ML Instant House Valuation

# Analysis

## Background

I will be creating a house valuation web app which will allow a user to input a set of property features (such as the number of rooms, floor size, property age etc.) and be given an instant valuation based on a machine learning solution. The idea began by brainstorming several machine-learning related tools; first a weather forecasting tool, then a stock price predictor, but finally I settled with a house price valuation tool. I found it more interesting to be forced to build, analyse, and develop a solution myself, as there was limited material for this project online. The project began more as a research task, but I decided to develop it as a web service, after stumbling across a [Diamond price prediction](http://diamonds.foostack.ai/) project (http://diamonds.foostack.ai/) (now inactive) online.

## Current systems analysis

To get an idea of how my system should function, I will look at popular pre-existing house valuation tools. While most of these systems are not that distinct to each other, I would like an insight on their parallels and how I can achieve something similar. Ideally, I would like a suggestion to the data and methods the different tools use; however, I doubt I will be able to as it is proprietary data. The UI and aesthetics are not my main concern in the analysis, but I will also follow and expand on some of the design components that are similar between them all.

### Zoopla

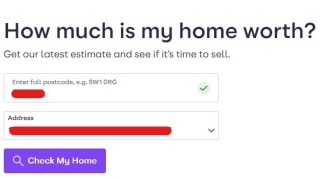


Figure 1 Zoopla user form

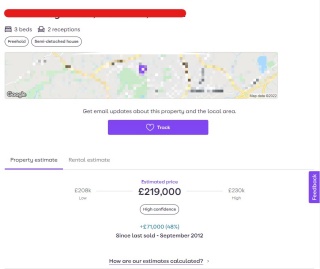


Figure 2 Zoopla price output

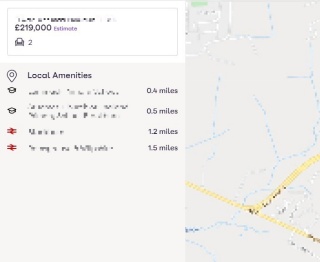


Figure 3 Zoopla area navigator

I like Zoopla’s system as it is amazingly simple to use and understand, its UI is visually appealing, and it includes other data apart from just the valuation, such as local amenities, making it even more helpful to an end user.

The Zoopla valuation uses an AVM (Automated Valuation Model), built on proprietary data collected by Hometrack, to instantly valuate the property. Additional open-source data is also listed, e.g., HM Land Registry, which I find ideal as it will allow me to explore and potentially adopt the same datasets from the same reliable sources. I will try providing additional data, such as sold prices and area statistics, much like Zoopla; however, unlike Zoopla, I will adopt a system where the end user is asked for the property’s location as well as other property features, such as its number of rooms etc., which could allow for the valuation of properties that potentially do not exist.

### Yopa

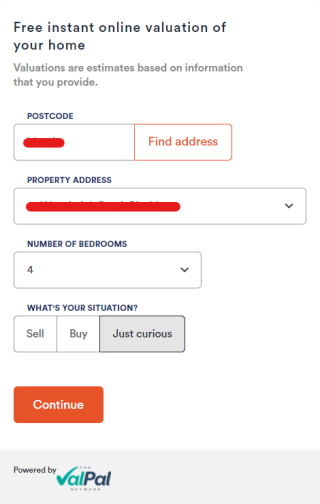


Figure 4 Yopa user form

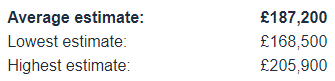


Figure 5 Yopa price output

I like the simplicity of the Yopa valuation, however only requiring the number of bedrooms and location does not seem to yield a perfectly accurate result. In my system, I would ideally require at least 5 features for the resulting estimation, such as number of rooms, floor area and location. Yopa, like Zoopla, also implements a low and high estimation alongside its valuation, and I will also attempt to implement this.

### OnTheMarket

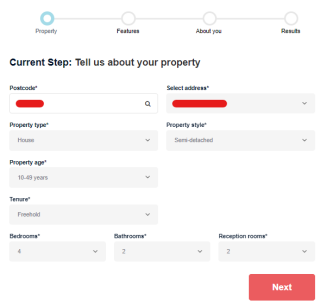


Figure 6 OnTheMarket user form, page 1

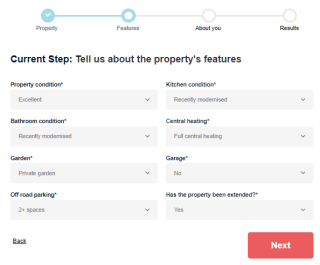


Figure 7 OnTheMarket user form, page 2

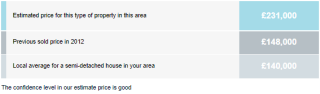


Figure 8 OnTheMarket price output

OnTheMarket valuations are powered by Property Price Advice and are calculated using algorithms which consider local market conditions, recently sold prices and similar neighbouring properties. They claim that their “algorithms and statistical models provide figures within 15 per cent of real-time property values”.

The system requires 15 features in addition to the location, which makes the valuation seem more accurate to the end user, unlike the Yopa valuation which only required the number of bedrooms. However, I will aim to reduce the number of required features, and keep all my parameters on a single page, to make the process easier and less daunting. I will try sourcing data which includes similar features such as the property age, number of rooms, property type etc.

### CheckMyStreet

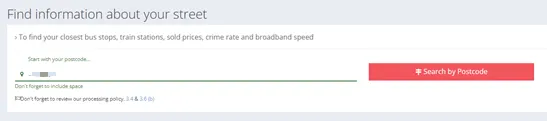


Figure 9 CheckMyStreet user form

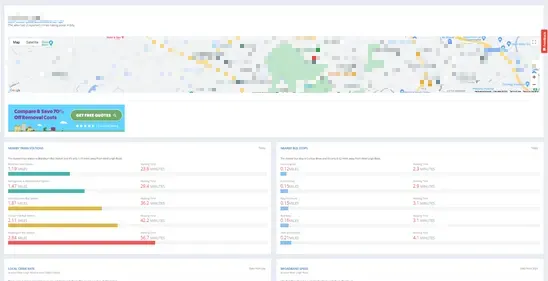


Figure 10 CheckMyStreet output screen

While this is not a valuation tool, I have decided to have a look at it because it provides useful information to the end user, such as local bus stops and train stations, the crime rate, and the local average price. It also calculates the distances and walking times, which could introduce added complexity to my project if I choose to implement it. It uses public data such as the HM Land Registry. I would also like to provide this information alongside the valuation as I believe it could be more helpful and beneficial to the user.

## End User Interview

I asked 3 home-owners questions on their thoughts of the project and how I should develop it; I think the responses of potential users will be particularly helpful in the direction of which the development of my project goes:

1. **Would you consider an online property valuation as an accurate way to get the price of a property?**
   1. I would consider an online valuation to get an idea of a house’s value. I would use a tool like this before contacting an estate agent to assess the property in real life.
   2. I’d use an instant valuation tool to see if a property is worth pursuing, it would save a lot of time instead of having to contact somebody myself.
   3. I don’t think a service like this would be highly accurate, but I do think it would give a useful estimation and could give a reliable prediction.
2. **What characteristics of a property do you think would most impact its valuation?**
   1. I think the age, size and property type would be useful in telling the price of a house. However, I believe the value of properties in the area would have greatest impact on a house’s price.
   2. The price of surrounding properties would be the most useful in the valuation of a house, along with its total square footage, number of rooms, and general local area information like population and crime rate.
   3. The quality of the neighbourhood and the prices of other houses nearby have the most impact on a house’s value. Property features such as the size and the type also impact a property’s price.
3. **What additional features would you like to see, alongside the valuation?** 
   1. I’d like to see the average price of houses in the local area, or the average price of similar property types.
   2. I think it’d be helpful to see local area statistics such as the population and average house price in the area. Additionally, I don’t think a point estimation is immensely helpful, so I’d like to see a valuation within an interval, with a low and high estimation.
   3. It would be interesting to get predictions for the price of a property years in the future. It would also be helpful to see the previous sold price of a property.
4. **How would you like the information displayed to you?**
   1. I’d like to see the information displayed as simply as it can, clutter and unnecessary information can be quite annoying - like asking for email and other data that has nothing to do with the actual predictions. I would prefer all the information easily visible in one window.
   2. I think displaying a map of the property’s area along with its valuation would be helpful, especially for people who consider moving into the area. There should be an option to have the predictions emailed to you, but not as a requirement.
   3. A simple web form should suffice for inputting the property details. The output could be given in a separate page or pop-up with all the information.

In conclusion, the homeowners would consider using the service as a rough estimate for a property’s value before seeking professional advice. All 3 participants think that local area information could be useful in the valuation, as well as the property size and type. I liked the idea to include local area statistics in the output to the user, and I agree that unnecessary clutter should be avoided when serving the information to the end user.

## Problem Modelling

My initial thoughts on how to approach the problem. I will try implement the system in four (or potentially five) stages:

* *Stage 1.* Data sourcing - collecting and stitching data from various sources to build a large dataset (aiming for around 1 million rows).
* *Stage 2.* Data sanitisation and pre-processing - transforming the dataset and its features to allow for predictive modelling
* *Stage 3.* Model training - testing and training various machine learning models on the dataset to maximise accuracy
* *Stage 4.* Web development - implementing the server and client side for users to access.
* *(if possible) Stage 5.* App deployment - deploying the client-server model to the web for potential use.

Below covers the general implementation and structure of the project, split into its distinct stages.

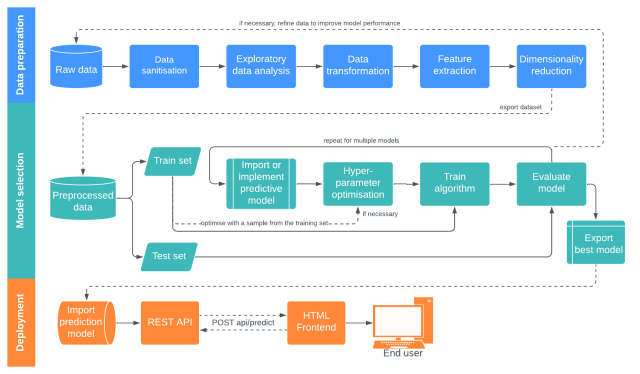


Figure 11 Project flowchart

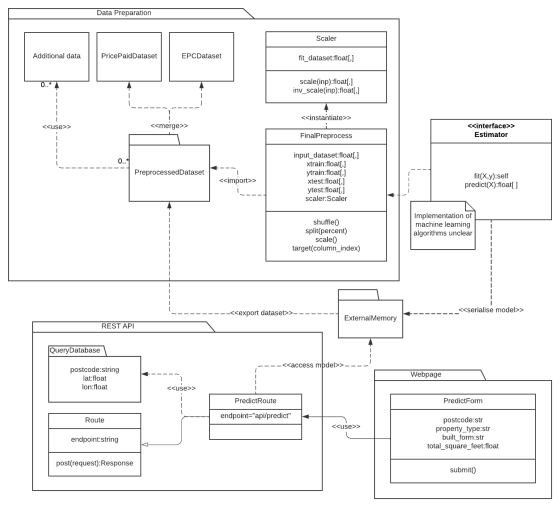


Figure 12 UML diagram

I am currently unclear on what machine learning models to use and how to pre-process the data. These decisions will be made as the data is collected and in exploratory data analysis. I will fit multiple models on the dataset and test their performance to pick an algorithm, I may also ensemble/stack multiple models if it proves useful. I may need a database for features of the model which require context from the inputs provided by the end user, such as geospatial or local area data.

The web service itself, which will serve the predictions to the end user, will look somewhere like:

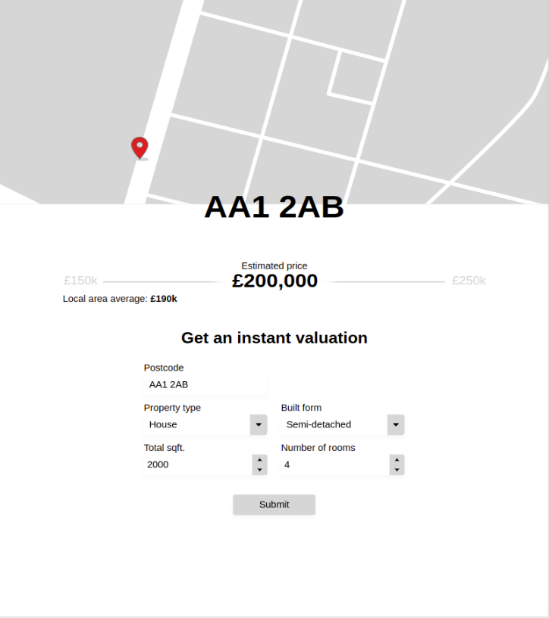


Figure 13 Initial design

Drawing inspiration from the Zoopla and OnTheMarket user interfaces, with a simple form, an interval prediction, and a map of the area using Google Maps API.

## Algorithms

Below I will be detailing some of the algorithms that I will use/test in my project. I will implement each predictive algorithm and test it against my dataset; however, I will only use one of the predictive algorithms alongside conformal prediction. Using metrics such as MSE, R2 etc., I will determine which model performs the best and will select that model manually. Additionally, after each model is implemented and fitted against my data, I will produce visualisations to give a clearer and more interpretable depiction of how well it is performing. Each model will be fitted with all the available data and conformal prediction to produce intervals for given inputs; this will avoid me having to retrain models after selection.

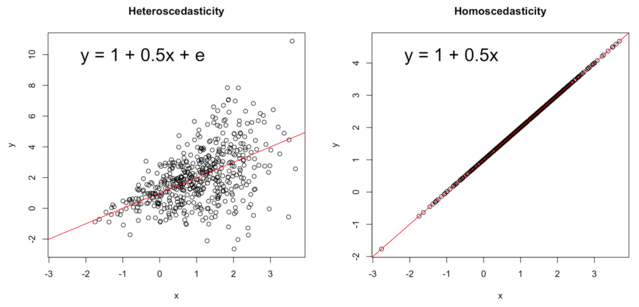
### Conformal prediction

Conformal prediction is a technique used to assess the uncertainty of predictions produced by a machine learning model. Given an input, conformal prediction estimates a prediction interval in regression problems and a set of classes in classification problems. Both the prediction interval and sets cover the true value with high probability (usually provided by the user). Steps to producing a conformal prediction model:

1. Identify a score function to measure the discrepancy between model outputs and expected outputs *y*. This metric is critical as it decides what prediction sets can be obtained. For instance, in regression problems, we could take the absolute error  as the score function.
2. Compute nonconformity scores for the model by splitting the training set into a proper set and calibration set. The model is only trained on the proper training set; and scores are computed on the calibration set where   for the *n*th sample.
3. Use the underlying algorithm to make a point prediction  and take the (th quantile *Q* from the score set - where *α* is a user-provided significance level. Finally, output interval as the model’s prediction.

### Stochastic gradient descent and linear regression model

Regression models are used to describe relationships between variables by fitting a line to the observed data. Multiple linear regression is used to estimate the relationship between multiple independent variables and one dependent variable. The model makes assumptions about the dataset:

* **Homogeneity of variance (homoscedasticity)** - the observations in the dataset must be about the same distance from the regression line

*Figure 14 Heteroscedasticity vs Homoscedasticity*

1. **Independence of observations** - there are no hidden relationships among the independent variables.
2. **Normality** - the data follows a normal distribution.
3. **Linearity** - the line of best fit through the data points is a straight line, not some polynomial/exponential curve.

Multiple linear regression generalises a dataset as the output being a weighted summation of its inputs.

In matrix form, it can be written as a dot product, as shown:

Where,  
: predicted  
*x*: independent variables vector  
*w*: coefficients vector, i.e., weights.  
*b*: constant term i.e., bias.

The stochastic gradient descent algorithm is used to adjust the weights and bias for the model. The algorithm requires a differentiable cost function to derive the gradients of each adjustable parameter. I will be using the **half mean squared** cost function, as it will differentiate nicely making it a lot easier to implement. The cost function is defined as such,

The goal is to find the values of the  and  terms which minimise the error. This can be done by taking the partial derivative (gradient) of the cost function *J* with respect each parameter,

Now iteratively update the parameters until they converge,

Where *η* is the learning rate.

The implementation of linear regression with stochastic gradient descent will be a great starting point when implementing the more complex models such as the multilayer perceptron, which similarly uses the weighted input function to make predictions and a gradient descent algorithm to optimise its parameters.

### Neural Network/Multilayer Perceptron model

A multilayer perceptron, or more broadly a neural network, is a feed-forward neural network consisting of at least three layers of nodes; an input layer, a hidden layer, and an output layer.

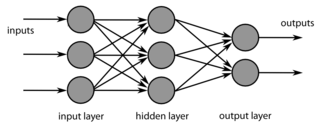


Figure 15 Typical feed-forward structure

The input layer receives an input signal to be processed. The model’s prediction is performed by the output layer. The hidden layers between the input and output layer are the main computational engine of the model, performing several manipulations and processing. MLPs can be used to approximate any continuous function and can solve problems which are not linearly separable by using non-linear activation functions.

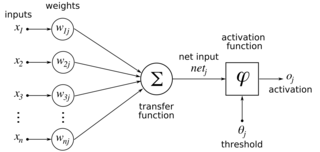
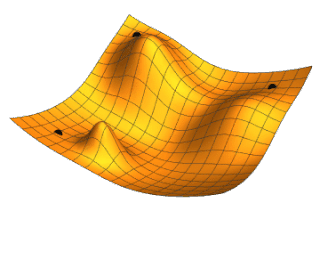


Figure 16 Artificial Neuron

An MLP learns in two stages:

1. **Forward propagation**, where data is fed from the input layer through the hidden layers to the output layer. For a general model of *L* layers with weights *W* and activation functions *σ*, forward propagation can be expressed as a function composition as shown:
2. **Backpropagation**, where, by using the chain rule, the partial derivatives of the loss function with respect to the various weights and biases are fed back through the network from the output layer through to the input layer. The differentiation produces a gradient along which the parameters may be adjusted towards the minima. See **Backpropagation algorithm** for more detail.

*Figure 17 Gradient descent illustration*

* These stages are repeated until the loss converges towards a minimum.

### Backpropagation algorithm for Multilayer Perceptron

Backpropagation is a widely used algorithm for training feed-forward neural networks used to compute the gradient of a loss function with respect to the weights of a network for a training set of input-output examples. The gradient computed by backpropagation can be used with other algorithms such as Stochastic Gradient Descent (or alternatives) to iteratively optimise the weights and minimise error in a network.

Below shows the mathematical logic behind the algorithm. The following example is for a feed-forward network of *L* layers.

This can be written as an iterative formula:

For the model above, the objective is to minimise the cost function of the training sample, by optimising the various weights of each layer. To do this, we use differentiation to produce a gradient along which the parameters can be adjusted towards the minima. We begin by finding the derivative of the loss with respect to *x*, which can be calculated using the chain rule as such:

Chain rule:

The gradient of the loss with respect to *x* is the transpose of the above, therefore, the matrices are transposed, and the order of multiplication is reversed, such that:

The backpropagation consists of then evaluating this expression from right to left, i.e., starting from the output layer, computing the gradient at each layer on the way. To avoid unnecessary repeated computation of expressions, we can introduce a new variable *δl* - the “error at layer *l.”* This variable is a vector with length of the number of nodes in its associated layer. Each value can be interpreted as the loss attributable to that node for the given input-output sample. ​*δl* can be computed recursively, starting backwards from the last layer *L,* and moving through to the input layer, such that:

The gradients of the losswith respect to weights for the layer *l* can now be calculated:

Finally, the weights can be updated (demonstrated here with stochastic gradient descent):

Where,

*L*: Number of layers in network

*σ*: Non-linear activation function

*J*: Loss function taking parameters *y* (expected output), and ​(actual output)

*Wl*: Weight matrix of layer *l*, i.e., all the connections between nodes in the current and next layer.

*AT*: Transpose of matrix *A*

∇*ba*: Gradient of *a* with respect to *b*

: Hadamard (elementwise) matrix multiplication of *A* and *B*, products taken from left to right.

*al*​: Activated output of layer *l.*

*δl*​: Error atlayer *l.*

*η*: Learning rate of SGD algorithm

### Adam Optimisation for Backpropagation Algorithm

Of all the available optimisation techniques, I will be using Adam optimisation as it converges to minima quicker compared to its alternatives.

Adam optimisation combines the advantages of two prior methods of stochastic gradient descent:

* **Adaptive Gradient Algorithm** (AdaGrad), which uses a per-parameter learning rate that improves performance on problems with sparse gradients, such as natural language processing or computer vision.
* **Root Mean Square Propagation** (RMSProp), which also uses per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e.g., how quickly it is changing). This means the algorithm does well on noisy non-stationary data, such as sales or online data.

The method only requires first-order gradients with little memory requirement, making it efficient and less computationally intensive.

Given some parameter to optimise *ϕ* and gradient of cost function *g*, we can compute the first moment estimate (mean), *m*, and the second moment estimate (uncentered variance), *v*.

As *m* and *v* are initialised as 0, they are biased towards zero, especially during the initial time steps or when the decay rates are small. Therefore, to counteract these biases, we need to compute bias-corrected first and second moment estimates:

Finally, we can update parameter *ϕ*:

Where,

*ϕ*: Objective parameter

*g*: Gradient of some cost function w.r.t to the objective parameter

*m*: first moment (mean) estimate - initialised at 0.

*v*: second moment (variance) estimate - initialised at 0.

*t*: time step - initialised at 0, incremented before each iteration.

*η*: initial learning rate

*β*1​: first moment estimate coefficient - default 0.9.

*β*2​: second moment estimate coefficient - default 0.999.

*ϵ*: term to avoid zero division – default 10-8

### CART (Classification and Regression Tree) algorithm

The CART (Classification and Regression Tree) algorithm is a divide-and-conquer algorithm used to build a decision tree (as a predictive model) to predict the output of a given sample, using attributes observed in each training set. The model is very intuitive and easy to interpret/explain.

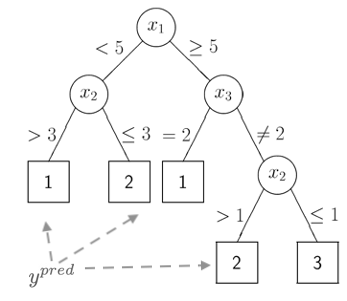


Figure 18 CART model

Furthermore, it is robust and does not require any prior normalisation or scaling of the dataset. It is also unaffected by outliers and finds important variables automatically, unlike linear regression models. However, decision trees are prone to overfitting, and slight changes to the dataset can cause substantial changes in tree structure, causing instability. The algorithm, while can be modelled to perform the task, is inadequate for applying regression and predicting continuous values, and cannot extrapolate beyond its observed set and will produce discrete predictions. To mitigate some of these issues, decision trees are often used in ensemble methods including random forests and gradient boosting machines, which aim to reduce overfitting.

The algorithm performs binary splits using the features of a dataset to build a tree, until all the nodes are ideally “pure.”

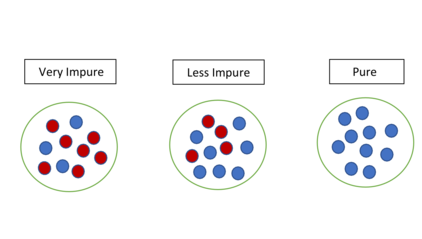


Figure 19 Entropy illustration

Optimal splits are found by comparing the impurity of multiple potential splits. Popular metrics used to measure impurity are:

* Gini impurity (classification) - calculates the probability that a specific feature is classified incorrectly when selected randomly. A pure node of Gini index 0 indicates that all its contained elements are of one unique class.

Where*Pk* ​ is the probability of class *k*.

* Variance reduction (regression) - variance is used for calculating the homogeneity of a node. If a node is entirely homogeneous, i.e., pure, then the variance is zero.

Where ​ is the mean of input data *y.*

To determine the quality of a split using some impurity metric, we use information gain, which can be described as the reduction of information entropy *H* from a prior state (parent node) to a new state (split into children nodes):

Where,  
*p*: parent node

*l*: left subtree

*r*: right subtree

*nA*​: number of elements in a set *A*

*H*(*x*): entropy metric function

The algorithm starts at the root node, with the entire dataset. It then compares potential splits (all permutations of features and unique thresholds) and chooses the split with the highest information gain. It then recursively repeats this process on its resulting splits until the nodes are pure, or until the process is stopped manually. The algorithm is stopped by user specified parameters, such as when the tree grows to a maximum depth provided or if a split produces a minimum number of samples.

### Random forest model

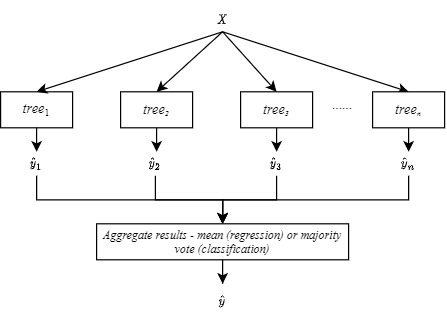


Figure 20 Random Forest architecture

Random forests are a bagging ensemble learning method which construct multiple decision trees at training time. In the bagging method, also known as bootstrap aggregation, a random sample from the training set is selected with replacement - allowing data points to be used multiple times. After several data samples are generated, several estimators are then individually trained. When making a prediction, outputs from each individual tree are aggregated to yield a more accurate result; for regression, the mean is taken, and for classification the majority vote.

### Gradient boosting machine

Figure 21 Gradient boosting machine architecture

Gradient boosting is a boosting ensemble learning method that combines a set of weak learners (decision trees) into a single strong learner to minimise training errors. In boosting, a random sample of data is selected, fitted with a model, and trained sequentially; each model tries to improve the errors of its predecessor, by training on the previous predictor’s residual errors. The method combines both the gradient descent and boosting methods, hence the name gradient boosting.

### API and HTTP requests

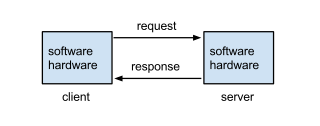


Figure 22 Client-server model

Throughout my project I will make multiple web requests to:

* scrape data from the web
* download datasets programmatically.
* make API requests with findthatpostcode.uk to validate and geocode user input.
* make API requests with Google Maps for illustration purposes.
* Additionally, I will use a small internal API within my project to connect the Python backend and the HTML/JavaScript frontend.

## Objectives

In my project I aim to accomplish several things:

1. Provide the user with reliable and accurate predictions:
   1. Building a comprehensive up-to-date dataset with easily interpretable features:
      1. Collect and combine raw data from multiple data sources:
         1. with over 5 million rows of raw data
         2. and 10+ columns of unique features
      2. Exploratory data analysis and data visualisation in the data research stage
      3. Compile a final training dataset with:
         1. with over a million rows of unique data points
         2. And 5+ distinct features
   2. Build, train, and test predictive models:
      1. Linear regression
      2. Multilayer perceptron (neural network)
      3. K-Nearest neighbours
      4. Regression decision tree
      5. Random forest
      6. Gradient boosting machine
   3. Build a predictive model that:
      1. is moderately small in storage size < 2MB.
      2. yields accurate predictions, with R2 scores > 0.8 (i.e., model is fitted to account for 80% of the dataset’s variability)
   4. Provide a prediction interval, for which the true value is probable to lie within, using conformal regression techniques.
2. Serve predictions to the user in a web app:
   1. Geocode postcodes to corresponding coordinates, utilising the *findthatpostcode.uk API*.
   2. Use *Google Maps API* to visualise the area with the given postcode.
   3. Provide useful statistics about the user-input postcode:
      1. Useful census data
      2. Index of Deprivation data
   4. Fast performance (by limiting dependencies and complexity)
   5. Display outputs and errors with maximum readability and ease:
      1. Responsive design
      2. Aesthetically consistent
      3. Subtle micro-animations
3. Deploy the web app across the internet:
   1. Use *GitHub* for source control.
   2. Use *Google App Engine* to host the site.

## Limitations

My project will be confronted with some limitations:

* It is impossible for the prediction model to predict the price of a property exactly, however, the answers to the questionnaire show that this is not what users want, rather, they want a rough ballpark estimate, to give them an idea before professional consultation.
* Computational limitations/requirements, such as resource limits within the free tier of Google App Engine and hardware constraints of the machine used in training, forces me to limit the complexity and size of the prediction models. Doing this could train models which may underperform as compared to their maximum potential.
* Due to the subjective nature of house valuations, it is difficult to confirm whether a valuation is actually accurate or not; aside from empirically comparing a prediction with its actual sold price. However, the sold price may not be indicative of the true value of the property as the property may have been or under/over-valued. Therefore, I will also compare several valuations with similar tools, highlighted in *Current Systems Analysis*, alongside the empirical tests used in model selection.

