the initial, lightly trained XGBoost, feature importances are shown in Figure 3. Most of the importance is given to the `product group description`, one of the few variables without missing values, representing one of the six countries where items were sold or produced. XGBoost also considers `machine size` and `product class` important, which seems plausible. One anomaly is `Unnamed:0`, an ID column that should not contribute to predictions.

In contrast, the fully trained LightGBM focuses on features that all seem plausible. The importance distribution is much less skewed than in XGBoost, indicating that the model utilizes a broader range of features rather than relying on a few dominant ones. Notably, the five most important features include two created during preprocessing (`age of machine` and `year made`), supporting the value of those feature engineering steps. CatBoost yields very similar feature importance patterns; the main difference is that its importances are more skewed, suggesting a slightly lower quality of learned patterns. The corresponding plot is available in the provided code.

Lastly, due to a library flag error, my CatBoost model could not be re-instantiated for inference and would require retraining. Performing inference for LightGBM on single data points was

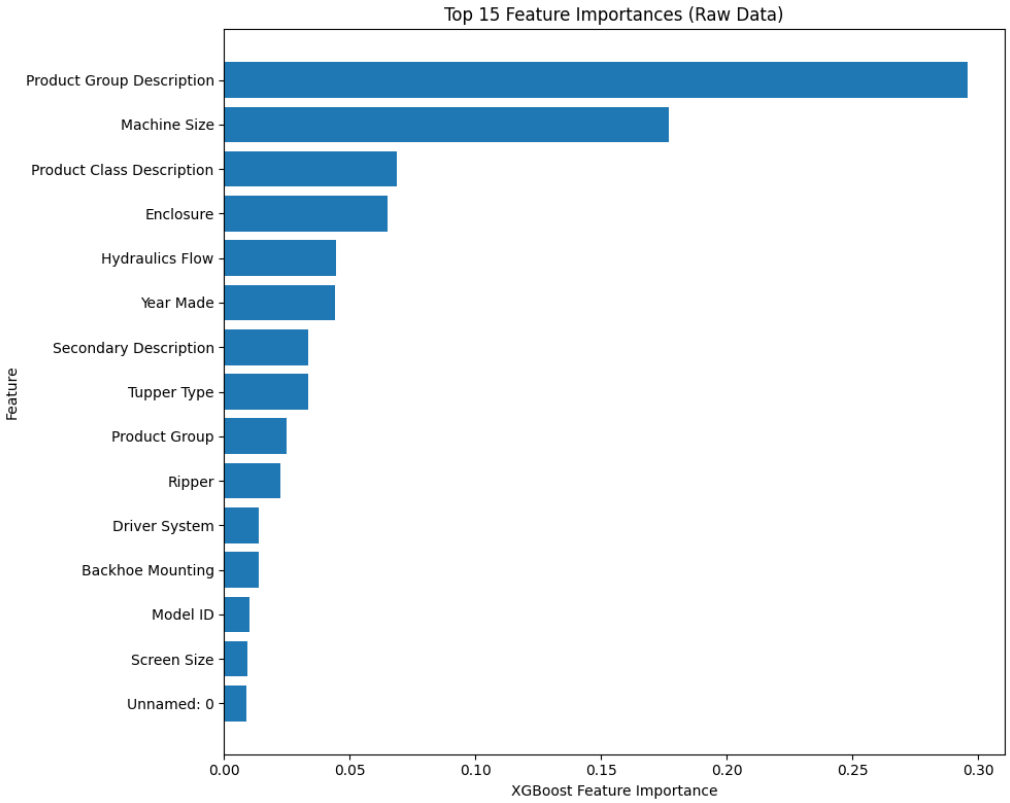


Figure 3: Top 15 feature importances from the initial XGBoost model.

also cumbersome, highlighting the trade-offs of using pre-made libraries rather than fully custom implementations using, e.g., JAX.

Discussion and Future Work

The biggest challenge by far was the data quality and lack of context. If I had more time for this project, I would:

- Analyze price changes over time per category, which almost certainly happens (although incorporating this into the model would not be straightforward).
- Experiment with a more aggressively pruned and augmented dataset.
- Try combining categorical variables into histograms to create continuous-like inputs.
- Deploy the model on `Streamlit` to support easier visualization and interaction.

I also briefly tried different imputation and pruning combinations, but they either performed worse or made no real difference. Boosting models clearly benefited more from minimal interference and good feature design than from aggressive preprocessing.

Conclusion

This challenge was all about staying down-to-earth under uncertainty. With every row having at least one missing value and over half the dataset being NaN, there was no perfect preprocessing strategy. The key was to support even the smallest decision with some reasoning, avoiding extremes, and using architectures that handle this kind of mess well and build intuition through experimentation.

By progressively improving the dataset, adding a single meaningful feature (machine age), and transforming the target, I achieved an R^2 of around 0.93 — explaining 93% of the variance in a dataset that initially looked almost unusable.

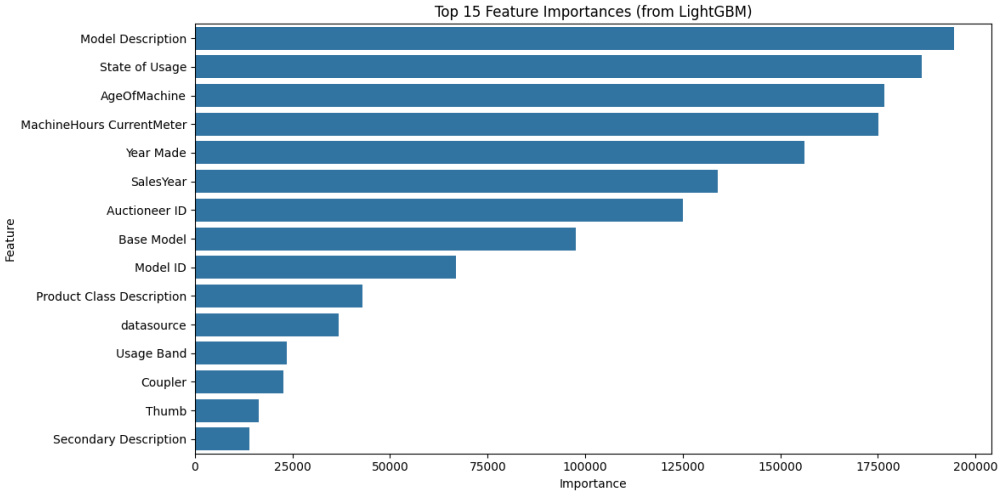


Figure 4: Top 15 feature importances from the fully trained LightGBM model.