Piotr Bugajski

Michał Woźniak

When searching for interesting articles we found an article called „reduction of information assymetry in the used car market using random forest method".

The article uses a dataset from the German used car market, sourced from Kaggle. However, the provided link is now expired, and no backup or alternative download instructions are given (this is the expired link - https://www.kaggle.com/orgesleka/used-cars-database). This makes it impossible for readers to access the same data and replicate the analysis. We later found the dataset on Kaggle and uploaded it into our github, but it was no easy task. Here's the link - https://www.kaggle.com/datasets/sijovm/used-cars-data-from-ebay-kleinanzeigen
As you can see, the dataset is not very popular on Kaggle (9 upvotes) so it was not easy to find it (low visibility) since it's diluted with other Germany car datasets that are more popular in their search engine.

Dataset description of variables, the most pivotal part of any modelling, is indeed included (371,528 observations) with the description of each variable.

dateCrawled – date of the last indexation by the web crawler
name – name of the car
seller – seller of the car. There are two types of sellers: individual sellers and car dealers
offerType – type of offer
price – price of the car as advertised
abtest (no explanation?)
vehicleType – vehicle type (estate, SUV, limousine, etc.)
yearOfRegistration – year in which the car was first registered. With this variable, it will be possible to calculate the age of the vehicle.
gearbox – automatic or manual transmission
powerPS – engine power measured in horsepower
model – vehicle model
kilometer – vehicle mileage. From a preliminary examination, one can expect understated values due to the seller's desire to make an unfair profit.
monthOfRegistration – month in which the vehicle was registered.
fuelType – propulsion type: petrol, diesel, electric, or hybrid
brand – make of the car
notRepairedDamage – binary variable indicating whether the vehicle has damage that has not been repaired
dateCreated – date the advertisement was placed on the website
nrOfPictures – number of photographs included in the advertisement
postalCode – postal code
lastSeenOnline – date of last activity on the advertisement

The description is fine but immediately after the authors say, "The dataset characterised above will be processed and analysed using the methods and analytical tools presented later in this paper."

Which is not presented later, because after these words there is a section 3 B called "Exploratory data analysis". Normally, in this section some graphs , summary statistics and interesting tidbits about variables would be introduced, information about outliers, missing values, problematic features, distributions. Anything that could help us reproduce this article. In the article however, this whole section is dedicated to explaining the definitions of descriptive statistics, why explanatory data analysis is a must and the definitions of data processing, modelling, algorithms and even data cleaning or what is “presentation of results”. Nothing about any data, nothing that could help us, just textbook definitions. Why?

Before commenting the results, the authors say: “In this article two stages will be analysed: Exploratory Data Analysis and Models & Algorithms.” We could agree with modelling (but to an extent) but there’s nothing about EDA at all.

- Data visualization
- Descriptive statistics
- Correlations
- Outliers
- Missing values
- Distribution analysis

There is nothing like that in there. Nothing that could help us in replicating these results at all.

Finally, the authors focus on modelling, Random Forest model to be exact. A 70/30 train/test split is mentioned, with random_state=0 in a code snippet. However, there is no explanation of whether the full dataset was used, whether the split was stratified, or any rationale for the chosen split.

The Random Forest algorithm was selected without justification or comparison to alternative or baseline models. No rationale for model choice is provided.

The tuning of hyperparameters appears to be done manually and ad hoc, with no systematic search or cross-validation. The process is not described in detail. The authors seem to have done this through the “trial and error” method.

Their main random forest model was evaluated with with 15 trees (n_estimators=15) and a minimum split size of 8 (min_samples_split=8). All other hyperparameters were left at default values.
The model was trained on the training set and evaluated on the test set. The initial model achieved an R2 of approximately 0.782 on the test set, meaning it explained about 78% of the variance in car prices.

Then, they tried tuning the hyperparameters to achieve a better result. The number of trees was increased to 25, which yielded no improvement in R2. Increasing the minimum split size to 12 reduced accuracy (R2 dropped to ~0.759). No systematic grid search, cross-validation, or robust hyperparameter optimization was performed—only a few manual tweaks were tested.

Two encoding methods were compared:

One-hot encoding: Expanded categorical variables into binary columns, increasing dimensionality (23 columns). This approach yielded higher predictive accuracy (R2 0.78) but increased computational cost and risk of sparsity.
Label encoding: Assigned integer codes to categories, retaining original dimensionality (20 columns). This method was faster but led to lower accuracy (R2 0.74), likely due to the model misinterpreting categorical variables as ordinal.