## Feasibility of potential solutions

To build predictive models, I have at my disposal a variety of different tools and frameworks, most notably:

* **Python** – Python is widely used for data science and web development, both which my project entail. With its simple syntax and large ecosystem containing a wide selection of libraries built for data science, Python is an ideal solution. However, due to dynamic typing and being an interpreted language, it suffers from performance issues. Despite this, the language supports importing directly from C, which major Python libraries like NumPy are built in; therefore, we can leverage the speed of C whilst still using Python. Despite this, my from-scratch implementations will still be slow and inefficient, especially recursive algorithms such as the CART algorithm, therefore I will need to find optimisations I can make to allow it to perform faster.
* **Java** – Java is also a strong contender as it is fast, well supported, and has a huge ecosystem. Additionally, the language is secure due to its use of bytecode run on Java Virtual Machines. However, JVMs cause Java to have greater memory and processing requirements, making it more expensive on older/slower devices.
* **R** – The R programming language is a graphic-based language used for statistical computing, analysis, and visualisation in machine learning. It is better suited for research papers and reports and is not built for large-scale use.

I will use **Python** over Java and R, as I have considerable experience with it and will be able to dive directly into the programming, which for this project is quite advanced, instead of having to learn the syntax and intricacies as I would have to in Java. Additionally, many of the libraries suited for data science and machine learning in Java and R often try mirroring the semantics of their Python equivalents. Therefore, I find it much more reliable to stick to the source, which is built for Python.

Furthermore, I will build an isolated front-end using a simple **HTML-JavaScript stack**, rather than overengineering with frameworks such as React and Vue. Using frameworks and libraries would introduce unnecessary complexity and could overall hinder the performance of the small project that I am creating, instead of providing any real benefit.

Some key Python technologies I will be using for various stages of the project include:

* NumPy, Pandas and Scikit-learn for data manipulation and modelling.
* Matplotlib and Seaborn for data visualisation
* Flask for the server/backend.

In the HTML-JavaScript frontend I will be utilising:

* Python Flask – to serve the site by linking the backend to the frontend.
* jQuery – a small JavaScript library to make HTML event handling and document traversal much simpler to handle, thus speeding up development.
* SASS – a CSS extension to reduce code repetition, used for styling the user interface.

To test the site locally, I will *dockerise* the Flask app in a Docker container to emulate real Google App Engine deployment. Docker allows developers to build applications in *containers*, encapsulated environments containing all the dependencies required to run the program (code, runtime, system tools, libraries etc.), allowing the app to run in any environment. Additionally, I will be using Flask’s inbuilt development server with *hot-reloading* for a rapid workflow.

## Data

I will require vast amounts of housing data, with hundreds of thousands of rows, to build a predictive model that can accurately valuate a given property. Additionally, I am aiming to build a dataset with around 6-10 features (columns), therefore I will require around 30 features total to select the best from. As such, I will be collecting and stitching together data from multiple public sources, aiming to build a dataset of ideally around a million rows, however anything above 250,000 rows will suffice. I will be using:

* [**HM Land Registry Price Paid**](https://www.gov.uk/government/statistical-data-sets/price-paid-data-downloads) data – just under 4GB, with millions of rows of data, containing details of each transaction received at HM Land Registry since 1 January 1995.
* [**Energy Performance Certificates**](https://epc.opendatacommunities.org/) data – around 10GB, with millions of rows of data spread across multiple files and folders formatted as such: all-domestic-certificates/{LSOA}-{CITY}/certificates.csv. The data contains several columns about a property’s different attributes and energy efficiency.
* [**ONS National Statistics Postcode Lookup**](https://geoportal.statistics.gov.uk/datasets/ons::national-statistics-postcode-lookup-february-2022) – dataset mapping every UK postcode to its respective geographic coordinates, LSOA (Lower Super Output Area) code, and various other attributes.
* [**National Statistics UK House Price Index**](https://www.gov.uk/government/publications/about-the-uk-house-price-index) - time series dataset recording the house price index monthly since 1995.

Since some of the data is extremely large, I will utilise parallel processing and vectorization with Python libraries such as [NumPy](https://numpy.org/), [Pandas](https://pandas.pydata.org/), and [Dask Dataframes](https://docs.dask.org/en/stable/dataframe.html), to process and manipulate the data. Additionally, I will store data locally on the training machine (i.e., my computer) in .parquet file format as it is built to support very efficient compression and encoding schemes, making read/write speeds faster and more bearable, especially for larger data structures.

The app itself will leverage public APIs to convert user-input postcodes to latitude/longitude coordinates hence eliminating the need to build a server-side database to store such data.

Ultimately, this project does not require a database, or related structures, to store any type of end-user data as most of the project’s data aspect will be used pre-implementation in optimisation and training of algorithms - before users are even in the picture.

# Design

## Project workflow

My project structure will consist of three main parts:

1. Data sourcing, exploratory data analysis and manipulation
   1. Collect raw data.
   2. Analyse attributes of data which could benefit and hinder predictions.
   3. Transform data to be more explanative for the price of a property.
2. Predictive modelling
   1. Train multiple models on data
   2. Evaluate performance with analysis and visualisation.
   3. Select best model and serialize for persistent use.
3. App development
   1. Locally develop and test a web application, connecting the back-end model to a user interface.
   2. Containerise application to emulate real deployment and test.
   3. Deploy app to Google App Engine and serve site online to users.

The flowchart below better details my workflow for this project, detailing the general structure and processes used.

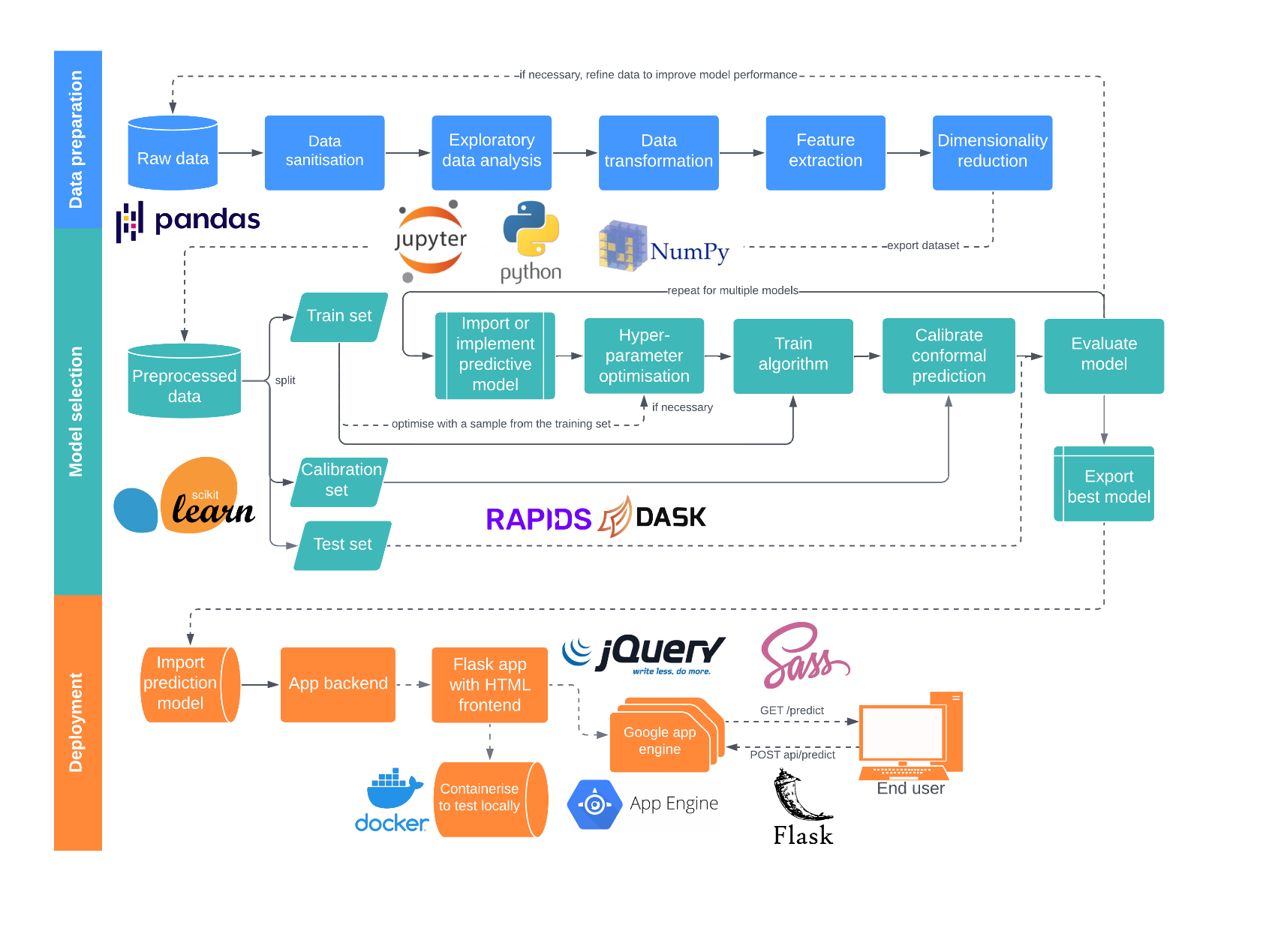


Figure 23 More detailed project flowchart

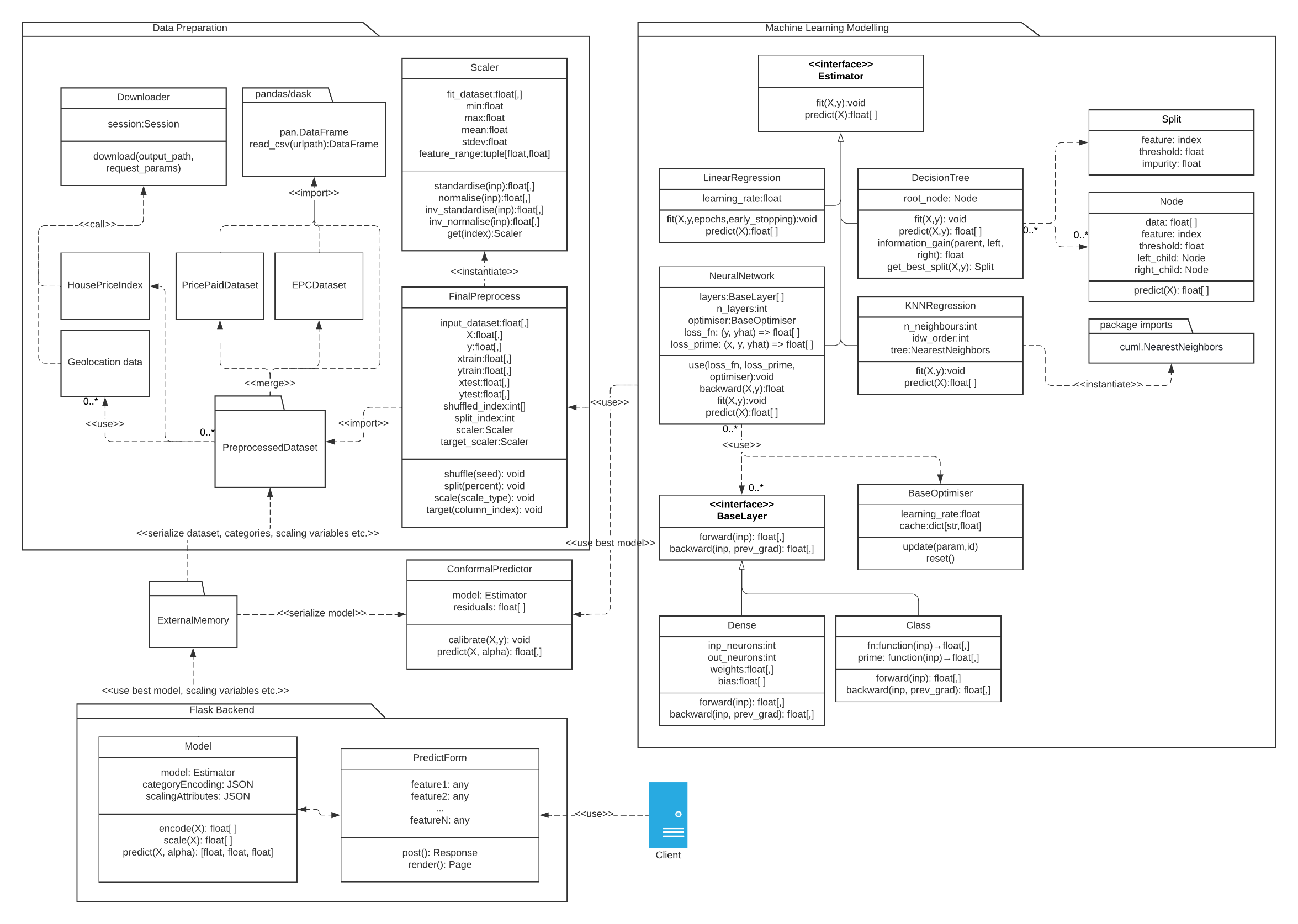
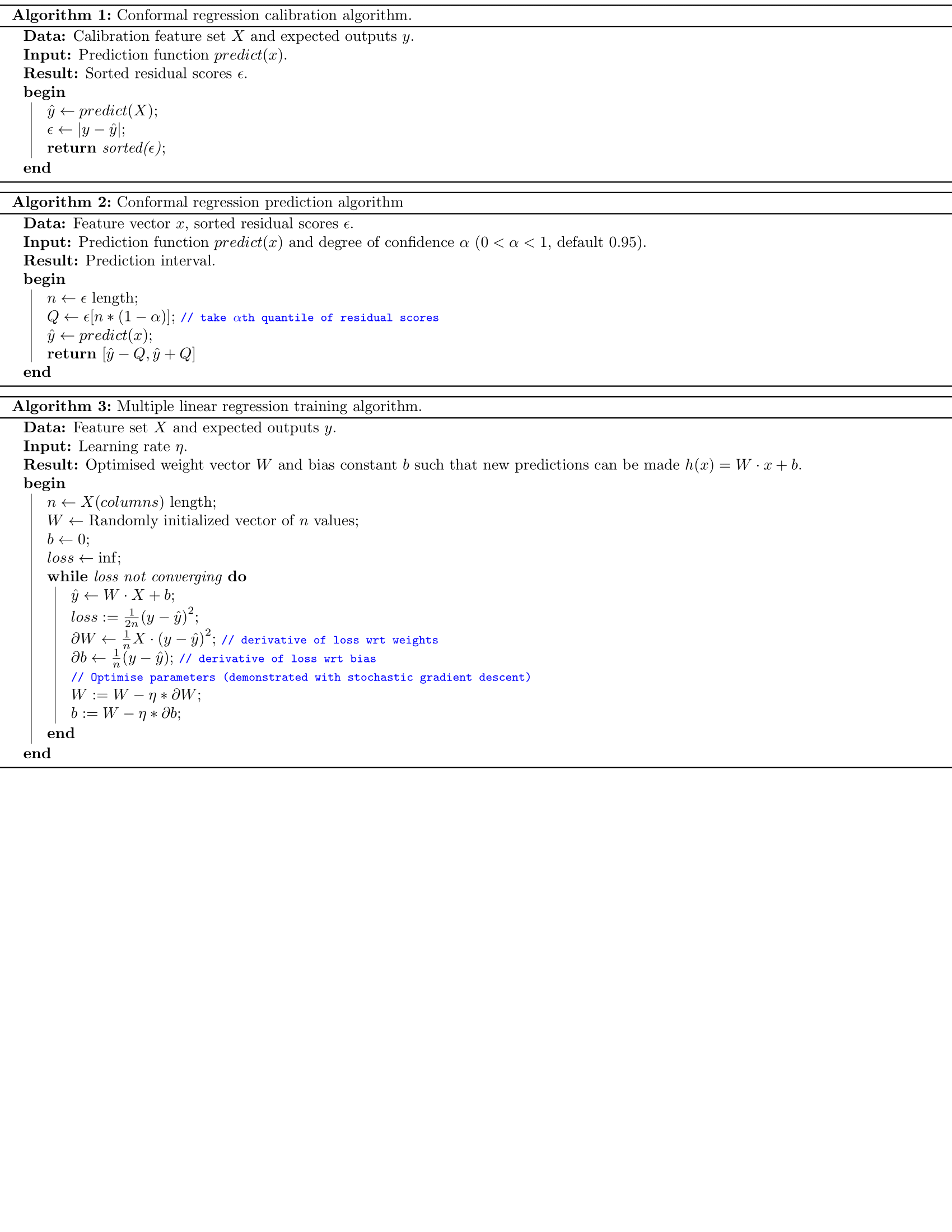


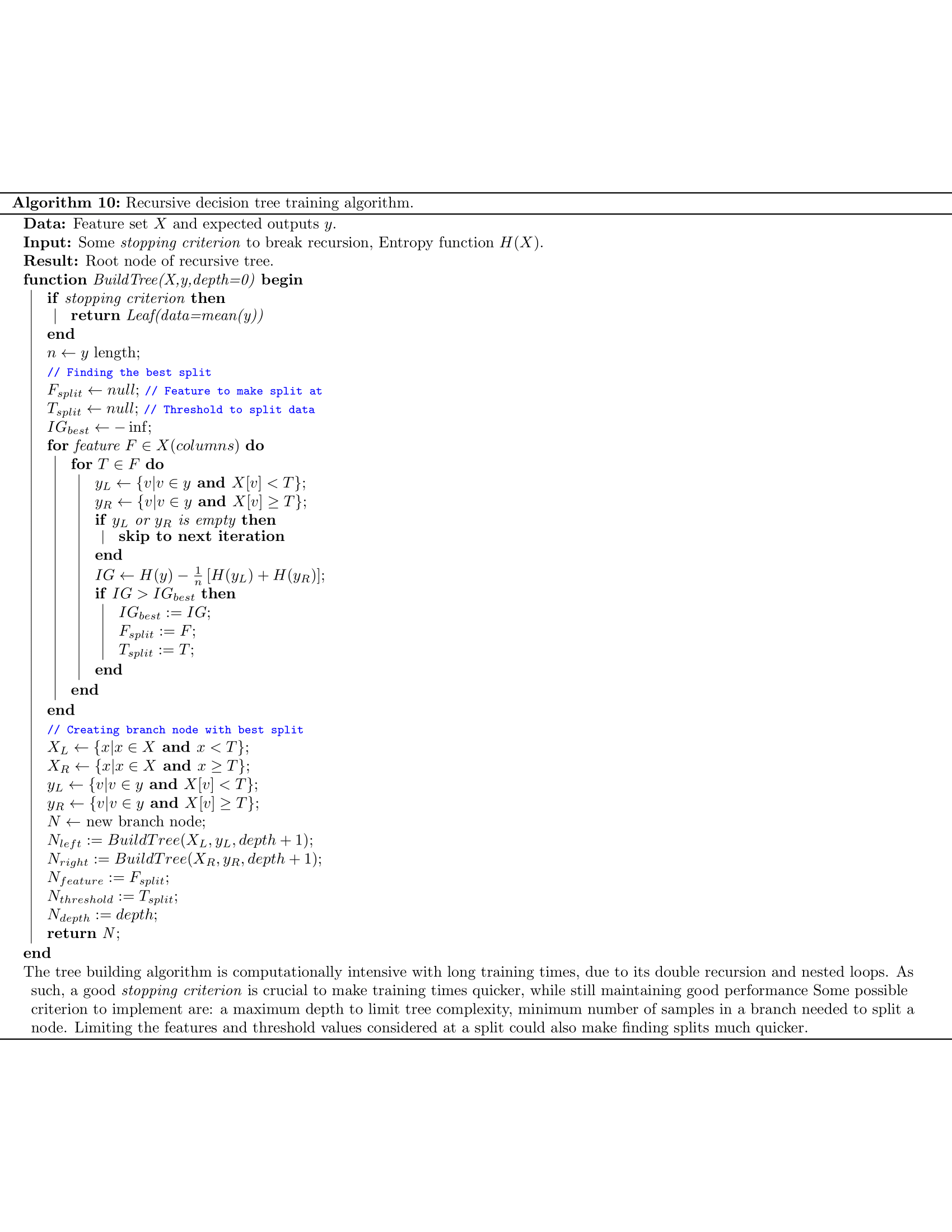
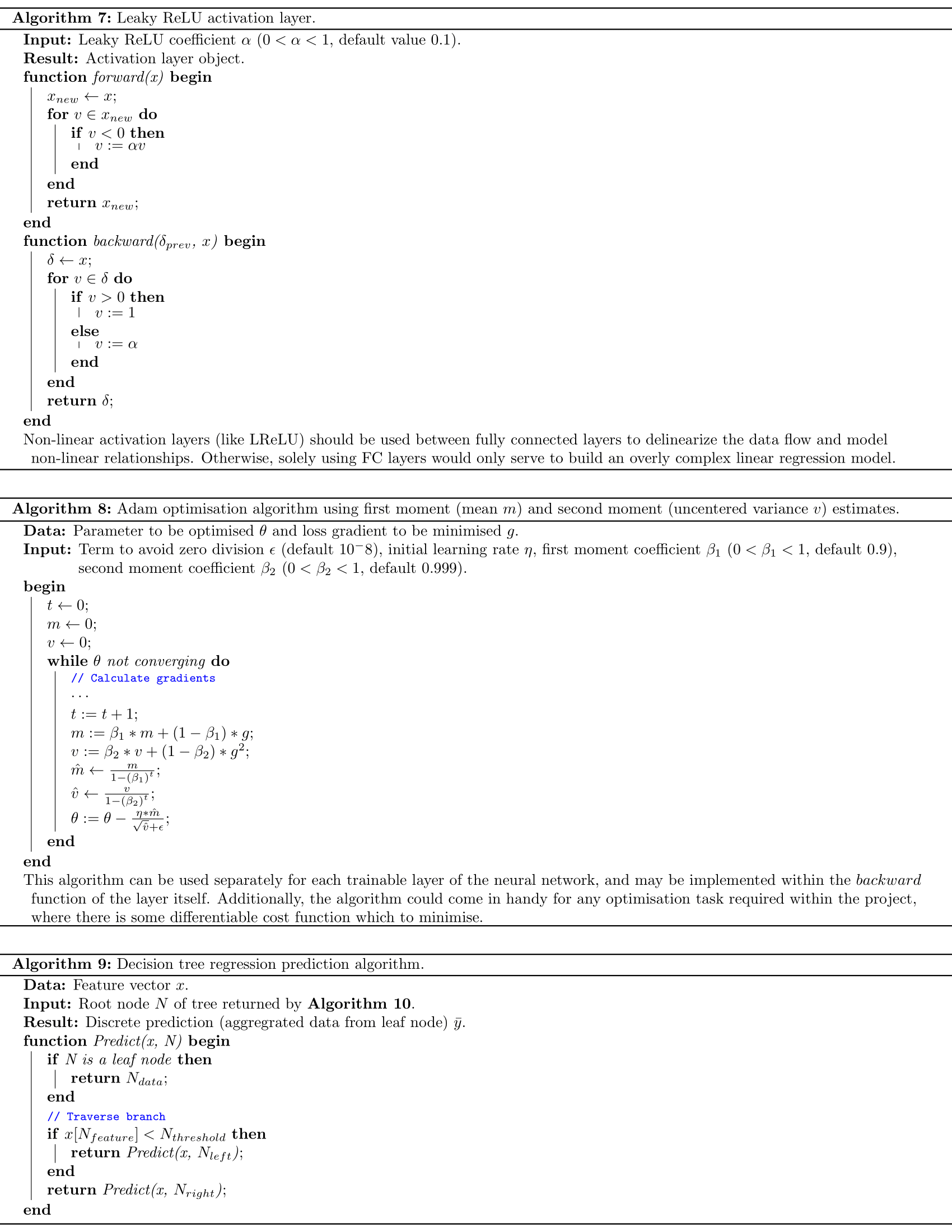
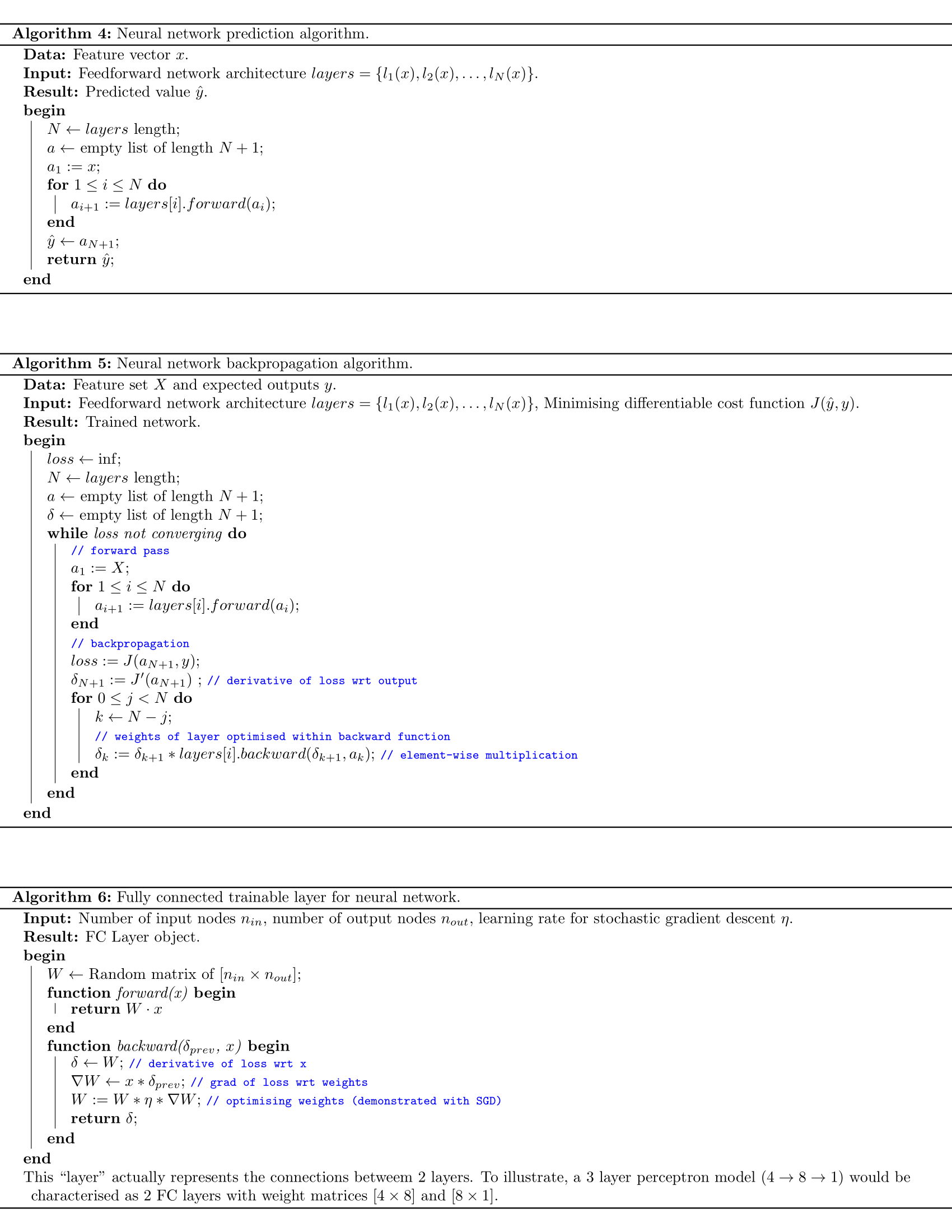
Figure 24 More detailed UML class diagram

## Algorithms and Pseudocode

I will be utilising many complex algorithms, including multiple linear regression, multilayer perceptron neural network, decision tree regression, k-nearest neighbours, conformal prediction etc. I will also outline some of my performance metrics used to evaluate these algorithms and explain how they are interpreted. Furthermore, I will outline some of the smaller algorithms such as outlier detection and data rescaling, used in the data preparation stage. Since my project is more research-based, most algorithms will be used in the pre-implementation stage.

### Pseudocode





## Data Structure

The data used to train the algorithms will be the result of data analysis and feature selection. I will be collecting and stitching together data from multiple sources to produce around 15 to 20 column features.

