Predictions on listed housing units in Liverpool

A volunteer data science project

Carlos Tiago Arinto

December 2024

Contents

List of Figures

Chapter 1

Introduction

The aim of this project is to showcase some of the skills acquired in the author's academic background, while voluntarily and independently developing a project in a topic of interest.

The field of data science encompasses a comprehensive spectrum of knowledge. It ranges from acquiring to storing, preparing and visualising data, while also making it work with predictive algorithms, themselves with adjustable parameters, so that new information can be extracted. Lastly, interpreting the results obtained from the acquired models, often through chosen metrics, is a final but important step to include. All of these stages welcome, and even require, an underlying general understanding of the data being worked with.

Zeroing on real estate listings, the conducted work's main goal is to build a realistic system, without relying on any pre-existent dataset, that would enable a user to, as accurately as possible, estimate what features a desired home would have, given the input of others. As a secondary objective, there was also an effort to include a practical procedure to visualize the obtained data, allowing for a quick and intuitive way to take conclusions from it, without the need of any programming skills. The fulfilment of these tasks presents the challenge of connecting the collection, persistence, training and display frameworks, which this project meant to accomplish. An emphasis was also put on comparing different data preprocessing approaches and machine learning algorithms, with each target variable having over 100,000 possible model set-ups, from the implemented methods.

This report will attempt to present a detailed overview on how the project was built, relevant "under the hood" details, and results, therefore passing through multiple stages of the complete execution of a data science project.

Chapter 2

Code of ethics

This non-commercial educational project uses data held publicly available by Zoopla (ZPG Ltd.).

Despite being a legal practice, to web scrape ethically, it is important to have a look at the website's *robots.txt* [2] page, which tells which url subdirectories have, or have not, got permission to be accessed by the scraper. With the ones used for the data collection being "/for-sale/houses", "/for-sale/details" and "/new-homes/details", no disallowance was found with respect to scraping these subdirectories, as of the time of the writing of this report.

No data which requires privileged access (ex: log-in), sensitive data belonging to an individual or any other non-publicly available source were used.

There was a further attempt to avoid any copyright infringements, by ensuring the GOV.UK guidelines [3] under the "Text and data mining for non-commercial research" section were taken into consideration. This allows researchers to "make copies of any copyright material for the purpose of computational analysis" if they have lawful access to the work, and for non-commercial research.

This project was done in good faith for educational and non-commercial reasons. It must not be used in any other context. If so, the user may be liable to any actions ZPG Ltd. decide to take.

Should a party belonging to ZPG Ltd. find a disagreement with any of the claims made above and consider itself aggrieved, it is welcome and encouraged to contact the author so it can be resolved.

A similar disclaimer is also present in the code file responsible for the scrape.

Chapter 3

Technologies used and software integration

The following diagram (3.1) contains an overview on the programming tools used as well as the overall software integration:

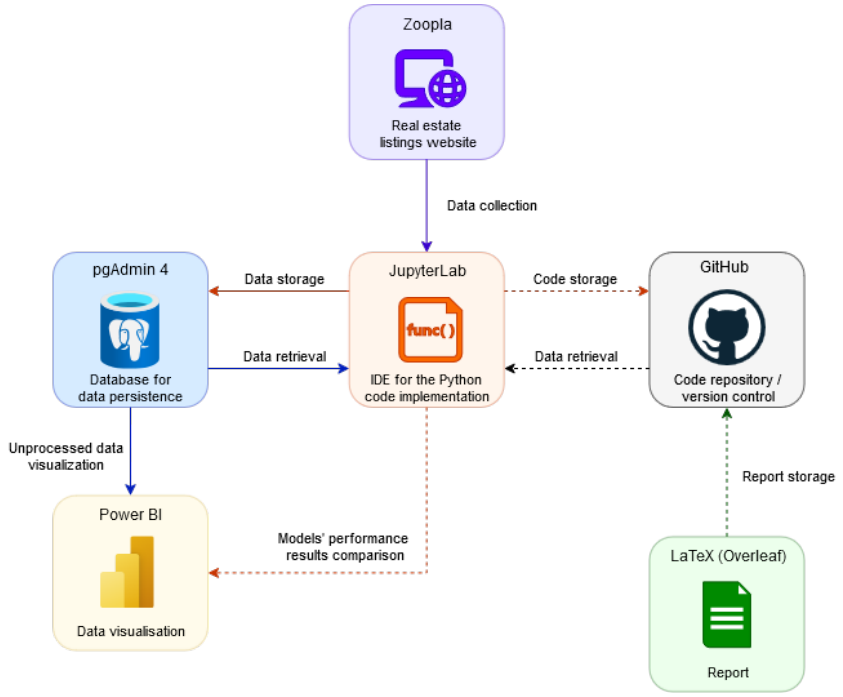


Figure 3.1: Website and software tools and integration overview. The solid lines correspond to direct/automatic integration, and the dashed ones to a manual or indirect interaction.

Table 3.1 displays in a more detailed manner the software, programming languages and packages used, as well as their general purpose:

The main code was done using Python (3.11) on a JupyterLab IDE, using Pandas/Numpy for data manipulation, psycopg2 for the database integration, and sci-kit for the majority of the data preprocessing and entirety of machine learning models. The database was built with PostgreSQL, and Microsoft Power BI was the tool providing the visualization of the data. GitHub was used as a repository and for version-control, and lastly the present report was done using LaTeX.

Software	Programming/Formula Language	Library/ Package	Function
JupyterLab (8.11.0)	Python (3.11.2)	beautifulsoup4 (4.12.2)	HTML parser which enabled the collection of data via web scraping
		psycopg2 (2.9.9)	Adapter allowing queries done in the Python environment to connect to, and be executed in, the database server
		pandas (2.1.1)	Provider of convenient data structures and operations, facilitating the manipulation and analysis of data
		scikit-learn (1.4.2)	Supply a multitude of algorithms for the various stages of data preprocessing, training and tuning of prediction models, and performance metrics to assess their capabilities
		parallel-pandas (0.6.4)	Parallelizing the execution of (pandas') data-related tasks, with the aim of increasing the code's performance and scalability
pgAdmin 4 (7.5)	(Postgre)SQL (15)	n/a	Queries for a preliminary analysis and understanding of the data
Microsoft Power BI (2.138.1452)	DAX	n/a	Queries on data whose results can then be displayed in visuals and dashboards
Overleaf	LaTeX(2e)	multiple	Formatting options and media displaying

Table 3.1: Software, programming languages, and packages used, alongside their respective function.

Chapter 4

Development stages

4.1 Data collection

On the Internet, there are multiple available datasets, for the most varied topics. However, during planning, there was a desire to build a personal one.

A script was written using *Beautiful Soup* to web scrap a listings website (Zoopla), filtered to results in Liverpool, for buying/selling (not renting) a property. One by one, every listing had some properties of interest collected, such as price, tenure, number of bathrooms, energy efficiency, etc... which were afterwards stored in a database.

All available listing pages were accessed, yet the nature of the collection meant the retrievable data would be somewhat limited to what was available, with the option to collect more catalogued houses in the future, should it prove necessary.

In the end, a total of 1890 listings were scraped, with 26 features each.

4.2 Data persistence

To save the data, an SQL database was used. It offers advantages compared to a spreadsheet program (ex: LibreOffice Calc, Microsoft Excel), more noticeably:

— easy of use of queries instead of formulas;

— separation of data and calculations;

— clearer representation of table relationships.

Other positive aspects, such as scalability, may be less noticeable on the current number of rows and dimensions, but they remain a good habit to get used to, both for expanding the current project or work in different ones.

PostgreSQL was the relational database management system of choice. Relational "won" over non-relational since scalability, in both computational time and space, would unlikely be an issue, while the user-friendliness was a welcome perk, cemented with the pgAdmin management tool user interface. PostgreSQL was picked over other RDMSs given its versatility and increasing popularity, characteristics that helped it seem like a good long-term choice.

Figure 4.1 depicts how the database was designed, with an Entity Relationship Diagram (ERD) containing the table relations, the features data was collected for, and the respective type specifications.

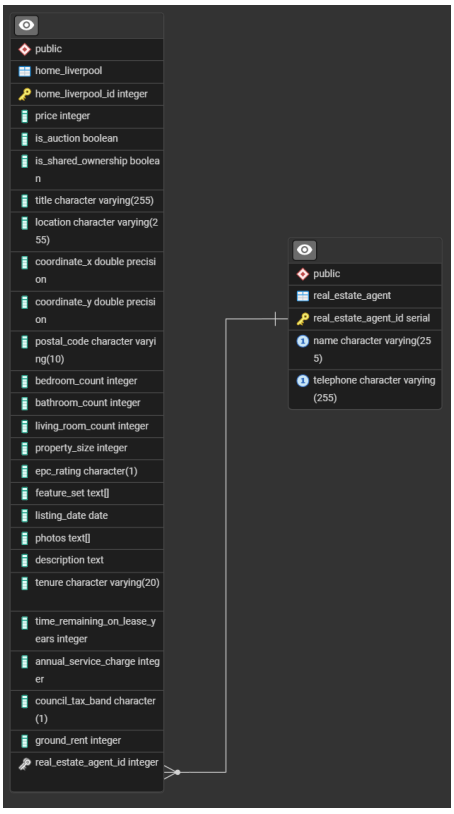


Figure 4.1: ERD displaying how the project's database was designed.

4.3 Creating a model

4.3.1 Data preprocessing

The preparation of the information that would feed the models comprised several steps. Research was done not only on each one of these individually, but also on how to amalgamate them in an order that could be, at least plausibly, the most logical.

Debates can be found on which arrangement is the ideal. Some parts seem to generate a certain amount of discussion amongst the community, others are more easily agreeable. Figure 4.2 depicts the preprocessing ordering decided for this project.

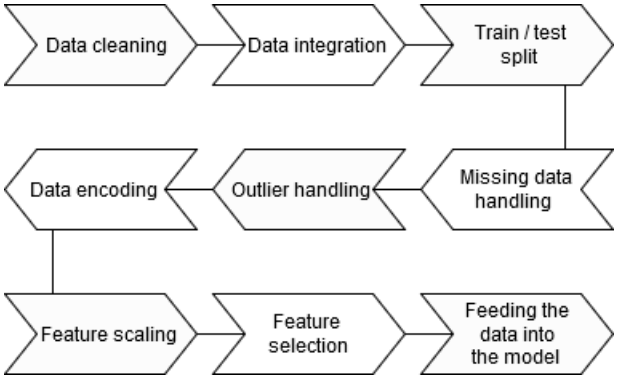


Figure 4.2: Preprocessing pipeline carried out.

Rather than focusing on hyperparameter tuning (although not ignoring it), a decision was made to implement various algorithms in many of these steps. Then, the model's performance results comparison is instead made on how these multiple methods interacted with the collected data, and with each other. More on this in the Results section.

What follows is a brief description on each preprocessing pipeline constituent, the reasoning behind where in the program (or "when", logically) it was placed, and, if applicable, the algorithmic options made available on it.

4.3.1.1 Data cleaning

Description - With the dataset formed, it is now time to ensure its correctness. This includes cleaning table names (e.g., typos, spelling differences), removing duplicates, and guaranteeing variables in homologue columns are

in the same as well as correct data types. Sometimes, feature engineering is also necessary, as was the case with this dataset. It will shortly be explained why.

Use - Personally handling the collection and persistence of the data allowed skipping some of the cleaning tasks, since, during the process, they had already been ensured. The "primary key" and "unique" constraints guaranteed there were no unwanted duplicates. Variables were cast to the correct types during the scrape, and their accurateness was ensured once more by the database column types they were saved into, with an exception. The listing date "datetime" type, despite being persisted as a *date* type column, is returned from the database as a Python *object*, which was then appropriately converted into Pandas' *datetime64*. Afterwards, the listing date's *yyyy-mm-dd* format was divided into the integers *year*, *month* and *day*, since, as far as the author knows, models are unable to interpret "datetime" type objects, even those aimed at time series prediction problems, calling for this feature engineering adjustment. This split also greatly facilitates subsequent encoding and mathematical operations.

Ordering rationale - The mentioned assurances are crucial to guarantee the data is treated appropriately in the later steps. A consequence of not doing so could be found immediately in other scenarios, as a spelling nuance on a table's foreign key name could already cause an error when trying to merge it with the primary key of its homologue column. If these same columns were assigned different data types, a similar issue would arise, making the cleaning an essential, and arguably mandatory, inaugural task.

4.3.1.2 Data integration

Description - This procedure is responsible for bringing together data from multiple sources, such as datasets, or simply tables. Multiple benefits stemming from this exist. Consolidating data from multiple tables/dataframes in order to provide a unified, global analysis and seamlessly apply policies and procedures when needed is the one that will be focused on here.

Use - The dataframes corresponding to the tables *real_estate_agent* and *home_liverpool* were joined on the *real_estate_agent_id* column, bringing the whole dataset into one united data structure.

Ordering rationale - With the train/test split phase being a precursor for the remaining steps, the only real decision was whether to integrate before or after it. The author went with before since doing it after the splitting would mean the merge has to be done twice rather than once, with no apparent benefit.

4.3.1.3 Train/test split

Description - This act of partitioning the data consists in dividing the data into two subsets (in terms of number of samples, as opposed to column-wise), so that one of them (usually the larger) is fed into and trains the model. The other subset is meant to simulate new, unseen data, as if their target variables are unknown (but we actually know them), so that we can then compare the model's predictions on this subset with its true, known values. Intuitively, these two groups are referred to as the *train* and *test* data. To reiterate, this act serves as a means to measure the performance of predictive models. Without this, and assuming all the data was used in a single training operation, then there would either be no way to test the capabilities of the model beforehand, or we would have to use data already used during the training to test it. The latter is also problematic, as the model would after all be trained with the very data it is meant to predict, resulting in optimistic but deceptive results, which may or may not reproduce their success when presented with new, unseen data. Despite this being the method used, it is worth noting that, in some cases, it can be more advisable to use an alternative way to test predictive models. For datasets considered small or unrepresentative , which is to say they do not possess enough value variety, or they do but in atypical proportions, this method can be unideal. This is because firstly, if the dataset is already limited, further reducing it to have a test subset could lead to an even higher bias; secondly, if it contains a disproportionate amount of values of one category (or a certain numeric importance), then an unfortunate split could put most of the minority on the test set rather than the training, meaning the model did not train for and does not know how to predict for them. More complex methods such as *k-fold cross validation* could circumvent this.

Use - The train/test split was done using random sampling (with a seed for reproducible results) in a 80/20 proportion, for the train and test sets respectively. At the time of implementation, it was uncertain how much data would be scraped, as it could be done periodically, making the dataset size relatively unknown. Nevertheless, this method, and split ratio, seemed capable of providing a reasonable performance. The author does not, however, discard an implementation of *k-fold* as a possible alternative.

Ordering rationale - All the later preprocessing steps are going to implement changes that should only be done either in the training data, such as Handling Missing Data, or based on it, like Feature Scaling. The algorithms used will, as a general rule, analyse all the data points available to them, before deciding how they will carry their functions on. As a quick example, scaling the data with mean normalisation would require knowing the average of a certain feature. If the whole dataset is considered, it will

have a more precise but deceptive scaling, since it will be making use of the data it is supposed to predict to get a more accurate scale. Thus, data that it is not supposed to be known is already weighting on what will be fed to the model's algorithm, introducing bias. This phenomenon is known as "data leakage". Even handling missing data could be influenced by splitting at a later stage, if the procedure is to simply delete data points with missing entries. In any case, if, otherwise, there is imputation, then not only can the imputation method be relying on more values than those it is supposed to know, to decide how to impute, but, understandably, it would make little sense to impute what we are trying to predict. For these reasons, the catalyst step to measure a model's performance, train/test split, was done at this point.

4.3.1.4 Handling Missing Data

Description - Dealing with missing data is a common and important step for various reasons. For openers, some machine learning (and even preprocessing) algorithms do not have a way to interpret missing data, rendering them unable to work with it. Secondly, it can be beneficial to fill in those values so as to preserve as much data as possible (provided the percentage of missing values is not too large, to control bias). Lastly, understanding the nature of the missingness enables the choice of an appropriate solution so that the model is not wrongly biased with wrongful imputations or deletions. The missingness nature can be catalogued in three ways:

MCAR - Missing Completely At Random categorizes data that is missing stochastically, which is to say, it has no relationship with other features in the dataset, including unobserved data. It is also characterized for not following any understandable pattern;

MAR - Missing At Random differs from the previous in the sense that the data is missing for reasons that can be determined through other features of the data. Its nature does not depend on the "unknown" but does so on the observed data;

MNAR - Missing Not At Random, also referred to as "non-ignorable missing data", is deemed so when there's some logic behind why it is missing, which is understood or inferable through the data we have. It thus depends on the unobserved data (we can think of it as whatever data would be necessary to allow us to understand which external factors are interfering with it and how).