The authors note the trade-offs between these approaches but **do not** provide a systematic analysis or justification for the final choice.

It's hard to say there's any code but we have something we could base our reproducibility on. We have some small code chunks, for example:

```
In []: RFG_OHE =
RandomForestRegressor(n_estimators = 15,
min_samples_split = 8, random_state = 0)
       RFG_OHE.fit(X_trains, y_train)

Out[]: RandomForestRegressor(bootstrap=True,
ccp_alpha=0.0, criterion='mse',
                      max_depth=None,
                      max_features='auto',
                      max_leaf_nodes=None,
                      max_samples=None,
                      min_impurity_decrease=0.0,
                      min_impurity_split=None,
                      min_samples_leaf=1,
                      min_samples_split=8,
                      min_weight_fraction_leaf=0.0
                      ,
                      n_estimators=15,
                      n_jobs=None,
                      oob_score=False,
                      random_state=0, verbose=0,
                      warm_start=False)
```

So there is some code (actually the "results" section is the only section with the code, and it's not like there's a lot of it) so we could reproduce some parameter tuning and random forest training. However, as mentioned before, there is nothing for the missing values, encoding, feature engineering, transformations, what they did with outliers.

Results are presented as point estimates with little critical analysis or discussion of limitations.

To sum up again, from reproducible research perspective:

- **Data inaccessible due to expired link and lack of backup**
- **No full codebase or computational environment details**
- **No detailed pipeline for preprocessing or feature engineering**
- **No baseline models or comparison**
- **Manual and limited hyperparameter tuning**
- **No validation of overfitting or robustness**

Therefore we tried to replicate it. And upgrade it, because we see a lot of mistakes.

Before we begin, to best reproduce our code, please use the libraries version used below:

numpy==2.2.6

optuna==4.3.0

pandas==2.2.3

scikit_learn==1.6.1

Let’s get into the modelling.

First, in the preprocessing part:

- we calculated the age of a car as a difference between the time the ad was scraped and the registration year.
- We also dropped all old cars (>30 years old), new cars (current and later model years) and errors to ensure the model is robust.
- We dropped all damaged cars or those with unknown mechanical state
- We also removed variables below (with justification):

  seller: only 1 observation has a different value than the rest
  name: includes information that other variables provide, too granular in its current form
  model: too granular in its current form, vehicleType and brand should be enough to model a price of a car
  offerType: only 3 observations have a different value than the rest
  abtest: irrelevant to the study
  postalCode: irrelevant to the study
  lastSeen: irrelevant to the study
  dateCreated: irrelevant to the study
  nrOfPictures: all values are equal to 0

Overall, we dropped from 371,528 observations to 220,891.
After cleaning and removal, we ended up with these variables as our predictors:

- **Numerical Features:**
  - **<u>price</u>** (target variable)
  - powerPS
  - kilometer
  - age
- **Categorical Features (encoded):**
  - gearbox (manual/automatic, encoded as 0/1)
  - fuelType (encoded as 0/1)

Also, we presented a simple statistical summary (after cleaning and encoding):

- price: mean ≈ 7,287; median = 4,499; min = 1; max = 200,000 (before cleaning we had prices like 99999999 for example. Do we even know if the authors of the article removed that?)
- powerPS: mean ≈ 133; min = 1; max = 20,000
- kilometer: mean ≈ 122,425; min = 5,000; max = 150,000
- age: mean ≈ 11.9; min = 1; max = 30
- gearbox and fuelType: encoded as binary variables

With our dataset ready for modelling, we defined our predictors (X) as all columns except price and set y = price as the target variable. We split the data into training and test sets using a 70/30 ratio and fixed random_state=2812 to ensure reproducibility of our results.

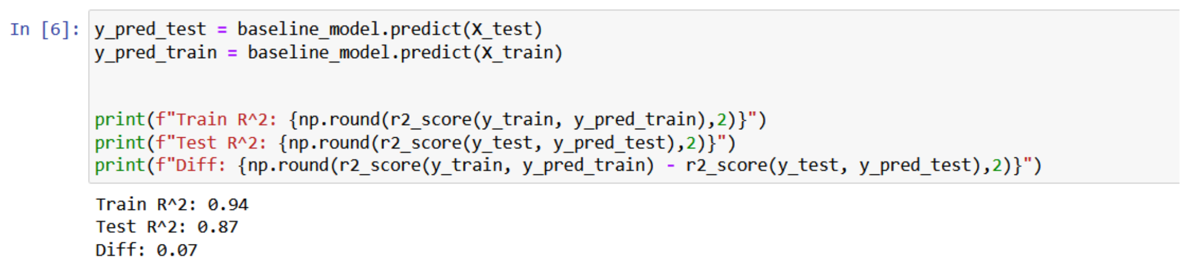
We began our modeling process by establishing a clear, reproducible baseline. We used the RandomForestRegressor from scikit-learn, setting the key hyperparameters to match those reported in the article:

- n_estimators=15
- min_samples_split=8

we also added two small tweaks:

- n_jobs=-1 (to leverage all CPU cores)
- random_state=2812 (as mentioned before, for reproducibility)

The model was trained on the training set and evaluated on the test set using the R^2 score. This provided a direct comparison point to the article's results and a foundation for further improvement.

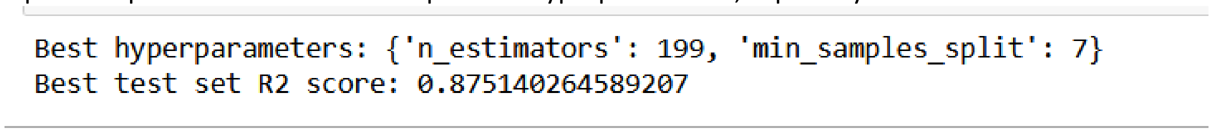


```
In [6]: y_pred_test = baseline_model.predict(X_test)
        y_pred_train = baseline_model.predict(X_train)


        print(f"Train R^2: {np.round(r2_score(y_train, y_pred_train),2)}")
        print(f"Test R^2: {np.round(r2_score(y_test, y_pred_test),2)}")
        print(f"Diff: {np.round(r2_score(y_train, y_pred_train) - r2_score(y_test, y_pred_test),2)}")

        Train R^2: 0.94
        Test R^2: 0.87
        Diff: 0.07
```

The correct preprocessing alone boosted our R2 way higher than the article's, without applying optimal hyperparameter tuning.

One of our main improvements over the article was the introduction of systematic, automated hyperparameter tuning using **Optuna**. Optuna is an open-source hyperparameter optimization framework that efficiently searches for the best hyperparameter values for machine learning models. Unlike the article, which only tried a handful of manual tweaks, we used Optuna to objectively and reproducibly identify the most effective model settings. We defined an objective function that trained a RandomForestRegressor with hyperparameters suggested by Optuna and returned the R^2 score on a validation set.

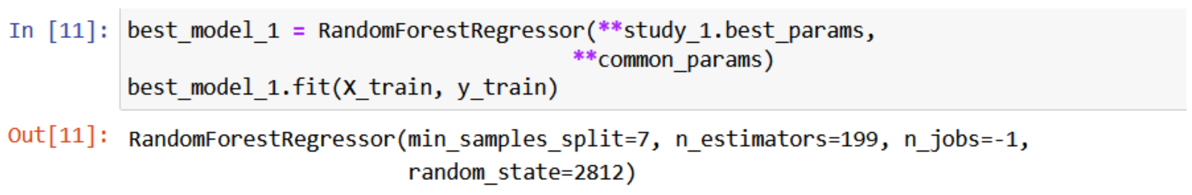
Optuna explored a defined search space for hyperparameters, especially:



```
Best hyperparameters: {'n_estimators': 199, 'min_samples_split': 7}
Best test set R2 score: 0.875140264589207
```

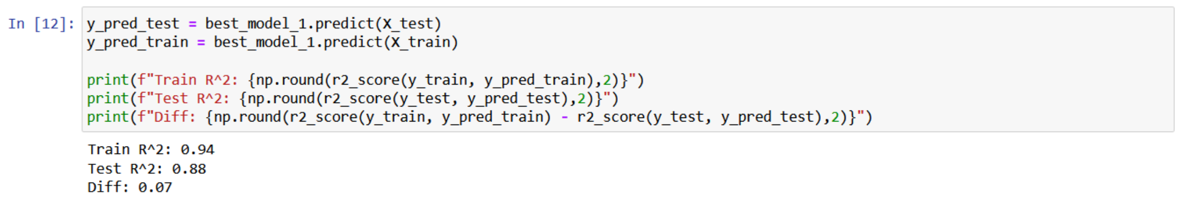
- n_estimators (number of trees, e.g., up to 199)
- min_samples_split (minimum number of split samples)