The data sources:

|  |  |  |
| --- | --- | --- |
| HM Land Registry data | Datatype | Example Data |
| Transaction UID | String | {AAB12C34-5678-9D12-…} |
| Price | Integer | 100000 |
| Date of transfer | Datetime | 1995-01-01 (YYYY-MM-DD) |
| Postcode | String | PR1 4HD |
| Property type | String (Categorical) | D (Detached) |
| Old/New | Boolean (Y/N) | Y (Newly built property) |
| Duration (tenure) | String (Categorical) | F (Freehold) |
| Primary addressable object name | String | 1 |
| Secondary addressable object Name | String or null | NULL |
| Street | String | Larkhill Rd |
| Locality | String | Frenchwood |
| Town/City | String | Preston |
| District | String | Preston |
| County | String | Lancashire |
| PPD Category Type | String (Categorical) | A (Standard Price Paid entry) |

|  |  |  |
| --- | --- | --- |
| Energy Performance Data (only relevant columns included) | Datatype | Example Data |
| Address Line 1 | String | 1 Larkhill Rd |
| Address Line 2 | String or null | NULL |
| Postcode | String | PR1 4HD |
| Property type | String (Categorical) | House |
| Built form | String (Categorical) | Enclosed Mid-Terrace |
| Total floor area | Numeric | 100.00 |
| Number of habitable rooms | Integer | 5 |
| Extension count | Integer | 0 |
| Glazed type | String (Categorical) | double glazing, unknown install date |
| Construction age band | String (Categorical) | England and Wales: before 1900 |
| Tenure | String (Categorical) | owner-occupied |

|  |  |  |
| --- | --- | --- |
| ONS Postcode Lookup Data (only relevant columns included) | Datatype | Example Data |
| Unit postcode | String | AB1 0AA |
| Decimal degrees latitude | Numeric | 99.999999 |
| Decimal degrees longitude | Numeric | 0.000000 |

|  |  |  |
| --- | --- | --- |
| UK House Price Index | Datatype | Example Data |
| Date | Datetime (DD/MM/YYYY) | 01/01/1995 |
| Region name | String | England and Wales |
| Area code | String | K04000001 |
| Index | Numeric | 26.44007 |

The above datasets will then be merged and stitched together, such that the resulting dataset is formatted as shown:

|  |  |  |
| --- | --- | --- |
| Final dataset | Datatype | Example Data |
| Date of transfer | Datetime (YYYY-DD-MM) | 1995-01-01 |
| Address | String | 1 Street Rd, AB1 0AA |
| Price | Integer | 16000 |
| Price (Inflation adjusted) | Numeric | 605.142… |
| Old/New | Boolean (Y/N) | N |
| Property type | String (Categorical) | House |
| Built form | String (Categorical) | Enclosed Mid-Terrace |
| Property type 2 | String (Categorical) | Mid-Terrace House |
| Total floor area | Numeric | 99.00 |
| Glazed type | String (Categorical) | double glazing |
| Extension count | Integer | 0 |
| Number of habitable rooms | Integer | 4 |
| Construction age band | String (Categorical) | England and Wales: before 1900 |
| Tenure | String (Categorical) | owner-occupied |
| Month | Integer (1-12) | 1 |
| Year | Integer (≥ 1995) | 1995 |
| House price index | Numeric | 26.440… |
| Latitude | Numeric | 51.503… |
| Longitude | Numeric | -0.127… |

After the process of cardinality reduction, category encoding, feature selection and rescaling, the final training dataset will be of around 5 column features. The dataset’s columns must have comparable distributions for predictive models to properly capture the underlying structure of the data. I will use standard scaling to achieve this.

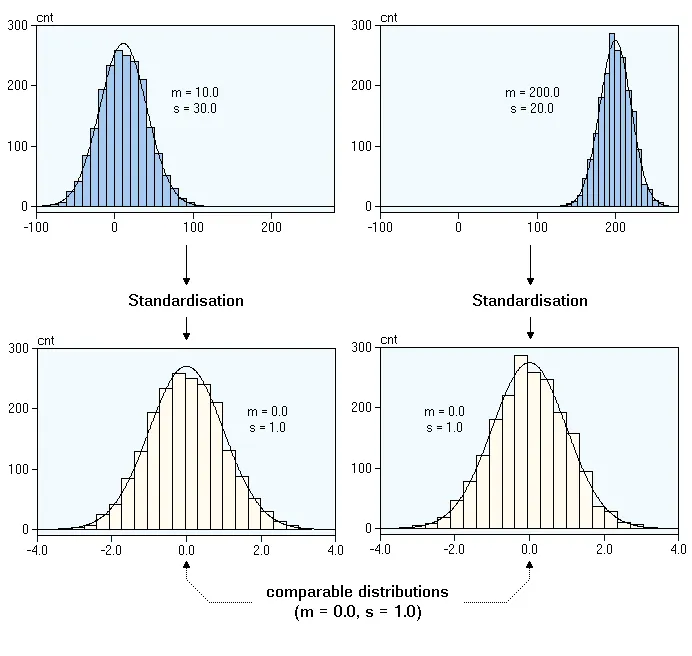


Figure 25 2 different distributions standardised for comparability.

The dataset will be large with ideally around 1 million+ rows, with enough variability such that the algorithms do not *overfit, i.e.,* cannot generalise the underlying structure of the dataset and instead “memorises” the observed data. In this case, the model will appear to perform well in training, but underperforms on new unobserved data. To avoid this, the Pandas dataframe (including both the features and dependent variable) will be converted into a NumPy matrix of rescaled data points. This data will then be split into training, calibration (for conformal prediction), and test (consisting of unobserved data) sets.

|  |  |
| --- | --- |
| Rescaled dataset | Datatype |
| Price (Inflation Adjusted + Rescaled) | Numeric columns with mean *µ*=0 and standard deviation σ=1. |
| Feature 1 |
| Feature 2 |
| … |
| Feature N |

To avoid data leakage, the mean *µ* and standard deviation *σ* scaling factors will only be computed using the training data and each set will be scaled accordingly. Furthermore, the scaling factors will be computed column-wise, and will result into vectors *µ and σ* of length equal to the number of columns.

The datasets will then have their label column removed such that there will be six NumPy 2D numeric arrays:

|  |  |  |
| --- | --- | --- |
| Array | Data shape | Purpose |
| Training X |  | Rescaled training features to fit predictive models |
| Training y |  | Expected outputs for training features used to fit models. |
| Calibration X |  | Rescaled calibration features to compute residual scores for conformal regression |
| Calibration y |  | Expected outputs for calibration features |
| Validation X |  | Rescaled features to test performance of predictive model |
| Validation y |  | Expected outputs for validation features used for testing |

Where,

N: Number of rows

*f*: Number of feature columns

## User interfaces

The app itself will be simple, consisting of only two screens; a landing page with a form, and an output screen displaying the prediction calculated by the model. Despite the simplicity, I would like the webapp to be aesthetically pleasing, with subtle micro-animations and page transitions to achieve a premium finish. Additionally, I will display the output as a modal window that overlays on top of the main page, creating a seamless experience for the end user.

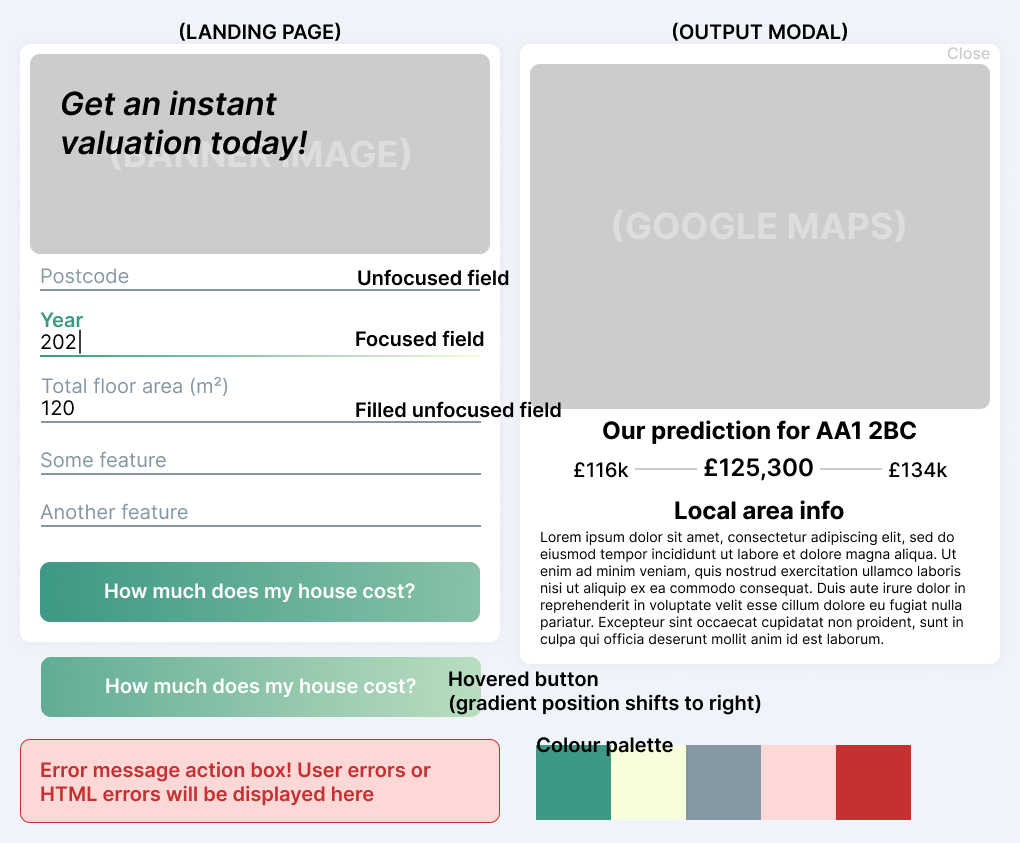


Figure 26 Landing page and output modal design

# Implementation

## File Structure

base/

stage-1-research/

algorithms/

conformal\_regression/

\_\_init\_\_.py

decision\_tree/

\_\_init\_\_.py

base.py

node.py

knn/

\_\_init\_\_.py

linear\_regression/

\_\_init\_\_.py

mlp/

base.py

layers.py

model.py

optimisers.py

data/

raw/

all-domestic-certificates/

...

{LSAO}-{city}/

certificates.csv

recommendations.csv

postcode-lookup.csv

house-price-index.csv

parts/

pricepaid/...

epc/...

merged/...

dataset.parquet

models/

model1.bin

model2.bin

...

model{n}.bin

p1-data-sourcing.ipynb

p2-data-exploration.ipynb

p3-model-selection.ipynb

stage2-app/

static/

css/

styles.css

sass/

styles.sass

templates/

index.jinja

model/

algorithms/

{best model}/... *(folder copied from base/stage-1-research/algorithms/{name})*

conformal.py

data/

conf\_resid.bin

encoding.json

model.bin *(file copied from base/stage-1-research/models/model{n}.bin)*

scaling.json

\_\_init\_\_.py

predictor.py

app.yaml

main.py

requirements.txt

utils.py

## Pre-implementation/Design – Data sourcing

*# base/stage-1-research/p1-data-sourcing.ipynb*

I will begin the implementation by collecting data from multiple sources and stitching them together in one informative dataset. I will use open data from the HM Land Registry and various other public sources.

**import** os**,** shutil**,** zipfile

**from** requests **import** Session

**from** tqdm **import** tqdm

**import** numpy **as** np

**import** pandas **as** pd

**import** dask.dataframe **as** dd

**from** dask.diagnostics **import** ProgressBar

### Price paid data

The [***HM Land Registry price paid data***](https://www.gov.uk/guidance/about-the-price-paid-data) tracks the property sales in England and Wales submitted to HM Land Registry for registration, based on the raw data released every month, with data from 1995 to now (January 2022).

Column headers:

* Price
* Date of Transfer
* Postcode
* Property Type: *D* (Detached), *S* (Semi-Detached), *T* (Terraced), *F* (Flats/Maisonettes), *O* (Other)
* Old/New: *Y* (New), *N* (Old)
* Duration – Tenure of property: F (Freehold), L (Leasehold), etc.
* PAON - Primary Addressable Object Name (e.g., house number)
* SAON - Secondary Addressable Object Name (e.g., unit number)
* Street
* Locality
* Town/City
* District
* PPD Category Type

Initially, I will take a small chunk of the first fifty thousand rows to test and apply transformations before applying them to the full dataset. I will be adjusting the dataset to minimise its memory usage by dropping columns and converting data types.

ppdUrl **=** "http://prod.publicdata.landregistry.gov.uk.s3-website-eu-west-1.amazonaws.com/pp-complete.csv"

ppdNames **=** [

"UID", "PRICE", "DATE\_OF\_TRANSFER", "POSTCODE", "PROPERTY\_TYPE", "OLD\_NEW",

"DURATION", "PAON", "SAON", "STREET", "LOCALITY", "TOWN\_CITY", "DISTRICT",

"COUNTY", "PPD\_CAT", "RECORD\_STATUS"

]

ppdSample **=** dd**.**read\_csv(ppdUrl, names**=**ppdNames, dtype**=**object)**.**head(n**=**50\_000)

Sample memory: 50.77MB

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **UID** | **PRICE** | **DATE\_OF\_TRANSFER** | **POSTCODE** | **PRP\_TYPE** | **OLD\_NEW** | **DU...** | **PAON** | **SAON** | **STREET** | **LOCALITY** | **TOWN\_CITY** | **DISTRICT** | **COUNTY** | **PPD\_CAT** | **RECORD\_STATUS** |
| **0** | {5BBE...} | 42000 | 1995-12-21 00:00 | NE4 9DN | S | N | F | 8 | NaN | MATFEN | FENHAM | NEWCASTLE UPON TYNE | NEWCASTLE UPON TYNE | TYNE AND WEAR | A | A |
| **1** | {20E2...} | 95000 | 1995-03-03 00:00 | RM16 4UR | S | N | F | 30 | NaN | HEA... | GRAYS | GRAYS | THURROCK | THURROCK | A | A |
| **2** | {D893...} | 74950 | 1995-10-03 00:00 | CW10 9ES | D | Y | F | 15 | NaN | SHR... | MIDDL... | MIDDLEWICH | CONGLETON | CHESHIRE | A | A |
| **3** | {F9F7...} | 43500 | 1995-11-14 00:00 | TS23 3LA | S | N | F | 19 | NaN | SLE... | BILLI... | BILLINGHAM | STOCKTON-ON-TEES | STOCKTON-ON-TEES | A | A |
| **4** | {E166...} | 63000 | 1995-09-08 00:00 | CA25 5QH | S | N | F | 8 | NaN | CRO... | CLEAT... | CLEATOR MOOR | COPELAND | CUMBRIA | A | A |

The only important columns in the dataset are the price and dates, the location data is not particularly helpful, and I will be substituting them for coordinates instead. However, I will keep other columns which could be helpful as features.

ppdFilter **=** [

"PRICE",

"DATE\_OF\_TRANSFER",

"POSTCODE",

"OLD\_NEW",

"PAON", *#\**

"SAON", *#\**

"STREET" *#\**

]

*#\* WILL LATER BE DROPPED*

ppdSample **=** ppdSample[ppdFilter] *# filtering columns*

*# printing number of NaNs in columns*

print("NaNs in column:\n", ppdSample**.**isna()**.**sum()[**lambda** x: x **>** 0])

NaNs in column:

POSTCODE 36

PAON 5

SAON 46280

STREET 750

*# removing rows with NaN in any column (ignoring SAON column)*

ppdSample **=** (ppdSample

**.**dropna(subset**=**list(filter(**lambda** x: x **!=** "SAON", ppdFilter)))

**.**reset\_index(drop**=True**))

I will now convert the data types of the remaining columns to more manipulatable and less memory intensive datatypes.

ppdSample["PRICE"] **=** ppdSample**.**PRICE**.**astype("float") *# min max ~2bn | float32 - 4 bytes*

ppdSample["DATE\_OF\_TRANSFER"] **=** dd**.**to\_datetime(ppdSample**.**DATE\_OF\_TRANSFER) *# Converting from string to datetime*

ppdSample["OLD\_NEW"] **=** ppdSample**.**OLD\_NEW**.**astype("category") *# converting to categorical*

Now, I will merge PAON, SAON, STREET, and POSTCODE columns into one ADDRESS column in the format “PAON SAON STREET POSTCODE.” I will apply the same formatting to the addresses in the EPC dataset to allow cross-referencing and merging the datasets into one.

**def** **ppdFormatAddr**(x):

**return** [

' '**.**join(z**.**strip() **for** z **in** y **if** isinstance(z, str)) *# joining address columns*

**for** y **in** x[["SAON", "PAON", "STREET", "POSTCODE"]]**.**values

] *# List comprehension for performance*

ppdSample["ADDRESS"] **=** ppdFormatAddr(ppdSample)

*# Keeping POSTCODE column for geolocation purposes*

ppdSample **=** ppdSample**.**drop(columns**=**["PAON", "SAON", "STREET"])

0 8 MATFEN PLACE NE4 9DN

1 30 HEATH ROAD RM16 4UR

2 15 SHROPSHIRE CLOSE CW10 9ES

3 19 SLEDMERE CLOSE TS23 3LA

4 8 CROSSINGS CLOSE CA25 5QH

Name: ADDRESS, dtype: object

Processed sample set: 50.77MB => 8.09MB

|  | **dtype** | **memory\_usage** | **null** | **unique** |
| --- | --- | --- | --- | --- |
| **PRICE** | float64 | 393736 | 0.0 | 2900.0 |
| **DATE\_OF\_TRANSFER** | datetime64[ns] | 393736 | 0.0 | 356.0 |
| **POSTCODE** | object | 3171295 | 0.0 | 45548.0 |
| **OLD\_NEW** | category | 49441 | 0.0 | 2.0 |
| **ADDRESS** | object | 4085965 | 0.0 | 49158.0 |
| **Index** | NaN | 128 | NaN | NaN |

The sample set is now in an appropriate format; therefore, I will apply the transformations to the entire dataset. I will process and export the dataset using Dask, which uses parallelisation across multiple threads to enhance performance.

Processing pipeline:

* Import the entire dataset with prefiltered columns and correct datatypes
* Drop rows with empty entries (ignoring the sparse SAON column)
* Format addresses in each partition with ppdFormatAddr function and set as index
* Drop duplicate addresses, keeping the most recent entry
* Drop obsolete columns
* Store dataset into external storage

*# data url*

ppdUrl **=** "http://prod.publicdata.landregistry.gov.uk.s3-website-eu-west-1.amazonaws.com/pp-complete.csv"

*# all column names*

ppdNames **=** [

"UID", "PRICE", "DATE\_OF\_TRANSFER", "POSTCODE", "PROPERTY\_TYPE", "OLD\_NEW",

"DURATION", "PAON", "SAON", "STREET", "LOCALITY", "TOWN\_CITY", "DISTRICT",

"COUNTY", "PPD\_CAT", "RECORD\_STATUS"

]

*# columns to filter*

ppdFilter **=** ["PRICE", "DATE\_OF\_TRANSFER", "POSTCODE", "OLD\_NEW", "PAON", "SAON", "STREET"]

*# column datatypes*

ppdDtype **=** {

"PRICE": np**.**float32,

"DATE\_OF\_TRANSFER": object,

"POSTCODE": object,

"OLD\_NEW": pd**.**CategoricalDtype(),

"PAON": object,

"SAON": object,

"STREET": object

}

*# format function*

**def** **ppdFormatAddr**(x):

**return** [

' '**.**join(z**.**strip() **for** z **in** y **if** isinstance(z, str))

**for** y **in** x[["SAON", "PAON", "STREET", "POSTCODE"]]**.**values

] *# List comprehension for performance*

*# processing pipeline*

ppdPipeline **=** (dd

**.**read\_csv(

ppdUrl,

names**=**ppdNames,

usecols**=**ppdFilter,

parse\_dates**=**["DATE\_OF\_TRANSFER"],

dtype**=**ppdDtype

) *# read data*

**.**dropna(subset**=**filter(**lambda** x: x **!=** "SAON", ppdFilter)) *# drop NaN values*

**.**map\_partitions(**lambda** x: x**.**assign(ADDRESS**=**ppdFormatAddr)) *# format addresses*

**.**drop\_duplicates(subset**=**"ADDRESS",keep**=**"last") *# drop duplicate addresses*

**.**map\_partitions(**lambda** x: x**.**set\_index("ADDRESS")) *# set index to addresses*

**.**drop(columns**=**["SAON", "PAON", "STREET"]) *# drop unnecessary columns*

)

*# create dataset if it does not already exist*

*# PPD\_PATH: base/stage1-research/data/parts/pricepaid*

*# MERGED\_PATH: base/stage1-research/data/parts/merged*

**if** **not** os**.**path**.**exists(PPD\_PATH) **and** **not** os**.**path**.**exists(MERGED\_PATH):

**with** ProgressBar():

ppdPipeline**.**to\_parquet(PPD\_PATH, compression**=**"snappy")

ppd **=** dd**.**read\_parquet(PPD\_PATH)

Final dataset:

2.46GB

14,876,050 rows

|  | **PRICE** | **DATE\_OF\_TRANSFER** | **POSTCODE** | **OLD\_NEW** |
| --- | --- | --- | --- | --- |
| **ADDRESS** |  |  |  |  |
| **87 THEOBALD ROAD NR1 2NX** | 37500.0 | 1995-06-30 | NR1 2NX | N |
| **16 AFON RHOS ESTATE LL55 4SE** | 44500.0 | 1995-11-22 | LL55 4SE | N |
| **69 CRANHILL ROAD BA16 0BZ** | 48500.0 | 1995-04-13 | BA16 0BZ | N |
| **4 BRIARY LANE SG8 9BZ** | 53000.0 | 1995-01-06 | SG8 9BZ | N |
| **133 WOOD LANE CH5 3JF** | 67500.0 | 1995-12-12 | CH5 3JF | N |

The final dataset is more than ideal, with around 15 million rows of unique data points; this will allow for some leeway when inner joining with the EPC dataset, which may not contain all the same addresses, potentially significantly reducing the number of datapoints.

### Energy Performance of Buildings data

[Energy Performance of Buildings Data](https://epc.opendatacommunities.org/) provides access to Energy Performance Certificates and Display Energy Certificate data for buildings across England and Wales, based on data released quarterly from 1 October 2008 up to 30 September 2021 (most recent release as of development). This data must be downloaded manually as it requires authentication to access the site. Furthermore, the data is huge, requiring upwards of 30GB in memory, so I will download manually for added peace of mind in case something goes wrong, as it always does.

I will only be requiring the Energy Performance Certificate data, containing 92 column headers including:

* Address headers (Address lines 1 to 3)
* Postcode
* Number of habitable rooms
* Total square footage
* Property type

I will be following a similar workflow as processing the Price Paid data. I will begin by using the first certificate file as a sample set and filter out most unnecessary columns. I will then convert the data types of the remaining columns.

**from** glob **import** glob

epcPath **=** os**.**path**.**join(RAW\_PATH, "all-domestic-certificates", "\*", "certificates.csv")

epcSample **=** pd**.**read\_csv(glob(epcPath)[0], low\_memory**=False**)

Sample memory: 207.64MB

|  | **LMK\_KEY** | **ADDRESS1** | **ADDRESS2** | **ADDRESS3** | **...** | **POSTCODE** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | 20894... | 15, Elm... | NaN | NaN |  | GL51 9JJ |
| **1** | 12960... | 15, Lim... | Prestbury | NaN |  | GL52 3EF |
| **2** | 11028... | 44, Nau... | NaN | NaN |  | GL53 7BH |
| **3** | 23833... | Flat 8,... | 24, Eldorado Road | NaN |  | GL50 2PT |
| **4** | 16564... | 6, Char... | NaN | NaN |  | GL51 9HH |

As is stands there are far too many unnecessary columns; I will filter the dataset to contain columns I believe to be potentially useful as features for predictive models. Additionally, I will replace placeholder values such as "NO DATA!" to NaN values and subsequently drop them.

epcFilter **=** [

"ADDRESS1", *#\**

"ADDRESS2", *#\**

"POSTCODE", *#\**

"PROPERTY\_TYPE",

"BUILT\_FORM",

"TOTAL\_FLOOR\_AREA",

"NUMBER\_HABITABLE\_ROOMS",

"EXTENSION\_COUNT",

"GLAZED\_TYPE",

"CONSTRUCTION\_AGE\_BAND",

"TENURE"

]

*#\* WILL LATER BE DROPPED*

epcSample **=** epcSample[epcFilter] *# filtering columns*

*# values to be converted to NaN*

nanValues **=** (

"NULL",

"INVALID",

"INVALID!",

"NODATA!",

"NO DATA!",

"N/A",

"Not applicable",

"Not recorded",

"not defined",

"Blank"

)

epcSample[epcSample**.**isin(nanValues)] **=** np**.**nan *# mapping values to NaN*

*# printing number of NaNs in columns*

print("NaNs in column:\n", epcSample**.**isna()**.**sum()[**lambda** x: x **>** 0])

NaNs in column:

ADDRESS2 25366

BUILT\_FORM 1141

NUMBER\_HABITABLE\_ROOMS 4471

EXTENSION\_COUNT 4471

GLAZED\_TYPE 9938

CONSTRUCTION\_AGE\_BAND 4460

TENURE 1432

*# dropping rows with any NaN values (ignoring ADDRESS2 column)*

epcSample **=** (epcSample

**.**dropna(subset**=**list(filter(**lambda** x: x **!=** "ADDRESS2", epcFilter)))

**.**reset\_index(drop**=True**))

*# converting datatypes*

*# float columns*

float\_col **=** ["TOTAL\_FLOOR\_AREA", "NUMBER\_HABITABLE\_ROOMS", "EXTENSION\_COUNT"]

*# categorical columns*

cat\_col **=** [

"PROPERTY\_TYPE", "BUILT\_FORM", "GLAZED\_TYPE", "CONSTRUCTION\_AGE\_BAND",

"TENURE"

]

epcSample[float\_col] **=** epcSample[float\_col]**.**astype("float") *# converting columns to float*

epcSample[cat\_col] **=** epcSample[cat\_col]**.**astype("category") *# converting columns to category*

I will now format the address columns as “ADDRESS1 ADDRESS2 POSTCODE” in uppercase, with commas removed from text. This will be in the same “SAON PAON STREET POSTCODE” format as the price paid data, allowing them to link together via the address key.

**def** **epcFormatAddr**(x):

**return** [

' '**.**join(z**.**strip()**.**replace(",", "")**.**upper() **for** z **in** y

**if** **not** pd**.**isna(z))

**for** y **in** x[["ADDRESS1", "ADDRESS2", "POSTCODE"]]**.**values

] *# List comprehension for performance*

epcSample["ADDRESS"] **=** epcFormatAddr(epcSample) *# formatting addresses*

epcSample **=** epcSample**.**drop(columns**=**["ADDRESS1", "ADDRESS2", "POSTCODE"]) *# dropping unnecessary columns*

0 15 ELMFIELD ROAD GL51 9JJ

1 15 LIME CLOSE PRESTBURY GL52 3EF

2 44 NAUNTON LANE GL53 7BH

3 6 CHARLES STREET GL51 9HH

4 25 BELWORTH COURT GL51 6HQ

Name: ADDRESS, dtype: object

Processed sample set: 207.64MB => 4.50MB

|  | **dtype** | **memory\_usage** | **null** | **unique** |
| --- | --- | --- | --- | --- |
| **PROPERTY\_TYPE** | category | 38719 | 0.0 | 5.0 |
| **BUILT\_FORM** | category | 38823 | 0.0 | 6.0 |
| **TOTAL\_FLOOR\_AREA** | float64 | 305808 | 0.0 | 7559.0 |
| **NUMBER\_HABITABLE\_ROOMS** | float64 | 305808 | 0.0 | 21.0 |
| **EXTENSION\_COUNT** | float64 | 305808 | 0.0 | 5.0 |
| **GLAZED\_TYPE** | category | 39180 | 0.0 | 8.0 |
| **CONSTRUCTION\_AGE\_BAND** | category | 39895 | 0.0 | 13.0 |
| **TENURE** | category | 39179 | 0.0 | 8.0 |
| **ADDRESS** | object | 3390932 | 0.0 | 30564.0 |
| **Index** | NaN | 128 | NaN | NaN |

Now that the sample set has been transformed, these manipulations can be translated to the entire dataset using a process almost identical to the Price paid dataset pipeline.