While both MCAR and MAR are tolerant to simply deleting or filling in absent entries, for MNAR this wouldn't be possible, as the underlying

cause needs to be taken into account. For this case, data can not be simply removed or replaced without employing specialized methods, such as Multiple Imputation or Bayesian Inference [1].

Use - Although not always possible to say with certainty in which of these characterizations the data falls on, the author believes, through some examination, that the missing data in the many categories does not follow any noticeable pattern. It is likely caused by the home owners not providing the real estate agents with all the house specifications, or the latter not having yet updated them on the website. This would mean there's no apparent reasoning or connection with the already observed data, deeming it MCAR. For this reason, there would be no impediment in employing the deletion and imputation of the data with the following algorithms:

Numerical

- **Mean imputation** - The missing data is imputed with the mean (average) column value;
- **Median imputation** - The median value of the column fills the empty entries. Let N be the number of rows and C_n the $n(th)$ element of column C:

$$\begin{cases} C_{\frac{N+1}{2}}, \text{ if N is odd} \\ \\ \frac{C_{\frac{N}{2}} + C_{\frac{N}{2}+1}}{2}, \text{ if N is even} \end{cases} \tag{4.1}$$

  which in words means nothing more than the "middle" value of those available, if N is odd, or the average between the two central ones, if N is even.

- **Forward Fill imputation** - When a value is found missing, it is filled with the last observed value. Also, let N be the number of rows and $n \in \mathbb{Z}^+ : n \in (1, N-1)$, when the initial n rows need to be imputed, they do not yet have a value for Forward Fill to use on them. For these cases, Backward Fill was used (only on those opening entries), filling them with the next observable value instead.

Categorical

- **Mode imputation** - The missing entries are filled with the category that occurs in the data the most. In case of a draw, the first one in alphabetical precedence is used.

Variables of the *boolean* type could also have made use of Mode Imputation, but other types, including lists or special numerical cases such as

the *day*, *month* and *year* components, require special care. That discussion will, however, fall outside of the scope of this Liverpool houses case-study, as these "special" features did not require any missing data handling.

Lastly, all data types share the method of simply deleting the data point if its value is missing in the feature being checked.

Through some previous selection done in 4.3.1.4, features with a missing data ratio bigger than a customizable parameter, here chosen as $m = 0.5$ ($> 50\%$ missing), were removed.

Ordering rationale - Handling missing data has both arguments in favour and against being done before steps like Handling Outliers or even Feature Scaling. Being done afterwards has the benefit of not having imputed data interfering with the scale used, or with what is considered an outlier. However, understanding what constitutes a removable or a "keeper" outlier can be prone to a certain degree of human or algorithmic error, meaning an outlier that is removed during outlier handling may be part of a data point that offers a valuable representation of the data, and should be considered for imputation. Thus, handling missing data beforehand could be a way to balance the meaningfulness of outliers, allowing them to (positively or negatively) influence this step, but be potentially removed on the one afterwards, and not influence the remainder of preprocessing. Finally, algorithms such as *sklearn*'s Target Encoder are unable to process missing data, reinforcing the need to deal with it before scaling.

4.3.1.5 Handling Outliers

Description - Outliers are values which can skew the data in a wrong and often noticeable way. There are mainly two types: they can be natural, meaning despite very deviated from the mean or median, they are possible, or they can be aberrant, if their origin lies on data collection mistakes, typographical errors, sensors malfunctioning or other anomalies. This distinction is relevant as the former can be kept if considered a valid representation of the data (such as a minority which needs to be accounted for), whereas the latter is better off discarded. Training a model with unrepresentative data contributes to having it adapt to data it is unlikely to find in new observations, a phenomenon known in the literature as overfitting. This step is thus relevant to improve the quality of the model, and it is done in two stages: first, one needs to decide what constitutes an outlier, which can be done with expert elicitation or with functions that make assumptions on the distribution of the data and mark their own "boundaries"; secondly, it is time to choose what to do with those outliers. They can be not just numerical, but in some cases categorical, when certain classes have an unexpectedly high or low number of observations.

Use - The outliers detected were seen as natural rather than aberrant. The level of certainty of this judgment comes from the fact that a deviated value on a feature such as *price* would follow a certain behaviour in the order of magnitude of features such as *number of bedrooms*, or *tenure* in terms of the expected class. This lowers, to an extent, the probability of abnormal values being due to human or technological errors.

4.3.1.5.1 Detecting Outliers

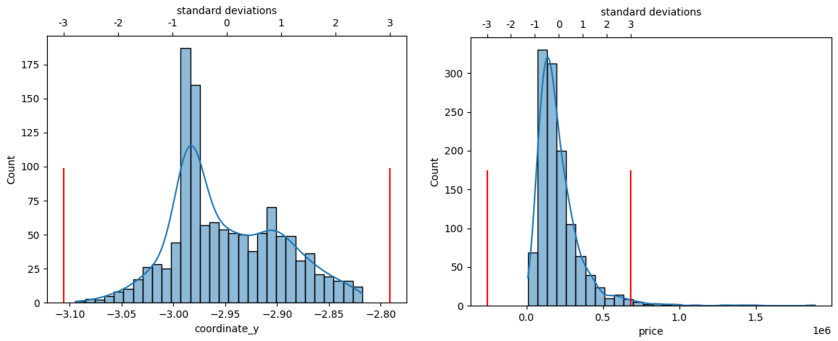
Multiple techniques were used to expose outliers, each with a configurable way to be more or less lenient in the thresholds established. They can also interact better or worse with certain data distributions. In this implementation, the "sknewness" of each feature's distribution was calculated to gatekeep the identification of outliers from taking place, when the method being used is less appropriate for the given distribution's shape. The implemented techniques were as follows:

- **Z-score -** More suitable for normal distributions, it is so due to its detection method relying on a standard deviation based threshold.

  Let ν represent the minimum or maximum non-outlier numerals, σ be the unit for standard deviations, μ the mean of the feature being inspected and Z the *Z-score* coefficient indicating the intended number of standard deviations the thresholds should be at, we have:

  $$\nu_{min} = \mu + Z * \sigma$$
  $$\nu_{max} = \mu - Z * \sigma.$$

  Image 4.3a shows the distribution of the variable *coordinate y* used in this experiment. Using $Z = 3$, the red lines denote the boundaries outside of which the values will be deemed outliers. The plotted kernel density estimate line provides a smoothed, probabilistic view of the distribution. It is now possible to have a more accurate assumption on how skewed the distribution is.



(a) *Coordinate y*'s distribution (normal) (b) *Price*'s distribution (positively skewed)

Figure 4.3: In each graph, the red lines represent the boundaries obtained using this method for outlier consideration, and the plotted line a kernel density function for smoothing purposes. The top x axis showcases, in sigmas, how deviated from the mean the observations are.

With s as the skewness, only features with $-0.5 < s < 0.5$ were analysed by this method. In 4.3b we can see how a *Z-score* based method does when presented with a skew. On the side of the "tail", it sets a relatively more forgiving limit, resulting in less values being considered as outliers. Since this would be an unideal practice, the skewness filter was added, restricting the application of outlier detection methods only to the features whose distributions they are suitable for.

- **Inter Quartile Range -** In a skewed distribution, the inherent more extreme values would impact where in the distribution the mean would be situated. On that note, using a technique which relies on the evaluation of the distance to the mean, such as *Z-scores*'s σ, to identify outliers, would likely prove less effective (causing more forgiveness on the "tail's" extreme values, since they shift the mean, and thus the centre of σ). This is where IQR can step in as an alternative. It combines the use of percentiles, instead of deviations, with a predefined coefficient. The use of percentiles establishes thresholds based on the number of observations, more concretely, between the first and third quartiles. This results in its consideration having the median, rather than a mean, as a "centring" point. Therefore, it is not influenced by how extreme the values are, which is to say the length of the skew's tail, or lack of one on the opposite side. The coefficient essentially extends, or shortens, the threshold for outlier

consideration, engendering more lenience or restriction in where this discrimination begins. That value is usually 1.5, to approximate the limits to that of 3σ (3 standard deviations), but let us remember that these "deviation-like" limits wouldn't have the mean as its centre, but the median,

The form for IQR is given by:

$$IQR = Q3 - Q1$$
$$\nu_{min} = Q1 - (1.5 * IQR)$$
$$\nu_{max} = Q3 + (1.5 * IQR)$$

Where Q1 and Q3 are the values corresponding to the first and third quartile (25^{th} and 75^{th} percentile) limits respectively.

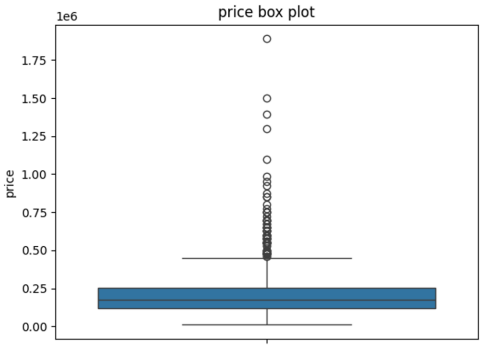


Figure 4.4: *Price*'s IQR and outliers.

Using the *price* feature as an example, notice how 4.4 shows the difference between a mean (4.3b) and median approach. It is less affected by extremes, namely the more radical ones (such as those above £1 million). By using the coefficient that tries to approximate the limits to those of 3σ, we have a more fair (comparable) view on how the median centring is less shifted towards the extremes (since they are radical but not plentiful) and establishes its limits "sooner". In practice, this translates to the upper outliers being found around the £500.000 mark rather than the £650.000 in 4.3b, the former being likely a more sensible outlier characterisation for this variable.

- **Percentile Threshold -** Similar to IQR, PT uses percentiles to establish the boundaries for outlier consideration. Whereas IQR is

adjustable with its coefficient, PT has the flexibility of not being "bound" to quartiles and thus able to use any chosen percentile. The main difference lies in the fact IQR is more impacted by the observations between the 25^{th} and 75^{th} percentile to pick its bounds. They can, however, have one other meaningful difference. There seems to be some omission in the literature specifying whether IQR's limits can in fact extend beyond the minimum and maximum observed values. The norm in how the algorithm is explained appears to suggest it is indeed possible to go over the observed extremities. This is relevant since, due to a lack of coefficient, the "pure" percentile approach would instead guaranteedly place its thresholds within the pool of observed values. In a situation where the "extremes" are common, such as the *number of bathrooms* of a house being "1", this difference would mean that all of these houses would be deemed outliers by PT, but not necessarily by IQR. To conclude, this method is guaranteed to catch outliers, unlike IQR, and resilient to skewed distributions as opposed to Z-score, but perhaps less advisable in distributions where the extremes are abundantly populated.

In order to calculate the thresholds we have:

$$\nu_{min} = \alpha_p$$
$$\nu_{max} = \alpha_{100-p}$$

where α_p is the value corresponding to the pth (lower) percentile, while $100 - p$ refers to the $100 - p$th (upper) percentile.

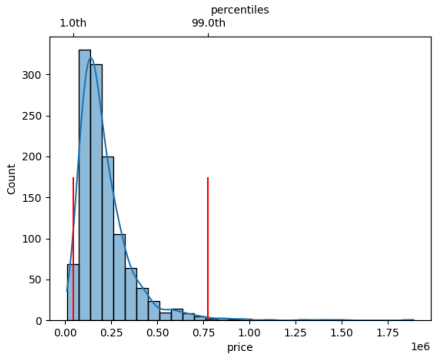


Figure 4.5: *Price*'s distribution marked by the 1^{st} and 99^{st} percentiles.

Unlike *Z-score* or IQR, lower outliers were now detected by using PT (4.5). This distinctiveness could have been beneficial in detecting

some auction listings, whose prices are biased but relatively close to the mean and median (although an *is auction* feature exists in this dataset to help identify such listings, it could have not been the case). Other features, however, did not benefit from PT in the same way. This will be explored in more detail in the Trimming segment ahead.

4.3.1.5.2 Removing Outliers

- **Expert Elicitation -** Technically a detection approach, a custom method can be deployed to remove specific values chosen by an expert. This human-based input makes use of a person's experience and understanding of the data to make tailored observation removals. As shown in 4.6, the most expensive house had a price difference to the second costliest equal to nearly two times (1.86 to be more precise) the average house price. Similarly, *coordinate X* had one value noticeably more separated to the others (4.7). The same happened with the feature *home_liverpool_id*, with an id number significantly lower than the others, possibly from a house that has been listed for a larger amount of time. This last case was, however, not necessary to handle as the feature was, at this point, already removed.

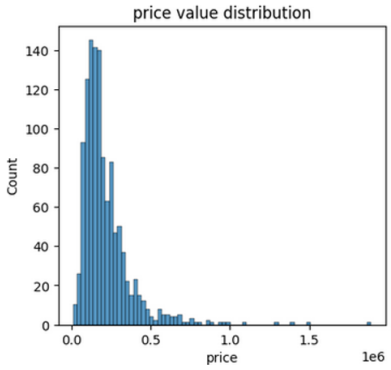


Figure 4.6: Houses' *price* distribution.

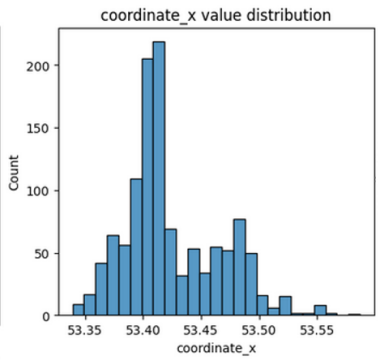
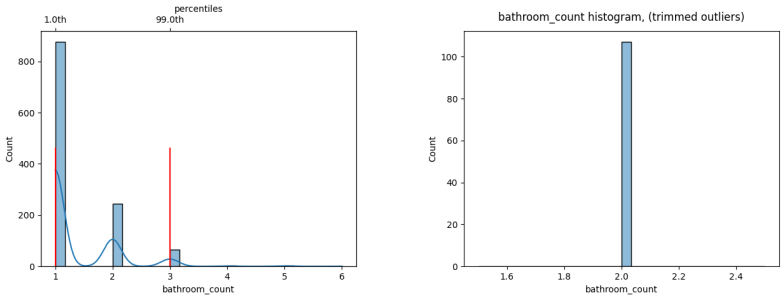


Figure 4.7: *Coordinate x* (longitude) distribution.

- **Trimming -** The process of either deleting or disregarding the outliers from future use, it also minimizes the introduction of bias, as the data that is kept remains unaltered. Can also be more recommendable when the nature of the outliers stems from anomalies (aberrant).



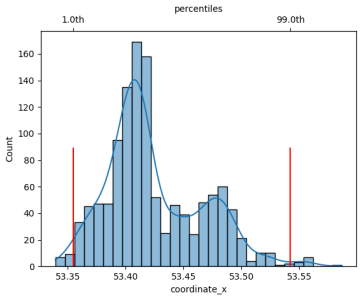
(a) *Bathroom count*'s distribution marked by the 1^{st} and 99^{st} percentiles

(b) 4.8a after outliers were trimmed

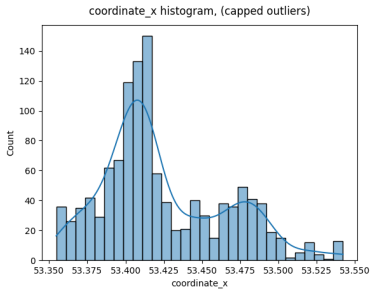
Figure 4.8: The before and after outlier trimming of *price* with the use of the percentile method. One can observe how the exclusion of populated extremes can in some cases result in a very biased an unrepresentative sample.

This Liverpool house listings case study provided an opportunity to showcase a situation where highly populated extremes, when trimmed, could lead to a substantial difference on the distribution, especially in a strongly discrete variable, such as *bathroom count* (4.8). On a feature such as *price*, however, if one considers the deflated cost of auctioned houses as an anomaly, their removal can be more advisable than methods such as capping.

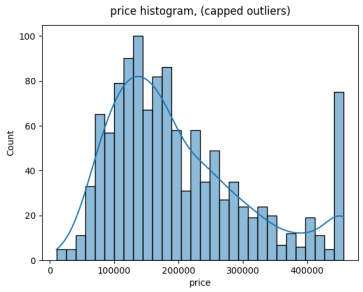
- **Capping -** When the discordant observations are instead replaced by the threshold value. It introduces some bias due to populating the bounds with more observations, albeit allowing these data points to be kept, a welcome trait when the data set is, proportionally to the outliers or as a whole, deemed small. It is also a more appropriate method when the outlier nature is "natural".