The optimization process involved running multiple trials, each time training a model with a new set of hyperparameters and evaluating its performance. After optimization, Optuna reported the best hyperparameters, which we then used to retrain the model on the training data.



```
In [11]: best_model_1 = RandomForestRegressor(**study_1.best_params,
                                              **common_params)
         best_model_1.fit(X_train, y_train)

Out[11]: RandomForestRegressor(min_samples_split=7, n_estimators=199, n_jobs=-1,
                               random_state=2812)
```

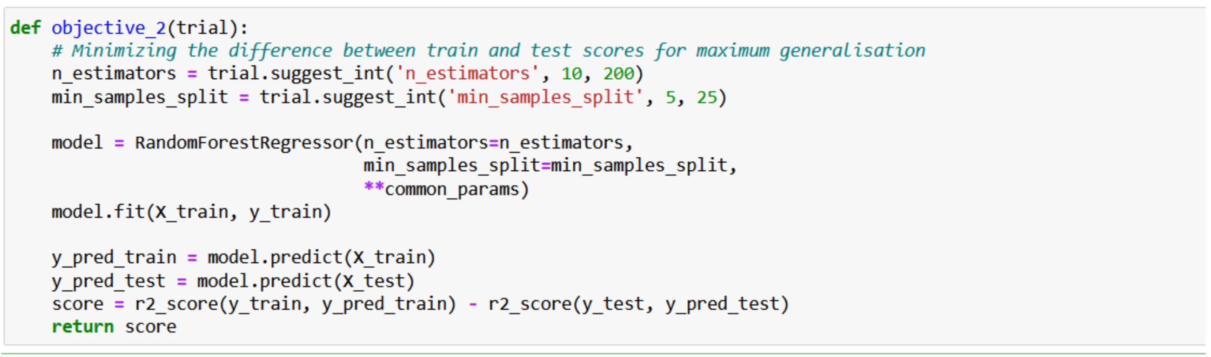
And as a result of this training, we got the result below:



```
In [12]: y_pred_test = best_model_1.predict(X_test)
         y_pred_train = best_model_1.predict(X_train)

         print(f"Train R^2: {np.round(r2_score(y_train, y_pred_train),2)}")
         print(f"Test R^2: {np.round(r2_score(y_test, y_pred_test),2)}")
         print(f"Diff: {np.round(r2_score(y_train, y_pred_train) - r2_score(y_test, y_pred_test),2)}")

         Train R^2: 0.94
         Test R^2: 0.88
         Diff: 0.07
```

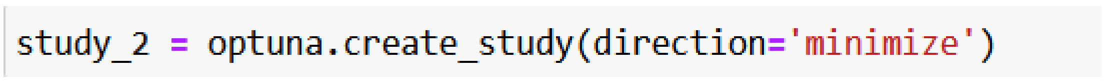
Although the train r2 didn't change from the baseline train model and the test R2 only changed very slightly, thanks to optuna we got the optimal hyperparameters. Now while increasing the overall R2 wasn't achieved, we tried to optimise the difference between train and test to be smaller as to reduce overfitting using optuna's optimize() function.

First, we defined our trial for optuna to try and minimize the overfitting:



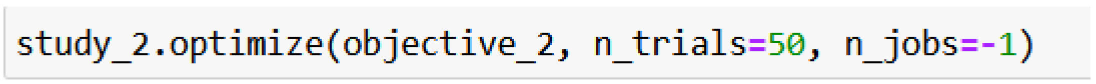
```
def objective_2(trial):
    # Minimizing the difference between train and test scores for maximum generalisation
    n_estimators = trial.suggest_int('n_estimators', 10, 200)
    min_samples_split = trial.suggest_int('min_samples_split', 5, 25)

    model = RandomForestRegressor(n_estimators=n_estimators,
                                  min_samples_split=min_samples_split,
                                  **common_params)
    model.fit(X_train, y_train)

    y_pred_train = model.predict(X_train)
    y_pred_test = model.predict(X_test)
    score = r2_score(y_train, y_pred_train) - r2_score(y_test, y_pred_test)
    return score
```

The code chunk checks the range of n_estimators from 10 to 200 and min_samples_split from 5 to 25 and searches for the best combination out of those using this function from optuna:



```
study_2 = optuna.create_study(direction='minimize')
```