Processing pipeline:

* Import all files as a *glob* using Dask with prefiltered columns and correct datatypes.
* Drop rows with empty entries (ignoring ADDRESS2 column)
* Format addresses in each partition with epcFormatAddr function and set as index
* Drop duplicate addresses, keeping the most recent entry
* Drop obsolete columns
* Store dataset into external storage

*# path to energy performance certificates as a glob*

epcPath **=** os**.**path**.**join(RAW\_PATH, "all-domestic-certificates", "\*", "certificates.csv")

*# columns to filter*

epcFilter **=** [

"ADDRESS1", "ADDRESS2", "POSTCODE", "PROPERTY\_TYPE", "BUILT\_FORM",

"TOTAL\_FLOOR\_AREA", "NUMBER\_HABITABLE\_ROOMS", "EXTENSION\_COUNT",

"GLAZED\_TYPE", "CONSTRUCTION\_AGE\_BAND", "TENURE"

]

*# values to convert to NaN*

nanValues **=** [

"NULL",

"INVALID",

"INVALID!",

"NODATA!",

"NO DATA!",

"N/A",

"Not applicable",

"Not recorded",

"not defined",

"Blank",

]`

*# column datatypes*

epcDtype **=** {

'ADDRESS1': object,

'ADDRESS2': object,

'POSTCODE': object,

'PROPERTY\_TYPE': pd**.**CategoricalDtype(),

'BUILT\_FORM': pd**.**CategoricalDtype(),

'TOTAL\_FLOOR\_AREA': np**.**float64,

'NUMBER\_HABITABLE\_ROOMS': np**.**float64,

'EXTENSION\_COUNT': np**.**float64,

'GLAZED\_TYPE': pd**.**CategoricalDtype(),

'CONSTRUCTION\_AGE\_BAND': pd**.**CategoricalDtype(),

'TENURE': pd**.**CategoricalDtype()

}

*# formatting function*

**def** **epcFormatAddr**(x):

**return** [

' '**.**join(z**.**strip()**.**replace(",", "")**.**upper() **for** z **in** y

**if** **not** pd**.**isna(z))

**for** y **in** x[["ADDRESS1", "ADDRESS2", "POSTCODE"]]**.**values

] *# List comprehension for performance*

*# processing pipeline*

epcPipeline **=** (dd

**.**read\_csv(epcPath, usecols**=**epcFilter, na\_values**=**nanValues, dtype**=**epcDtype) *# read data from folder*

**.**dropna(subset**=**list(filter(**lambda** x: x **!=** "ADDRESS2", epcFilter)), how**=**"any") *# drop NaNs (ignoring sparse ADDRESS2 column)*

**.**map\_partitions(**lambda** x: x**.**assign(ADDRESS**=**epcFormatAddr))**.**drop\_duplicates(subset**=**"ADDRESS",keep**=**"last") *# drop duplicate addresses*

**.**map\_partitions(**lambda** x: x**.**set\_index("ADDRESS")) *# set index to addresses*

**.**drop(columns**=**["ADDRESS1", "ADDRESS2", "POSTCODE"]) *# drop obsolete columns*

)

*# create dataset if it does not already exist*

*# PPD\_PATH: base/stage1-research/data/parts/pricepaid*

*# MERGED\_PATH: base/stage1-research/data/parts/merged*

**if** **not** os**.**path**.**exists(EPC\_PATH) **and** **not** os**.**path**.**exists(MERGED\_PATH):

**with** ProgressBar():

epcPipeline**.**to\_parquet(EPC\_PATH, compression**=**"snappy")

epc **=** dd**.**read\_parquet(EPC\_PATH)

Final dataset:

1.64GB

14,148,102 rows

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **PRP\_TYPE** | **BUILT\_FORM** | **AREA** | **GLAZED\_TYPE** | **EXT\_COUNT** | **N\_ROOMS** | **AGE\_BAND** | **TENURE** |
| **ADDRESS** |  |  |  |  |  |  |  |  |
| **54 C...** | Bungalow | Semi-D... | 59.0 | double g... | 0.0 | 4.0 | 1967-1975 | rent... |
| **18 T...** | Bungalow | Semi-D... | 70.8 | double g... | 0.0 | 4.0 | 1950-1966 | rent... |
| **3 HA...** | House | Mid-T... | 86.0 | double g... | 1.0 | 5.0 | 1950-1966 | owne... |
| **31 L...** | Flat | Detac... | 52.0 | double g... | 0.0 | 3.0 | 2007 o... | owne... |
| **16 H...** | House | Detac... | 121.0 | double g... | 1.0 | 6.0 | 1991-1995 | owne... |

### Merging datasets

Now that both datasets have been prepared, I can now combine them to create one large dataset using an inner join on the address column.

*# merging pipeline*

merge **=** (dd

**.**merge(ppd, epc, how**=**"inner", left\_index**=True**, right\_index**=True**) *# intersection of price-paid and certificate data*

**.**reset\_index(drop**=True**) *# dropping address index as it is no longer necessary*

**.**map\_partitions(**lambda** x: x**.**set\_index("DATE\_OF\_TRANSFER")**.**sort\_index()) *# set date to new index*

)

*# creating merged dataset if it does not already exist*

*# MERGED\_PATH: base/stage1-research/data/parts/merged*

**if** **not** os**.**path**.**exists(MERGED\_PATH):

**with** ProgressBar():

merge**.**to\_parquet(MERGED\_PATH, schema**=**"infer", compression**=**"snappy")

df **=** dd**.**read\_parquet(MERGED\_PATH)**.**compute()

Merged dataset:

614.61MB

5,566,962 rows

|  | **PRICE** | **PCD** | **OLD\_NEW** | **PR...** | **BUILT\_FORM** | **AREA** | **GLAZ...** | **E\_COUNT** | **N\_ROOMS** | **AGE\_BAND** | **TENURE** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **DATE\_OF...** |  |  |  |  |  |  |  |  |  |  |  |
| **1995-01-01** | 16000.0 | HX1... | N | House | Enclo... | 99.00 | dou... | 0.0 | 4.0 | Befo... | owne... |
| **1995-01-02** | 35000.0 | NN1... | N | House | Mid-T... | 85.00 | dou... | 1.0 | 4.0 | 1900-1929 | owne... |
| **1995-01-03** | 15000.0 | NE34... | N | House | Semi-D... | 74.00 | dou... | 0.0 | 4.0 | 1967-1975 | rent... |
| **1995-01-03** | 48000.0 | HR8... | N | Bu... | Detac... | 78.00 | dou... | 1.0 | 3.0 | 1950-1966 | owne... |
| **1995-01-03** | 82000.0 | TR13... | N | Bu... | Detac... | 118.88 | dou… | 0.0 | 4.0 | 1976-1975 | rent... |

The new dataset is now far smaller than the individual parts, I anticipated this and am still happy with the result it produced. Since the data is much smaller, data manipulation will be much quicker and more efficient. Therefore, I can now switch from using the Dask library in favour of the standard Pandas library by computing the Dask Dataframe into memory.

### Downloading additional data for feature engineering

During my dataset analysis, I will require additional data to further transform my dataset.

This includes:

* Geographical data from the [National Statistics Postcode Lookup (February 2022)](https://geoportal.statistics.gov.uk/datasets/national-statistics-postcode-lookup-february-2022-1/about), e.g., longitude, latitude
* HPI (House Price Index) data from [National Statistics UK House Price Index (January 2022)](https://www.gov.uk/government/publications/about-the-uk-house-price-index)

from tqdm import tqdm

**class** **Downloader**:

**def** \_\_init\_\_(

self,

\_session,

pbar\_enabled: bool **=** **False**,

pbar\_kwargs**=**{},

):

self**.**session **=** \_session

self**.**pbar\_enabled **=** pbar\_enabled

self**.**pbar\_kwargs **=** pbar\_kwargs

**def** **dlFile**(self, file, chunk\_size**=**1024, **\*\***reqKwargs) **->** **None**:

r **=** self**.**session**.**get(

**\*\***reqKwargs, stream**=True**) *# sending get request to provided url*

r**.**raise\_for\_status() *# raise error if request failure*

**if** self**.**pbar\_enabled:

pbar **=** tqdm(desc**=**file,

total**=**int(r**.**headers**.**get("content-length")),

unit**=**"iB",

unit\_scale**=True**,

**\*\***self**.**pbar\_kwargs) *# Initialising progress bar*

**with** open(file, "wb") **as** f:

*# Iterating chunks as they come from server*

**for** chunk **in** r**.**iter\_content(chunk\_size**=**chunk\_size):

f**.**write(chunk) *# writing chunk to file*

f**.**flush() *# flushing input buffer*

**if** self**.**pbar\_enabled:

pbar**.**update(len(chunk)) *# Updating progress bar by size of chunk*

**if** self**.**pbar\_enabled: pbar**.**close()

#### Postcode Lookup

**if** **not** os**.**path**.**exists(PCD\_PATH): *# base/stage1-research/data/raw/postcode-lookup.csv*

url **=** "https://www.arcgis.com/sharing/rest/content/items/1a0444ee3c43452ea16c530966ae8984/data"

extractPath **=** PCD\_PATH**.**removesuffix(".csv")

**if** **not** os**.**path**.**exists(extractPath):

zipPath **=** extractPath **+** ".zip"

*# downloading zip file from website*

**with** Session() **as** sess:

Downloader(sess, pbar\_enabled**=True**)**.**dlFile(zipPath, url**=**url)

*# extracting zip to folder*

**with** zipfile**.**ZipFile(zipPath, "r") **as** handle:

handle**.**extractall(extractPath)

os**.**remove(zipPath) *# removing zip file*

*# taking necessary columns and writing into csv file*

(pd

**.**read\_csv(os**.**path**.**join(extractPath, "Data", "NSPL\_FEB\_2022\_UK.csv"), usecols**=**["pcds", "lat", "long"]) *# only taking relevant columns*

**.**rename(columns**=**{"long":"lng"})

**.**rename(columns**=lambda** x: x**.**upper())

**.**to\_csv(PCD\_PATH, index**=False**) *# base/stage1-research/data/raw/postcode-lookup.csv*

)

shutil**.**rmtree(extractPath) *# removing folder*

geo **=** pd**.**read\_csv(PCD\_PATH) *# base/stage1-research/data/raw/postcode-lookup.csv*

Postcode Lookup data:

214.96MB

2,673,018 rows

|  | **PCDS** | **LAT** | **LNG** |
| --- | --- | --- | --- |
| **0** | AB1 0AA | 57.101474 | -2.242851 |
| **1** | AB1 0AB | 57.102554 | -2.246308 |
| **2** | AB1 0AD | 57.100556 | -2.248342 |
| **3** | AB1 0AE | 57.084444 | -2.255708 |
| **4** | AB1 0AF | 57.096656 | -2.258102 |

#### House Price Index

**if** **not** os**.**path**.**exists(HPI\_PATH): *# base/stage1-research/data/raw/house-price-index.csv*

url **=** "http://publicdata.landregistry.gov.uk/market-trend-data/house-price-index-data/Indices-2022-02.csv"

*# Parsing file and writing to csv file*

(pd**.**read\_csv(url, usecols**=**["Date", "Region\_Name", "Index"],parse\_dates**=**["Date"])

**.**query("Region\_Name == 'England and Wales'", engine**=**"python") *# entire E+W index instead of individual counties etc.*

**.**set\_index("Date")**.**sort\_index()

**.**Index**.**to\_csv(HPI\_PATH) *# base/stage1-research/data/raw/house-price-index.csv*

)

hpi **=** pd**.**read\_csv(HPI\_PATH) *# base/stage1-research/data/raw/house-price-index.csv*

House Price Index data:

24.58KB

326 rows

|  | **Date** | **Index** |
| --- | --- | --- |
| **0** | 1995-01-01 | 26.440075 |
| **1** | 1995-02-01 | 26.369132 |
| **2** | 1995-03-01 | 26.401129 |
| **3** | 1995-04-01 | 26.604794 |
| **4** | 1995-05-01 | 26.645437 |

### Conclusion

All the necessary data has now been collected locally; however, it is still unusable and therefore useless. Now, I will move onto exploring and analysing the data, manipulating the data along the way, to allow the data to better fit and explain the price data.

## Pre-implementation/Design – Exploratory Data Analysis

*# base/stage-1-research/p2-data-exploration.ipynb*

import os**,** json

import numpy **as** np

import pandas **as** pd

import dask.dataframe **as** dd

*# visualisation*

from pprint import pprint, pformat

import matplotlib.pyplot **as** plt

import seaborn **as** sns

**%matplotlib** inline

plt**.**rcParams["figure.figsize"] **=** (7, 5)

plt**.**rcParams["figure.dpi"] **=** 80

sns**.**set\_theme(style**=**"whitegrid")

df = dd.read\_parquet(MERGED\_PATH).compute() *# base/stage1-research/data/parts/merged*

Now, I will take the data prepared in the previous section and transform the data into a format which can be understood by a machine learning model.

I will begin by visualising the dataset with the median monthly housing price as plotted through time.

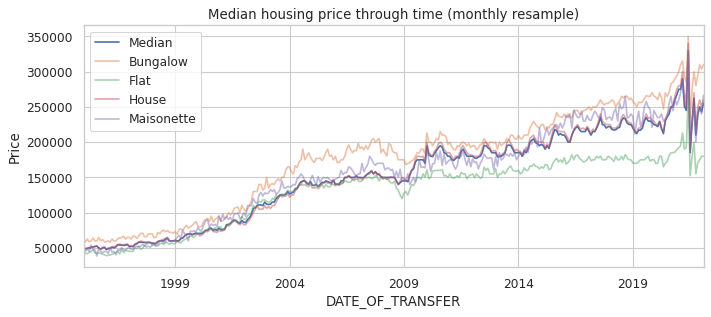


Figure 27 Graph of average house price over time

As shown, the average price of a property has grown exponentially throughout the years. This behaviour could be leveraged by a prediction model, however for ideal predictions, this needs to be *stationarised* to be able to make predictions regardless of the time the training data was observed. Another alternative is to provide time features to the predictive model, making it learn its exponential properties on its own. I will do both.

### Inflation adjustment

To be able to predict the price of a property regardless of time, I need to detrend/stationarise the prices using the house price index. I will use the adjusted prices for the sample outputs in non-extrapolation models (such as decision tree ensembles), as well as visualisation and outlier removal. Additionally, I will add the HPI and data as features and will use the raw prices as outputs for extrapolation models that can learn the time correlation instead.

To adjust the prices:

​Where,

*y*: Date of observation

To reverse this transformation, and bring the adjusted price back to a dated price, I simply need to multiply by the index of the target date:

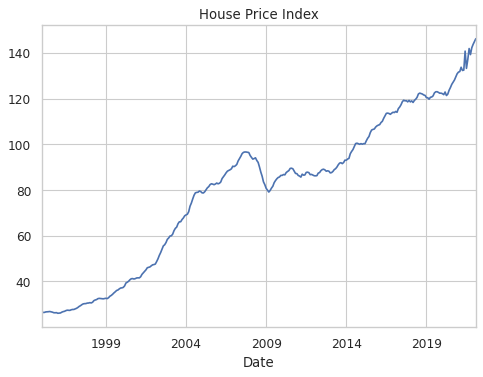
​Where,

*x*: Target date

*y*: Date of observation

*# base/stage1-research/data/raw/house-price-index.csv*

hpi **=** pd**.**read\_csv(HPI\_PATH, parse\_dates**=**["Date"])**.**set\_index("Date")**.**Index



idx **=** pd**.**DatetimeIndex(df**.**index**.**values**.**astype("datetime64[M]")) *# converting dates to start of month dates*

*# filling any missing dates in house price index*

missing **=** idx**.**unique()[**~**idx**.**unique()**.**isin(hpi**.**index)]

**if** len(missing) **>** 0:

hpi **=** pd**.**concat([hpi, pd**.**Series(index**=**missing, dtype**=**"float64")])**.**fillna(method**=**"bfill") *# backfill nans with previous index values*

hpi\_reindexed **=** hpi[idx]**.**values *# map df dates to multipliers*

df**.**insert(0, "PRICE\_ADJ", df**.**PRICE**.**values **/** hpi\_reindexed) *# adjusting prices: price \* multiplier*

*# adding time features*

df["MONTH"] **=** df**.**index**.**month

df["YEAR"] **=** df**.**index**.**year

df["HPI"] **=** hpi[idx]**.**values

I will now visualise the inflation-adjusted time series, as compared to the original.

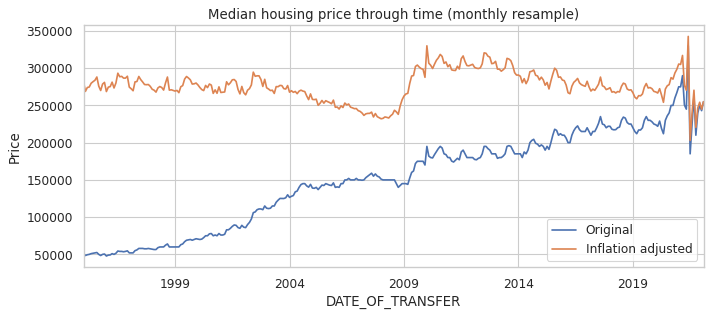


Figure 28 Graph of inflation-adjusted price over time

There is a visible seasonality in the series, with rises in price in summer months, and dips in winter. Additionally, the effects of the 2008 global economic crisis are made very apparent, and we can see more recent instability in house prices due to the COVID-19 pandemic - as it drastically breaks from the seasonal pattern in 2020.

Now, I will transform the dataset so that it is more interpretable by a machine learning algorithm. I need to convert the dataset to numeric values instead of strings. I will also reduce some of the dimensionality currently present in the data, including the cardinality of the categorical columns, and the number of features in the dataset.

### Geospatial information

Currently, the postcode of a property is completely uninterpretable by a predictive model. To leverage this useful location data, I will transform my dataset by mapping the postcodes to coordinate data instead.

pcdLookup **=** (

pd**.**read\_csv(PCD\_PATH) *# base/stage1-research/data/parts/merged*

**.**set\_index("PCDS")

)

pcdLookup**.**head()

|  | **LAT** | **LNG** |
| --- | --- | --- |
| **PCDS** |  |  |
| **AB1 0AA** | 57.101474 | -2.242851 |
| **AB1 0AB** | 57.102554 | -2.246308 |
| **AB1 0AD** | 57.100556 | -2.248342 |
| **AB1 0AE** | 57.084444 | -2.255708 |
| **AB1 0AF** | 57.096656 | -2.258102 |

df **=** pd**.**merge(df, pcdLookup, how**=**"left", left\_on**=**"POSTCODE", right\_index**=True**)**.**dropna() *# mapping postcodes to lat, long*

df **=** df**.**drop(columns**=**["POSTCODE"]) *# can now drop postcode column*

To show the significance of the coordinate data in relation to this dataset, I will produce a scatter plot of the coordinates with a colour map of the price.

sample = df[(df.PRICE\_ADJ.quantile(.1)**<**df.PRICE\_ADJ) **&** (df.PRICE\_ADJ**<**df.PRICE\_ADJ.quantile(.9))].sample(n=250\_000) *# sampling 250k points between the 10th and 90th percentiles of prices*

f, ax = plt.subplots(figsize=(8,8))

sns.scatterplot(data=sample, x="LNG", y="LAT", alpha=0.075, hue="PRICE\_ADJ", size="PRICE\_ADJ", sizes=(2, 10), ax=ax)

ax.set\_title("Price map (10th-90th percentile)")



Figure 29 Graph of price by location

The plot shows a clear relationship between location and price; further south, towards London, the price of a property is significantly higher than in the north. This behaviour will be particularly useful for a model to learn and are the most helpful features in the dataset.

### Categorical data

Categorical data can be especially useful to a predictive model if the categories are formatted properly. Since all the data passed into the algorithms must be numerical, the categorical columns must be encoded to integer/float values. One way of doing this is One-hot Encoding where each category in a column is binary mapped to a zero or one. However, one-hot encoding each category will lead to an extremely high dimensionality, since each individual categorical column will be mapped to several new columns. Therefore, I will be using Ordinal Encoding, which will number/rank the categories instead. Before doing this, I will reduce the high cardinality in the data by removing categories in columns with low counts, and merging categories where appropriate.

**for** col **in** df**.**select\_dtypes("category"):

df[col] **=** df[col]**.**cat**.**remove\_unused\_categories()

Before encoding, I will tidy up the categories held within the PROPERTY\_TYPE and BUILT\_FORM columns.

print(df**.**PROPERTY\_TYPE**.**value\_counts())

df **=** df[df**.**PROPERTY\_TYPE **!=**"Park home"] *# filtering out “Park home” as category only has 19 records*

df["PROPERTY\_TYPE"] **=** df**.**PROPERTY\_TYPE**.**cat**.**remove\_unused\_categories()

print("\nPROPERTY\_TYPE - New categories:", list(df**.**PROPERTY\_TYPE**.**cat**.**categories))

House 4091970

Flat 882359

Bungalow 477227

Maisonette 115324

Park home 19

Name: PROPERTY\_TYPE, dtype: int64

PROPERTY\_TYPE - New categories: ['Bungalow', 'Flat', 'House', 'Maisonette']

print(df**.**BUILT\_FORM**.**value\_counts())

df["BUILT\_FORM"] **=** (

df**.**BUILT\_FORM

**.**str**.**replace("Enclosed ", "")

**.**str**.**replace("Mid-","")

**.**str**.**replace("End-","")

**.**astype("category")

**.**cat**.**remove\_unused\_categories()

)

print("\nBUILT\_FORM - New categories:", list(df**.**BUILT\_FORM**.**cat**.**categories))

Semi-Detached 1800563

Mid-Terrace 1666569

Detached 1241212

End-Terrace 720056

Enclosed End-Terrace 78718

Enclosed Mid-Terrace 59762

Name: BUILT\_FORM, dtype: int64

BUILT\_FORM - New categories: ['Detached', 'Semi-Detached', 'Terrace']

I will now use the columns to make a new property type column which could be useful. Ultimately, the "usefulness" of this column will be determined by feature selection, but it could be beneficial combining the columns rather than leaving them isolated. The "Flat" and "Maisonette" categories will not be coupled with their accompanying built form category types.

prp\_map **=** {

("House","Detached"): "Detached house",

("House","Semi-Detached"): "Semi-detached house",

("House","Terrace"): "Terrace house",

("Bungalow","Detached"): "Detached bungalow",

("Bungalow","Semi-Detached"): "Semi-detached bungalow",

("Bungalow","Terrace"): "Terrace bungalow",

("Flat","Detached"): "Flat",

("Flat","Semi-Detached"): "Flat",

("Flat","Terrace"): "Flat",

("Maisonette","Detached"): "Maisonette",

("Maisonette","Semi-Detached"): "Maisonette",

("Maisonette","Terrace"): "Maisonette",

} *# mapping (PROPERTY\_TYPE, BUILT\_FORM) to new column value*

df["N\_PROPERTY\_TYPE"] **=** list(zip(df**.**PROPERTY\_TYPE, df**.**BUILT\_FORM)) *# zipping columns as pairs*

df["N\_PROPERTY\_TYPE"] **=** df**.**N\_PROPERTY\_TYPE**.**map(prp\_map)**.**astype("category") *# mapping pairs using dictionary*

print("N\_PROPERTY\_TYPE - categories:", list(df**.**N\_PROPERTY\_TYPE**.**cat**.**categories))

N\_PROPERTY\_TYPE - categories: ['Detached bungalow', 'Detached house', 'Flat', 'Maisonette', 'Semi-detached bungalow', 'Semi-detached house', 'Terrace bungalow', 'Terrace house']

I will continue tidying up the remaining columns.

print(df**.**GLAZED\_TYPE**.**value\_counts())

df **=** df[**~**df**.**GLAZED\_TYPE**.**isin([

"triple glazing", "double, known data", "triple, known data" *# these categories do not have a lot of records and can therefore be removed*

])]

df["GLAZED\_TYPE"] **=** df**.**GLAZED\_TYPE**.**cat**.**remove\_unused\_categories()

print("\nGLAZED\_TYPE - New categories:", list(df**.**GLAZED\_TYPE**.**cat**.**categories))

double glazing installed before 2002 1992825

double glazing, unknown install date 1732087

double glazing installed during or after 2002 1719791

secondary glazing 66849

single glazing 45732

triple glazing 8175

double, known data 1321

triple, known data 100

Name: GLAZED\_TYPE, dtype: int64

GLAZED\_TYPE - New categories: ['double glazing installed before 2002', 'double glazing installed during or after 2002', 'double glazing, unknown install date', 'secondary glazing', 'single glazing']

print(df**.**CONSTRUCTION\_AGE\_BAND**.**value\_counts())

df["CONSTRUCTION\_AGE\_BAND"] **=** (

df**.**CONSTRUCTION\_AGE\_BAND

**.**str**.**replace("England and Wales: ", "") *# stripping “England and Wales” prefix*

**.**replace(["2007-2011", "2012 onwards"], "2007 onwards") *# merging `2007-2011` and `2012 onwards` to `2007 onwards` columns*

**.**astype("category")

)

print("\nCONSTRUCTION\_AGE\_BAND - New categories:",list(df**.**CONSTRUCTION\_AGE\_BAND**.**cat**.**categories))

England and Wales: 1900-1929 980478

England and Wales: 1930-1949 930925

England and Wales: 1950-1966 880279

England and Wales: 1967-1975 617905

England and Wales: before 1900 577752

England and Wales: 1983-1990 371020

England and Wales: 1996-2002 304103

England and Wales: 1976-1982 293451

England and Wales: 2003-2006 269729

England and Wales: 1991-1995 193808

England and Wales: 2007 onwards 122659

England and Wales: 2007-2011 14173

England and Wales: 2012 onwards 1002

Name: CONSTRUCTION\_AGE\_BAND, dtype: int64

CONSTRUCTION\_AGE\_BAND - New categories: ['1900-1929', '1930-1949', '1950-1966', '1967-1975', '1976-1982', '1983-1990', '1991-1995', '1996-2002', '2003-2006', '2007 onwards', 'before 1900']

print(df**.**TENURE**.**value\_counts())

df["TENURE"] **=** (

df**.**TENURE

**.**replace("Owner-occupied", "owner-occupied") *# merging categories*

**.**replace("Rented (private)", "rental (private)")

**.**replace("Rented (social)", "rental (social)")

**.**replace("Not defined - use in the case of a new dwelling for which the intended tenure in not known. It is no","unknown")

**.**cat**.**remove\_unused\_categories()

)

print("\nTENURE - New categories:", list(df**.**TENURE**.**cat**.**categories))

owner-occupied 3988859

rental (private) 1140190

Owner-occupied 154158

Unknown 108172

rental (social) 86042

Rented (private) 72821

Rented (social) 4925

Not defined – use... 2117

Name: TENURE, dtype: int64

TENURE - New categories: ['owner-occupied', 'rental (private)', 'rental (social)', 'unknown']

I can now encode the categories within a column by ranking them in order based on their average property price. I will later store these mappings to a JSON file to encode inputs when serving requests to the API.

**def** **RankingEncode**(df, target, categories):

**if** isinstance(categories, str): categories **=** [categories]

**return** {

cat: {

k: i

**for** i, k **in** enumerate(

df**.**groupby(cat)[target]**.**median()**.**sort\_values()**.**index**.**values

)

}

**for** cat **in** categories

}

Visualising the algorithm’s rankings:

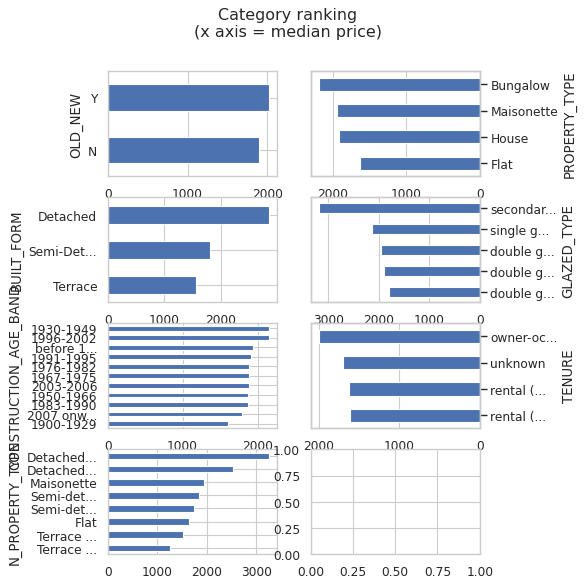


Figure 30 Ordinal ranking of features

The categorical variables have been assigned increasing integers corresponding to their effect on the price, and the OLD\_NEW column has been binary encoded. However, the ranked order for the CONSTRUCTION\_AGE\_BAND column does not make logical sense; therefore, I will rank the categories in chronological order instead.

*# re-encoding age bands in increasing chronological order*

mappings["CONSTRUCTION\_AGE\_BAND"] **=** {

k: i

**for** i, k **in** enumerate(

*# ordering categories in chronological order*

["before 1900", **\***list(df**.**CONSTRUCTION\_AGE\_BAND**.**cat**.**categories)[:**-**1]]

)

}

New chronological encoding:

{'before 1900': 0,

'1900-1929': 1,

'1930-1949': 2,

'1950-1966': 3,

'1967-1975': 4,

'1976-1982': 5,

'1983-1990': 6,

'1991-1995': 7,

'1996-2002': 8,

'2003-2006': 9,

'2007 onwards': 10}

*# encoding categories*

**for** col, map\_ **in** mappings**.**items():

df[col] **=** df[col]**.**map(map\_)

The new encoded dataset looks like:

(Column names condensed for size)

|  | **PRICE\_ADJ** | **PRICE** | **OLD** | **PRP** | **BLT** | **AREA** | **GLA** | **EXT** | **ROOMS** | **AGE** | **TEN** | **M** | **YEAR** | **HPI** | **LAT** | **LNG** | **N\_PRP** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **DATE** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **1995-01-01** | 605.142016 | 16000.0 | 0 | 1 | 0 | 99.00 | 0 | 0.0 | 4.0 | 0 | 3 | 1 | 1995 | 26.4 | 53.7 | -1.88 | 1 |
| **1995-01-02** | 1323.748160 | 35000.0 | 0 | 1 | 0 | 85.00 | 1 | 1.0 | 4.0 | 1 | 3 | 1 | 1995 | 26.4 | 52.2 | -0.87 | 1 |
| **1995-01-03** | 567.320640 | 15000.0 | 0 | 1 | 1 | 74.00 | 1 | 0.0 | 4.0 | 4 | 1 | 1 | 1995 | 26.4 | 54.9 | -1.41 | 4 |
| **1995-01-03** | 1815.426048 | 48000.0 | 0 | 3 | 2 | 78.00 | 1 | 1.0 | 3.0 | 3 | 3 | 1 | 1995 | 26.4 | 52.0 | -2.43 | 6 |
| **1995-01-03** | 3101.352831 | 82000.0 | 0 | 3 | 2 | 118.88 | 2 | 0.0 | 4.0 | 4 | 1 | 1 | 1995 | 26.4 | 50.0 | -5.26 | 6 |

### Feature selection

To reduce the number of unnecessary features, I will test the features' importance, and remove features that are not significant.