(a) *Coordinate x*'s distribution marked by the 1^{st} and 99^{st} percentiles



(b) 4.9a after outliers were capped



(c) *Price*'s distribution after outliers, detected with IQR (4.4), were capped

Figure 4.9: The before and after outlier trimming of *coordinate x* and *price*, selected with the use of the percentile and IQR methods respectively. A low amount of extremes maintains the overall probability distribution (4.9b), whereas a higher count can result in a maxima "lump" (4.9c), swaying a model's understanding of the data, especially around it.

Such was the case with *coordinate x*. With a scarce number of, likely natural, outliers (4.9a), this technique is capable of keeping the "shape" of the sample distribution, as is observable in 4.9b. Distinctively, in 4.4 we saw how there were several *price* instances above the upper limit "whisker". After capping them to the whisker's defined threshold, a significant increase in the sample size of the new maximum takes place (4.9c). The bias being introduced by the former outliers is, nonetheless, reduced, but the observations near the cap can now be, proportionally to the outlier amount, associated with the features of these discordants, which may induce the model in er-

ror.

It was fortunate that this case study enabled the demonstration of the advantages but also risks of using outlier-handling methods, which will vary according to the data data they are applied to.

Ordering rationale - The presence of outliers would require encoders to build their codification structure around classes that should not be included and would end up being removed. Naturally, scaling would also be impacted, in both natural and aberrant outlier cases. A case could also be made for Dimensionality Reduction, with reasons such as impurity scores being biased by the presence of the extremes, and projection axes, which generalize the data, fitting the non-outlier points worse.

4.3.1.6 Data Encoding

Description - Although understandable to humans, non-numeric data is incomprehensible for machine learning models, who learn through mathematical operations, such as derivatives and distance calculations. To benefit from the information this type of data provides, one must convert this input into a format that allows for the arithmetic procedures, necessary for learning, to occur: numerical. As a reminder, "numeric" data that does not stem from a unit of measurement, and works instead as an identifier, like telephone numbers, is commonly typed as *strings* or *objects*, and should also be targeted with this transformation.

Categorical data can at times have a logic. Ordinal classes, such as efficiency ratings "A", "B", "C", will benefit from an encoder that takes the ordering into account, whereas nominal classes such as "freehold/leasehold" and names do not require it. Other characteristics, including the feature's cardinality or its correlation with the target variable, will affect the appropriateness of the encoding methods.

Lastly, we will see a case where the definition of this step extends beyond the "word-number" conversion.

Use - While *booleans* were simply converted into their numeric counterparts (False $\rightarrow$ 0, True $\rightarrow$ 1), three options were made available for the standard use of the Data Encoding step, with an extra one to deal with a data type of cyclic nature: *date*.

Figure 4.10 will be used as a term of comparison between pre and post encoding, by the three used methods, of the categorical feature *name*.

	name
481	Strike
182	RWinvest London
482	Sutton Kersh - Auctions
573	Andrew Louis
894	Purplebricks, Head Office
...	...
1164	Sutton Kersh - Auctions
500	Clive Watkin - Crosby Sales
941	ERE Property - Leeds
421	RWinvest London
1829	Sutton Kersh - West Derby & Central Liverpool ...

Figure 4.10: Original *names* referring to the real estate agents who listed the properties (right), and the data point id (left).

The implemented methods were the following:

- **One-hot encoding -** The variable, with cardinality n, was split into n columns, where each column represents a possible category. Assuming each data point cannot belong to more than one category, it will then have the column that matched its original category assigned a "1", and all the other ones originated from the split assigned a "0". The encoder is fit on the training dataset, which likely contains at least one example of each category. If new training or test data contain categories not yet seen, a parameter was set to have any new label be represented by only "0's", placed across the columns of the already known categories. It may start becoming noticeable that a high cardinality variable will need an equally significant number of columns to represent its encoded version. Such increase in the datasets dimensionality can be undesirable for the execution time performance of the algorithm. In the present dataset, we have $n \leq 1390$, which could vary depending on whether there was deletion in Handling Missing Data and/or removal in Handling Outliers. Even with a deletion and trimming combination, the dimensionality would nonetheless increase manifold. Due to the relatively modest number of samples, however, this encoder did not result in a noticeable execution time bottleneck in the program, and was thus kept for comparison purposes. Figure 4.11 shows a snippet of the resulting bit-like representation of the variable *name*.

	encoded_name_Sutton Kersh - Allerton & South Liverpool Sales	encoded_name_Sutton Kersh - Auctions	encoded_name_Sutton Kersh - City Centre	\
481	0.0	0.0	0.0	
182	0.0	0.0	0.0	
482	0.0	1.0	0.0	
573	0.0	0.0	0.0	
894	0.0	0.0	0.0	
...	...	...	...	
1164	0.0	1.0	0.0	
500	0.0	0.0	0.0	
941	0.0	0.0	0.0	
421	0.0	0.0	0.0	
1829	0.0	0.0	0.0	

Figure 4.11: One-hot encoded real estate agent *names*

- **Base-N encoding -** A method which, in increments of 1, assigns a number to each of the classes' categories, and then represents that number in base-N notation. Each place of that base will become a column. Note that one can also obtain a Binary Encoder by simply defining the *base* parameter to 2.

  With the number of necessary columns being $\sqrt[N]{n}$, rounded up, this method greatly reduces the dimensionality overload when compared to other techniques, such as One-hot.

  The base used in this project was base16, and its representation of the variable *name* can be seen in 4.12.

	encoded_name_0	encoded_name_1	\
481	0	1	
182	0	2	
482	0	3	
573	0	4	
894	0	5	
...	...	...	
1164	0	3	
500	3	9	
941	1	15	
421	0	2	
1829	0	13	

Figure 4.12: Base16 encoded real estate agent *names*

- **Target Encoder (with smoothing) -** A mixture of the global mean of the target variable μ_g and a category's target variable mean μ_c is used to replace that same category. The blending between μ_g and μ_c can emphasize the former if the smoothing parameter is higher, and the latter if lower. This can benefit features that have a very low popularity on some of their categories, making it "safer" to represent them with a global mean instead of their own but likely "overfitted" one.

  Rather than choosing a specific *smooth* parameter value, an empiric Bayes estimate is used instead, allowing for a more automated choice,

less reliant on human guess. One further topic to discuss is how both μ's are obtained when the target is itself categorical. In that case, the used library applies a *LabelBinarizer* "one-vs-all" approach, which, for the problem at hand, is equivalent to replacing the predictor being encoded with a One-hot Encoding representation of the target, in N_t columns where N_t is the target's cardinality. Then, μ_c is taken from the partial mean of the target's category, corresponding to the predictor's category at hand, and μ_g is simply the average of the target's encoded labels. In Figure 4.13 it is possible to see how Target Encoding works when applied to the predictor *name* and when the target is the categorical variable *tenure*.

	encoded_name_Freehold	encoded_name_Leasehold	encoded_name_Share of freehold
481	0.741857	0.245978	0.011345
182	0.000000	1.000000	0.000000
482	0.689560	0.310414	0.000000
573	0.376563	0.623044	0.000000
894	0.747356	0.252625	0.000000
...	...	...	...
1164	0.689560	0.310414	0.000000
500	0.833653	0.166286	0.000000
941	0.000000	1.000000	0.000000
421	0.000000	1.000000	0.000000
1829	0.768306	0.231635	0.000000

Figure 4.13: Target encoded real estate agent *names*

Let us now take a look at a slightly different use-case for encoding.

- **Cyclic encoding -** At last, it is time to figure how to address a variable of type *date*. If one simply kept the days and months as they are, an algorithm would see "day 30" the same way it would see any other conventional "30", that being more logically distant to "1" than, for instance, "20". However, that is not the case in data of cyclic nature, "month 12" is closer to "month 2" than "month 6", and so, in order to keep the logic we know to be true, we must find a function that respects this property, and encode the data in a fitting manner. One way to accomplish this is through the use of the cyclic functions of sin and cos. Both *day* and *month* were thus represented with a pair of sin and cos columns (A.1), noting that both are required to avoid ambiguity.

	encoded_listing_date_day_sin	encoded_listing_date_day_cos
481	0.848644	0.528964
182	0.790776	-0.612106
482	0.848644	0.528964
573	0.724793	0.688967
894	-0.998717	-0.050649
...	...	...
1164	0.937752	0.347305
500	0.848644	0.528964
941	-0.968077	-0.250653
421	0.848644	0.528964
1829	-0.299363	-0.954139

Figure 4.14: Cyclic encoding of *day* using the sine and cosine functions.

Ordering rationale - Encoding after Feature Scaling would mean that some encoders, which can build up on the numbers depending on the cardinality of the feature, could cause non-numeric features to end up weighting more in a model's calculations than the numerical ones, which would already be scaled at that point. There are specific cases, however, where the order may not matter, more concretely when the encoder has an already "normalised" representation of the data which also matches the scaling method being used (say, One-hot Encoder with Min-max Normalisation). Nonetheless, as a rule of thumb, encoding before scaling appears to always be a safe choice, while reversing the order may usually not be. Similarly to an m.l. model, Dimensionality Reduction methods make use of calculations of, for instance, means and covariances, for which having the data in numerical form is a must.

4.3.1.7 Feature Scaling

Description - Feature scaling adjusts the numeric values of the data so that their order of magnitude becomes more modest, as well as similar between features. This becomes of relevance when algorithms such as Principal Component Analysis (PCA), given the use of gradient descent, benefit from more "circular" cost functions in order to make the gradient converge to the centre (the global minima) as efficiently, "direction-wise", as possible. Scaling the data means similar learning rates across the features, which in turn allows the cost function to be less elliptical or odd-shaped, and more circular as desired. Also impacted are algorithms which use distance measures, such as the Euclidean Distance, between values as a loss function (e.g. Linear Regression), or as a way to categorize observations in classification (e.g. K-Nearest Neighbours). Scaling will prevent higher magnitude features from having more impact in these final multi-dimensional distance calculations than the ones with values of a more modest nature.

Use - Using the same logic as with the predictors, scaling the target variable allows models which train using L1 or L2 penalties on loss func-

tions to do so without the disproportional features biasing their own weight penalties. If it happened, other parameters would likely be wrongly overshadowed, and more steps would be required to reach a loss close to zero (which is to say, to learn properly). Another benefit is that the MSE itself becomes more interpretable and comparable with evaluations from other features, as its values will in result also be scaled. With this in mind, a decision was made to scale both the predictor and target variables.

Now, a look into the methods used in this project to scale them:

- **Min-max Normalisation -** One of the most common scaling methods, it scales the data so that it is confined in the interval $[0, 1]$, with 0 and 1 corresponding to the minimum and maximum values respectively (4.15). With $x^{i'}$ as the scaled value of a certain feature i, we have:

$$x^{i'} = \frac{x^i - x^i_{min}}{x^i_{max} - x^i_{min}}. \tag{4.2}$$

  With the scaling bounds relying on the extreme values, its vulnerability to outliers becomes apparent. That weakness became less noticeable since it had already been dealt with during a previously discussed step of preprocessing, but it is nonetheless of interest to study the success of Min-max in that mitigated circumstance. This scaler also maintains the distribution shape, which can aid interpretability.

	bedroom_count
481	0.666667
182	0.000000
482	1.000000
573	0.666667
894	0.333333
...	...
1164	0.333333
500	0.666667
941	0.333333
421	0.000000
1829	0.666667

Figure 4.15: *Bedroom count* after Min-max Normalisation scaling.

- **Mean Normalisation -** Here, the mean is subtracted from each value so that the scaling becomes centred around it. It is then divided by the value range, in the form

$$x^{i'} = \frac{x^i - \mu^i}{x^i_{max} - x^i_{min}}. \tag{4.3}$$

  This way, the data becomes more centred and slightly more resistant to outlier influence. Despite being closely related to Min-max, Mean

Normalisation's (MN) performance will still be judged, being paired with both normal and skewed feature distributions. Its result after being applied to a closely Gaussian *bedroom count* can be found in 4.16.

	bedroom_count
481	0.223881
182	-0.442786
482	0.557214
573	0.223881
894	-0.109453
...	...
1164	-0.109453
500	0.223881
941	-0.109453
421	-0.442786
1829	0.223881

Figure 4.16: *Bedroom count* after Mean Normalisation scaling.

- **Standardization -** With a similar logic to MN, the division of the values is now done by the standard deviation, as in

$$x^{i'} = \frac{x^i - \mu^i}{\sigma^i}, \tag{4.4}$$

instead of the range (4.17d). It not only has mean centring, but also unit variance as a result. It is more applicable to the same scenarios as the ones mentioned in MN, and extends to those who benefit from a similar variance across features, such as Stochastic Gradient Descent (SGD), a method usable by the implemented Dimensionality Reduction algorithm: PCA.

- **Robust Scaler -** The mean centring provides a limited improvement on dealing with outliers, yet more resilient scaling choices exist. Robust Scaler, defined as:

$$x^{i'} = \frac{x^i - \tilde{\mu}^i}{IQR^i}, \tag{4.5}$$

where $\tilde{\mu}^i$ is the feature's median, has the particularity of neither the choice of a median centring nor the scaling using IQR being impacted by extreme values. Even if the Handling Outliers step had already been addressed, and concealed some of its usefulness, it is still of interest in the project to see how it performs comparatively to other methods. A demonstration on how it scales a feature can be found in (4.17e). The IQR scaling aids with skewed distributions, as was the case with most of the variables in the used dataset (e.g. *price, living*

room count), and so, seeing how it interacts with an outlier handler such as using expert elicitation or Z-scores offered the chance to prove that the skew-related "weaknesses" associated with them could, to an extent, be covered using Robust Scaler.

- **Power Transformer -** Akin to Standardization, it produces a more Gaussian-like probability distribution for each feature, while also encouraging their homoscedasticity, another way of referring to a more similar variance amongst each other. It differs, however, in the way this is achieved. The setting of a hyperparameter, λ, allows the tuning of the transformation function used on the feature vector, so that it becomes closer to a normal distribution. Although both static (in libraries other than Scikit-learn) and automatic ways to set λ exist, the focus will go towards the one used in this project: Yeo-Johnson. It is described as

$$x^{i'} = \begin{cases} \frac{(x^i+1)^{\lambda^i}-1}{\lambda^i} \text{ if } \lambda^i \neq 0,\ x^i \geq 0 \\ \log(x^i+1) \text{ if } \lambda^i = 0,\ x^i \geq 0 \\ \frac{-(-x^i+1)^{2-\lambda^i}-1}{2-\lambda^i} \text{ if } \lambda^i \neq 2,\ x^i < 0 \\ -\log(x^i+1) \text{ if } \lambda^i = 2,\ x^i < 0\ . \end{cases} \tag{4.6}$$

  In an automated fashion, Yeo-Johnson tries multiple transformations, with both logarithmic and exponential functions. It then resorts to maximum likelihood in order to choose the hyperparameter that, when applied to a transformation function, best resembles the distribution of the data observed.

  In figure 4.17f we find a snippet of the result of this transformation on one of the features.

	bedroom_count
481	3.0
182	1.0
482	4.0
573	3.0
894	2.0
...	...
1164	2.0
500	3.0
941	2.0
421	1.0
1829	3.0

(a) Snippet of the original *bedroom count* values.

	bedroom_count
481	0.666667
182	0.000000
482	1.000000
573	0.666667
894	0.333333
...	...
1164	0.333333
500	0.666667
941	0.333333
421	0.000000
1829	0.666667

(b) *Bedroom count* after Min-max Normalisation scaling.

	bedroom_count
481	0.223881
182	-0.442786
482	0.557214
573	0.223881
894	-0.109453
...	...
1164	-0.109453
500	0.223881
941	-0.109453
421	-0.442786
1829	0.223881

(c) *Bedroom count* after Mean Normalisation scaling.

	bedroom_count
481	0.709586
182	-1.403403
482	1.766080
573	0.709586
894	-0.346909
...	...
1164	-0.346909
500	0.709586
941	-0.346909
421	-1.403403
1829	0.709586

(d) *Bedroom count* after Standardization scaling.

	bedroom_count
481	1.0
182	-1.0
482	2.0
573	1.0
894	0.0
...	...
1164	0.0
500	1.0
941	0.0
421	-1.0
1829	1.0

(e) *Bedroom count* after Robust scaling.

	bedroom_count
481	2.724468
182	0.956908
482	3.566242
573	2.724468
894	1.857927
...	...
1164	1.857927
500	2.724468
941	1.857927
421	0.956908
1829	2.724468

(f) *Bedroom count* after Power Transformer scaling.

Figure 4.17: Effect of the implemented scaling methods on the variable *bedroom count*.

Ordering rationale - The reasoning behind the placement of this step before Dimensionality Reduction has already been mentioned in the Description. Implemented Dimensionality Reduction algorithms such as PCA and LDA make use of the Euclidean distances between data points. Thus, to avoid biases stemming from the different magnitudes across features, feature scaling was the prioritized step of the two.