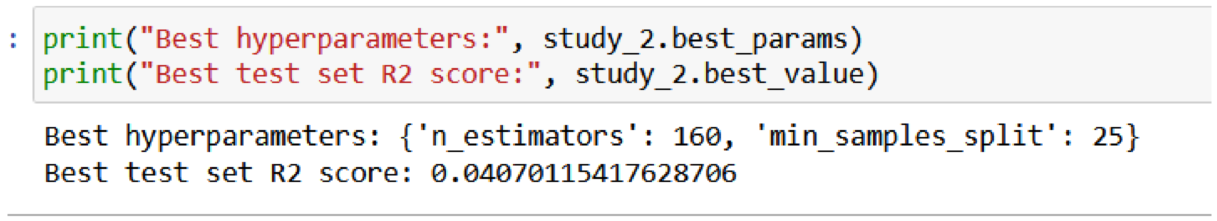
After the optimization target ('minimize') has been declared we run



```
study_2.optimize(objective_2, n_trials=50, n_jobs=-1)
```

To allow optuna to test 50 different combinations.

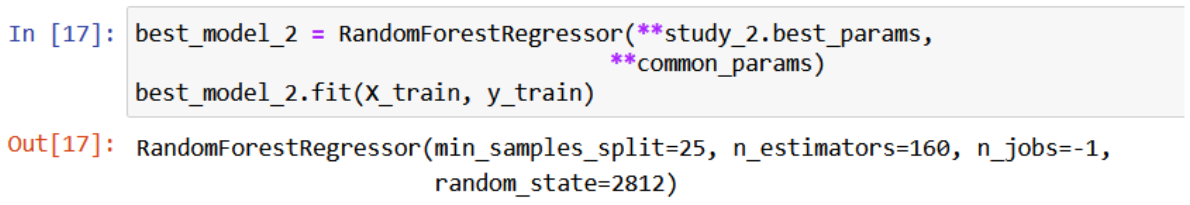
As a result:



```
: print("Best hyperparameters:", study_2.best_params)
  print("Best test set R2 score:", study_2.best_value)

  Best hyperparameters: {'n_estimators': 160, 'min_samples_split': 25}
  Best test set R2 score: 0.040701154176287O6
```

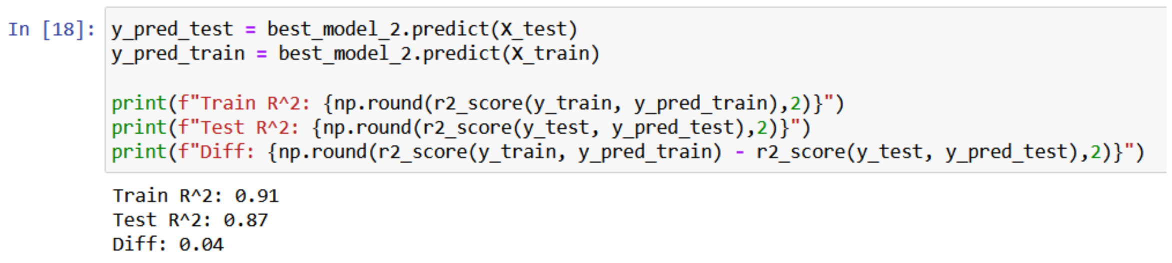
Optuna found the best hyperparameters that minimize the difference between train and test the best is the combination of 160 estimators and 25 minimum split samples.

We then used this combination to train the optimized model:



```
In [17]: best_model_2 = RandomForestRegressor(**study_2.best_params,
                                              **common_params)
         best_model_2.fit(X_train, y_train)

Out[17]: RandomForestRegressor(min_samples_split=25, n_estimators=160, n_jobs=-1,
                               random_state=2812)
```

And tested it:



```
In [18]: y_pred_test = best_model_2.predict(X_test)
         y_pred_train = best_model_2.predict(X_train)

         print(f"Train R^2: {np.round(r2_score(y_train, y_pred_train),2)}")
         print(f"Test R^2: {np.round(r2_score(y_test, y_pred_test),2)}")
         print(f"Diff: {np.round(r2_score(y_train, y_pred_train) - r2_score(y_test, y_pred_test),2)}")

         Train R^2: 0.91
         Test R^2: 0.87
         Diff: 0.04
```

While the overall r2 dropped slightly, the difference between train and test is almost twice as smaller (from 0.08 to 0.04) as before which is what we wanted. Our model is very precise.

To sum up, the article was hard and quite frustrating to replicate, and during said replication we found a lot of shortcomings and things we could optimize their results.

The entire data as well as the code for both can be found on our GitHub.