I will use two metrics to evaluate feature importance:

* Mutual information
* Pearson's correlation coefficient, calculated as such:

sample = df.sample(n=100\_000)

X = sample.drop(columns=["PRICE","PRICE\_ADJ"]) *# removing labels from feature importance calculation*

y = sample.PRICE *# evaluating importance on raw prices*

*# using library utilities to evaluate*

from sklearn.feature\_selection import mutual\_info\_regression, r\_regression

mut = mutual\_info\_regression(X, y, random\_state=0)

pearson = np.abs(r\_regression(X, y))

Chart, bar chart

Description automatically generated

Figure 31 Mutual information and Absolute Pearson Correlation of features w.r.t. property price

corr = sample.astype(float).drop(columns="PRICE\_ADJ").corr() *# matrix correlation using pandas library*

*# masking triangle to improve visualisation*

mask = np.zeros\_like(corr, dtype=bool)

mask[np.triu\_indices\_from(mask)]=**True**

|  | **PRICE** | **OLD\_NEW** | **PROPE...** | **BUILT...** | **TOTAL...** | **GLAZE...** | **EXTEN...** | **NUMBE...** | **CONST...** | **TENURE** | **MONTH** | **YEAR** | **HPI** | **LAT** | **LNG** | **N\_PRO...** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **PRICE** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **OLD\_NEW** | -0.033 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **PROPE...** | 0.027 | -0.129 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **BUILT...** | 0.123 | -0.023 | 0.302 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **TOTAL...** | 0.350 | -0.049 | 0.113 | 0.326 |  |  |  |  |  |  |  |  |  |  |  |  |
| **GLAZE...** | 0.042 | 0.092 | -0.070 | 0.023 | 0.035 | nan |  |  |  |  |  |  |  |  |  |  |
| **EXTEN...** | 0.112 | -0.089 | 0.082 | 0.111 | 0.347 | -0.041 |  |  |  |  |  |  |  |  |  |  |
| **NUMBE...** | 0.257 | -0.080 | 0.173 | 0.343 | 0.738 | 0.004 | 0.368 | nan | nan | nan | nan | nan | nan | nan | nan | nan |
| **CONST...** | -0.031 | 0.305 | -0.100 | 0.171 | -0.095 | 0.150 | -0.278 | -0.128 |  |  |  |  |  |  |  |  |
| **TENURE** | 0.078 | -0.133 | 0.197 | 0.214 | 0.179 | -0.004 | 0.111 | 0.221 | -0.037 |  |  |  |  |  |  |  |
| **MONTH** | 0.004 | 0.009 | 0.011 | 0.023 | 0.006 | 0.003 | 0.002 | 0.009 | 0.012 | 0.023 | nan | nan | nan | nan | nan | nan |
| **YEAR** | 0.162 | -0.204 | 0.077 | 0.064 | 0.029 | -0.021 | 0.048 | 0.036 | -0.005 | 0.256 | -0.032 |  |  |  |  |  |
| **HPI** | 0.159 | -0.162 | 0.055 | 0.043 | 0.017 | -0.024 | 0.041 | 0.019 | 0.011 | 0.210 | 0.025 | 0.963 |  |  |  |  |
| **LAT** | -0.161 | 0.026 | 0.036 | -0.034 | -0.027 | -0.049 | -0.013 | 0.021 | -0.021 | 0.006 | -0.001 | -0.007 | 0.005 |  |  |  |
| **LNG** | 0.131 | -0.013 | -0.048 | -0.019 | -0.003 | 0.016 | -0.010 | -0.046 | -0.005 | -0.039 | -0.006 | -0.021 | -0.025 | -0.223 |  |  |
| **N\_PRO...** | 0.143 | -0.027 | 0.316 | 0.898 | 0.379 | 0.020 | 0.143 | 0.408 | 0.164 | 0.225 | 0.025 | 0.066 | 0.044 | -0.030 | -0.013 |  |

corr[mask]=np.nan

I will now choose the 6 best features. Since YEAR and HPI are highly correlated, and both perform equally as well, I will arbitrarily choose the HPI column between them.

df **=** df[["PRICE", "PRICE\_ADJ", "HPI", "LAT", "LNG", "TOTAL\_FLOOR\_AREA", "NUMBER\_HABITABLE\_ROOMS", "N\_PROPERTY\_TYPE"]]

df["PROPERTY\_TYPE"] **=** df**.**pop("N\_PROPERTY\_TYPE")

|  | **PRICE** | **PRICE\_ADJ** | **HPI** | **LAT** | **LNG** | **AREA** | **NUMBER\_HABIT...** | **PROPER...** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **DATE\_OF\_...** |  |  |  |  |  |  |  |  |
| **1995-01-01** | 16000 | 605.14 | 26.44 | 53.72 | -1.88 | 99.00 | 4.0 | 1 |
| **1995-01-02** | 35000 | 1323.7 | 26.44 | 52.24 | -0.87 | 85.00 | 4.0 | 1 |
| **1995-01-03** | 15000 | 567.32 | 26.44 | 54.96 | -1.41 | 74.00 | 4.0 | 4 |
| **1995-01-03** | 48000 | 1815.4 | 26.44 | 52.03 | -2.43 | 78.00 | 3.0 | 6 |
| **1995-01-03** | 82000 | 3101.3 | 26.44 | 50.09 | -5.26 | 118.88 | 4.0 | 6 |

The dataset has now been significantly reduced in dimensionality, stripping away unnecessary information, which will improve the efficiency and performance in training prediction algorithms.

I will now do some final steps before training: outlier removal and feature rescaling. These steps are necessary to provide prediction models with more comparable (thus more informative) distributions and data points.

### Distribution and skewness

*Skewed* data can be described as data where the distribution curve is asymmetrical. Skewed data can either be *left-skewed* or *right-skewed*. Data with right skew (positive skew) will have a mean > median > mode. Data with left skew (negative skew) will have a mean < median < mode.

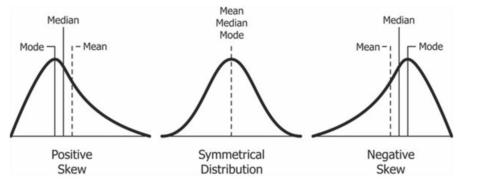


Figure 32 Diagram demonstrating skewness of distribution

Skewed data can adversely affect a model's performance especially in regression-based models, such as linear regression, where the model assumes the data follows normal distribution and has no outliers. Therefore, the model can tend to favour more extreme values, rather than values which are within the true range. Tree-based models, which I may use, are not susceptible to this sort of situation, however, to train other types of models, I will need to transform the data to properly balance it.

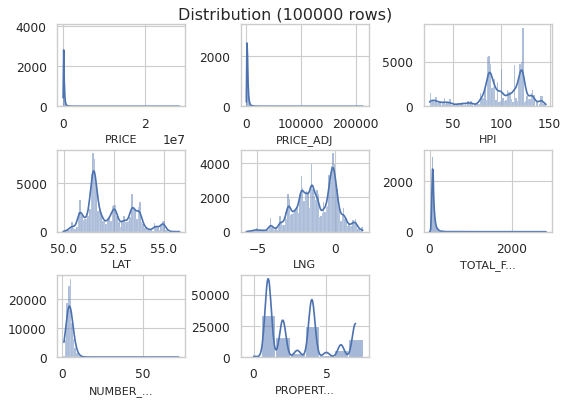


Figure 33 Distribution of data for a sample of 100,000 rows

skew **=** ["PRICE\_ADJ", "TOTAL\_FLOOR\_AREA", "NUMBER\_HABITABLE\_ROOMS"]

print("Data skew:")

df[skew]**.**skew()

Data skew:

PRICE\_ADJ 264.080204

TOTAL\_FLOOR\_AREA 270.696094

NUMBER\_HABITABLE\_ROOMS 2.919180

dtype: float64

The graph clearly shows extreme positive skew in the price, floor area and habitable room columns. Since the dataset should ideally follow normal distribution, I will *unskew* this data by removing outliers and potentially transforming the data for a better distribution. I will detect outliers by filtering values according to their percentile within the dataset. A data point outside of a given percentile range will be interpreted as an outlier value and will be removed.

**def** **PercentileOutliers**(x, llim, ulim):

x **=** np**.**asarray(x)

**if** x**.**ndim **==** 2:

*# converting numeric parameter to list of [numeric parameter]*

**if** isinstance(llim, (int, float)):

llim **=** [llim] **\*** x**.**shape[1]

**if** isinstance(ulim, (int, float)):

ulim **=** [ulim] **\*** x**.**shape[1]

*# llim/ulim is a list of percentiles with length of input columns*

lower **=** np**.**array(

[np**.**percentile(x[:, i], llim[i]) **for** i **in** range(x**.**shape[1])])

upper **=** np**.**array(

[np**.**percentile(x[:, i], ulim[i]) **for** i **in** range(x**.**shape[1])])

**else**:

**if** **not** isinstance(llim, (int, float)):

**raise** **ValueError**(f"Provide a numeric `llim` for 1-d data not {type(llim)}")

**if** **not** isinstance(ulim, (int, float)):

**raise** **ValueError**(f"Provide a numeric `ulim` for 1-d data not {type(ulim)}")

lower **=** np**.**percentile(x, llim)

upper **=** np**.**percentile(x, ulim)

**return** (lower **<** x) **&** (x **<** upper) *# returning mask with shape of array `x`*

l **=** len(df)

df **=** df[PercentileOutliers(

df[skew],

*# [price, area, rooms]*

llim **=** [1, 5, 0],

ulim **=** [99, 95, 98]

)**.**all(axis**=**1) *# A row must contain no outliers to be included in the dataset*

]

print(f"{len(df):,} rows, {l**-**len(df):,} rows removed\n\nNew data skew:")

df[skew]**.**skew()

4,903,637 rows, 653,647 rows removed

New data skew:

PRICE\_ADJ 2.009345

TOTAL\_FLOOR\_AREA 0.909617

NUMBER\_HABITABLE\_ROOMS 0.390720

dtype: float64

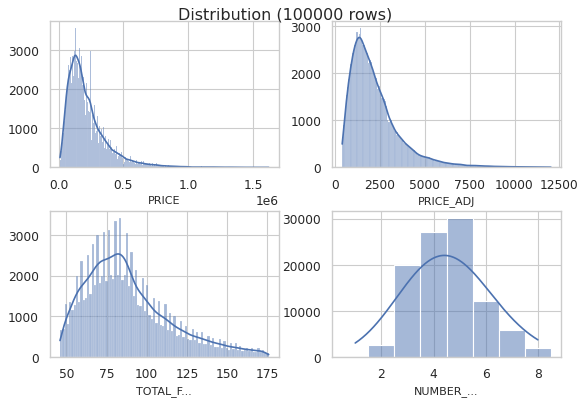
****

Figure 34 New distributions of skewed columns after outlier removal

I am much happier with the new distributions; while they are still positively skewed, they are not as extreme as their previous distributions. Removing any more data or performing skew correcting transformations (such as log transformation) could impact performance or add unnecessary complexity (such as having to apply and reverse transformations in making predictions).

### Conclusion

I will now export the new dataset. The non-interpretable string categories have now been encoded into numeric values, and much of the outliers and instability has been removed; the data has been successfully formatted in a way in which a model can understand it to produce predictions.

*# saving mappings for remaining categorical columns*

*# base/stage2-app/model/data/encoding.json*

json**.**dump({k**.**lower(): mappings[k] **for** k **in** df**.**select\_dtypes("category")**.**columns},

open(os**.**path**.**join(SERVER\_DIR, "encoding.json"), "w"),

indent**=**2)

df **=** df**.**dropna()

df**.**to\_parquet(DATASET\_PATH) *# base/stage1-research/data/dataset.parquet*

Note: The HPI feature was replaced by the YEAR feature, however, all the above implementation remains the same.

## Implementation – Model selection

*# base/stage-1-research/p3-model-selection.ipynb*

import os**,** json

import warnings

from joblib import load, dump

from tqdm import tqdm

import numpy **as** np

import pandas **as** pd

from typing import Callable, Union

from numpy import ndarray

*# ml algos dependencies*

from cuml import (

NearestNeighbors, *# knn implementation dependency*

RandomForestRegressor,

ForestInference *# gpu accelerated front-end for xgboost, random forest etc.*

)

import xgboost **as** xgb

import treelite *# model compiler*

*# hyperparameter optimisation*

from hyperopt import hp, fmin, tpe, Trials

*# visualisation*

from pprint import pformat

import matplotlib.pyplot **as** plt

import seaborn **as** sns

**%matplotlib** inline

plt**.**rcParams["figure.figsize"] **=** (7, 5)

plt**.**rcParams["figure.dpi"] **=** 64

sns**.**set\_theme(style**=**"whitegrid")

SEED = 10 *# reproducibility*

df **=** pd**.**read\_parquet(DATASET\_PATH) *# base/stage-1-research/data/dataset.parquet*

### Final Pre-processing

To use distance-based algorithms, which use distance between data points to determine similarity, I will have to scale the data to a closer range. Tree-based algorithms are insensitive to the scaling of data since it only splits a node based on a single feature. Additionally, scaling the data will make results and models more interpretable, as you will more clearly be able to explain the strength of a particular feature determined by the value of its coefficient for the target variable.

I will use standardisation (Z-score normalisation) to rescale the dataset, calculated by the formula:

Where,  
*μ*: mean of the dataset  
*σ*: standard deviation of the dataset

The standardised data will have zero mean and unit variance; therefore, it will allow comparison between different features, which would usually have different unit scales. It is important I compute the mean and standard deviation using only the train set, as using the entire dataset could cause unintentional data leakage to the prediction model, i.e., it could unintentionally provide information about unobserved data.

Another method I may try is normalisation or min-max scaling which rescales data to a given range. It is calculated with the formula:

Where,  
[*a,b*]: interval in which the dataset will be scaled

Again, the min and max values should be computed using only the train set to prevent any unintentional data leakage to the prediction model.

**class** **Scaler**:

**def** \_\_init\_\_(self):

**pass**

**def** **fit**(self, x):

self**.**\_fit **=** np**.**array(x) *# fitting an array type for scaling*

**return** self

**def** **fit\_transform**(self, x, method**=**"standardise", **\*\***kwargs): *# fitting and scaling at once*

**return** self**.**fit(x), self**.**scale(x, method, **\*\***kwargs)

**def** **standardise**(self, x):

self**.**\_mean **=** self**.**\_fit**.**mean(axis**=**0) *# computing mean for columns*

self**.**\_std **=** self**.**\_fit**.**std(axis**=**0) *# computing std. deviation for columns*

*# standard scaling transformation*

**return** (np**.**array(x) **-** self**.**\_mean)**/**self**.**\_std

**def** **inv\_standardise**(self,x):

*# reversing standard scaling transformation*

**return** np**.**array(x) **\*** self**.**\_std **+** self**.**\_mean

**def** **min\_max**(self, x, feature\_range**=**(0,1)):

self**.**\_a, self**.**\_b **=** feature\_range

self**.**\_min **=** self**.**\_fit**.**min(axis**=**0) *# computing minimum values for columns*

self**.**\_max **=** self**.**\_fit**.**max(axis**=**0) *# computing maximum values for columns*

*# min-max scaling transformation*

**return** ((np**.**array(x)**-**self**.**\_min)**\***(self**.**\_b**-**self**.**\_a))**/**(self**.**\_max**-**self**.**\_min) **+** self**.**\_a

**def** **inv\_min\_max**(self,x):

*# reversing min-max scaling transformation*

**return** ((np**.**array(x)**-**self**.**\_a)**\***(self**.**\_max**-**self**.**\_min))**/**(self**.**\_b**-**self**.**\_a) **+** self**.**\_min

**def** **scale**(self, x, method:str**=**"standardise", **\*\***kwargs):

scale\_method **=** {

"standardise":self**.**standardise,

"zscore":self**.**standardise,

"min\_max":self**.**min\_max,

"normalise":self**.**min\_max,

}[method]

**return** scale\_method(x)

**def** **inv**(self, x, method**=**"standardise",**\*\***kwargs):

inv\_method **=** {

"standardise":self**.**inv\_standardise,

"zscore":self**.**inv\_standardise,

"min\_max":self**.**inv\_min\_max,

"normalise":self**.**inv\_min\_max,

}[method]

**return** inv\_method(x)

**def** **serialize**(self, names):

*# serializing scaling attributes in a dictionary to export as JSON format*

serialized **=** {}

**if** self**.**\_fit**.**ndim **==** 2:

*# serializing 2d array scaling attributes as*

*# { name: { mean: x, std: y }, ...}*

**for** i, name **in** enumerate(names):

**if** hasattr(self, "\_mean"): *# adding mean, std if standard scaling used*

serialized[name] **=** {

"mean": self**.**\_mean[i],

"std": self**.**\_std[i]

}

**if** hasattr(self, "\_min"): *# adding min, max if min-max scaling was used*

serialized[name] **=** {

"min": self**.**\_min[i],

"max": self**.**\_max[i]

}

**else**: *# otherwise serializing as single column*

**if** hasattr(self, "\_mean"):

serialized[names] **=** {

"mean": self**.**\_mean,

"std": self**.**\_std

}

**if** hasattr(self, "\_min"):

serialized[names] **=** {

"min": self**.**\_min,

"max": self**.**\_max

}

*# adding min-max feature range if min-max scaling was used*

**if** hasattr(self,"\_a"):

serialized["minmax\_range"] **=** [self**.**\_a, self**.**\_b]

**return** serialized

*# testing scaling on small sample set and visualising distributions*

sample **=** df**.**sample(n**=**100\_000)

sample\_std **=** Scaler()**.**fit\_transform(sample, method**=**"standardise")[1] *# standard scaling*

sample\_norm **=** Scaler()**.**fit\_transform(sample, method**=**"normalise")[1] *# min-max scaling*

f, axs **=** plt**.**subplots(1,3,sharey**=True**)

sns**.**violinplot(data**=**sample**.**values, scale**=**"width", orient**=**"h", ax**=**axs[0])**.**set\_xlabel("Original")

sns**.**violinplot(data**=**sample\_std, scale**=**"width", orient**=**"h", ax**=**axs[1])**.**set\_xlabel("Standardised")

sns**.**violinplot(data**=**sample\_norm, scale**=**"width", orient**=**"h", ax**=**axs[2])**.**set\_xlabel("Min-max")

axs[0]**.**set\_xlim(**-**10, 125)

axs[0]**.**set\_yticklabels([col[:6]**+**"..." **if** len(col) **>** 9 **else** col **for** col **in** sample**.**columns])

axs[0]**.**tick\_params("y", labelsize**=**"small")

Chart, diagram

Description automatically generated

Min-max

Figure 35 Testing data rescaling

### Pre-processed data

I will begin by scraping the latest HPI from a reliable source using BeautifulSoup and use it to take predictions with the adjusted prices to current prices, for better context and interpretability.

from requests import get

from bs4 import BeautifulSoup

**def** **getLatestIndex**():

response **=** get("https://landregistry.data.gov.uk/app/ukhpi")

soup **=** BeautifulSoup(response**.**content)

**return** float(soup**.**find(class\_**=**"c-headline-figure\_\_house-price-index")**.**text**.**strip()[:**-**1])

LATEST\_INDEX **=** getLatestIndex()

LATEST\_INDEX

145.15

I will split the dataset into 3 parts:

* training set - the data which will be provided to the model.
* calibration set - used to compute nonconformity scores for conformal prediction.
* validation set - the data which will be used to evaluate the performance of the model. The calibration and validation sets will be scaled according to the properties of the training set, as to avoid unintentional data leakage which could potentially give unrealistic performance.

Additionally, I will have 2 different output samples:

* y - the raw prices
* y2 - the adjusted prices

Depending on what model I choose to use, I will select either the raw or the adjusted prices, determined by whether the model can perform extrapolation or not, for example tree models which cannot extrapolate.

*# Removing output sample columns from input dataset*

X, y, y2 **=** df**.**drop(columns**=**["PRICE", "PRICE\_ADJ"]), df**.**PRICE, df**.**PRICE\_ADJ

*# splitting*

**def** **split**(df, tr\_pct**=**0.9, cb\_pct**=.**05):

tr\_spl **=** int(len(df)**\***tr\_pct)

cb\_spl **=** tr\_spl **+** int(len(df)**\***cb\_pct)

**return** df[:tr\_spl], df[tr\_spl:cb\_spl], df[cb\_spl:]

np**.**random**.**seed(SEED)

tr\_idx, cal\_idx, val\_idx **=** split(np**.**random**.**permutation(len(df))) *# indices for shuffled dataset*

*# scaling training data*

X\_scaler, tr\_X **=** Scaler()**.**fit\_transform(X**.**iloc[tr\_idx])

y\_scaler, tr\_y **=** Scaler()**.**fit\_transform(y**.**iloc[tr\_idx])

y2\_scaler, tr\_y2 **=** Scaler()**.**fit\_transform(y2**.**iloc[tr\_idx])

*# scaling and splitting validation/calibration data*

cal\_X, val\_X **=** X\_scaler**.**scale(X**.**iloc[cal\_idx]), X\_scaler**.**scale(X**.**iloc[val\_idx])

cal\_y, val\_y **=** y\_scaler**.**scale(y**.**iloc[cal\_idx]), y\_scaler**.**scale(y**.**iloc[val\_idx])

cal\_y2, val\_y2 **=** y2\_scaler**.**scale(y2**.**iloc[cal\_idx]), y2\_scaler**.**scale(y2**.**iloc[val\_idx])

*# unscaled expected outputs for validation set*

inv\_y **=** y**.**values[val\_idx]

inv\_y2 **=** y2**.**values[val\_idx] **\*** LATEST\_INDEX *# adjusted to world prices for interpretability*

*# serializing scaling properties for server use*

SCALING\_PATH **=** os**.**path**.**join(SERVER\_DIR, "scaling.json") *# base/stage-2-app/model/data*

json**.**dump({

**\*\***y\_scaler**.**serialize("price"),

**\*\***y2\_scaler**.**serialize("price\_adj"),

**\*\***X\_scaler**.**serialize(X**.**columns**.**str**.**lower())

}, open(SCALING\_PATH,"w"), indent**=**2)

full dataset 4,903,637 rows -

-----------------------------------------------

training set 4,413,273 rows 90.00%

calibration set 245,181 rows 5.00%

validation set 245,183 rows 5.00%

### Model selection – Evaluation metrics

Some of the metrics that will be used in evaluating models:

### Model selection – Conformal regression algorithm

*# algorithms/conformal\_regression/\_\_init\_\_.py*

ALPHA **=** 0.15

**class** **ConformalRegression**(object):

**def** \_\_init\_\_(self, model):

self**.**model **=** model

**def** **calibrate**(self, cal\_X, cal\_y):

yhat **=** self**.**model**.**predict(cal\_X)

self**.**resid\_ **=** np**.**abs(cal\_y**-**yhat)

**return** self

**def** **fit**(self, X, y, cal\_X, cal\_y, **\*\***kwargs):

self**.**model**.**fit(X, y, **\*\***kwargs)

**return** self**.**calibrate(cal\_X,cal\_y)

**def** **predict**(self, X, alpha **=** ALPHA):

yhat **=** self**.**model**.**predict(X)

**if** alpha **is** **None**:

**return** yhat

**if** alpha **>=** 1 **or** alpha **<=** 0:

**raise** **ValueError**("'alpha' must be in interval (0, 1) or None")

quantile **=** np**.**quantile(self**.**resid\_, 1 **-** alpha)

yhat\_low **=** yhat **-** quantile

yhat\_up **=** yhat **+** quantile

**return** yhat, np**.**column\_stack([yhat\_low, yhat\_up])

### Model selection – Multiple linear regression

Initially, I will begin by fitting a multiple linear regression model on my data. I expect this to perform poorly, as the linear regression will not capture the complex relationships and will assume linearity between the multiple predictor variables and the target variable. However, this will act as a baseline model for comparative purposes and will lay down some of the basic principles behind the multilayer perceptron model.

#### Implementation

*# algorithms/linear\_regression/\_\_init\_\_.py*

**class** **LinearRegression**:

@staticmethod

**def** **loss\_fn**(y,yhat):

**return** np**.**mean(np**.**square(yhat**-**y))

**def** \_\_init\_\_(self, lr:float**=.**01):

self**.**lr**=**lr

self**.**history **=** **None**

self**.**bias **=** 0

**def** **predict**(self,X:ndarray):

**return** X**.**dot(self**.**weights) **+** self**.**bias *# y = w.x + b*

**def** **fit**(self,

X:ndarray,

y:ndarray,

val\_X:ndarray**=None**,

val\_y:ndarray**=None**,

epochs:int**=**1000,

early\_stopping:int**=**50,

delta:float**=**0.01,

verbose:bool**=True**

):

X,y **=** np**.**asarray(X), np**.**asarray(y)

self**.**weights **=** np**.**zeros(X**.**shape[1])

self**.**history **=** dict(loss**=**list())

**if** val\_X **is** **not** **None** **and** val\_y **is** **not** **None**:

self**.**history["val\_loss"] **=** []

val\_X, val\_y **=** np**.**asarray(val\_X), np**.**asarray(val\_y)

patience **=** **None**

*# initialise early stopping parameters*

**if** isinstance(early\_stopping, int) **and** early\_stopping **>** 0:

patience **=** early\_stopping

best\_loss **=** np**.**inf

**for** i **in** (pbar**:=**tqdm(range(epochs), disable**=not** verbose)):

**if** patience **is** **not** **None** **and** patience **<=** 0:

**break**

yhat **=** self**.**predict(X)

*# cost gradient wrt params*

weight\_grad **=** np**.**dot(X**.**T, yhat**-**y) **/** len(y) *# gradient of loss wrt weights*

bias\_grad **=** np**.**mean(yhat**-**y) *# gradient of loss wrt bias*

*# updating params (stochastic gradient descent)*

self**.**weights **-=** self**.**lr **\*** weight\_grad

self**.**bias **-=** self**.**lr **\*** bias\_grad

loss **=** self**.**loss\_fn(y,yhat)

self**.**history["loss"]**.**append(loss)

*# testing model on validation set if provided*

**if** "val\_loss" **in** self**.**history:

self**.**history["val\_loss"]**.**append(self**.**loss\_fn(val\_y,self**.**predict(val\_X)))

*# logging epoch info if enabled*

**if** verbose:

msg **=** {"loss": self**.**history["loss"][**-**1]}

**if** "val\_loss" **in** self**.**history:

msg["val\_loss"] **=** self**.**history["val\_loss"][**-**1]

**if** patience **is** **not** **None**:

msg["patience"] **=** patience

pbar**.**set\_postfix(msg)

*# skip to next iteration if early stopping is disabled*

**if** patience **is** **None**:

**continue**

*# if model has improved greater than minimum delta*

**if** best\_loss **-** loss **>=** delta:

patience **=** early\_stopping *# reset patience*

best\_loss **=** loss

**continue** *# skip to next iteration*

patience **-=** 1 *# decrement patience (code won’t be reached if model has improved)*

**return** self

#### Fitting and evaluating

fn **=** "models/LinearRegression.bin"

**if** **not** os**.**path**.**exists(fn):

reg **=** LinearRegression()**.**fit(tr\_X, tr\_y, val\_X**=**val\_X, val\_y**=**val\_y)

dump(reg, fn)

reg **=** ConformalRegression(load(fn))**.**calibrate(cal\_X, cal\_y)

Chart, line chart, histogram

Description automatically generated

MAE (78,682) is 35.49% of mean

RMSE (117,887) is 53.18% of mean

MAPE 47.35%

R2 Score is 0.479

Chart

Description automatically generated

The model performed better than expected, however it still wasn't great. We can begin to see which features have the most importance, such as the total floor area and location data. I will avoid other linear models, as they will likely perform similarly, and will instead try neighbour and tree-based methods instead.

### Model selection – K-Nearest neighbours

I will now try the K-Nearest Neighbour algorithm to test if the distances between points, especially in their coordinates, can significantly impact the outcome price. The KNN algorithm works by using feature similarity to predict the values of new unseen data. A new point is assigned a value based on how closely it resembles other points in the training set, i.e. the distance between them. The distance can be calculated using the Euclidean distance, or other methods such as the Manhattan, or Chebyzchev, distance. It then sorts the points and aggregates the closest points for the outcome. This could be majority voting for classification, or the neighbourhood average for regression. The model is very sensitive to the **curse of dimensionality**, where adding more features doesn't benefit the model, and rather adds noise and hinders its performance.

#### Implementation

I will implement a custom model that utilises cuML's nearest neighbour search

that will return the prediction as the **inverse distance weighted** average of the nearest neighbours, which will favour closer points.

To avoid zero division errors, I will only compute the weights of non-zero rows and will fill any remaining rows with a weight of 1 for zero distance, and 0 for all other distances. Since zero means the user's input is identical to some observed data, setting all other weights to 0 will use the zero-distance sample as the output.