4.3.1.8 Dimensionality Reduction

Description - Dimensionality Reduction (DR) is a preprocessing step capable of improving multiple angles of a prediction model, tackling problems originating from the presence of an excessive or redundant number of features. One of these problems is multicollinearity. This happens when one

of the independent variables is highly correlated with one, or more, other independent variables. The model can then struggle to find which data changes have which impact, resulting in larger standard errors and less reliable models. DR also helps with overfitting, keeping only what it deems "essential", but without steering the model in a way that creates a deceiving bias. Decreasing feature redundancy will help with multicollinearity and overfitting, while removing low impact features will also (mainly) aid with the latter. In this resulting reduced space, operations with high complexity, namely matrix inversions ($O(n^c)$), become exponentially faster. Finally, the feature values will also be less present in the extremes of their dimension, which would cause them to be more distant to the other data points, hindering the capacity to generalize the data. Less computational tasks and a more compact subspace are two factors enabling the desirable speed and accuracy performance gains, respectively. Other applications for DR exist, but will be skipped for the sake staying within the scope of the work conducted by the author.

It is worth briefly distinguishing the terms feature projection and feature selection. While both work to reduce the dimensionality of the data, feature selection is the process of choosing the most impactful independent variables, with regard to the target, and discarding the remaining. Interpretability is not negatively impacted, as the chosen features remain intact. On the other hand, feature projection finds a multidimensional subspace, defined by a selection of axes which better explain and segregate the data, obtained from information gathered from the original space. Combining a chosen number of axes (the n most descriptive ones) with the original feature values will result in the data points projected in this new subspace. Note that these new dimensions describing the data, were obtained through mathematical operations which take into account how all the data points relate, and are thus a blending of the original features. Here lies the main difference between selection, where we keep a subsample of the original features, and projection, where we obtain new although harder to interpret ones.

Use - Both feature selection and projection methods were implemented, with the latter having both linear and non-linear options.

4.3.1.8.1 Feature Selection

- **Random Forests -** RF consists in multiple decision trees, each of them made from a random sample of the dataset, and within that sample a subset of randomly selected features. The decision nodes will recursively split the sample into more decision nodes. To decide how any split happens, an evaluator such as maximum entropy gain,

minimum MSE, and others is used, creating this way a decision node. The splitting happens until one of many possible user-chosen hyper-parameters is met, for instance: the maximum tree depth is reached, the number of data points resulting from the split is unable to be split again without falling under the minimum accepted amount, the node reaching a certain degree of purity (classification) or squared error (regression), and others.

With the used library *scikit-learn*, it is possible to calculate the feature importances during the training process. In other words, a value is assigned to how important each feature was predicting the target based on how much is decreases the impurity, squared error, absolute error, or any other *criterion* used to judge the quality of a split. Once the feature importances are obtained, one can decide to keep the l best and discard the remaining.

In this project, the author decided to choose l such that $l = |\phi_l \geq \mu_B|$, meaning only features whose importance ϕ was equal to or above their overall importance mean would be kept. Figure 4.18 contains a plot of the top 15 feature importances, while using 100 trees and other default sklearn's hyperparameters. The importances were obtained via the *Giny impurity* criterion.

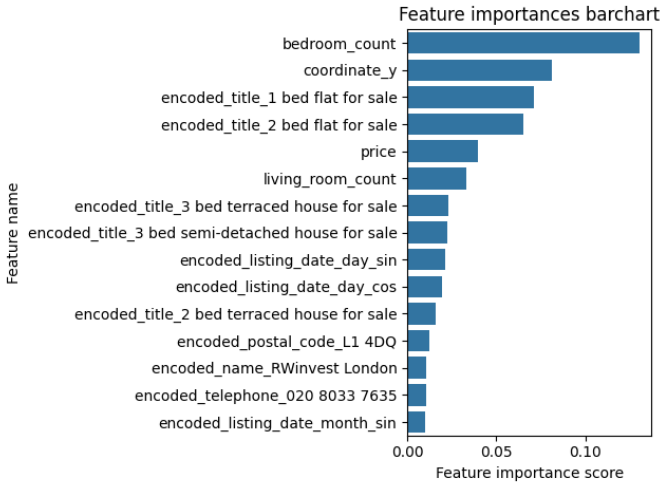


Figure 4.18: Feature importances for the target variable *tenure* using the Random Forests Classifier algorithm. The result was obtained using 100 trees and the importance is derived from the normalized *Gini impurity* criterion. Only the top 15 importances are present for the sake of clarity.

4.3.1.8.2 Feature Projection

- **Principal Components Analysis -** Starting the projection methods, PCA computes each of the d features' mean, in order to then calculate the predictors' covariance matrix Σ. With this $d*d$ matrix it is now possible to obtain the eigenvalues and their corresponding eigenvectors. The eigenvalues can be interpreted as a measure of how well the axes of the eigenvector generalize the data. The original feature vectors can now be projected using the obtained d eigenvectors, resulting in d principal components. However this representation would maintain the number of dimensions. Thus, after sorting the eigenvalues, we can instead select the l largest (where l is the desired number of dimensions) so that only the corresponding eigenvectors are kept to project the original data into the new feature subspace. To conclude, the new feature vectors are then obtained via the dot product between the l eigenvectors (transposed) and the original predictors' matrix, resulting in the simpler l-dimensional projection of the data.

  To clear out any possible confusion, the author notes that "principal

components" can be an ambiguous term in the literature, referring to either the eigenvectors, or the projected feature vector. In this report, its definition will be that of the projected feature vector, with eigenvectors representing the directions.

In this project, l was chosen based on the *explained variance ratio* of each principal component, in the form

$$l = \begin{cases} m \text{ if } C(m) \geq 0.9 \wedge m \leq 100 \\ \\ n \text{ if } C(n) \geq 0.8 \wedge n > 100, \end{cases} \tag{4.7}$$

where C is the cumulative sum of the explained variance ratios. This means the algorithm attempts to reach 90% of the explained variance up until 100 components, or it stops at 80% if beyond that amount. This ensures a compromise between a high generalization and keeping the number of dimensions to a not too computationally expensive degree.

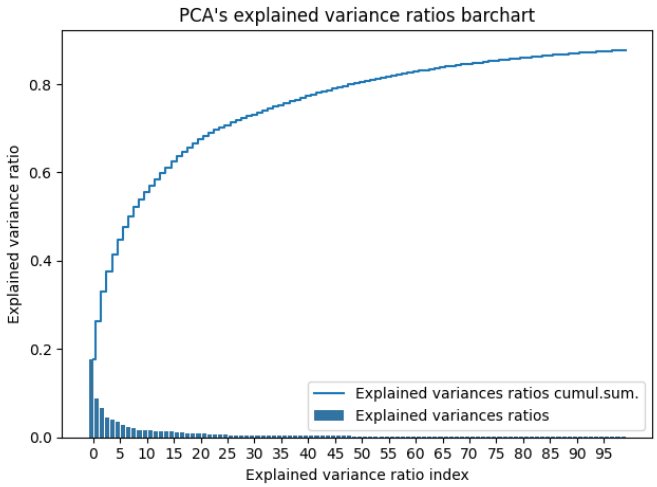


Figure 4.19: Snippet of one hundred explained variance ratios using PCA, and respective cumulative sum for the target variable *price*. This example reached $C \approx 0.88$ at index 100, reducing the dimensionality by approximately 94%.

One example can be seen in figure 4.19, with the dimensionality being reduced from $d = 1254$ to $l = 100$. Note the plot and l will vary depending on the preprocessing settings used, but the provided snippets are illustrative of the general behaviour in the different settings.

- **Linear Discriminant Analysis -** A supervised learning classifier which, unlike PCA, uses the dependent variable labels to to aid its understanding of the data. It does share some similarities however, as it is also able to find linear combinations with the original features, a factor which then enables it to be used for for dimensionality reduction alike. To that end, for all dependent variable labels $c \in C$ and all predictors $f \in F$, it starts by computing $\mu_{f,c}$, that is, the means of each feature f, for each group of samples whose target label is c. Then, the within-class and between-class scatter matrices are computed. These matrices represent, respectively, the variation within each label, whose minimization guarantees a good compactness between the data points, and the variation between labels which is to be maximized to ensure the data is as generalized as possible.

  After computing both matrices, the eigenvalues and eigenvectors are now obtainable and a similar logic to PCA ensues. Once sorted, the l eigenvectors with the largest eigenvalues are selected to project the data into the new l-dimensional subspace, whose new feature vectors can be referred to as the chosen linear discriminants.

  In this project, it was found that $l = 2$ was usually enough to reach an explained variance ratio of over 0.9 (4.20). Notwithstanding, the desired number of discriminants was set to the standard $min(d, c-1)$ since the number of classes of any categorical target was always low enough to guarantee a dimensionality that does not significantly affect the computation time, and thus enable slightly more variance to be explained than with only two discriminants.

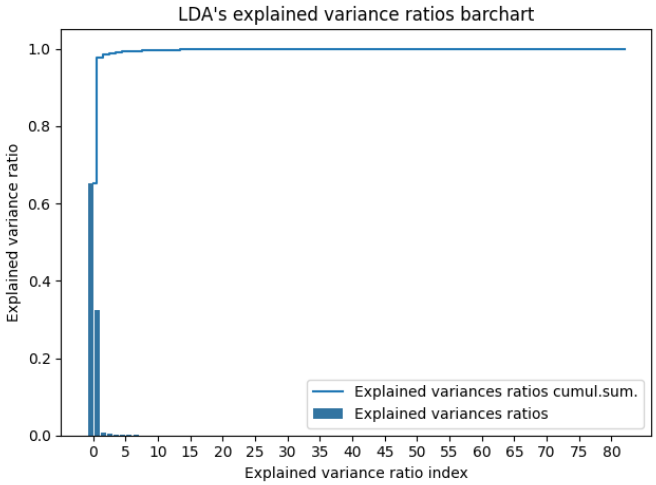


Figure 4.20: LDA's explained variance ratios and respective cumulative sum for the target variable *location*, under certain preprocessing settings. Here, $l = 2$ would already be enough to explain $> 97\%$ of the variance.

- **Truncated Singular Value Decomposition -** This matrix decomposition method starts by partitioning the original dataset matrix $X_{n \times f}$, into three matrices, in the form

$$X = U * \Sigma * V^T, \tag{4.8}$$

where U is the $n \times n$ left-singular vectors matrix, Σ is a $n \times f$ diagonal matrix containing the singular values of X, sorted column-wise in descending order, and V^T is an $f \times f$ (transposed) right-singular vectors matrix.

The "truncated" reference in the name derives from the fact only the first l columns of Σ, which represent those with the highest singular values, are selected, rendering Σ a diagonal $l \times l$ matrix. Subsequently U_t becomes $n \times l$, V_t^T becomes $l \times f$, and the new truncated SVD matrix can be described as:

$$X_t = U * \Sigma_t * V_t^T. \tag{4.9}$$

While differing from PCA, since it performs the decomposition in the original data matrix rather than in the covariance one, it is akin to it

in achieving a lower dimensionality subspace through the linear combination of the features. This projection of the data can be concluded by doing

$$X_p = X * V_t \tag{4.10}$$

with X_p being the now reduced data matrix in the l-dimensional subspace.

Once more, a cumulative sum explained variance threshold was chosen in order to decide an "ideal" number of projected feature vectors. This time the chosen threshold was 99% of the explained variance, which proved achievable within a small number of dimensions (4.21).

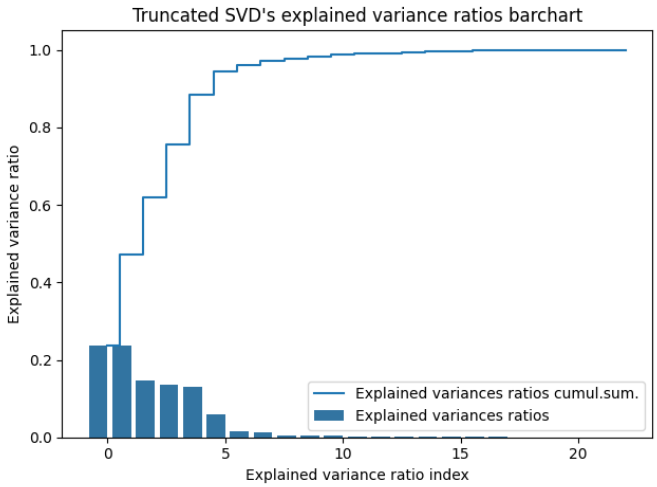


Figure 4.21: Truncated SVD's explained variance ratios and respective cumulative sum for the target variable *price* under a group of preprocessing settings. An $l = 12$ would, in this scenario, be enough to explain $> 99\%$ of the variance, reducing the dimensionality by 48%.

- **Kernel PCA -** Offers an "upgrade" to PCA in when dealing with non-linearly separable data. Since it projects the data into $l < d$ dimensions, PCA will typically fall short in finding a good axes in the subspace whose hyperplane splits the data well, if it wasn't already

happening in the original space. Kernel PCA solves this by instead projecting the data into an h-dimensional space, where $h > d > l$, using a kernel function, such that new multiple principal axis are, in this space, now capable of linearly separating the data. After the data has been projected into h dimensions, PCA can be performed, this time calculating its eigenvalues and eigenvectors from the kernel matrix (obtained from the kernel function), instead of the covariance matrix.

On the choice of l, the author suspects the absence of *sklearn*'s explained variance attributes may be connected to this explained variance being relative not to the original feature vectors but to the components of the up-scaled space these features are initially projected to, by the kernel method. It was hence decided the eigenvalues would be a reasonable alternative indicator, and 4.22 shows an example of their analysis.

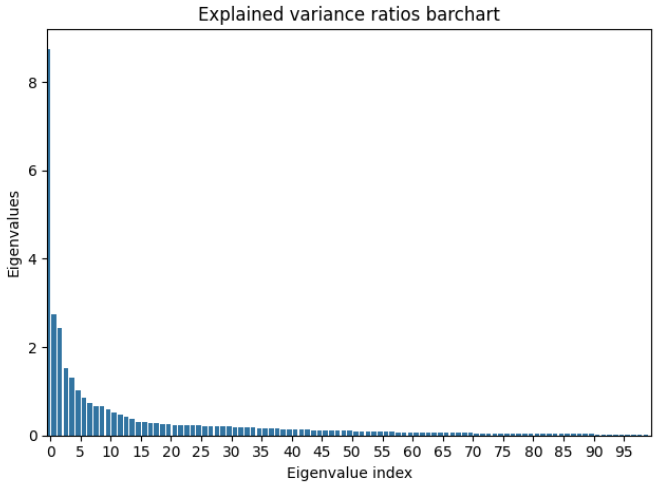


Figure 4.22: The one hundred highest eigenvalues using the Kernel PCA algorithm on the data. The settings used include *price* as the target variable, and the polynomial kernel function. It is possible to notice the accentuated "elbow" mark registered around the 6^{th} index.

We can notice the "elbow" of the curve sits around index 5, with the most significant drop being right after the first index. This out-

come was somewhat consistent with different preprocessing settings. Nonetheless, the author chose a value of $l = 15$ as the added dimensionality would come at virtually no computational cost, while enabling enhanced predictive capabilities from the extra generalization.

Ordering rationale - The explanation behind this being the last preprocessing step has, in part, already been given in the previous steps. In short, being done posterior to the train/test split prevents data leakage in the DR operations; with missing data being already imputed or removed DR algorithms can obtain covariance matrices and other required intermediary calculations which wouldn't be feasible otherwise; outliers should be handled beforehand to avoid swaying μ's and other values used in algorithms such as LDA and PCA; the mathematical steps of most DR methods aside from decision trees require numerical data, requiring encoding for non-numerical types and benefiting from encoding specific ones such as *date*; last but not least, doing it after the scaling keeps the order of magnitude of some features, such as *price* from overshadowing others in how much they influence the feature projections or selections (e.g. much larger or smaller covariance values). As a reminder, the ordering may at times be flexible depending on the methods used during these preprocessing steps, as some of them do not interfere with the others (e.g. Random Forests feature selection is indifferent to scaling methods). Nonetheless, this rationale can be deemed generally favourable to a "synergetic" preprocessing of the data.

4.3.2 Model Algorithms used

With the goal of predicting house listing characteristics which are well-defined (such as pricing, tenure, and so forth), the problem can be considered a Supervised Learning one. Justifiably, two classification and two regression algorithms were included. When it comes to Unsupervised Learning, it can be of use if "tendencies" or the detection of listing anomalies become topics of interest. For now, that kind of analysis falls outside of aim of this work.