*# algorithms/knn/\_\_init\_\_.py*

**class** **KNNRegressor**(object):

**def** \_\_init\_\_(self, n\_neighbours:int**=**5, idw\_power:int**=**1):

self**.**n\_neighbours\_ **=** n\_neighbours

self**.**idw\_power\_ **=** idw\_power

**def** **fit**(self, X, y, **\*\***kwargs):

self**.**fit\_X\_ **=** X

self**.**fit\_y\_ **=** y

self**.**nearest\_neigh\_ **=** NearestNeighbors(**\*\***kwargs)

self**.**nearest\_neigh\_**.**fit(X)

**return** self

**def** **weights\_fn**(self, dist):

weights **=** np**.**empty\_like(dist)

*# masking all non-zero distance samples*

non\_zero **=** np**.**all(dist**!=**0, axis**=**1)

*# setting non-zero weights to 1/dist^p*

weights[non\_zero] **=** 1 **/** np**.**power(dist[non\_zero], self**.**idw\_power\_ )

*# binary mapping zero distance to 1 and 0 for all other.*

weights[**~**non\_zero] **=** (dist[**~**non\_zero] **==** 0)**.**astype(float)

**return** weights

**def** **predict**(self, X):

*# getting k nearest neighbours*

dist, idx **=** self**.**nearest\_neigh\_**.**kneighbors(X, self**.**n\_neighbours\_)

*# distance weighted average using weighting function*

**return** np**.**average(self**.**fit\_y\_[idx], axis**=**1, weights**=**self**.**weights\_fn(dist))

#### Fitting and evaluating

fn **=** "models/KNearestNeigh.bin"

**if** **not** os**.**path**.**exists(fn):

reg **=** KNNRegressor(n\_neighbours**=**5)**.**fit(tr\_X[:,1:], tr\_y2) *# omitting YEAR column as not informative for stationary adjusted prices*

dump(reg, fn)

reg **=** ConformalRegression(load(fn))**.**calibrate(cal\_X[:,1:], cal\_y2)

MAE (63,421) is 28.61% of mean

RMSE (103,374) is 46.63% of mean

MAPE 21.28%

R2 Score is 0.767

Chart

Description automatically generated

### Model selection – Decision tree regression

Now, I will attempt to implement a regression tree. I don’t expect the model to perform well on its own and will probably need to an ensemble method for best performance. My implementation will likely also be inefficient and take a considerable amount of time in training, as I will be using recursion and expensive computations to build the tree. Additionally, I will suffer Python's various bottlenecks and overhead, hindering performance, where better, more established implementations use C++ instead.

#### Implementation

*# algorithms/decision\_tree/base.py*

**class** **BaseNode**(object):

**def** **predict**(self,X:ndarray):

**raise** **NotImplementedError**()

**def** **numeric**(x, name, **\***, dtype**=**(float,int), min**=**float("-inf"), max**=**float("inf"), min\_closed**=True**, max\_closed**=True**):

**if** **not** isinstance(x, dtype):

type\_ **=** dtype**.**\_\_name\_\_ **if** hasattr(dtype,"\_\_name\_\_") **else** [dt**.**\_\_name\_\_ **for** dt **in** dtype]

**raise** **TypeError**(f"'{name}' parameter must be of type {type\_}")

interval **=** {

(**True**,**True**): **lambda** x: min **<=** x **<=** max, *# if closed interval*

(**True**,**False**): **lambda** x: min **<=** x **<** max, *# if min closed, max open*

(**False**,**True**): **lambda** x: min **<** x **<=** max, *# if min open, max closed*

(**False**,**False**): **lambda** x: min **<** x **<** max, *# if open interval*

}

mibr **=** "[" **if** min\_closed **else** "("

mabr **=** "]" **if** max\_closed **else** ")"

**if** **not** interval[(min\_closed, max\_closed)](x):

**raise** **ValueError**(f"'{name}' parameter must be in interval {mibr}{min},{max}{mabr}")

**return** x

AggregateFunction **=** Callable[[ndarray],float]

CriterionFunction **=** Callable[[ndarray],ndarray]

ThreshSelectFunction **=** Callable[[ndarray],ndarray]

*# algorithms/decision\_tree/node.py*

**class** **DecisionNode**(BaseNode):

**def** \_\_init\_\_(self,feature\_idx,threshold,left,right):

self**.**feature\_idx **=** feature\_idx

self**.**threshold **=** threshold

self**.**left **=** left

self**.**right **=** right

**class** **LeafNode**(BaseNode):

**def** \_\_init\_\_(self,data):

self**.**data **=** data

**def** **predict**(self,X:ndarray):

*# finally reach leaf, so return data*

**return** self**.**data

*# algorithms/decision\_tree/\_\_init\_\_.py*

from collections import namedtuple

Node **=** namedtuple("Node", "feature\_idx threshold left right data", defaults**=**[**None**]**\***5)

**class** **DecisionTree**(object):

*# ===== aggregation functions =====*

*# == regression ==*

@staticmethod

**def** **mean\_aggr**(y):

*# aggregating predictions by mean*

**return** np**.**mean(y)

*# == classification ==*

@staticmethod

**def** **majority\_aggr**(y):

*# returning class with majority prediction*

vals, counts **=** np**.**unique(y, return\_counts**=True**)

**return** vals[np**.**argmax(counts)]

*# ===== criterion functions =====*

*# == regression ==*

@staticmethod

**def** **variance**(y):

**return** np**.**sum(np**.**square(y **-** y**.**mean())) **/** len(y)

*# == classification ==*

@staticmethod

**def** **\_calc\_proba**(y):

**return** np**.**unique(y,return\_counts**=True**)[1] **/** len(y)

@staticmethod

**def** **gini**(y):

proba **=** DecisionTree**.**\_calc\_proba(y)

**return** 1 **-** np**.**sum(np**.**square(proba))

@staticmethod

**def** **entropy**(y):

proba **=** DecisionTree**.**\_calc\_proba(y)

**return** **-** np**.**sum(proba**\***np**.**log2(proba))

*# ===== impurity reduction / information gain =====*

**def** **information\_gain**(self, parent, left, right) **->** float:

*# information gain = (parent impurity) - (average children impurity)*

parent\_impurity **=** self**.**criterion(parent)

left\_impurity **=** self**.**criterion(left) **\*** len(left)

right\_impurity **=** self**.**criterion(right) **\*** len(right)

**return** parent\_impurity **-** (left\_impurity**+**right\_impurity)**/**len(parent)

reg\_type **=** {"reg", "regression", "regressor"}

cls\_type **=** {"cls", "classification", "classifier"}

**def** \_\_init\_\_(self,

max\_depth:int**=**5,

max\_features:float**=**1.,

min\_samples\_split:int**=**2,

min\_samples\_leaf:int**=**1,

min\_split\_gain:float**=**0,

select\_thresholds:ThreshSelectFunction**=**"default",

aggregate:Union[str,AggregateFunction]**=**"regression",

criterion:Union[str,CriterionFunction]**=**"variance",

seed**=None**

):

self**.**random\_state **=** np**.**random**.**RandomState(seed)

self**.**root **=** **None**

self**.**max\_depth **=** numeric(max\_depth, "max\_depth", dtype**=**int, min**=**3)

self**.**max\_features **=** numeric(max\_features, "max\_features", dtype**=**float, min**=**0, max**=**1, min\_closed**=False**)

self**.**min\_samples\_leaf **=** numeric(min\_samples\_leaf, "min\_samples\_leaf", dtype**=**int, min**=**1)

self**.**min\_samples\_split **=** numeric(min\_samples\_split, "min\_samples\_split", dtype**=**int)

self**.**min\_split\_gain **=** numeric(min\_split\_gain, "min\_split\_gain", min**=**0.)

*# threshold selection function validation*

sel\_options **=** {

"default": **lambda** x: x *# no special threshold selection*

}

self**.**select\_thresholds **=** sel\_options**.**get(select\_thresholds, select\_thresholds)

**if** **not** callable(self**.**select\_thresholds):

**raise** **TypeError**(f"'select\_thresholds' parameter must be in {set(sel\_options**.**keys())} or 'callable(ndarray) -> ndarray'")

*# aggregate function validation*

aggr\_options **=** {

**\*\***{k:self**.**mean\_aggr **for** k **in** self**.**reg\_type}, *# mean aggregation for regression*

**\*\***{k:self**.**majority\_aggr **for** k **in** self**.**cls\_type} *# majority aggregation for classification*

}

self**.**aggregate **=** aggr\_options**.**get(aggregate, aggregate)

**if** **not** callable(self**.**aggregate):

**raise** **TypeError**(f"'aggregate' parameter must be in {self**.**reg\_type **|** self**.**cls\_type} or 'callable(ndarray) -> float'")

*# criterion function validation*

criterion\_options **=** {

"gini":self**.**gini,

"entropy":self**.**entropy,

"variance":self**.**variance,

}

self**.**criterion **=** criterion\_options**.**get(criterion,criterion)

**if** **not** callable(self**.**criterion):

**raise** **TypeError**(f"'criterion' parameter must be in {set(criterion\_options**.**keys())} or 'callable(ndarray) -> ndarray'")

**def** **build\_tree**(self, X, y, depth**=**0):

n\_samples, n\_features **=** X**.**shape

**if** n\_samples **>** self**.**min\_samples\_leaf **and** depth **<=** self**.**max\_depth:

feature\_idx, threshold, left, right, gain **=** self**.**get\_best\_split(X, y, n\_samples, n\_features) *# finding best split*

*# splitting if information gain greater than some parameter (default 0)*

**if** gain **>** self**.**min\_split\_gain:

left\_subtree **=** self**.**build\_tree(X[left], y[left], depth **+** 1)

right\_subtree **=** self**.**build\_tree(X[right], y[right], depth **+** 1)

**return** Node(feature\_idx, threshold, left\_subtree, right\_subtree)

*# leaf node if max depth reached, minimum samples in leaf, minimum samples in split*

*# aggregrating sample points into single point*

leaf\_data **=** self**.**aggregate(y)

**return** Node(data**=**leaf\_data)

**def** **get\_best\_split**(self, X, y, n\_samples, n\_features):

split **=** **None**

idx **=** np**.**arange(n\_samples)

best **=** float("-inf")

*# selecting features randomly*

sel\_features **=** self**.**random\_state**.**permutation(n\_features)[:int(round(n\_features**\***self**.**max\_features))]

*# iterating through features*

**for** feature\_idx **in** sel\_features:

feature **=** X[:,feature\_idx] *# getting current column of dataset*

thresh\_potential **=** self**.**select\_thresholds(np**.**unique(feature)) *# selecting potential thresholds*

*# iterating through thresholds to find best feature-threshold combination*

**for** threshold **in** thresh\_potential:

left **=** idx[feature **<=** threshold] *# getting indices of data points less than threshold for left subtree*

right **=** idx[feature **>** threshold] *# getting indices of data points greater than threshold for right subtree*

*# if minimal data after split skip execution and begin next iteration*

**if** len(left) **<=** self**.**min\_samples\_split **or** len(right) **<=** self**.**min\_samples\_split:

**continue**

*# calculating quality of split*

gain **=** self**.**information\_gain(y, y[left], y[right])

**if** gain **>=** best: *# update best split*

split **=** ( feature\_idx, threshold, left, right, gain )

best **=** gain

**return** split

**def** **fit**(self,X,y):

self**.**root **=** self**.**build\_tree(X,y)

**return** self

**def** **\_predict**(self,X,node):

*# if leaf node, return data*

**if** node**.**data **is** **not** **None**:

**return** node**.**data

*# recursively traversing down tree*

val **=** X[node**.**feature\_idx]

**if** val **<=** node**.**threshold: *# traverse left if value of feature in X less than threshold*

**return** self**.**\_predict(X,node**.**left)

**else**: *# otherwise traverse right*

**return** self**.**\_predict(X,node**.**right)

**def** **predict**(self,X):

**return** np**.**array([self**.**\_predict(x,self**.**root) **for** x **in** X])

#### Fitting and evaluating

**def** **cluster\_thresholds**(array, diff=0.005, aggr**=**np**.**mean):

*# aggregrating similar thresholds to reduce number of potential splits*

tmp **=** array**.**copy()

groups **=** []

**while** len(tmp):

*# select seed*

seed **=** tmp**.**min()

mask **=** (tmp **-** seed) **<=** diff

groups**.**append(aggr(tmp[mask, None]))

tmp **=** tmp[**~**mask]

**return** groups

fn **=** "models/DecisionTree.bin"

**if** **not** os**.**path**.**exists(fn):

reg **=** DecisionTree(

max\_depth**=**7, *# stop growing at depth 7*

min\_samples\_leaf**=**512, *# to speed up training*

aggregate**=**"regression", *# mean aggregrate samples at leaf node*

criterion**=**"variance", *# use variance reduction*

*# cluster and aggregate thresholds with 0.005 difference between them, categorical features should stay unaffected*

select\_thresholds**=**cluster\_thresholds,

seed**=**SEED

)**.**fit(tr\_X[:,1:], tr\_y2)

dump(reg, fn)

reg **=** ConformalRegression(load(fn))**.**calibrate(cal\_X[:,1:], cal\_y2)

MAE (81,266) is 36.66% of mean

RMSE (125,512) is 56.62% of mean

MAPE 28.47%

R2 Score is 0.657

Graphical user interface, chart

Description automatically generated

As seen above, the model produces discrete predictions based on the leaf nodes of the fitted tree. I believe increasing the maximum depth (thereby increasing the number of leaves) would help improve the bands of predictions that can be seen on the scatter diagram. However, this would be highly inefficient due to the limitations of my implementation. Therefore, I will instead try using an ensemble method to combine multiple trees using a library instead, which will solve many of the issues faced.

### Model selection – Random Forest

#### Hyperparameter optimisation

I will begin by optimising the random forest's parameters on a small set of data, with a low number of estimators, for quick and effective computation. The models will be optimised to minimise the RMSE (root mean squared error) over 100 iterations using the hyperopt library.

pbounds **=** {

"max\_depth": hp**.**uniformint("max\_depth", 5, 20), *# maximum depth to grow trees to, too high/low could result in overfitting/underfitting*

"max\_features": hp**.**uniform("max\_features", 0.5, 1.0), *# maximum number of features to consider when making splits*

"min\_samples\_split": hp**.**uniformint("min\_samples\_split", 1, 64), *# minimum number of samples required to split*

"min\_samples\_leaf": hp**.**uniformint("min\_samples\_leaf", 1, 64), *# minimum number of samples in each leaf node*

*# static parameters*

"n\_bins":256,

"n\_estimators": 50, *# number of trees to build. low number for now, will increase when fitting resulting model*

"random\_state": SEED *# for reproducibility, may still have some randomness due to n\_streams > 1.*

}

n **=** 100\_000

idx **=** np**.**random**.**permutation(len(tr\_X))[:n]

sample\_x, sample\_y **=** tr\_X[idx,1:]**.**astype(np**.**float32), tr\_y2[idx]**.**astype(np**.**float32)

fn **=** "models/\_rf\_trials.hyperopt"

trials **=** Trials() **if** **not** os**.**path**.**exists(fn) **else** load(fn)

result **=** fmin(

fn **=** objective(

model **=** RandomForestRegressor,

X **=** sample\_x,

y **=** sample\_y,

cv **=** 5,

),

space **=** pbounds,

algo **=** tpe**.**suggest,

max\_evals **=** 100,

trials **=** trials

)

**if** **not** os**.**path**.**exists(fn):

dump(trials, fn)

rfparams **=** {k:v[0] **for** k,v **in** trials**.**best\_trial["misc"]["vals"]**.**items()}

rfparams["max\_depth"] **=** int(rfparams["max\_depth"])

rfparams["min\_samples\_split"] **=** int(rfparams["min\_samples\_split"])

rfparams["min\_samples\_leaf"] **=** int(rfparams["min\_samples\_leaf"])

100%|██████████| 100/100 [00:00<?, ?trial/s, best loss=?]

best loss: 0.2328524947166443,

params: {'max\_depth': 20,

'max\_features': 0.9965555438979297,

'min\_samples\_leaf': 6,

'min\_samples\_split': 7}

Chart, scatter chart

Description automatically generated

#### Fitting and evaluating

fn **=** "models/RandomForest.bin"

**if** **not** os**.**path**.**exists(fn):

*# gpu accelerated training*

rf **=** RandomForestRegressor(

**\*\***rfparams,

n\_bins **=** 256,

n\_estimators **=** 100,

)**.**fit(tr\_X[:,1:]**.**astype(np**.**float32), tr\_y2**.**astype(np**.**float32))

*# serialising as treelite model to make available*

*# for cpu inference and cross-platform*

tl\_model **=** rf**.**convert\_to\_treelite\_model()

**del** rf

tl\_model**.**to\_treelite\_checkpoint(fn)

**else**:

tl\_model **=** treelite**.**Model**.**deserialize(fn)

reg **=** ConformalRegression(ForestInference()**.**load\_from\_treelite\_model(tl\_model, output\_class**=False**))**.**calibrate(cal\_X[:,1:], cal\_y2)

MAE (55,801) is 25.17% of mean

RMSE (90,258) is 40.71% of mean

MAPE 19.38%

R2 Score is 0.823

Chart

Description automatically generated

I am pleased with the performance of the random forest as it is not only versatile, but also extremely fast, as a result of the **treelite** compiler and the gpu-accelerated **ForestInference** front-end. I will now test gradient boosting machines (**XGBoost**) accelerated with treelite and ForestInference, to test if it can also perform with similar efficiency.

### Model selection – Gradient Boosting Machine (XGBoost)

#### Hyperparameter Optimisation

Like the random forest, I will begin by optimising the parameters of the gradient boosting machine using hyperopt.

pbounds **=** {

"learning\_rate": hp**.**uniform("learning\_rate", 0.01, 0.25),

"max\_depth": hp**.**uniformint("max\_depth", 5, 20), *# maximum depth to grow trees to, too high/low could result in overfitting/underfitting*

"colsample\_bynode": hp**.**uniform("colsample\_bynode", 0.5, 1.0), *# maximum number of features to consider when making splits*

"min\_child\_weight": hp**.**uniformint("min\_child\_weight", 1, 32), *# minimum number of samples in each leaf node*

*# static parameters*

"random\_state": SEED,

"n\_estimators": 50,

"tree\_method": "gpu\_hist" *# gpu acceleration*

}

n **=** 50\_000

idx **=** np**.**random**.**permutation(len(tr\_X))[:n]

sample\_x, sample\_y **=** tr\_X[idx, 1:]**.**astype(np**.**float32), tr\_y2[idx]**.**astype(np**.**float32)

fn **=** "models/\_xgb\_trials.hyperopt"

trials **=** Trials() **if** **not** os**.**path**.**exists(fn) **else** load(fn)

result **=** fmin(

fn **=** objective(

model **=** xgb**.**XGBRegressor,

X **=** sample\_x,

y **=** sample\_y,

cv **=** 2,

),

space **=** pbounds,

algo **=** tpe**.**suggest,

max\_evals **=** 100,

trials **=** trials

)

**if** **not** os**.**path**.**exists(fn):

dump(trials, fn)

xgbparams **=** {k:v[0] **for** k,v **in** trials**.**best\_trial["misc"]["vals"]**.**items()}

xgbparams["max\_depth"] **=** int(round(xgbparams["max\_depth"]))

xgbparams["min\_child\_weight"] **=** int(round(xgbparams["min\_child\_weight"]))

100%|██████████| 100/100 [00:00<?, ?trial/s, best loss=?]

best loss: 0.22684506326913834,

params: {'colsample\_bynode': 0.8708868938454073,

'learning\_rate': 0.2088532944217484,

'max\_depth': 11,

'min\_child\_weight': 17}

Chart, scatter chart

Description automatically generated with medium confidence

#### Fitting and evaluating

fn **=** "models/XGBoost.bin"

**if** **not** os**.**path**.**exists(fn):

dtrain **=** xgb**.**DMatrix(tr\_X[:,1:], tr\_y2)

dtest **=** xgb**.**DMatrix(val\_X[:,1:], val\_y2)

xgbparams["random\_state"] **=** SEED

xgbparams["tree\_method"] **=** "gpu\_hist"

xgb\_ **=** xgb**.**train(xgbparams, dtrain, evals**=**[(dtrain, "train"), (dtest, "val")], num\_boost\_round**=**2000, early\_stopping\_rounds**=**10)

xgb\_**.**save\_model(fn)

**else**:

xgb\_ **=** xgb**.**Booster()

xgb\_**.**load\_model(fn)

reg **=** ConformalRegression(ForestInference()**.**load(fn, output\_class**=False**))**.**calibrate(cal\_X[:,1:], cal\_y2)

MAE (54,270) is 24.48% of mean

RMSE (87,746) is 39.58% of mean

MAPE 18.80%

R2 Score is 0.832

Chart

Description automatically generated

As expected, the XGBoost model performed well, and marginally better than the Random Forest model. I will now attempt to implement and test a multilayer perceptron, a.k.a. neural network, completely from scratch, and see whether it can also perform to XGBoost’s level.

### Model selection – Multilayer perceptron

#### Implementation

I will be implementing the multilayer perceptron through a modular layer-based implementation.

*# algorithms/mlp/base.py*

NDArrayFunction **=** Callable[[ndarray], ndarray]

LossFunction **=** Callable[[ndarray, ndarray], ndarray]

WeightsInitFunction **=** Callable[[int, int], ndarray]

**class** **BaseOptimiser**(object):

**def** \_\_init\_\_(self):

*# self.reset()*

**pass**

**def** **reset**(self):

**raise** **NotImplementedError**()

**def** **update**(self, grad:ndarray, id:str) **->** ndarray:

**raise** **NotImplementedError**()

**class** **BaseLayer**(object):

**def** \_\_init\_\_(self, id:str**=None**):

self**.**id **=** id

*# self.reset()*

**def** **reset**(self):

self**.**inp **=** **None**

**def** **forward**(self, inp:ndarray) **->** ndarray:

**raise** **NotImplementedError**()

**def** **backward**(self, grad:ndarray, optimiser:BaseOptimiser) **->** ndarray:

**raise** **NotImplementedError**()

**class** **ActivationLayer**(BaseLayer):

**def** \_\_init\_\_(self, fn:NDArrayFunction, prime:NDArrayFunction, id:str**=None**):

super()**.**\_\_init\_\_(id)

self**.**fn **=** fn

self**.**prime **=** prime

**def** **forward**(self, inp):

self**.**inp **=** inp *# storing x for back propagation*

**return** self**.**fn(inp)

**def** **backward**(self, grad, optimiser): *# optimiser parameter for consistency, will not be used*

**return** self**.**prime(self**.**inp) **\*** grad *# hadamard product - applying chain rule*

##### Layers

Each layer type will be implemented with a forward and backward methods. The forward method will feed input through the network, applying the current layer's function on the previous' output. And the backward method will propagate the loss back through the network, updating trainable parameters along the way. I will only be implementing 2 layer types, the perceptron layer, with trainable weights, and the leaky ReLU non-linear activation function.

I will use Xavier or He initialisation for the weights:

While the leaky ReLU function is non-differentiable at 0, it is relatively rare in the context of deep learning and can thus be set arbitrarily, typically to *c* or 1.

*# algorithms/mlp/layers.py*

**class** **PerceptronLayer**(BaseLayer):

@staticmethod

**def** **xavier\_init**(n\_in, n\_out, seed**=None**):

np**.**random**.**seed(seed)

**return** np**.**random**.**normal(

loc **=** 0.0,

scale **=** np**.**sqrt(2**/**(n\_in **+** n\_out)),

size **=** (n\_in, n\_out),

)

@staticmethod

**def** **he\_init**(n\_in, n\_out, seed**=None**):

np**.**random**.**seed(seed)

**return** np**.**random**.**normal(

loc **=** 0.0,

scale **=** np**.**sqrt(2**/**n\_in),

size **=** (n\_in, n\_out),

)

**def** \_\_init\_\_(self, n\_in:int, n\_out:int, weights\_init:Union[str,WeightsInitFunction]**=**"he", add\_bias**=True**, id:str**=None**, seed**=None**):

super()**.**\_\_init\_\_(id)

self**.**seed **=** seed

self**.**n\_in **=** n\_in

self**.**n\_out **=** n\_out

self**.**add\_bias **=** add\_bias

options **=** {

"he":self**.**he\_init,

"kaiming":self**.**he\_init,

"xavier":self**.**xavier\_init,

"glorot":self**.**xavier\_init,

}

self**.**weights\_init **=** options**.**get(weights\_init,weights\_init)

**if** **not** callable(self**.**weights\_init):

**raise** **ValueError**(f"`weights\_init` must be a string in {set(options**.**keys())} or callable(n\_in, n\_out, seed):ndarray[n\_in, n\_out]")

self**.**reset()

**def** **reset**(self):

super()**.**reset()

*# initialise weights*

self**.**weights **=** self**.**weights\_init(self**.**n\_in, self**.**n\_out, self**.**seed)

self**.**bias **=** np**.**zeros((1,self**.**n\_out))

**def** **forward**(self, inp):

self**.**inp **=** inp *# storing input for back propagation*

**return** np**.**dot(self**.**inp, self**.**weights) **+** self**.**bias *# y = w.x + b*

**def** **backward**(self, grad, optimiser):

grad\_inp **=** np**.**dot(grad, self**.**weights**.**T) *# gradient of loss wrt input for backpropagation*

grad\_weights **=** np**.**dot(self**.**inp**.**T, grad) *# gradient of loss wrt weights*

grad\_bias **=** np**.**mean(grad, axis**=**0) *# gradient of loss wrt bias*

self**.**weights **-=** optimiser**.**update(grad\_weights, self**.**id**+**"weights")

**if** self**.**add\_bias:

self**.**bias **-=** optimiser**.**update(grad\_bias, self**.**id**+**"bias")

**return** grad\_inp

**class** **LeakyReLU**(ActivationLayer):

**def** \_\_init\_\_(self, alpha:float, id:str**=None**):

self**.**alpha **=** alpha

super()**.**\_\_init\_\_(self**.**fn, self**.**prime, id)

**def** **fn**(self, inp):

*# x >= 0: x*

*# x < 0: x \* alpha*

**return** np**.**maximum(inp, self**.**alpha**\***inp)

**def** **prime**(self, inp):

*# x >= 0: 1*

*# x < 0: alpha*

**return** np**.**where(inp **<** 0, self**.**alpha, np**.**ones\_like(inp))

##### Parameter optimisation

*# algorithms/mlp/optimisers.py*

**class** **AdamOptimiser**(BaseOptimiser):

**def** \_\_init\_\_(self, lr:float**=**0.01, beta1:float**=**0.9, beta2:float**=**0.999, eps:float**=**1e-8):

self**.**lr **=** lr

self**.**beta1 **=** beta1

self**.**beta2 **=** beta2

self**.**eps **=** eps

self**.**reset()

**def** **reset**(self):

self**.**cache\_m **=** dict()

self**.**cache\_v **=** dict()

self**.**cache\_t **=** dict()

**def** **update**(self, grad, id):

*# retrieve previous iteration variables*

m **=** self**.**cache\_m**.**get(id, 0)

v **=** self**.**cache\_v**.**get(id, 0)

t **=** self**.**cache\_t**.**get(id, 1)

*# update variables for current iteration*

self**.**cache\_m[id] **=** self**.**beta1 **\*** m **+** (1 **-** self**.**beta1) **\*** grad

self**.**cache\_v[id] **=** self**.**beta2 **\*** v **+** (1 **-** self**.**beta2) **\*** grad **\*\*** 2

*# bias-corrected variable*

m\_corrected **=** self**.**cache\_m[id] **/** (1 **-** self**.**beta1 **\*\*** t)

v\_corrected **=** self**.**cache\_v[id] **/** (1 **-** self**.**beta2 **\*\*** t)

self**.**cache\_t[id] **=** t **+** 1

*# delta calculation*

delta **=** self**.**lr **\*** m\_corrected **/** (np**.**sqrt(v\_corrected) **+** self**.**eps)

**return** delta

##### Neural network logic (feedforward, backpropagation, prediction)

*# algorithms/mlp/model.py*

from sklearn.base import BaseEstimator

**def** **mse**(y, yhat):

**return** np**.**mean(np**.**power(y**-**yhat, 2))

**def** **mse\_prime**(y, yhat):

**return** 2**\***(yhat**-**y)**/**y**.**size

**class** **NeuralNetwork**(BaseEstimator):

**def** \_\_init\_\_(self, layers:list[BaseLayer], optimiser:BaseOptimiser, loss\_fn:LossFunction**=**mse, loss\_prime:LossFunction**=**mse\_prime):

self**.**layers **=** layers

self**.**loss\_fn **=** loss\_fn

self**.**loss\_prime **=** loss\_prime

self**.**optimiser **=** optimiser

**def** **reset**(self):

*# resetting model components*

self**.**history **=** dict()

self**.**optimiser**.**reset()