4.3.2.0.1 Regression

- **LASSO Regression -** Short for Least Absolute Shrinkage and Selection Operator, LASSO is, at its core, a linear regression model where the standard ordinary least squares loss function has the added penalty: L1 regularization. It is obtained through the sum of the absolute values of all feature weights, multiplied by a regularization

coefficient in the form:

$$L1 = \alpha \sum_{i=0}^{d} |w_i| \ . \tag{4.11}$$

where w_i are the weights corresponding to each feature, and α the regularization coefficient (this letter was chosen to remain homogenous with *sklearn*'s terms, yet, in literature, it is usually referred to as λ).

The goal is to minimize the overall loss function, comprised of both least squares and L1. which can be written as

$$min_w \frac{|y - wX|_2^2}{2n} + L1 \tag{4.12}$$

with y as the true value and wX as the predicted value for a given data point X. The minimization of the w coefficient implies that some of the features will have their impact essentially nullified. Since the choice is tied to the fitting capability of the model, the features whose coefficient will become 0 are the ones who were least correlated with the independent variable, and can thus be ignored with minute or even null consequence. We can see now that Lasso Regression has got embedded its own feature selection. Notwithstanding, in this project, it will still work alongside the dimensionality reduction done during preprocessing.

When it comes to the choice of α, a value that's too high will result in a harsher regularization, likely to bring even some of the more relevant features to a coefficient of 0, which could harm the model. Low values appear to work better in general, but too low could result in all the features, even the more irrelevant "noisy" ones being kept. After multiple runs being evaluated, the selected value was $\alpha = 0.002$ along with a cyclic update of the coefficients during the minimization process, instead of a random one.

- **Random Forest Regressor -** Continuing on with the summary of this algorithm done in the Dimensionality Reduction subsection, now focused on prediction, each tree will receive the portion of the new data point that matches the features it is responsible for splitting. Contrary to DR preprocessing, the feature importances are not the primary focus during regression. Rather, after all trees decide on which leaf the new data point falls on, the average of all the values of these leaves is taken, a process usually referred to as "aggregation", obtaining the final prediction.

The implemented model's hyperparameters include: 100 trees, the number of features per tree is $\sqrt{d}$, a number the author believes alongside the logarithm usually works well in literature, and (mean) squared error as the *criterion* deciding how the splitting happens.

4.3.2.0.2 Classification

- **Stochastic Gradient Descent Classifier -** The SGD classifier consists in a (at the moment, on *sklearn*, only linear) classifier whose loss function is optimized via SGD. The linear classifier tries to find a linear function which best separates the target's classes. The optimization part occurs by finding the steepest descent, perpendicular to the loss function (previously mentioned in the description of Feature Scaling).

  Gradient Descent makes use of all the training data in order to calculate the gradient which will be used to update the hyperparameters. Stochastic GD, on the other hand, picks a single (and random, hence "stochastic") point and uses that point alone to get a faster but approximated gradient, to then update the hyperparameters, in that iteration. That means it trains and converges much faster than GD. However, taking less data into consideration will cause in the resulting fit to generally not be as good as that obtainable from standard (batch) GD.

  Despite a currently modest amount of samples, further web scrapping could increase the data indefinitely. As section Results will show, SGDC produced capable models in terms of predictive performance, making this algorithm a reasonable choice while accounting for scalability.

  As for the used model parameters, the author went with one of the possible Support Vector Machine implementations (plus the SGD optimizer), translatable into a *hinge* loss function, alongside a *L2* penalty with $\alpha = 1 \times 10^{-4}$ and a maximum of 1000 epochs.

- **Multilayer Perceptron Classifier -** Expanding on a regular perceptron, a MLC consists not only of an input and output layers, but also hidden ones, each with its own set of neurons.

  As the input layer connects its inputs with the neurons of the first hidden layer, each input-neuron connection will have a weight associated with it, which will serve as the coefficient of that input. Every neuron will have its own a bias, b, which will be added to the weighted input, in the form $w_i x_i + b$. One limitation of the baseline Perceptron

is that this function is linear, and thus it will be difficult to finding a good fit to separate non-linear data. This is where MPC's activation functions come into play. Each neuron in the hidden layers will have a complex activation function, more likely to have the necessary terms to, with the right weights, accurately describe the behaviour of the data and distinguish the target classes.

In the input layer, MPC begins by assigning a random weight to each input and then combining all weighted inputs into a sum, as $\sum_{i=1}^{n} w_i x_i + b$. This sum is then fed into the activation function of each neuron in the first hidden layer who, after their own computations, combine the results once more and feed them into the next hidden layer, a process repeated recursively. Note that although the activation function is typically the same for all neurons, their outputs will be different, since each neuron on a given layer will have, independently from the other neurons, different weights for each input received from the previous layer. Once the output layer is reached, the sum goes through one last neuron's activation function, producing the final output. This output comes, nonetheless, from operations that used randomly initialized weights in this initial complete propagation, and so a brief explanation on how the algorithm actually learns will ensue. Only when the output layer is reached for the first time, can the algorithm begin to understand how it should tune itself. With the first instance of the predicted output computed, a loss function is used to calculate the error between the true and obtained values. SGD is the used method (adding to the loss function a regularization term), and, after the gradients of the loss function are calculated with respect to each weight and bias (partial derivatives of the loss function), they are propagated to the previous layer. The magnitude of these gradients will be proportional to that of the learning rate. There, the weight and bias gradients are subtracted to the weights connected with the current layer, and bias of that neuron respectively, updating them. This process repeats itself recursively until the input layer is reached. In a back and forth fashion, the forward (getting a prediction) and backward (updating the weights and bias) propagations happen until the training loss change falls below a certain threshold for a given number of epochs, or any other stop condition (such as the maximum number of calls to the loss function being reached) has been met.

In the current implementation, the MPC has *Rectified Linear Unit* (ReLU) as its activation function, *adam* optimizer with $\alpha = 1 \times 10^{-4}$, and a constant learning rate of 2.8×10^{-4}. Given a dimensionality

of over two dozen features, extending up to more than a thousand depending on the preprocessing steps (e.g. the use of One-hot encoding), linear separations being the better choice is not guaranteed, and so, using a non-linear method could prove beneficial. MPC provides this functionality while also establishing a pathway to develop into a more typical deep neural network for more intensive training, if deemed necessary.

4.4 Optimisation

There are multiple advantages to optimising code, extending as far as yielding energy cost savings. Notwithstanding, in this project, its purpose was to lessen any rampant growth of model training time in case more data is scraped or otherwise added in the future (scalability). Additionally, for analysis such as finding which preprocessing group gives out the best predictive performance, any speed gains can be re-used every time one of the groups is tested. In the present case, the product of all the implemented options leads to 51,840 possible ways to do the preprocessing of the data. If taking into account each prediction method as well as over 20 possible target variables, the number of tests done could be over one million, growing exponentially as more options are added.

Focusing on the 51,840 tests, the overall goal of the author was to utilize the hardware threads and cores available on the machine, making use of parallelization capabilities in the code whenever possible. Each test comprised full preprocessing execution as well as model training to obtain the results for the used prediction metrics, therefore being the bulk of the program's run-time. Hence, this was the section to prioritize efficiency improvements on. Other logical cares which allowed loops to end earlier and already computed values to be re-used rather than computed again were also taken. Nonetheless, they would entail a more case-by-case reasoning to be explained. Since they did not appear as a potential bottleneck, the focus of this chapter will reside more on the parallelization aspects and batching procedures.

On a first stage, a more general approach was taken, using libraries that enabled a better hardware thread and core workload distribution, in order to parallelize loops and some library methods. Secondly, the author aimed to identify possible bottlenecks in the execution time of each test, with a relevant one being identified in one of the preprocessing steps, to be specified in the Scikit-learn's Joblib subsection.

Let us now have an overview on the different libraries used to improve the scalability and raw speed-up of the program:

4.4.1 Vectorization

In the context of programming, and more concretely Pandas, vectorization consists in batching operations rather than performing them one element at a time. Nonetheless, one possible school of thought is that, at a low logical level, operations will still be done to each element, individually. Notwithstanding, vectorizing in Pandas is more than that. Pandas runs natively over NumPy, which in turn uses the C programming language, a more optimized, low-level language, cutting down execution times by removing some of the abstraction overhead inherent to Python. Additionally, data structures such as NumPy arrays and Pandas' Series are made to hold only one data type at a time (integer, float, string or *datetime*). Python's structures on the other hand, can have multiple types at once, meaning it needs to treat the structure as one of *objects*, even if they only contain one type. Pandas/NumPy pre-emptively know this type (rather than having to determine it) and optimize the processing of that structure based on it. Regarding parallelization, Pandas and NumPy utilize the available cores (and hardware threads) of the machine's CPU to run the loop-based operations done on the mentioned libraries' structures concurrently, as opposed to serially, spanning them in multiple processes (with one or more software threads). This enables making use of the elements in parallel. Lastly, as opposed to the mentioned libraries, Python stores references to values where, regardless of the pointers being together in memory, the values themselves can be stored separately, resulting in a slightly less efficient access.

In the current work, vectorization was employed through the choice of directly using NumPy/Pandas' structures when performing an operation that involves them, instead of resorting to Python-native ones. Additionally, ready available methods specific to these libraries were preferred to homologue alternatives.

Estimating the impact this care brought can be problematic, as it would require recoding Series, DataFrame and NumPy array operations using Python-native methods, so that the two approaches can be compared and their performance judged. As this is impractical, an exact speed up number may not be easily attainable, but we can have an idea of the improvements by doing so on one or more examples. In 4.23 we notice that the task of eliminating the features that have too many missing values to be deemed useful computes approximately 7.2 times faster using the batched Pandas alternative, when compared to the element-wise, more Python-centred one, a meaningful gain. This result was obtained from a sum of 100 executions for a more solid estimate, using the *timeit* library (A.4). In practical terms, this elimination is done twice per run (training and test data), and so, over the course of the 51,840 preprocessing groupings tests done, it adds up to:

$$
\begin{aligned}
t_{seconds} &= \frac{0.781064 - 1.109218}{100} * 2 * 51840 \\
&\approx 697s \\
t_{minutes} &= \frac{t_{seconds}}{60} \\
&\approx 11.6m.
\end{aligned}
\tag{4.13}
$$

This allows us to conclude that a single vectorization implementation can, in this project, save over eleven and a half minutes per prediction model and target variable tested. Regardless of the time portion it takes in the global run of the program, this can be seen as a perceivable and welcome time-saving, which elucidates the rapidness impact all (beneficial) vectorization steps can have on the execution, when added together.

```python
def nonvectorized_method():
    cols_to_drop = []
    for column_name in X_train.columns:
        num_of_elements = X_train[column_name].shape[0]
        nan_counter = 0
        for element in X_train[column_name]:
            if pd.isnull(element):
                nan_counter += 1
        nan_mean = nan_counter / num_of_elements
        if nan_mean > missing_data_threshold:
            cols_to_drop.append(column_name)
    X_train.drop(columns=cols_to_drop, inplace=True)
```

(a) Non-vectorized operation on a DataFrame.

```python
def vectorized_method():
    X_train = X_train[X_train.columns[(X_train.isna().mean()<missing_data_threshold)]]
```

(b) Vectorized operation on the DataFrame.

```
# Timing vectorized and non-vectorized code #

vectorized method time (100x): 0.109218 seconds
non-vectorized method time (100x): 0.781064 seconds
```

(c) Measuring vectorized and non-vectorized computational times. The result is a 100 tests sum, in seconds.

Figure 4.23: Comparison of the time taken by an element-wise (non-vectorized) and batched (vectorized) approaches to perform a filtering operation on a DataFrame. The result, in seconds, is the sum of 100 runs (for each case) using the *timeit* library.

It is finally worth noting that, although usually beneficial, vectorization can at times be disadvantageous, when for instance dealing with strings. This can happen when libraries do not possess a more efficient way to deal with this type of data and end up resorting to Python functions in their underlying code, resulting in a normal Python method, but with and added overhead.

4.4.2 Scikit-learn's Joblib

For high-level parallelism, Joblib is a library that offers multiprocessing capabilities to the code at hand. It does so through the *n_jobs* parameter, which defines the number of processes or threads to run in parallel. The conducted work uses the default "loky" backend (a multi-processing backend [4]) meaning from here onwards, *n_jobs* will, in this context, be referred to as the number of processes.

Some Scikit-learn algorithms have an identical *n_jobs* parameter, whose purpose is to set up Joblib's own *n_jobs* on the algorithm's internal code. Therefore, both preprocessing and prediction segments contained algorithms suitable for parallelization.

In the current implementation, the chosen value was $n\_jobs = -1$ for both preprocessing Scikit-learn algorithms and prediction models alike. This value means Joblib will attempt to distribute the looping workloads by all available CPU's, maximising the parallelization's potential for the algorithm it is implemented on. As an example, a performance bottleneck was found during preprocessing, when using Random Forest Regression to perform feature selection.

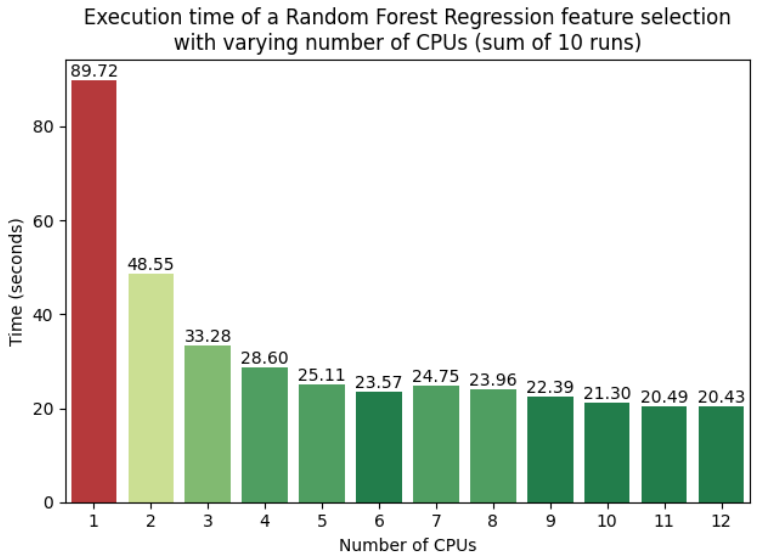


Figure 4.24: Execution time of feature selection using Random Forest Regression, from one CPU to the maximum contained in the machine: twelve. Each time represents the sum of 10 runs (for more robust estimates).

As depicted in 4.24, by employing all 12 available CPUs the runtime of this task was cut by 77%, another way of interpreting a $\approx 339\%$ speed increase. The execution time, in this case, is inversely proportional to the number of CPUs in its majority, but not directly due to the operations which are not parallelisable, as well as possible load imbalances when the partitions are not equally divisable by the number of CPUs. On a more general view, addressing this bottleneck on both the training and test set feature selections led to a 280% performance boost in the overall run-time of the program, when performing this type of feature selection (A.5).

4.4.3 Parallel-pandas

Still on the side of high-level parallelism, Parallel-pandas is a library which deserialises a number of Pandas' methods. It splits the DataFrame and Series structures into portions, each of them sent to a different thread or process. Then, it performs the Pandas' method on each of them, obtaining the partial results. Finally, it combines them accordingly, obtaining the final value.

This library is implemented through the use of "p_methods", which use the same spelling as the pure Pandas counterparts, with a "p_" prefix (e.g. *df.mean()*'s homologue is *df.p_mean()*).

In the inclusion of this library in the project, the "mean" and "sum" methods were the two viable candidates to be replaced with their "p_" equivalents. Other Pandas' methods were either not used or applied to one column at a time, making them not compatible with Parallel-pandas as it is only prepared to replace DataFrame methods, but not Series. A test which could be done in future work is to convert these Series into single-column DataFrames so that the library can then be applied. The data copying overhead would likely outweigh the benefits for the current size of the dataset, but worth considering if more data is scraped. For the implemented methods, the outcomes were dissimilar. For the *sum* and *p_sum* comparison, an $\approx 7\%$ performance boost was gained (4.1). The *p_mean*, however, ended up being significantly slower than its *mean* counterpart. Possible explanations could be that the relatively modest number of rows which, coupled with the simplicity of the mean operation made the communication costs between the CPUs overshadow any possible gains from the workload distribution. Load imbalances could be a contributor as well, however, neither of these would justify the disparity between the results obtained from the mean and sum functions. Looking at the source code of the library [5], it is unclear to the author what the exact cause could be since Parallel-pandas apparently "simply" calls the original Pandas' method and splits it amongst "workers" using Pythons' multiprocessing package. The current hypothesis is that op-

erations/initializations done inside the library to set up parallelization have an inherent difference depending on the method they are working it, which on relatively simple cases such as a sum or a mean, could end up overshadowing their complexity, resulting in the mentioned overheads becoming the true bottleneck.

Execution times in seconds (1000 runs)	Pandas' method	Parallel-pandas method *(p_method)*	Speed-up / slow down from parallelization
mean	0.307330	118.778	$\approx$ 386x slower
sum	0.008946	0.008330	$\approx$ 1.07x faster

Table 4.1: Comparison between Pandas and Parallel-pandas speed on the applicable methods. The times are sums of 1000 runs to reduce the error margin of the speed-up/slow down results.

This was done alongside Joblib, with no apparent grounds for a poor synergy as the mentioned library was applied to Scikit-learn objects through the *n_jobs* parameter. Therefore, the parallelization happening in the methods within those objects does not intercept with the methods explicitly called by the user and now replaced with parallel-pandas. Vectorization is also synergetic as applying "p_methods" to DataFrames is inherently vectorizing the execution of that method.

4.4.4 Numba

As a Python compiler capable of converting numerical functions into machine code, Numba offers a more low-level, fine-grained optimization ahead of what one can obtain with the abstraction of pure Python.

The inclusion of this compiler in the program took place in the form of the *@jit* decorator. There was, nonetheless, a lack of methods where it could be applicable since, despite working well with NumPy, Numba is not ready to address functions whose parameter types that it does not support such as dictionaries, DataFrames, and others. In the case where it was usable, the difference was negligible, with the decorated method performing 2% worse than the original one, on an estimate of 100,000 runs.

A reason for the present negligible performance losses is that Numba functions have their own initialization costs, which means that for simple computations and a limited amount of data, these may take long enough to cancel or even reverse the benefits of the low-level optimization. On

a related note, when faced with methods Numba does not recognize and support, it will end up running them using Python, but with the added interaction overhead. This second explanation was not the case for the method decorated with *@jit* in this program, but is still worth mentioning due to how naturally it can occur.

4.5 Results

Over the course of the web scrapping done across the multiple pages of the listings website, a total of 26 features (28 considering the date partitioning) were collected, for each of the 1890 houses or apartments listed during the process.

Each time a machine learning algorithm was selected for prediction, it was tested for 51,840 different preprocessing groupings. This would be the same as saying every preprocessing step had all its options tested with alongside all those of the remaining steps, so that the optimal way to prepare the data could be found for said machine learning algorithm. Notwithstanding, the word "optimal" can be looked at from different angles, commonly referred to as "metrics". This project evaluated the Root Mean Squared Error (RMSE), the Mean Absolute Error (MAE) and the Mean Absolute Percentage Error (MAPE) as the metrics for regression, and Accuracy, Precision and Recall for classification.

To help legitimize the fairness of the analysis, both preprocessing methods and machine learning algorithms that involve a degree of randomness were initialized with a seed (mostly the number 42). PCA was an exception, since its randomness came from a randomised solver [6], automatically used when the data meets certain conditions, and which can not be seeded by the user. More on it in the Classification metrics' analysis segment. Nonetheless, this practice helps, or even guarantees, reproducibility between the runs and fairness when comparing different method groupings.

4.5.1 Regression metrics

Table 4.2 contains the best results obtained, by regression metric and m.l. algorithm used, for the dependent variables *price* and *number of bathrooms*. More variables could have been presented, yet, additional weeks would be required to try all the set ups necessary for the inclusion of their analysis. The author considers this extra cost, combined with a more convoluted table representation of the results (large number of columns), would fall out of the scope of the project, possibly even hindering it. Hence, these variables are deemed sufficient to conduct a solid analysis on the results.

Metric	RMSE		MAPE (%)	
M.l. algorithm	Lasso Regression	Random Forest Regression	Lasso Regression	Random Forest Regression
Price	56,066*	64,206	24.94	23.19
min	5000	5000	-	-
max	725,000	725,000	-	-
Number of bathrooms	0.4068	0.3846	9.33	8.22
min	1	1	-	-
max	4	4	-	-

Table 4.2: Best results obtained for the used regressors by regression metric. *min* and *max* represent the minimum and maximum of the target variable above them, for the preprocessing options that lead to the presented best metric result. *Originally, the best obtained value was 54,960. For reproducibility reasons, however, the presented value was the one chosen, as the best (reproducible) one.

4.5.1.1 RMSE

The RMSE is one of the multiple ways to assess the goodness-of-fit of a model. It calculates the sum of the squared residuals (sum of the distances between the predicted and actual values, squared) and applies a square root on the resulting value. Its formula is given by:

$$\text{RMSE} = \sqrt{\frac{1}{n^{y^{true}}} \sum_{i=0}^{ny-1} (y_i^{pred} - y_i^{true})^2} \tag{4.14}$$

with $n^{y^{true}}$ being the number of *y test* samples and (y_i^{pred}, y_i^{true}) being the i-th predicted and true dependent variable values respectively.

The obtained results fell under 7.8% and 9% of the price value range for the Lasso and Random Forest models respectively, supporting the viewpoint of a good performance achieved by the model (4.2). Comparatively, Lasso Regression performed at least 14.5% better than Random Forest for the *price* target variable, and 5.5% worse on the target *number of bathrooms*.

As for the usability of RMSE, a larger emphasis is put on the residuals than, for instance, MAE, as they are squared and the average is applied before the square root. However this penalization is softer than with MSE. This entails that RMSE offers a compromise between MAE and MSE, being moderately vulnerable to unscaled and outlier values. Yet, in this work,

both scaling and outliers were addressed during preprocessing. To what extent this residual penalization is beneficial may remain a subjective matter and studying a way to decide which metric to rely on the most based on that factor (and accounting for the outlier and scaling preprocessing done) is perhaps a worthy topic to explore in future research.

4.5.1.2 MAPE

By calculating the percentage of the MAE in relation to the true value, one obtains the MAPE, also written as:

$$\begin{aligned} \text{MAPE} &= \frac{\text{MAE}}{\max(\epsilon, y^{true})} \\ &\Leftrightarrow \frac{1}{n^{y^{true}}} \sum_{i=0}^{n^{y^{true}}-1} \frac{|y_i^{pred} - y_i^{true}|}{\max(\epsilon, y_i^{true})} \end{aligned} \tag{4.15}$$

where ϵ is a symbolic error value to ensure the MAPE can be calculated when $y_i^{true} = 0$.

The differences in the best result obtained for each algorithm can be seen directly in 4.2, with Random Forest R. having a 7% edge on *price* and 12% on *number of bathrooms*, over Lasso R. The lower MAPEs on price could be due to factors such as the existence of auction-based values and pre-removal of features with a more extensive text body (*features*, *description*). The importance of the latter is that a future implementation with text-analysis could help extracting keywords such as "garage", "garden" or "pool" to aid the model in its understanding of the prices. Nonetheless, the performance for the bathroom count was satisfactory, one likely justification being its overall higher independence on amenities described in the removed text features, than a variable such as *price*.

On top of the better outlier robustness derived from its MAE-based mathematical foundation (lack of any extra residual penalty), MAPE also offers a relative view on the mean absolute error. This allows it to be unaffected by the scale of the target variable, and consequently interpreted directly.

4.5.1.3 MAE

Evaluating the mean absolute value of the residuals we have MAE. It is given by:

$$\text{MAE} = \frac{1}{n^{y^{true}}} \sum_{i=0}^{n^{y^{true}}-1} |y_i^{pred} - y_i^{true}|, \tag{4.16}$$

in the same manner as MAPE, but without the denominator associated with the proportion of the residual to the true value.

In the author's opinion, the outcomes give the models they originate from an optimistic outlook on their predictive performance. While *price* obtained MAEs of only $\approx 5.4\%$ (RFR) and $\approx 5.7\%$ (Lasso) of the range of true values for y, *number of bathrooms* did even better, with 4.9% for RFR and $\approx 5.3\%$ for Lasso R (B.1). In terms of this metric, Random Forest R. modestly outperformed Lasso in both dependent variables. It did better by 5% on *price* and $\approx 7.7\%$ on its dependent variable counterpart *number of bathrooms*.

After MAPE, MAE still remains one of the more interpretable metrics, without any further changes to the residuals other than making them positive and then finding their average. Its interpretation and/or comparison is still susceptible to the scaling, which was not a problem as all scalers used in this project could be and were reversed before obtaining MAE. It is also a solid choice in the presence of outliers since the residual penalty is not intensified. It is the author's view it could be a more trustworthy metric for a variable such as *price* since the outlier detection method that led to the best results still allowed for noticeable outliers, such as auctioned houses, to be included (hence the 5000 minimum). This may not be the case for variables such as *number of bathrooms* as the outliers have already been significantly cut.

4.5.1.4 Analysis

Both algorithms managed to obtain reasonable models, and performed relatively close to one another. Random Forest R. had the overall edge in all the metrics, however, performing between 5 and 12% better than Lasso, except for the best *price* RMSE, where Lasso fared better by 14.5%. This could be justified by Lasso's non-cross validated α making it lower the coefficients of somewhat important features quicker than desirable, meaning the model will converge under a more feature-biased context. In like manner to table B.2, the preprocessing settings corresponding to the best metric results (which is to say, that led to them) were collected. The aim was to identify trends for the most successful settings, overall or by metric.

Starting with the numerical imputation method, for the independent variables (X), mean imputation provided the best results in the majority of the cases. For the target variable, y, results were dissimilar between the target variables. On *number of bathrooms*, the optimal numerical imputation method was a draw between Meam Imp. and Forward Fill. An hypothesis that would support the emergence of the latter is that, since this variable had a skewed distribution (4.8a), it is comprehensible that

mean imputation may not always be the best choice, given the variable's deviation from the median and therefore greater likelihood for the imputed value to be further from the true one. It could be expectable to see the median imputation taking the spotlight instead, but given the tight range of values the number of bathrooms in a home can have, there are now two conditions benefiting the propagation of a value by Forward Fill: it is more likely to propagate the median, which makes sense in a skewed distribution, as well as being more likely be fully accurate, given the reduced amount of value options. When it comes to the numerical imputation method for $y = price$, the results show any method would be capable of providing the best output, as this step was, in this case, not impactful enough to change the outcome.

For categorical imputation of X, deletion was the overall method of choice for all metrics and dependent variables. Features such as *title* or *tenure* did not have an overwhelming mode (see A.6), diminishing the appropriateness of this type of imputation. As for y, the chosen method did not have a meaningful relevance, and thus any was acceptable.

For the optimal outlier detection method, the most consistent choice was IQR. The second most popular choice was "any", happening on *price*'s Lasso and *number of bathroom*'s RF. The choice of an outlier detection method more appropriate to skewed distributions went on par with what could be anticipated, as both analysed target variables belonged to that distribution type.

When it came to how to deal with the outliers, trimming was succinctly the go-to choice on all metrics and for both target variables. The modestly sized dataset implies a modest test set, possibly not registering some of the more extreme outliers found in the training set. In that case, their presence in the training set would only wrongly bias the model, encouraging their removal as opposed to keeping some with the more lenient expert approach, or capping them.

One-hot encoding was the undisputed winner for both target variables, and on all metrics. Most features had a cardinality between 3 and 117, which would increase the dimensionality of the dataset manifold, but still not too drastically (from 15 features to $[322, 426]$, depending on whether there was imputation or deletion). Yet the variable *location* had between 728 and 1054 different values (once again depending on the imputation factor), with its encoding resulting in a dimensionality of $[1061, 1492]$, which could lead to the assumption that One-hot Encoder may introduce multicollinearity, unideal for Lasso Regression, and sparsity, not desirable for Random Forest R. Notwithstanding, other encoding contenders were not free of unwanted traits. For starters, Base-N has a slight inherent ordering system, which was not ideal for nominal categorical values. Target

Encoding, with the chosen amount of smoothing, could struggle to accurately represent the less popular categories ("Share of freehold" in A.6a), overfitting on them. A larger presence if these low popular features on the test set would result in a less prepared model given its bias when learning about them. Conversely, if they are mostly absent from the test set, the results could still be good, even if the model struggles to understand them nonetheless. In the end, one can conclude One-hot's suitability for nominal features and resilience to variables with a rare representation overshadowed the impact of sparsity and multicollinearity for the current dataset.

Scaling's top choice was less undivided, but Standardization was responsible for half of the best results obtained, loosely followed by no scaling (responsible for a quarter), Yeo-Johnson Power Transformer ensuing (responsible for 16.7%) and finally Robust Scaler (at the remaining 8.3%). Despite a preliminary outlier handling during preprocessing, Standardization provides some robustness to this factor when compared to both Normalisation variants, stemming from the use of a standard deviation term, which can prove valuable to counteract, to an extent, any outliers that pass through "undetected" (not considered as such). However, other methods such as Robust Scaler and even Power Transformer also grant this benefit, even amplifying it. With further analysis, it was shown that these other mentioned scalers, including Min-max, achieved metric results within a value as small as 1% or 2% that of Standardization. Yet it was clear that Mean-normalisation was the least performing choice, offering results $> 11\%$ away from the optimal choice. It is also worth noting that Random Forest should be fully independent from the usage of a scaler, yet, the author considered somewhat bemusing that, for RFR, Mean Scaling fared more than 10% away from the optimal value, on at least one metric. The author believes this difference eliminates, with little room for doubt, the possibility that some precision errors may have slightly nuanced the predictive results, comparatively to the runs with other scaling options, and so, the reasoning behind Mean-normalisation's noticeably poorer performance will be a topic requiring a more thorough investigation.

Lastly, on dimensionality reduction, $^{8}/_{12}$ best results were attributed to Random Forest feature selection, and the remaining $^{4}/_{12}$ to Truncated SVD feature projection. LDA makes use of the dependent variable's labels to achieve its desired within-class compactness and between-class variability, and is therefore not meant to be used in a regression context, justifying its absence. Another algorithm not present was PCA, which can be understandable upon closer inspection. Firstly, it was brought to a supervised rather than unsupervised learning context, where it has to compete against algorithms who make use of the (labelled) target variable for their learning. Additionally, PCA projection gives precedence to linear combinations with

a larger variance. Under the perception of the author, the lower the variance of a variable, the less likely it is to be close to the axis of the eigenvector corresponding to the first principal component. This means that the axes closer to these lower variance variables tend to get more represented by eigenvectors with a low eigenvalue, more likely to not make it to the top selected ones for the projection and dimensionality reduction. To sum up, variables of a lower variance are more keen to be dismissed. This could be problematic if those variables, despite a low variance, are actually well correlated with the target variable (impactful). The plot in A.7 shows the feature importances given by the method that provided the most optimal results, RFR. Conversely, after checking the variances of each feature (under the same conditions, see A.8), one can notice that the sixth most impactful variable (*encoded_location_24_Linacre Lane, Bootle, Merseyside L20*) has a variance > 667 times smaller than the seventh one (*encoded_listing_date_day_sin*), and nearly 1200 times smaller than than the ninth most important one (*living_room_count*). Finishing the line of thought, the used data possessed cases that demonstrate where PCA can make mistakes when choosing the axes that better project the data, hindering its performance and keeping it astray from delivering the best predictive results in any used regression metric. Notwithstanding, PCA was still a powerful predictor as, despite its flaws, one can see in figures A.7 and A.8 that the variances did still, in the end, have a not perfect but reasonable correlation with the feature importances, a possible reason why PCA was on multiple metrics not the ideal choice but a close second. Kernel PCA stayed out of the prime choices as well. Despite its added capacity to deal with non-linearly separable data, it would appear that, if such was the case for the current data, it would still not compensate for the drawbacks inherent to PCA just mentioned. Furthermore, if the data was already mostly linearly separable, then with KPCA there would be smaller chance of improvement coming from the initial higher-dimensional projection and, instead, there would be a higher risk of overfitting. Truncated SVD lies on the same principles as PCA, which, as mentioned, was a close second behind RFR and TSVD on their respective wins. One can assume TSVD's success is related to the reasonable correlation between the predictors' variance and their importance. Moreover, to the author's understanding, TSVD works better on sparse data as it does not need to compute a computationally expensive covariance matrix, or deal with any problems that arise from its non-positive definiteness. Given the success of One-hot Encoding in this research, most of the dimensionality in the present data ends up being on account of encoded variables, which using the aforementioned method would result in a sparse dataset, supporting TSVD's success. To conclude, adding to the easier interpretability, Random Forest's success may have gained its margin due to the fact that

feature selection promotes the linear relationships between the variables, less prone to overfitting. This contrasts with the more complex non-linear relationships that can be captured with projection methods which, albeit at times beneficial, can, in some circumstances, such as with the current dataset, induce a degree of overfitting, potentially high enough to give the best results lead to RF.

4.5.2 Classification metrics

On the categorical side of target variables, 4.3 presents the best results by classification metric and m.l. algorithm used.