**for** l **in** self**.**layers: l**.**reset()

**def** **predict**(self,X:ndarray) **->** ndarray:

out **=** np**.**asarray(X)

*# forward propagation*

**for** l **in** self**.**layers:

out **=** l**.**forward(out)

*# clearing cached layer inputs*

**for** layer **in** self**.**layers:

layer**.**inp **=** **None**

**return** out

**def** **backpropagate**(self, X:ndarray, y:ndarray) **->** float:

out **=** self**.**predict(X)

grad\_out **=** self**.**loss\_prime(y,out)

**for** l **in** self**.**layers[::**-**1]: *# propagate gradient backwards through network*

grad\_out **=** l**.**backward(grad\_out,self**.**optimiser)

**return** self**.**loss\_fn(y,out)

@staticmethod

**def** **batches**(X:ndarray, y:ndarray, batch\_size:int, shuffle:bool) **->** tuple[ndarray, ndarray]:

n **=** len(X)

*# random indices if shuffling enabled*

idx **=** np**.**random**.**permutation(n) **if** shuffle **else** np**.**arange(n)

**for** i **in** range(0,n,batch\_size):

batch **=** idx[i:i**+**batch\_size] *# taking indices from index list*

**yield** X[batch], y[batch] *# lazy evaluation*

**def** **fit**(self,

X:ndarray,

y:ndarray,

val\_X:ndarray **=** **None**,

val\_y:ndarray **=** **None**,

epochs:int **=** 1000,

batch\_size:int **=** 1,

early\_stopping:int **=** 20,

delta:float **=** 0.01,

shuffle:bool **=** **False**,

verbose:bool **=** **True**

):

**for** i, l **in** enumerate(self**.**layers):

**if** l**.**id **is** **None**:

l**.**id **=** f"layer{i}"

*# X,y argument validation*

X, y **=** np**.**asarray(X), np**.**asarray(y)

**if** X**.**ndim **==** 1 **or** y**.**ndim **==** 1:

**raise** **ValueError**("`X` and `y` arrays must be 2 dimensional")

self**.**history **=** {"loss":[]} *# initialising training loss history*

*# val\_X,val\_y argument validation*

**if** val\_X **is** **not** **None** **and** val\_y **is** **not** **None**:

val\_X, val\_y **=** np**.**asarray(val\_X), np**.**asarray(val\_y)

**if** val\_X**.**ndim **==** 1 **or** val\_y**.**ndim **==** 1:

**raise** **ValueError**("`val\_X` and `val\_y` arrays must be 2 dimensional")

self**.**history["val\_loss"] **=** [] *# initialising validation loss history*

patience **=** **None**

*# early\_stopping argument validation*

**if** isinstance(early\_stopping, int) **and** early\_stopping **>** 0:

patience **=** early\_stopping

best\_loss **=** np**.**inf *# intialising best loss for model improvement calculation*

**else**:

warnings**.**warn(f"`early\_stopping` must be positive non-zero integer not `{early\_stopping}`, defaulting to None")

**for** i **in** (pbar**:=**tqdm(range(epochs), disable**=not** verbose)):

*# end training if not improved*

**if** patience **is** **not** **None** **and** patience **<=** 0:

pbar**.**close()

**if** verbose:

print(f"Early stopping at epoch {i}")

**break**

epoch\_loss **=** 0

*# split training set into batches for less intensive computation*

**for** Xb, yb **in** self**.**batches(X,y,batch\_size,shuffle):

epoch\_loss **+=** self**.**backpropagate(Xb,yb) **\*** len(Xb) *# back propagating model and summing error for batches*

epoch\_loss **/=** len(X) *# taking mean error across all batches*

self**.**history["loss"]**.**append(epoch\_loss)

*# testing model on validation set if provided*

**if** "val\_loss" **in** self**.**history:

self**.**history["val\_loss"]**.**append(self**.**loss\_fn(self**.**predict(val\_X), val\_y))

*# logging epoch info if enabled*

**if** verbose:

msg **=** {"loss": self**.**history["loss"][**-**1]}

**if** "val\_loss" **in** self**.**history:

msg["val\_loss"] **=** self**.**history["val\_loss"][**-**1]

**if** patience **is** **not** **None**:

msg["patience"] **=** patience

pbar**.**set\_postfix(msg)

*# skip to next iteration if early stopping is disabled*

**if** patience **is** **None**:

**continue**

*# if model has improved greater than minimum delta*

**if** best\_loss **-** epoch\_loss **>=** delta:

patience **=** early\_stopping *# reset patience*

best\_loss **=** epoch\_loss

**continue** *# skip to next iteration*

patience **-=** 1 *# decrement patience (code wont be reached if model improved)*

*# clearing cached layer inputs*

**for** layer **in** self**.**layers:

layer**.**inp **=** **None**

**return** self

#### Hyperparameter Optimisation

from algorithms.mlp.layers import PerceptronLayer, LeakyReLU

from algorithms.mlp.optimisers import AdamOptimiser

from algorithms.mlp.model import NeuralNetwork

**def** **build\_mlp**(hidden\_layers, n\_units, lrelu\_alpha):

layers **=** [

PerceptronLayer(len(X**.**columns), n\_units, seed**=**SEED),

LeakyReLU(lrelu\_alpha)

]

**for** i **in** range(hidden\_layers**-**1):

layers **+=** [

PerceptronLayer(n\_units, n\_units, seed**=**SEED),

LeakyReLU(lrelu\_alpha)

]

layers**.**append(PerceptronLayer(n\_units, 1, add\_bias**=False**))

**return** NeuralNetwork(layers, AdamOptimiser())

pbounds **=** {

"hidden\_layers": hp**.**uniformint("hidden\_layers", 1, 4),

"n\_units": hp**.**uniformint("n\_units", 24, 56),

"lrelu\_alpha": hp**.**uniform("lrelu\_alpha", 0, 0.5),

}

n **=** 100\_000

idx **=** np**.**random**.**permutation(len(tr\_X))[:n]

sample\_x, sample\_y **=** tr\_X[idx]**.**astype(np**.**float32), tr\_y[idx]**.**astype(np**.**float32)

fn **=** "models/\_mlp\_trials.hyperopt"

trials **=** Trials() **if** **not** os**.**path**.**exists(fn) **else** load(fn)

**with** warnings**.**catch\_warnings():

warnings**.**simplefilter("ignore")

result **=** fmin(

fn **=** objective(

model **=** build\_mlp,

X **=** sample\_x,

y **=** sample\_y**.**reshape(**-**1,1),

cv **=** 5,

fit\_params **=** {

"early\_stopping": 5, *# speed up training*

"batch\_size": 1024,

"verbose": **False**,

}

),

space **=** pbounds,

algo **=** tpe**.**suggest,

max\_evals **=** 100,

trials **=** trials

)

**if** **not** os**.**path**.**exists(fn):

dump(trials, fn)

nnparams **=** {k:v[0] **for** k,v **in** trials**.**best\_trial["misc"]["vals"]**.**items()}

nnparams["hidden\_layers"] **=** int(round(nnparams["hidden\_layers"]))

nnparams["n\_units"] **=** int(round(nnparams["n\_units"]))

100%|██████████| 100/100 [10:48<00:00, 6.49s/trial, best loss: 0.25280905636164314]

best loss: 0.25280905636164314,

params: {'hidden\_layers': 2, 'lrelu\_alpha': 0.04031941259789426, 'n\_units': 35}

Chart, scatter chart

Description automatically generated

#### Fitting and evaluating

fn **=** "models/MultilayerPerceptron.bin"

**if** **not** os**.**path**.**exists(fn):

nnparams["hidden\_layers"] **=** 3 *# 3-5 layers were around the same*

reg **=** build\_mlp(**\*\***nnparams)**.**fit(

tr\_X,

tr\_y**.**reshape(**-**1,1),

val\_X **=** val\_X,

val\_y **=** val\_y**.**reshape(**-**1,1),

epochs **=** 1000,

batch\_size **=** 1024,

early\_stopping **=** 25,

delta**=**0.005,

)

dump(reg, fn)

reg **=** ConformalRegression(load(fn))**.**calibrate(cal\_X, cal\_y**.**reshape(**-**1,1))

7%|▋ | 72/1000 [13:15<2:50:51, 11.05s/it, loss=0.161, val\_loss=0.161, patience=1]

Early stopping at epoch 72

I will now plot the performance of the model as training progressed, to potentially spot any irregularities or issues.

Chart, line chart

Description automatically generated

Apart from the occasional spike, the loss exponentially decreased and began to converge, therefore the model has been successfully trained.

MAE (41,696) is 18.81% of mean

RMSE (65,708) is 29.64% of mean

MAPE 23.26%

R2 Score is 0.838

Chart

Description automatically generated

The from-scratch multilayer perceptron model performed better than expected and outperformed the XGBoost and Random Forest. This model was a great success, and I will now be comparing all the tested models against each other and finally making a judgement as to what to use in the final app.

### Conclusion

|  | **r2** | **mape** |
| --- | --- | --- |
| **LinearRegression** | 0.479313 | 47.352439 |
| **KNearestNeighbours(ADJ)** | 0.767500 | 21.275510 |
| **DecisionTree(ADJ)** | 0.657260 | 28.470305 |
| **RandomForest(ADJ)** | 0.822758 | 19.384993 |
| **XGBoost(ADJ)** | 0.832487 | 18.801916 |
| **MultilayerPerceptron** | 0.838236 | 23.260539 |

Chart, bar chart

Description automatically generated

The XGBoost model slightly outperforms the Multilayer Perceptron model. However, since the XGBoost was imported via a library, and the MLP was implemented from scratch, I will export the MLP as the final model. Moreover, the XGBoost model is limited to predictions within its observed set, i.e. cannot extrapolate, whereas the MLP can; therefore, I believe the Multilayer Perceptron to be ideal in this case.

SOURCE **=** "models/MultilayerPerceptron.bin"

DESTINATION **=** os**.**path**.**join(SERVER\_DIR, "model.bin")

os**.**system(f"cp {SOURCE} {DESTINATION}");

I will now also pre-calibrate and export the calibration scores to the server directory. By calibrating beforehand, I will not need to calibrate the model in the server, avoiding any intensive computation on the deployment server which would use resources not available for free.

calibrated **=** ConformalRegression(load(SOURCE))**.**calibrate(cal\_X, cal\_y**.**reshape(**-**1,1))

dump(calibrated**.**resid\_, os**.**path**.**join(SERVER\_DIR, "conf\_resid.bin"));

In conclusion, I have chosen to use the MLP model alongside conformal prediction techniques as the predictive model that will power my house price valuation tool. Finally, I need to implement and deploy the front-end app that will act to serve these predictions.

## Implementation – HTML web-app and deployment

*# base/stage-2-webapp/main.py*

**import** sys

**from** os **import** path

ROOT\_DIR = path.dirname(path.dirname(path.abspath(\_\_file\_\_)))

sys.path.append(ROOT\_DIR)

**import** json

**from** flask **import** Flask, request, redirect, url\_for, jsonify, render\_template

**from** werkzeug.exceptions **import** HTTPException

**from** model **import** build\_model *# base/stage-2-webapp/model/\_\_init\_\_.py*

**import** utils *# base/stage-2-webapp/utils.py*

app = Flask(\_\_name\_\_) *# entrypoint for gunicorn webapp*

model\_ = build\_model() *# loading model*

*# compiling sass files into css to serve in static site*

**if**(app.debug):

**from** sass **import** compile

compile(dirname=("static/sass", "static/css"))

*# handling unexpected exceptions*

@app.errorhandler(Exception)

**def** handle\_exception(e):

**if** isinstance(e, (HTTPException)):

error = e.description

**elif** app.debug:

error = str(e)

**else**:

error = "An unexpected error occurred!"

**return** redirect(url\_for("render\_landing", error=error))

@app.route("/api/predict")

**def** predict():

args = request.args.to\_dict() *# extracting URL arguments*

redirect\_ = args.pop("redirect", "false").lower()

alpha = args.pop("alpha", 0.45) *# prediction interval alpha*

postcode = args.pop("postcode")

*#* *fetching data about postcode using findthatpostcode.uk API*

pcd\_data = utils.findthatpostcode(postcode)

args = dict(\*\*args, \*\*pcd\_data["location"]) *# combining model arguments*

yhat = model\_.predict(args, alpha) *# making prediction*

**if** redirect\_: *#* *redirecting to results page if redirect parameter true*

**return** redirect(url\_for("render\_result", pcd\_data=pcd\_data, \*\*yhat))

**return** jsonify(yhat) *# otherwise sending JSON response.*

@app.route("/")

**def** render\_landing(): *# landing page without modal window*

**return** render\_template("index.jinja", error=request.args.get("error"))

@app.route("/r")

**def** render\_result(): *# modal window to output results*

**if** (set(request.args.keys()) != {"high","point","low","pcd\_data"}):

**return** redirect(url\_for("render\_landing")) *# redirect to landing if data missing*

pcd\_data = json.loads(request.args.get("pcd\_data").replace("'",'"'))

high = float(request.args.get("high"))

point = float(request.args.get("point"))

low = float(request.args.get("low"))

**return** render\_template("index.jinja", *# render landing with modal overlay*

pcd\_data=pcd\_data,

high=utils.formatNumber(high, 3),

point=f'{int(float(f"{point:.4g}")):,}',

low=utils.formatNumber(low, 3),

modal**=True**)

*# base/stage-2-webapp/utils.py*

**import** requests

**from** werkzeug.exceptions **import** BadRequest

*# Utility function to format big numbers*

*# 128,956 => 129k, 1,245,234 => 1.25M*

**def** formatNumber(n, precision):

ranges = ((1e6, "M"), (1e3, "k")) *# sorted greatest prefix -> smallest prefix*

**for** r **in** ranges:

**if** (n **>=** r[0]): *# is the number greater than the current greatest range*

**return** f"{n**/**r[0]:.{precision}g}{r[1]}"

**return** n *# return full number if less than lowest range*

*# findthatpostcode API*

**def** findthatpostcode(postcode):

*# fetching postcode information from API*

response = requests.get(f"https://findthatpostcode.uk/postcodes/{postcode}.json")

**if**(response.status\_code **!=** 200): *# response status code is not OK*

**raise** BadRequest(f"Invalid postcode: {postcode}")

extract = ["location","laua\_name","ward\_name","oac11","imd","pcd"]

data = {k:response.json()["data"]["attributes"][k] **for** k **in** extract} *# extracting into dictionary*

data["location"]["lng"] = data["location"].pop("lon") *# renaming lon to lng*

**return** data

*# base/stage-2-webapp/model/\_\_init\_\_.py*

**import** sys, os, json

*# When loading the MLP model, joblib will look for an algorithms module with a NeuralNetwork class to deserialize the model.* ***See page 88 for MLP implementation.***

**from** . **import** algorithms

sys.modules["algorithms"] = algorithms

**import** joblib

**from** numpy **import** array

**from** .algorithms.conformal **import** ConformalRegression *#* ***See page 71 for ConformalRegression implementation***

**from** .predictor **import** Predictor *# base/stage-2-webapp/predictor.py*

FEATURES = {"year","lat","lng","total\_floor\_area","number\_habitable\_rooms","property\_type"}

*# path to data folder: base/stage-2-webapp/data/\**

DATA\_PATH = os.path.join(os.path.dirname(os.path.realpath(\_\_file\_\_)), "data")

MODEL\_PATH = os.path.join(DATA\_PATH, "model.bin") *# model binary file*

ENCODING\_PATH = os.path.join(DATA\_PATH, "encoding.json") *# category encoding json file*

SCALING\_PATH = os.path.join(DATA\_PATH, "scaling.json") *# scaling factors json file*

RESID\_PATH = os.path.join(DATA\_PATH, "conf\_resid.json") *# conformal prediction residual scores*

**def** xtransform(scaling): *# function builder for z-score normalisation of input*

**def** func(inp):

inp\_ = inp.copy()

for k, v in inp\_.items():

feat = scaling.get(k, {"mean": 0., "std": 1.})

*# rescaling features to get normalised values*

inp\_[k] = v **-** feat["mean"] **/** feat["std"]

**return** array([list(inp\_.values())]) *# array shape must be 2d like [[a,b,c,...]]*

**return** func

**def** ytransform(output\_scale): *# function builder to reverse z-score normalisation of outputs*

**def** func(inp):

*# reversing normalisation*

**return** inp **\*** output\_scale["std"] **+** output\_scale["mean"]

**return** func

*# loading encoding and scaling factors*

**def** load\_json\_files():

encoding = json.load(open(ENCODING\_PATH, "r"))

scaling = json.load(open(SCALING\_PATH, "r"))

output\_scale = scaling.pop("price") *# separating price scaling factors from features*

**return** encoding, scaling, output\_scale

*# building model and loading files*

**def** build\_model():

encoding, scaling, output\_scale = load\_json\_files()

conf = ConformalRegression(joblib.load(MODEL\_PATH))

conf.resid\_ = joblib.load(RESID\_PATH)

**return** Predictor(

conf,

FEATURES,

encoding,

xtransform(scaling),

ytransform(output\_scaling)

)

*# base/stage-2-webapp/model/predictor.py*

**from** werkzeug.exceptions **import** BadRequest

**class** **Predictor**(object):

**def** \_\_init\_\_(self,

model,

features,

encoding=**None**,

xtransform=lambda x: x,

ytransform=lambda x: x):

self.model = model

self.features = set(features)

self.encoding = encoding

self.xtransform = xtransform

self.ytransform = ytransform

**def** encode(self, inp):

**if** self.encode **is** **None**:

**return** inp

inp\_ = inp.copy()

**for** feat, encode **in** self.encoding.items():

**try**:

inp\_[feat] = encode[inp\_[feat]] *# encoding category value*

**except** KeyError:

*# unknown field so raise an error*

**raise** BadRequest(f"Invalid value for feature '{feat}': {inp[feat]}")

**return** inp\_

**def** predict(self, inp, alpha=**None**):

*# checking for missing or unknown features*

unknown = inp.keys() - self.features *# X intersection F*

missing = self.features - inp.keys() *# F intersection X*

**if** len(unknown) **>** 0: *# unknown features so raise an error.*

**raise** BadRequest(f"Unknown fields: {list(unknown)}")

**if** len(missing) **>** 0: *# missing features so raise an error.*

**raise** BadRequest(f"Missing fields: {list(missing)}")

inp\_ = self.encode(inp) *# encoding categories*

*# converting and validating encoded datatypes are numerical*

**for** k, v **in** inp\_.items():

**try**:

inp\_[k] = float(v) *# converting to float*

**except** Exception: *# unexpected datatype so raise an error*

**raise** BadRequest(f"Invalid datatype: {k}")

inp\_ = self.xtransform(inp\_) *# normalising input*

*# prediction interval using conformal regression if alpha specified*

**if** alpha **is** **not** **None**:

**try**:

yhat, interval = [

self.ytransform(y).reshape(-1) *# reshaping to 1d array*

**for** y **in** self.model.predict(inp\_, alpha=alpha)

]

**return** {"low": interval[0], "point": yhat[0], "high": interval[1]}

**except** TypeError:

**pass** *# alpha parameter not in predict function so return point prediction*

*# otherwise point prediction*

yhat = self.ytransform(self.model.predict(inp\_))

**return** {"point": yhat[0]}

{# base/stage2-webapp/templates/index.jinja #}

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <link rel="stylesheet" type="text/css" href="static/css/styles.css" />

    <script src="https://code.jquery.com/jquery-3.6.1.min.js"

        integrity="sha256-o88AwQnZB+VDvE9tvIXrMQaPlFFSUTR+nldQm1LuPXQ=" crossorigin="anonymous"></script>

    <title>ML House Valuation</title>

</head>

<body class="{{'shake' if error}}" style="{{'overflow: hidden;' if modal}}">

    <div class="card">

        <div class="card\_\_hero">

            <img

                src="https://images.unsplash.com/photo-1540663580421-b05a3e7099ca?ixlib=rb-1.2.1&ixid=MnwxMjA3fDB8MHxwaG90by1wYWdlfHx8fGVufDB8fHx8&auto=format&fit=crop&w=1206&q=80" />

            <h1 class="heading\_\_text">

                Get an instant valuation today!

            </h1>

        </div>

        <form id="form" action="/api/predict">

            <input hidden name="redirect" value="true" />

            <div class="form\_\_group">

                <input required class="form\_\_field" placeholder=" " id="postcode" name="postcode" />

                <label for="postcode" class="form\_\_label">Postcode</label>

            </div>

            <div class="form\_\_group">

                <input required type="number" class="form\_\_field" placeholder=" " id="year" name="year" min="1995"

                    value="2022" max="2035" step="any" />

                <label for="year" class="form\_\_label">Year</label>

            </div>

            <div class="form\_\_group">

                <select class="form\_\_field" id="property\_type" name="property\_type" required>

                    <option hidden> </option>

                    <option value="Detached house">Detached house</option>

                    <option value="Semi-detached house">Semi-detached house</option>

                    <option value="Terrace house">Terrace house</option>

                    <option value="Detached bungalow">Detached bungalow</option>

                    <option value="Semi-detached bungalow">Semi-detached bungalow</option>

                    <option value="Terrace bungalow">Terrace bungalow</option>

                    <option value="Flat">Flat</option>

                    <option value="Maisonette">Maisonette</option>

                </select>

                <label for="property\_type" class="form\_\_label">Property type</label>

            </div>

            <div class="form\_\_group">

                <input required type="number" class="form\_\_field" placeholder=" " id="total\_floor\_area"

                    name="total\_floor\_area" min="40" step="any" />

                <label for="total\_floor\_area" class="form\_\_label">Total floor area

                    <small>(m<sup>2</sup>)</small></label>

            </div>

            <div class="form\_\_group">

                <input required type="number" class="form\_\_field" placeholder=" " id="number\_habitable\_rooms"

                    name="number\_habitable\_rooms" min="1" />

                <label for="number\_habitable\_rooms" class="form\_\_label">Number of habitable rooms</label>

            </div>

            <div class="action">

                <input type="submit" class="action\_\_btn" value="How much does my house cost?">

            </div>

        </form>

    </div>

    {% if error %}

    <div class="error">

        {{error}}

    </div>

    <script>

        window.scrollTo({

            top: document.body.scrollHeight,

            behavior: 'smooth'

        })

    </script>

    {%endif%}

    {% if modal %}

    <div class="modal" style="overflow: auto;">

        <div class="card" style="padding-top:2rem;">

            <a id="modal\_\_close" class="modal\_\_close" href="{{ url\_for('render\_landing') }}">Close</a>

            <div>

                <h1>{{pcd\_data.ward\_name}}</h1>

                <span style="font-size: 1.25rem;">{{pcd\_data.laua\_name}}</span>

            </div>

            <div class="card\_\_hero">

                <div id="map" style="aspect-ratio:5/3;"></div>

            </div>

            <h2>Our prediction for {{pcd\_data.pcd}}</h2>

            <div class="prediction">

                <span>£{{low}}</span>

                <span class="prediction\_\_middle">£{{point}}</span>

                <span>£{{high}}</span>

            </div>

            <br>

            <h2>Area classifications</h2>

            <h3>Output area classification (2011)</h3>

            <span>{{pcd\_data.oac11.supergroup}}<br>

                &emsp;&rdca; {{pcd\_data.oac11.group}}<br>

                &emsp;&emsp;&rdca; {{pcd\_data.oac11.subgroup}} <code>({{pcd\_data.oac11.code}})</code>

            </span>

            <br>

            {% if pcd\_data.imd %}

            <h3>Indices of deprivation (2019)</h3>

            <span><strong>{{"{:,}".format(pcd\_data.imd)}}</strong> out of 32,844 in England <small>(where 1 is the

                    most deprived)</small></span>

            <span><strong>{{(pcd\_data.imd/32844\*100)|round|int}}%</strong> of LSOAs are more deprived than this

                one.</span>

            {%endif%}

        </div>

    </div>

    <script>

        const center = {{ pcd\_data.location }};

        let map;

        window.initMap = function () {

            map = new google.maps.Map(document.getElementById('map'), {

                center,

                disableDefaultUI: true,

                zoom: 16

            });

            new google.maps.Marker({

                position: center,

                map,

            });

        };

        $(".modal\_\_close").one("click", function (e) {

            // animate modal closing then "redirect" to landing page

            $(".modal")

                .removeClass("modal")

                .addClass("modal\_\_exiting")

                .one("animationend", () => {

                    e.target.click();

                });

            return false;

        });

    </script>

    <script async

        src="https://maps.googleapis.com/maps/api/js?key=AIzaSyDn19axHSHB9IAFXQqkUMc2ktX15ILlBpw&callback=initMap">

        </script>

    {%endif%}

</body>

</html>

/\* base/stage-2-webapp/static/sass/styles.scss \*/

@import url('https://fonts.googleapis.com/css2?family=DM+Sans:wght@400;500;700&display=swap');

$text-colour: #585563;

$primary: #3D9985;

$secondary: #F6FEDB;

$grey: #8597a3;

body {

  font-family: "DM Sans", sans-serif;

  color: $text-colour;

  background-color: #f1f3fb;

}

.card {

  margin: auto;

  margin-bottom: 4px;

  display: flex;

  flex-direction: column;

  max-height: 100%;

  max-width: 480px;

  background-color: white;

  border-radius: 10px;

  padding: 1rem;

  position: absolute;

  top: 50%;

  left: 50%;

  transform: translate(-50%, -50%);

}

.error {

  @extend .card;

  margin-bottom: 1rem;

  background-color: #fed7d7;

  color: #c53030;

  border: 1px solid #c53030;

}

.card\_\_hero {

  position: relative;

  border-radius: 10px;

  overflow: hidden;

  & > img {

    height: 100%;

    width: 100%;

    object-fit: cover;

  }

  & > h1 {

    position: absolute;

    top: 20%;

    max-width: 400px;

    right: 2rem;

    font-size: 1.75rem;

    @media screen and (max-width: 768px) {

      min-width: 80%;

      left: 50%;

      transform: translateX(-50%);

      text-align: center;

    }

  }

}

#map {

  border-radius: 10px;

  margin: .5rem 0;

}

p {

  line-height: 1.222;

}

h1, h2, h3 {

  @extend p;

  margin-block: 0;

}

h1 {

  font-size: 1.5rem;

  font-weight: 700;

}

h2 {

  font-size: 1.25rem;

  font-weight: 700;

}

.form\_\_group {

  position: relative;

  padding: 15px 0 0;

  margin-top: 0.75rem;

  &:hover {

    > .form\_\_label {

      transition: 0.25s ease;

      color: $primary;

    }

  }

}

.form\_\_field {

  width: 100%;

  border: 0;

  border-bottom: 2px solid $grey;

  outline: 0;

  font-size: 1rem;

  padding: 7px 0;

  background: transparent;

  transition: border-color 0.25s;

  &::placeholder {

    color: transparent;

  }

  &:placeholder-shown ~ .form\_\_label{

    cursor: text;

    top: 1em;

  }

  &:focus {

    ~ .form\_\_label {

      position: absolute;

      top: 0;

      display: block;

      transition: 0.25s ease;

      font-size: 1rem;

      color: $primary;

      font-weight:700;

    }

    padding-bottom: 6px;

    border-width: 3px;

    border-image: linear-gradient(to right, $primary, $secondary);

    border-image-slice: 1;

  }

}

select.form\_\_field:invalid ~ .form\_\_label{

  cursor: text;

  top: 1em;

}

.form\_\_label {

  position: absolute;

  top: 0;

  display: block;

  transition: 0.25s ease;

  color: $grey;

}

.form\_\_field{

  &:required,&:invalid { box-shadow:none; }

}

.center {

  position: absolute;

  top: 50%;

  left: 50%;

  transform: translate(-50%, -50%);

}

.action {

  margin-top: 2rem;

}

.action\_\_btn {

  font: inherit;

  padding: 1em;

  width: 100%;

  font-weight: 500;

  border-radius: 6px;

  color: white;

  border: 0;

  $gradient:  linear-gradient(120deg, $primary 25%, $secondary 125%);

  background-image: $gradient;

  background: $gradient;

  background-size: 200% 100%;

  background-position: 0 0;

  transition: background-position .5s;

  &:focus {

    outline: 0;

  }

  &:hover {

    background-position: 100% 0;

  }

}

@keyframes shake\_\_animate {

  0% {

    transform: rotate(-.5deg);

  }

  50% {

    transform: rotate(.5deg);

  }

}

.shake {

  animation: 0.2s ease 0s 2.5 shake\_\_animate;

}

@keyframes backdrop\_\_animate {

  0% {

    background-color: transparent;

    backdrop-filter: blur(0px);

  }

  100% {

    background-color: rgba(0, 0, 0, 0.3);

    backdrop-filter: blur(2px);

  }

}

@keyframes modal\_\_animate {

  0% {

    opacity: 0;

    transform: translate(-50%, 0%);

  }

  33% {

    opacity: 0;

  }

  100% {

    opacity: 100;

    transform: translate(-50%, -50%);

  }

}

.modal {

    position: fixed;

    animation: 0.5s ease 0s 1 backdrop\_\_animate;

    background-color: rgba(0, 0, 0, 0.3);

    backdrop-filter: blur(2px);

    top: 0;

    right: 0;

    bottom: 0;

    left: 0;

    z-index: 999;

    & > div {

      animation: 2s ease 0s 1 modal\_\_animate;

    }

}

.modal\_\_exiting {

  position: fixed;

  animation: 0.5s ease 0s 1 backdrop\_\_animate reverse;

  background-color: transparent;

  top: 0;

  right: 0;

  bottom: 0;

  left: 0;

  z-index: 999;

  & > div {

    animation: 2s ease 0s 1.5 modal\_\_animate reverse;

    opacity: 0;

  }

}

.modal\_\_close {

    font: inherit;

    color: #bbb;

    font-size: 90%;

    line-height: 30px;

    width: 70px;

    position: absolute;

    top: 0;

    right: 0;

    text-align: center;

    text-decoration: none;

    &:hover {

      transition: 0.25s ease;

      color: #888;

      font-weight: 500;

    }

}

.prediction {

  display: flex;

  margin: auto;

  justify-content: space-between;

  align-items: center;

  position: relative;

  width: 100%;

  z-index: 0;

  & > span {

    background-color: white;

    padding: 0.5rem 1rem;

  }

}

.prediction\_\_middle {

  font-size: 1.25rem;

  font-weight: 700;

}

.prediction\_\_middle::before {

    background-color: #bbb;

    content: '';

    display: block;

    height: 1px;

    position: absolute;

    left: 0;

    top: 50%;

    width: 100%;

    z-index: -1;

}

# base/stage-2-webapp/app.yaml

# Use "gcloud app deploy" command for Google App Engine deployment

runtime: python39

entrypoint: gunicorn -n :$PORT main:app

handlers:

  - url: /static

    static\_dir: static

  - url: /.\*

    script: auto

instance\_class: F1

automatic\_scaling:

  max\_instances: 1

# Testing

## Test plan

1. Data testing
   1. Raw data volume – 5 million+ rows, 10+ columns
   2. Category encoding – has the categorical data been encoded into numeric format for use with a prediction model?
   3. Cardinality reduction – is the data dimension appropriate and have unnecessary columns been removed?
   4. Outlier removal – have significant outliers been successfully removed?
   5. Data rescaling – has the data been scaled such that features have comparable distributions?
   6. Final data volume – 2 million+ training columns, 5+ training features
2. Model testing (for each trained model)
   1. Storage size < 2MB
   2. R2 score > 0.8 – R2 score is a measure of how well a prediction model explains the data. An R2 score of 0.8 indicates that the model explains 80% of the output data’s variability (the data’s spread/scatter).
3. Backend testing
   1. Initial asset loading – does the model build and load correctly and are the exported JSON assets imported correctly?
   2. Findthatpostcode.uk API – am I able to leverage the data usefully?
   3. Prediction API feature encoding – are the inputs encoded correctly?
      1. Test on valid inputs
      2. Test on unknown value for category field
   4. Prediction API feature scaling – are the inputs rescaled correctly (tested against manually scaled features)?
   5. Prediction API error handling – are missing or unknown fields recognised and handled with an appropriate error message?
      1. Missing fields
      2. Unknown fields
      3. Invalid datatypes
   6. Conformal prediction – does it work with different alpha values including alpha=0?
   7. Backend service deployment – does the backend API produce results quickly (response time < 1000ms) both locally and over the internet?
4. Frontend testing
   1. Design goal – does the final UI look like than the initial design intended for the project?
   2. Google Maps API – does the app successfully use Google Maps to visualise the given postcode?
   3. Error handling – are error messages displayed to the user?
   4. Responsive – does the webpage fit to different screen sizes (mobile, tablet, laptop, etc.)?
   5. Frontend service deployment – has the app been deployed correctly and is the webapp accessible over the internet?
   6. Loading time – does the webpage render quickly (<1500ms)?
5. Comparing results with existing systems – a successful prediction should produce a result around the same as Zoopla.

### Tests and fixes

|  |  |  |
| --- | --- | --- |
| Test | Details | Status |
| 1A | Raw data volume:  5,566,962 rows, 15 columns (including new columns added in Exploratory Data Analysis) | *Passed* |
| 1B | *See page 60*  Dataset encoded into numeric values. | *Passed* |
| 1C | *See page 62*  Cardinality reduced from 15 columns to 6. Categories within columns also significantly reduced. | *Passed* |
| 1D | *See page 65*  Appropriate outliers removed but significantly reduced data volume (600k rows deleted) | *Passed* |
| 1E | *See page 70, Figure 35*  Data normalised using z-score normalisation. | *Passed* |
| 1F | Final data volume:  4,903,637 rows, 8 columns (6 features, 2 output columns) | *Passed* |

|  |  |  |  |
| --- | --- | --- | --- |
| Model name | Test 2B – Storage < 2MB? | Test 2C – r2 score > 0.8? | page |
| Linear regression | 8.4kB – *Passed* | 0.47931 – *Failed* | 73 |
| K-nearest neighbours | 300.1MB – *Failed* | 0.76750 – *Failed* | 76 |
| Decision tree | 22.5kB – *Passed* | 0.65726 – *Failed* | 81 |
| Random forest | 2.3GB – *Failed* | 0.82276 – *Passed* | 84 |
| XGBoost | 30.4MB – *Failed* | 0.83249 – *Passed* | 86 |
| Multilayer perceptron | 72.1kB – *Passed* | 0.83824 – *Passed* | 94 |

**Test script:**

# base/stage2-webapp/tests.py

from model import load\_json\_files, build\_model

import requests

from werkzeug.exceptions import BadRequest

from pprint import pprint

from model.algorithms.conformal import ConformalRegression

from model.predictor import Predictor

from model import build\_model, xtransform

from numpy import ndarray

def test3a():

    encoding: dict

    scaling: dict

    encoding, scaling, \_= load\_json\_files()

    conf:ConformalRegression = build\_model().model

    if(encoding): print("Encoding file loaded successfully...")

    else: print("Encoding file loaded unsucessfully...")

    if(scaling): print("Scaling file loaded successfully...")

    else: print("Scaling file loaded unsucessfully...")

    if(isinstance(conf.resid\_, ndarray)): print("Residual scores loaded successfully...")

    else: print("Residual scores loaded unsucessfully...")

    if(conf.model): print("Model built successfully...")

    else: print("Model built unsucessfully...")

def test3b():

    postcode = "PR1 4HD"

    response = requests.get(f"https://findthatpostcode.uk/postcodes/{postcode}.json")

    if(response.status\_code != 200): raise BadRequest(f"Invalid postcode: {postcode}")

    extract = ["location","laua\_name","ward\_name","oac11","imd","pcd"]

    data = {k:response.json()["data"]["attributes"][k] for k in extract}

    data["location"]["lng"] = data["location"].pop("lon")

    pprint(data)

def test3c(model: Predictor):

    input = dict(year=2023, lat=52, lng=0, total\_floor\_area=120, number\_habitable\_rooms=4, property\_type="Semi-detached house")

    errorInput = dict(year=2023, lat=52, lng=0, total\_floor\_area=120, number\_habitable\_rooms=4, property\_type="Unexpected")

    pprint(model.encode(input))

    try:

        pprint(model.encode(errorInput))

    except Exception as e: print(e)

def test3d():

    scaling = {

        "feature1": { "mean": 10., "std": 2. },

        "feature2": { "mean": 3., "std": 0.5 },

        "feature3": { "mean": 100., "std": 32. },

        "feature4": { "mean": 4.5, "std": 2. },

    }

    features = dict(feature1=20., feature2=4.5, feature3=128., feature4=5.75)

    pprint(xtransform(scaling)(features))

def test3e(model:Predictor):

    unknown = dict(year=2023, lat=52, lng=0, total\_floor\_area=120, unknown=1000)

    missing = dict(lat=52, lng=0, total\_floor\_area=120)

    try:

        model.predict(unknown)

    except Exception as e:

        print(e)

    try:

        model.predict(missing)

    except Exception as e:

        print(e)

**Test script console output:**



|  |  |  |
| --- | --- | --- |
| Test | Details | Status |
| 3a | test3a() Output:  Encoding file loaded successfully...  Scaling file loaded successfully...  Residual scores loaded successfully...  Model built successfully... | *Passed* |
| 3b | Tested on **PR1 4HD** postcode  test3b() output:  {'imd': 1323,  'laua\_name': 'Preston',  'location': {'lat': 53.757096, 'lng': -2.688869},  'oac11': {'code': '4A2',  'group': 'Rented family living',  'subgroup': 'Private renting new arrivals',  'supergroup': 'Multicultural metropolitans'},  'pcd': 'PR1 4HD',  'ward\_name': 'City Centre'} | *Passed* |

**Encode mapping:**

{

  "property\_type": {

    "Terrace bungalow": 0,

    "Terrace house": 1,

    "Flat": 2,

    "Semi-detached bungalow": 3,

    "Semi-detached house": 4,

    "Maisonette": 5,

    "Detached bungalow": 6,

    "Detached house": 7

  }

}

|  |  |  |  |
| --- | --- | --- | --- |
| Test | Input | Expected output | Actual output |
| 3ci | {..., "property\_type":"Semi-detached house",} | {..., "property\_type":4,} | {..., "property\_type":4,} |
| 3cii | {..., "property\_type":"Unexpected",} | Invalid value error | 400 Bad Request: Invalid value for feature: property\_type = Unexpected |
| 3d | Scaling factors:  {  "a": { "mean": 10, "std": 2 },  "b": { "mean": 3, "std": 0.5 },  "c": { "mean": 100, "std": 32 },  "d": { "mean": 4.5, "std": 2 },  } Input  { "a":20, "b":4.5, "c":128, "d": 5.75 } | a=5  b=3  c=0.875  d=0.625 | array([[  15.,  -1.5,  124.875,  3.5  ]])  Actual ≠ Expected |
| 3Ei | {..., unknown=1000, ...} | Unknown fields error | 400 Bad Request: Unknown fields: ['unknown'] |
| 3EII | {  "lat":52,  "lng":0,  "total\_floor\_area":120  } | Missing fields error | 400 Bad Request: Missing fields: ['number\_habitable\_rooms', 'property\_type', 'year'] |
| 3EIII | { "year":"invalid", ... } | Invalid datatype error for year field | 400 Bad Request: Invalid datatype 'year' |

Test 3d fix:

# base/stage-2-webapp/model/\_\_init\_\_.py

...

def xtransform(scaling):

    def func(inp):

        inp\_ = inp.copy()

        for k, v in inp\_.items():

            feat = scaling.get(k, {"mean": 0., "std": 1.})

            # rescaling features to normalised values

            inp\_[k] = v - feat["mean"] / feat["std"]

            inp\_[k] = (v - feat["mean"]) / feat["std"]

        return array([list(inp\_.values())])

    return func

...

|  |  |  |  |
| --- | --- | --- | --- |
| Retest | Input | Expected output | Actual output |
| 3d | Scaling factors:  {  "a": { "mean": 10, "std": 2 },  "b": { "mean": 3, "std": 0.5 },  "c": { "mean": 100, "std": 32 },  "d": { "mean": 4.5, "std": 2 },  } Input  { "a":20, "b":4.5, "c":128, "d": 5.75 } | a=5  b=3  c=0.875  d=0.625 | array([[  5.,  3.,  0.875,  0.625  ]])  Actual = Expected |

**Test 3F: (testing on local deployment)**

|  |  |
| --- | --- |
| 3F – Prediction parameters | |
| postcode | PR1 4HD |
| year | 2023 |
| property\_type | Semi-detached house |
| Total\_floor\_area | 88 |
| number\_habitable\_rooms | 5 |

|  |  |  |
| --- | --- | --- |
| 3F - alpha | Output | Status |
| None | '>=' not supported between instances of 'str' and 'int' | *Failed* |
| 0 | '>=' not supported between instances of 'str' and 'int' | *Failed* |

Fix:

# base/stage2-webapp/main

def predict():

    args = request.args.to\_dict()

    redirect\_ = args.pop("redirect", "false").lower()

    alpha = args.pop("alpha", 0.45)

    postcode = args.pop("postcode")

try:

        alpha = float(alpha)

    except ValueError:

        alpha = None

...

|  |  |  |
| --- | --- | --- |
| 3F Re-test – alpha | Output | Status |
| None | Object of type ndarray is not JSON serializable | *Failed* |
| 0 | (Expect same output as alpha=None)  'alpha' must be in interval (0, 1) or None | *Failed* |
| 0.05 | {  "high": 343387.2082634162,  "low": 85797.00181245149,  "point": 214592.10503793383  } | *Passed* |
| 0.2 | {  "high": 275225.796295883,  "low": 153958.41377998466,  "point": 214592.10503793383  } | *Passed* |

Fixing alpha=None:

# base/stage2-webapp/model/predictor.py - Predictor class, predict function

...

inp\_ = self.xtransform(inp\_)  # normalising input

        # interval prediction

        if alpha is not None:

            yhat, interval = [

                self.ytransform(y).reshape(-1)

                for y in self.model.predict(inp\_, alpha)

            ]

            return {"low": interval[0], "point": yhat[0], "high": interval[1]}

        # point prediction

yhat = self.ytransform(self.model.predict(inp\_))

        yhat = self.ytransform(self.model.predict(inp\_)[0,0])

        return {"point": yhat}

...

Fixing alpha=0:

# base/stage2-webapp/model/algorithms/conformal.py - ConformalRegression class

...

    def predict(self, X, alpha = None):

        yhat = self.model.predict(X)

if alpha is None:

        if alpha is None or alpha==0:

            return yhat

...

# base/stage2-webapp/model/predictor.py - Predictor class, predict function

...

        # interval prediction

        if alpha is not None:

if alpha not in [None, 0]:

...

|  |  |  |
| --- | --- | --- |
| 3F Re-test – alpha | Output | Status |
| None | { "point": 214592.10503793383 } | *Passed* |
| 0 | (Expect same output as alpha=None)  { "point": 214592.10503793383 } | *Passed* |

**Test 3G** batch script for performance testing backend API (100 iterations):

:: base/stage-2-webapp/test.bat

@echo off

set n=100

set sum=0

set min=999999

set max=50

:: iterating through subroutine n times

for /l %%i in (1,1,%n%) do call :subroutine

:: computing mean

set /a mean=%sum%/%n%

:: displaying min, mean and max in milliseconds

echo Min: %min:~0,-3%ms

echo Mean: %mean:~0,-3%ms

echo Max: %max:~0,-3%ms

pause

GOTO :eof

:subroutine

:: extracting output of curl command to variable

for /f "tokens=\*" %%j in (

  'curl %host% -G^

   --data-urlencode "postcode=PR1 4HD"^

   --data-urlencode "year=2023"^

   --data-urlencode "property\_type=Semi-detached house"^

   --data-urlencode "total\_floor\_area=88"^

   --data-urlencode "number\_habitable\_rooms=5"^

   --data-urlencode "alpha=0.05"^

   --data-urlencode "redirect=false"^

   -o NUL -w "%%{time\_total}" -s'

) do set x=%%j

:: removing decimal point for integer arithmetic operations.

set x=%x:.=%

:: removing leading zero if x < 1 second

if %x:~0,1% EQU 0 set x=%x:~1%

if %x% LSS %min% set min=%x%

if %x% GTR %max% set max=%x%

set /a sum=%sum%+%x%

GOTO :eof

|  |  |  |
| --- | --- | --- |
| test | Details | Status |
| 3g | (tested on local deployment)  set host=http://127.0.0.1:5000/api/predict  Test script output: | *Passed* |
| 3G | (tested on live deployment)  set host=http://ml-house-prices.appspot.com/api/predict  Test script output: | *Passed* |

|  |  |  |
| --- | --- | --- |
| test | Details | Status |
| 4a | Initial design:    Final UI: | *Passed* |
| 4b | Google Maps output for PR1 4HD. | *Passed* |
| 4c | Unfilled fields:    Invalid year:    Invalid postcode:    Unexpected backend error: | *Passed* |

**Test 4D**

|  |  |  |
| --- | --- | --- |
| Viewport | Screenshot | Status |
| iPhone 12  1170x2532 |  | *Passed* |
| Galaxy Note 10+  1440x3040 |  | *Passed* |
| iPad Pro  2048x2732 |  | *Passed* |
| Desktop  1024x768 |  | *Passed* |

|  |  |  |
| --- | --- | --- |
| test | Details | Status |
| 4E | (tested on live deployment using PageSpeed Insights)  Desktop:    Mobile: | *Passed* |

**Test 5 – using 3 random postcodes/addresses:**

|  |  |
| --- | --- |
| Zoopla input | Prediction for year 2023 |
| 32 Ringwood drive, SO52 9GY | Point estimation = £378,000 |
| 4 hILDESLEY court, RG20 7LA | Point estimation = £291,000 |
| 90 trimmingham lane, hx2 7pt | Point estimation = £577,000 |

Property information provided by *propertychecker.co.uk*.

|  |  |  |
| --- | --- | --- |
| API input | Prediction for year 2023 | Status |
| POSTCODE=SO52 9GY  year=2023  (according to PropertyChecker)  PROPERTY\_TYPE=Semi-detached house  TOTAL\_FLOOR\_aREA=76.0  NUMBER\_HABITABLE\_ROOMS=5 | Undervalued compared to Zoopla within reasonable range (£60k difference) | *Passed* |
| POSTCODE=RG20 7LA  year=2023  (according to PropertyChecker)  PROPERTY\_TYPE=Maisonette  TOTAL\_FLOOR\_aREA=95.0  NUMBER\_HABITABLE\_ROOMS=4 | Same estimate as Zoopla (only £9k difference) | *Passed* |
| POSTCODE=hx2 7pt  year=2023  (according to PropertyChecker)  PROPERTY\_TYPE=Detached House  TOTAL\_FLOOR\_aREA=175.0  NUMBER\_HABITABLE\_ROOMS=6 | Undervalued compared to Zoopla (£70k difference), however point prediction (£505,400) and low prediction (£474k) are consistent with the property’s last sold price (£480,000 in December 2019) | *Passed* |

# Evaluation

## Objectives reflection and future improvements

1. Provide the user with reliable and accurate predictions:
   1. Building a comprehensive up-to-date dataset with easily interpretable features:
      1. Collect and combine raw data from multiple data sources:
         1. with over 5 million rows of raw data
         2. and 10+ columns of unique features
      2. Exploratory data analysis and data visualisation in the data research stage
      3. Compile a final training dataset with:
         1. with over a million rows of unique data points
         2. And 5+ distinct features

I believe I met these objectives quite well; I managed to take multiple data sources and compile them together in an effective and meaningful way. My analysis of the data was quite informative, and I think I did a good job visualising and commenting on my interpretation/understanding of different aspects of the data. I exceeded my expectations on the data volume and produced a final dataset of over 5 million rows. In hindsight, while I did achieve this objective quite well, if I did this again, I would replace the Year feature with the House Price Index feature, or I would replace the price values that were used to train the models with the adjusted price instead. This would allow me to ask the user for both the month and the year for their valuation and, in the backend, would scrape the web to extract the HPI for the user’s given date and scale the results accordingly. While this would eliminate the ability to extrapolate future predictions for the house valuation, I believe this would make past or current valuations much more reliable and accurate. Additionally, in future I would implement a data pipeline at the end of the EDA section that could build the dataset automatically instead of the more exploratory/manual approach that I used. This would be helpful as it could open up the use of more advanced techniques such as CRON scheduling in my web service to automatically build the dataset and train the model, keeping the data and prediction models up to date.

* 1. Build, train, and test predictive models:
     1. Linear regression
     2. Multilayer perceptron (neural network)
     3. K-Nearest neighbours
     4. Regression decision tree
     5. Random forest
     6. Gradient boosting machine

I implemented, tested, and commented on all the above predictive models. I found it difficult to grasp the intuition of the technical and mathematical concepts behind many of the models above, especially the neural network model, however, I think I did well in taking my primitive understanding and applying it to a functional implementation. Many of my models were sub-optimal and, while they did yield good results, were quite slow in training and I could have better optimisation techniques on my implementations to speed them up. One notable case was the Regression decision tree, and in the end, I had to resort to using a small sample from the large dataset to train it. Furthermore, this forced me to use pre-existing implementations from libraries for the Random Forest ensemble and Gradient boosting machine ensemble, as training multiple Decision trees would’ve taken far too long to be feasible. Additionally, within the k-Nearest neighbours implementation, I was forced to use a library ball-tree that would segment the data, to make finding the closest neighbours faster. Implementing the ball-tree on my own would have resulted in the same issues I faced with the regression tree. In hindsight, I would perhaps use Cython or CPython which would give C-like performance with code that is written mostly in Python syntax to optimise and significantly improve the performance of my code.

* 1. Build a predictive model that:
     1. is moderately small in storage size < 2MB.
     2. yields accurate predictions, with R2 scores > 0.8 (i.e., model is fitted to account for 80% of the dataset’s variability)

Following model selection, I chose to use the multilayer perceptron which had an r2 score of 0.84 and took up only 72kB of storage. This was by far the most challenging implementation, and I was quite happy with the modular approach I took. Using a layer-based network allowed me to dynamically change the neural net structure and using early stopping techniques allowed for efficient hyperparameter optimisation to maximise the model’s performance. It was also very easy to test individual parts of the neural network and make fixes where necessary instead of analysing the entire model and attempting to figure out where bugs lie within the whole structure. Furthermore, the modular design makes the model easily extensible, and, in future, I could add new layers with their own functionality without modifying any existing code.

* 1. Provide a prediction interval, for which the true value is probable to lie within, using conformal regression techniques.

I believe the conformal prediction service I implemented was effective in providing the user with an interval prediction, providing a broader prediction instead of a point prediction. However, the interval length is constant for the given alpha value and does not make any adjustments according to user inputs. Furthermore, the interval length is quite large for higher confidence intervals, and it required me to use smaller confidence levels to provide more appropriate predictions. In future, I would implement a newer technique called “conformalized quantile regression” that is fully adaptive and tend to have smaller interval lengths. This technique would be far more complex to implement, and thus more computationally expensive, than the naïve approach I took, however it would achieve better, much more reliable results.

1. Serve predictions to the user in a web app:
   1. Geocode postcodes to corresponding coordinates, utilising the *findthatpostcode.uk API*.
   2. Use *Google Maps API* to visualise the area with the given postcode.
   3. Provide useful statistics about the user-input postcode:
      1. Useful census data
      2. Index of Deprivation data
   4. Fast performance (by limiting dependencies and complexity)
   5. Display outputs and errors with maximum readability and ease:
      1. Responsive design
      2. Aesthetically consistent
      3. Subtle micro-animations

I am quite happy with the way the front-end site turned out; the site performs well, and I achieved an aesthetically consistent and pleasing user interface with dynamic animations and a design that fits on any device. In future, I would utilise breakpoints and media queries to make the design even more responsive – dynamically changing the layout and size of elements according to different screen sizes. Additionally, I would implement a dark mode which can be toggled on and off, but automatically defaulted to the user’s system or browser settings.

1. Deploy the web app across the internet:
   1. Use *GitHub* for source control.
   2. Use *Google App Engine* to host the site.

I used GitHub to allow a more rapid and versatile workflow, enabling me to make changes from across different devices in different locations. Additionally, I used Google App Engine to host and deploy my webapp, allowing users to access the site over the Internet. However, I could have better utilized the services by linking them together; automatically pushing updates to the hosted site whenever the repository receives a new commit. This would introduce more complex concepts such as continuous integration and continuous deployment. Additionally, I would use other source control concepts such as branches where I make changes and commits on the development branch and merge to the main branch when the code was ready. This main branch would then be deployed into Google App Engine. This would avoid mistakes in the code making breaking changes on the live deployment which was an issue I faced several times.

## End User Feedback

1. **How do you rate the user experience and interface of the app?**
   1. I appreciated how simple and straightforward everything was laid out, while also having a nice sleek design. The little animations and effects really made the site that much more enjoyable to use plus, the transition into the modal screen was smooth and seamless. I found the app was quite responsive and gave me results quickly; however, initially the form did take a while to load.
   2. I think the site was too simple, with only 2 different interfaces. I think introducing new features and different tools within the website would be much better for the user experience and interface. Despite the simplicity, I quite liked some of the effects and I enjoyed the modal animation. Also, the app loaded quite fast, and the predictions were served rapidly seamlessly with minimal delay.
2. **How useful do you think the results the app provided were?**
   1. The local area statistics were quite helpful as it gave me a brief rundown on the area and the map displayed was quite nice to see. The predictions themselves were useful as it wasn’t an exact prediction, rather the high and low predictions gave me a rough idea of how much my property was worth before I booked an on-site property valuation by a local agent.
   2. In my opinion, the interval prediction was too broad and gave slightly too rough of a valuation. I think if you included an option to adjust the confidence level of the prediction, it would’ve made the results more useful and a better indication to the value of the property. Despite this, I did find that when I compared the valuation with similar services, the point predictions were consistent and within the same range as the others. Additionally, the area visualisation and local area statistics were helpful, and I think including even more information would make the results that much more useful.
3. **What features would you like added to the app?**
   1. It would be helpful to see more statistics about the area, such as crime rate, local amenities etc. Additionally, a property search feature like Zoopla would be a valuable feature to implement as well. Being able to search an address with a postcode and I would find it extremely useful to access information about different properties, such as previous sold price, property type, and potentially street view or images of properties themselves. This would be very handy alongside the valuation tool.
   2. A useful feature would be a confidence level option to adjust how broad of a prediction I want to receive from the interval prediction model. Another feature I would find helpful would be an expandable map view which would include more information and perhaps highlight and display local hotspots like supermarkets and shops on the map. I would also like to see the distance and time to the nearest train stations and bus stops.

I believe the end users were satisfied with the outcome of the project. They were both happy with the results it provided and liked that the predictions were reliable and consistent and could serve as a rough idea before professional consultation. Furthermore, both users enjoyed the user experience, and found the app to be performant and responsive.

I agree with the response that the intervals were too broad; I think if I had used the “conformalized quantile regression” technique (a technique producing smaller intervals with better coverage, thus more reliable) it would’ve made sense to include the feature that the response suggested and implement an option for the confidence level of the prediction interval.

The search feature would be a good addition to the app’s functionality, and it could introduce some new complexity such as having a database and using search algorithms to find properties by locality, floor area etc. Additionally, the expandable map view with highlighted areas would also be a great feature to add and would introduce even more complexity with the use shortest-path algorithms to find routes to local hotspots/amenities.

## Conclusion

Overall, I met all my objectives, and my final project was successful. I learnt many new concepts especially in artificial intelligence and statistical analysis. The project was particularly challenging in:

1. Researching and understanding the intuition, theory, and implementation of the predictive algorithms. The most difficult to understand was the multilayer perceptron model and forced me to learn completely new topics like matrices and linear algebra.
2. Using the right techniques and statistical understanding to transform and build the dataset, especially with little understanding of statistics. Additionally, applying my primitive knowledge and using the dataset in a constructive way when modelling the data proved extremely difficult, and I went over several different datasets and models, some of which I completely skipped over, before I even got anything that would remotely work.
3. Developing the front-end webapp – making small tweaks and adjustments so the user interface looked and felt exactly right. One crucial factor was performance, and it was difficult to get right using the admittedly primitive understanding of web design that I currently have. Additionally, I had to optimise and speed up algorithms at the cost of some accuracy, however, significantly improving what would be an excruciating slow and painful user experience.
4. Deployment – choosing the right services and tweaking the deployment so that it could not only host the site but do so inexpensively was difficult, since most services have usage thresholds which could potentially incur charges. Therefore, I had to ensure any algorithms being used were not too computationally expensive and would not use too much CPU or memory.

In future, if I were to develop a similar project, I would make significant changes to the architecture of the project. In the research stage/model backend, I would adopt a data/model training pipeline to automatically build the datasets and models, allowing for models to remain up to date by scheduled tasks in the deployment stage. Furthermore, in the front-end app, I would adopt the MVC (model view controller) architecture, making the front-end more modular and extensible. This would allow for rapid development of new features without making many changes to the existing codebase, thus significantly reducing errors, bugs and would result in cleaner, more readable code. Aside from more general architecture changes, I would attempt to implement better, more complex algorithms, such as “self-organising feature maps” for automatic dimensionality reduction or “conformalized quantile regression” for better interval predictions, etc. Furthermore, I would implement more side features alongside the main valuation feature, suggested by responses in the end-user feedback, including property searching and area visualisation with distance/routes to local amenities. I would also include more local area statistics using different APIs, for example using the police.uk API to extract and display data about local crime rates. I would make these additions using the modular approach laid down by the MVC architecture and I would keep this approach in mind within the UI as well, with card and modal layouts separated into clearly defined sections.

To conclude, I successfully built and deployed an end-to-end artificial intelligence web-app to calculate property valuations given some property features. The project included many different aspects of software development and data science including exploratory data analysis, machine learning, web scraping, backend API development, frontend web development, and app deployment. I even used bits of parallelisation and multiprocessing. Moreover, I successfully deployed a live version of the app on the Internet, which is accessible at this link <https://ml-house-prices.appspot.com/>.