It is worth mentioning that two of the metrics used (Precision and Recall) are, originally, an array of results in a multi-class classification setting, where each value corresponds to a class. Therefore, the way chosen to objectively compare the tested runs was to use a technique referred to as "macro-averaging", which translates to averaging out the $1 \times d - 1$ arrays of Precision or Recall values. The author defends that, although a robust technique, macro-averaging could be improved since it gives the same weight to each class, even the ones whose appearance is rare and more prone to volatile results, increasing the margin of error of the final value. Further exploring this topic will fall outside of the scope of this project, but one possible solution could be adding a weight to each class based on a combination of frequency and sample size, more punishing for very low sample amounts and/or frequencies, and less so for higher ones that already allow for a reasonable degree of confidence in the understanding of a class.

Execution times in seconds (1000 runs)	Accuracy		Precision		Recall	
M.l. algorithm	SGDC	MPC	SGDC	MPC	SGDC	MPC
Tenure	88.89	89.22	0.6298	0.5685	0.8647	0.8154

Table 4.3: Best results obtained for the used classifiers by classification metric. Precision and recall have been macro-averaged accounting for the extremely rare occurrence of one of the three possible classes.

4.5.2.1 Accuracy

Arguably the most intuitive and well known performance metric, accuracy represents the ratio between the successfully done predictions and the com-

plete set of predictions. It can be generalised as:

$$\text{Accuracy} = \frac{\sum TP_c + \sum TN_c}{\sum TP_c + \sum TN_c + \sum FP_c + \sum FN_c},\ c \in \mathbb{N} \cap [0, d-1] \quad (4.17)$$

with TP as the True Positives, TN as the true Negatives, FP as the False Positives and FN as the False Negatives, and each class represented as c. In the context of multi-class classification, one class at a time (c) is deemed the "positive" class, and the remaining are clustered as the "negative" ones. Then, one can sum all TPs (from the viewpoint of each c being the "positive" class), and do the same for the remaining terms of the expression. Nonetheless, it is possible to simplify the formula for this metric with $Accuracy = \frac{\text{successful predictions}}{\text{all predictions}}$.

Both Stochastic Gradient Descent Classifier and Multilayer Perceptron Classifier performed well, correctly predicting the tenure of a home nearly 9/10 times, and with a difference of only 0.4% in their best results, better on the side of MPC.

Being a solid metric, accuracy is not without any flaws. If there was a majorly dominating class and the model predicted every test instance as being of that class, the accuracy result would be high even if the model was incapable of predicting anything else. Despite being an extreme example, in this project, the tenure class "Share of Freehold" was only registered in 0.13% of the instances and absent in the test set, meaning this high accuracy attests for the model's capability to predict the relatively evenly distributed "Freehold" and "Leasehold" classes, but tells us nothing about its capability to predict "Share of Freehold". Nevertheless, this metric offers a solid degree of reliability and some of the most interpretable results of any metric.

4.5.2.2 Precision

This metric tells the ratio between the successful predictions of a certain class, and all the times that class was predicted (whether rightfully or not). It is given by:

$$\text{Precision}_c = \frac{TP_c}{TP_c + FP_c} \quad (4.18)$$

with the macro-averaged precision being given by the formula $Precision = \frac{\sum \text{Precision}_c}{d}$.

The train/test split method utilized a seed so that the runs were not randomised, allowing a fair comparison. With the used seed, a very rare *tenure* class (*Share of freehold*) was present in the training set but not in the test set. Nevertheless, preprocessing steps that delete data (such as deleting missing values or trimming outliers) happened to remove any instances

of this class in the training set as well, causing the model to only know the remaining classes, and Precision to not need to calculate any value related to *Share of freehold*. Therefore, it was the author's decision to analyse Precision only in the runs where all classes were considered during training (which happened, for example, when missing data was imputed rather than deleted). This was done because the author believes it would not be of much use to analyse how the model behaves in a "universe" where it only accounts for some of the classes, given the existence of models which account for all of them. With this clarification, one knows that, despite the very low occurrence of the rare variable, it was part of the considered models' training, and thus False Positives were possible, even though True Positives were not. In other words, the Precision value for *Share of Freehold* was always 0 in the considered runs, hence its more modest macro-averaged result, visible in 4.3. Nevertheless, Precision presented the most disparate results between the two m.l. models of all classification metrics. The SGDC's Precision was 10.8% higher than MPC's. The difference between this disparity and the one in Recall suggests the discrepancy lies in the number of False Positives, more common in MPC. The overall value was modest (a display of the weaknesses of macro-averaging) but it mainly displays the (expectable) inefficacy of the algorithm in understanding the rare class mentioned. If the Precision value was macro-averaged without accounting for the zeroed value of that class, it would show ≈ 0.9447 for SGDC and ≈ 0.8527 for MPC, the same degree of disparity, but considerably better results.

With this metric, it is possible to analyse the performance of the algorithm for each class individually, but one should be careful judging its capabilities as a whole. In this case, it exposed the model's vulnerability to understand when a certain home or apartment is a "Share of freehold". Although in the particular case of *tenure* macro-averaging was not crucial, its use can, in general, facilitate the overview of the algorithm's performance regarding False Positives, especially when the cardinality of a feature is high enough to make a case-by-case analysis impractical. Notwithstanding, Figure A.9 shows how the highest obtained macro-averaged precision (mentioned in the previous paragraph and visible in 4.3) is not, by itself, an indicator of the global model performance. We can notice how the amount of False Positives for *Share of freehold* does not affect its precision value (always 0 since there are no True Positives), no matter how high. It can, however, certainly hide a low predictive capability (for an idea, this run, with the highest precision, had an accuracy of $\approx 33.7\%$).

4.5.2.3 Recall

The ratio between a class' successful predictions and all the times that class appears as a true value, Recall can be written in the form:

$$\text{Recall}_c = \frac{TP_c}{TP_c + FN_c} \tag{4.19}$$

and its macro-average as $Recall = \frac{\sum \text{Recall}_c}{d}$.

When there are no instances of a class on the test set, having True Positives and False Negatives in relation to it is impossible. In this research, *Share of freehold* was such a case, leading to its Recall being undefined. Just like for Precision, the author decided to only compare the results in which the rare variable was part of the training set. This time, however, it was considered fairer to count only the variables with a defined Recall for the denominator of the macro-average. The result seen in 4.3 is thus more generous than Precision's, being 6% higher on SGDC than on MPC.

Recall presents the same benefits and drawbacks discussed in the last paragraph of the Precision segment, with the difference that it measures the model's ability to accurately predict all instances of a class present in the test set (TP and FN).

4.5.2.4 Analysis

The SGDC was, overall, the model that achieved the best performance, with a marginal loss in Accuracy, but an edge of $\approx 11\%$ in Precision and 6% in Recall. The current implementation used the gradient of the hinge-loss function (with an L2 penalty), which could be seen as a "special" Support Vector Machine. This translates to two main differences when compared to the other gradient-based algorithm MPC: the assumption of linear separability, and the proclivity to overfitting. If the data is more linearly separable, SGDC could be better suited to generalize the model than MPC, which could itself attempt to find a non-linear function and end up overfitting. Given the more modest size of the dataset, the used MPC implementation with 1 hidden layer of 100 neurons could still entail a number of parameters high enough to also cause overfitting, which could justify the slightly better performance of SGDC.

The most prevalent numerical imputation method in the preprocessing settings that led to the best results in each metric was, for the independent variables, the median imputation (B.2). This is more on par with the expected than the results obtained in the in the Regression metrics subsection (4.5.1), given the skewed distribution of the variables. The choice could, however, have been influenced from PCA's solver randomization, as will be

explained shortly. If so, one could consider the most common method to be the runner-up: deletion. For the dependent variable (*tenure*), numerical imputation was not applicable, and so, as expected, the choice of the method did not impact the results. In table B.2, Accuracy does present a choice of median imputation, but the author notes that the sci-kit package implementation of PCA, which was the dimensionality reduction method associated with these results, contains an "auto-solver" [6], which adapts to the dimensionality of the data (it changed between runs depending on the encoder used). This means that, with an encoder such as One-hot, the PCA's solver is set to "randomized", which in turn may cause a particular run to have slightly different results than another, even if with the same settings. In this case, it happened when the target's numerical imputation method was set as the mean, making it appear to be "chosen", even though only the categorical one was being applied,

The choice of categorical imputation method did not seem particularly relevant for the independent variables. For the target y, deletion was the undivided choice as opposed to imputing with the mode, meaning the latter was likely introducing a high enough amount of noise.

If one disregards PCA's possible bias, the outlier detection results were split between Z-score and "any". One possible reason for the apparently anomalous choice of a method more suitable for normal distributions when the majority, in the current dataset, was skewed, was that more important features for the model training had distribution more similar to a normal one. A more appropriate outlier detection in these variables could thus outweighed the factor of a larger presence of skewed distributions in the predictors. The "any" appearance is related to the expert-made trim done in the next stage, outlier removal, which is based on human knowledge and disregards the algorithmic decision-making of what is considered an outlier.

All outlier removal methods had a similar presence in the best results. If we once again ignore the results stemming from PCA's randomised solver, then the method of choice is split between capping, for SGDC runs, and the expert trim, for MPC. As SGDC is more robust to overfitting, having more data can sensibly help the hinge loss function find hyperplanes (with one-versus-one or one-versus-all approaches) that better split the data. Thus, capping the data could help SGDC with a better generalisation, especially singe hinge-loss aims to maximise the margin between classes, focusing more on the points closer to the hyperplane, or misclassified, as opposed to the more extreme (capped) ones. This, in turn, results in the trade-off between bias and generalisation favouring the latter (in essence, more information and diminished drawbacks). MPC's tendency to learn complex relationships, between the predictors and target, discourages more bias, that could come from capping, having a natural preference for a more fine-tuned out-

lier treatment made by an expert instead. Even though trimming was more common in the regression metrics' analysis, the independent variables are slightly different now (*tenure* has shifted to being a dependent variable and both *price* and *number of bathrooms* are predictors) which could have been enough to shift which outlier removal methods are present in the best results, and how often.

Regarding encoding, and once more reading past the results derived from the randomised PCA solver in "Accuracy", the best Precision and Recall macro-averaged values had once again a frequency split, now between the Target Encoder for SGDC and Base-N encoder for MPC. Despite the possible struggle for Target Encoder to deal with the (lack of generalization obtainable from) rarer features, accounting for the implemented smoothing, if these are mostly absent from the test set, then SGDC robustness to overfitting may in fact welcome the extra correlation the encoded variable will have with the one being predicted, stemming from Target Encoder, without facing the drawbacks from overfitting on the low-sample classes. It is important to remember that such an absence could provide good results for this test set, but less so in one when they become more prevalent. Nevertheless, this method increases the risk of overfitting for MPC, which instead had its best results using a Base-N encoder. This encoder has an advantage in comparison with One-hot when used together with MPC: less sparsity. In the author's view, this is advantageous as it could make it harder for MPC to disregard important features which could not be deemed that way due to their "decomposed" parts from the encoding being, individually, less so.

Standardization was the overall most seen method when it comes to scaling, essentially explainable with the same reasoning described in the Regression metrics analysis.

Concluding with dimensionality reduction, Truncated SVD and PCA were the two main methods in the winning metric results. Although Base-N generates a less sparse dataset than One-hot, it still increased the dimensionality by 53.3% (from 15 to 23), meaning some sparsity, and thus higher chances for multicollinearity. This is undesirable for methods that rely on the computation of covariance or scatter matrices, as was the case with the projection-based ones implemented in this project, aside from TSVD, which may have gained an edge from this factor. Nonetheless, PCA still managed to perform well, as despite the negative aspects mentioned in the Regression metrics's analysis, the variance of the variables did have a noticeable correlation with their respective importance, which benefits PCA.

4.6 Data visualization

Data tends to be stored in an alphanumeric form. Although simple and memory-inexpensive, more graphical formats can often display this information in a more user-friendly fashion. Rows and columns of alphanumericals can be replaced with graphs or other visual elements, as a methodology to summarize and give insight on the data. With interactive dashboards standing for queries, any user, regardless of how tech-savvy, can understand and pass over the conclusions derived from the data. Microsoft Power BI is a tool which facilitates this visual side of business intelligence, while also providing a formula expression language, Data Analysis Expressions (DAX), for more tailored insights.

Multiple dashboards were created, one to analyse the original data and four to display the performance results of both regression and classification algorithms deployed, with regard to multiple metrics. In the performance dashboard, the dependent variables these algorithms are applied to are *price* and *tenure*. Others could be considered but are unessential in the scope of illustrating the visualization capabilities present in the project.

4.6.1 Original data dashboard

Despite the inability to embed an interactive dashboard on a static report, Figure 4.25 shows all charts, slicers and maps implemented. All elements are connected to the dataset and will adapt instantly and simultaneously to any selection made by the user. Regardless of the accustomization with technical queries or manipulation of data, these visual representations enable any user to present insights with a simple finger press or controller such as a computer mouse. Some examples include:

- Checking price averages by month/quarter/year, number of bedrooms, tenure, and others, or even a combination of multiple filters. Aside from a global overview, this also allows for a comparison of the cost of a specific house and others under similar specifications;
- Suggest real estate agents based on postal code, market share, and type of houses (high-end/budget) they are currently selling;
- A combination of the options described above for a more tailored analysis (e.g. finding all freehold houses, with 2 or 3 bedrooms, containing a "pool" in their description). The dashboard comes with an adaptive map visualization, which facilitates an individual analysis for each matching home.

Figure 4.25: Original listed homes data. From left to right, top to bottom: Average price per month; average price by number of bedrooms (and the sample size); average price of shared ownership, auctions, global within the current search context and purely global homes; average price per year excluding auctions and shared ownership; adaptive map filled with all the listed homes pinpointed by their coordinates, along with slicers for keyword searching and other parameters; top 5 high-end real estate agents regarding listed house price averages; top 5 budget real estate agents by the same criteria; real estate agent market share by number of listings; average price per postal code; gauge chart with the average price of the current search context versus the average price of that postal code.

Appendix A.2 contains a simple post-filtering snippet of the dashboard. It simulates a user interaction to analyse the average house price of a home matching a certain criteria, showing their respective and interactive locations. Selecting any of the homes in the map will update the remaining visuals to focus on information regarding that choice only, so that a single case analysis is also possible, if desired.

4.6.2 Performance dashboard

A dashboard was built for each classification and regression metric analysed. It provides a global overview on how different preprocessing and model choices performed (4.26).

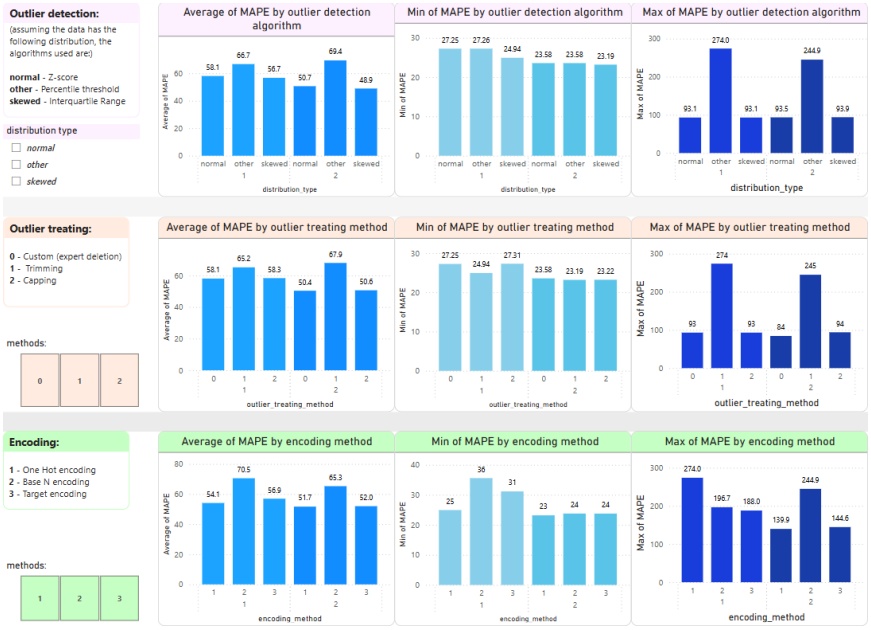


Figure 4.26: Snippet of the regression analysis metric *Mean Absolute Percentage Error* (MAPE) for the target variable *price*. The labelling for each preprocessing option can be consulted on the left, along with a slicer for a quick singular selection. Not captured on the snippet, the lower "X" axis labels 1 and 2 correspond to the *Lasso Regr.* and *Random Forest Regr.* models respectively.

If the user selects one specific algorithm, it can change the context to how that algorithm alone performed when grouped with the remaining steps (A.3). Lastly, selecting one preprocessing method in each step will enable the analysis of a specific build, with regard to how it performs for each metric and prediction model. This enables simulations of previously ran tests to be inspected instantly.

Chapter 5

Conclusion

With the inclusion of collection, storage, preprocessing and visualization of data, alongside the training and evaluation of predictive models, this work hopes to have demonstrated an at least basic understanding on how to build a data science pipeline, as well as working with the software necessary to accomplish it. It brings forward the idea of freedom to gather any (allowed) publicly available online data of interest, and working with it to build a system capable of presenting information succinctly, and making predictions based on it. The addition of multiple methods in each stage of the data preprocessing, and machine learning algorithms, allowed for an added versatility and comparison in the search for the optimal set-up, this time more focused on how the different algorithms interact with one-another, rather than on their parameter tuning.

Information contained in the 1890 web scrapped and database stored real estate listings, and corresponding 26 features, was made readily available in an interactive dashboard, intuitive for users of undiscriminating backgrounds. This aspect of dashboard interpretability was also extended to the model results deriving from the different preprocessing groupings, so that the impacts of each method can be isolated, or specific combinations tested on the spot. The predictive capability of the models fluctuated heavily, depending on the algorithm that trained them, and how the data was prepared. Nonetheless, RMSE, on the numerical targets tested, went as low as 7.8% and 12.8% of the *price* and *number of bathrooms* value ranges respectively, showing a degree of robustness. On the categorical side, *tenure*, the only variable of this type tested, was predicted with an accuracy of over 89%. The obtained results demonstrate not only the influence preprocessing could have in the performance of an algorithm, given their fluctuation, but also the capacity to achieve a respectable predictive prowess, given

a modest sample size of data. Some conclusions on the algorithmic synergy between preprocessing methods and also machine leaning ones were also deductible from these results, one such example being how it appears sensible joining standardization with TSVD, given the former centres the data and ensures a fair variance measure between the variables, for the latter. Albeit a tailored choice of methods to the data at hand always being the ideal approach, running similar tests on different datasets could add a level of validation to some of the findings done in this work, either refuting them as one-offs, or by finding matching patterns that could serve as future guidelines on what set ups are more likely to be worth prioritizing running tests on, based on not just theoretical, but practical synergies shown. Last but not least, it is also conveyed in this work that optimization initiatives, combining the hardware capabilities of the machine through parallelization with more efficient ways to compute operations via vectorization, were able to provide speed-ups of at least 280%, an incentive for not only a quicker execution, but also bolstered scalability, and reduced energy costs.

There are, notwithstanding, areas that could be explored in a posterior iteration of this project or, in more general terms, future research based on it. Despite the assured functionality of both preprocessing and model training scripts, the extensive way in which they were coded may not be ideal in a more production oriented context. Using a sci-kit (or equivalent) pipeline [7] the lost granularity is compensated by the gained maintainability, organisation and more shareable nature of the simplified code. A deeper emphasis could also be put in evaluating different parameter values, possibly with the help of cross-validation, in the suitable methods. These more ideal parameter tunings would further enhance the model's predictive capabilities, and possibly bring out new set-ups that did not perform at their best with the currently chosen parameters. On the optimisation side, building CUDA kernels, using Numba, would enable a GPU-based acceleration to the multiple operations as well. Due to specifications of the kernels, this would have required a refactoring of the current code (such as kernel values having to be returned indirectly via mutable structures, like arrays, since typical return statements are not allowed), falling outside of the scope of this project. It is, in any case, an approach that could be worth exploring in future work, especially if amount of data is increased.

Nevertheless, the author believes the most crucial goals set out for this personal project have been met, and hopes the work done can solidify his knowledge of and connection to the field, as well as, possibly, that of others.

Appendix A

Extra images

	encoded_listing_date_day_sin	encoded_listing_date_day_cos
481	0.848644	0.528964
182	0.790776	-0.612106
482	0.848644	0.528964
573	0.724793	0.688967
894	-0.998717	-0.050649
...	...	...
1164	0.937752	0.347305
500	0.848644	0.528964
941	-0.968077	-0.250653
421	0.848644	0.528964
1829	-0.299363	-0.954139

Figure A.1: Cyclic encoding of *day* using sine and cosine functions.

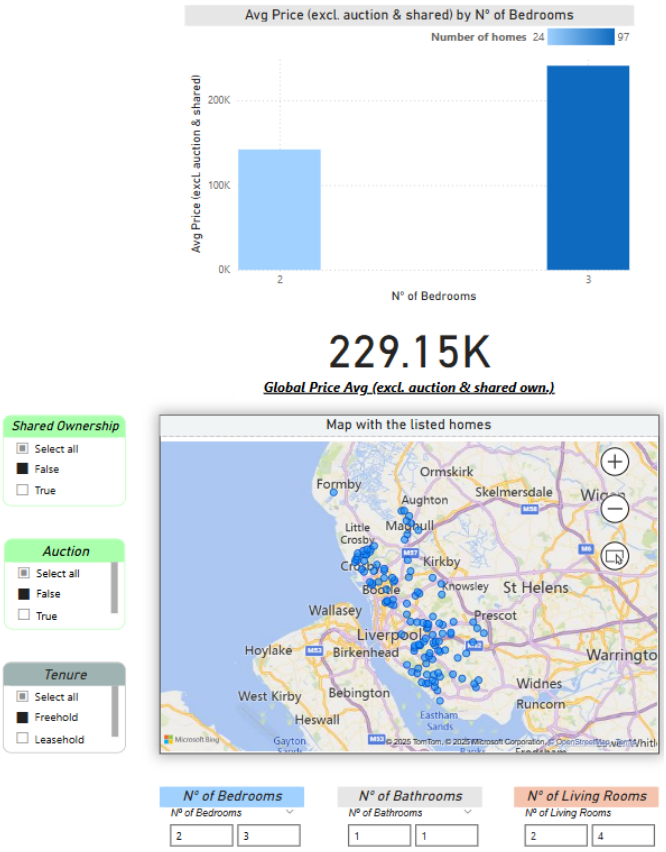


Figure A.2: Example of a search being made for a freehold home, with 2 to 3 bedrooms, 1 bathroom and 2 to 4 living rooms. The bar chart indicates the average price for the 2 and 3-bedroom houses as well as the number of findings for each. In the map one can see the location of and choose any of the matching homes (once one is selected, the remaining visuals will update to the details of that home specifically).

(a) Snippet showing the effect of using PCA and IQR on the accuracy metric for different models and preprocessing steps.

(b) Effect of using LDA and IQR on the accuracy metric for different models and preprocessing steps.

Figure A.3: Caption next page.

Figure A.3: (Previous page.) Example of an interaction where the user selects IQR as the outlier detection method (not visible due to image size limitations) and alternates between PCA and LDA in feature selection. Accuracy is the performance metric and *tenure* the variable being predicted. Emphasis on how the maximum accuracy of the different encoding methods displays Base-N encoding as the distinguishably better option for the LDA algorithm (A.3b) . For PCA, however, the opposite happens with One Hot and Target encodings performing significantly better than Base-N (A.3a). Conclusions such as this can be further backed by the discrepancy on the (in this case, *Encoding*) average prediction accuracy bar chart.

```
vectorized_method_time = timeit.timeit(vectorized_method, number=100)
nonvectorized_method_time = timeit.timeit(nonvectorized_method, number=100)
```

Figure A.4: Code used to obtain the timing of each method

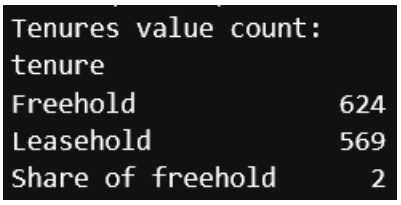
```
Total run-time for the current build (1 CPU):
19.241160 seconds.
```

(a) Run-time of the whole program using 1 CPU.

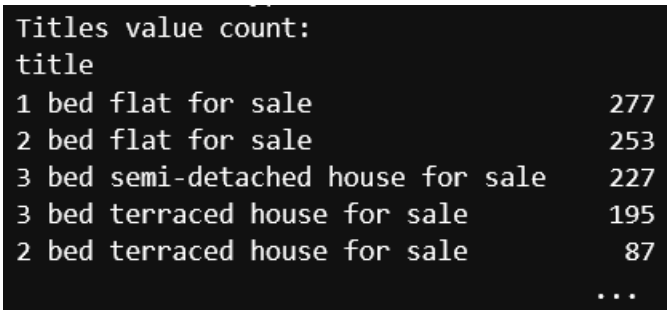
```
Total run-time for the current build (all 12 CPUs):
5.053823 seconds.
```

(b) Run-time of the whole program using all 12 CPUs.

Figure A.5: Comparison of the run-times between the more serial and parallel implementations. A.5b became $\approx 280\%$ faster than A.5a. The test was done for a given preprocessing, prediction model and target-variable build.



(a) Tenure values distribution.



(b) Title values distribution.

Figure A.6: Value distributions of the categorical variables *tenure* and *title*

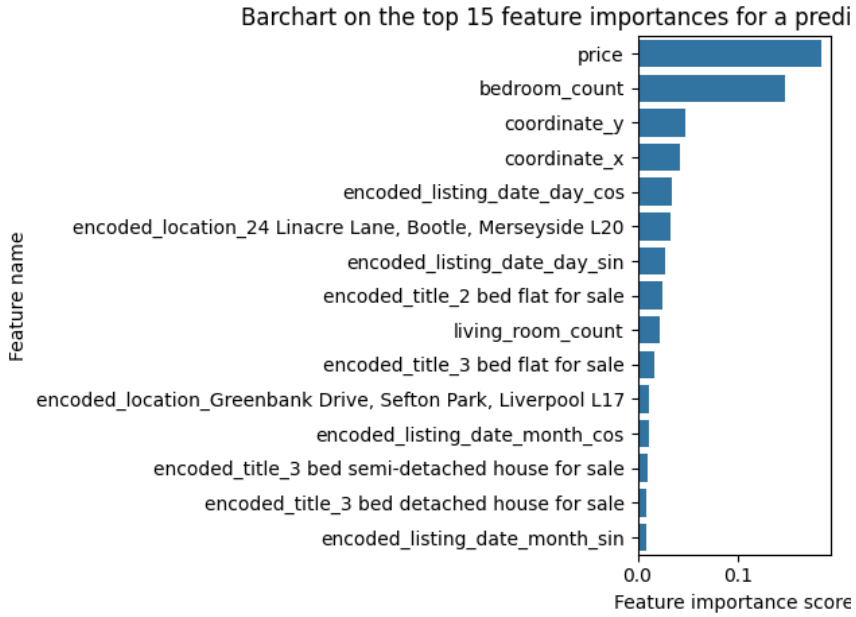


Figure A.7: Feature importances for the target variable *number of bathrooms* using the Random Forest Regressor algorithm. The result was obtained using 100 trees and the importance is derived from the normalized *Gini impurity* criterion. Only the top 15 importances are present for better visibility.

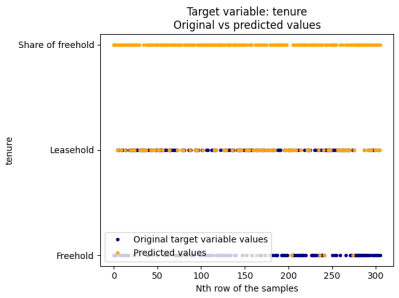
```
Variance of the predictors:
price                                  1.000839
coordinate_x                           1.000839
coordinate_y                           1.000839
bedroom_count                          1.000839
living_room_count                      1.000839
encoded_listing_date_month_sin         0.066595
encoded_listing_date_month_cos         0.124513
encoded_listing_date_day_sin           0.559375
encoded_listing_date_day_cos           0.383455
```

(a) Variance of the predictors (snippet 'a' of the output).

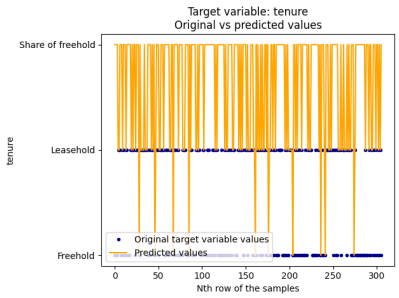
```
encoded_location_20 Water Street, Liverpool L2          0.010787
encoded_location_24 Elm Road, Walton, Liverpool L4      0.000838
encoded_location_24 Linacre Lane, Bootle, Merseyside L20 0.000838
```

(b) Variance of the predictors (snippet 'a' of the output).

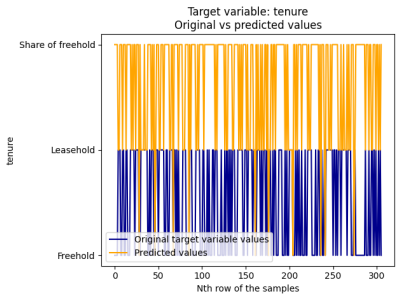
Figure A.8: Snippets of the output of the predictors' variances, using *number of bathrooms* as the target variable.



(a) Dot plot of both true and predicted values.



(b) Dot plot of the true values together with a line chart of the predicted ones.



(c) Line chart of both true and predicted values.

Figure A.9: Chosen example of true versus predicted values plots. All plots are equivalent, being merely represented differently by dot plots, line charts, or a mix, to aid visualization. This run corresponds to best (macro-averaged) Precision score obtained for the target variable *tenure*, using the SGD Classifier. It serves to demonstrate that even individually high Precision values, and lower but reasonable macro-averages do not guaranteedly translate to an overall good generalisation capability of the model. In this case, the number of *Share of freehold* False Positives was visibly high, but since there were no True Positives for that class, the amount of FPs will not impact its Precision (0), despite certainly impacting the quality of the model

Appendix B

Extra tables

Metric	MAE	
M.l. algorithm	Lasso Regression	Random Forest Regression
Price	45,233*	38,844
min	5000	5000
max	800,000	725,000
Number of bathrooms	0.1585	0.1479
min	1	1
max	4	4

Table B.1: Best MAE results obtained for the used regressors. *min* and *max* represent the minimum and maximum of target variable above, given the preprocessing options that lead to the presented best metric result. *Originally, the best obtained value was 44,703. For reproducibility, however, the presented value was the one chosen, in practice, as the best (reproducible) one.

Execution times in seconds (1000 runs)	Accuracy		Precision		Recall	
M.l. algorithm	SGDC	MPC	SGDC	MPC	SGDC	MPC
Tenure	88.89	89.22	0.6298	0.5685	0.8647	0.8154
numeric imputation method for X	Median imputat.	Median imputat.	Mean imputat. / Median imputat. / Forward fill imputat.	deletion	Forward fill imputat.	deletion
categorical imputation method for X	Mode imputat.	deletion	(any)	(any)	deletion	(any)
numeric imputation method for y	Mean imputat.	Mean imputat.	(any)	(any)	(any)	(any)
categorical imputation method for y	deletion	deletion	deletion	deletion	deletion	deletion
outlier detection method	IQR	IQR	Z-score	(any)	Z-score	(any)
outlier treating method	Trimming	Trimming	Capping	Custom (expert)	Trimming / Capping	Custom (expert)
encoding method	One-hot Encoder	One-hot Encoder	Target Encoder	Base-N Encoder	Target Encoder	Base-N Encoder
scaling method	Mean normalis.	Robust Scaler	(any)	Standardization	Robust Scaler	Standardization
feature selection method	PCA	PCA	LDA	Truncated SVD	Kernel PCA	Truncated SVD

Table B.2: Preprocessing settings which provide the best results obtained in 4.3. A cell filled with "(any)" means the result could be obtained in any of the options available for that setting, rendering it inconsequential. Note that Accuracy's results are in a setting where PCA is used in conjunction with One-hot Encoder, meaning that the SVD solver in the resulting dimensionality will be randomised. This leads to slightly better or worse results depending on the run, ultimately meaning the associated preprocessing methods may have only been the best in certain runs. Evaluation of all steps but feature selection should therefore, in this case, be more focused on the preprocessing methods of the Precision and Recall columns.

Appendix C

Bibliography

Academic

[1] Z. Ma and G. Chen, "Bayesian methods for dealing with missing data problems," *Journal of the Korean Statistical Society*, vol. 47, no. 3, pp. 297–313, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1226319218300176

Non-academic

[2] Z. Ltd., last accessed 10 November 2023. [Online]. Available: https://www.zoopla.co.uk/robots.txt

[3] GOV.UK, last accessed 10 January 2025. [Online]. Available: https://www.gov.uk/guidance/exceptions-to-copyright

[4] Scikit-learn, last accessed 14 March 2025. [Online]. Available: https://scikit-learn.org/stable/computing/parallelism.html

[5] D. Pavel, last accessed 18 March 2025. [Online]. Available: https://github.com/dubovikmaster/parallel-pandas/tree/master

[6] Scikit-learn, last accessed 24 April 2025. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

[7] ——, last accessed 1 May 2025. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html